Intel® 64 and IA-32 Architectures Optimization Reference Manual

Documentation Changes

May 2023

Document Number: 355308-001

Contents

Revision History

Revision	Description	Date
-001	Initial release	May 2023

Preface

This document is an update to the optimization recommendations contained in the Intel® 64 and IA-32 Architectures Optimization Reference Manual, also known as the Software Optimization Manual. This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Nomenclature

Documentation Changes include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.

Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 Architecture software optimization topics covered by this reference manual.

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 1
2	Updates to Chapter 2
3	Updates to Chapter 3
4	Updates to Chapter 7
5	Updates to Chapter 10
6	Updates to Chapter 11
7	Updates to Chapter 15
8	Updates to Chapter 18
9	Updates to Chapter 20

Documentation Changes

Changes to the Intel® 64 and IA-32 Architectures Optimization Reference Manual volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

1. Updates to Chapter 1

Change bars and **violet** text show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Introduction.

Changes to this chapter:

- Section 1.2:
  - — References to Intel® Xeon® Scalable Processor Family were updated.
  - — 13th generation Intel® Core™ processor section was added.
  - — Updated trademarking as necessary.

CHAPTER 1 INTRODUCTION

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual* describes how to optimize software to take advantage of the performance characteristics of IA-32 and Intel 64 architecture processors.

The target audience for this manual includes software programmers and compiler writers. This manual assumes that the reader is familiar with the basics of the IA-32 architecture and has access to the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. A detailed understanding of Intel 64 and IA-32 processors is often required. In many cases, knowledge of the underlying microarchitectures is required.

The design guidelines discussed in this manual for developing high-performance software generally apply to current and future IA-32 and Intel 64 processors. In most cases, coding rules apply to software running in 64-bit mode of Intel 64 architecture, compatibility mode of Intel 64 architecture, and IA-32 modes (IA-32 modes are supported in IA-32 and Intel 64 architectures). Coding rules specific to 64-bit modes are noted separately.

NOTE

A public repository is available with open source code samples from select chapters of this manual. These code samples are released under a 0-Clause BSD license. Intel provides additional code samples and updates to the repository as the samples are created and verified.

Public repository: https://github.com/intel/optimization-manual.

Link to license: https://github.com/intel/optimization-manual/blob/master/COPYING.

1.1 TUNING YOUR APPLICATION

Tuning an application for high performance on any Intel 64 or IA-32 processor requires understanding and basic skills in:

- Intel 64 and IA-32 architecture.
- C and Assembly language.
- Hot-spot regions in the application that impact performance.
- Optimization capabilities of the compiler.
- Techniques used to evaluate application performance.

The Intel® VTune™ Performance Analyzer can help you analyze and locate hot-spot regions in your applications. On the Intel® Core™ i7, Intel® Core™2 Duo, Intel® Core™ Duo, Intel® Core™ Solo, Pentium® 4, Intel® Xeon®, and Intel® Pentium® M processors, this tool can monitor an application through a selection of performance monitoring events and analyze the performance event data that is gathered during code execution.

This manual also describes data that can be gathered using the performance counters through the processor's performance monitoring events.

1.2 ABOUT THIS MANUAL

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families, and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel Atom® processor Z8000 series is based on the Airmont microarchitecture.

The Intel Atom® processor Z3400 series and the Intel Atom® processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Scalable processor family, Intel® Xeon® processor E3-1500m v5 product family, and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel Atom® processor C series, the Intel Atom® processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Scalable processor family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some processors in the 3rd generation Intel® Xeon® Scalable processor family are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

The 12th generation Intel® Core™ processors are based on the Alder Lake performance hybrid architecture and support Intel 64 architecture.

The 13th generation Intel® Core™ processors are based on the Raptor Lake performance hybrid architecture and support Intel 64 architecture.

The 4th generation Intel® Xeon® Scalable processor family is based on the Sapphire Rapids microarchitecture and supports Intel 64 architecture.

The chapters in this manual are summarized as follows:

- **Chapter 1: Introduction** — Defines the purpose and outlines the contents of this manual.
- **Chapter 2: Intel® 64 and IA-32 Processor Architectures** — Describes the microarchitecture of recent Intel 64 and IA-32 processor families, and other features relevant to software optimization.
- **Chapter 3: General Optimization Guidelines** — Describes general code development and optimization techniques that apply to all applications designed to take advantage of the common features of current Intel processors.
- **Chapter 4: Intel Atom® Processor Architecture** — Describes the microarchitecture of recent Intel Atom processor families, and other features relevant to software optimization.
- **Chapter 5: Coding for SIMD Architectures** — Describes techniques and concepts for using the SIMD integer and SIMD floating-point instructions provided by the MMX™ technology, Streaming SIMD Extensions, Streaming SIMD Extensions 2, Streaming SIMD Extensions 3, SSSE3, and SSE4.1.
- **Chapter 6: Optimizing for SIMD Integer Applications** — Provides optimization suggestions and common building blocks for applications that use the 128-bit SIMD integer instructions.
- **Chapter 7: Optimizing for SIMD Floating-point Applications** — Provides optimization suggestions and common building blocks for applications that use the single-precision and double-precision SIMD floating-point instructions.
- **Chapter 8: INT8 Deep Learning Inference** — Describes INT8 as a data type for Deep learning Inference on Intel technology. The document covers both AVX-512 implementations and implementations using the new Intel® DL Boost Instructions.
- **Chapter 9: Optimizing Cache Usage** — Describes how to use the PREFETCH instruction, cache control management instructions to optimize cache usage, and the deterministic cache parameters.
- **Chapter 10: Introducing Sub-NUMA Clustering** — Describes Sub-NUMA Clustering (SNC), a mode for improving average latency from last level cache (LLC) to local memory.
- **Chapter 11: Multicore and Hyper-Threading Technology** — Describes guidelines and techniques for optimizing multithreaded applications to achieve optimal performance scaling. Use these when targeting multicore processor, processors supporting Hyper-Threading Technology, or multiprocessor (MP) systems.
- **Chapter 12: Intel® Optane™ DC Persistent Memory** — Provides optimization suggestions for applications that use Intel® Optane™ DC Persistent Memory.
- **Chapter 13: 64-Bit Mode Coding Guidelines** — This chapter describes a set of additional coding guidelines for application software written to run in 64-bit mode.
- **Chapter 14: SSE4.2 and SIMD Programming for Text-Processing/Lexing/Parsing**— Describes SIMD techniques of using SSE4.2 along with other instruction extensions to improve text/string processing and lexing/parsing applications.
- **Chapter 15: Optimizations for Intel® AVX, FMA, and Intel® AVX2**— Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions, FMA, and Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- **Chapter 16: Intel Transactional Synchronization Extensions** — Tuning recommendations to use lock elision techniques with Intel Transactional Synchronization Extensions to optimize multi-threaded software with contended locks.
- **Chapter 17: Power Optimization for Mobile Usages** — This chapter provides background on power saving techniques in mobile processors and makes recommendations that developers can leverage to provide longer battery life.
- **Chapter 18: Software Optimization for Intel® AVX-512 Instructions**— Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions 512.

- **Chapter 19: Intel® Advanced Vector Extensions 512-FP16 Instruction Set for Intel® Xeon® Processors** — Describes the addition of the FP16 ISA for Intel AVX-512 to handle IEEE 754-2019 compliant half-precision floating-point operations.
- **Chapter 20: Intel® Advanced Matrix Extensions (Intel® AMX) —** Describes best practices to optimally code to the metal on Intel® Xeon® Processors based on Sapphire Rapids SP microarchitecture. It extends the public documentation on Optimizing DL code with DL Boost instructions.
- **Chapter 21: Cryptography & Finite Field Arithmetic Enhancements** — Describes the new instruction extensions designated for acceleration of cryptography flows and finite field arithmetic.
- **Chapter 22: Intel® QuickAssist Technology** — Describes software development guidelines for the Intel® QuickAssist Technology (Intel® QAT) API. This API supports both the Cryptographic and Data Compression services.
- **Chapter 23: Knights Landing Microarchitecture and Software Optimization** — Describes the microarchitecture of processor families based on the Knights Landing microarchitecture, and software optimization techniques targeting Intel processors based on the Knights Landing microarchitecture.
- **Appendix A: Application Performance Tools** — Introduces tools for analyzing and enhancing application performance without having to write assembly code.
- **Appendix B: Using Performance Monitoring Events** — Provides information on the Top-Down Analysis Method and information on how to use performance events specific to the Intel Xeon processor 5500 series, processors based on Sandy Bridge microarchitecture, and Intel Core Solo and Intel Core Duo processors.
- **Appendix C: Intel Architecture Optimization with Large Code Pages** — Provides information on how the performance of runtimes can be improved by using large code pages.
- **Appendix D: IA-32 Instruction Latency and Throughput** — Provides latency and throughput data for the IA-32 instructions. Instruction timing data specific to recent processor families are provided.
- **Appendix E: Earlier Generations of Intel® 64 and IA-32 Processor Architectures** — Describes the microarchitecture of earlier generations of Intel 64 and IA-32 processor families, and other features relevant to software optimization.
- **Appendix F: Earlier Generations of Intel Atom® Microarchitecture and Software Optimization** — Describes the microarchitecture of earlier generations of processor families based on Intel Atom microarchitecture, and software optimization techniques targeting Intel Atom microarchitecture.

1.3 RELATED INFORMATION

For more information on the Intel® architecture, techniques, and the processor architecture terminology, the following are of particular interest:

- Intel® 64 and IA-32 Architectures Software Developer's Manual.
- Developing Multi-threaded Applications: A Platform Consistent Approach.
- Get Started with Intel® Fortran Compiler Classic and Intel® Fortran Compiler.
- Intel® C++ Compiler Classic Developer Guide and Reference.
- Intel® Developer Catalog.
- Intel® oneAPI Data Analytics Library.

More relevant links include:

- AI & Machine Learning: Development tools and resources.
- Development Topics & Technologies.
- Intel® 64 Architecture Processor Topology Enumeration.
- Intel® Distribution of OpenVino™ Toolkit.

- Intel Processor support and information.
- Intel® Hyper-Threading Technology (Intel® HT Technology).
- Intel® Instruction Set Extensions Technology Support.
- Intel® Many Integrated Core Architecture.
- Intel® QuickAssist Technology (Intel® QAT).
- Intel® SSE4 Programming Reference.
- Intel® VTune™ Profiler User Guide.

2. Updates to Chapter 2

Change bars and **violet** text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Intel® 64 and IA-32 Processor Architectures.

Changes to this chapter:

- Corrected branding and style across chapter.
- Section 2.1:
  - — Updated title of section from Sapphire Rapids Architecture to Sapphire Rapids Microarchitecture.
  - — Refined technology features associated with the Sapphire Rapids microarchitecture.
  - — 2.1.1: Changed: ***Its*** *I/O* to ***the*** *I/O*....
- Section 2.3:
  - — Updated to include new performance recommendations.
  - — Updated Figure 2-1 and 2-3 to include *H in Port 1
  - — Update Tables 2-1 and 2-2 to include additional footnote regarding *H performance improvements.

CHAPTER 2
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter gives an overview of features relevant to software optimization for current generations of Intel® 64 and IA-32 processors[1]. These features are:

- Microarchitectures that enable executing instructions with high throughput at high clock speeds, a high-speed cache hierarchy, and high-speed system bus.
- Intel® Hyper-Threading Technology[2] (Intel® HT Technology) support.
- Intel 64 architecture on Intel 64 processors.
- Single Instruction Multiple Data (SIMD) instruction extensions: MMX™ technology, Streaming SIMD Extensions (Intel® SSE), Streaming SIMD Extensions 2 (Intel® SSE2), Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® SSE4.1, and Intel® SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND.
- Fused Multiply Add Extensions.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- ADX and RDSEED.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
- Intel® Thread Director.

2.1 SAPPHIRE RAPIDS MICROARCHITECTURE

Intel processors based on Sapphire Rapids microarchitecture use Golden Cove cores and support the following additional features:

- Intel® Advanced Matrix Extensions (Intel® AMX) (Chapter 20).
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) (Chapter 19).
- Intel® Data Streaming Accelerator (Intel® DSA)[3].
- Intel® In-Memory Analytics Accelerator (Intel® IAA)[4].
- Intel® Quick Assist Technology (Intel® QAT)(Chapter 22)

2.1.1 4th Generation Intel® Xeon® Scalable Family of Processors

Intel's fourth generation Xeon® Scalable Family of Processors changes from a single-die monolithic design to multi-die Tiles.

The server products are scalable from dual-socket to eight-socket configurations (Section 3.11).

The I/O is increased with PCI Express 5.0, DDR5 memory, and Compute Express Link 1.1.

1. For previous generations of Intel 64 and IA-32 processors, see Appendix E, "Earlier Generations of Intel® 64 and IA-32 Processor Architectures." Intel Atom® processors are covered in Chapter 4, "Intel Atom® Processor Architectures."
2. Intel HT Technology requires a computer system with an Intel processor supporting hyper-threading and an Intel HT Technology-enabled chipset, BIOS, and operating system. Performance varies depending on the hardware and software used.
3. Please see the intel® DSA Specification and Intel® DSA User Guide.
4. Please see the Intel® IAA Specification.

Packaging includes a multi-die chip with up to 4 tiles. Each tile is a 400mm2 SoC, providing both compute cores and I/O.

Each tile contains 15 Golden Cove cores (see Section 2.3). Its memory controller provides two channels of DDR5 with a maximum of eight channels across 4 tiles, and 28 PCIe 5.0 lanes for a maximum of 112 across 4 tiles.

2.2 ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE

The Alder Lake performance hybrid architecture combines two Intel architectures, bringing together the Golden Cove performant cores and the Gracemont efficient Atom cores onto a single SoC. For details on the Golden Cove microarchitecture, see Section 2.3, "Golden Cove Microarchitecture." For details on the Gracemont microarchitecture, see Section 4.1, "Gracemont Microarchitecture."

2.2.1 12th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

12th Generation Intel® Core™ processors supporting performance hybrid architecture consist of up to eight Performance cores (P-cores) and eight Efficient cores (E-cores). These processors also include a 3MB Last Level Cache (LLC) per IDI module, where a module is one P-core or four E-cores. It has symmetrical ISA and comes in variety of configurations.

P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency. P-cores on these processors can also have Intel Hyper-Threading Technology enabled. All cores can be active simultaneously when the operating system (OS) decides to schedule on all processors.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a performance hybrid architecture. In 12th Generation Intel Core processors supporting performance hybrid architecture, ISA is converged to a common baseline between the P-cores and E-cores. In order to maintain symmetric ISA, the E-cores do not support the following features: Intel AVX-512, Intel AVX-512 FP-16, and Intel® TSX. The E-cores do support Intel AVX2 and Intel AVX-VNNI.

2.2.2 Hybrid Scheduling

2.2.2.1 Intel® Thread Director

Intel® Thread Director continually monitors software in real-time giving hints to the operating system's scheduler allowing it to make more intelligent and data-driven decisions on thread scheduling. With Intel Thread Director, hardware provides runtime feedback to the OS per thread based on various IPC performance characteristics, in the form of:

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

Intel Thread Director is first introduced in desktop and mobile variants of the 12th generation Intel Core processor based on Alder Lake performance hybrid architecture.

A processor containing both P-cores and E-cores with different performance characteristics creates a challenge for the operating system's scheduler. Additionally, different software threads see different performance ratios between the P-cores and E-cores. For example, the performance ratio between the P-cores and E-cores for highly vectorized floating-point code is higher than the performance ratio for scalar integer code. So, when the operating system needs to make an optimal scheduling decision it needs to be aware of the characteristics of the software threads that are candidates for scheduling. If not enough P-cores are available and there is a mix of software threads with different characteristics, the operating system should schedule those threads that benefit most from the P-cores onto those cores and schedule the others on the E-cores.

Intel Thread Director provides the necessary hint to the operating system about the characteristics of the software thread executing on each of the logical processors. The hint is dynamic and reflects the recent characteristics of the thread, i.e., it may change over time based on the dynamic instruction mix of the thread. The processor also considers microarchitecture factors to define the dynamic software thread characteristics.

Thread specific hardware support is enumerated via the CPUID instruction and enabled by the operating system via writing to configuration MSRs. The Intel Thread Director implementation on processors based on Alder Lake performance hybrid architecture defines four thread classes:

0. Non-vectorized integer or floating-point code.
1. Integer or floating-point vectorized code, excluding Intel® Deep Learning Boost (Intel® DL Boost) code.
2. Intel DL Boost code.
3. Pause (spin-wait) dominated code.

The dynamic code does not have to be 100% of the class definition. It should be large enough to be considered belonging to that class. Also, dynamic microarchitectural metrics such as consumed memory bandwidth or cache bandwidth may move software threads between classes. Example pseudo-code sequences for the Intel Thread Director classes available on processors based on Alder Lake performance hybrid architecture are provided in the examples 2-1 through 2-4.

Intel Thread Director also provides a table in system memory, only accessible to the operating system, that defines the P-core vs. E-core performance ratio per class. This allows the operating system to pick and choose the right software thread for the right logical processor.

In addition to the performance ratio between P-cores and E-cores, Intel Thread Director provides the energy efficiency ratio between those cores. The operating system can then use this information when it prefers energy savings over maximum performance. For example, a background task such as indexing can be scheduled on the most energy efficient core since its performance is less critical.

Example 2-1. Class 0 Pseudo-code Snippet

```
while (1)
{
    asm("xor rax, rax;"
          "add rax, 5;"
          "inc rax;"
    );
}
```

Example 2-2. Class 1 Pseudo-code Snippet

```
while (1)
{
    asm("vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"

        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
    );
}
```

Example 2-3. Class 2 Pseudo-code Snippet

```
while (1)
{
    __asm(
        vpdpbusd ymm2, ymm0, ymm1
        vpdpbusd ymm3, ymm0, ymm1
        vpdpbusd ymm4, ymm0, ymm1
        vpdpbusd ymm5, ymm0, ymm1
        vpdpbusd ymm6, ymm0, ymm1
        vpdpbusd ymm7, ymm0, ymm1
        vpdpbusd ymm8, ymm0, ymm1
        vpdpbusd ymm9, ymm0, ymm1
        vpdpbusd ymm10, ymm0, ymm1
        vpdpbusd ymm11, ymm0, ymm1
        vpdpbusd ymm12, ymm0, ymm1
        vpdpbusd ymm13, ymm0, ymm1
    );
}
```

Example 2-4. Class 3 Pseudo-code Snippet

```
while (1)
{
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
    );
}
```

For more detailed information on this technology, refer to the Intel® 64 and IA-32 Architectures Software Developer's Manual.

2.2.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture

E-cores are designed to provide better performance than a logical P-core with both hardware sibling hyper-threads busy.

2.2.2.3 Scheduling with a Multi-E-Core Module

E-cores within an idle module help provide better performance than E-cores in a busy module.

2.2.2.4 Scheduling Background Threads on x86 Hybrid Architecture

In most scenarios, background threads can leverage scalability and multithread efficiency of E-cores.

2.2.3 Recommendations for Application Developers

The following are recommendations when using processors supporting performance hybrid architecture:

- Stay up to date on updates on operating systems and optimized libraries.
- Software needs to avoid setting hard affinities on either threads or processes in order to allow the operating system to provide the optimal core selection for Intel Hybrid.
- Software should replace active spin-waits with lightweight waits ideally using the new UMWAIT/TPAUSE and older PAUSE instructions which will allow for better hints to the scheduler on time spinning.
- Software can utilize the Windows Power Throttling information using process information and thread information APIs, to give hints to the scheduler on the Quality of Service (QoS) required for a particular thread or process to improve both performance and energy efficiency.
- Leverage Windows frameworks and media APIs for multimedia application development. Windows Media Foundation framework is optimized for hybrid architecture and enables media applications to run efficiently while preventing glitches.
- The Windows IrqPolicyMachineDefault policy enables Windows to optimally target interrupts to the right core, and more so on hybrid architecture.

For additional recommendations and information on performance hybrid architecture, refer to the white papers on the Performance Hybrid Architecture page.

2.3 GOLDEN COVE MICROARCHITECTURE

The Golden Cove microarchitecture is the successor of Ice Lake microarchitecture. The Golden Cove microarchitecture introduces the following enhancements:

- Wider machine: 5$\rightarrow$6 wide allocation, 10$\rightarrow$12 execution ports, and 4$\rightarrow$8 wide retirement.
- Significant increases in the size of key structures enable deeper OOO execution and expose more instruction level parallelism.
- Greater capabilities per execution port, e.g., 5th integer ALU execution ports with expanded capability and a new fast floating-point adder.
- Intel® Advanced Matrix Extensions (Intel® AMX)[1]: Built-in integrated Tiled Matrix Multiplication / Machine Learning Accelerator.
- Improved branch prediction.
- Improvements for large code footprint workloads, e.g., larger branch prediction structures, enhanced code prefetcher, and larger instruction TLB.
- Wider fetch: legacy decode pipeline fetch bandwidth increase to 32B/cycles, 4$\rightarrow$6 decoders, increased micro-op cache size, and increased micro-op cache bandwidth.
- Maximum load bandwidth increased from 2 loads/cycle to 3 loads/cycle.
- Larger 4K Pages DTLB, increase in the number of outstanding Page Miss handlers.
- Increased number of outstanding misses (16 FB, 32$\rightarrow$48 Deeper MLC miss queues).

1. Intel AMX are not available on client parts.

- Enhanced data prefetchers for increased memory parallelism.
- Mid-level cache size increased to 2MB on server parts; remains 1.25MB on client parts.

2.3.1 Golden Cove Microarchitecture Overview

The basic pipeline functionality of the Golden Cove microarchitecture is depicted in Figure 2-1.

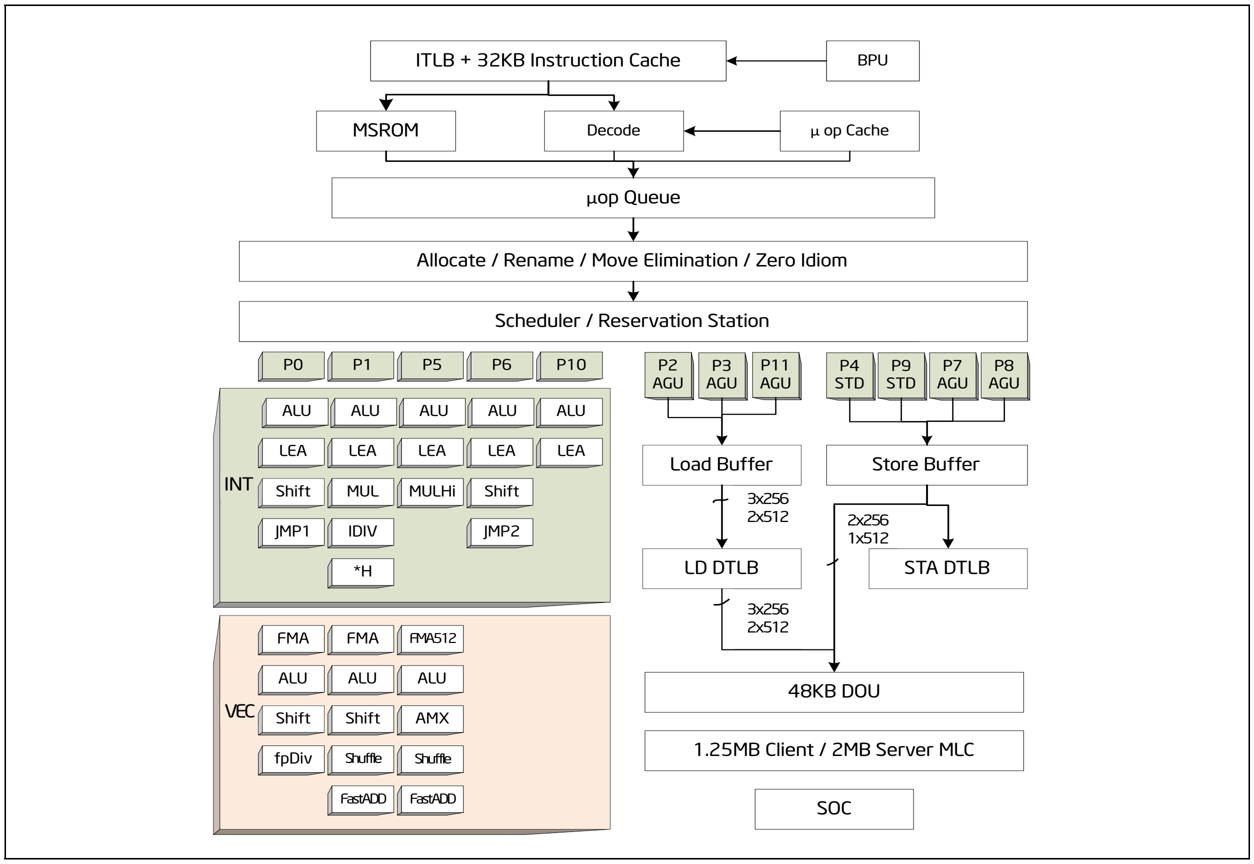


Figure 2-1. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture

The Golden Cove front end is depicted in Figure 2-2. The front end is built to feed the wider and deeper out-of-order core:

- Legacy decode pipeline fetch bandwidth increased from 16 to 32 bytes/cycle.
- The number of decoders increased from four to six, allowing decode of up to 6 instructions per cycle.
- The micro-op cache size increased, and its bandwidth increased to deliver up to 8 micro-ops per cycle.
- Improved branch prediction.

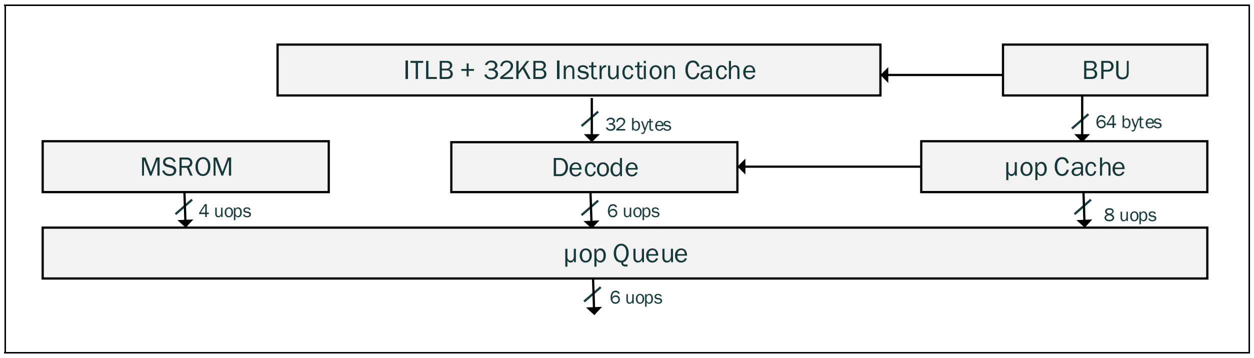


Figure 2-2. Processor Front End of the Golden Cove Microarchitecture

Improvements for large code footprint workloads:

- Double the size of the instruction TLB: 128→256 entries for 4K pages, 16→32 entries for 2M/4M pages.
- Bigger branch prediction structures.
- Enhanced code prefetcher.
- Improved LSD coverage.
- The IDQ can hold 144 uops per logical processor in single thread mode, or 72 uops per thread when SMT is active.

Additional improvements include:

- Significant increase in size of key buffer structures to enable deeper OOO execution and expose more instruction level parallelism.
- Wider machine:
  - — Wider allocation (5→6 uops per cycle) and retirement (4→8 uops per cycle) width.
  - — Increase in number of execution ports (10→12).
  - — Greater capabilities per execution port.

Table 2-1 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-1. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU[3] LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle*				FMA** Fast Adder Vec ALU Shuffle					

NOTES:

1. "*" in this table indicates that these features are not available for 512-bit vectors.
2. "**" in this table indicates that these features are not available for 512-bit vectors in Client parts.
3. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

Table 2-2 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel® SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-2. Golden Cove Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	5[2]	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2[3]	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	2x256-bit 1x512-bit	(v)add, (v)cmp. (v)max, (v)min, (v)sub, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2sl, (v)cvtss2sl
	3x256-bit 2x512-bit	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2x256-bit 1x512-bit	(v)psllv*, (v)psrlv*, vector shift count in imm8
VEC Add (in VEC FMA)	2x256-bit 1x512-bit	(v)add*, (v)cmp*, (v)max*, (v)min*, (v)sub*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si

Table 2-2. Golden Cove Microarchitecture Execution Units and Representative Instructions[1] (Contd.)

Execution Unit	# of Unit	Instructions
VEC Fast Add	2x256-bit 1x512-bit	(v)add*, (v)addsub*, (v)sub*
Shuffle	2x256-bit 1x512-bit	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw (new cross lane shuffle on both ports)
Vec Mul/FMA	2x256-bit (1 or 2)x512-bit	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.
2. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.
3. *ibid.*

Table 2-3 describes bypass delay in cycles between producer and consumer operations.

Table 2-3. Bypass Delay Between Producer and Consumer Micro-Ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	**SIMD/0,1/1**	**FMA/0,1/4**	**MUL/0,1/4**	**Fast Adder/1,5/3**	**SIMD/5/1,3**	**SHUF/ 1,5/1, 3**	**V2I/0/3**
SIMD/0,1/1	0	1	1	1	0	0	0
FMA/0,1/4	1	0	1	0	0	0	0
MUL/0,1/4	1	0	1	0	0	0	0
Fast Adder/0,1/3	1	0	1	-1	0	0	0
SIMD/5/1,3	0	1	1	1	0	0	0
SHUF/1,5/1,3	0	0	1	0	0	0	0
V2I/0/3	0	0	1	0	0	0	0
I2V/5/1	0	1	1	0	0	0	0

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to a 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.
- "V2I/0/3" applies to a 3-cycle vector-to-integer uop dispatched to port 0.

- “I2V/5/1” applies to a 1-cycle integer-to-vector uop dispatched to port 5.
- “Fast Adder/1,5/3” applies to either a 3-cycle 256-bit uop dispatched to either port 1 or port 5, or a 512-bit uop dispatched to port 5. This operation supports two cycles back-to-back between a pair of Fast Adder operations.

A new Fast Adder[1] unit is added as 512-bit on port 5 in VEC stack, and as 256-bit on ports 1 and 5. The Fast Adder performs floating-point ADD/SUB operations in 3 cycles.

Back-to-back ADD/SUB operations that are both executed on the Fast Adder unit perform the operations in two cycles.

- In 128/256-bit, back-to-back ADD/SUB operations executed on the Fast Adder unit perform the operations in two cycles.
- In 512-bit, back-to-back ADD/SUB operations are executed in two cycles if both operations use the Fast Adder unit on port 5.

The following instructions are executed by the Fast Adder unit:

- (V)ADDSUBSS/SD/PS/PD
- (V)ADDSS/SD/PS/PD
- (V)SUBSS/SD/PS/PD

2.3.1.1 Cache Subsystem and Memory Subsystem

The cache subsystem and memory subsystem changes in the Golden Cove microarchitecture are:

- Maximum load bandwidth increased from 2 to 3 loads per cycle. Bandwidth of Intel AVX-512 loads, Intel AMX loads, and MMX/x87 loads remain at a maximum of 2 loads per cycle.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Number of entries for 4K pages in the load DTLB increased from 64 to 96.
- Page Miss handler can handle up to four D-side page walks in parallel instead of two.
- Increased number of outstanding DCU and MLC misses.
- Enhanced data prefetchers for increased memory parallelism.
- Partial store forwarding allowing forwarding data from store to load also when only part of the load was covered by the store (in case the load's offset matches the store's offset).

2.3.1.2 Avoiding Destination False Dependency

Some SIMD instructions incur false dependency on the destination operand. The following instructions are affected:

- VFMULCSH, VFMULCPH
- VFCMULCSH, VFCMULCPH
- VPERMD, VPERMQ, VPERMPS, VPERMPD
- VRANGE[SS,PS,SD,PD]
- VGETMANTSH, VGETMANTSS, VGETMANTSD
- VGETMANTPS, VGETMANTPD (memory versions only)
- VPMULLQ

1. The Fast Adder unit is not available on 512-bit vectors in Client parts.

Recommendation: *Use dependency breaking zero idioms on the destination register before the affected instructions to avoid potential slowdown from the false dependency.*

Example 2-5. Breaking False Dependency through Zero Idiom

Code with False Dependency Impact	Mitigation: Break False Dependency with Zero Idiom
vaddps zmm3, zmm4, zmm5 vmovaps [rsi], zmm3 vfmulcph zmm3, zmm2, zmm1 ;False dependency on zmm3. Will not execute out-of-order until vaddps writes zmm3.	vaddps zmm3, zmm4, zmm5 vmovaps [rsi], zmm3 vpxord zmm3, zmm3, zmm3 ;Dependency-breaking zero idiom. vfmulcph zmm3, zmm2, zmm1 ;Execute out-of-order without waiting for vaddps result.

2.4 ICE LAKE CLIENT MICROARCHITECTURE

The Ice Lake client microarchitecture introduces the following new features that allow optimizations of applications for performance and power consumption:

- Targeted vector acceleration.
- Crypto acceleration.
- Intel® Software Guard Extensions (Intel® SGX) enhancements.
- Cache line writeback instruction (CLWB).

2.4.1 Ice Lake Client Microarchitecture Overview

The Ice Lake client microarchitecture builds on the successes of the Skylake client microarchitecture. The basic pipeline functionality of the Ice Lake Client microarchitecture is depicted in Figure 2-3.

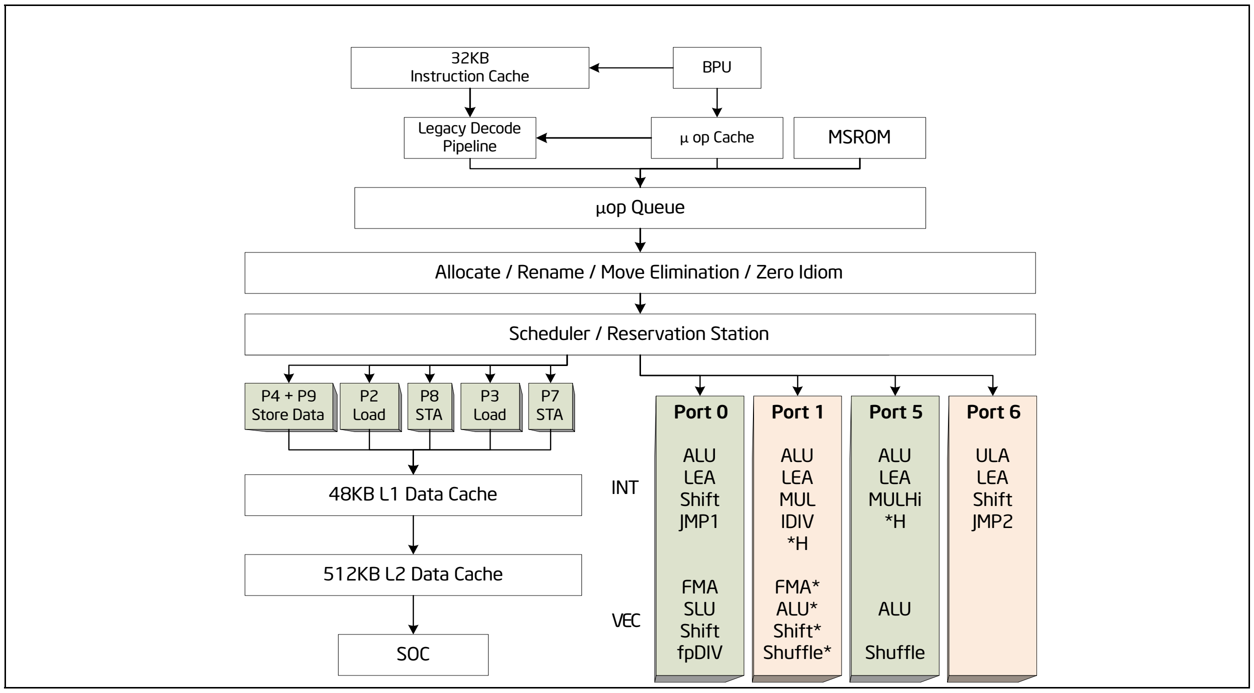


Figure 2-3. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

NOTES:

1. "*" in the figure above indicates these features are not available for 512-bit vectors.
2. "INT" represents GPR scalar instructions.
3. "VEC" represents floating-point and integer vector instructions.
4. "MULHi" produces the upper 64 bits of the result of an iMul operation that multiplies two 64-bit registers and places the result into two 64-bits registers.
5. The "Shuffle" on port 1 is new, and supports only in-lane shuffles that operate within the same 128-bit sub-vector.
6. The "IDIV" unit on port 1 is new, and performs integer divide operations at a reduced latency.
7. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

The Ice Lake client microarchitecture introduces the following new features:

- Significant increase in size of key structures enable deeper OOO execution.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Intel AVX-512 (new for client processors): 512-bit vector operations, 512-bit loads and stores to memory, and 32 new 512-bit registers.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA), reduced latency Integer Divider.
- 2×BW for AES-NI peak throughput for existing binaries (microarchitectural).
- Rep move string acceleration.
- 50% increase in size of the L1 data cache.
- Reduced effective load latency.
- 2×L1 store bandwidth: 1 → 2 stores per cycle.
- Enhanced data prefetchers for increased memory parallelism.
- Larger 2nd level TLB.
- Larger uop cache.
- Improved branch predictor.
- Large page ITLB size in single thread mode doubled.
- Larger L2 cache.

The Ice Lake client microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

2.4.1.1 The Front End

The front end changes in Ice Lake Client microarchitecture include:

- Improved branch predictor.
- Large page ITLB in single thread mode increased from 8 to 16 entries.
- Larger uop cache.
- The IDQ can hold 70 uops per logical processor vs. 64 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2×70 vs. 2×64 per core). If only one logical processor is active in the core, the IDQ can hold 70 uops vs. 64 uops.
- The LSD in the IDQ can detect loops of up to 70 uops per logical processor irrespective single thread or multi thread operation.

2.4.1.2 The Out of Order and Execution Engines

The Out of Order and execution engines changes in Ice Lake Client microarchitecture include:

- A significant increase in size of reorder buffer, load buffer, store buffer, and reservation stations enable deeper OOO execution and higher cache bandwidth.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA).
- Reduced latency Integer Divider.
- A new iDIV unit was added that significantly reduces the latency and improves the of throughput of integer divide operations.

Table 2-4 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-4. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Address	Store Data
FMA Vec ALU Vec Shift FP Div	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

NOTES:

1. "*" in this table indicates these features are not available for 512-bit vectors.

Table 2-5 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-5. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si

Table 2-5. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
Shuffle	2	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

Table 2-6 describes bypass delay in cycles between producer and consumer operations.

Table 2-6. Bypass Delay Between Producer and Consumer Micro-ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	**SIMD/0,1/1**	**FMA/0,1/4**	**VIMUL/0,1/4**	**SIMD/5/1,3**	**SHUF/5/1, 3**	**V2I/0/3**	**I2V/5/1**
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	0	0	1	0	0	0	NA
I2V/5/1	0	1	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.
- "V2I/0/3" applies to a 3-cycle vector-to-integer uop dispatched to port 0.
- "I2V/5/1" applies to a 1-cycle integer-to-vector uop to port 5.

2.4.1.3 Cache and Memory Subsystem

The cache hierarchy changes in Ice Lake Client microarchitecture include:

- 50% increase in size of the L1 data cache.
- 2×L1 store bandwidth: 3 → 4 AGUs, 1 → 2 store data.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Higher cache bandwidth compared to previous generations.
- Larger 2nd level TLB: 1.5K entries → 2K entries.
- Enhanced data prefetchers for increased memory parallelism.
- L2 cache size increased from 256KB to 512KB.
- L2 cache associativity increased from 4 ways to 8 ways.
- Significant reduction in effective load latency.

Table 2-7. Cache Parameters of the Ice Lake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Latency[1] (cycles)	Peak Bandwidth (bytes/cycles)	Sustained Bandwidth (bytes/cycles)	Update Policy
First Level (DCU)	48KB/8	64	5	2×64B loads + 1x64B or 2x32B stores	Same as peak	Writeback
Second Level (MLC)	512KB/8	64	13	64	48	Writeback
Third Level (LLC)	Up to 2MB per core/up to 16 ways	64	xx[2]	32	21	Writeback

NOTES:
1. Software-visible latency/bandwidth will vary depending on access patterns and other factors.
2. This number depends on core count.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, shared L2 TLB for 4K and 4MB pages and a dedicated L2 TLB for 1GB pages.

Table 2-8. TLB Parameters of the Ice Lake Client Microarchitecture

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Instruction	4KB	128	64	8
Instruction	2MB/4MB	16	8	8
First Level Data (loads)	4KB	64	64 competitively shared	4
First Level Data (loads)	2MB/4MB	32	32 competitively shared	4
First Level Data (loads)	1GB	8	8 competitively shared	8
First Level Data (stores)	Shared for all page sizes	16	16 competitively shared	16
Second Level	Shared for all page sizes	2048[1]	2048 competitively shared	16

NOTES:
1. 4K pages can use all 2048 entries. 2/4MB pages can use 1024 entries (in 8 ways), sharing them with 4K pages. 1GB pages can use the other 1024 entries (in 8 ways), also sharing them with 4K pages.

Paired Stores

Ice Lake Client microarchitecture includes two store pipelines in the core, with the following features:

- Two dedicated AGU for LDs on ports 2 and 3.
- Two dedicated AGU for STAs on ports 7 and 8.
- Two fully featured STA pipelines.
- Two 256-bit wide STD pipelines (AVX-512 store data takes two cycles to write).
- Second senior store pipeline to the DCU via store merging.

Ice Lake Client microarchitecture can write two senior stores to the cache in a single cycle if these two stores can be paired together. That is:

- The stores must be to the same cache line.
- Both stores are of the same memory type, WB or USWC.
- None of the stores cross cache line or page boundary.

In order to maximize performance from the second store port try to:

- Align store operations whenever possible.
- Place consecutive stores in the same cache line (not necessarily as adjacent instructions).

As seen in Example 2-6, it is important to take into consideration all stores, explicit or not.

Example 2-6. Considering Stores

Stores are Paired Across Loop Iterations	Stores Not Paired Due to Stack Update in Between
Loop:	Loop:
compute reg	call function to compute reg
...	...
store [X], reg	store [X], reg
add X, 4	add X, 4
jmp Loop ; stores from different iterations of the loop can be paired all together because they usually would be same line	jmp Loop ; stores from different iterations of the loop cannot be paired anymore because of the call store to stack
	; the call is disturbing pairing

In some cases it is possible to rearrange the code to achieve store pairing. Example 2-7 provides details.

Example 2-7. Rearranging Code to Achieve Store Pairing

Stores to Different Cache Lines - Not Paired	Unrolling May Solve the Problem

Loop:
    ... compute ymm1 ...
    vmovaps [x], ymm1
    ... compute ymm2 ...
    vmovaps [y], ymm2
    add x, 32
    add y, 32
    jmp Loop   ; this loop cannot pair any store because of alternating store to different cache lines [x] and [y]
	Loop:
	    ... compute ymm1 ...
	    vmovaps [x], ymm1
	    ... compute new ymm1 ...
	    vmovaps [x+32], ymm1
	    ... compute ymm2 ...
	    vmovaps [y], ymm2
	    ... compute new ymm2 ...
	    vmovaps [y+32], ymm2
	    add x, 64
	    add y, 64
	    jmp Loop ; the loop was unrolled 2 times and stores re-arranged to make sure two stores to the same cache line are placed one after another. Now stores to addresses [x] and [x+32] are to the same cache line and could be paired together and executed in same cycle

2.4.1.4 New Instructions

New instructions and architectural changes in Ice Lake Client microarchitecture are listed below. Actual support may be product dependent.

- Crypto acceleration
  - SHA NI for acceleration of SHA1 and SHA256 hash algorithms.
  - Big-Number Arithmetic (IFMA): VPMADD52 - two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance.
  - Galois Field New Instructions (GFNI) for acceleration of various encryption algorithms, error correction algorithms, and bit matrix multiplications.
  - Vector AES and Vector Carry-less Multiply (PCLMULQDQ) instructions to accelerate AES and AES-GCM.
- Security Technologies
  - Intel® SGX enhancements to improve usability and applicability: EDMM, multi-package server support, support for VMM memory oversubscription, performance, larger secure memory.
- Sub Page protection for better performance of security VMMs.
- Targeted Acceleration
  - Vector Bit Manipulation Instructions: VBMI1 (permutes, shifts) and VBMI2 (Expand, Compress, Shifts)- used for columnar database access, dictionary based decompression, discrete mathematics, and data-mining routines (bit permutation and bit-matrix-multiplication).
  - VNNI with support for integer 8 and 16 bits data types- CNN/ML/DL acceleration.
  - Bit Algebra (POPCNT, Bit Shuffle).
  - Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Platform analysis features for more efficient performance software tuning and debug.
  - AnyThread removal.

— 2x general counters (up to 8 per-thread).

— Fixed Counter 3 for issue slots.

— New performance metrics for built-in support for Level 1 Top-Down method (% of Issue slots that are front-end bound, back-end bound, bad speculation, retiring) while leaving the 8 general purpose counters free for software use.

2.4.1.5 Ice Lake Client Microarchitecture Power Management

Processors based on Ice Lake Client microarchitecture are the first client processors whose cores may execute at a different frequency from one another. The frequency is selected based on the specific instruction mix; the type, width and number of vector instructions of the program that executes on each core, the ratio between active time and idle time of each core, and other considerations such as how many cores share similar characteristics.

Most of the power management features of Skylake Server Microarchitecture (see Section 2.5) is applicable to Ice Lake Client microarchitecture as well. The main differences are the following:

- The typical P0n max frequency difference between Intel® Advanced Vector Extensions (Intel® AVX-512) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) on Ice Lake Client microarchitecture is much lower than on Skylake Server microarchitecture. Therefore, the negative impact on overall application performance is much smaller.
- All processors based on Ice Lake Client microarchitecture contain a single 512-bit FMA unit, whereas some of the processors based on Skylake Server microarchitecture contain two such units. Both processors contain two 256-bit FMA units. The power consumed by Ice Lake Client FMA units is the same, whereas on Skylake Server the 512-bit units consume twice as much.

Compute heavy workloads, especially those that span multiple Ice Lake client cores, execute at a lower frequency than P0n, both under Intel AVX-512 and under Intel AVX2 instruction sets, due to power limitations. In this scenario, Intel AVX-512 architecture, which requires less dynamic instructions to complete the same task than Intel AVX2 architecture, consumes less power and thus may achieve higher frequency. The net result may be higher performance due to the shorter path length and a bit higher frequency.

There are still some cases where coding to the Intel AVX-512 instruction set yields lower performance than when coding to the Intel AVX2 instruction set. Sometimes it is due to microarchitecture artifacts of longer vectors, in other cases the natural vectors are just not long enough. Most compilers are still maturing their Intel AVX-512 support, and it may take them a few more years to generate optimal code.

The general recommendation in the Skylake Server Power Management section (see Section 2.5.3) still holds. Developers should code to the Intel AVX-512 instruction set and compare the performance to their Intel AVX2 workload on Ice Lake Client microarchitecture, before making the decision to proceed with a complete port.

2.5 SKYLAKE SERVER MICROARCHITECTURE

The Intel® Xeon® Processor Scalable Family is based on the Skylake Server microarchitecture. Processors based on the Skylake microarchitecture can be identified using CPUID's DisplayFamily_DisplayModel signature, which can be found in Table 2-1 of CHAPTER 2 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

The Skylake Server microarchitecture introduces the following new features[1] that allow you to optimize your application for performance and power consumption.

- A new core based on the Skylake Server microarchitecture with process improvements based on the Kaby Lake microarchitecture.
- Intel AVX-512 support.
- More cores per socket (max 28 vs. max 22).

1. Some features may not be available on all products.

- 6 memory channels per socket in Skylake microarchitecture vs. 4 in the Broadwell microarchitecture.
- Bigger L2 cache, smaller non inclusive L3 cache.
- Intel® Optane™ support.
- Intel® Omni-Path Architecture (Intel® OPA).
- Sub-NUMA Clustering (SNC) support.

The green stars in Figure 2-4 represent new features in Skylake Server microarchitecture compared to Skylake microarchitecture for client; a 1MB L2 cache and an additional Intel AVX-512 FMA unit on port 5 which is available on some parts.

Since port 0 and port 1 are 256-bits wide, Intel AVX-512 operations that will be dispatched to port 0 will execute on both port 0 and port 1; however, other operations such as *lea* can still execute on port 1 in parallel. See the red block in Figure 2-8 for the fusion of ports 0 and 1.

Notice that, unlike Skylake microarchitecture for client, the Skylake Server microarchitecture has its front end loop stream detector (LSD) disabled.

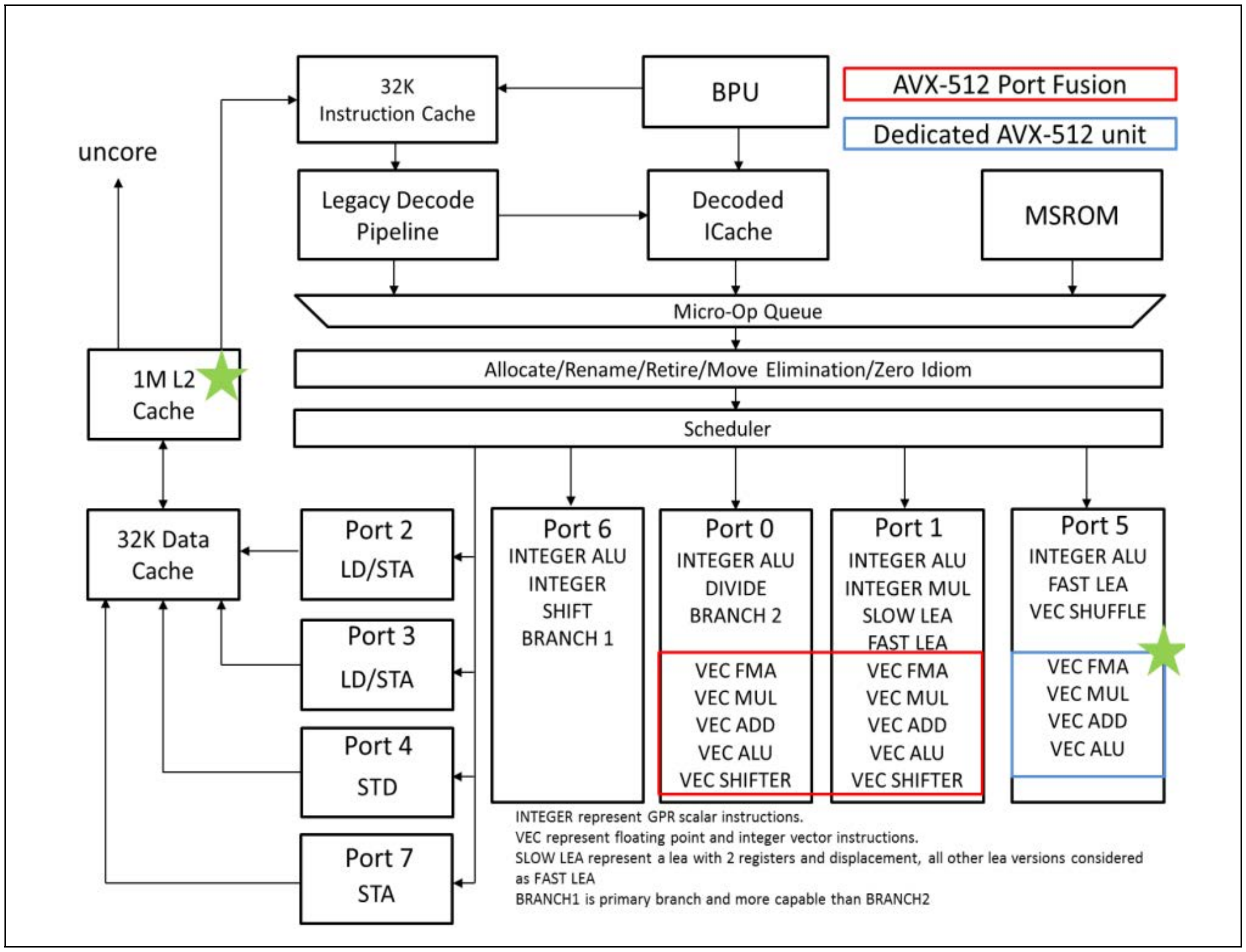

32K Instruction Cache
BPU
AVX-512 Port Fusion
Dedicated AVX-512 unit
uncore
Legacy Decode Pipeline
Decoded ICache
MSROM
Micro-Op Queue
1M L2 Cache
Allocate/Rename/Retire/Move Elimination/Zero Idiom
Scheduler
32K Data Cache
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Port 6 INTEGER ALU INTEGER SHIFT BRANCH 1
Port 0 INTEGER ALU DIVIDE BRANCH 2 VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 1 INTEGER ALU INTEGER MUL SLOW LEA FAST LEA VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 5 INTEGER ALU FAST LEA VEC SHUFFLE VEC FMA VEC MUL VEC ADD VEC ALU
INTEGER represent GPR scalar instructions.
VEC represent floating point and integer vector instructions.
SLOW LEA represent a lea with 2 registers and displacement, all other lea versions considered as FAST LEA
BRANCH1 is primary branch and more capable than BRANCH2

Figure 2-4. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture

2.5.1 Skylake Server Microarchitecture Cache

The Intel Xeon Processor Scalable Family based on Skylake Server microarchitecture has significant changes in core and uncore architecture to improve performance and scalability of several components compared with the previous generation of the Intel Xeon processor family based on Broadwell microarchitecture.

2.5.1.1 Larger Mid-Level Cache

Skylake Server microarchitecture implements a mid-level (L2) cache of 1 MB capacity with a minimum load-to-use latency of 14 cycles. The mid-level cache capacity is four times larger than the capacity in previous Intel Xeon processor family implementations. The line size of the mid-level cache is 64B and it is 16-way associative. The mid-level cache is private to each core.

Software that has been optimized to place data in mid-level cache may have to be revised to take advantage of the larger mid-level cache available in Skylake Server microarchitecture.

2.5.1.2 Non-Inclusive Last Level Cache

The last level cache (LLC) in Skylake is a non-inclusive, distributed, shared cache. The size of each of the banks of last level cache has shrunk to 1.375 MB per bank. Because of the non-inclusive nature of the last level cache, blocks that are present in the mid-level cache of one of the cores may not have a copy resident in a bank of last level cache. Based on the access pattern, size of the code and data accessed, and sharing behavior between cores for a cache block, the last level cache may appear as a victim cache of the mid-level cache and the aggregate cache capacity per core may appear to be a combination of the private mid-level cache per core and a portion of the last level cache.

2.5.1.3 Skylake Server Microarchitecture Cache Recommendations

A high-level comparison between Skylake Server microarchitecture cache and the previous generation Broadwell microarchitecture cache is available in the table below.

Table 2-9. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L1 Data Cache Unit (DCU)	Size [KB]	32	32
	Latency [cycles]	4-6	4-6
	Max bandwidth [bytes/cycles]	96	192
	Sustained bandwidth [bytes/cycles]	93	133
	Associativity [ways]	8	8
L2 Mid-level Cache (MLC)	Size [KB]	256	1024 (1MB)
	Latency [cycles]	12	14
	Max bandwidth [bytes/cycles]	32	64
	Sustained bandwidth [bytes/cycles]	25	52
	Associativity [ways]	8	16

Table 2-9. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

L3 Last-level Cache (LLC)	Size [MB]	Up to 2.5 per core	up to 1.375[1] per core
	Latency [cycles]	50-60	50-70
	Max bandwidth [bytes/cycles]	16	32
	Sustained bandwidth [bytes/cycles]	14	15

NOTES:

1. Some Skylake Server parts have some cores disabled and hence have more than 1.375 MB per core of L3 cache.

The figure below shows how Skylake Server microarchitecture shifts the memory balance from shared-distributed with high latency, to private-local with low latency.

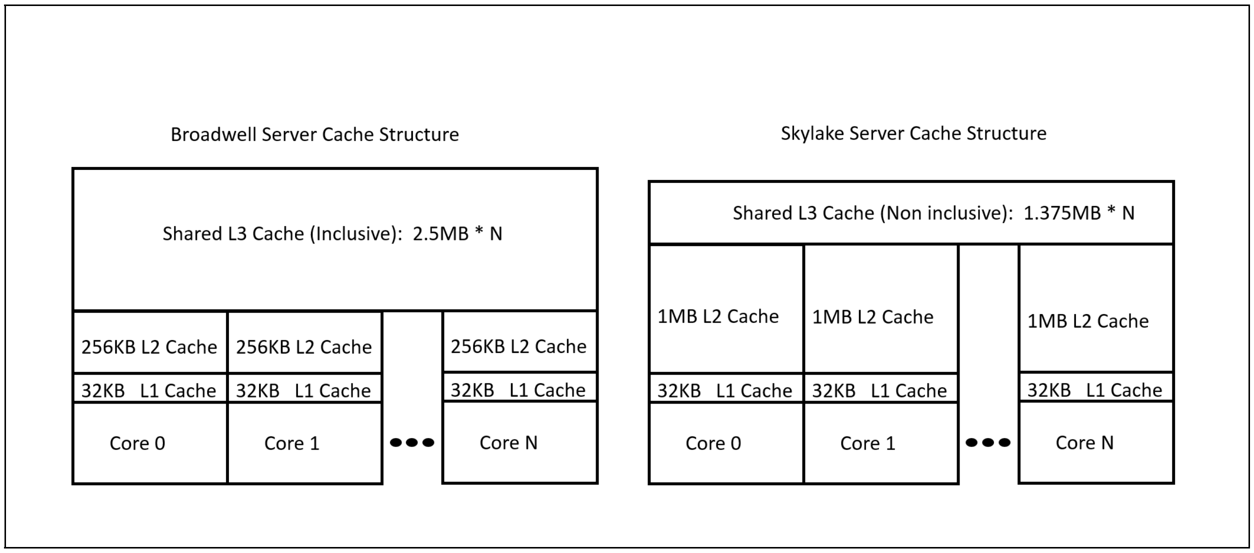

Broadwell Server Cache Structure
Shared L3 Cache (Inclusive): 2.5MB * N
256KB L2 Cache
32KB L1 Cache
Core 0
Core 1
Core N
Skylake Server Cache Structure
Shared L3 Cache (Non inclusive): 1.375MB * N
1MB L2 Cache
32KB L1 Cache
Core 0
Core 1
Core N

Figure 2-5. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures

The potential performance benefit from the cache changes is high, but software will need to adapt its memory tiling strategy to be optimal for the new cache sizes.

***Recommendation**: Rebalance application shared and private data sizes to match the smaller, non-inclusive L3 cache, and larger L2 cache.*

Choice of cache blocking should be based on application bandwidth requirements and changes from one application to another. Having four times the L2 cache size and twice the L2 cache bandwidth compared to the previous generation Broadwell microarchitecture enables some applications to block to L2 instead of L1 and thereby improves performance.

***Recommendation**: Consider blocking to L2 on Skylake Server microarchitecture if L2 can sustain the application's bandwidth requirements.*

The change from inclusive last level cache to non-inclusive means that the capacity of mid-level and last level cache can now be added together. Programs that determine cache capacity per core at run time should now use a combination of mid-level cache size and last level cache size per core to estimate the effective cache size per core. Using just the last level cache size per core may result in non-optimal use of available on-chip cache; see Section 2.5.2 for details.

***Recommendation:** In case of no data sharing, applications should consider cache capacity per core as L2 and L3 cache sizes and not only L3 cache size.*

2.5.2 Non-Temporal Stores on Skylake Server Microarchitecture

Because of the change in the size of each bank of last level cache on Skylake Server microarchitecture, if an application, library, or driver only considers the last level cache to determine the size of on-chip cache-per-core, it may see a reduction with Skylake Server microarchitecture and may use non-temporal store with smaller blocks of memory writes. Since non-temporal stores evict cache lines back to memory, this may result in an increase in the number of subsequent cache misses and memory bandwidth demands on Skylake Server microarchitecture, compared to the previous Intel Xeon processor family.

Also, because of a change in the handling of accesses resulting from non-temporal stores by Skylake Server microarchitecture, the resources within each core remain busy for a longer duration compared to similar accesses on the previous Intel Xeon processor family. As a result, if a series of such instructions are executed, there is a potential that the processor may run out of resources and stall, thus limiting the memory write bandwidth from each core.

The increase in cache misses due to overuse of non-temporal stores and the limit on the memory write bandwidth per core for non-temporal stores may result in reduced performance for some applications.

To avoid the performance condition described above with Skylake Server microarchitecture, include mid-level cache capacity per core in addition to the last level cache per core for applications, libraries, or drivers that determine the on-chip cache available with each core. Doing so optimizes the available on-chip cache capacity on Skylake Server microarchitecture as intended, with its non-inclusive last level cache implementation.

2.5.3 Skylake Server Power Management

This section describes the interaction of Skylake Server's Power Management and its Vector ISA.

Skylake Server microarchitecture dynamically selects the frequency at which each of its cores executes. The selected frequency depends on the instruction mix; the type, width, and number of vector instructions that execute over a given period of time. The processor also takes into account the number of cores that share similar characteristics.

Intel® Xeon® processors based on Broadwell microarchitecture work similarly, but to a lesser extent since they only support 256-bit vector instructions. Skylake Server microarchitecture supports Intel® AVX-512 instructions, which can potentially draw more current and more power than Intel® AVX2 instructions.

The processor dynamically adjusts its maximum frequency to higher or lower levels as necessary, therefore a program might be limited to different maximum frequencies during its execution.

Table 2-10 includes information about the maximum Intel® Turbo Boost technology core frequency for each type of instruction executed. The maximum frequency (P0n) is an array of frequencies which depend on the number of cores within the category. The more cores belonging to a category at any given time, the lower the maximum frequency.

Table 2-10. Maximum Intel® Turbo Boost Technology Core Frequency Levels

Level	Category	Frequency Level	Max Frequency (P0n)	Instruction Types
0	Intel® AVX2 light instructions	Highest	Max	Scalar, AVX128, SSE, Intel® AVX2 w/o FP or INT MUL/FMA
1	Intel® AVX2 heavy instructions + Intel® AVX-512 light instructions	Medium	Max Intel® AVX2	Intel® AVX2 FP + INT MUL/FMA, Intel® AVX-512 without FP or INT MUL/FMA
2	Intel® AVX-512 heavy instructions	Lowest	Max Intel® AVX-512	Intel® AVX-512 FP + INT MUL/FMA

For per SKU max frequency details (reference figure 1-15), refer to the Intel® Xeon® Processor Scalable Family Technical Resources page.

Figure 2-6 is an example for core frequency range in a given system where each core frequency is determined independently based on the demand of the workload.

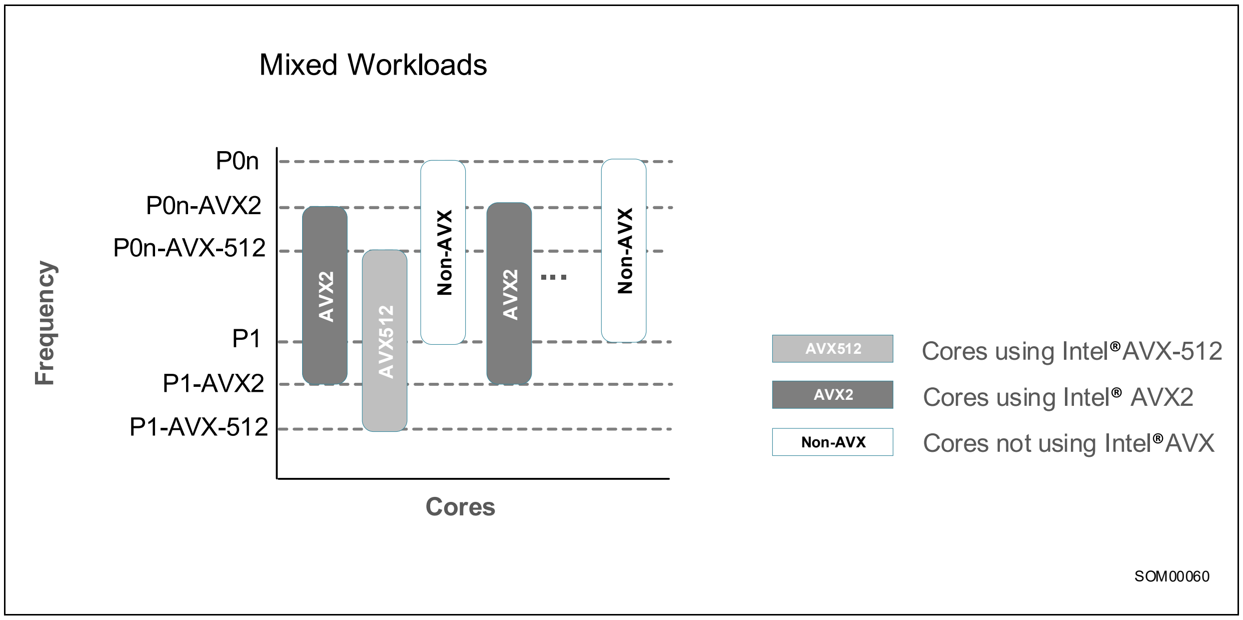


Figure 2-6. Mixed Workloads

The following performance monitoring events can be used to determine how many cycles were spent in each of the three frequency levels.

- CORE_POWER.LVL0_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n.
- CORE_POWER.LVL1_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX2.
- CORE_POWER.LVL2_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX-512.

When the core requests a higher license level than its current one, it takes the PCU up to 500 micro-seconds to grant the new license. Until then the core operates at a lower peak capability. During this time period the PCU evaluates how many cores are executing at the new license level and adjusts their frequency as necessary, potentially lowering the frequency. Cores that execute at other license levels are not affected.

A timer of approximately 2ms is applied before going back to a higher frequency level. Any condition that would have requested a new license resets the timer.

NOTES

A license transition request may occur when executing instructions on a mis-speculated path.

A large enough mix of Intel AVX-512 light instructions and Intel AVX2 heavy instructions drives the core to request License 2, despite the fact that they usually map to License 1. The same is true for Intel AVX2 light instructions and Intel SSE heavy instructions that may drive the core to License 1 rather than License 0. For example, The Intel® Xeon® Platinum 8180 processor moves from license 1 to license 2 when executing a mix of 110 Intel AVX-512 light instructions and 20 256-bit heavy instructions over a window of 65 cycles.

Some workloads do not cause the processor to reach its maximum frequency as these workloads are bound by other factors. For example, the LINPACK benchmark is power limited and does not reach the processor's maximum frequency. The following graph shows how frequency degrades as vector width grows, but, despite the frequency drop, performance improves. The data for this graph was collected on an Intel Xeon Platinum 8180 processor.

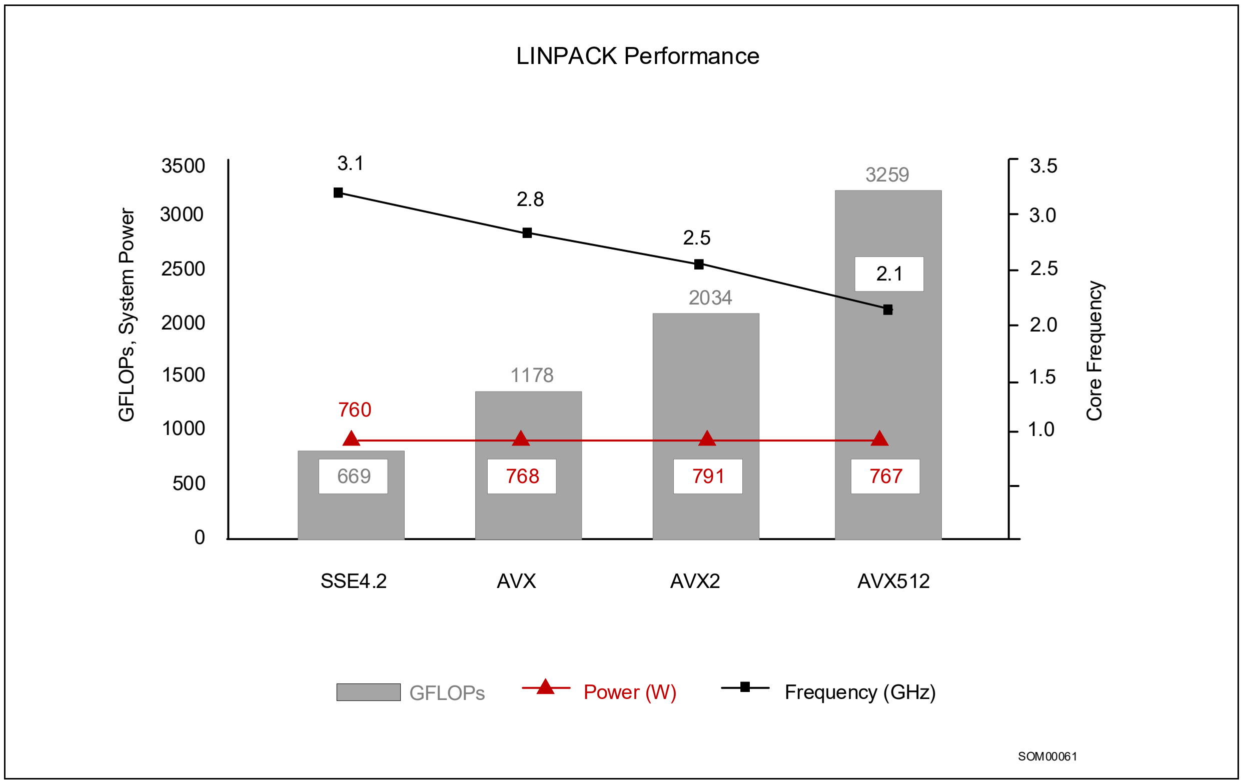


Figure 2-7. LINPACK Performance

Workloads that execute Intel AVX-512 instructions as a large proportion of their whole instruction count can gain performance compared to Intel AVX2 instructions, even though they may operate at a lower frequency. For example, maximum frequency bound Deep Learning workloads that target Intel AVX-512 heavy instructions at a very high percentage can gain 1.3x-1.5x performance improvement vs. the same workload built to target Intel AVX2 (both operating on Skylake Server microarchitecture).

It is not always easy to predict whether a program's performance will improve from building it to target Intel AVX-512 instructions. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to the use of zmm registers. However, some programs that use zmm registers may not gain as much, or may even lose performance. It is recommended to try multiple build options and measure the performance of the program.

Recommendation: To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -xCORE-AVX2 -mtune=skylake-avx512 (Linux* and macOS*)
  /QxCORE-AVX2 /tune=skylake-avx512 (Windows*)
- -xCORE-AVX512 -qopt-zmm-usage=low (Linux* and macOS*)
  /QxCORE-AVX512 /Qopt-zmm-usage:low (Windows*)
- -xCORE-AVX512 -qopt-zmm-usage=high (Linux* and macOS*)
  /QxCORE-AVX512 /Qopt-zmm-usage:high (Windows*)

See Section 18.26, “CLDEMOTE” for more information about these options.

The GCC Compiler has the option -mprefer-vector-width=none|128|256|512 to control vector width preference. While -march=skylake-avx512 is designed to provide the best performance for the Skylake Server microarchitecture some programs can benefit from different vector width preferences. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance. -mprefer-vector-width=256 is the default for skylake-avx512.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512
- -march=skylake-avx512 -mprefer-vector-width=512

Clang/LLVM is currently implementing the option -mprefer-vector-width=none|128|256|512, similar to GCC. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512 (plus -mprefer-vector-width=256, if available)
- -march=skylake-avx512 (plus -mprefer-vector-width=512, if available)

2.6 SKYLAKE CLIENT MICROARCHITECTURE

The Skylake Client microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake Client microarchitecture is depicted in Figure 2-8.

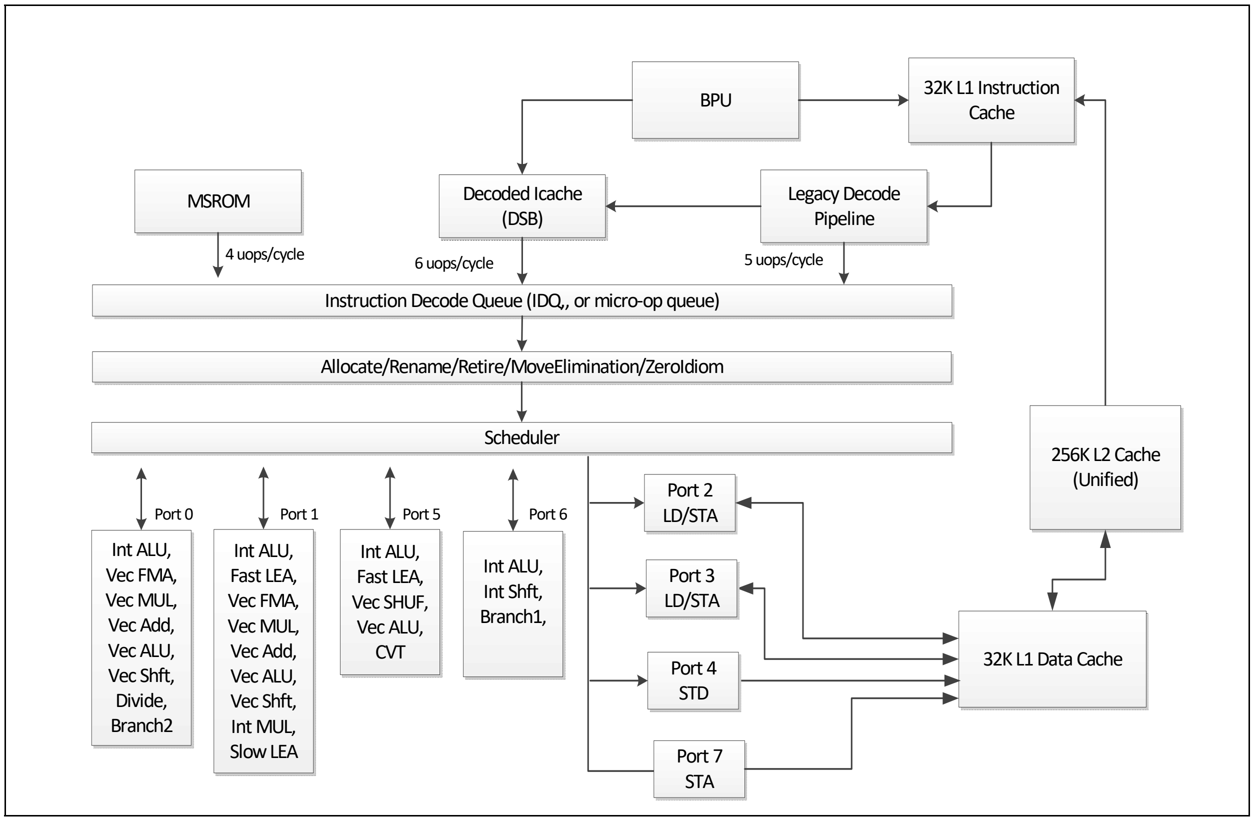

BPU
32K L1 Instruction Cache
MSROM
Decoded Icache (DSB)
Legacy Decode Pipeline
4 uops/cycle
6 uops/cycle
5 uops/cycle
Instruction Decode Queue (IDQ,, or micro-op queue)
Allocate/Rename/Retire/MoveElimination/ZeroIdiom
Scheduler
256K L2 Cache (Unified)
Port 0
Port 1
Port 5
Port 6
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Int ALU, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Divide, Branch2
Int ALU, Fast LEA, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Int MUL, Slow LEA
Int ALU, Fast LEA, Vec SHUF, Vec ALU, CVT
Int ALU, Int Shft, Branch1,
32K L1 Data Cache

Figure 2-8. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture

The Skylake Client microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. A four-core configuration can be supported similar to the arrangement shown in Appendix E, "Earlier Generations of Intel® 64 and IA-32 Processor Architectures," Figure E-2.

2.6.1 The Front End

The front end in the Skylake Client microarchitecture provides the following improvements over previous generation microarchitectures:

- Legacy Decode Pipeline delivery of 5 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The DSB delivers 6 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The IDQ can hold 64 uops per logical processor vs. 28 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 uops (64 vs. 56 uops in ST operation).
- The LSD in the IDQ can detect loops up to 64 uops per logical processor irrespective ST or SMT operation.
- Improved Branch Predictor.

2.6.2 The Out-of-Order Execution Engine

The Out of Order and execution engine changes in Skylake Client microarchitecture include:

- Larger buffers enable deeper OOO execution compared to previous generations.
- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Longer pause latency enables better power efficiency and better SMT performance resource utilization.

Table 2-11 summarizes the OOO engine's capability to dispatch different types of operations to various ports.

Table 2-11. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

Table 2-12 lists execution units and common representative instructions that rely on these units. Throughput improvements across the SSE, AVX and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-12. Skylake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

A significant portion of the Intel SSE, Intel AVX and general-purpose instructions also have latency improvements. Appendix C lists the specific details. Software-visible latency exposure of an instruction sometimes may include additional contributions that depend on the relationship between micro-ops flows of the producer instruction and the micro-op flows of the ensuing consumer instruction. For example, a two-uop instruction like VPMULLD may experience two cumulative bypass delays of 1 cycle each from each of the two micro-ops of VPMULLD.

Table 2-13 describes the bypass delay in cycles between a producer uop and the consumer uop. The left-most column lists a variety of situations characteristic of the producer micro-op. The top row lists a variety of situations characteristic of the consumer micro-op.

Table 2-13. Bypass Delay Between Producer and Consumer Micro-ops

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "VIMUL/0,1/4" applies to 4-cycle vector integer multiply uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.

2.6.3 Cache and Memory Subsystem

The cache hierarchy of the Skylake Client microarchitecture has the following enhancements:

- Higher Cache bandwidth compared to previous generations.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Processor can do two page walks in parallel compared to one in Haswell microarchitecture and earlier generations.
- Page split load penalty down from 100 cycles in previous generation to 5 cycles.
- L3 write bandwidth increased from 4 cycles per line in previous generation to 2 per line.
- Support for the CLFLUSHOPT instruction to flush cache lines and manage memory ordering of flushed data using SFENCE.
- Reduced performance penalty for a software prefetch that specifies a NULL pointer.
- L2 associativity changed from 8 ways to 4 ways.

Table 2-14. Cache Parameters of the Skylake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2. The partition column of Table 2-15 indicates the resource sharing policy when Hyper-Threading Technology is active.

Table 2-15. TLB Parameters of the Skylake Client Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.6.4 Pause Latency in Skylake Client Microarchitecture

The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as WaitForSingleObject on Windows* OS or futex on Linux.

The PAUSE instruction is intended to:

- Temporarily provide the sibling logical processor (ready to make forward progress exiting the spin loop) with competitively shared hardware resources. The competitively-shared microarchitectural resources that the sibling logical processor can utilize in the Skylake Client microarchitecture are listed below.
  - Front end slots in the Decode ICache, LSD and IDQ.
  - Execution slots in the RS.
- Save power consumed by the processor core compared with executing equivalent spin loop instruction sequence in the following configurations.
  - One logical processor is inactive (e.g., entering a C-state).
  - Both logical processors in the same core execute the PAUSE instruction.

— HT is disabled (e.g. using BIOS options).

The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake Client microarchitecture it has been extended to as many as 140 cycles.

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.

As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.

The following is an example of how to use the PAUSE instruction with a dynamic loop iteration count.

Notice that in the Skylake Client microarchitecture the RDTSC instruction counts at the machine's guaranteed P1 frequency independently of the current processor clock (see the INVARIANT TSC property), and therefore, when running in Intel® Turbo-Boost-enabled mode, the delay will remain constant, but the number of instructions that could have been executed will change.

Use Poll Delay function in your lock to wait a given amount of guaranteed P1 frequency cycles, specified in the “clocks” variable.

Example 2-8. Dynamic Pause Loop Example

```
#include <x86intrin.h>
#include <stdint.h>

/* A useful predicate for dealing with timestamps that may wrap.
 Is a before b? Since the timestamps may wrap, this is asking whether it’s
 shorter to go clockwise from a to b around the clock-face, or anti-clockwise.
 Times where going clockwise is less distance than going anti-clockwise
 are in the future, others are in the past. e.g. a = MAX-1, b = MAX+1 (=0),
 then a > b (true) does not mean a reached b; whereas signed(a) = -2,
 signed(b) = 0 captures the actual difference */

static inline bool before(uint64_t a, uint64_t b)
{
   return ((int64_t)b - (int64_t)a) > 0;
}

void pollDelay(uint32_t clocks)
{
   uint64_t endTime = _rdtsc()+ clocks;

   for (; before(_rdtsc(), endTime); )
      _mm_pause();
}
```

For contended spinlocks of the form shown in the baseline example below, we recommend an exponential back off when the lock is found to be busy, as shown in the improved example, to avoid significant performance degradation that can be caused by conflicts between threads in the machine. This is more important as we increase the number of threads in the machine and make changes to the architecture that might aggravate these conflict conditions. In multi-socket Intel server processors with shared memory, conflicts across threads take much longer to resolve as the number of threads contending for the same lock increases. The exponential back off is designed to avoid these conflicts between the threads thus avoiding the potential performance degradation. Note that in the example below, the number of PAUSE instructions are increased by a factor of 2 until some MAX_BACKOFF is reached which is subject to tuning.

Example 2-9. Contended Locks with Increasing Back-off Example

```
/*******************/
/*Baseline Version */
/*******************/

// atomic {if (lock == free) then change lock state to busy}
while (cmpxchg(lock, free, busy) == fail)
{
    while (lock == busy)
    {
        __asm__ ("pause");
    }
}
```

```
/*******************/
/*Improved Version */
/*******************/

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
       for (int i=mask; i; --i){
           __asm__ ("pause");
       }
         mask = mask < max ? mask<<1 : max;
   }
}
```

2.7 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT TECHNOLOGY)

Intel® Hyper-Threading Technology (Intel® HT Technology) enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package, or within each processor core in a physical processor package. In its first implementation in the Intel Xeon processor, Hyper-Threading Technology makes a single physical processor (or a processor core) appear as two or more logical processors. Intel Xeon Phi processors based on the Knights Landing microarchitecture support 4 logical processors in each processor core; see Chapter 23 for detailed information of Intel HT Technology that is implemented in the Knights Landing microarchitecture.

Most Intel Architecture processor families support Hyper-Threading Technology with two logical processors in each processor core, or in a physical processor in early implementations. The rest of this section describes features of the early implementation of Hyper-Threading Technology. Most of the descriptions also apply to later Hyper-Threading Technology implementations supporting two logical processors. The microarchitecture sections in this chapter provide additional details to individual microarchitecture and enhancements to Hyper-Threading Technology.

The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an Intel HT

Technology capable processor looks like two processors to software, including operating system and application code.

By sharing resources needed for peak demands between two logical processors, Intel HT Technology is well suited for multiprocessor systems to provide an additional performance boost in throughput when compared to traditional MP systems.

Figure 2-9 shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting Intel HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an "add" operation from logical processor 0 and another "add" operation and load from logical processor 1 can be executed simultaneously by the execution engine.

In the first implementation of Intel HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor. This minimizes the die area cost of implementing Intel HT Technology while still achieving performance gains for multithreaded applications or multitasking workloads.

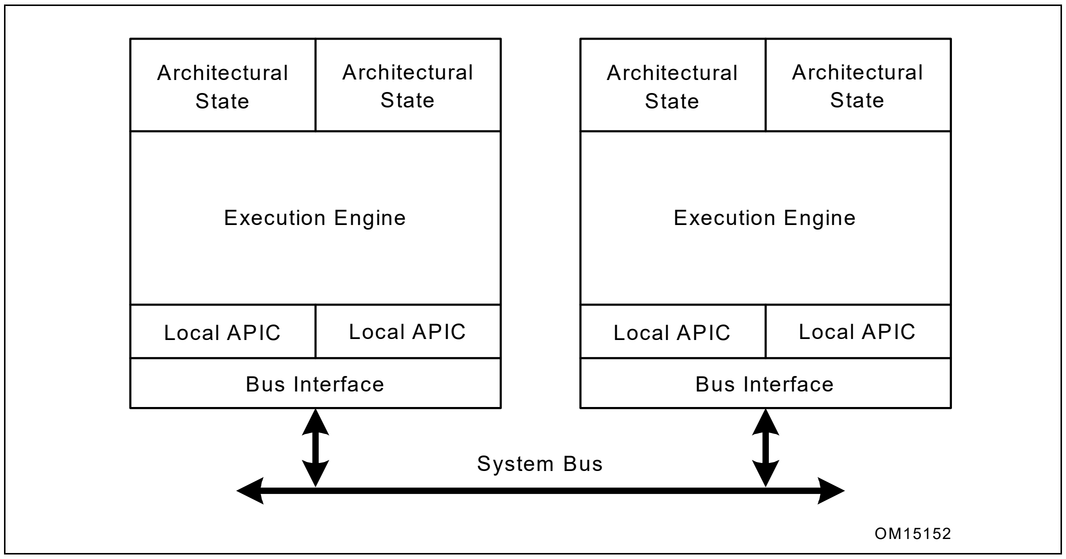


Figure 2-9. Hyper-Threading Technology on an SMP

The performance potential due to HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor.
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput.

2.7.1 Processor Resources and HT Technology

The majority of microarchitecture resources in a physical processor are shared between the logical processors. Only a few small data structures were replicated for each logical processor. This section describes how resources are shared, partitioned or replicated.

2.7.1.1 Replicated Resources

The architectural state is replicated for each logical processor. The architecture state consists of registers that are used by the operating system and application code to control program behavior and store data for computations. This state includes the eight general-purpose registers, the control registers, machine state registers, debug registers, and others. There are a few exceptions, most notably the memory type range registers (MTRRs) and the performance monitoring resources. For a complete list of the architecture state and exceptions, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D.

Other resources such as instruction pointers and register renaming tables were replicated to simultaneously track execution and state changes of the two logical processors. The return stack predictor is replicated to improve branch prediction of return instructions.

In addition, a few buffers (for example, the 2-entry instruction streaming buffers) were replicated to reduce complexity.

2.7.1.2 Partitioned Resources

Several buffers are shared by limiting the use of each logical processor to half the entries. These are referred to as partitioned resources. Reasons for this partitioning include:

- Operational fairness.
- Permitting the ability to allow operations from one logical processor to bypass operations of the other logical processor that may have stalled.

For example: a cache miss, a branch misprediction, or instruction dependencies may prevent a logical processor from making forward progress for some number of cycles. The partitioning prevents the stalled logical processor from blocking forward progress.

In general, the buffers for staging instructions between major pipe stages are partitioned. These buffers include µop queues after the execution trace cache, the queues after the register rename stage, the reorder buffer which stages instructions for retirement, and the load and store buffers.

In the case of load and store buffers, partitioning also provided an easier implementation to maintain memory ordering for each logical processor and detect memory ordering violations.

2.7.1.3 Shared Resources

Most resources in a physical processor are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units. Some shared resources which are linearly addressed, like the DTLB, include a logical processor ID bit to distinguish whether the entry belongs to one logical processor or the other.

2.7.2 Microarchitecture Pipeline and Intel® HT Technology

This section describes the Intel HT Technology microarchitecture and how instructions from the two logical processors are handled between the front end and the back end of the pipeline.

Although instructions originating from two programs or two threads execute simultaneously and not necessarily in program order in the execution core and memory hierarchy, the front end and back end contain several selection points to select between instructions from the two logical processors. All selection points alternate between the two logical processors unless one logical processor cannot make use of a pipeline stage. In this case, the other logical processor has full use of every cycle of the pipeline stage. Reasons why a logical processor may not use a pipeline stage include cache misses, branch mispredictions, and instruction dependencies.

2.7.3 Execution Core

The core can dispatch up to six μops per cycle, provided the μops are ready to execute. Once the μops are placed in the queues waiting for execution, there is no distinction between instructions from the two logical processors. The execution core and memory hierarchy is also oblivious to which instructions belong to which logical processor.

After execution, instructions are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned such that each uses half the entries.

2.7.4 Retirement

The retirement logic tracks when instructions from the two logical processors are ready to be retired. It retires the instruction in program order for each logical processor by alternating between the two logical processors. If one logical processor is not ready to retire any instructions, then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the processor needs to write the store data into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

2.8 SIMD TECHNOLOGY

SIMD computations (see Figure 2-10) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see Figure 2-11).

The Pentium III processor extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see Figure 2-11). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-10 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

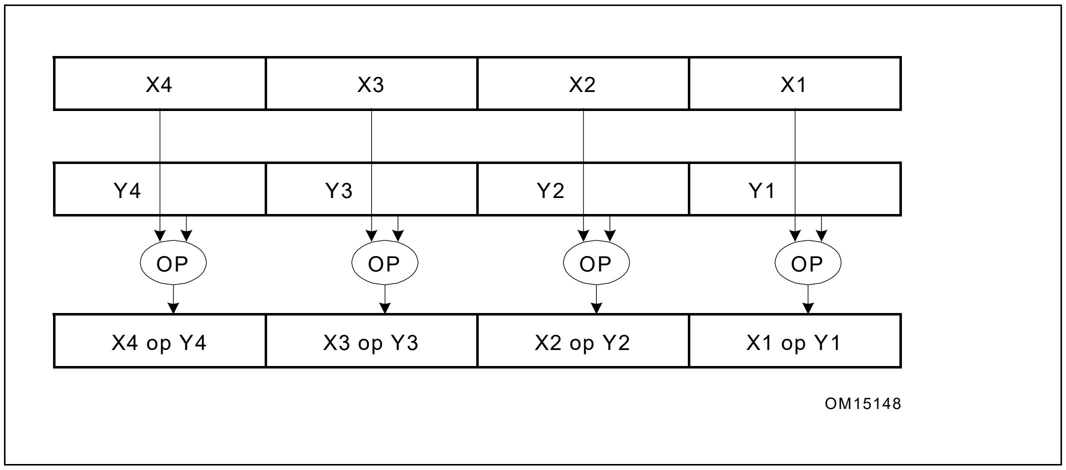


Figure 2-10. Typical SIMD Operations

The Pentium 4 processor further extended the SIMD computation model with the introduction of Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Intel Xeon processor 5100 series introduced Supplemental Streaming SIMD Extensions 3 (SSSE3).

SSE2 works with operands in either memory or in the XMM registers. The technology extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in SSE2 that operate on two packed double-precision floating-point data elements or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See Figure 2-11.

SSSE3 provides additional enhancement for SIMD computation with 32 instructions on digital video and signal processing.

SSE4.1, SSE4.2 and AESNI are additional SIMD extensions that provide acceleration for applications in media processing, text/lexical processing, and block encryption/decryption.

The SIMD extensions operates the same way in Intel 64 architecture as in IA-32 architecture, with the following enhancements:

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.
- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

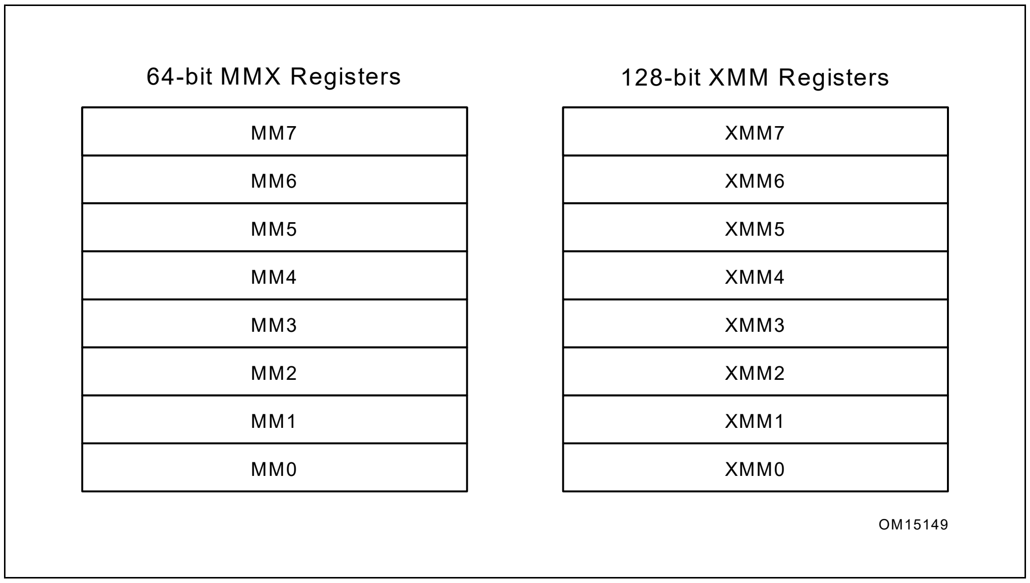


Figure 2-11. SIMD Instruction Register Usage

SIMD improves the performance of 3D graphics, speech recognition, image processing, scientific applications and applications that have the following characteristics:

- Inherently parallel.
- Recurring memory access patterns.
- Localized recurring operations performed on the data.
- Data-independent control flow.

2.9 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. They are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

SSE, SSE2, and MMX technologies are architectural extensions. Existing software will continue to run correctly, without modification on Intel microprocessors that incorporate these technologies. Existing software will also run correctly in the presence of applications that incorporate SIMD technologies.

SSE and SSE2 instructions also introduced cacheability and memory ordering instructions that can improve cache usage and application performance.

For more on SSE, SSE2, SSE3 and MMX technologies, see the following chapters in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*:

- Chapter 9, "Programming with Intel® MMX™ Technology."
- Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."
- Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."
- Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI."
- Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2."
- Chapter 15, "Programming with Intel® AVX-512."
- Chapter 16, "Programming with Intel® Transactional Synchronization Extensions."

2.9.1 MMX™ Technology

MMX Technology introduced:

- 64-bit MMX registers.
- Support for SIMD operations on packed byte, word, and doubleword integers.

Recommendation: Integer SIMD code written using MMX instructions should consider more efficient implementations using SSE/Intel AVX instructions.

2.9.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

- 128-bit XMM registers.
- 128-bit data type with four packed single-precision floating-point operands.
- Data prefetch instructions.
- Non-temporal store instructions and other cacheability and memory ordering instructions.
- Extra 64-bit SIMD integer support.

SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

2.9.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

- 128-bit data type with two packed double-precision floating-point operands.
- 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Support for SIMD arithmetic on 64-bit integer operands.

- Instructions for converting between new and existing data types.
- Extended support for data shuffling.
- Extended support for cacheability and memory ordering operations.

SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

2.9.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

- SIMD floating-point instructions for asymmetric and horizontal computation.
- A special-purpose 128-bit load instruction to avoid cache line splits.
- An x87 FPU instruction to convert to integer independent of the floating-point control word (FCW).
- Instructions to support thread synchronization.

SSE3 instructions are useful for scientific, video and multi-threaded applications.

2.9.5 Supplemental Streaming SIMD Extensions 3

The Supplemental Streaming SIMD Extensions 3 introduces 32 new instructions to accelerate eight types of computations on packed integers. These include:

- 12 instructions that perform horizontal addition or subtraction operations.
- 6 instructions that evaluate the absolute values.
- 2 instructions that perform multiply and add operations and speed up the evaluation of dot products.
- 2 instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- 2 instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- 6 instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- 2 instructions that align data from the composite of two operands.

2.9.6 SSE4.1

SSE4.1 introduces 47 new instructions to accelerate video, imaging and 3D applications. SSE4.1 also improves compiler vectorization and significantly increase support for packed dword computation. These include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction provides a streaming hint for WC loads.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations of word integers.
- One instruction improves masked comparisons.

- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

2.9.7 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
  - Basic and advanced string library functions from strlen, strcmp, to strcspn.
  - Delimiter processing, token extraction for lexing of text streams.
  - Parser, schema validation including XML processing.
- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.9.8 AESNI and PCLMULQDQ

AESNI introduces 7 new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES standard supports cipher key of sizes 128, 192, and 256 bits. The respective cipher key sizes correspond to 10, 12, and 14 rounds of iteration.

AES encryption involves processing 128-bit input data (plain text) through a finite number of iterative operation, referred to as "AES round", into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the "equivalent inverse cipher" instead of the "inverse cipher".

The cryptographic processing at each round involves two input data, one is the "state", the other is the "round key". Each round uses a different "round key". The round keys are derived from the cipher key using a "key schedule" algorithm. The "key schedule" algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

2.9.9 Intel® Advanced Vector Extensions (Intel® AVX)

Intel® Advanced Vector Extensions (Intel® AVX) offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.

- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

Intel AVX instruction set and 256-bit register state management detail are described in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D. Optimization techniques for Intel AVX are discussed in Chapter 15, "Optimizations for Intel® AVX, Intel® AVX2, and Intel® FMA."

2.9.10 Half-Precision Floating-Point Conversion (F16C)

VCVTPH2PS and VCVTPS2PH are two instructions supporting half-precision floating-point data type conversion to and from single-precision floating-point data types. These two instruction extends on the same programming model as Intel AVX.

2.9.11 RDRAND

The RDRAND instruction retrieves a random number supplied by a cryptographically secure, deterministic random bit generator (DBRG). The DBRG is designed to meet NIST SP 800-90A standard.

2.9.12 Fused-Multiply-ADD (FMA) Extensions

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract operations. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

2.9.13 Intel® AVX2

Intel AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.9.14 General-Purpose Bit-Processing Instructions

The fourth generation Intel Core processor family introduces a collection of bit processing instructions that operate on the general purpose registers. The majority of these instructions uses the VEX-prefix encoding scheme to provide non-destructive source operand syntax.

There instructions are enumerated by three separate feature flags reported by CPUID. For details, see Section 5.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 and chapters 3, 4 and 5 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

2.9.15 Intel® Transactional Synchronization Extensions

The fourth generation Intel Core processor family introduces Intel® Transactional Synchronization Extensions (Intel® TSX), which aim to improve the performance of lock-protected critical sections of multithreaded applications while maintaining the lock-based programming model.

For background and details, see Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Software tuning recommendations for using Intel TSX on lock-protected critical sections of multithreaded applications are described in Chapter 16, "Intel® TSX Recommendations."

2.9.16 RDSEED

The RDSEED instruction retrieves a random number supplied by a cryptographically secure, enhanced deterministic random bit generator Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP 800-90C standards.

2.9.17 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics.

3. Updates to Chapter 3

Change bars and **violet** text show changes to Chapter 3of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: General Optimization Guidelines.

Changes to this chapter:

- Updated capitalization of headings throughout chapter.
- Updated branding throughout chapter.
- Typo and punctuation corrections as necessary.
- Section 3.4.2.1:
  - Changed Micro-fusion to Microfusion in heading to match Macrofusion.
- Section 3.5.2.3:
  - Removed outdated technologies references, focusing on information starting from Skylake microarchitecture.
  - Provided latest information regarding *H micro-operations.
  - Provided a new figure showing location of *H in Port one.
- Section 3.11:
  - Updated various items for clarity.
  - Section 3.11.2: Added corrected recipes.
  - Table 3-8: Changed column orientation to be more legible.
  - Table 3-11: Config section now includes workers field.
- Section 3.12: Updated for clarity.

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

This chapter discusses general optimization techniques that can improve the performance of applications running on Intel® processors. These techniques take advantage of microarchitectural features described in Chapter 2, "Intel® 64 and IA-32 Processor Architectures." Optimization guidelines focusing on Intel multi-core processors, Hyper-Threading Technology, and 64-bit mode applications are discussed in Chapter 11, "Multicore and Intel® Hyper-Threading Technology (intel® HT)," and Chapter 13, "64-bit Mode Coding Guidelines."

Practices that optimize performance focus on three areas:

- Tools and techniques for code generation.
- Analysis of the performance characteristics of the workload and its interaction with microarchitectural sub-systems.
- Tuning code to the target microarchitecture (or families of microarchitecture) to improve performance.

Some hints on using tools are summarized first to simplify the first two tasks. The rest of the chapter will focus on recommendations for code generation or code tuning to the target microarchitectures.

This chapter explains optimization techniques for the Intel® C++ Compiler, the Intel® Fortran Compiler, and other compilers.

3.1 PERFORMANCE TOOLS

Intel offers several tools to help optimize application performance, including compilers, performance analysis, and multithreading tools.

3.1.1 Intel® C++ and Fortran Compilers

Intel compilers support multiple operating systems (Windows*, Linux*, Mac OS*, and embedded). The Intel compilers optimize performance and give application developers access to advanced features, including:

- Flexibility to target 32-bit or 64-bit Intel processors for optimization.
- Compatibility with many integrated development environments or third-party compilers.
- Automatic optimization features to take advantage of the target processor's architecture.
- Automatic compiler optimization reduces the need to write different code for different processors.
- Common compiler features that are supported across Windows, Linux, and Mac OS include:
    - General optimization settings.
    - Cache-management features.
    - Interprocedural optimization (IPO) methods.
    - Profile-guided optimization (PGO) methods.
    - Multithreading support.
    - Floating-point arithmetic precision and consistency support.
    - Compiler optimization and vectorization reports.

3.1.2 General Compiler Recommendations

Generally speaking, a compiler tuned for a target microarchitecture can be expected to match or outperform hand-coding. However, if performance problems are noted with the compiled code, some compilers (like Intel C++ and Fortran compilers) allow the coder to insert intrinsics or inline assembly to exert control over generated code. If inline assembly is used, the user must verify that the code generated is high quality and yields good performance.

Default compiler switches are targeted for common cases. An optimization may be made to the compiler default if it benefits most programs. If the root cause of a performance problem is a poor choice on the part of the compiler, using different switches or compiling the targeted module with a different compiler may be the solution. See the "Quick Reference Guide to Optimization with Intel® C++ and Fortran Compilers" for additional suggestions on compiler Optimization Options, including processor-specific ones.

3.1.3 VTune™ Performance Analyzer

VTune uses performance monitoring hardware to collect statistics and coding information about your application and its interaction with the microarchitecture. This allows software engineers to measure performance characteristics of the workload for a given microarchitecture. VTune supports all current and past Intel processor families.

The VTune Performance Analyzer provides two kinds of feedback:

- Indication of a performance improvement gained by using a specific coding recommendation or microarchitectural feature.
- Information on whether a change in the program has improved or degraded performance with respect to a particular metric.

The VTune Performance Analyzer also provides measures for a number of workload characteristics, including:

- Retirement throughput of instruction execution as an indication of the degree of extractable instruction-level parallelism in the workload.
- Data traffic locality as an indication of the stress point of the cache and memory hierarchy.
- Data traffic parallelism as an indication of the degree of effectiveness of amortization of data access latency.

NOTE

Improving performance in one part of the machine does not necessarily bring significant gains to overall performance. It is possible to degrade overall performance by improving performance for some particular metric.

Where appropriate, coding recommendations in this chapter include descriptions of the VTune Performance Analyzer events that provide measurable data on the performance gain achieved by following the recommendations. For more on using the VTune analyzer, refer to the application's online help.

3.2 PROCESSOR PERSPECTIVES

Many coding recommendations work well across current microarchitectures. However, there are situations where a recommendation may benefit one microarchitecture more than another.

3.2.1 CPUID Dispatch Strategy and Compatible Code Strategy

When optimum performance on all processor generations is desired, applications can take advantage of the CPUID instruction to identify the processor generation and integrate processor-specific instructions into the source code. The Intel C++ Compiler supports the integration of different versions of the code for different target processors. The selection of which code to execute at runtime is made based on the CPU identifiers. Binary code targeted for different processor generations can be generated under the control of the programmer or by the compiler. Refer to the "Intel® C++ Compiler Classic Developer Guide and Reference" cpu_dispatch and cpu_specific sections for more information on CPU dispatching (a.k.a function multi-versioning).

For applications that target multiple generations of microarchitectures, and where minimum binary code size and single code path is important, a compatible code strategy is the best. Optimizing applications using techniques developed for the Intel Core microarchitecture combined with Nehalem microarchitecture are likely to improve code efficiency and scalability when running on processors based on current and future generations of Intel 64 and IA-32 processors.

3.2.2 Transparent Cache-Parameter Strategy

If the CPUID instruction supports function leaf 4, also known as deterministic cache parameter leaf, the leaf reports cache parameters for each level of the cache hierarchy in a deterministic and forward-compatible manner across Intel 64 and IA-32 processor families.

For coding techniques that rely on specific parameters of a cache level, using the deterministic cache parameter allows software to implement techniques in a way that is forward-compatible with future generations of Intel 64 and IA-32 processors, and cross-compatible with processors equipped with different cache sizes.

3.2.3 Threading Strategy and Hardware Multithreading Support

Intel 64 and IA-32 processor families offer hardware multithreading support in two forms: multi-core technology and HT Technology.

To fully harness the performance potential of hardware multithreading in current and future generations of Intel 64 and IA-32 processors, software must embrace a threaded approach in application design. At the same time, to address the widest range of installed machines, multithreaded software should be able to run without failure on a single processor without hardware multithreading support and should achieve performance on a single logical processor that is comparable to an unthreaded implementation (if such comparison can be made). This generally requires architecting a multithreaded application to minimize the overhead of thread synchronization. Additional guidelines on multithreading are discussed in Chapter 11, "Multicore and Intel® Hyper-Threading Technology (intel® HT)."

3.3 CODING RULES, SUGGESTIONS, AND TUNING HINTS

This section includes rules, suggestions, and hints. They are targeted for engineers who are:

- Modifying source code to enhance performance (user/source rules).
- Writing assemblers or compilers (assembly/compiler rules).
- Doing detailed performance tuning (tuning suggestions).

Coding recommendations are ranked in importance using two measures:

- Local impact (high, medium, or low) refers to a recommendation's affect on the performance of a given instance of code.
- Generality (high, medium, or low) measures how often such instances occur across all application domains. Generality may also be thought of as "frequency."

These recommendations are approximate. They can vary depending on coding style, application domain, and other factors.

The purpose of the high, medium, and low (H, M, and L) priorities is to suggest the relative level of performance gain one can expect if a recommendation is implemented.

Because it is not possible to predict the frequency of a particular code instance in applications, priority hints cannot be directly correlated to application-level performance gain. In cases in which application-level performance gain has been observed, we have provided a quantitative characterization of the gain (for information only). In cases in which the impact has been deemed inapplicable, no priority is assigned.

3.4 OPTIMIZING THE FRONT END

Optimizing the front end covers two aspects:

- Maintaining steady supply of micro-ops to the execution engine — Mispredicted branches can disrupt streams of micro-ops, or cause the execution engine to waste execution resources on executing streams of micro-ops in the non-architected code path. Much of the tuning in this respect focuses on working with the Branch Prediction Unit. Common techniques are covered in Section 3.4.1, "Branch Prediction Optimization."
- Supplying streams of micro-ops to utilize the execution bandwidth and retirement bandwidth as much as possible — For Intel Core microarchitecture and Intel Core Duo processor family, this aspect focuses maintaining high decode throughput. In Sandy Bridge microarchitecture, this aspect focuses on keeping the hot code running from Decoded ICache. Techniques to maximize decode throughput for Intel Core microarchitecture are covered in Section 3.4.2, "Fetch and Decode Optimization."

3.4.1 Branch Prediction Optimization

Branch optimizations have a significant impact on performance. By understanding the flow of branches and improving their predictability, you can increase the speed of code significantly.

Optimizations that help branch prediction are:

- Keep code and data on separate pages. This is very important; see Section 3.6, "Optimizing Memory Accesses," for more information.
- Eliminate branches whenever possible.
- Arrange code to be consistent with the static branch prediction algorithm.
- Use the PAUSE instruction in spin-wait loops.
- Inline functions and pair up calls and returns.
- Unroll as necessary so that repeatedly-executed loops have sixteen or fewer iterations (unless this causes an excessive code size increase).
- Avoid putting multiple conditional branches in the same 8-byte aligned code block (i.e, have their last bytes' addresses within the same 8-byte aligned code) if the lower 6 bits of their target IPs are the same. This restriction has been removed in Ice Lake Client and later microarchitectures.

3.4.1.1 Eliminating Branches

Eliminating branches improves performance because:

- It reduces the possibility of mispredictions.
- It reduces the number of required branch target buffer (BTB) entries. Conditional branches that are never taken do not consume BTB resources.

There are four principal ways of eliminating branches:

- Arrange code to make basic blocks contiguous.
- Unroll loops, as discussed in Section 3.4.1.6, "Loop Unrolling."
- Use the CMOV instruction.
- Use the SETCC instruction.

The following rules apply to branch elimination:

Assembly/Compiler Coding Rule 1. (MH impact, M generality) *Arrange code to make basic blocks contiguous and eliminate unnecessary branches.*

Assembly/Compiler Coding Rule 2. (M impact, ML generality) *Use the SETCC and CMOV instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to SETCC or CMOV trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel 64 and IA-32 processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.*

Consider a line of C code that has a condition dependent upon one of the constants:

```
X = (A < B) ? CONST1 : CONST2;
```

This code conditionally compares two values, A and B. If the condition is true, X is set to CONST1; otherwise it is set to CONST2. An assembly code sequence equivalent to the above C code can contain branches that are not predictable if there are no correlation in the two values.

Example 3-1 shows the assembly code with unpredictable branches. The unpredictable branches can be removed with the use of the SETCC instruction. Example 3-2 shows optimized code that has no branches.

Example 3-1. Assembly Code with an Unpredictable Branch

```
    cmp a, b                    ; Condition
    jbe L30                     ; Conditional branch
    mov ebx const1              ; ebx holds X
    jmp L31                     ; Unconditional branch
L30:
    mov ebx, const2
L31:
```

Example 3-2. Code Optimization to Eliminate Branches

```
xor   ebx, ebx      ; Clear ebx (X in the C code)
cmp   A, B
setge bl            ; When ebx = 0 or 1
                    ; OR the complement condition
sub   ebx, 1        ; ebx=11...11 or 00...00
and   ebx, CONST3; CONST3 = CONST1-CONST2
add   ebx, CONST2; ebx=CONST1 or CONST2
```

The optimized code in Example 3-2 sets EBX to zero, then compares A and B. If A is greater than or equal to B, EBX is set to one. Then EBX is decreased and AND'd with the difference of the constant values. This sets EBX to either zero or the difference of the values. By adding CONST2 back to EBX, the correct value is written to EBX. When CONST2 is equal to zero, the last instruction can be deleted.

Another way to remove branches is to use the CMOV and FCMOV instructions. Example 3-3 shows how to change a TEST and branch instruction sequence using CMOV to eliminate a branch. If the TEST sets the equal flag, the value in EBX will be moved to EAX. This branch is data-dependent, and is representative of an unpredictable branch.

Example 3-3. Eliminating Branch with CMOV Instruction

```
    test ecx, ecx
    jne  1H
    mov  eax, ebx


1H:
; To optimize code, combine jne and mov into one cmovcc instruction that checks the equal flag
    test     ecx, ecx              ; Test the flags
    cmoveq   eax, ebx              ; If the equal flag is set, move
                                   ; ebx to eax- the 1H: tag no longer needed
```

An extension to this concept can be seen in the AVX-512 masked operations, as well as in some instructions such as VPCMP which can be used to eliminate data dependent branches; see Section 18.4.

3.4.1.2 Static Prediction

Branches that do not have a history in the BTB (see Section 3.4.1, "Branch Prediction Optimization") are predicted using a static prediction algorithm:

- Predict forward conditional branches to be NOT taken.
- Predict backward conditional branches to be taken.
- Predict indirect branches to be NOT taken.

The following rule applies to static prediction:

Assembly/Compiler Coding Rule 3. (M impact, H generality) *Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target.*

Example 3-4 illustrates the static branch prediction algorithm. The body of an IF-THEN conditional is predicted.

Example 3-4. Static Branch Prediction Algorithm

```
//Forward condition branches not taken (fall through)
    IF<condition> {....
    ↓
    }

IF<condition> {...
    ↓
    }

//Backward conditional branches are taken
    LOOP {...
    ↑ –– }<condition>

//Unconditional branches taken
     JMP
    ------→
```

Example 3-5 and Example 3-6 provide basic rules for a static prediction algorithm. In Example 3-5, the backward branch (JC BEGIN) is not in the BTB the first time through; therefore, the BTB does not issue a prediction. The static predictor, however, will predict the branch to be taken, so a misprediction will not occur.

Example 3-5. Static Taken Prediction

```
Begin:  mov     eax,  mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
```

The first branch instruction (JC BEGIN) in Example 3-6 is a conditional forward branch. It is not in the BTB the first time through, but the static predictor will predict the branch to fall through**.** The static prediction algorithm correctly predicts that the CALL CONVERT instruction will be taken, even before the branch has any branch history in the BTB.

Example 3-6. Static Not-Taken Prediction

```
        mov     eax, mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
        mov     eax, 0
Begin:  call    Convert
```

The Intel Core microarchitecture does not use the static prediction heuristic. However, to maintain consistency across Intel 64 and IA-32 processors, software should maintain the static prediction heuristic as the default.

3.4.1.3 Inlining, Calls, and Returns

The return address stack mechanism augments the static and dynamic predictors to optimize specifically for calls and returns. It holds 16 entries, which is large enough to cover the call depth of most programs. If there is a chain of more than 16 nested calls and more than 16 returns in rapid succession, performance may degrade.

To enable the use of the return stack mechanism, calls and returns must be matched in pairs. If this is done, the likelihood of exceeding the stack depth in a manner that will impact performance is very low.

The following rules apply to inlining, calls, and returns:

Assembly/Compiler Coding Rule 4. (MH impact, MH generality) *Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns.*

Calls and returns are expensive; use inlining for the following reasons:

- Parameter passing overhead can be eliminated.
- In a compiler, inlining a function exposes more opportunity for optimization.
- If the inlined routine contains branches, the additional context of the caller may improve branch prediction within the routine.
- A mispredicted branch can lead to performance penalties inside a small function that are larger than those that would occur if that function is inlined.

Assembly/Compiler Coding Rule 5. (MH impact, MH generality) *Selectively inline a function if doing so decreases code size or if the function is small and the call site is frequently executed.*

Assembly/Compiler Coding Rule 6. (ML impact, ML generality) *If there are more than 16 nested calls and returns in rapid succession; consider transforming the program with inline to reduce the call depth.*

Assembly/Compiler Coding Rule 7. (ML impact, ML generality) *Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred.*

Assembly/Compiler Coding Rule 8. (L impact, L generality) *If the last statement in a function is a call to another function, consider converting the call to a jump. This will save the call/return overhead as well as an entry in the return stack buffer.*

Assembly/Compiler Coding Rule 9. (M impact, L generality) *Do not put more than four branches in a 16-byte chunk.*

Assembly/Compiler Coding Rule 10. (M impact, L generality) *Do not put more than two end loop branches in a 16-byte chunk.*

3.4.1.4 Code Alignment

Careful arrangement of code can enhance cache and memory locality. Likely sequences of basic blocks should be laid out contiguously in memory. This may involve removing unlikely code, such as code to handle error conditions, from the sequence. See Section 3.7, "Prefetching," on optimizing the instruction prefetcher.

Assembly/Compiler Coding Rule 11. (M impact, H generality) *When executing code from the Decoded ICache, direct branches that are mostly taken should have all their instruction bytes in a 64B cache line and nearer the end of that cache line. Their targets should be at or near the beginning of a 64B cache line.*

When executing code from the legacy decode pipeline, direct branches that are mostly taken should have all their instruction bytes in a 16B aligned chunk of memory and nearer the end of that 16B aligned chunk. Their targets should be at or near the beginning of a 16B aligned chunk of memory.

Assembly/Compiler Coding Rule 12. (M impact, H generality) *If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, it should be placed on a different code page.*

3.4.1.5 Branch Type Selection

The default predicted target for indirect branches and calls is the fall-through path. Fall-through prediction is overridden if and when a hardware prediction is available for that branch. The predicted branch target from branch prediction hardware for an indirect branch is the previously executed branch target.

The default prediction to the fall-through path is only a significant issue if no branch prediction is available, due to poor code locality or pathological branch conflict problems. For indirect calls, predicting the fall-through path is usually not an issue, since execution will likely return to the instruction after the associated return.

Placing data immediately following an indirect branch can cause a performance problem. If the data consists of all zeros, it looks like a long stream of ADDs to memory destinations and this can cause resource conflicts and slow down branch recovery. Also, data immediately following indirect branches may appear as branches to the branch predication hardware, which can branch off to execute other data pages. This can lead to subsequent self-modifying code problems.

Assembly/Compiler Coding Rule 13. (M impact, L generality) *When indirect branches are present, try to put the most likely target of an indirect branch immediately following the indirect branch. Alternatively, if indirect branches are common but they cannot be predicted by branch prediction hardware, then follow the indirect branch with a UD2 instruction, which will stop the processor from decoding down the fall-through path.*

Indirect branches resulting from code constructs (such as switch statements, computed GOTOs or calls through pointers) can jump to an arbitrary number of locations. If the code sequence is such that the target destination of a branch goes to the same address most of the time, then the BTB will predict accurately most of the time. Since only one taken (non-fall-through) target can be stored in the BTB, indirect branches with multiple taken targets may have lower prediction rates.

The effective number of targets stored may be increased by introducing additional conditional branches. Adding a conditional branch to a target is fruitful if:

- The branch direction is correlated with the branch history leading up to that branch; that is, not just the last target, but how it got to this branch.
- The source/target pair is common enough to warrant using the extra branch prediction capacity. This may increase the number of overall branch mispredictions, while improving the misprediction of indirect branches. The profitability is lower if the number of mispredicting branches is very large.

User/Source Coding Rule 1. (M impact, L generality) *If an indirect branch has two or more common taken targets and at least one of those targets is correlated with branch history leading up to the branch, then convert the indirect branch to a tree where one or more indirect branches are preceded by conditional branches to those targets. Apply this "peeling" procedure to the common target of an indirect branch that correlates to branch history.*

The purpose of this rule is to reduce the total number of mispredictions by enhancing the predictability of branches (even at the expense of adding more branches). The added branches must be predictable for this to be worthwhile. One reason for such predictability is a strong correlation with preceding branch history. That is, the directions taken on preceding branches are a good indicator of the direction of the branch under consideration.

Example 3-7 shows a simple example of the correlation between a target of a preceding conditional branch and a target of an indirect branch.

Example 3-7. Indirect Branch With Two Favored Targets

```
function ()
{
int n = rand();          // random integer 0 to RAND_MAX
    if ( ! (n & 0x01) ) { // n will be 0 half the times
        n = 0;           // updates branch history to predict taken
    }
    // indirect branches with multiple taken targets
    // may have lower prediction rates

 switch (n) {
    case 0: handle_0(); break;    // common target, correlated with
                                  // branch history that is forward taken
    case 1: handle_1(); break;    // uncommon
    case 3: handle_3(); break;    // uncommon
    default: handle_other();      // common target
     }
}
```

Correlation can be difficult to determine analytically, for a compiler and for an assembly language programmer. It may be fruitful to evaluate performance with and without peeling to get the best performance from a coding effort.

An example of peeling out the most favored target of an indirect branch with correlated branch history is shown in Example 3-8.

Example 3-8. A Peeling Technique to Reduce Indirect Branch Misprediction

```
function ()
{
   int n = rand();                             // Random integer 0 to RAND_MAX
     if( ! (n & 0x01) ) THEN
          n = 0;                               // n will be 0 half the times
     if (!n) THEN
          handle_0();                          // Peel out the most common target
                                               // with correlated branch history

     {
       switch (n) {
          case 1: handle_1(); break;           // Uncommon
          case 3: handle_3(); break;           // Uncommon

          default: handle_other();             // Make the favored target in
                                               // the fall-through path
          }
     }
}
```

3.4.1.6 Loop Unrolling

Benefits of unrolling loops are:

- Unrolling amortizes the branch overhead, since it eliminates branches and some of the code to manage induction variables.
- Unrolling allows one to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if you have enough free registers to keep variables live as you stretch out the dependence chain to expose the critical path.
- Unrolling exposes the code to various other optimizations, such as removal of redundant loads, common subexpression elimination, and so on.

The potential costs of unrolling loops are:

- Unrolling loops whose bodies contain branches increases demand on BTB capacity. If the number of iterations of the unrolled loop is 16 or fewer, the branch predictor should be able to correctly predict branches in the loop body that alternate direction.

Assembly/Compiler Coding Rule 14. (H impact, M generality) *Unroll small loops until the overhead of the branch and induction variable accounts (generally) for less than 10% of the execution time of the loop.*

Assembly/Compiler Coding Rule 15. (M impact, M generality) *Unroll loops that are frequently executed and have a predictable number of iterations to reduce the number of iterations to 16 or fewer. Do this unless it increases code size so that the working set no longer fits in the instruction cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches).*

Example 3-9 shows how unrolling enables other optimizations.

Example 3-9. Loop Unrolling

```
Before unrolling:
        do i = 1, 100
                if ( i mod 2 == 0 ) then a( i ) = x
                        else a( i ) = y
        enddo
After unrolling
        do i = 1, 100, 2
                a( i ) = y
                a( i+1 ) = x
        enddo
```

In this example, the loop that executes 100 times assigns X to every even-numbered element and Y to every odd-numbered element. By unrolling the loop you can make assignments more efficiently, removing one branch in the loop body.

3.4.2 Fetch and Decode Optimization

Intel Core microarchitecture provides several mechanisms to increase front end throughput. Techniques to take advantage of some of these features are discussed below.

3.4.2.1 Optimizing for Microfusion

An Instruction that operates on a register and a memory operand decodes into more micro-ops than its corresponding register-register version. Replacing the equivalent work of the former instruction using the register-register version usually require a sequence of two instructions. The latter sequence is likely to result in reduced fetch bandwidth.

Assembly/Compiler Coding Rule 16. (ML impact, M generality) *For improving fetch/decode throughput, Give preference to memory flavor of an instruction over the register-only flavor of the same instruction, if such instruction can benefit from micro-fusion.*

The following examples are some of the types of micro-fusions that can be handled by all decoders:

- All stores to memory, including store immediate. Stores execute internally as two separate micro-ops: store-address and store-data.
- All “read-modify” (load+op) instructions between register and memory, for example:
```
ADDPS   XMM9, OWORD PTR [RSP+40]
FADD    DOUBLE PTR [RDI+RSI*8]
XOR     RAX, QWORD PTR [RBP+32]
```
- All instructions of the form “load and jump,” for example:
```
JMP     [RDI+200]
RET
```
- CMP and TEST with immediate operand and memory.

An Intel 64 instruction with RIP relative addressing is not micro-fused in the following cases:

- When an additional immediate is needed, for example:
  ```
  CMP     [RIP+400], 27
  MOV     [RIP+3000], 142
  ```
- When an RIP is needed for control flow purposes, for example:
  ```
  JMP     [RIP+5000000]
  ```

In these cases, Intel Core microarchitecture and Sandy Bridge microarchitecture provide a 2 micro-op flow from decoder 0, resulting in a slight loss of decode bandwidth since 2 micro-op flow must be steered to decoder 0 from the decoder with which it was aligned.

RIP addressing may be common in accessing global data. Since it will not benefit from micro-fusion, compiler may consider accessing global data with other means of memory addressing.

3.4.2.2 Optimizing for Macrofusion

Macrofusion merges two instructions to a single micro-op. Intel Core microarchitecture performs this hardware optimization under limited circumstances.

The first instruction of the macro-fused pair must be a CMP or TEST instruction. This instruction can be REG-REG, REG-IMM, or a micro-fused REG-MEM comparison. The second instruction (adjacent in the instruction stream) should be a conditional branch.

Since these pairs are common ingredient in basic iterative programming sequences, macrofusion improves performance even on un-recompiled binaries. All of the decoders can decode one macro-fused pair per cycle, with up to three other instructions, resulting in a peak decode bandwidth of 5 instructions per cycle.

Each macro-fused instruction executes with a single dispatch. This process reduces latency, which in this case shows up as a cycle removed from branch mispredict penalty. Software also gain all other fusion benefits: increased rename and retire bandwidth, more storage for instructions in-flight, and power savings from representing more work in fewer bits.

The following list details when you can use macrofusion:

- CMP or TEST can be fused when comparing:

  REG-REG. For example: CMP EAX,ECX; JZ label
  REG-IMM. For example: CMP EAX,0x80; JZ label
  REG-MEM. For example: CMP EAX,[ECX]; JZ label
  MEM-REG. For example: CMP [EAX],ECX; JZ label
- TEST can fused with all conditional jumps.
- CMP can be fused with only the following conditional jumps in Intel Core microarchitecture. These conditional jumps check carry flag (CF) or zero flag (ZF). jump. The list of macrofusion-capable conditional jumps are:

  JA or JNBE
  JAE or JNB or JNC
  JE or JZ
  JNA or JBE
  JNAE or JC or JB
  JNE or JNZ

CMP and TEST can not be fused when comparing MEM-IMM (e.g. CMP [EAX],0x80; JZ label). Macrofusion is not supported in 64-bit mode for Intel Core microarchitecture.

- Nehalem microarchitecture supports the following enhancements in macrofusion:
  — CMP can be fused with the following conditional jumps (that was not supported in Intel Core microarchitecture):
    - JL or JNGE
    - JGE or JNL

    - JLE or JNG
    - JG or JNLE

  — Macrofusion is supported in 64-bit mode.

- Enhanced macrofusion support in Sandy Bridge microarchitecture is summarized in Table 3-1 with additional information in Section E.2.2.1 and Example 3-14:

Table 3-1. Macro-Fusible Instructions in Sandy Bridge Microarchitecture

Instructions	TEST	AND	CMP	ADD	SUB	INC	DEC
JO/JNO	Y	Y	N	N	N	N	N
JC/JB/JAE/JNB	Y	Y	Y	Y	Y	N	N
JE/JZ/JNE/JNZ	Y	Y	Y	Y	Y	Y	Y
JNA/JBE/JA/JNBE	Y	Y	Y	Y	Y	N	N
JS/JNS/JP/JPE/JNP/JPO	Y	Y	N	N	N	N	N
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	Y	Y	Y	Y	Y	Y	Y

- Enhanced macrofusion support in Haswell microarchitecture is summarized in Table 3-2. Macrofusion is supported CMP/TEST/OP with reg-imm, reg-mem, and reg-reg addressing but not mem-imm addressing.

Table 3-2. Macro-Fusible Instructions in Haswell Microarchitecture

Opcode		JCC	ADD / SUB / CMP	INC / DEC	TEST / AND
70	0F 80	Jo	N	N	Y
71	0F 81	Jno	N	N	Y
72	0F 82	Jc / Jb	Y	N	Y
73	0F 83	Jae / Jnb	Y	N	Y
74	0F 84	Je / Jz	Y	Y	Y
75	0F 85	Jne / Jnz	Y	Y	Y
76	0F 86	Jna / Jbe	Y	N	Y
77	0F 87	Ja / Jnbe	Y	N	Y
78	0F 88	Js	N	N	Y
79	0F 89	Jns	N	N	Y
7A	0F 8A	Jp / Jpe	N	N	Y
7B	0F 8B	Jnp / Jpo	N	N	Y
7C	0F 8C	Jl / Jnge	Y	Y	Y
7D	0F 8D	Jge / Jnl	Y	Y	Y
7E	0F 8E	Jle / Jng	Y	Y	Y
7F	0F 8F	Jg / Jnle	Y	Y	Y

Assembly/Compiler Coding Rule 17. (M impact, ML generality) *Employ macrofusion where possible using instruction pairs that support macrofusion. Prefer TEST over CMP if possible. Use unsigned variables and unsigned jumps when possible. Try to logically verify that a variable is non-negative at the time of comparison. Avoid CMP or TEST of MEM-IMM flavor when possible. However, do not add other instructions to avoid using the MEM-IMM flavor.*

Example 3-10. Macrofusion, Unsigned Iteration Count

	Without Macrofusion	With Macrofusion

C code	for (int[1] i = 0; i < 1000; i++)
	    a++;
		for (unsigned int[2] i = 0; i < 1000; i++)
		    a++;

Disassembly	for (int i = 0; i < 1000; i++) mov dword ptr [i], 0 jmp First Loop: mov eax, dword ptr [i] add eax, 1 mov dword ptr [i], eax First: cmp dword ptr [i], 3E8H[3] jge End a++; mov eax, dword ptr [a] addqq eax,1 mov dword ptr [a], eax jmp Loop End:	for (unsigned int i = 0; i < 1000; i++) xor eax, eax mov dword ptr [i], eax jmp First Loop: mov eax, dword ptr [i] add eax, 1 mov dword ptr [i], eax First: cmp eax, 3E8H [4] jae End a++; mov eax, dword ptr [a] add eax, 1 mov dword ptr [a], eax jmp Loop End:

NOTES:

1. Signed iteration count inhibits macrofusion.
2. Unsigned iteration count is compatible with macrofusion.
3. CMP MEM-IMM, JGE inhibit macrofusion.
4. CMP REG-IMM, JAE permits macrofusion.

Example 3-11. Macrofusion, If Statement

	Without Macrofusion	With Macrofusion

C code	int[1] a = 7;
	if (a < 77)
	    a++;
	else
	    a--;
		unsigned int[2] a = 7;
		if (a < 77)
		    a++;
		else
		    a--;

Disassembly	int a = 7; mov dword ptr [a], 7 if (a < 77) cmp dword ptr [a], 4DH [3] jge Dec	unsigned int a = 7; mov dword ptr [a], 7 if (a < 77) mov eax, dword ptr [a] cmp eax, 4DH jae Dec

Example 3-11. Macrofusion, If Statement (Contd.)

	Without Macrofusion	With Macrofusion
	a++; mov eax, dword ptr [a] add eax, 1 mov dword ptr [a], eax else jmp End a--; Dec: mov eax, dword ptr [a] sub eax, 1 mov dword ptr [a], eax End::	a++; add eax,1 mov dword ptr [a], eax else jmp End a--; Dec: sub eax, 1 mov dword ptr [a], eax End::

NOTES:

1. Signed iteration count inhibits macrofusion.
2. Unsigned iteration count is compatible with macrofusion.
3. CMP MEM-IMM, JGE inhibit macrofusion.

Assembly/Compiler Coding Rule 18. (M impact, ML generality) *Software can enable macro fusion when it can be logically determined that a variable is non-negative at the time of comparison; use TEST appropriately to enable macrofusion when comparing a variable with 0.*

Example 3-12. Macrofusion, Signed Variable

Without Macrofusion	With Macrofusion
test ecx, ecx jle OutSideTheIF cmp ecx, 64H jge OutSideTheIF <IF BLOCK CODE> OutSideTheIF:	test ecx, ecx jle OutSideTheIF cmp ecx, 64H jae OutSideTheIF <IF BLOCK CODE> OutSideTheIF:

For either signed or unsigned variable 'a'; "CMP a,0" and "TEST a,a" produce the same result as far as the flags are concerned. Since TEST can be macro-fused more often, software can use "TEST a,a" to replace "CMP a,0" for the purpose of enabling macrofusion.

Example 3-13. Macrofusion, Signed Comparison

C Code	Without Macrofusion	With Macrofusion
if (a == 0)	cmp a, 0 jne lbl ... lbl:	test a, a jne lbl ... lbl:
if (a >= 0)	cmp a, 0 jl lbl; ... lbl:	test a, a jl lbl ... lbl:

Sandy Bridge microarchitecture enables more arithmetic and logic instructions to macro-fuse with conditional branches. In loops where the ALU ports are already congested, performing one of these macrofusions can relieve the pressure, as the macro-fused instruction consumes only port 5, instead of an ALU port plus port 5.

In Example 3-14, the "add/cmp/jnz" loop contains two ALU instructions that can be dispatched via either port 0, 1, 5. So there is higher probability of port 5 might bind to either ALU instruction causing JNZ to wait a cycle. The “sub/jnz” loop, the likelihood of ADD/SUB/JNZ can be dispatched in the same cycle is increased because only SUB is free to bind with either port 0, 1, 5.

Example 3-14. Additional Macrofusion Benefit in Sandy Bridge Microarchitecture

Add + cmp + jnz alternative	Loop control with sub + jnz

```
lea      rdx, buff
xor      rcx, rcx
xor      eax, eax
loop:
add      eax, [rdx + 4 * rcx]
add      rcx, 1
cmp      rcx, LEN
jnz      loop
```
	```
	lea      rdx, buff - 4
	xor      rcx, LEN
	xor      eax, eax
	loop:
	add      eax, [rdx + 4 * rcx]
	sub      rcx, 1
	jnz      loop
	```

3.4.2.3 Length-Changing Prefixes (LCP)

The length of an instruction can be up to 15 bytes in length. Some prefixes can dynamically change the length of an instruction that the decoder must recognize. Typically, the pre-decode unit will estimate the length of an instruction in the byte stream assuming the absence of LCP. When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle. Normal queuing throughout of the machine pipeline generally cannot hide LCP penalties.

The prefixes that can dynamically change the length of a instruction include:

- Operand size prefix (0x66).
- Address size prefix (0x67).

The instruction MOV DX, 01234h is subject to LCP stalls in processors based on Intel Core microarchitecture, and in Intel Core Duo and Intel Core Solo processors. Instructions that contain imm16 as part of their fixed encoding but do not require LCP to change the immediate size are not subject to LCP stalls. The REX prefix (4xh) in 64-bit mode can change the size of two classes of instruction, but does not cause an LCP penalty.

If the LCP stall happens in a tight loop, it can cause significant performance degradation. When decoding is not a bottleneck, as in floating-point heavy code, isolated LCP stalls usually do not cause performance degradation.

Assembly/Compiler Coding Rule 19. (MH impact, MH generality) *Favor generating code using imm8 or imm32 values instead of imm16 values.*

If imm16 is needed, load equivalent imm32 into a register and use the word value in the register instead.

Double LCP Stalls

Instructions that are subject to LCP stalls and cross a 16-byte fetch line boundary can cause the LCP stall to trigger twice. The following alignment situations can cause LCP stalls to trigger twice:

- An instruction is encoded with a MODR/M and SIB byte, and the fetch line boundary crossing is between the MODR/M and the SIB bytes.
- An instruction starts at offset 13 of a fetch line references a memory location using register and immediate byte offset addressing mode.

The first stall is for the 1st fetch line, and the 2nd stall is for the 2nd fetch line. A double LCP stall causes a decode penalty of 11 cycles.

The following examples cause LCP stall once, regardless of their fetch-line location of the first byte of the instruction:

```
ADD DX, 01234H
ADD word ptr [EDX], 01234H
ADD word ptr 012345678H[EDX], 01234H
ADD word ptr [012345678H], 01234H
```

The following instructions cause a double LCP stall when starting at offset 13 of a fetch line:

```
ADD word ptr [EDX+ESI], 01234H
ADD word ptr 012H[EDX], 01234H
ADD word ptr 012345678H[EDX+ESI], 01234H
```

To avoid double LCP stalls, do not use instructions subject to LCP stalls that use SIB byte encoding or addressing mode with byte displacement.

False LCP Stalls

False LCP stalls have the same characteristics as LCP stalls, but occur on instructions that do not have any imm16 value.

False LCP stalls occur when (a) instructions with LCP that are encoded using the F7 opcodes, and (b) are located at offset 14 of a fetch line. These instructions are: not, neg, div, idiv, mul, and imul. False LCP experiences delay because the instruction length decoder can not determine the length of the instruction before the next fetch line, which holds the exact opcode of the instruction in its MODR/M byte.

The following techniques can help avoid false LCP stalls:

- Upcast all short operations from the F7 group of instructions to long, using the full 32 bit version.
- Ensure that the F7 opcode never starts at offset 14 of a fetch line.

Assembly/Compiler Coding Rule 20. (M impact, ML generality) *Ensure instructions using 0xF7 opcode byte does not start at offset 14 of a fetch line; and avoid using these instruction to operate on 16-bit data, upcast short data to 32 bits.*

Example 3-15. Avoiding False LCP Delays with 0xF7 Group Instructions

A Sequence Causing Delay in the Decoder	Alternate Sequence to Avoid Delay
neg word ptr a	movsx eax, word ptr a neg eax mov word ptr a, AX

3.4.2.4 Optimizing the Loop Stream Detector (LSD)

The LSD detects loops that have many iterations and fit into the µop-queue. The µop-queue streams the loop until a branch miss-prediction inevitably ends it.

LSD improves fetch bandwidth. In single thread mode, it saves power by allowing the front-end to sleep. In multi-thread mode, front-resource can better serve the other thread.

Loops qualify for LSD replay if all the following conditions are met:

- Loop body size up to 60 µops, with up to 15 taken branches, and up to 15 64-byte fetch lines.
- No CALL or RET.
- No mismatched stack operations (e.g., more PUSH than POP).
- More than ~20 iterations.

Many calculation-intensive loops, searches, and software string moves match these characteristics. These loops exceed the BPU prediction capacity and always terminate in a branch misprediction.

Assembly/Compiler Coding Rule 21. (MH impact, MH generality) *Break up a loop body with a long sequence of instructions into loops of shorter instruction blocks of no more than the size of the LSD.*

Allocation bandwidth in Ice Lake Client microarchitecture increased from 4 µops per cycle to 5 µops per cycle.

Assume a loop that qualifies for LSD has 23 µops in the loop body. The hardware unrolls the loop such that it still fits into the µop-queue, in this case twice. The loop in the µop-queue thus takes 46 µops.

The loop is sent to allocation 5 µops per cycle. After 45 out of the 46 µops are sent, in the next cycle only a single µop is sent, which means that in that cycle, 4 of the allocation slots are wasted. This pattern repeats itself, until the loop is exited by a misprediction. Hardware loop unrolling minimizes the number of wasted slots during LSD.

3.4.2.5 Optimization for Decoded ICache

The decoded ICache is a new feature in Sandy Bridge microarchitecture. Running the code from the Decoded ICache has two advantages:

- Higher bandwidth of micro-ops feeding the out-of-order engine.
- The front end does not need to decode the code that is in the Decoded ICache; this saves power.

There is overhead in switching between the Decoded ICache and the legacy decode pipeline. If your code switches frequently between the front end and the Decoded ICache, the penalty may be higher than running only from the legacy pipeline.

To ensure "hot" code is feeding from the decoded ICache:

- Make sure each hot code block is less than about 750 instructions. Specifically, do not unroll to more than 750 instructions in a loop. This should enable Decoded ICache residency even when hyper-threading is enabled.
- For applications with very large blocks of calculations inside a loop, consider loop-fission: split the loop into multiple loops that fit in the Decoded ICache, rather than a single loop that overflows.
- If an application can be sure to run with only one thread per core, it can increase hot code block size to about 1500 instructions.

Dense Read-Modify-Write Code

The Decoded ICache can hold only up to 18 micro-ops per each 32 byte aligned memory chunk. Therefore, code with a high concentration of instructions that are encoded in a small number of bytes, yet have many micro-ops, may overflow the 18 micro-op limitation and not enter the Decoded ICache. Read-modify-write (RMW) instructions are a good example of such instructions.

RMW instructions accept one memory source operand, one register source operand, and use the source memory operand as the destination. The same functionality can be achieved by two or three instructions: the first reads the memory source operand, the second performs the operation with the second register source operand, and the last writes the result back to memory. These instructions usually result in the same number of micro-ops but use more bytes to encode the same functionality.

One case where RMW instructions may be used extensively is when the compiler optimizes aggressively for code size.

Here are some possible solutions to fit the hot code in the Decoded ICache:

- Replace RMW instructions with two or three instructions that have the same functionality. For example, "adc [rdi], rcx" is only three bytes long; the equivalent sequence "adc rax, [rdi]" + "mov [rdi], rax" has a footprint of six bytes.
- Align the code so that the dense part is broken down among two different 32-byte chunks. This solution is useful when using a tool that aligns code automatically, and is indifferent to code changes.
- Spread the code by adding multiple byte NOPs in the loop. Note that this solution adds micro-ops for execution.

Align Unconditional Branches for Decoded ICache

For code entering the Decoded ICache, each unconditional branch is the last micro-op occupying a Decoded ICache Way. Therefore, only three unconditional branches per a 32 byte aligned chunk can enter the Decoded ICache.

Unconditional branches are frequent in jump tables and switch declarations. Below are examples for these constructs, and methods for writing them so that they fit in the Decoded ICache.

Compilers create jump tables for C++ virtual class methods or DLL dispatch tables. Each unconditional branch consumes five bytes; therefore up to seven of them can be associated with a 32-byte chunk. Thus jump tables may not fit in the Decoded ICache if the unconditional branches are too dense in each 32Byte-aligned chunk. This can cause performance degradation for code executing before and after the branch table.

The solution is to add multi-byte NOP instructions among the branches in the branch table. This may increases code size and should be used cautiously. However, these NOPs are not executed and therefore have no penalty in later pipe stages.

Switch-Case constructs represents a similar situation. Each evaluation of a case condition results in an unconditional branch. The same solution of using multi-byte NOP can apply for every three consecutive unconditional branches that fits inside an aligned 32-byte chunk.

Two Branches in a Decoded ICache Way

The Decoded ICache can hold up to two branches in a way. Dense branches in a 32 byte aligned chunk, or their ordering with other instructions may prohibit all the micro-ops of the instructions in the chunk from entering the Decoded ICache. This does not happen often. When it does happen, you can space the code with NOP instructions where appropriate. Make sure that these NOP instructions are not part of hot code.

Assembly/Compiler Coding Rule 22. (M impact, M generality) *Avoid putting explicit references to* ESP *in a sequence of stack operations (*POP, PUSH, CALL, RET*).*

3.4.2.6 Other Decoding Guidelines

Assembly/Compiler Coding Rule 23. (ML impact, L generality) *Use simple instructions that are less than eight bytes in length.*

Assembly/Compiler Coding Rule 24. (M impact, MH generality) *Avoid using prefixes to change the size of immediate and displacement.*

Long instructions (more than seven bytes) may limit the number of decoded instructions per cycle. Each prefix adds one byte to the length of instruction, possibly limiting the decoder's throughput. In addition, multiple prefixes can only be decoded by the first decoder. These prefixes also incur a delay when decoded. If multiple prefixes or a prefix that changes the size of an immediate or displacement cannot be avoided, schedule them behind instructions that stall the pipe for some other reason.

3.5 OPTIMIZING THE EXECUTION CORE

The superscalar, out-of-order execution core(s) in recent generations of microarchitectures contain multiple execution hardware resources that can execute multiple micro-ops in parallel. These resources generally ensure that micro-ops execute efficiently and proceed with fixed latencies. General guidelines to make use of the available parallelism are:

- Follow the rules (see Section 3.4) to maximize useful decode bandwidth and front end throughput. These rules include favoring single micro-op instructions and taking advantage of micro-fusion, Stack pointer tracker and macrofusion.
- Maximize rename bandwidth. Guidelines are discussed in this section and include properly dealing with partial registers, ROB read ports and instructions which causes side-effects on flags.
- Scheduling recommendations on sequences of instructions so that multiple dependency chains are alive in the reservation station (RS) simultaneously, thus ensuring that your code utilizes maximum parallelism.

- Avoid hazards, minimize delays that may occur in the execution core, allowing the dispatched micro-ops to make progress and be ready for retirement quickly.

3.5.1 Instruction Selection

Some execution units are not pipelined, this means that micro-ops cannot be dispatched in consecutive cycles and the throughput is less than one per cycle.

It is generally a good starting point to select instructions by considering the number of micro-ops associated with each instruction, favoring in the order of: single micro-op instructions, simple instruction with less than 4 micro-ops, and last instruction requiring microsequencer ROM (micro-ops which are executed out of the microsequencer involve extra overhead).

Assembly/Compiler Coding Rule 25. (M impact, H generality) *Favor single-micro-operation instructions. Also favor instruction with shorter latencies.*

A compiler may be already doing a good job on instruction selection. If so, user intervention usually is not necessary.

Assembly/Compiler Coding Rule 26. (M impact, L generality) *Avoid prefixes, especially multiple non-0F-prefixed opcodes.*

Assembly/Compiler Coding Rule 27. (M impact, L generality) *Do not use many segment registers.*

Assembly/Compiler Coding Rule 28. (M impact, M generality) *Avoid using complex instructions (for example, enter, leave, or loop) that have more than four µops and require multiple cycles to decode. Use sequences of simple instructions instead.*

Assembly/Compiler Coding Rule 29. (MH impact, M generality) *Use push/pop to manage stack space and address adjustments between function calls/returns instead of enter/leave. Using enter instruction with non-zero immediates can experience significant delays in the pipeline in addition to misprediction.*

Theoretically, arranging instructions sequence to match the 4-1-1-1 template applies to processors based on Intel Core microarchitecture. However, with macrofusion and micro-fusion capabilities in the front end, attempts to schedule instruction sequences using the 4-1-1-1 template will likely provide diminishing returns.

Instead, software should follow these additional decoder guidelines:

- If you need to use multiple micro-op, non-microsequenced instructions, try to separate by a few single micro-op instructions. The following instructions are examples of multiple micro-op instruction not requiring micro-sequencer:

```
ADC/SBB
CMOVcc
Read-modify-write instructions
```

- If a series of multiple micro-op instructions cannot be separated, try breaking the series into a different equivalent instruction sequence. For example, a series of read-modify-write instructions may go faster if sequenced as a series of read-modify + store instructions. This strategy could improve performance even if the new code sequence is larger than the original one.

3.5.1.1 Integer Divide

Typically, an integer divide is preceded by a CWD or CDQ instruction. Depending on the operand size, divide instructions use DX:AX or EDX:EAX for the dividend. The CWD or CDQ instructions sign-extend AX or EAX into DX or EDX, respectively. These instructions have denser encoding than a shift and move would be, but they generate the same number of micro-ops. If AX or EAX is known to be positive, replace these instructions with:

```
xor dx, dx
```

or

```
xor edx, edx
```

Modern compilers typically can transform high-level language expression involving integer division where the divisor is a known integer constant at compile time into a faster sequence using IMUL instruction instead. Thus programmers should minimize integer division expression with divisor whose value can not be known at compile time.

Alternately, if certain known divisor value are favored over other unknown ranges, software may consider isolating the few favored, known divisor value into constant-divisor expressions.

Section 13.2.4 describes more detail of using MUL/IMUL to replace integer divisions.

3.5.1.2 Using LEA

In Sandy Bridge microarchitecture, there are two significant changes to the performance characteristics of LEA instruction:

- LEA can be dispatched via port 1 and 5 in most cases, doubling the throughput over prior generations. However this apply only to LEA instructions with one or two source operands.

Example 3-16. Independent Two-Operand LEA Example

```
    mov     edx, N
    mov     eax, X
    mov     ecx, Y


loop:
    lea     ecx, [ecx + ecx]        // ecx = ecx*2
    lea     eax, [eax + eax *4]     // eax = eax*5
    and     ecx, 0xff
    and     eax, 0xff
    dec     edx
    jg      loop
```

- For LEA instructions with three source operands and some specific situations, instruction latency has increased to 3 cycles, and must dispatch via port 1:
  - — LEA that has all three source operands: base, index, and offset.
  - — LEA that uses base and index registers where the base is EBP, RBP, or R13.
  - — LEA that uses RIP relative addressing mode.
  - — LEA that uses 16-bit addressing mode.

Example 3-17. Alternative to Three-Operand LEA

3 operand LEA is slower	Two-operand LEA alternative	Alternative 2
#define K 1	#define K 1	#define K 1
uint32 an = 0;	uint32 an = 0;	uint32 an = 0;
uint32 N= mi_N;	uint32 N= mi_N;	uint32 N= mi_N;
mov ecx, N	mov ecx, N	mov ecx, N
xor esi, esi;	xor esi, esi;	xor esi, esi;
xor edx, edx;	xor edx, edx;	mov edx, K;
cmp ecx, 2;	cmp ecx, 2;	cmp ecx, 2;
jb finished;	jb finished;	jb finished;
dec ecx;	dec ecx;	mov eax, 2
		dec ecx;
loop1:	loop1:	loop1:
mov edi, esi;	mov edi, esi;	mov edi, esi;
lea esi, [K+esi+edx];	lea esi, [K+edx];	lea esi, [esi+edx];
and esi, 0xFF;	lea esi, [esi+edx];	and esi, 0xFF;
mov edx, edi;	and esi, 0xFF;	lea edx, [edi +K];
dec ecx;	mov edx, edi;	dec ecx;
jnz loop1;	dec ecx;	jnz loop1;
finished:	jnz loop1;	finished:
mov [an] ,esi;	finished:	mov [an] ,esi;
	mov [an] ,esi;	

The LEA instruction or a sequence of LEA, ADD, SUB and SHIFT instructions can replace constant multiply instructions. The LEA instruction can also be used as a multiple operand addition instruction, for example:

```
LEA ECX, [EAX + EBX*4 + A]
```

Using LEA in this way may avoid register usage by not tying up registers for operands of arithmetic instructions. This use may also save code space.

If the LEA instruction uses a shift by a constant amount then the latency of the sequence of µops is shorter if adds are used instead of a shift, and the LEA instruction may be replaced with an appropriate sequence of µops. This, however, increases the total number of µops, leading to a trade-off.

Assembly/Compiler Coding Rule 30. (ML impact, L generality) *If an LEA instruction using the scaled index is on the critical path, a sequence with ADDs may be better.*

3.5.1.3 ADC and SBB in Sandy Bridge Microarchitecture

The throughput of ADC and SBB in Sandy Bridge microarchitecture is 1 cycle, compared to 1.5-2 cycles in the prior generation. These two instructions are useful in numeric handling of integer data types that are wider than the maximum width of native hardware.

Example 3-18. Examples of 512-bit Additions

```
//Add 64-bit to 512 Number
     lea          rsi, gLongCounter
     lea          rdi, gStepValue
     mov          rax, [rdi]
     xor          rcx, rcx
oop_start:
     mov          r10, [rsi+rcx]
     add          r10, rax
     mov          [rsi+rcx], r10

     mov          r10, [rsi+rcx+8]
     adc          r10, 0
     mov          [rsi+rcx+8], r10


l    mov          r10, [rsi+rcx+16]
     adc          r10, 0
     mov          [rsi+rcx+16], r10
     mov          r10, [rsi+rcx+24]
     adc          r10, 0
     mov          [rsi+rcx+24], r10

     mov          r10, [rsi+rcx+32]
     adc          r10, 0
     mov          [rsi+rcx+32], r10

     mov r10, [rsi+rcx+40]
     adc r10, 0
     mov [rsi+rcx+40], r10



 mov r10, [rsi+rcx+48]
  adc r10, 0
  mov [rsi+rcx+48], r10

  mov r10, [rsi+rcx+56]
  adc r10, 0
  mov [rsi+rcx+56], r10
  add rcx, 64
  cmp rcx, SIZE
  jnz loop_start
```

```
// 512-bit Addition
loop1:
     mov          rax, [StepValue]
     add          rax, [LongCounter]
     mov          LongCounter, rax
     mov          rax, [StepValue+8]
     adc          rax, [LongCounter+8]
     mov          LongCounter+8, rax
     mov          rax, [StepValue+16]
     adc          rax, [LongCounter+16]




     mov          LongCounter+16, rax
     mov          rax, [StepValue+24]
     adc          rax, [LongCounter+24]

     mov          LongCounter+24, rax
     mov          rax, [StepValue+32]
     adc          rax, [LongCounter+32]

     mov          LongCounter+32, rax
     mov          rax, [StepValue+40]
     adc          rax, [LongCounter+40]

     mov          LongCounter+40, rax
     mov          rax, [StepValue+48]
     adc          rax, [LongCounter+48]


     mov          LongCounter+48, rax
     mov          rax, [StepValue+56]
     adc          rax, [LongCounter+56]

     mov          LongCounter+56, rax
     dec          rcx
     jnz          loop1
```

3.5.1.4 Bitwise Rotation

Bitwise rotation can choose between rotate with count specified in the CL register, an immediate constant and by 1 bit. Generally, The rotate by immediate and rotate by register instructions are slower than rotate by 1 bit. The rotate by 1 instruction has the same latency as a shift.

Assembly/Compiler Coding Rule 31. (ML impact, L generality) *Avoid ROTATE by register or ROTATE by immediate instructions. If possible, replace with a ROTATE by 1 instruction.*

In Sandy Bridge microarchitecture, ROL/ROR by immediate has 1-cycle throughput, SHLD/SHRD using the same register as source and destination by an immediate constant has 1-cycle latency with 0.5 cycle throughput. The "ROL/ROR reg, imm8" instruction has two micro-ops with the latency of 1-cycle for the rotate register result and 2-cycles for the flags, if used.

In Ivy Bridge microarchitecture, The "ROL/ROR reg, imm8" instruction with immediate greater than 1, is one micro-op with one-cycle latency when the overflow flag result is used. When the immediate is one, dependency on the overflow flag result of ROL/ROR by a subsequent instruction will see the ROL/ROR instruction with two-cycle latency.

3.5.1.5 Variable Bit Count Rotation and Shift

In Sandy Bridge microarchitecture, The "ROL/ROR/SHL/SHR reg, cl" instruction has three micro-ops. When the flag result is not needed, one of these micro-ops may be discarded, providing better performance in many common usages. When these instructions update partial flag results that are subsequently used, the full three micro-ops flow must go through the execution and retirement pipeline, experiencing slower performance. In Ivy Bridge microarchitecture, executing the full three micro-ops flow to use the updated partial flag result has additional delay. Consider the looped sequence below:

```
loop:
        shl eax, cl
        add ebx, eax
        dec edx ; DEC does not update carry, causing SHL to execute slower three micro-ops flow
        jnz loop
```

The DEC instruction does not modify the carry flag. Consequently, the SHL EAX, CL instruction needs to execute the three micro-ops flow in subsequent iterations. The SUB instruction will update all flags. So replacing DEC with SUB will allow SHL EAX, CL to execute the two micro-ops flow.

3.5.1.6 Address Calculations

For computing addresses, use the addressing modes rather than general-purpose computations. Internally, memory reference instructions can have four operands:

- Relocatable load-time constant.
- Immediate constant.
- Base register.
- Scaled index register.

Note that the latency and throughput of LEA with more than two operands are slower in Sandy Bridge microarchitecture (see Section 3.5.1.2, "Using LEA"). Addressing modes that uses both base and index registers will consume more read port resource in the execution engine and may experience more stalls due to availability of read port resources. Software should take care by selecting the speedy version of address calculation.

In the segmented model, a segment register may constitute an additional operand in the linear address calculation. In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

3.5.1.7 Clearing Registers and Dependency Breaking Idioms

Code sequences that modifies partial register can experience some delay in its dependency chain, but can be avoided by using dependency breaking idioms.

In processors based on Intel Core microarchitecture, a number of instructions can help clear execution dependency when software uses these instruction to clear register content to zero. The instructions include:

```
XOR REG, REG
SUB REG, REG
XORPS/PD XMMREG, XMMREG
PXOR XMMREG, XMMREG
SUBPS/PD XMMREG, XMMREG
PSUBB/W/D/Q XMMREG, XMMREG
```

In processors based on Sandy Bridge microarchitecture, the instruction listed above plus equivalent AVX counter parts are also zero idioms that can be used to break dependency chains. Furthermore, they do not consume an issue port or an execution unit. So using zero idioms are preferable than moving 0's into the register. The AVX equivalent zero idioms are:

```
VXORPS/PD XMMREG, XMMREG
VXORPS/PD YMMREG, YMMREG
VPXOR XMMREG, XMMREG
VSUBPS/PD XMMREG, XMMREG
VSUBPS/PD YMMREG, YMMREG
VPSUBB/W/D/Q XMMREG, XMMREG
```

Microarchitectures that support Intel AVX-512 have the equivalent of zero idioms for the 512-bit registers using the unmasked versions of the instructions:

```
VXORPS/PD ZMMREG, ZMMREG
VPXOR ZMMREG, ZMMREG
VSUBPS/PD ZMMREG, ZMMREG
VPSUBB/W/D/Q ZMMREG, ZMMREG
```

The XOR and SUB instructions can be used to clear execution dependencies on the zero evaluation of the destination register.

Assembly/Compiler Coding Rule 32. (M impact, ML generality) *Use dependency-breaking-idiom instructions to set a register to* 0*, or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move* 0 *into the register instead. This requires more code space than using* XOR *and* SUB*, but avoids setting the condition codes.*

Example 3-19 of using pxor to break dependency idiom on a XMM register when performing negation on the elements of an array.

```
int a[4096], b[4096], c[4096];
For ( int i = 0; i < 4096; i++ )
        C[i] = - ( a[i] + b[i] );
```

Example 3-19. Clearing Register to Break Dependency While Negating Array Elements

Negation (-x = (x XOR (-1)) - (-1) without breaking dependency	Negation (-x = 0 -x) using PXOR reg, reg breaks dependency
Lea eax, a lea ecx, b lea edi, c xor edx, edx movdqa xmm7, allone lp:	lea eax, a lea ecx, b lea edi, c xor edx, edx lp:
movdqa xmm0, [eax + edx] paddd xmm0, [ecx + edx] pxor xmm0, xmm7 psubd xmm0, xmm7 movdqa [edi + edx], xmm0 add edx, 16 cmp edx, 4096 jl lp	movdqa xmm0, [eax + edx] paddd xmm0, [ecx + edx] pxor xmm7, xmm7 psubd xmm7, xmm0 movdqa [edi + edx], xmm7 add edx,16 cmp edx, 4096 jl lp

Assembly/Compiler Coding Rule 33. (M impact, MH generality) *Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using MOVZX.*

Sometimes sign-extended semantics can be maintained by zero-extending operands. For example, the C code in the following statements does not need sign extension, nor does it need prefixes for operand size overrides:

```
static short INT a, b;
IF (a == b) {
  . . .
}
```

Code for comparing these 16-bit operands might be:

```
MOVZW  EAX, [a]
MOVZW  EBX, [b]
CMP    EAX, EBX
```

These circumstances tend to be common. However, the technique will not work if the compare is for greater than, less than, greater than or equal, and so on, or if the values in eax or ebx are to be used in another operation where sign extension is required.

Assembly/Compiler Coding Rule 34. (M impact, M generality) *Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension.*

The trace cache can be packed more tightly when instructions with operands that can only be represented as 32 bits are not adjacent.

Assembly/Compiler Coding Rule 35. (ML impact, L generality) *Avoid placing instructions that use 32-bit immediates which cannot be encoded as sign-extended 16-bit immediates near each other. Try to schedule µops that have no immediate immediately before or after µops with 32-bit immediates.*

3.5.1.8 Compares

Use TEST when comparing a value in a register with zero. TEST essentially ANDs operands together without writing to a destination register. TEST is preferred over AND because AND produces an extra result register. TEST is better than CMP ..., 0 because the instruction size is smaller.

Use TEST when comparing the result of a logical AND with an immediate constant for equality or inequality if the register is EAX for cases such as:

```
IF (AVAR & 8) { }
```

The TEST instruction can also be used to detect rollover of modulo of a power of 2. For example, the C code:

```
IF ( (AVAR % 16) == 0 ) { }
```

can be implemented using:

```
TEST    EAX, 0x0F
JNZ     AfterIf
```

Using the TEST instruction between the instruction that may modify part of the flag register and the instruction that uses the flag register can also help prevent partial flag register stall.

Assembly/Compiler Coding Rule 36. (ML impact, M generality) *Use the* TEST *instruction instead of* AND *when the result of the logical AND is not used. This saves µops in execution. Use a TEST of a register with itself instead of a CMP of the register to zero, this saves the need to encode the zero and saves encoding space. Avoid comparing a constant to a memory operand. It is preferable to load the memory operand and compare the constant to a register.*

Often a produced value must be compared with zero, and then used in a branch. Because most Intel architecture instructions set the condition codes as part of their execution, the compare instruction may be eliminated. Thus the operation can be tested directly by a JCC instruction. The notable exceptions are MOV and LEA. In these cases, use TEST.

Assembly/Compiler Coding Rule 37. (ML impact, M generality) *Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a* TEST *instruction instead of a compare. Be certain that any code transformations made do not introduce problems with overflow.*

3.5.1.9 Using NOPs

Code generators generate a no-operation (NOP) to align instructions. Examples of NOPs of different lengths in 32-bit mode are shown in Table 3-3.

Table 3-3. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

These are all true NOPs, having no effect on the state of the machine except to advance the EIP. Because NOPs require hardware resources to decode and execute, use the fewest number to achieve the desired padding.

The one byte NOP:[XCHG EAX,EAX] has special hardware support. Although it still consumes a µop and its accompanying resources, the dependence upon the old value of EAX is removed. This µop can be executed at the earliest possible opportunity, reducing the number of outstanding instructions, and is the lowest cost NOP.

The other NOPs have no special hardware support. Their input and output registers are interpreted by the hardware. Therefore, a code generator should arrange to use the register containing the oldest value as input, so that the NOP will dispatch and release RS resources at the earliest possible opportunity.

Try to observe the following NOP generation priority:

- Select the smallest number of NOPs and pseudo-NOPs to provide the desired padding.
- Select NOPs that are least likely to execute on slower execution unit clusters.
- Select the register arguments of NOPs to reduce dependencies.

3.5.1.10 Mixing SIMD Data Types

Previous microarchitectures (before Intel Core microarchitecture) do not have explicit restrictions on mixing integer and floating-point (FP) operations on XMM registers. For Intel Core microarchitecture, mixing integer and floating-point operations on the content of an XMM register can degrade performance. Software should avoid mixed-use of integer/FP operation on XMM registers. Specifically:

- Use SIMD integer operations to feed SIMD integer operations. Use PXOR for idiom.
- Use SIMD floating-point operations to feed SIMD floating-point operations. Use XORPS for idiom.
- When floating-point operations are bitwise equivalent, use PS data type instead of PD data type. MOVAPS and MOVAPD do the same thing, but MOVAPS takes one less byte to encode the instruction.

3.5.1.11 Spill Scheduling

The spill scheduling algorithm used by a code generator will be impacted by the memory subsystem. A spill scheduling algorithm is an algorithm that selects what values to spill to memory when there are too many live values to fit in registers. Consider the code in Example 3-20, where it is necessary to spill either A, B, or C.

Example 3-20. Spill Scheduling Code

```
LOOP
    C := …
    B := …
    A := A + …
```

For modern microarchitectures, using dependence depth information in spill scheduling is even more important than in previous processors. The loop-carried dependence in A makes it especially important that A not be spilled. Not only would a store/load be placed in the dependence chain, but there would also be a data-not-ready stall of the load, costing further cycles.

Assembly/Compiler Coding Rule 38. (H impact, MH generality) *For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies.*

A possibly counter-intuitive result is that in such a situation it is better to put loop invariants in memory than in registers, since loop invariants never have a load blocked by store data that is not ready.

3.5.1.12 Zero-Latency MOV Instructions

In processors based on Ivy Bridge microarchitecture, a subset of register-to-register move operations are executed in the front end (similar to zero-idioms, see Section 3.5.1.7). This conserves scheduling/execution resources in the out-of-order engine. Most forms of register-to-register MOV instructions can benefit from zero-latency MOV. Example 3-21 list the details of those forms that qualify and a small set that do not.

Example 3-21. Zero-Latency MOV Instructions

MOV instructions latency that can be eliminated	MOV instructions latency that cannot be eliminated
MOV reg32, reg32 MOV reg64, reg64 MOVUPD/MOVAPD xmm, xmm MOVUPD/MOVAPD ymm, ymm MOVUPS?MOVAPS xmm, xmm MOVUPS/MOVAPS ymm, ymm MOVDQA/MOVDQU xmm, xmm MOVDQA/MOVDQU ymm, ymm MOVDQA/MOVDQU zmm, zmm MOVZX reg32, reg8 (if not AH/BH/CH/DH) MOVZX reg64, reg8 (if not AH/BH/CH/DH)	MOV reg8, reg8 MOV reg16, reg16 MOVZX reg32, reg8 (if AH/BH/CH/DH) MOVZX reg64, reg8 (if AH/BH/CH/DH) MOVSX

Example 3-22 shows how to process 8-bit integers using MOVZX to take advantage of zero-latency MOV enhancement. Consider

X = (X * 3^N) MOD 256;

Y = (Y * 3^N) MOD 256;

When "MOD 256" is implemented using the "AND 0xff" technique, its latency is exposed in the result-dependency chain. Using a form of MOVZX on a truncated byte input, it can take advantage of zero-latency MOV enhancement and gain about 45% in speed.

Example 3-22. Byte-Granular Data Computation Technique

Use AND Reg32, 0xff	Use MOVZX
mov rsi, N mov rax, X mov rcx, Y loop: lea rcx, [rcx+rcx*2] lea rax, [rax+rax*4] and rcx, 0xff and rax, 0xff lea rcx, [rcx+rcx*2] lea rax, [rax+rax*4] and rcx, 0xff and rax, 0xff sub rsi, 2 jg loop	mov rsi, N mov rax, X mov rcx, Y loop: lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rdx, [rax+rax*4] movzx, rax, dl llea rdx, [rax+rax*4] movzx, rax, dl sub rsi, 2 jg loop

The effectiveness of coding a dense sequence of instructions to rely on a zero-latency MOV instruction must also consider internal resource constraints in the microarchitecture.

Example 3-23. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions

Needing more internal resource for zero-latency MOVs	Needing less internal resource for zero-latency MOVs
mov rsi, N mov rax, X mov rcx, Y	mov rsi, N mov rax, X mov rcx, Y
loop: lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rdx, [rax+rax*4] movzx, rax, dl lea rbx, [rcx+rcx*2] movzx, rcx, bl llea rdx, [rax+rax*4] movzx, rax, dl sub rsi, 2 jg loop	loop: lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rdx, [rax+rax*4] movzx, rax, dl llea rdx, [rax+rax*4] movzx, rax, dl sub rsi, 2 jg loop

In Example 3-23, RBX/RCX and RDX/RAX are pairs of registers that are shared and continuously over-written. In the right-hand sequence, registers are overwritten with new results immediately, consuming less internal resources provided by the underlying microarchitecture. As a result, it is about 8% faster than the left-hand sequence where internal resources could only support 50% of the attempt to take advantage of zero-latency MOV instructions.

3.5.2 Avoiding Stalls in Execution Core

Although the design of the execution core is optimized to make common cases executes quickly. A micro-op may encounter various hazards, delays, or stalls while making forward progress from the front end to the ROB and RS. The significant cases are:

- ROB Read Port Stalls.
- Partial Register Reference Stalls.
- Partial Updates to XMM Register Stalls.
- Partial Flag Register Reference Stalls.

3.5.2.1 Writeback Bus Conflicts

The writeback bus inside the execution engine is a common resource needed to facilitate out-of-order execution of micro-ops in flight. When the writeback bus is needed at the same time by two micro-ops executing in the same stack of execution units (see Table E-11 in Appendix E, "Earlier Generations of Intel® 64 and IA-32 Processor Architectures"), the younger micro-op will have to wait for the writeback bus to be available. This situation typically will be more likely for short-latency instructions experience a delay when it might have been otherwise ready for dispatching into the execution engine.

Consider a repeating sequence of independent floating-point ADDs with a single-cycle MOV bound to the same dispatch port. When the MOV finds the dispatch port available, the writeback bus can be occupied by the ADD. This delays the MOV operation.

If this problem is detected, you can sometimes change the instruction selection to use a different dispatch port and reduce the writeback contention.

3.5.2.2 Bypass Between Execution Domains

Floating-point (FP) loads have an extra cycle of latency. Moves between FP and SIMD stacks have another additional cycle of latency.

Example:

```
ADDPS  XMM0, XMM1
PAND  XMM0, XMM3
ADDPS  XMM2, XMM0
```

The overall latency for the above calculation is 9 cycles:

- 3 cycles for each ADDPS instruction.
- 1 cycle for the PAND instruction.
- 1 cycle to bypass between the ADDPS floating-point domain to the PAND integer domain.
- 1 cycle to move the data from the PAND integer to the second floating-point ADDPS domain.

To avoid this penalty, organize code to minimize domain changes. Sometimes bypasses cannot be avoided.

Account for bypass cycles when counting the overall latency of your code. If your calculation is latency-bound, you can execute more instructions in parallel or break dependency chains to reduce total latency.

Code that has many bypass domains and is completely latency-bound may run slower on the Intel Core microarchitecture than it did on previous microarchitectures.

3.5.2.3 Partial Register Stalls

Beginning with the Skylake microarchitecture, Partial Register Stalls are no longer treated using micro-operation (UOP) insertions. The hardware takes care of merging the partial register (for instance any of AL, AH or AX is merged into the RAX destination register). This eliminates the special allocation window used to insert merge micro-operation.

From Skylake to Ice Lake microarchitectures, operations that access *H registers (i.e., AH, BH, CH, DH) are executed exclusively on ports 1 and 5.

The *H micro-ops are executed with one cycle latency; however, one cycle of *additional* delay is required for ensuing UOPs because they depend on the results of the *H operation. This additional delay is required due to potential data swapping. A swap might happen, for example, with the instruction "Add AH, BL", or "ADD AL, BH." The pipeline functionality is illustrated in Figure 2-3.

Beginning with the Golden Cove Microarchitecture, the *H operations are limited to Port 1 (port1) with three cycles of latency. This penalty on *H operations helped performance improvement and timing requirements of the Golden Cove microarchitecture.

For more information about Golden Cove microarchitecture, see Section 2.3.1, "Golden Cove Microarchitecture Overview". Figure 2-1 shows the flow.

A closer look at the INT execution ports in Figure 3-1 shows the *H operation limited to Port 1:

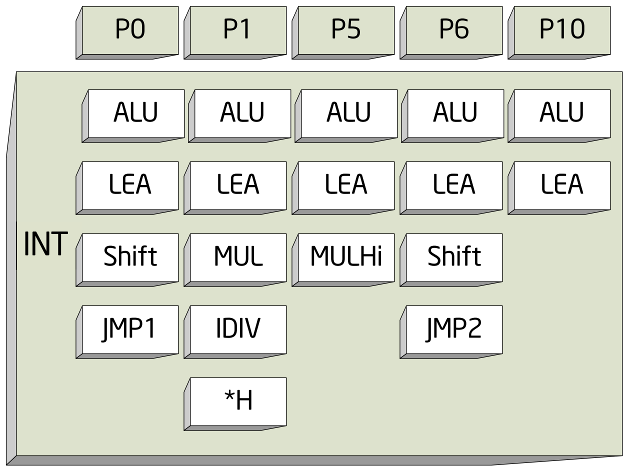


Figure 3-1. INT Execution Ports Within the Processor Core Pipeline

3.5.2.4 Partial XMM Register Stalls

Partial register stalls can also apply to XMM registers. The following SSE and SSE2 instructions update only part of the destination register:

```
MOVL/HPD XMM, MEM64
MOVL/HPS XMM, MEM32
MOVSS/SD between registers
```

Using these instructions creates a dependency chain between the unmodified part of the register and the modified part of the register. This dependency chain can cause performance loss.

Example 3-24 illustrates the use of MOVZX to avoid a partial register stall when packing three byte values into a register.

Follow these recommendations to avoid stalls from partial updates to XMM registers:

- Avoid using instructions which update only part of the XMM register.
- If a 64-bit load is needed, use the MOVSD or MOVQ instruction.
- If 2 64-bit loads are required to the same register from non continuous locations, use MOVSD/MOVHPD instead of MOVLPD/MOVHPD.
- When copying the XMM register, use the following instructions for full register copy, even if you only want to copy some of the source register data:

```
MOVAPS
MOVAPD
MOVDQA
```

Example 3-24. Avoiding Partial Register Stalls in SIMD Code

Using movlpd for memory transactions and movsd between register copies Causing Partial Register Stall	Using movsd for memory and movapd between register copies Avoid Delay
mov edx, x mov ecx, count movlpd xmm3,_1_ movlpd xmm2,_1pt5_ align 16	mov edx, x mov ecx, count movsd xmm3,_1_ movsd xmm2, _1pt5_ align 16

Example 3-24. Avoiding Partial Register Stalls in SIMD Code (Contd.)

Using movlpd for memory transactions and movsd between register copies Causing Partial Register Stall	Using movsd for memory and movapd between register copies Avoid Delay
lp: movlpd xmm0, [edx] addsd xmm0, xmm3 movsd xmm1, xmm2 subsd xmm1, [edx] mulsd xmm0, xmm1 movsd [edx], xmm0 add edx, 8 dec ecx jnz lp	lp: movsd xmm0, [edx] addsd xmm0, xmm3 movapd xmm1, xmm2 subsd xmm1, [edx] mulsd xmm0, xmm1 movsd [edx], xmm0 add edx, 8 dec ecx jnz lp

3.5.2.5 Partial Flag Register Stalls

A "partial flag register stall" occurs when an instruction modifies a part of the flag register and the following instruction is dependent on the outcome of the flags. This happens most often with shift instructions (SAR, SAL, SHR, SHL). The flags are not modified in the case of a zero shift count, but the shift count is usually known only at execution time. The front end stalls until the instruction is retired.

Other instructions that can modify some part of the flag register include CMPXCHG8B, various rotate instructions, STC, and STD. An example of assembly with a partial flag register stall and alternative code without the stall is shown in Example 3-25.

In processors based on Intel Core microarchitecture, shift immediate by 1 is handled by special hardware such that it does not experience partial flag stall.

Example 3-25. Avoiding Partial Flag Register Stalls

Partial Flag Register Stall	Avoiding Partial Flag Register Stall
xor eax, eax mov ecx, a sar ecx, 2 setz al ;SAR can update carry causing a stall	or eax, eax mov ecx, a sar ecx, 2 test ecx, ecx ; test always updates all flags setz al ;No partial reg or flag stall,

In Sandy Bridge microarchitecture, the cost of partial flag access is replaced by the insertion of a micro-op instead of a stall. However, it is still recommended to use less of instructions that write only to some of the flags (such as INC, DEC, SET CL) before instructions that can write flags conditionally (such as SHIFT CL).

Example 3-26 compares two techniques to implement the addition of very large integers (e.g., 1024 bits). The alternative sequence on the right side of Example 3-26 will be faster than the left side on Sandy Bridge microarchitecture, but it will experience partial flag stalls on prior microarchitectures.

Example 3-26. Partial Flag Register Accesses in Sandy Bridge Microarchitecture

Save partial flag register to avoid stall	Simplified code sequence
lea rsi, [A] lea rdi, [B] xor rax, rax mov rcx, 16 ; 16*64 =1024 bit	lea rsi, [A] lea rdi, [B] xor rax, rax mov rcx, 16

Example 3-26. Partial Flag Register Accesses in Sandy Bridge Microarchitecture

Save partial flag register to avoid stall	Simplified code sequence

lp_64bit:
    add rax, [rsi]
    adc rax, [rdi]
    mov [rdi], rax
    setc al ;save carry for next iteration
    movzx rax, al
    add rsi, 8
    add rdi, 8
    dec rcx
    jnz lp_64bit
	lp_64bit:
	    add rax, [rsi]
	    adc rax, [rdi]
	    mov [rdi], rax
	    lea rsi, [rsi+8]
	    lea rdi, [rdi+8]
	    dec rcx
	    jnz lp_64bit

3.5.2.6 Floating-Point/SIMD Operands

Moves that write a portion of a register can introduce unwanted dependences. The MOVSD REG, REG instruction writes only the bottom 64 bits of a register, not all 128 bits. This introduces a dependence on the preceding instruction that produces the upper 64 bits (even if those bits are not longer wanted). The dependence inhibits register renaming, and thereby reduces parallelism.

Use MOVAPD as an alternative; it writes all 128 bits. Even though this instruction has a longer latency, the μops for MOVAPD use a different execution port and this port is more likely to be free. The change can impact performance. There may be exceptional cases where the latency matters more than the dependence or the execution port.

Assembly/Compiler Coding Rule 39. (M impact, ML generality) *Avoid introducing dependences with partial floating-point register writes, e.g. from the* MOVSD XMMREG1, XMMREG2 *instruction. Use the* MOVAPD XMMREG1, XMMREG2 *instruction instead.*

The MOVSD XMMREG, MEM instruction writes all 128 bits and breaks a dependence.

3.5.3 Vectorization

This section provides a brief summary of optimization issues related to vectorization. There is more detail in the chapters that follow.

Vectorization is a program transformation that allows special hardware to perform the same operation on multiple data elements at the same time. Successive processor generations have provided vector support through the MMX technology, Intel Streaming SIMD Extensions (Intel SSE), Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3) and Intel Supplemental Streaming SIMD Extensions 3 (Intel SSSE3).

Vectorization is a special case of SIMD, a term defined in Flynn's architecture taxonomy to denote a single instruction stream capable of operating on multiple data elements in parallel. The number of elements which can be operated on in parallel range from four single-precision floating-point data elements in Intel SSE and two double-precision floating-point data elements in Intel SSE2 to sixteen byte operations in a 128-bit register in Intel SSE2. Thus, vector length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

The Intel C++ Compiler supports vectorization in three ways:

- The compiler may be able to generate SIMD code without intervention from the user.
- The can user insert pragmas to help the compiler realize that it can vectorize the code.
- The user can write SIMD code explicitly using intrinsics and C++ classes.

To help enable the compiler to generate SIMD code, avoid global pointers and global variables. These issues may be less troublesome if all modules are compiled simultaneously, and whole-program optimization is used.

User/Source Coding Rule 2. (H impact, M generality) *Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.*

User/Source Coding Rule 3. (M impact, ML generality) *Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence.*

The integer part of the SIMD instruction set extensions cover 8-bit,16-bit and 32-bit operands. Not all SIMD operations are supported for 32 bits, meaning that some source code will not be able to be vectorized at all unless smaller operands are used.

User/Source Coding Rule 4. (M impact, ML generality) *Avoid the use of conditional branches inside loops and consider using SSE instructions to eliminate branches.*

User/Source Coding Rule 5. (M impact, ML generality) *Keep induction (loop) variable expressions simple.*

3.5.4 Optimization of Partially Vectorizable Code

Frequently, a program contains a mixture of vectorizable code and some routines that are non-vectorizable. A common situation of partially vectorizable code involves a loop structure which include mixtures of vectorized code and unvectorizable code. This situation is depicted in Figure 3-2.

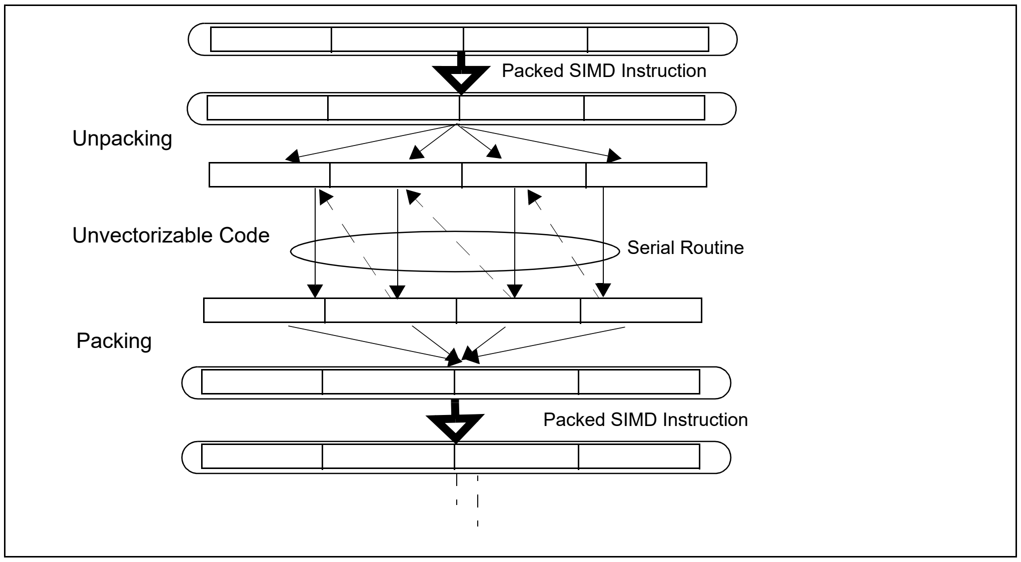


Figure 3-2. Generic Program Flow of Partially Vectorized Code

It generally consists of five stages within the loop:

- Prolog.
- Unpacking vectorized data structure into individual elements.
- Calling a unvectorizable routine to process each element serially.
- Packing individual result into vectorized data structure.
- Epilogue.

This section discusses techniques that can reduce the cost and bottleneck associated with the packing/unpacking stages in these partially vectorize code.

Example 3-27 shows a reference code template that is representative of partially vectorizable coding situations that also experience performance issues. The unvectorizable portion of code is represented generically by a sequence of calling a serial function named “foo” multiple times. This generic example is referred to as “shuffle with store forwarding”, because the problem generally involves an unpacking stage that shuffles data elements between register and memory, followed by a packing stage that can experience store forwarding issue.

There are more than one useful techniques that can reduce the store-forwarding bottleneck between the serialized portion and the packing stage. The following sub-sections presents alternate techniques to deal with the packing, unpacking, and parameter passing to serialized function calls.

Example 3-27. Reference Code Template for Partially Vectorizable Program

```
// Prolog  ///////////////////////////////
push ebp
mov ebp, esp

// Unpacking  ////////////////////////////
sub ebp, 32
and ebp, 0xfffffff0
movaps [ebp], xmm0


// Serial operations on components ///////
sub ebp, 4

mov eax, [ebp+4]
mov [ebp], eax
call foo
mov [ebp+16+4], eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
mov [ebp+16+4+4], eax


mov eax, [ebp+12]
mov [ebp], eax
call foo
mov [ebp+16+8+4], eax


mov eax, [ebp+12+4]
mov [ebp], eax
call foo
mov [ebp+16+12+4], eax


// Packing ///////////////////////////////
movaps xmm0, [ebp+16+4]

// Epilog ////////////////////////////////
pop ebp
ret
```

3.5.4.1 Alternate Packing Techniques

The packing method implemented in the reference code of Example 3-27 will experience delay as it assembles 4 doubleword result from memory into an XMM register due to store-forwarding restrictions.

Three alternate techniques for packing, using different SIMD instruction to assemble contents in XMM registers are shown in Example 3-28. All three techniques avoid store-forwarding delay by satisfying the restrictions on data sizes between a preceding store and subsequent load operations.

Example 3-28. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty

Packing Method 1	Packing Method 2	Packing Method 3
movd xmm0, [ebp+16+4] movd xmm1, [ebp+16+8] movd xmm2, [ebp+16+12] movd xmm3, [ebp+12+16+4] punpckldq xmm0, xmm1 punpckldq xmm2, xmm3 punpckldq xmm0, xmm2	movd xmm0, [ebp+16+4] movd xmm1, [ebp+16+8] movd xmm2, [ebp+16+12] movd xmm3, [ebp+12+16+4] psllq xmm3, 32 orps xmm2, xmm3 psllq xmm1, 32 orps xmm0, xmm1movlhps xmm0, xmm2	movd xmm0, [ebp+16+4] movd xmm1, [ebp+16+8] movd xmm2, [ebp+16+12] movd xmm3, [ebp+12+16+4] movlhps xmm1,xmm3 psllq xmm1, 32 movlhps xmm0, xmm2 orps xmm0, xmm1

3.5.4.2 Simplifying Result Passing

In Example 3-27, individual results were passed to the packing stage by storing to contiguous memory locations. Instead of using memory spills to pass four results, result passing may be accomplished by using either one or more registers. Using registers to simplify result passing and reduce memory spills can improve performance by varying degrees depending on the register pressure at runtime.

Example 3-29 shows the coding sequence that uses four extra XMM registers to reduce all memory spills of passing results back to the parent routine. However, software must observe the following conditions when using this technique:

- There is no register shortage.
- If the loop does not have many stores or loads but has many computations, this technique does not help performance. This technique adds work to the computational units, while the store and loads ports are idle.

Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing

```
mov eax, [ebp+4]
mov [ebp], eax
call foo
movd xmm0, eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
movd xmm1, eax
```

Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing (Contd.)

```
mov eax, [ebp+12]
mov [ebp], eax
call foo
movd xmm2, eax

mov eax, [ebp+12+4]
mov [ebp], eax
call foo
movd xmm3, eax
```

3.5.4.3 Stack Optimization

In Example 3-27, an input parameter was copied in turn onto the stack and passed to the unvectorizable routine for processing. The parameter passing from consecutive memory locations can be simplified by a technique shown in Example 3-30.

Example 3-30. Stack Optimization Technique to Simplify Parameter Passing

```
call foo
mov [ebp+16], eax

add ebp, 4
call foo
mov [ebp+16], eax


add ebp, 4
call foo
mov [ebp+16], eax

add ebp, 4
call foo
```

Stack Optimization can only be used when:

- The serial operations are function calls. The function “foo” is declared as: INT FOO(INT A). The parameter is passed on the stack.
- The order of operation on the components is from last to first.

Note the call to FOO and the advance of EDP when passing the vector elements to FOO one by one from last to first.

3.5.4.4 Tuning Considerations

Tuning considerations for situations represented by looping of Example 3-27 include:

- Applying one of more of the following combinations:
  - Choose an alternate packing technique.
  - Consider a technique to simply result-passing.
  - Consider the stack optimization technique to simplify parameter passing.
- Minimizing the average number of cycles to execute one iteration of the loop.
- Minimizing the per-iteration cost of the unpacking and packing operations.

The speed improvement by using the techniques discussed in this section will vary, depending on the choice of combinations implemented and characteristics of the non-vectorizable routine. For example, if the routine “foo” is short (representative of tight, short loops), the per-iteration cost of unpacking/packing tend to be smaller than situations where the non-vectorizable code contain longer operation or many dependencies. This is because many iterations of short, tight loop can be in flight in the execution core, so the per-iteration cost of packing and unpacking is only partially exposed and appear to cause very little performance degradation.

Evaluation of the per-iteration cost of packing/unpacking should be carried out in a methodical manner over a selected number of test cases, where each case may implement some combination of the techniques discussed in this section. The per-iteration cost can be estimated by:

- Evaluating the average cycles to execute one iteration of the test case.
- Evaluating the average cycles to execute one iteration of a base line loop sequence of non-vectorizable code.

Example 3-31 shows the base line code sequence that can be used to estimate the average cost of a loop that executes non-vectorizable routines.

Example 3-31. Base Line Code Sequence to Estimate Loop Overhead

```
push ebp
mov ebp, esp
sub ebp, 4

mov [ebp], edi
call foo


mov [ebp], edi
call foo

mov [ebp], edi
call foo


mov [ebp], edi
call foo

add ebp, 4
pop ebp
ret
```

The average per-iteration cost of packing/unpacking can be derived from measuring the execution times of a large number of iterations by:

((Cycles to run TestCase) - (Cycles to run equivalent baseline sequence)) / (Iteration count).

For example, using a simple function that returns an input parameter (representative of tight, short loops), the per-iteration cost of packing/unpacking may range from slightly more than 7 cycles (the shuffle with store forwarding case, Example 3-27) to ~0.9 cycles (accomplished by several test cases). Across 27 test cases (consisting of one of the alternate packing methods, no result-simplification/simplification of either 1 or 4 results, no stack optimization or with stack optimization), the average per-iteration cost of packing/unpacking is about 1.7 cycles.

Generally speaking, packing method 2 and 3 (see Example 3-28) tend to be more robust than packing method 1; the optimal choice of simplifying 1 or 4 results will be affected by register pressure of the runtime and other relevant microarchitectural conditions.

Note that the numeric discussion of per-iteration cost of packing/packing is illustrative only. It will vary with test cases using a different base line code sequence and will generally increase if the non-vectorizable routine requires longer time to execute because the number of loop iterations that can reside in flight in the execution core decreases.

3.6 OPTIMIZING MEMORY ACCESSES

This section discusses guidelines for optimizing code and data memory accesses. The most important recommendations are:

- Execute load and store operations within available execution bandwidth.
- Enable forward progress of speculative execution.
- Enable store forwarding to proceed.
- Align data, paying attention to data layout and stack alignment.
- Place code and data on separate pages.
- Enhance data locality.
- Use prefetching and cacheability control instructions.
- Enhance code locality and align branch targets.
- Take advantage of write combining.

3.6.1 Load and Store Execution Bandwidth

Typically, loads and stores are the most frequent operations in a workload, up to 40% of the instructions in a workload carrying load or store intent are not uncommon. Each generation of microarchitecture provides multiple buffers to support executing load and store operations while there are instructions in flight. These buffers were comprised of 128-bit wide entries for the Sandy Bridge and Ivy Bridge microarchitectures. The size was increased to 256-bit in Haswell, Broadwell and Skylake Client microarchitectures; and to 512-bit in Skylake Server, Cascade Lake, Cascade Lake Advanced Performance, and Ice Lake Client microarchitectures. To maximize performance, it is best to use the largest width available in the platform.

3.6.1.1 Making Use of Load Bandwidth in Sandy Bridge Microarchitecture

While prior microarchitecture has one load port (port 2), Sandy Bridge microarchitecture can load from port 2 and port 3. Thus two load operations can be performed every cycle and doubling the load throughput of the code. This improves code that reads a lot of data and does not need to write out results to memory very often (Port 3 also handles store-address operation). To exploit this bandwidth, the data has to stay in the L1 data cache or it should be accessed sequentially, enabling the hardware prefetchers to bring the data to the L1 data cache in time.

Consider the following C code example of adding all the elements of an array:

```
int buff[BUFF_SIZE];
int sum = 0;

for (i=0;i<BUFF_SIZE;i++){
    sum+=buff[i];
}
```

Alternative 1 is the assembly code generated by the Intel compiler for this C code, using the optimization flag for Nehalem microarchitecture. The compiler vectorizes execution using Intel SSE instructions. In this code, each ADD operation uses the result of the previous ADD operation. This limits the throughput to one load and ADD operation per cycle. Alternative 2 is optimized for Sandy Bridge microarchitecture by enabling it to use the additional load bandwidth. The code removes the dependency among ADD operations, by using two registers to sum the array values. Two load and two ADD operations can be executed every cycle.

Example 3-32. Optimizing for Load Port Bandwidth in Sandy Bridge Microarchitecture

Register dependency inhibits PADD execution	Reduce register dependency allow two load port to supply PADD execution

```
    xor     eax, eax
    pxor    xmm0, xmm0
    lea     rsi, buff


loop_start:
    paddd   xmm0, [rsi+4*rax]
    paddd   xmm0, [rsi+4*rax+16]
    paddd   xmm0, [rsi+4*rax+32]
    paddd   xmm0, [rsi+4*rax+48]
    paddd   xmm0, [rsi+4*rax+64]
    paddd   xmm0, [rsi+4*rax+80]
    paddd   xmm0, [rsi+4*rax+96]
    paddd   xmm0, [rsi+4*rax+112]
    add     eax, 32
    cmp     eax, BUFF_SIZE
    jl loop_start
sum_partials:
    movdqa  xmm1, xmm0
    psrldq  xmm1, 8
    paddd   xmm0, xmm1
    movdqa  xmm2, xmm0
    psrldq  xmm2, 4
    paddd   xmm0, xmm2
    movd    [sum], xmm0
```
	```
	    xor     eax, eax
	    pxor    xmm0, xmm0
	    pxor    xmm1, xmm1
	    lea     rsi, buff

	loop_start:
	    paddd   xmm0, [rsi+4*rax]
	    paddd   xmm1, [rsi+4*rax+16]
	    paddd   xmm0, [rsi+4*rax+32]
	    paddd   xmm1, [rsi+4*rax+48]
	    paddd   xmm0, [rsi+4*rax+64]
	    paddd   xmm1, [rsi+4*rax+80]
	    paddd   xmm0, [rsi+4*rax+96]
	    paddd   xmm1, [rsi+4*rax+112]
	    add     eax, 32
	    cmp     eax, BUFF_SIZE
	    jl loop_start
	sum_partials:
	    paddd   xmm0, xmm1
	    movdqa  xmm1, xmm0
	    psrldq  xmm1, 8
	    paddd   xmm0, xmm1
	    movdqa  xmm2, xmm0
	    psrldq  xmm2, 4
	    paddd   xmm0, xmm2
	    movd    [sum], xmm0
	```

3.6.1.2 L1D Cache Latency in Sandy Bridge Microarchitecture

Load latency from L1D cache may vary (see Table E-15 in Appendix E). The best case if 4 cycles, which apply to load operations to general purpose registers using one of the following:

- One register.
- A base register plus an offset that is smaller than 2048.

Consider the pointer-chasing code example in Example 3-33.

Example 3-33. Index versus Pointers in Pointer-Chasing Code

Traversing through indexes	Traversing through pointers

// C code example
index = buffer.m_buff[index].next_index;
// ASM example
loop:
    shl rbx, 6
    mov rbx, 0x20(rbx+rcx)
    dec rax
    cmp rax, -1
jne loop
	// C code example
	    node = node->pNext;
	// ASM example
	loop:
	    mov rdx, [rdx]
	    dec rax
	    cmp rax, -1
	    jne loop

The left side implements pointer chasing via traversing an index. Compiler then generates the code shown below addressing memory using base+index with an offset. The right side shows compiler generated code from pointer de-referencing code and uses only a base register.

The code on the right side is faster than the left side across Sandy Bridge microarchitecture and prior microarchitecture. However the code that traverses index will be slower on Sandy Bridge microarchitecture relative to prior microarchitecture.

3.6.1.3 Handling L1D Cache Bank Conflict

In Sandy Bridge microarchitecture, the internal organization of the L1D cache may manifest a situation when two load micro-ops whose addresses have a bank conflict. When a bank conflict is present between two load operations, the more recent one will be delayed until the conflict is resolved. A bank conflict happens when two simultaneous load operations have the same bit 2-5 of their linear address but they are not from the same set in the cache (bits 6 - 12).

Bank conflicts should be handled only if the code is bound by load bandwidth. Some bank conflicts do not cause any performance degradation since they are hidden by other performance limiters. Eliminating such bank conflicts does not improve performance.

The following example demonstrates bank conflict and how to modify the code and avoid them. It uses two source arrays with a size that is a multiple of cache line size. When loading an element from A and the counterpart element from B the elements have the same offset in their cache lines and therefore a bank conflict may happen.

The L1D Cache bank conflict issue does not apply to Haswell microarchitecture.

.

Example 3-34. Example of Bank Conflicts in L1D Cache and Remedy

```
int A[128];
int B[128];
int C[128];
for (i=0;i<128;i+=4){
    C[i]=A[i]+B[i];         the loads from A[i] and B[i] collide
    C[i+1]=A[i+1]+B[i+1];
    C[i+2]=A[i+2]+B[i+2];
    C[i+3]=A[i+3]+B[i+3];
}
```

// Code with Bank Conflicts	// Code without Bank Conflicts
`xor rcx, rcx`	`xor rcx, rcx`
`lea r11, A`	`lea r11, A`
`lea r12, B`	`lea r12, B`
`lea r13, C`	`lea r13, C`
`loop:`	`loop:`
`lea esi, [rcx*4]`	`lea esi, [rcx*4]`
`movsxd rsi, esi`	`movsxd rsi, esi`
`mov edi, [r11+rsi*4]`	`mov edi, [r11+rsi*4]`
`add edi, [r12+rsi*4]`	`mov r8d, [r11+rsi*4+4]`
`mov r8d, [r11+rsi*4+4]`	`add edi, [r12+rsi*4]`
`add r8d, [r12+rsi*4+4]`	`add r8d, [r12+rsi*4+4]`
`mov r9d, [r11+rsi*4+8]`	`mov r9d, [r11+rsi*4+8]`
`add r9d, [r12+rsi*4+8]`	`mov r10d, [r11+rsi*4+12]`
`mov r10d, [r11+rsi*4+12]`	`add r9d, [r12+rsi*4+8]`
`add r10d, [r12+rsi*4+12]`	`add r10d, [r12+rsi*4+12]`
`mov [r13+rsi*4], edi`	`inc ecx`
`inc ecx`	`mov [r13+rsi*4], edi`
`mov [r13+rsi*4+4], r8d`	`mov [r13+rsi*4+4], r8d`
`mov [r13+rsi*4+8], r9d`	`mov [r13+rsi*4+8], r9d`
`mov [r13+rsi*4+12], r10d`	`mov [r13+rsi*4+12], r10d`
`cmp ecx, LEN`	`cmp ecx, LEN`
`jb loop`	`jb loop`

3.6.2 Minimize Register Spills

When a piece of code has more live variables than the processor can keep in general purpose registers, a common method is to hold some of the variables in memory. This method is called register spill. The effect of L1D cache latency can negatively affect the performance of this code. The effect can be more pronounced if the address of register spills uses the slower addressing modes.

One option is to spill general purpose registers to XMM registers. This method is likely to improve performance also on previous processor generations. The following example shows how to spill a register to an XMM register rather than to memory.

Example 3-35. Using XMM Register in Lieu of Memory for Register Spills

Register spills into memory	Register spills into XMM
loop:	movq xmm4, [rsp+0x18]
mov rdx, [rsp+0x18]	mov rcx, 0x10
movdqa xmm0, [rdx]	movq xmm5, rcx
movdqa xmm1, [rsp+0x20]	loop:
pcmpeqd xmm1, xmm0	movq rdx, xmm4
pmovmskb eax, xmm1	movdqa xmm0, [rdx]
test eax, eax	movdqa xmm1, [rsp+0x20]
jne end_loop	pcmpeqd xmm1, xmm0
movzx rcx, [rbx+0x60]	pmovmskb eax, xmm1
	test eax, eax
	jne end_loop
	movzx rcx, [rbx+0x60]
add qword ptr[rsp+0x18], 0x10	padd xmm4, xmm5
add rdi, 0x4	add rdi, 0x4
movzx rdx, di	movzx rdx, di
sub rcx, 0x4	sub rcx, 0x4
add rsi, 0x1d0	add rsi, 0x1d0
cmp rdx, rcx	cmp rdx, rcx
jle loop	jle loop

3.6.3 Enhance Speculative Execution and Memory Disambiguation

Prior to Intel Core microarchitecture, when code contains both stores and loads, the loads cannot be issued before the address of the older stores is known. This rule ensures correct handling of load dependencies on preceding stores.

The Intel Core microarchitecture contains a mechanism that allows some loads to be executed speculatively in the presence of older unknown stores. The processor later checks if the load address overlapped with an older store whose address was unknown at the time the load executed. If the addresses do overlap, then the processor re-executes the load and all succeeding instructions.

Example 3-36 illustrates a situation that the compiler cannot be sure that "Ptr->Array" does not change during the loop. Therefore, the compiler cannot keep "Ptr->Array" in a register as an invariant and must read it again in every iteration. Although this situation can be fixed in software by a rewriting the code to require the address of the pointer is invariant, memory disambiguation improves performance without rewriting the code.

Example 3-36. Loads Blocked by Stores of Unknown Address

C code	Assembly sequence

struct AA {
AA ** array;
};
void nullify_array (AA *Ptr, DWORD Index, AA *ThisPtr)
{
while (Ptr->Array[--Index] != ThisPtr)
    {
    Ptr->Array[Index] = NULL ;
    } ;
} ;
	nullify_loop:
	mov dword ptr [eax], 0
	mov edx, dword ptr [edi]
	sub ecx, 4
	cmp dword ptr [ecx+edx], esi
	lea eax, [ecx+edx]
	jne nullify_loop

It is possible to disable speculative store bypass with the IA32_SPEC_CTRL.SSBD MSR.

Additional information on this topic can be found on the Software Security Guidance page.

3.6.4 Store Forwarding

The processor's memory system only sends stores to memory (including cache) after store retirement. However, store data can be forwarded from a store to a subsequent load from the same address to give a much shorter store-load latency.

There are two kinds of requirements for store forwarding. If these requirements are violated, store forwarding cannot occur and the load must get its data from the cache (so the store must write its data back to the cache first). This incurs a penalty that is largely related to pipeline depth of the underlying micro-architecture.

The first requirement pertains to the size and alignment of the store-forwarding data. This restriction is likely to have high impact on overall application performance. Typically, a performance penalty due to violating this restriction can be prevented. The store-to-load forwarding restrictions vary from one microarchitecture to another. Several examples of coding pitfalls that cause store-forwarding stalls and solutions to these pitfalls are discussed in detail in Section 3.6.4.1, "Store-to-Load-Forwarding Restriction on Size and Alignment." The second requirement is the availability of data, discussed in Section 3.6.4.2, "Store-Forwarding Restriction on Data Availability." A good practice is to eliminate redundant load operations.

It may be possible to keep a temporary scalar variable in a register and never write it to memory. Generally, such a variable must not be accessible using indirect pointers. Moving a variable to a register eliminates all loads and stores of that variable and eliminates potential problems associated with store forwarding. However, it also increases register pressure.

Load instructions tend to start chains of computation. Since the out-of-order engine is based on data dependence, load instructions play a significant role in the engine's ability to execute at a high rate. Eliminating loads should be given a high priority.

If a variable does not change between the time when it is stored and the time when it is used again, the register that was stored can be copied or used directly. If register pressure is too high, or an unseen function is called before the store and the second load, it may not be possible to eliminate the second load.

Assembly/Compiler Coding Rule 40. (H impact, M generality) *Pass parameters in registers instead of on the stack where possible. Passing arguments on the stack requires a store followed by a reload. While this sequence is optimized in hardware by providing the value to the load directly from* *the memory order buffer without the need to access the data cache if permitted by store-forwarding restrictions, floating-point values incur a significant latency in forwarding. Passing floating-point arguments in (preferably XMM) registers should save this long latency operation.*

Parameter passing conventions may limit the choice of which parameters are passed in registers which are passed on the stack. However, these limitations may be overcome if the compiler has control of the compilation of the whole binary (using whole-program optimization).

3.6.4.1 Store-to-Load-Forwarding Restriction on Size and Alignment

Data size and alignment restrictions for store-forwarding apply to processors based on Intel Core microarchitecture, Intel Core 2 Duo, Intel Core Solo and Pentium M processors. The performance penalty for violating store-forwarding restrictions is less for shorter-pipelined machines.

Store-forwarding restrictions vary with each microarchitecture. The following rules help satisfy size and alignment restrictions for store forwarding:

Assembly/Compiler Coding Rule 41. (H impact, M generality) *A load that forwards from a store must have the same address start point and therefore the same alignment as the store data.*

Assembly/Compiler Coding Rule 42. (H impact, M generality) *The data of a load which is forwarded from a store must be completely contained within the store data.*

A load that forwards from a store must wait for the store's data to be written to the store buffer before proceeding, but other, unrelated loads need not wait.

Assembly/Compiler Coding Rule 43. (H impact, ML generality) *If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. This is better than incurring the penalties of a failed store-forward.*

Assembly/Compiler Coding Rule 44. (MH impact, ML generality) *Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed.*

Example 3-37 depicts several store-forwarding situations in which small loads follow large stores. The first three load operations illustrate the situations described in Rule 44. However, the last load operation gets data from store-forwarding without problem.

Example 3-37. Situations Showing Small Loads After Large Store

```
mov [EBP],'abcd'
mov AL, [EBP]       ; Not blocked - same alignment
mov BL, [EBP + 1]   ; Blocked
mov CL, [EBP + 2]   ; Blocked
mov DL, [EBP + 3]   ; Blocked
mov AL, [EBP]       ; Not blocked - same alignment
                    ; n.b. passes older blocked loads
```

Example 3-38 illustrates a store-forwarding situation in which a large load follows several small stores. The data needed by the load operation cannot be forwarded because all of the data that needs to be forwarded is not contained in the store buffer. Avoid large loads after small stores to the same area of memory.

Example 3-38. Non-forwarding Example of Large Load After Small Store

```
mov [EBP], 'a'
mov [EBP + 1], 'b'
mov [EBP + 2], 'c'
mov [EBP + 3], 'd'
mov EAX, [EBP]   ; Blocked
     ; The first 4 small store can be consolidated into
     ; a single DWORD store to prevent this non-forwarding
     ; situation.
```

Example 3-39 illustrates a stalled store-forwarding situation that may appear in compiler generated code. Sometimes a compiler generates code similar to that shown in Example 3-39 to handle a spilled byte to the stack and convert the byte to an integer value.

Example 3-39. A Non-forwarding Situation in Compiler Generated Code

```
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
mov eax, DWORD PTR [esp+10h]  ; Stall
and eax, 0xff                 ; Converting back to byte value
```

Example 3-40 offers two alternatives to avoid the non-forwarding situation shown in Example 3-39.

Example 3-40. Two Ways to Avoid Non-forwarding Situation in Example 3-39

```
; A. Use MOVZ instruction to avoid large load after small
; store, when spills are ignored.
movz eax, bl                     ; Replaces the last three instructions
; B. Use MOVZ instruction and handle spills to the stack
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
movz eax, BYTE PTR [esp+10h]     ; Not blocked
```

When moving data that is smaller than 64 bits between memory locations, 64-bit or 128-bit SIMD register moves are more efficient (if aligned) and can be used to avoid unaligned loads. Although floating-point registers allow the movement of 64 bits at a time, floating-point instructions should not be used for this purpose, as data may be inadvertently modified.

As an additional example, consider the cases in Example 3-41.

Example 3-41. Large and Small Load Stalls

```
; A. Large load stall
mov     mem, eax        ; Store dword to address "MEM"
mov     mem + 4, ebx    ; Store dword to address "MEM + 4"
fld     mem             ; Load qword at address "MEM", stalls
; B. Small Load stall
fstp  mem               ; Store qword to address "MEM"
mov   bx, mem+2         ; Load word at address "MEM + 2", stalls
mov   cx, mem+4         ; Load word at address "MEM + 4", stalls
```

In the first case (A), there is a large load after a series of small stores to the same area of memory (beginning at memory address MEM). The large load will stall.

The FLD must wait for the stores to write to memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory).

In the second case (B), there is a series of small loads after a large store to the same area of memory (beginning at memory address MEM). The small loads will stall.

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). This can be avoided by moving the store as far from the loads as possible.

Store forwarding restrictions for processors based on Intel Core microarchitecture is listed in Table 3-4.

Table 3-4. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture

Store Alignment	Width of Store (bits)	Load Alignment (byte)	Width of Load (bits)	Store Forwarding Restriction
To Natural size	16	word aligned	8, 16	not stalled
To Natural size	16	not word aligned	8	stalled
To Natural size	32	dword aligned	8, 32	not stalled
To Natural size	32	not dword aligned	8	stalled
To Natural size	32	word aligned	16	not stalled
To Natural size	32	not word aligned	16	stalled
To Natural size	64	qword aligned	8, 16, 64	not stalled
To Natural size	64	not qword aligned	8, 16	stalled
To Natural size	64	dword aligned	32	not stalled
To Natural size	64	not dword aligned	32	stalled
To Natural size	128	dqword aligned	8, 16, 128	not stalled
To Natural size	128	not dqword aligned	8, 16	stalled
To Natural size	128	dword aligned	32	not stalled
To Natural size	128	not dword aligned	32	stalled
To Natural size	128	qword aligned	64	not stalled
To Natural size	128	not qword aligned	64	stalled
Unaligned, start byte 1	32	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	32	not byte 0 of store	8, 16	stalled
Unaligned, start byte 1	64	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	64	not byte 0 of store	8, 16, 32	stalled
Unaligned, start byte 1	64	byte 0 of store	64	stalled
Unaligned, start byte 7	32	byte 0 of store	8	not stalled
Unaligned, start byte 7	32	not byte 0 of store	8	not stalled
Unaligned, start byte 7	32	don’t care	16, 32	stalled
Unaligned, start byte 7	64	don’t care	16, 32, 64	stalled

3.6.4.2 Store-Forwarding Restriction on Data Availability

The value to be stored must be available before the load operation can be completed. If this restriction is violated, the execution of the load will be delayed until the data is available. This delay causes some execution resources to be used unnecessarily, and that can lead to sizable but non-deterministic delays. However, the overall impact of this problem is much smaller than that from violating size and alignment requirements.

In modern microarchitectures, hardware predicts when loads are dependent on and get their data forwarded from preceding stores. These predictions can significantly improve performance. However, if a load is scheduled too soon after the store it depends on or if the generation of the data to be stored is delayed, there can be a significant penalty.

There are several cases in which data is passed through memory, and the store may need to be separated from the load:

- Spills, save and restore registers in a stack frame.
- Parameter passing.
- Global and volatile variables.

- Type conversion between integer and floating-point.
- When compilers do not analyze code that is inlined, forcing variables that are involved in the interface with inlined code to be in memory, creating more memory variables and preventing the elimination of redundant loads.

Assembly/Compiler Coding Rule 45. (H impact, MH generality) *Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency instruction - for example, MUL or DIV. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain.*

Example 3-42 shows an example of a loop-carried dependence chain.

Example 3-42. Loop-Carried Dependence Chain

```
for ( i = 0; i < MAX; i++ ) {
    a[i] = b[i] * foo;
    foo = a[i] / 3;
}                    // foo is a loop-carried dependence.
```

Assembly/Compiler Coding Rule 46. (M impact, MH generality) *Calculate store addresses as early as possible to avoid having stores block loads.*

3.6.5 Data Layout Optimizations

User/Source Coding Rule 6. (H impact, M generality) *Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary.*

If the operands are packed in a SIMD instruction, align to the packed element size (64-bit or 128-bit).

Align data by providing padding inside structures and arrays. Programmers can reorganize structures and arrays to minimize the amount of memory wasted by padding. However, compilers might not have this freedom. The C programming language, for example, specifies the order in which structure elements are allocated in memory. For more information, see Section 5.4, "Stack and Data Alignment."

Example 3-43 shows how a data structure could be rearranged to reduce its size.

Example 3-43. Rearranging a Data Structure

```
struct unpacked { /* Fits in 20 bytes due to padding */
    int     a;
    char    b;
    int     c;
    char    d;
    int     e;
};

struct packed {  /* Fits in 16 bytes */
    int     a;
    int     c;
    int     e;
    char    b;
    char    d;
}
```

Cache line size of 64 bytes can impact streaming applications (for example, multimedia). These reference and use data only once before discarding it. Data accesses which sparsely utilize the data within a cache line can result in less efficient utilization of system memory bandwidth. For example, arrays of structures can be decomposed into several arrays to achieve better packing, as shown in Example 3-44.

Example 3-44. Decomposing an Array

```
struct {        /* 1600 bytes */
    int   a, c, e;
    char b, d;
} array_of_struct [100];

struct {        /* 1400 bytes */
    int   a[100], c[100], e[100];
    char b[100], d[100];
} struct_of_array;


struct {        /* 1200 bytes */
    int   a, c, e;
} hybrid_struct_of_array_ace[100];


struct {        /* 200 bytes */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

The efficiency of such optimizations depends on usage patterns. If the elements of the structure are all accessed together but the access pattern of the array is random, then ARRAY_OF_STRUCT avoids unnecessary prefetch even though it wastes memory.

However, if the access pattern of the array exhibits locality (for example, if the array index is being swept through) then processors with hardware prefetchers will prefetch data from STRUCT_OF_ARRAY, even if the elements of the structure are accessed together.

When the elements of the structure are not accessed with equal frequency, such as when element A is accessed ten times more often than the other entries, then STRUCT_OF_ARRAY not only saves memory, but it also prevents fetching unnecessary data items B, C, D, and E.

Using STRUCT_OF_ARRAY also enables the use of the SIMD data types by the programmer and the compiler.

Note that STRUCT_OF_ARRAY can have the disadvantage of requiring more independent memory stream references. This can require the use of more prefetches and additional address generation calculations. It can also have an impact on DRAM page access efficiency. An alternative, HYBRID_STRUCT_OF_ARRAY blends the two approaches. In this case, only 2 separate address streams are generated and referenced: 1 for HYBRID_STRUCT_OF_ARRAY_ACE and 1 for HYBRID_STRUCT_OF_ARRAY_BD. The second alterative also prevents fetching unnecessary data — assuming that (1) the variables A, C and E are always used together, and (2) the variables B and D are always used together, but not at the same time as A, C and E.

The hybrid approach ensures:

- Simpler/fewer address generations than STRUCT_OF_ARRAY.
- Fewer streams, which reduces DRAM page misses.
- Fewer prefetches due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

Assembly/Compiler Coding Rule 47. (H impact, M generality) *Try to arrange data structures such that they permit sequential access.*

If the data is arranged into a set of streams, the automatic hardware prefetcher can prefetch data that will be needed by the application, reducing the effective memory latency. If the data is accessed in a non-sequential manner, the automatic hardware prefetcher cannot prefetch the data. The prefetcher can recognize up to eight concurrent streams. See Chapter 9, "Optimizing Cache Usage," for more information on the hardware prefetcher.

User/Source Coding Rule 7. (M impact, L generality) *Beware of false sharing within a cache line (64 bytes).*

3.6.6 Stack Alignment

Performance penalty of unaligned access to the stack happens when a memory reference splits a cache line. This means that one out of eight spatially consecutive unaligned quadword accesses is always penalized, similarly for one out of 4 consecutive, non-aligned double-quadword accesses, etc.

Aligning the stack may be beneficial any time there are data objects that exceed the default stack alignment of the system. For example, on 32/64bit Linux, and 64bit Windows, the default stack alignment is 16 bytes, while 32bit Windows is 4 bytes.

Assembly/Compiler Coding Rule 48. (H impact, M generality) *Make sure that the stack is aligned at the largest multi-byte granular data type boundary matching the register width.*

Aligning the stack typically requires the use of an additional register to track across a padded area of unknown amount. There is a trade-off between causing unaligned memory references that spanned across a cache line and causing extra general purpose register spills.

The assembly level technique to implement dynamic stack alignment may depend on compilers, and specific OS environment. The reader may wish to study the assembly output from a compiler of interest.

Example 3-45. Examples of Dynamical Stack Alignment

```
// 32-bit environment
    push    ebp ; save ebp
    mov     ebp, esp ; ebp now points to incoming parameters
    andl    esp, $-<N> ;align esp to N byte boundary
    sub     esp, $<stack_size>; reserve space for new stack frame
    .        ; parameters must be referenced off of ebp
    mov     esp, ebp ; restore esp
    pop     ebp ; restore ebp


// 64-bit environment
    sub     esp, $<stack_size +N>
    mov     r13, $<offset_of_aligned_section_in_stack>
    andl    r13, $-<N> ; r13 point to aligned section in stack
    .        ;use r13 as base for aligned data
```

If for some reason it is not possible to align the stack for 64-bits, the routine should access the parameter and save it into a register or known aligned storage, thus incurring the penalty only once.

3.6.7 Capacity Limits and Aliasing in Caches

There are cases in which addresses with a given stride will compete for some resource in the memory hierarchy.

Typically, caches are implemented to have multiple ways of set associativity, with each way consisting of multiple sets of cache lines (or sectors in some cases). Multiple memory references that compete for the same set of each way in a cache can cause a capacity issue. There are aliasing conditions that apply to specific microarchitectures. Note that first-level cache lines are 64 bytes. Thus, the least significant 6 bits are not considered in alias comparisons.

3.6.8 Mixing Code and Data

The aggressive prefetching and pre-decoding of instructions by Intel processors have two related effects:

- Self-modifying code works correctly, according to the Intel architecture processor requirements, but incurs a significant performance penalty. Avoid self-modifying code if possible.
- Placing writable data in the code segment might be impossible to distinguish from self-modifying code. Writable data in the code segment might suffer the same performance penalty as self-modifying code.

Assembly/Compiler Coding Rule 49. (M impact, L generality) *If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch.*

Tuning Suggestion 1. *In rare cases, a performance problem may be caused by executing data on a code page as instructions. This is very likely to happen when execution is following an indirect branch that is not resident in the trace cache. If this is clearly causing a performance problem, try moving the data elsewhere, or inserting an illegal opcode or a* PAUSE *instruction immediately after the indirect branch. Note that the latter two alternatives may degrade performance in some circumstances.*

Assembly/Compiler Coding Rule 50. (H impact, L generality) *Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate 4-KByte pages or on separate aligned 1-KByte subpages.*

3.6.8.1 Self-Modifying Code

Self-modifying code (SMC) that ran correctly on Pentium III processors and prior implementations will run correctly on subsequent implementations. SMC and cross-modifying code (when multiple processors in a multiprocessor system are writing to a code page) should be avoided when high performance is desired.

Software should avoid writing to a code page in the same 1-KByte subpage that is being executed or fetching code in the same 2-KByte subpage of that is being written. In addition, sharing a page containing directly or speculatively executed code with another processor as a data page can trigger an SMC condition that causes the entire pipeline of the machine and the trace cache to be cleared. This is due to the self-modifying code condition.

Dynamic code need not cause the SMC condition if the code written fills up a data page before that page is accessed as code. Dynamically-modified code (for example, from target fix-ups) is likely to suffer from the SMC condition and should be avoided where possible. Avoid the condition by introducing indirect branches and using data tables on data pages (not code pages) using register-indirect calls.

3.6.8.2 Position Independent Code

Position independent code often needs to obtain the value of the instruction pointer. Example 3-46a shows one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 3-46b shows an alternative technique to put the value of IP into the ECX register using a matched pair of CALL/RET.

Example 3-46. Instruction Pointer Query Techniques

```
a) Using call without return to obtain IP does not corrupt the RSB
        call _label; return address pushed is the IP of next instruction
_label:
        pop ECX; IP of this instruction is now put into ECX


b) Using matched call/ret pair

        call _lblcx;
        ... ; ECX now contains IP of this instruction
        ...
_lblcx
        mov ecx, [esp];
        ret
```

3.6.9 Write Combining

Write combining (WC) improves performance in two ways:

- On a write miss to the first-level cache, it allows multiple stores to the same cache line to occur before that cache line is read for ownership (RFO) from further out in the cache/memory hierarchy. Then the rest of line is read, and the bytes that have not been written are combined with the unmodified bytes in the returned line.
- Write combining allows multiple writes to be assembled and written further out in the cache hierarchy as a unit. This saves port and bus traffic. Saving traffic is particularly important for avoiding partial writes to uncached memory.

Processors based on Intel Core microarchitecture have eight write-combining buffers in each core. Beginning with Nehalem microarchitecture, there are 10 buffers available for write-combining. Beginning with Ice Lake Client microarchitecture, there are 12 buffers available for write-combining.

Assembly/Compiler Coding Rule 51. (H impact, L generality) *If an inner loop writes to more than four arrays (four distinct cache lines), apply loop fission to break up the body of the loop such that only four arrays are being written to in each iteration of each of the resulting loops.*

Write combining buffers are used for stores of all memory types. They are particularly important for writes to uncached memory: writes to different parts of the same cache line can be grouped into a single, full-cache-line bus transaction instead of going across the bus (since they are not cached) as several partial writes. Avoiding partial writes can have a significant impact on bus bandwidth-bound graphics applications, where graphics buffers are in uncached memory. Separating writes to uncached memory and writes to writeback memory into separate phases can assure that the write combining buffers can fill before getting evicted by other write traffic. Eliminating partial write transactions has been found to have performance impact on the order of 20% for some applications. Because the cache lines are 64 bytes, a write to the bus for 63 bytes will result in partial bus transactions.

When coding functions that execute simultaneously on two threads, reducing the number of writes that are allowed in an inner loop will help take full advantage of write-combining store buffers. For write-combining buffer recommendations for Hyper-Threading Technology, see Chapter 11, “Multicore and Intel® Hyper-Threading Technology (intel® HT).”

Store ordering and visibility are also important issues for write combining. When a write to a write-combining buffer for a previously-unwritten cache line occurs, there will be a read-for-ownership (RFO). If a subsequent write happens to another write-combining buffer, a separate RFO may be caused for that cache line. Subsequent writes to the first cache line and write-combining buffer will be delayed until the second RFO has been serviced to guarantee properly ordered visibility of the writes. If the memory type for the writes is write-combining, there will be no RFO since the line is not cached, and there is no such delay. For details on write-combining, see Chapter 9, "Optimizing Cache Usage."

3.6.10 Locality Enhancement

Locality enhancement can reduce data traffic originating from an outer-level sub-system in the cache/memory hierarchy. This is to address the fact that the access-cost in terms of cycle-count from an outer level will be more expensive than from an inner level. Typically, the cycle-cost of accessing a given cache level (or memory system) varies across different microarchitectures, processor implementations, and platform components. It may be sufficient to recognize the relative data access cost trend by locality rather than to follow a large table of numeric values of cycle-costs, listed per locality, per processor/platform implementations, etc. The general trend is typically that access cost from an outer sub-system may be approximately 3-10X more expensive than accessing data from the immediate inner level in the cache/memory hierarchy, assuming similar degrees of data access parallelism.

Thus locality enhancement should start with characterizing the dominant data traffic locality. Section A, "Application Performance Tools," describes some techniques that can be used to determine the dominant data traffic locality for any workload.

Even if cache miss rates of the last level cache may be low relative to the number of cache references, processors typically spend a sizable portion of their execution time waiting for cache misses to be serviced. Reducing cache misses by enhancing a program's locality is a key optimization. This can take several forms:

- Blocking to iterate over a portion of an array that will fit in the cache (with the purpose that subsequent references to the data-block [or tile] will be cache hit references).
- Loop interchange to avoid crossing cache lines or page boundaries.
- Loop skewing to make accesses contiguous.

Locality enhancement to the last level cache can be accomplished with sequencing the data access pattern to take advantage of hardware prefetching. This can also take several forms:

- Transformation of a sparsely populated multi-dimensional array into a one-dimension array such that memory references occur in a sequential, small-stride pattern that is friendly to the hardware prefetch (see Section E.2.5.4, "Data Prefetching" in Appendix E).
- Optimal tile size and shape selection can further improve temporal data locality by increasing hit rates into the last level cache and reduce memory traffic resulting from the actions of hardware prefetching (see Section 9.5.11, "Hardware Prefetching and Cache Blocking Techniques").

It is important to avoid operations that work against locality-enhancing techniques. Using the lock prefix heavily can incur large delays when accessing memory, regardless of whether the data is in the cache or in system memory.

User/Source Coding Rule 8. (H impact, H generality) *Optimization techniques such as blocking, loop interchange, loop skewing, and packing are best done by the compiler. Optimize data structures either to fit in one-half of the first-level cache or in the second-level cache; turn on loop optimizations in the compiler to enhance locality for nested loops.*

Optimizing for one-half of the first-level cache will bring the greatest performance benefit in terms of cycle-cost per data access. If one-half of the first-level cache is too small to be practical, optimize for the second-level cache. Optimizing for a point in between (for example, for the entire first-level cache) will likely not bring a substantial improvement over optimizing for the second-level cache.

3.6.11 Non-Temporal Store Bus Traffic

Peak system bus bandwidth is shared by several types of bus activities, including reads (from memory), reads for ownership (of a cache line), and writes. The data transfer rate for bus write transactions is higher if 64 bytes are written out to the bus at a time.

Typically, bus writes to Writeback (WB) memory must share the system bus bandwidth with read-for-ownership (RFO) traffic. Non-temporal stores do not require RFO traffic; they do require care in managing the access patterns in order to ensure 64 bytes are evicted at once (rather than evicting several chunks).

Although the data bandwidth of full 64-byte bus writes due to non-temporal stores is twice that of bus writes to WB memory, transferring several chunks wastes bus request bandwidth and delivers significantly lower data bandwidth. This difference is depicted in Examples 3-47 and 3-48.

Example 3-47. Using Non-Temporal Stores and 64-byte Bus Write Transactions

```
#define STRIDESIZE 256
lea ecx, p64byte_Aligned
mov edx, ARRAY_LEN
xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0
movntps XMMWORD ptr [ecx + eax+48], xmm0
; 64 bytes is written in one bus transaction
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

Example 3-48. On-temporal Stores and Partial Bus Write Transactions

```
#define STRIDESIZE 256
Lea ecx, p64byte_Aligned
Mov edx, ARRAY_LEN
Xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0


; Storing 48 bytes results in several bus partial transactions
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

3.7 PREFETCHING

Recent Intel processor families employ several prefetching mechanisms to accelerate the movement of data or code and improve performance:

- Hardware instruction prefetcher.
- Software prefetch for data.
- Hardware prefetch for cache lines of data or instructions.

3.7.1 Hardware Instruction Fetching and Software Prefetching

Software prefetching requires a programmer to use PREFETCH hint instructions and anticipate some suitable timing and location of cache misses.

Software PREFETCH operations work the same way as do load from memory operations, with the following exceptions:

- Software PREFETCH instructions retire after virtual to physical address translation is completed.
- If an exception, such as page fault, is required to prefetch the data, then the software prefetch instruction retires without prefetching data.
- Avoid specifying a NULL address for software prefetches.

3.7.2 Hardware Prefetching for First-Level Data Cache

The hardware prefetching mechanism for L1 in Intel Core microarchitecture is discussed in Section E.3.4.2 in Appendix E.

Example 3-49 depicts a technique to trigger hardware prefetch. The code demonstrates traversing a linked list and performing some computational work on 2 members of each element that reside in 2 different cache lines. Each element is of size 192 bytes. The total size of all elements is larger than can be fitted in the L2 cache.

Example 3-49. Using DCU Hardware Prefetch

Original code	Modified sequence benefit from prefetch
mov ebx, DWORD PTR [First]	mov ebx, DWORD PTR [First]
xor eax, eax	xor eax, eax
scan_list:	scan_list:
mov eax, [ebx+4]	mov eax, [ebx+4]
mov ecx, 60	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov ecx, 60
do_some_work_1:	do_some_work_1:
add eax, eax	add eax, eax
and eax, 6	and eax, 6
sub ecx, 1	sub ecx, 1
jnz do_some_work_1	jnz do_some_work_1
mov eax, [ebx+64]	mov eax, [ebx+64]
mov ecx, 30	mov ecx, 30
do_some_work_2:	do_some_work_2:
add eax, eax	add eax, eax
and eax, 6	and eax, 6
sub ecx, 1	sub ecx, 1
jnz do_some_work_2	jnz do_some_work_2

Example 3-49. Using DCU Hardware Prefetch (Contd.)

Original code	Modified sequence benefit from prefetch
mov ebx, [ebx] test ebx, ebx jnz scan_list	mov ebx, [ebx] test ebx, ebx jnz scan_list

The additional instructions to load data from one member in the modified sequence can trigger the DCU hardware prefetch mechanisms to prefetch data in the next cache line, enabling the work on the second member to complete sooner.

Software can gain from the first-level data cache prefetchers in two cases:

- If data is not in the second-level cache, the first-level data cache prefetcher enables early trigger of the second-level cache prefetcher.
- If data is in the second-level cache and not in the first-level data cache, then the first-level data cache prefetcher triggers earlier data bring-up of sequential cache line to the first-level data cache.

There are situations that software should pay attention to a potential side effect of triggering unnecessary DCU hardware prefetches. If a large data structure with many members spanning many cache lines is accessed in ways that only a few of its members are actually referenced, but there are multiple pair accesses to the same cache line. The DCU hardware prefetcher can trigger fetching of cache lines that are not needed. In Example 3-50, references to the “Pts” array and “AltPts” will trigger DCU prefetch to fetch additional cache lines that won’t be needed. If significant negative performance impact is detected due to DCU hardware prefetch on a portion of the code, software can try to reduce the size of that contemporaneous working set to be less than half of the L2 cache.

Example 3-50. Avoid Causing DCU Hardware Prefetch to Fetch Unneeded Lines

```
while ( CurrBond != NULL )
    {
    MyATOM  *a1 = CurrBond->At1 ;
    MyATOM  *a2 = CurrBond->At2 ;


    if ( a1->CurrStep <= a1->LastStep &&
       a2->CurrStep <= a2->LastStep
       )
         {
         a1->CurrStep++ ;
         a2->CurrStep++ ;

         double  ux = a1->Pts[0].x - a2->Pts[0].x ;
         double  uy = a1->Pts[0].y - a2->Pts[0].y ;
         double  uz = a1->Pts[0].z - a2->Pts[0].z ;
         a1->AuxPts[0].x += ux ;
         a1->AuxPts[0].y += uy ;
         a1->AuxPts[0].z += uz ;
```

Example 3-50. Avoid Causing DCU Hardware Prefetch to Fetch Unneeded Lines (Contd.)

```
        a2->AuxPts[0].x += ux ;
        a2->AuxPts[0].y += uy ;
        a2->AuxPts[0].z += uz ;
        } ;
    CurrBond = CurrBond->Next ;
    } ;
```

To fully benefit from these prefetchers, organize and access the data using one of the following methods:

Method 1:

- Organize the data so consecutive accesses can usually be found in the same 4-KByte page.
- Access the data in constant strides forward or backward IP Prefetcher.

Method 2:

- Organize the data in consecutive lines.
- Access the data in increasing addresses, in sequential cache lines.

Example 3-51 demonstrates accesses to sequential cache lines that can benefit from the first-level cache prefetcher.

Example 3-51. Technique for Using L1 Hardware Prefetch

```
unsigned int *p1, j, a, b;
for (j = 0; j < num; j += 16)
{
a = p1[j];
b = p1[j+1];
// Use these two values
}
```

By elevating the load operations from memory to the beginning of each iteration, it is likely that a significant part of the latency of the pair cache line transfer from memory to the second-level cache will be in parallel with the transfer of the first cache line.

The IP prefetcher uses only the lower 8 bits of the address to distinguish a specific address. If the code size of a loop is bigger than 256 bytes, two loads may appear similar in the lowest 8 bits and the IP prefetcher will be restricted. Therefore, if you have a loop bigger than 256 bytes, make sure that no two loads have the same lowest 8 bits in order to use the IP prefetcher.

3.7.3 Hardware Prefetching for Second-Level Cache

The Intel Core microarchitecture contains two second-level cache prefetchers:

- **Streamer** — Loads data or instructions from memory to the second-level cache. To use the streamer, organize the data or instructions in blocks of 128 bytes, aligned on 128 bytes. The first access to one of the two cache lines in this block while it is in memory triggers the streamer to prefetch the pair line. To software, the L2 streamer’s functionality is similar to the adjacent cache line prefetch mechanism found in processors based on Intel NetBurst microarchitecture.
- **Data prefetch logic (DPL)** — DPL and L2 Streamer are triggered only by writeback memory type. They prefetch only inside page boundary (4 KBytes). Both L2 prefetchers can be triggered by software prefetch instructions and by prefetch request from DCU prefetchers. DPL can also be triggered by read for ownership (RFO) operations. The L2 Streamer can also be triggered by DPL requests for L2 cache misses.

Software can gain from organizing data both according to the instruction pointer and according to line strides. For example, for matrix calculations, columns can be prefetched by IP-based prefetches, and rows can be prefetched by DPL and the L2 streamer.

3.7.4 Cacheability Instructions

SSE2 provides additional cacheability instructions that extend those provided in SSE. The new cacheability instructions include:

- New streaming store instructions.
- New cache line flush instruction.
- New memory fencing instructions.

For more information, see Chapter 9, "Optimizing Cache Usage."

3.7.5 REP Prefix and Data Movement

The REP prefix is commonly used with string move instructions for memory related library functions such as MEMCPY (using REP MOVSD) or MEMSET (using REP STOS). These STRING/MOV instructions with the REP prefixes are implemented in MS-ROM and have several implementation variants with different performance levels.

The specific variant of the implementation is chosen at execution time based on data layout, alignment and the counter (ECX) value. For example, MOVSB/STOSB with the REP prefix should be used with counter value less than or equal to three for best performance.

String MOVE/STORE instructions have multiple data granularities. For efficient data movement, larger data granularities are preferable. This means better efficiency can be achieved by decomposing an arbitrary counter value into a number of doublewords plus single byte moves with a count value less than or equal to 3.

Because software can use SIMD data movement instructions to move 16 bytes at a time, the following paragraphs discuss general guidelines for designing and implementing high-performance library functions such as MEMCPY(), MEMSET(), and MEMMOVE(). Four factors are to be considered:

- **Throughput per iteration** — If two pieces of code have approximately identical path lengths, efficiency favors choosing the instruction that moves larger pieces of data per iteration. Also, smaller code size per iteration will in general reduce overhead and improve throughput. Sometimes, this may involve a comparison of the relative overhead of an iterative loop structure versus using REP prefix for iteration.
- **Address alignment** — Data movement instructions with highest throughput usually have alignment restrictions, or they operate more efficiently if the destination address is aligned to its natural data size. Specifically, 16-byte moves need to ensure the destination address is aligned to 16-byte boundaries, and 8-bytes moves perform better if the destination address is aligned to 8-byte boundaries. Frequently, moving at doubleword granularity performs better with addresses that are 8-byte aligned.
- **REP string move vs. SIMD move** — Implementing general-purpose memory functions using SIMD extensions usually requires adding some prolog code to ensure the availability of SIMD instructions, preamble code to facilitate aligned data movement requirements at runtime. Throughput comparison must also take into consideration the overhead of the prolog when considering a REP string implementation versus a SIMD approach.
- **Cache eviction** — If the amount of data to be processed by a memory routine approaches half the size of the last level on-die cache, temporal locality of the cache may suffer. Using streaming store instructions (for example: MOVNTQ, MOVNTDQ) can minimize the effect of flushing the cache. The threshold to start using a streaming store depends on the size of the last level cache. Determine the size using the deterministic cache parameter leaf of CPUID.

  Techniques for using streaming stores for implementing a MEMSET()-type library must also consider that the application can benefit from this technique only if it has no immediate need to reference the target addresses. This assumption is easily upheld when testing a streaming-store implementation on a micro-benchmark configuration, but violated in a full-scale application situation.

When applying general heuristics to the design of general-purpose, high-performance library routines, the following guidelines can are useful when optimizing an arbitrary counter value N and address alignment. Different techniques may be necessary for optimal performance, depending on the magnitude of N:

- When N is less than some small count (where the small count threshold will vary between microarchitectures -- empirically, 8 may be a good value when optimizing for Intel NetBurst microarchitecture), each case can be coded directly without the overhead of a looping structure. For example, 11 bytes can be processed using two MOVSD instructions explicitly and a MOVSB with REP counter equaling 3.
- When N is not small but still less than some threshold value (which may vary for different micro-architectures, but can be determined empirically), an SIMD implementation using run-time CPUID and alignment prolog will likely deliver less throughput due to the overhead of the prolog. A REP string implementation should favor using a REP string of doublewords. To improve address alignment, a small piece of prolog code using MOVSB/STOSB with a count less than 4 can be used to peel off the non-aligned data moves before starting to use MOVSD/STOSD.
- When N is less than half the size of last level cache, throughput consideration may favor either:
  - An approach using a REP string with the largest data granularity because a REP string has little overhead for loop iteration, and the branch misprediction overhead in the prolog/epilogue code to handle address alignment is amortized over many iterations.
  - An iterative approach using the instruction with largest data granularity, where the overhead for SIMD feature detection, iteration overhead, and prolog/epilogue for alignment control can be minimized. The trade-off between these approaches may depend on the microarchitecture.

  An example of MEMSET() implemented using stosd for arbitrary counter value with the destination address aligned to doubleword boundary in 32-bit mode is shown in Example 3-52.
- When N is larger than half the size of the last level cache, using 16-byte granularity streaming stores with prolog/epilog for address alignment will likely be more efficient, if the destination addresses will not be referenced immediately afterwards.

Example 3-52. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination

A 'C' example of Memset()	Equivalent Implementation Using REP STOSD
void memset(void *dst,int c,size_t size)	push edi
{	movzx eax, byte ptr [esp+12]
char *d = (char *)dst;	mov ecx, eax
size_t i;	shl ecx, 8
for (i=0;i<size;i++)	or ecx, eax
*d++ = (char)c;	mov ecx, eax
}	shl ecx, 16
	or eax, ecx
	mov edi, [esp+8] ; 4-byte aligned
	mov ecx, [esp+16] ; byte count
	shr ecx, 2 ; do dword
	cmp ecx, 127
	jle _main
	test edi, 4
	jz _main
	stosd ;peel off one dword
	dec ecx

Example 3-52. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination (Contd.)

A 'C' example of Memset()	Equivalent Implementation Using REP STOSD

	`_main: ; 8-byte aligned`
	`rep stosd`
	`mov ecx, [esp + 16]`
	`and ecx, 3 ; do count <= 3`
	`rep stosb ; optimal with <= 3`
	`pop edi`
	`ret`

Memory routines in the runtime library generated by Intel compilers are optimized across a wide range of address alignments, counter values, and microarchitectures. In most cases, applications should take advantage of the default memory routines provided by Intel compilers.

In some situations, the byte count of the data is known by the context (as opposed to being known by a parameter passed from a call), and one can take a simpler approach than those required for a general-purpose library routine. For example, if the byte count is also small, using REP MOVSB/STOSB with a count less than four can ensure good address alignment and loop-unrolling to finish the remaining data; using MOVSD/STOSD can reduce the overhead associated with iteration.

Using a REP prefix with string move instructions can provide high performance in the situations described above. However, using a REP prefix with string scan instructions (SCASB, SCASW, SCASD, SCASQ) or compare instructions (CMPSB, CMPSW, SMPSD, SMPSQ) is not recommended for high performance. Consider using SIMD instructions instead.

3.7.6 Enhanced REP MOVSB and STOSB Operation

Beginning with processors based on Ivy Bridge microarchitecture, REP string operation using MOVSB and STOSB can provide both flexible and high-performance REP string operations for software in common situations like memory copy and set operations. Processors that provide enhanced MOVSB/STOSB operations are enumerated by the CPUID feature flag: CPUID:(EAX=7H, ECX=0H):EBX.[bit 9] = 1.

3.7.6.1 Fast Short REP MOVSB

Beginning with processors based on Ice Lake Client microarchitecture, REP MOVSB performance of short operations is enhanced. The enhancement applies to string lengths between 1 and 128 bytes long. Support for fast-short REP MOVSB is enumerated by the CPUID feature flag: CPUID [EAX=7H, ECX=0H).EDX.FAST_SHORT_REP_MOVSB[bit 4] = 1. There is no change in the REP STOS performance.

3.7.6.2 Memcpy Considerations

The interface for the standard library function memcpy introduces several factors (e.g. length, alignment of the source buffer and destination) that interact with microarchitecture to determine the performance characteristics of the implementation of the library function. Two of the common approaches to implement memcpy are driven from small code size vs. maximum throughput. The former generally uses REP MOVSD+B (see Section 3.7.5), while the latter uses SIMD instruction sets and has to deal with additional data alignment restrictions.

For processors supporting enhanced REP MOVSB/STOSB, implementing memcpy with REP MOVSB will provide even more compact benefits in code size and better throughput than using the combination of REP MOVSD+B. For processors based on Ivy Bridge microarchitecture, implementing memcpy using Enhanced REP MOVSB and STOSB might not reach the same level of throughput as using 256-bit or 128-bit AVX alternatives, depending on length and alignment factors.

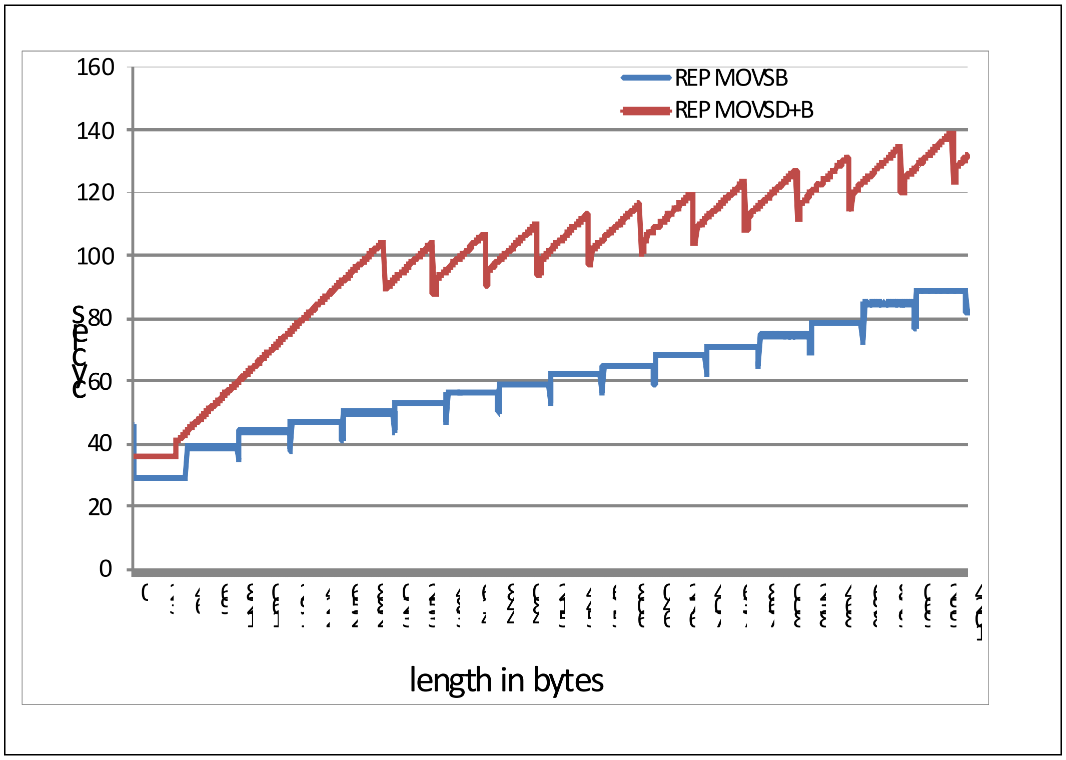


Figure 3-3. Memcpy Performance Comparison for Lengths up to 2KB

Figure 3-3 depicts the relative performance of memcpy implementation on a third-generation Intel Core processor using Enhanced REP MOVSB and STOSB versus REP MOVSD+B, for alignment conditions when both the source and destination addresses are aligned to a 16-Byte boundary and the source region does not overlap with the destination region. Using Enhanced REP MOVSB and STOSB always delivers better performance than using REP MOVSD+B. If the length is a multiple of 64, it can produce even higher performance. For example, copying 65-128 bytes takes 40 cycles, while copying 128 bytes needs only 35 cycles.

If an application wishes to bypass standard memcpy library implementation with its own custom implementation and have freedom to manage the buffer length allocation for both source and destination, it may be worthwhile to manipulate the lengths of its memory copy operation to be multiples of 64 to take advantage the code size and performance benefit of Enhanced REP MOVSB and STOSB.

The performance characteristic of implementing a general-purpose memcpy library function using a SIMD register is significantly more colorful than an equivalent implementation using a general-purpose register, depending on length, instruction set selection between SSE2, 128-bit AVX, 256-bit AVX, relative alignment of source/destination, and memory address alignment granularities/boundaries, etc.

Hence comparing performance characteristics between a memcpy using Enhanced REP MOVSB and STOSB versus a SIMD implementation is highly dependent on the particular SIMD implementation. The remainder of this section discusses the relative performance of memcpy using Enhanced REP MOVSB and STOSB versus unpublished, optimized 128-bit AVX implementation of memcpy to illustrate the hardware capability of Ivy Bridge microarchitecture.

Table 3-5. Relative Performance of Memcpy() Using Enhanced REP MOVSB and STOSB Vs. 128-bit AVX

Range of Lengths (bytes)	<128	128 to 2048	2048 to 4096
Memcpy_ERMSB/Memcpy_AVX128	0x7X	1X	1.02X

Table 3-5 shows the relative performance of the Memcpy function implemented using enhanced REP MOVSB versus 128-bit AVX for several ranges of memcpy lengths, when both the source and destination addresses are 16-byte aligned and the source region and destination region do not overlap. For memcpy length less than 128 bytes, using Enhanced REP MOVSB and STOSB is slower than what's possible using 128-bit AVX, due to internal start-up overhead in the REP string.

For situations with address misalignment, memcpy performance will generally be reduced relative to the 16-byte alignment scenario (see Table 3-6).

Table 3-6. Effect of Address Misalignment on Memcpy() Performance

Address Misalignment	Performance Impact
Source Buffer	The impact on Enhanced REP MOVSB and STOSB implementation versus 128-bit AVX is similar.
Destination Buffer	The impact on Enhanced REP MOVSB and STOSB implementation can be 25% degradation, while 128-bit AVX implementation of memcpy may degrade only 5%, relative to 16-byte aligned scenario.

Memcpy() implemented with Enhanced REP MOVSB and STOSB can benefit further from the 256-bit SIMD integer data-path in Haswell microarchitecture. See Section 15.16.3.

3.7.6.3 Memmove Considerations

When there is an overlap between the source and destination regions, software may need to use memmove instead of memcpy to ensure correctness. It is possible to use REP MOVSB in conjunction with the direction flag (DF) in a memmove() implementation to handle situations where the latter part of the source region overlaps with the beginning of the destination region. However, setting the DF to force REP MOVSB to copy bytes from high towards low addresses will experience significant performance degradation.

When using Enhanced REP MOVSB and STOSB to implement memmove function, one can detect the above situation and handle first the rear chunks in the source region that will be written to as part of the destination region, using REP MOVSB with the DF=0, to the non-overlapping region of the destination. After the overlapping chunks in the rear section are copied, the rest of the source region can be processed normally, also with DF=0.

3.7.6.4 Memset Considerations

The consideration of code size and throughput also applies for memset() implementations. For processors supporting Enhanced REP MOVSB and STOSB, using REP STOSB will again deliver more compact code size and significantly better performance than the combination of STOSD+B technique described in Section 3.7.5.

When the destination buffer is 16-byte aligned, memset() using Enhanced REP MOVSB and STOSB can perform better than SIMD approaches. When the destination buffer is misaligned, memset() performance using Enhanced REP MOVSB and STOSB can degrade about 20% relative to aligned case, for processors based on Ivy Bridge microarchitecture. In contrast, SIMD implementation of memset() will experience smaller degradation when the destination is misaligned.

Memset() implemented with Enhanced REP MOVSB and STOSB can benefit further from the 256-bit data path in Haswell microarchitecture. see Section 15.16.3.3.

3.8 REP STRING OPERATIONS

Several REP string performance enhancements are available beginning with processors based on Golden Cove microarchitecture.

3.8.1 Fast Zero Length REP MOVSB

REP MOVSB performance of zero length operations is enhanced. The latency of a zero length REP MOVSB is now the same as the latency of lengths 1 to 128 bytes. When both Fast Short REP MOVSB and Fast Zero Length REP MOVSB features are enabled, REP MOVSB performance is flat 9 cycles per operation, for all strings 0-128 byte long whose source and destination operands reside in the processor first level cache.

Support for fast zero-length REP MOVSB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_ZERO_LENGTH_REP_MOVSB[bit 10] = 1.

3.8.2 Fast Short REP STOSB

REP STOSB performance of short operations is enhanced. The enhancement applies to string lengths between 0 and 128 bytes long. When Fast Short REP STOSB feature is enabled, REP STOSB performance is flat 12 cycles per operation, for all strings 0-128 byte long whose destination operand resides in the processor first level cache.

Support for fast-short REP STOSB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_SHORT_REP_STOSB[bit 11] = 1.

3.8.3 Fast Short REP CMPSB and SCASB

REP CMPSB and SCASB performance is enhanced. The enhancement applies to string lengths between 1 and 128 bytes long. When the Fast Short REP CMPSB and SCASB feature is enabled, REP CMPSB and REP SCASB performance is flat 15 cycles per operation, for all strings 1-128 byte long whose two source operands reside in the processor first level cache.

Support for fast short REP CMPSB and SCASB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_SHORT_REP_CMPSB_SCASB[bit 12] = 1.

3.9 FLOATING-POINT CONSIDERATIONS

When programming floating-point applications, it is best to start with a high-level programming language such as C, C++, or Fortran. Many compilers perform floating-point scheduling and optimization when it is possible. However in order to produce optimal code, the compiler may need some assistance.

3.9.1 Guidelines for Optimizing Floating-Point Code

User/Source Coding Rule 9. (M impact, M generality) *Enable the compiler's use of Intel SSE, Intel SSE2, Intel AVX, Intel AVX2, and possibly more advanced SIMD instruction sets (Intel AVX-512) with appropriate switches. Favor scalar SIMD code generation to replace x87 code generation.*

Follow this procedure to investigate the performance of your floating-point application:

- Understand how the compiler handles floating-point code.
- Look at the assembly dump and see what transforms are already performed on the program.
- Study the loop nests in the application that dominate the execution time.
- Determine why the compiler is not creating the fastest code.
- See if there is a dependence that can be resolved.
- Determine the problem area: bus bandwidth, cache locality, trace cache bandwidth, or instruction latency. Focus on optimizing the problem area. For example, adding PREFETCH instructions will not help if the bus is already saturated. If trace cache bandwidth is the problem, added prefetch µops may degrade performance.

Also, in general, follow the general coding recommendations discussed in this chapter, including:

- Blocking the cache.
- Using prefetch.
- Enabling vectorization.
- Unrolling loops.

User/Source Coding Rule 10. (H impact, ML generality) *Make sure your application stays in range to avoid denormal values, underflows.*

Out-of-range numbers cause very high overhead.

When converting floating-point values to 16-bit, 32-bit, or 64-bit integers using truncation, the instructions CVTTSS2SI and CVTTSD2SI are recommended over instructions that access x87 FPU stack. This avoids changing the rounding mode.

User/Source Coding Rule 11. (M impact, ML generality) *Usually, math libraries take advantage of the transcendental instructions (for example, FSIN) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider an alternate, software-based approach, such as a look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision and the size of the look-up table, and by taking advantage of the parallelism of the Intel SSE and the Intel SSE2 instructions.*

3.9.2 Floating-Point Modes and Exceptions

When working with floating-point numbers, high-speed microprocessors frequently must deal with situations that need special handling in hardware or code.

3.9.2.1 Floating-Point Exceptions

The most frequent cause of performance degradation is the use of masked floating-point exception conditions such as:

- Arithmetic overflow.
- Arithmetic underflow.
- Denormalized operand.

Refer to Chapter 4 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for definitions of overflow, underflow and denormal exceptions.

Denormalized floating-point numbers impact performance in two ways:

- Directly when are used as operands.
- Indirectly when are produced as a result of an underflow situation.

If a floating-point application never underflows, the denormals can only come from floating-point constants.

User/Source Coding Rule 12. (H impact, ML generality) *Denormalized floating-point constants should be avoided as much as possible.*

Denormal and arithmetic underflow exceptions can occur during the execution of x87 instructions or Intel SSE/Intel SSE2/Intel SSE3 instructions. Processors based on Intel NetBurst microarchitecture handle these exceptions more efficiently when executing Intel SSE/Intel SSE2/Intel SSE3 instructions and when speed is more important than complying with the IEEE standard. The following paragraphs give recommendations on how to optimize your code to reduce performance degradations related to floating-point exceptions.

3.9.2.2 Dealing with Floating-Point Exceptions in x87 FPU Code

Every special situation listed in Section 3.9.2.1, "Floating-Point Exceptions," is costly in terms of performance. For that reason, x87 FPU code should be written to avoid these situations.

There are basically three ways to reduce the impact of overflow/underflow situations with x87 FPU code:

- Choose floating-point data types that are large enough to accommodate results without generating arithmetic overflow and underflow exceptions.
- Scale the range of operands/results to reduce as much as possible the number of arithmetic overflow/underflow situations.
- Keep intermediate results on the x87 FPU register stack until the final results have been computed and stored in memory. Overflow or underflow is less likely to happen when intermediate results are kept in the x87 FPU stack (this is because data on the stack is stored in double extended-precision format and overflow/underflow conditions are detected accordingly).
- Denormalized floating-point constants (which are read-only, and hence never change) should be avoided and replaced, if possible, with zeros of the same sign.

3.9.2.3 Floating-Point Exceptions in SSE/SSE2/SSE3 Code

Most special situations that involve masked floating-point exceptions are handled efficiently in hardware. When a masked overflow exception occurs while executing Intel SSE/Intel SSE2/Intel SSE3/Intel AVX/Intel AVX2/Intel AVX-512 code, processor hardware can handles it without performance penalty.

Underflow exceptions and denormalized source operands are usually treated according to the IEEE 754 specification[1], but this can incur significant performance delay. If a programmer is willing to trade pure IEEE 754 compliance for speed, two non-IEEE 754 compliant modes are provided to speed situations where underflows and input are frequent: FTZ mode and DAZ mode.

When the FTZ mode is enabled, an underflow result is automatically converted to a zero with the correct sign. Although this behavior is not compliant with IEEE 754, it is provided for use in applications where performance is more important than IEEE 754 compliance. Since denormal results are not produced when the FTZ mode is enabled, the only denormal floating-point numbers that can be encountered in FTZ mode are the ones specified as constants (read only).

The DAZ mode is provided to handle denormal source operands efficiently when running a SIMD floating-point application. When the DAZ mode is enabled, input denormals are treated as zeros with the same sign. Enabling the DAZ mode is the way to deal with denormal floating-point constants when performance is the objective.

If departing from the IEEE 754 specification is acceptable and performance is critical, run Intel SSE/Intel SSE2/Intel SSE3/Intel AVX/Intel AVX2/Intel AVX-512 applications with FTZ and DAZ modes enabled.

NOTE

The DAZ mode is available with both the Intel SSE and Intel SSE2 extensions, although the speed improvement expected from this mode is fully realized only in SSE code and later.

3.9.3 Floating-Point Modes

For x87 code, using the FLDCW instruction to change floating modes can be an expensive operation in many cases.

Recent processor generations provide hardware optimization for FLDCW that allows programmers to alternate between two constant values efficiently. For the FLDCW optimization to be effective, the two constant FCW values are only allowed to differ on the following 5 bits in the FCW:

1. "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229.

```
FCW[8-9]     ; Precision control
FCW[10-11]   ; Rounding control
FCW[12]      ; Infinity control
```

If programmers need to modify other bits (for example: mask bits) in the FCW, the FLDCW instruction is still an expensive operation.

In situations where an application cycles between three (or more) constant values, FLDCW optimization does not apply, and the performance degradation occurs for each FLDCW instruction.

One solution to this problem is to choose two constant FCW values, take advantage of the optimization of the FLDCW instruction to alternate between only these two constant FCW values, and devise some means to accomplish the task that requires the 3rd FCW value without actually changing the FCW to a third constant value. An alternative solution is to structure the code so that, for periods of time, the application alternates between only two constant FCW values. When the application later alternates between a pair of different FCW values, the performance degradation occurs only during the transition.

It is expected that SIMD applications are unlikely to alternate between FTZ and DAZ mode values. Consequently, the SIMD control word does not have the short latencies that the floating-point control register does. A read of the MXCSR register has a fairly long latency, and a write to the register is a serializing instruction.

There is no separate control word for single and double precision; both use the same modes. Notably, this applies to both FTZ and DAZ modes.

Assembly/Compiler Coding Rule 52. (H impact, M generality) *Minimize changes to bits 8-12 of the floating-point control word. Changes for more than two values (each value being a combination of the following bits: precision, rounding and infinity control, and the rest of bits in FCW) leads to delays that are on the order of the pipeline depth.*

3.9.3.1 Rounding Mode

Many libraries provide float-to-integer library routines that convert floating-point values to integer. Many of these libraries conform to ANSI C coding standards which state that the rounding mode should be truncation. With the Pentium 4 processor, one can use the CVTTSD2SI and CVTTSS2SI instructions to convert operands with truncation without ever needing to change rounding modes. The cost savings of using these instructions over the methods below is enough to justify using Intel SSE and Intel SSE2 wherever possible when truncation is involved.

For x87 floating-point, the FIST instruction uses the rounding mode represented in the floating-point control word (FCW). The rounding mode is generally "round to nearest", so many compiler writers implement a change in the rounding mode in the processor in order to conform to the C and FORTRAN standards. This implementation requires changing the control word on the processor using the FLDCW instruction. For a change in the rounding, precision, and infinity bits, use the FSTCW instruction to store the floating-point control word. Then use the FLDCW instruction to change the rounding mode to truncation.

In a typical code sequence that changes the rounding mode in the FCW, a FSTCW instruction is usually followed by a load operation. The load operation from memory should be a 16-bit operand to prevent store-forwarding problem. If the load operation on the previously-stored FCW word involves either an 8-bit or a 32-bit operand, this will cause a store-forwarding problem due to mismatch of the size of the data between the store operation and the load operation.

To avoid store-forwarding problems, make sure that the write and read to the FCW are both 16-bit operations.

If there is more than one change to the rounding, precision, and infinity bits, and the rounding mode is not important to the result, use the algorithm in Example 3-53 to avoid synchronization issues, the overhead of the FLDCW instruction, and having to change the rounding mode. Note that the example suffers from a store-forwarding problem which will lead to a performance penalty. However, its performance is still better than changing the rounding, precision, and infinity bits among more than two values.

Example 3-53. Algorithm to Avoid Changing Rounding Mode

```
_fto132proc
    lea     ecx, [esp-8]
    sub     esp, 16             ; Allocate frame
    and     ecx, -8             ; Align pointer on boundary of 8
    fld     st(0)               ; Duplicate FPU stack top

    fistp   qword ptr[ecx]
    fild    qword ptr[ecx]
    mov     edx, [ecx+4]        ; High DWORD of integer
    mov     eax, [ecx]          ; Low DWIRD of integer
    test    eax, eax
    je      integer_QnaN_or_zero

arg_is_not_integer_QnaN:
    fsubp   st(1), st           ; TOS=d-round(d), { st(1) = st(1)-st & pop ST}
    test    edx, edx            ; What's sign of integer
    jns     positive            ; Number is negative
    fstp    dword ptr[ecx]      ; Result of subtraction
    mov     ecx, [ecx]          ; DWORD of diff(single-precision)
    add     esp, 16
    xor     ecx, 80000000h
    add     ecx,7fffffffh       ; If diff<0 then decrement integer
    adc     eax,0               ; INC EAX (add CARRY flag)
    ret
positive:

    positive:
    fstp    dword ptr[ecx]      ; 17-18 result of subtraction
    mov     ecx, [ecx]          ; DWORD of diff(single precision)
    add     esp, 16
    add     ecx, 7fffffffh      ; If diff<0 then decrement integer
    sbb     eax, 0              ; DEC EAX (subtract CARRY flag)
    ret
integer_QnaN_or_zero:
    test    edx, 7fffffffh
    jnz     arg_is_not_integer_QnaN
    add  esp, 16
    ret
```

Assembly/Compiler Coding Rule 53. (H impact, L generality) *Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision, and infinity bits.*

3.9.3.2 Precision

If single precision is adequate, use it instead of double precision. This is true because:

- Single precision operations allow the use of longer SIMD vectors, since more single precision data elements can fit in a register.
- If the precision control (PC) field in the x87 FPU control word is set to single precision, the floating-point divider can complete a single-precision computation much faster than either a

double-precision computation or an extended double-precision computation. If the PC field is set to double precision, this will enable those x87 FPU operations on double-precision data to complete faster than extended double-precision computation. These characteristics affect computations including floating-point divide and square root.

Assembly/Compiler Coding Rule 54. (H impact, L generality) *Minimize the number of changes to the precision mode.*

3.9.4 x87 vs. Scalar SIMD Floating-Point Trade-Offs

There are a number of differences between x87 floating-point code and scalar floating-point code (using Intel SSE and Intel SSE2). The following differences should drive decisions about which registers and instructions to use:

- When an input operand for a SIMD floating-point instruction contains values that are less than the representable range of the data type, a denormal exception occurs. This causes a significant performance penalty. An SIMD floating-point operation has a flush-to-zero mode in which the results will not underflow. Therefore subsequent computation will not face the performance penalty of handling denormal input operands. For example, in the case of 3D applications with low lighting levels, using flush-to-zero mode can improve performance by as much as 50% for applications with large numbers of underflows.
- Scalar floating-point SIMD instructions have lower latencies than equivalent x87 instructions. Scalar SIMD floating-point multiply instruction may be pipelined, while x87 multiply instruction is not.
- Although x87 supports transcendental instructions, software library implementation of transcendental function can be faster in many cases.
- x87 supports 80-bit precision, double extended floating-point. SSE support a maximum of 32-bit precision. SSE2 supports a maximum of 64-bit precision.
- Scalar floating-point registers may be accessed directly, avoiding FXCH and top-of-stack restrictions.
- The cost of converting from floating-point to integer with truncation is significantly lower with Intel SSE and Intel SSE2 in the processors based on Intel NetBurst microarchitecture than with either changes to the rounding mode or the sequence prescribed in the Example 3-53.

Assembly/Compiler Coding Rule 55. (M impact, M generality) *Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Most SSE2 arithmetic operations have shorter latency then their X87 counterpart and they eliminate the overhead associated with the management of the X87 register stack.*

3.9.4.1 Scalar Intel® SSE/Intel® SSE2

In code sequences that have conversions from floating-point to integer, divide single-precision instructions, or any precision change, x87 code generation from a compiler typically writes data to memory in single-precision and reads it again in order to reduce precision. Using Intel SSE/Intel SSE2 scalar code instead of x87 code can generate a large performance benefit using Intel NetBurst microarchitecture and a modest benefit on Intel Core Solo and Intel Core Duo processors.

Recommendation: Use the compiler switch to generate scalar floating-point code using XMM rather than x87 code.

When working with Intel SSE/Intel SSE2 scalar code, pay attention to the need for clearing the content of unused slots in an XMM register and the associated performance impact. For example, loading data from memory with MOVSS or MOVSD causes an extra micro-op for zeroing the upper part of the XMM register.

3.9.4.2 Transcendental Functions

If an application needs to emulate math functions in software for performance or other reasons (see Section 3.9.1, "Guidelines for Optimizing Floating-Point Code"), it may be worthwhile to inline math library calls because the CALL and the prologue/epilogue involved with such calls can significantly affect the latency of operations.

3.10 MAXIMIZING PCIE PERFORMANCE

PCIe performance can be dramatically impacted by the size and alignment of upstream reads and writes (read and write transactions issued from a PCIe agent to the host's memory). As a general rule, the best performance, in terms of both bandwidth and latency, is obtained by aligning the start addresses of upstream reads and writes on 64-byte boundaries and ensuring that the request size is a multiple of 64-bytes, with modest further increases in bandwidth when larger multiples (128, 192, 256 bytes) are employed. In particular, a partial write will cause a delay for the following request (read or write).

A second rule is to avoid multiple concurrently outstanding accesses to a single cache line. This can result in a conflict which in turn can cause serialization of accesses that would otherwise be pipelined, resulting in higher latency and/or lower bandwidth. Patterns that violate this rule include sequential accesses (reads or writes) that are not a multiple of 64-bytes, as well as explicit accesses to the same cache line address. Overlapping requests—those with different start addresses but with request lengths that result in overlap of the requests—can have the same effect. For example, a 96-byte read of address 0x00000200 followed by a 64-byte read of address 0x00000240 will cause a conflict—and a likely delay—for the second read.

Upstream writes that are a multiple of 64-byte but are non-aligned will have the performance of a series of partial and full sequential writes. For example, a write of length 128-byte to address 0x00000070 will perform similarly to 3 sequential writes of lengths 16, 64, and 48 to addresses 0x00000070, 0x00000080, and 0x00000100, respectively.

For PCIe cards implementing multi-function devices, such as dual or quad port network interface cards (NICs) or dual-GPU graphics cards, it is important to note that non-optimal behavior by one of those devices can impact the bandwidth and/or latency observed by the other devices on that card. With respect to the behavior described in this section, all traffic on a given PCIe port is treated as if it originated from a single device and function.

For the best PCIe bandwidth:

1. Align start addresses of upstream reads and writes on 64-byte boundaries.
2. Use read and write requests that are a multiple of 64-bytes.
3. Eliminate or avoid sequential and random partial line upstream writes.
4. Eliminate or avoid conflicting upstream reads, including sequential partial line reads.

Techniques for avoiding performance pitfalls include cache line aligning all descriptors and data buffers, padding descriptors that are written upstream to 64-byte alignment, buffering incoming data to achieve larger upstream write payloads, allocating data structures intended for sequential reading by the PCIe device in such a way as to enable use of (multiple of) 64-byte reads. The negative impact of unoptimized reads and writes depends on the specific workload and the microarchitecture on which the product is based.

3.10.1 Optimizing PCIe Performance for Accesses Toward Coherent Memory and MMIO Regions (P2P)

In order to maximize performance for PCIe devices in the processors listed in Table 3-7 the software should determine whether the accesses are toward coherent (system) memory or toward MMIO regions (P2P access to other devices). If the access is toward MMIO region, then software can command HW to set the RO bit in the TLP header, as this would allow hardware to achieve maximum throughput for these types of accesses. For accesses toward coherent memory, software can command HW to clear the RO bit in the TLP header (no RO), as this would allow hardware to achieve maximum throughput for these types of accesses.

Table 3-7. Intel Processor CPU RP Device IDs for Processors Optimizing PCIe Performance

Processor	CPU RP Device IDs
Intel® Xeon processors based on Broadwell microarchitecture	6F01H-6F0EH
Intel® Xeon processors based on Haswell microarchitecture	2F01H-2F0EH

3.11 SCALABILITY WITH CONTENDED LINE ACCESS IN 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS

A two-socket system like that found in the Sapphire Rapids microarchitecture can have up to 224 (2 sockets x 56 cores/socket x 2 threads/core) hardware threads. Scalability and performance bottlenecks may happen when all of these hardware threads compete for the same address.

3.11.1 Causes of Performance Bottlenecks

When multiple hardware threads go after the same address (for example, AA), this address is queued in the Ingress Queue, with one entry for each hardware thread. Due to the resource limitation of the Ingress Queue, the CPU core is throttled to slow the rate of requests when this queue overflows. This usually occurs with contention for a lock.

3.11.2 Performance Bottleneck Detection

When multiple cores are contending on the same lock, several outstanding requests are mapped to that same address. The Phys_addr_match event can count as such an event. This CHA event increments by one every other cycle when there is more than one outstanding request to the same address.

Here are the PMU event id and Umask for the 2 CHA events that are very useful for detecting contention:

1. Phys_addr_match event:	Event id: 0x19, Umask: 0x80
2. CHA_clockticks event:	Event id: 0x01, Umask: 0x01

These events have to be measured on a per-CHA basis, and if the ratio of the counts between phys_addr_match to CHA_clockticks is more than 0.15 on any CHA that indicates > 30% of the CHA cycles (2x the ratio as this event can count only once every two cycles) are spent with multiple requests outstanding to the same address.

Here is the recipe to measure these events with Linux Perf:

```
$ sudo perf stat -a -e 'uncore_cha/event=0x19,umask=0x80/,uncore_cha/event=0x1,umask=0x1/' --per-socket
--no-merge -- sleep 30
```

Once confirmed that the ratio of phys_addr_match events to the CHA clockticks is more than 0.15, the next step is figuring out where this may be happening in the code. Intel CPUs provide a PMU mechanism wherein a load operation is randomly selected and tracked through completion, and the true latency is recorded if it is over a given threshold. The threshold value is specified in cycles and must be in the power of 2. In the following "perf mem record" command, define a command to sample all loads that take more than 128 cycles to complete.

```
$ sudo perf mem record -a --ldlat 128 sleep 1
```

Once the above data is collected, execute the following command to process the data collected:

```
$ sudo perf mem report
```

Information similar to the table below will be generated. Such information will include details on hot loads along with data linear address and the actual latency that the load experienced. This can be used to identify the necessary fixes to the code.

Table 3-8. Samples: 365K of Events 'anon group{cpu/mem-loads-aux/,cpu/mem-loads,ldat=128/pp}', Event Count (a--r0x): 67900852

Overhead	Samples	Local Weight	Memory Access	Symbol	Shared Object	Data Symbol	Data Object	Snoop	TLB Access	Locked	Blocked	Local INSTR Latency
0.22% 0.07%	1	1 38060	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	47251
0.18% 0.06%	1	1 31338	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	40411
0.17% 0.06%	1	1 29572	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	36652

3.11.3 Solutions for Performance Bottlenecks

The following is a list of suggested solutions:

1. Run multiple instances of the workload with a scale-out approach instead of a single instance with scale-up so that the contention for per instance hot variables (including locks) is reduced.
2. Guard the cmpxchg by checking that the destination memory is expected with a load, test, and branch beforehand.

3. Implement a backoff mechanism so that the cmpxchg is issued less. For example, in locks, exponential backoff is a common and effective method to prevent all cores from being in lockstep. In the case of contention for a lock, checking to see if it is accessible by a load before trying to write to it through a cmpxchg will help.

The code in Example 3-54 provides an example:

Example 3-54. Locking Algorithm for the Sapphire Rapids Microarchitecture

```
 lock_loop:
 while (lock is not free) // just a load operation
 execute pause;

// now the lock is free, so try to acquire it.
 Exponential Backoff spin // so all the cores don't come back at the same time
 Execute cmpxchg on the lock
 if the lock is not successfully acquired, goto lock_loop
```

Additionally, as the core counts continue to increase, exploring other algorithmic fixes that dissolve or reduce contention on memory variables (including locks) is essential. For example, instead of frequently updating a hot statistical variable from all threads, consider updating a copy of it per thread (without contention) and later aggregate the updated per-thread copies on a less frequent basis or use some existing atomic-free concurrency methods such as rseq[1]. As another example, restructure locking algorithms to use hierarchical locking when excessive contention is detected on a global lock.

3.11.4 Case Study: SysBench/MariaDB Metric CHA % Cycles Fast Asserted

With SysBench/MariaDB 10.3.34[2], the workload's throughput drops as the number of threads increases. Another metric we can use is the CHA% Cycles Fast Asserted. It is a signal to slow down the cores when the Ingress Queue fills up. This is another way to identify scalability issues. The graph below plots the number of active client threads representing the work intensity on the horizontal axis. The percentage of Fast Asserts is plotted on the vertical axis.

The baseline case (blue line) had a sharp throughput with increased thread count, as all cores reduced their throughput as they suffered from the increasing percent of Fast Asserts. With the same work distributed instances (red line), Fast asserts dropped. Similarly, with a software fix (gray line), again, the Fast Asserts dropped even though only one instance was in execution.

1. https://git.kernel.org/pub/scm/libs/librseq/librseq.git/tree/doc/man/rseq.2
2. The most current version is MariaDB 10.3.39

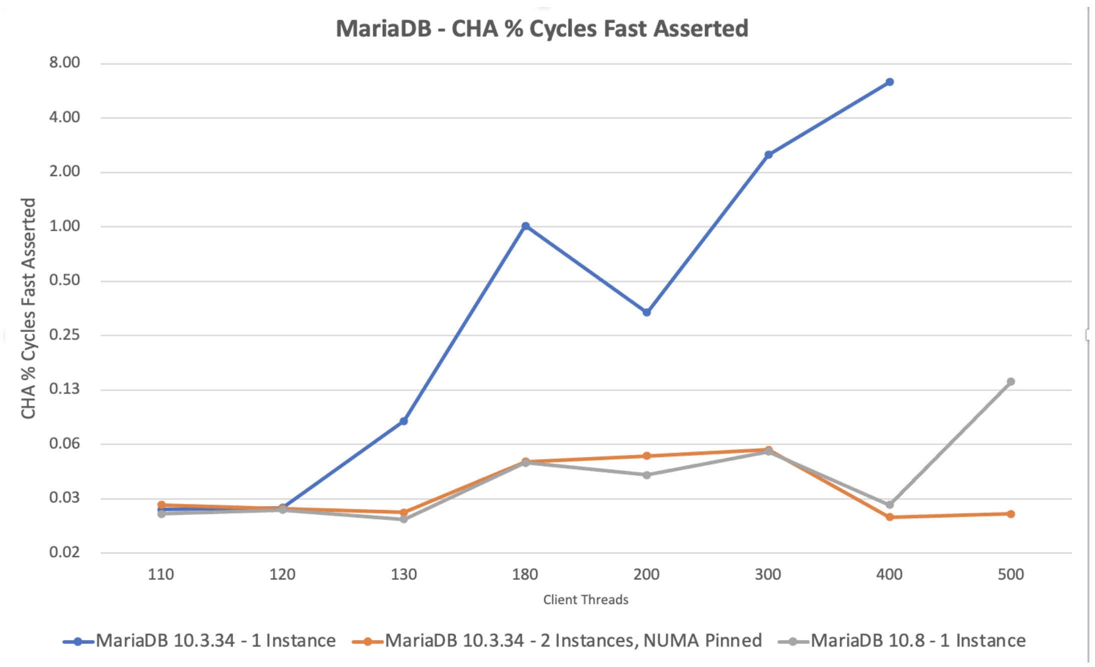


Figure 3-4. MariaDB - CHA % Cycles Fast Asserted

3.11.5 Instruction Sequence Slowdowns

The Golden Cove CPU microarchitecture upon which the Sapphire Rapids microarchitecture is based has increased the cost of mixing Legacy SSE and VEX without clearing the state of upper registers for power efficiency reasons.

3.11.5.1 Causes of Instruction Sequence Slowdowns

The Golden Cove CPU microarchitecture eliminated some hardware speed paths for power efficiency and replaced them with microcode. The instruction sequence in Table 3-9 mixes VEX and Legacy SSE. It has, for example, higher core cycles than on the previous generation Sunny Cove CPU microarchitecture for the Ice Lake version of the 3rd Generation of Intel® Xeon® Scalable processors. The higher core cycles are due to the execution of additional micro-operations.

Table 3-9. Instruction Sequence Mixing VEX on the Sapphire Rapids and Ice Lake Server Microarchitectures

Intel Assembly Code Syntax	Ice lake Server Microarchitecture (Sunny Cove Cores)		Sapphire Rapids Microarchitecture (Golden Cove Cores)	
	Inst Retired	Core Cycles	Inst Retired	Core Cycles

VPXOR XMM3, XMM3, XMM3;
VEXTRACTI128 XMM3, YMM3, 1;
PXOR XMM3, XMM3
	3.00	1	3.00	388.04

3.11.5.2 Detecting Instruction Sequence Slowdowns

The event ASSISTS.SSE_AVX_MIX can be used to determine if there are VEX to legacy SSE transitions. The following Linux perf command-line can be used while the workload is running:

```
$ sudo perf stat -e 'assists.sse_avx_mix'1 <workload>
```

With the Intel® TMA (Topdown Methodology) (there is a metric called Mixing_Vectors which gives the percentage of injected blend uops out of all the uops issued. Usually, a Mixing_Vectors metric over 5% is worth investigating. You can find more details in Appendix B1 of the Optimizations Guide.

3.11.5.3 Fixing Instruction Sequence Slowdowns

The following is a list of suggested solutions:

1. When possible, use VEX-encoded instructions for all the SIMD instructions when possible.
2. Insert a VZEROUPPER to tell the hardware that the state of the higher registers is clean between the VEX and the legacy SSE instructions. Often the best way to do this is to insert a VZEROUPPER before returning from any function that uses VEX (that does not produce a VEX register) and before any call to an unknown function.

VZEROUPPER was inserted in the code sequence below and there are no SSE_AVX_MIX assists. With this change, the Core Cycles do not have a performance inversion relative to the previous generation.

Table 3-10. Fixed Instruction Sequence with Improved Performance on Sapphire Rapids Microarchitecture

Intel Assembly Code Syntax	Ice lake Microarchitecture (Sunny Cove Cores)		Sapphire Rapids Microarchitecture (Golden Cove Cores)		ASSISTS.SSE_AVX_MIX

VPXOR XMM3, XMM3, XMM3;
VEXTRACTI128 XMM3, YMM3, 1;
PXOR XMM3, XMM3
	Inst Retired	Core Cycles	Inst Retired	Core Cycles	

	4.00	2.00	4.00	1.00	0

3.11.6 Misprediction for Branches >2GB

The Golden Cove CPU is a wider machine and might exhibit a higher Top-down Microarchitecture Analysis (TMA) Bad Speculation percentage. See B.1.1 for additional information about TMA. Some sources of Bad Speculation are branch prediction misses. In this case, however, Bad Speculation is due to the wider machine and less efficient branch prediction for certain indirect branches.

3.11.6.1 Causes of Branch Misprediction >2GB

For a near absolute indirect JMP/CALL branch instruction (opcodes FF /4 and FF /2), the branch distance (ADDR_TARGET - ADDR_BRANCH) affects the performance of the branch predictor. The branch predictor uses fewer resources to predict the branch if its distance can be specified with a 32-bit signed displacement (JMP/CALL imm32). If the distance is larger (>2GB), the predictor uses more resources to predict the branch and performance may suffer.

3.11.6.2 Detecting Branch Mispredictions >2GB

You can use the Last Branch Record (LBR) to identify jumps greater than 2GB. The collection of performance analysis tools based on perf on Linux supports this. The following is an example output from the tool. It shows that 21% of the call/jumps of >2GB offset are mispredicted. The histogram of one of the

1. Using upstream perf. If OS doesn't have support for the event use cpu/event=0xc1,umask=0x10,name=assists_sse_avx_mix/

indirect branches at address 0x555555603664 shows that it is to one target and in a library. The profile mask is to use LBR, and the duration is 10 seconds. It does a system-level profile.

```
% ./do.py profile --profile-mask=0x100 -s 10

count of indirect call/jump of >2GB offset: 93200
count of mispredicted indirect call/jump of >2GB offset: 19943
misprediction ratio for indirect branch at address 0x7ffff577eca4: 4.23%
misprediction ratio for indirect branch at address 0x5555556030c4: 32.23%
misprediction ratio for indirect branch at address 0x555555603664: 22.30%
misprediction ratio for indirect branch at address 0x555555603c24: 13.84%
...
indirect_0x555555603664 histogram:
0x7ffff7af2670:  50501 100.0%
```

Figure 3-5. Identifying >2GB Branches

3.11.6.3 Fixing Branch Mispredictions >2GB

Arrange the code so the jumps don't span the >2GB range. This can be done through a variety of approaches:

1. If possible, statically link all the libraries into the executable.
2. For **.text** to library code, use the Glibc environment variable LD_PREFER_MAP_32BIT_EXEC=1 to restrict the addresses into the 4GB range.
3. For dynamically compiled code, keep it close to the .text address or copy the frequently called entries into the dynamically compiled code address region. See the Google V8 Blog.

In a case study with WordPress/PHP running eight containers with and without the 2GB fix, the CPI and performance scores improve by 6%.

Table 3-11. WordPress/PHP Case Study: With and Without a 2GB Fix for Branch Misprediction

		WP4.2 / PHP7.4.29 - NO FIX	WP4.2 / PHP7.4.29 - 2G FIX in Glibc	2G FIX/ NO FIX
Config	Workers	8c x 42	8c x 42	-
	Cores Per socket	56	56	1.00
	Sockets	2	2	1.00
	Total Cores	112	112	1.00
	Total Thread Count	224	224	1.00
Performance	Throughput	1.00	1.06	1.06
	CPI	1.12	1.05	0.96
Path Length	Instructions per Unit of Work	33,789,862.68	33,730,155.10	1.00
Cycles per Transaction	Cycles per Unit of Work	37,803,310.48	35,359,628.33	0.94

3.12 OPTIMIZING COMMUNICATION WITH PCI DEVICES ON INTEL® 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS

The Sapphire Rapids microarchitecture introduced a new set of instructions designed to optimize communication between SW running on IA cores and PCI devices on the platform.

3.12.1 Signaling Devices with Direct Move

Most software-to-device interaction follows a producer-to-consumer relationship where the software creates work for the device and then signals it to inform the device that work is available. Descriptor rings are the ubiquitous pattern here and once descriptors are added to the ring, the signal (or "doorbell") consists of an update to the tail pointer register on the device. This is a write to an MMIO-mapped BAR register.

Such writes tend to be relatively expensive operations –the latency to complete the write to the device is high relative to the CPU operating speed. Since writes are ordered by default, this creates a bubble during which subsequent writes cannot be drained from store buffers. Signaling can therefore affect performance via store backpressure.

As a result, some software libraries avoid frequent signaling by batching relatively large quantities of work descriptors with each doorbell update. However, this is not always possible, and it introduces latency.

The Sapphire Rapids microarchitecture introduces "Direct Store" instructions to optimize signaling; there are two instructions in the family:

- MOVDIRI: 4/8B direct store.
- MOVDIR64B: 64B atomic direct copy.

Direct Stores are weakly ordered (like non-temporal or USWC-mapped memory writes) regardless of the underlying memory type (which is usually UC for MMIO-mapped locations). Since they do not order subsequent writes the performance issue described above does not occur.

Since they are intended for signaling, direct stores will never combine with other stores to the same address as can happen with non-temporal or USWC writes. Each write is guaranteed to occur as issued. In the case of MOVDIR64B, the full 64B will be delivered as a single write to the device. This is the only ISA that carries an architectural guarantee of >8B atomicity.

These instructions benefit from the fact that signaling use cases typically do not care if subsequent writes are observed before the doorbell itself because the ordering is relaxed. However, since typically the door-bell must not be observable before earlier writes (such writes are creating the work descriptors), SW should insert a store fence immediately before the direct store.

Having a fence before the direct store does not normally limit performance– except when many direct stores are issued. If there is an SFENCE before each, the fence on direct store N+1 imposes an order on direct store N, which can remove some of the benefits. The guideline is to avoid this where possible. One technique that may work if multiple doorbells to different addresses are being issued (such as for a NIC driver that is handling multiple descriptor rings), is to group the direct stores to different locations together and insert a single SFENCE before the group.

It is also worth noting that the device write latency can vary widely with the address being written. This is especially true on large CPUs implemented as multiple tiles. So if SW has the luxury of choosing between multiple addresses, it is possible to envisage adaptive schemes that "match" an address to a SW thread (especially if that thread is pinned to a single core) by selecting the best performing such address during an initialization stage.

3.12.1.1 MOVDIR64B: Additional Considerations

As noted above MOVDIR64B is a copy operation; it moves data from one 64B-aligned address to another. Typical usage is that the source address is a memory location, and the destination is MMIO mapped to a device, whereupon it confers the benefits described above. However, since the source data is usually written immediately before the MOVDIR64B, additional considerations include:

- It is unnecessary to fence to ensure the source data is written before the MOVDIR64B since the source data is written to the same address that the MOVDIR64B reads. In some scenarios, no store fence is needed in conjunction with MOVDIR64B. The correct operation of the system depends on being observed before the MOVDIR64B if no other data is written to memory.
- It is critical to allow store forwarding of the source data for the best performance.
- The source data should be aligned to 64B and written at the same granularity that the MOVDIR64B reads. For the Sapphire Rapids microarchitecture, this is 64B: the source data should, therefore, be written using 64B Intel® AVX-512 Instructions for the best performance.

3.12.1.2 Streaming Data

MOVDIR64B can also be used to stream data to a device by copying a block of memory because it is weakly ordered. This is similar behavior to mapping the destination memory locations as USWC, except:

- The destination address can remain mapped UC.
- The writes are guaranteed to arrive at the device as 64B writes, which is not guaranteed with any other method.

3.13 SYNCHRONIZATION

3.13.1 User-Level Monitor, User-Level MWAIT, and TPAUSE

New instructions for user-level monitor and MWAIT act like legacy monitor and MWAIT instructions with additional functionality identified as the timeout and ring-3 (user space) application support. TPAUSE is similar to legacy pause instruction but is designed to accept time interval and sleep state parameters. User-level MWAIT and TPAUSE support the same C0.1 light sleep and C0.2 deeper sleep states. These instructions are helpful in user space applications that support a busy poll, synchronization, or asynchronous IO, such as waiting for an event. A minor code modification yields power benefits along with low latency wake-up.

3.13.1.1 Checking for User-Level Monitor, MWAIT, and TPAUSE Support

This section describes how to check whether a processor supports user-level monitor, user-level MWAIT, or TPAUSE; if user-level monitor, user-level MWAIT, or TPAUSE instruction is supported, then CPUID. (EAX=07H, ECX=0): ECX [bit 5] is enumerated as 1.

Example 3-55. Identification of WAITPKG with CPUID

```
…identify the existence of cpuid instruction
… ;
… ;
Identify signature is genuine Intel …;
mov eax, 7; Request for feature flags
mov ecx, 0; Request for feature flags
cpuid; 0FH, A2H CPUID instruction
test ecx, 00000020h;
Is waitpkg bit (bit 5) in feature flags equal to 1 jnz Found
```

3.13.1.2 User-Level Monitor, User-Level MWAIT, and TPAUSE Operations

User-level monitor initializes the monitor hardware in such a way that, after execution of the user-level MWAIT, a store to a monitored address acts as a wakeup event. So, the User level monitor and the user-level MWAIT work together to obtain a sleep state. TPAUSE is a single instruction request to enter one of the same two sleep states for a defined time

There are possibilities of a "false wake-up" because of other events, notably interrupts or timeouts. The application may re-execute user-level MWAIT/TPAUSE if it has been falsely woken. If the application needs to determine the source of the predefined OS sleep wakeup, RFLAGS.CF is set Otherwise it is assumed that the application can detect changes at the monitored address (MWAIT) or poll for activity (TPAUSE).

3.13.1.3 Recommended Usage of Monitor, MWAIT, and TPAUSE Operations

A frequent paradigm in packet processing applications is to have dedicated HW threads polling a NIC receive descriptor ring for ingress traffic. This kind of "busy polling" arrangement wastes energy when the traffic rates are low. Changing the polling loop to perform user-level Monitor/MWAIT on the next descriptor to be written can save substantial power in periods of low traffic. The same scheme could be used with any "work distributor," assigning work by writing to selected memory locations.

Accelerators frequently offload tasks from SW in an asynchronous manner. For example, the Intel® Data Streaming Accelerator (Intel® DSA) performs copy operations and can return the status of the completed operation by writing to memory. If an application uses the user-level monitor/MWAIT at a memory location where the status field will be written, it can be woken when the task is complete. Instead of monitoring, the device may issue an interrupt that can act as a wake-up event.

Alternatively, applications may decide to choose TPAUSE as a wait event. This has the advantage of being independent of the number of event sources.

In all cases, a small change in the user space application is needed to convert a busy poll application to something more energy efficient with low latency wake-up.

Synchronous application: when two hardware threads from the same core use user-level monitor and user-level MWAIT, it can progress effectively as some of the hardware resources are available to the other thread when a hyperthread issues the user-level MWAITs.

To achieve the best performance using user-level monitor and user-level MWAIT:

- The entire contents of monitored locations must be verified after user-level MWAIT to avoid a false wake-up.
- It is the developer's responsibility to check the contents of monitored locations:
  - — Before issuing monitor.
  - — Before issuing user-level MWAIT.
  - — After user-level MWAIT. See Example 3-56.
- If an application expects a store to a monitored location, the timeout value should be as high as it is supported.

Since user-level MWAIT and TPAUSE are a hint to a processor, a user should selectively identify locations in the application.

Example 3-56. Code Snippet in an Asynchronous Example

```
void * m_address;  // it is expected device will update m_address to 1
unsigned char ret;
while (1) {
        if (*m_address != 0) // if device already finished operation, no need to user monitor/user mwait
                break;
        if (*m_address == 0) { // check monitored location before issuing umonitor instruction
                _umonitor (m_address);
                if (*m_address == 0) {         // check monitored location before issuing umwait instruction
                        ret = _umwait(0, 0x186A0);    // some high value in timeout
                }
        }
}
```

4. Updates to Chapter 7

Change bars and **violet** text show changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Optimizing for SIMD Floating-point Applications.

Changes to this chapter:

- Example 7-5 was corrected.
- Example 7-6 was corrected.

CHAPTER 7
OPTIMIZING FOR SIMD FLOATING-POINT APPLICATIONS

This chapter discusses rules for optimizing the single-instruction, multiple-data (SIMD) floating-point instructions available in SSE, SSE2 SSE3, and SSE4.1. The chapter also provides examples illustrating the optimization techniques for single-precision and double-precision SIMD floating-point applications.

7.1 GENERAL RULES FOR SIMD FLOATING-POINT CODE

The rules and suggestions in this section help optimize floating-point code containing SIMD floating-point instructions. Generally, it is essential to understand and balance port utilization to create efficient SIMD floating-point code. Basic rules and suggestions include the following:

- Follow all guidelines in Chapter 3: "General Optimization Guidelines" and Chapter 5: "Coding for SIMD Architectures".
- Mask exceptions to achieve higher performance. When exceptions are unmasked, software performance is slower.
- Utilize the flush-to-zero and denormals-are-zero modes for higher performance to avoid the penalty of dealing with denormals and underflows.
- Use the reciprocal instructions followed by iteration for increased accuracy. These instructions yield reduced accuracy but execute much faster. Note the following:
  - If reduced accuracy is acceptable, use them with no iteration.
  - If near full accuracy is needed, use a Newton-Raphson iteration.
  - If full accuracy is needed, then use divide and square root, which provide more accuracy, but slow down performance.

7.2 PLANNING CONSIDERATIONS

Whether adapting an existing application or creating a new one, using SIMD floating-point instructions to achieve optimum performance gain requires programmers to consider several issues. When choosing candidates for optimization, look for code segments that are computationally intensive and floating-point intensive. Also, consider efficient use of the cache architecture.

The sections that follow answer the questions that should be raised before implementation:

- Can data layout be arranged to increase parallelism or cache utilization?
- Which part of the code benefits from SIMD floating-point instructions?
- Is the current algorithm the most appropriate for SIMD floating-point instructions?
- Is the code floating-point intensive?
- Do single-precision floating-point or double-precision floating-point computations provide enough range and precision?
- Does the result of computation affected by enabling flush-to-zero or denormals-to-zero modes?
- Is the data arranged for efficient utilization of the SIMD floating-point registers?
- Is this application targeted for processors without SIMD floating-point instructions?

See Section 5.2, “Considerations for Code Conversion to SIMD Programming.”

7.3 USING SIMD FLOATING-POINT WITH X87 FLOATING-POINT

Because the XMM registers used for SIMD floating-point computations are separate registers and are not mapped to the existing x87 floating-point stack, SIMD floating-point code can be mixed with x87 floating-point or 64-bit SIMD integer code.

With Intel Core microarchitecture, 128-bit SIMD integer instructions provide substantially higher efficiency than 64-bit SIMD integer instructions. Software should favor using SIMD floating-point and integer SIMD instructions with XMM registers where possible.

7.4 SCALAR FLOATING-POINT CODE

SIMD floating-point instructions operate only on the lowest order element in the SIMD register. These instructions are known as scalar instructions. They allow the XMM registers to be used for general-purpose floating-point computations.

In terms of performance, scalar floating-point code can be equivalent to or exceed x87 floating-point code and has the following advantages:

- SIMD floating-point code uses a flat register model, whereas x87 floating-point code uses a stack model. Using scalar floating-point code eliminates the need to use FXCH instructions. These have performance limits on the Intel Pentium 4 processor.
- Mixing with MMX technology code without penalty.
- Flush-to-zero mode.
- Shorter latencies than x87 floating-point.

When using scalar floating-point instructions, it is unnecessary to ensure that the data appears in vector form. However, the optimizations for alignment, scheduling, instruction selection, and other optimizations covered in Chapter 3 and Chapter 5 should be observed.

7.5 DATA ALIGNMENT

SIMD floating-point data is 16-byte aligned. Referencing unaligned 128-bit SIMD floating-point data will result in an exception unless MOVUPS or MOVUPD (move unaligned packed single or unaligned packed double) is used. The unaligned instructions used on aligned or unaligned data will also suffer a performance penalty relative to aligned accesses.

See also: Section 5.4, “Stack and Data Alignment.”

7.5.1 Data Arrangement

Because SSE and SSE2 incorporate SIMD architecture, arranging data to use the SIMD registers fully produces optimum performance. This implies contiguous data for processing, which leads to fewer cache misses. Correct data arrangement can quadruple data throughput using SSE, or double throughput when using SSE2. Performance gains can occur because four data elements can be loaded with 128-bit load instructions into XMM registers using SSE (MOVAPS). Similarly, two data elements can be loaded with 128-bit load instructions into XMM registers using SSE2 (MOVAPD).

Refer to Section 5.4, “Stack and Data Alignment,” for data arrangement recommendations. Duplicating and padding techniques overcome misalignment problems that in some data structures and arrangements. This increases the data space but avoids penalties for misaligned data access.

For some applications (3D geometry, for example), traditional data arrangement requires some changes to use the SIMD registers and parallel techniques fully. Traditionally, the data layout has been an array of structures (AoS). A new data layout has been proposed to fully use the SIMD registers in such applications: a structure of arrays (SoA) resulting in more optimized performance.

7.5.1.1 Vertical versus Horizontal Computation

Most floating-point arithmetic instructions in SSE/SSE2 provide a more significant performance gain on vertical data processing for parallel data elements. This means that each element of the destination results from an arithmetic operation performed from the source elements in the same vertical position (Figure 7-1).

To supplement these homogeneous arithmetic operations on parallel data elements, SSE and SSE2 provide data movement instructions (e.g., SHUFPS, UNPCKLPS, UNPCKHPS, MOVLHPS, MOVHLPS, etc.) that facilitate moving data elements horizontally.

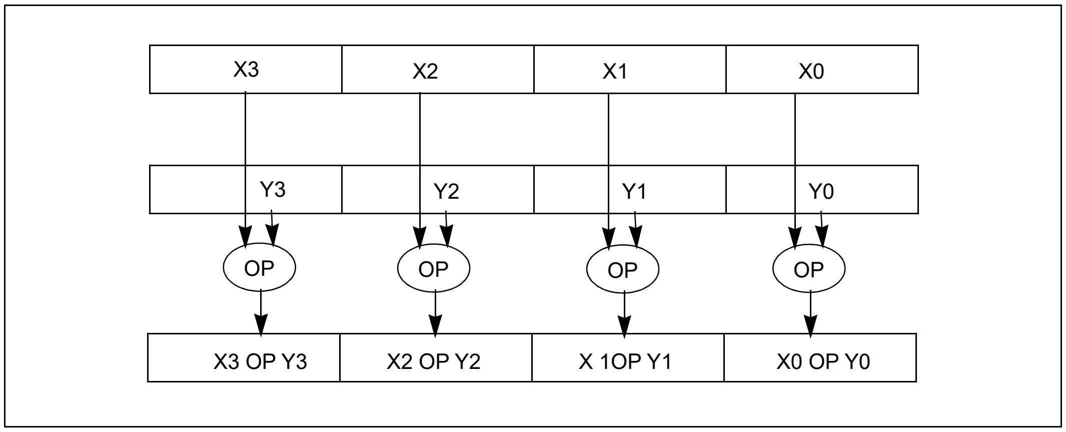


Figure 7-1. Homogeneous Operation on Parallel Data Elements

The organization of structured data significantly impacts SIMD programming efficiency and performance. This can be illustrated using two common type of data structure organizations:

- Array of Structure (AoS): This refers to arranging an array of data structures. Within the data structure, each member is a scalar. This is shown in Figure 7-2. Typically, a repetitive computation sequence is applied to each element of an array, i.e., a data structure. The computational sequence for the scalar members of the structure is likely to be non-homogeneous within each iteration. AoS is generally associated with a horizontal computation model.

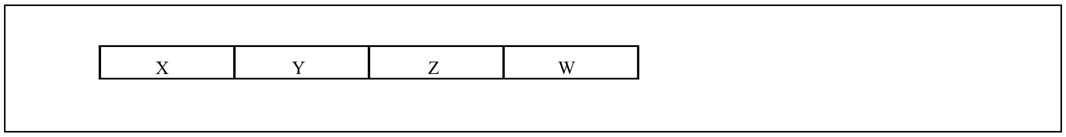


Figure 7-2. Horizontal Computation Model

- Structure of Array (SoA): Here, each member of the data structure is an array. Each element of the array is a scalar. This is shown in Table 7-1. The repetitive computational sequence is applied to scalar elements and homogeneous operation can be easily achieved across consecutive iterations within the same structural member. Consequently, SoA is generally amenable to the vertical computation model.

Table 7-1. SoA Form of Representing Vertices Data

Vx array	X1	X2	X3	X4	.....	Xn
Vy array	Y1	Y2	Y3	Y4	.....	Yn
Vz array	Z1	Z2	Z3	Y4	.....	Zn
Vw array	W1	W2	W3	W4	.....	Wn

SIMD instructions with vertical computation on the SoA arrangement can achieve higher efficiency and performance than AoS and horizontal computation. This can be seen with dot-product operation on vectors. The dot product operation on the SoA arrangement is shown in Figure 7-3.

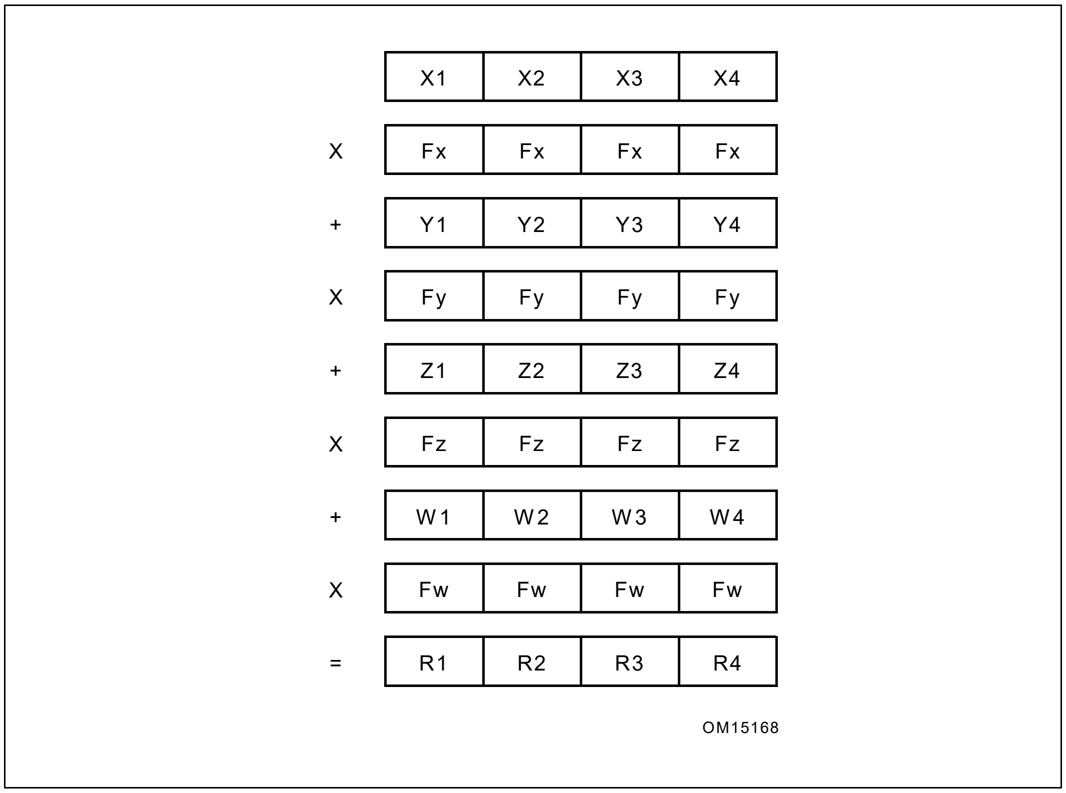


Figure 7-3. Dot Product Operation

Example 7-1 shows how one result would be computed for seven instructions if the data were organized as AoS and using SSE alone: four results would require 28 instructions.

Example 7-1. Pseudocode for Horizontal (xyz, AoS) Computation

```
mulps      ; x*x', y*y', z*z'
movaps     ; reg->reg move, since next steps overwrite
shufps     ; get b,a,d,c from a,b,c,d
addps      ; get a+b,a+b,c+d,c+d
movaps     ; reg->reg move
shufps     ; get c+d,c+d,a+b,a+b from prior addps
addps      ; get a+b+c+d,a+b+c+d,a+b+c+d,a+b+c+d
```

Now consider the case when the data is organized as SoA. Example 7-2 demonstrates how four results are computed for five instructions.

Example 7-2. Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation

```
mulps   ; x*x' for all 4 x-components of 4 vertices
mulps   ; y*y' for all 4 y-components of 4 vertices
mulps   ; z*z' for all 4 z-components of 4 vertices
addps   ; x*x' + y*y'
addps   ; x*x'+y*y'+z*z'
```

For the most efficient use of the four component-wide registers, reorganizing the data into the SoA format yields increased throughput and hence much better performance for the instructions used.

This simple example shows that vertical computation can yield 100% use of the available SIMD registers to produce four results. Note that results may vary for other situations. Suppose the data structures are represented in a format that is not "friendly" to vertical computation. In that case, it can be rearranged "on the fly" to facilitate better utilization of the SIMD registers. This operation is referred to as a "swizzling" operation. The reverse operation is referred to as "deswizzling."

7.5.1.2 Data Swizzling

Swizzling data from SoA to AoS format can apply to multiple application domains, including 3D geometry, video and imaging. Two different swizzling techniques can be adapted to handle floating-point and integer data. Example 7-3 illustrates a swizzle function that uses SHUFPS, MOVLHPS, and MOVHLPS instructions.

Example 7-3. Swizzling Data Using SHUFPS, MOVLHPS, MOVHLPS

```
typedef struct _VERTEX_AOS {
    float x, y, z, color;
} Vertex_aos;                          // AoS structure declaration
typedef struct _VERTEX_SOA {
    float x[4], float y[4], float z[4];
    float color[4];
} Vertex_soa;                          // SoA structure declaration
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// SWIZZLE XYZW --> XXXX
asm {
        mov  rbx, in                   // get structure addresses
        mov  rdx, out
        movaps  xmm1, [rbx ]           // w0 z0 y0 x0
        movaps  xmm2, [rbx + 16]       // w1 z1 y1 x1
        movaps  xmm3, [rbx + 32]       // w2 z2 y2 x2
        movaps  xmm4, [rbx + 48]       // w3 z3 y2 x3
        movaps  xmm7, xmm4             // xmm7= w3 z3 y3 x3
        movhlps  xmm7, xmm3            // xmm7= w3 z3 w2 z2
        movaps  xmm6, xmm2             // xmm6= w1 z1 y1 x1
        movlhps  xmm3, xmm4            // xmm3= y3 x3 y1 x1
        movhlps  xmm2, xmm1            // xmm2= w1 z1 w0 z0
        movlhps  xmm1, xmm6            // xmm1= y1 x1 y0 x0
```

Example 7-3. Swizzling Data (Contd.)Using SHUFPS, MOVLHPS, MOVHLPS (Contd.)

```
        movaps   xmm6, xmm2                // xmm6= w1 z1 w0 z0
        movaps   xmm5, xmm1                // xmm5= y1 x1 y0 x0
        shufps   xmm2, xmm7, 0xDDh         // xmm2= w3 w2 w1 w0 => W
        shufps   xmm1, xmm3, 0x88h         // xmm1= x3 x2 x1 x0 => X
        shufps   xmm5, xmm3, 0xDDh         // xmm5= y3 y2 y1 y0 => Y
        shufps   xmm6, xmm7, 0x88h         // xmm6= z3 z2 z1 z0 => Z

        movaps   [rdx], xmm1               // store X
        movaps   [rdx+16], xmm5            // store Y
        movaps   [rdx+32], xmm6            // store Z
        movaps   [rdx+48], xmm2            // store W
    }
}
```

Example 7-4 shows a similar data-swizzling algorithm using SIMD instructions in the integer domain.

Example 7-4. Swizzling Data Using UNPCKxxx Instructions

```
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// SWIZZLE XYZW --> XXXX
asm {
        mov rbx, in                        // get structure addresses
        mov rdx, out

        movdqa       xmm1, [rbx + 0*16]    //w0 z0 y0 x0
        movdqa       xmm2, [rbx + 1*16]    //w1 z1 y1 x1
        movdqa       xmm3, [rbx + 2*16]    //w2 z2 y2 x2
        movdqa       xmm4, [rbx + 3*16]    //w3 z3 y3 x3
        movdqa       xmm5, xmm1
        punpckldq    xmm1, xmm2            // y1 y0 x1 x0
        punpckhdq    xmm5, xmm2            // w1 w0 z1 z0
        movdqa       xmm2, xmm3
        punpckldq    xmm3, xmm4            // y3 y2 x3 x2
        punpckhdq    xmm2, xmm4            // w3 w2 z3 z2
        movdqa       xmm4, xmm1
        punpcklqdq   xmm1, xmm3            // x3 x2 x1 x0
        punpckhqdq   xmm4, xmm3            // y3 y2 y1 y0
        movdqa       xmm3, xmm5
        punpcklqdq   xmm5, xmm2            // z3 z2 z1 z0
        punpckhqdq   xmm3, xmm2            // w3 w2 w1 w0

        movdqa       [rdx+0*16], xmm1      //x3 x2 x1 x0
        movdqa       [rdx+1*16], xmm4      //y3 y2 y1 y0
        movdqa       [rdx+2*16], xmm5      //z3 z2 z1 z0
        movdqa       [rdx+3*16], xmm3      //w3 w2 w1 w0
}
```

The technique in Example 7-3 (loading 16 bytes, using SHUFPS and copying halves of XMM registers) is preferable over an alternate approach of loading halves of each vector using MOVLPS/MOVHPS on newer microarchitectures. This is because loading 8 bytes using MOVLPS/MOVHPS can create code dependency and reduce the throughput of the execution engine.

The performance considerations of Example 7-3, and Example 7-4 often depend on each microarchitecture's characteristics. For example, in Intel Core microarchitecture, executing a SHUFPS tend to be slower than a PUNPCKxxx instruction. In Enhanced Intel Core microarchitecture, SHUFPS and PUNP-CKxxx instruction execute with one cycle throughput due to the 128-bit shuffle execution unit. The next important consideration is that only one port can execute PUNPCKxxx rather than MOVLHPS/MOVHLPS executing on multiple ports. The performance of both techniques improves on Intel Core microarchitecture over previous microarchitectures due to 3 ports for executing SIMD instructions. Both techniques further improve the Enhanced Intel Core microarchitecture due to the 128-bit shuffle unit.

7.5.1.3 Data Deswizzling

In the deswizzle operation, we want to arrange the SoA format back into AoS format so the XXXX, YYYY, and ZZZZ are rearranged and stored in memory as XYZ. Example 7-5 illustrates one deswizzle function for floating-point data.

Example 7-5. Deswizzling Single-Precision SIMD Data

```
void deswizzle_asm(Vertex_soa *in, Vertex_aos *out)
{
  __asm {
    mov         rcx, in              // load structure addresses
    mov         rdx, out
    movaps      xmm0, [rcx]          //x3 x2 x1 x0
    movaps      xmm1, [rcx + 16]     //y3 y2 y1 y0
    movaps      xmm2, [rcx + 32]     //z3 z2 z1 z0
    movaps      xmm3, [rcx + 48]     //w3 w2 w1 w0

    movaps      xmm5, xmm0
    movaps      xmm7, xmm2
    unpcklps    xmm0, xmm1           // y1 x1 y0 x0
    unpcklps    xmm2, xmm3           // w1 z1 w0 z0
    movdqa      xmm4, xmm0
    movlhps     xmm0, xmm2           // w0 z0 y0 x0
    movhlps     xmm2, xmm4           // w1 z1 y1 x1

    unpckhps    xmm5, xmm1           // y3 x3 y2 x2
    unpckhps    xmm7, xmm3           // w3 z3 w2 z2
    movdqa      xmm6, xmm5
    movlhps     xmm5, xmm7           // w2 z2 y2 x2
    movhlps     xmm7, xmm6           // w3 z3 y3 x3
    movaps      [rdx+0*16], xmm0     //w0 z0 y0 x0
    movaps      [rdx+1*16], xmm2     //w1 z1 y1 x1
    movaps      [rdx+2*16], xmm5     //w2 z2 y2 x2
    movaps      [rdx+3*16], xmm7     //w3 z3 y3 x3
   }
}
```

Example 7-6 shows a similar deswizzle function using SIMD integer instructions. Both techniques demonstrate loading 16 bytes and performing horizontal data movement in registers. This approach is likely more efficient than alternative techniques of storing 8-byte halves of XMM registers using MOVLPS and MOVHPS.

Example 7-6. Deswizzling Data Using SIMD Integer Instructions

```
void deswizzle_rgb(Vertex_soa *in, Vertex_aos *out)
{
        //---deswizzling---rgb---
        // assume: xmm0=rrrr, xmm1=gggg, xmm2=bbbb, xmm3=aaaa
   mov          rcx, in                       // load structure addresses
   mov          rdx, out

   movdqa       xmm0, [rcx]                   // load r4 r3 r2 r1 => xmm0
   movdqa       xmm1, [rcx+16]                // load g4 g3 g2 g1 => xmm1
   movdqa       xmm2, [rcx+32]                // load b4 b3 b2 b1 => xmm2
   movdqa       xmm3, [rcx+48]                // load a4 a3 a2 a1 => xmm3

// Start deswizzling here
   movdqa       xmm5, xmm0
   movdqa       xmm7, xmm2
   punpckldq    xmm0, xmm1                    //g2 r2 g1 r1
   punpckldq    xmm2, xmm3                    //a2 b2 a1 b1
   movdqa       xmm4, xmm0
   punpcklqdq   xmm0, xmm2                    // a1 b1 g1 r1 => v1
   punpckhqdq   xmm4, xmm2                    // a2 b2 g2 r2 => v2
   punpckhdq    xmm5, xmm1                    // g4 r4 g3 r3
   punpckhdq    xmm7, xmm3                    // a4 b4 a3 b3
   movdqa       xmm6, xmm5
   punpcklqdq   xmm5, xmm7                    // a3 b3 g3 r3 => v3
   punpckhqdq   xmm6, xmm7                    // a4 b4 g4 r4 => v4
   movdqa       [rdx], xmm0                   // v1
   movdqa       [rdx+16], xmm4                // v2
   movdqa       [rdx+32], xmm5                // v3
   movdqa       [rdx+48], xmm6                // v4

// DESWIZZLING ENDS HERE
   }
}
```

7.5.1.4 Horizontal ADD Using SSE

Although vertical computations generally use SIMD performance better than horizontal computations, code must use a horizontal operation in some cases.

MOVLHPS/MOVHLPS and shuffle can be used to sum data horizontally. For example, starting with four 128-bit registers, to sum up each register horizontally while having the final results in one register, use the MOVLHPS/MOVHLPS to align the upper and lower parts of each register. This allows you to use a vertical add. With the resulting partial horizontal summation, full summation follows easily.

Figure 7-4 presents a horizontal add using MOVHLPS/MOVLHPS. Example 7-7 and Example 7-8 provide the code for this operation.

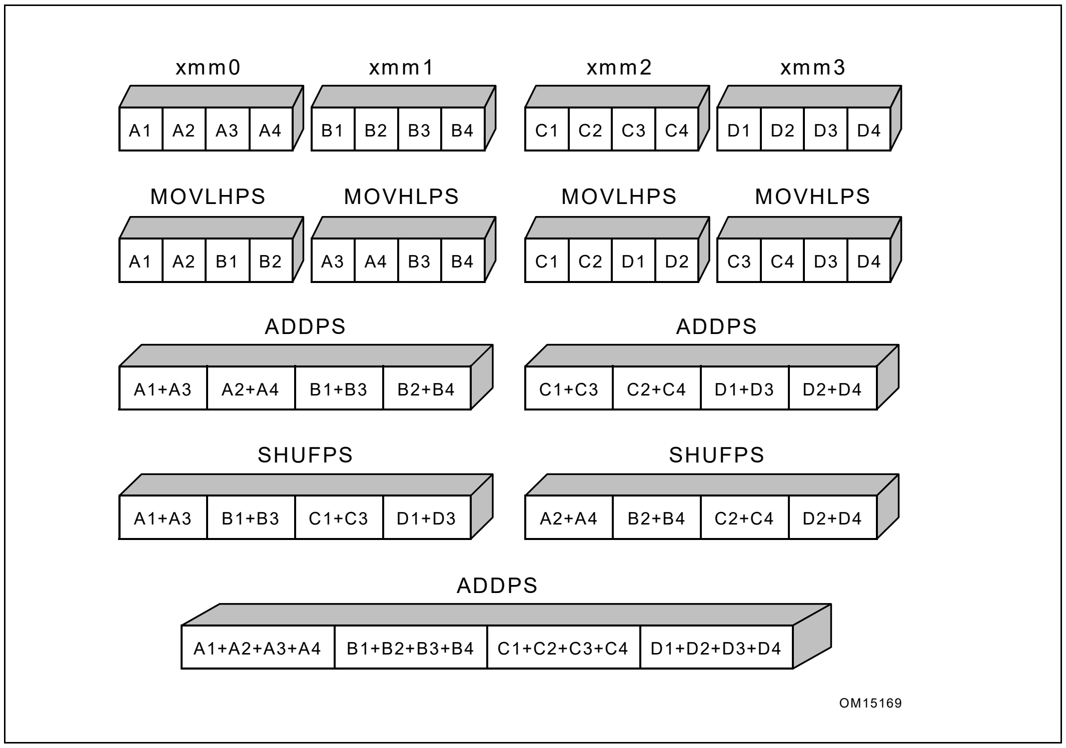


Figure 7-4. Horizontal Add Using MOVHLPS/MOVLHPS

Example 7-7. Horizontal Add Using MOVHLPS/MOVLHPS

```
void horiz_add(Vertex_soa *in, float *out) {
  __asm {
    mov         rcx, in                  // load structure addresses
    mov         rdx, out
    movaps      xmm0, [rcx]              // load A1 A2 A3 A4 => xmm0
    movaps      xmm1, [rcx+16]           // load B1 B2 B3 B4 => xmm1
    movaps      xmm2, [rcx+32]           // load C1 C2 C3 C4 => xmm2
    movaps      xmm3, [rcx+48]           // load D1 D2 D3 D4 => xmm3
 // START HORIZONTAL ADD
    movaps      xmm5, xmm0               // xmm5= A1,A2,A3,A4
    movlhps     xmm5, xmm1               // xmm5= A1,A2,B1,B2
    movhlps     xmm1, xmm0               // xmm1= A3,A4,B3,B4
    addps       xmm5, xmm1               // xmm5= A1+A3,A2+A4,B1+B3,B2+B4
    movaps      xmm4, xmm2
    movlhps     xmm2, xmm3               // xmm2= C1,C2,D1,D2
    movhlps     xmm3, xmm4               // xmm3= C3,C4,D3,D4
    addps       xmm3, xmm2               // xmm3= C1+C3,C2+C4,D1+D3,D2+D4
    movaps      xmm6, xmm3               // xmm6= C1+C3,C2+C4,D1+D3,D2+D4
    shufps      xmm3, xmm5, 0xDD         //xmm6=A1+A3,B1+B3,C1+C3,D1+D3
    shufps      xmm5, xmm6, 0x88         // xmm5= A2+A4,B2+B4,C2+C4,D2+D4
    addps       xmm6, xmm5               // xmm6= D,C,B,A
```

Example 7-7. Horizontal Add Using MOVHLPS/MOVLHPS (Contd.)

```
// END HORIZONTAL ADD
  movaps     [rdx], xmm6
 }
}
```

Example 7-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS

```
void horiz_add_intrin(Vertex_soa *in, float *out)
{
 __m128 v, v2, v3, v4;
 __m128 tmm0,tmm1,tmm2,tmm3,tmm4,tmm5,tmm6;
                                                  // Temporary variables
    tmm0 = _mm_load_ps(in->x);                    // tmm0 = A1 A2 A3 A4
    tmm1 = _mm_load_ps(in->y);                    // tmm1 = B1 B2 B3 B4
    tmm2 = _mm_load_ps(in->z);                    // tmm2 = C1 C2 C3 C4
    tmm3 = _mm_load_ps(in->w);                    // tmm3 = D1 D2 D3 D4
    tmm5 = tmm0;                                  // tmm0 = A1 A2 A3 A4
    tmm5 = _mm_movelh_ps(tmm5, tmm1);             // tmm5 = A1 A2 B1 B2
    tmm1 = _mm_movehl_ps(tmm1, tmm0);             // tmm1 = A3 A4 B3 B4
    tmm5 = _mm_add_ps(tmm5, tmm1);                // tmm5 = A1+A3 A2+A4 B1+B3 B2+B4
    tmm4 = tmm2;
    tmm2 = _mm_movelh_ps(tmm2, tmm3);             // tmm2 = C1 C2 D1 D2
    tmm3 = _mm_movehl_ps(tmm3, tmm4);             // tmm3 = C3 C4 D3 D4
    tmm3 = _mm_add_ps(tmm3, tmm2);                // tmm3 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = tmm3;                                  // tmm6 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = _mm_shuffle_ps(tmm3, tmm5, 0xDD);      // tmm6 = A1+A3 B1+B3 C1+C3 D1+D3
    tmm5 = _mm_shuffle_ps(tmm5, tmm6, 0x88);      // tmm5 = A2+A4 B2+B4 C2+C4 D2+D4
    tmm6 = _mm_add_ps(tmm6, tmm5);                // tmm6 = A1+A2+A3+A4 B1+B2+B3+B4
                                                  // C1+C2+C3+C4 D1+D2+D3+D4
  _mm_store_ps(out, tmm6);
}
```

7.5.2 Use of CVTTPS2PI/CVTTSS2SI Instructions

The CVTTPS2PI and CVTTSS2SI instructions implicitly encode the truncate/chop rounding mode in the instruction. They take precedence over the rounding mode specified in the MXCSR register. This behavior can eliminate the need to change the rounding mode from round-nearest, to truncate/chop, then return to round-nearest to resume computation.

Avoid frequent changes to the MXCSR register since a penalty associated with writing this register. Typically, when using CVTTPS2P/CVTTSS2SI, rounding control in MXCSR can always be set to round-nearest.

7.5.3 Flush-to-Zero and Denormals-are-Zero Modes

The flush-to-zero (FTZ) and denormals-are-zero (DAZ) modes are incompatible with IEEE Standard 754[1]. They are provided to improve performance for applications where underflow is common and generating a denormalized result is unnecessary.

1. "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229. https://ieeexplore.ieee.org/document/8766229

See Section 3.9.2, “Floating-Point Modes and Exceptions.”

7.6 SIMD OPTIMIZATIONS AND MICROARCHITECTURES

Pentium M, Intel Core Solo, and Intel Core Duo processors have a different microarchitecture than the Intel NetBurst microarchitecture. Intel Core microarchitecture offers significantly more efficient SIMD floating-point capability than previous microarchitectures. In addition, instruction latency and throughput of SSE3 instructions are improved considerably in Intel Core microarchitectures over previous microarchitectures.

7.6.1 SIMD Floating-point Programming Using SSE3

SSE3 enhances SSE and SSE2 with nine instructions targeted for SIMD floating-point programming. In contrast to many SSE/SSE2 instructions offering homogeneous arithmetic operations on parallel data elements and favoring the vertical computation model, SSE3 offers instructions that perform asymmetric arithmetic and arithmetic operations on horizontal data elements.

ADDSUBPS and ADDSUBPD are two instructions with asymmetric arithmetic processing capability (see Figure 7-5). HADDPS, HADDPD, HSUBPS, and HSUBPD offer horizontal arithmetic processing capability (see Figure 7-6). In addition: MOVSLDUP, MOVSHDUP, and MOVDDUP load data from memory (or XMM register) and replicate data elements simultaneously.

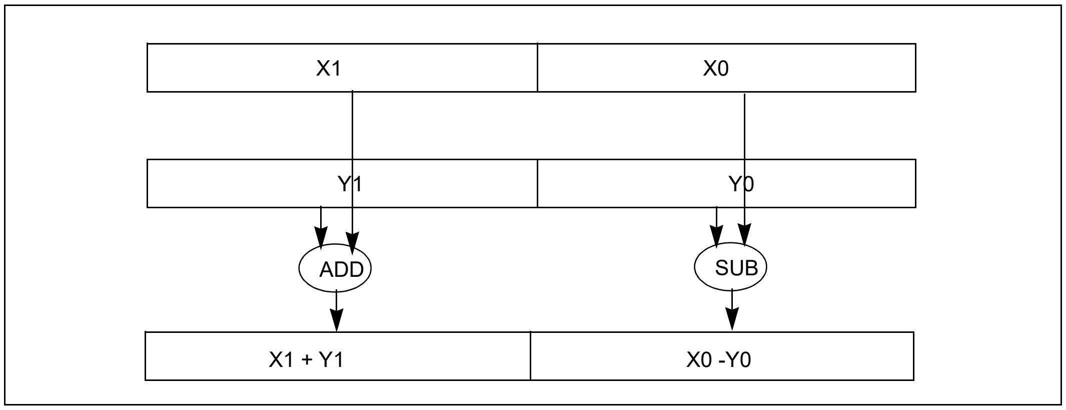


Figure 7-5. Asymmetric Arithmetic Operation of the SSE3 Instruction

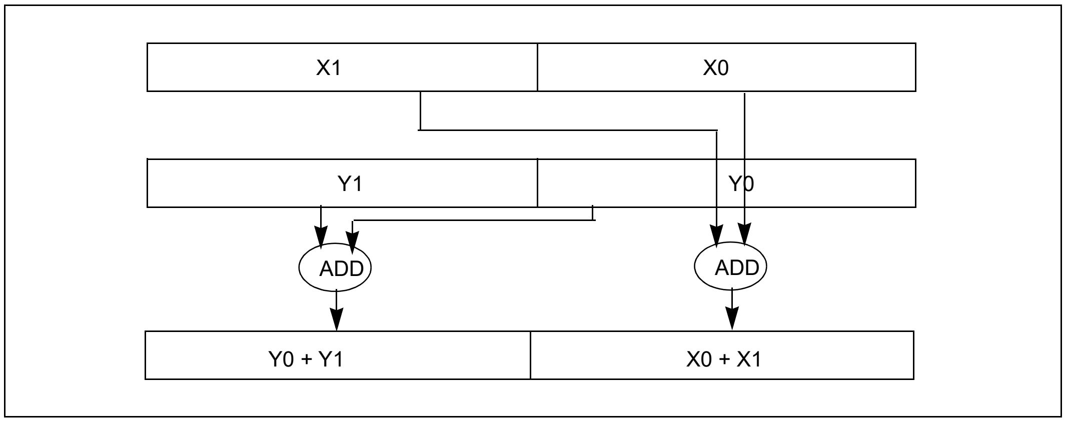


Figure 7-6. Horizontal Arithmetic Operation of the SSE3 Instruction HADDPD

7.6.1.1 SSE3 and Complex Arithmetics

The flexibility of SSE3 in dealing with AOS-type data structures can be demonstrated by the example of multiplication and division of complex numbers. For example, a complex number can be stored in a structure consisting of its real and imaginary parts. This naturally leads to the use of an array of structure. Example 7-9 demonstrates using SSE3 instructions to multiply single-precision complex numbers. Example 7-10 shows using SSE3 instructions to divide complex numbers.

Example 7-9. Multiplication of Two Pairs of Single-Precision Complex Number

```
// Multiplication of  (ak + i bk ) * (ck + i dk )
// a + i b can be stored as a data structure
movsldup    xmm0, Src1; load real parts into the destination,
                    ; a1, a1, a0, a0

movaps      xmm1, src2; load the 2nd pair of complex values,
                    ; i.e. d1, c1, d0, c0
mulps       xmm0, xmm1; temporary results, a1d1, a1c1, a0d0,
                    ; a0c0

shufps      xmm1, xmm1, b1; reorder the real and imaginary
                    ; parts, c1, d1, c0, d0
movshdup    xmm2, Src1; load the imaginary parts into the
                    ; destination, b1, b1, b0, b0

mulps       xmm2, xmm1; temporary results, b1c1, b1d1, b0c0,
                    ; b0d0
addsubps    xmm0, xmm2; b1c1+a1d1, a1c1 -b1d1, b0c0+a0d0,
                    ; a0c0-b0d0
```

Example 7-10. Division of Two Pairs of Single-Precision Complex Numbers

```
// Division of (ak + i bk ) / (ck + i dk )
movshdup    xmm0, Src1; load imaginary parts into the
                    ; destination, b1, b1, b0, b0
movaps      xmm1, src2; load the 2nd pair of complex values,
                    ; i.e. d1, c1, d0, c0
mulps       xmm0, xmm1; temporary results, b1d1, b1c1, b0d0,
                    ; b0c0

shufps      xmm1, xmm1, b1; reorder the real and imaginary
                    ; parts, c1, d1, c0, d0
movsldup    xmm2, Src1; load the real parts into the
                    ; destination, a1, a1, a0, a0

mulps       xmm2, xmm1; temp results, a1c1, a1d1, a0c0, a0d0
addsubps    xmm0, xmm2; a1c1+b1d1, b1c1-a1d1, a0c0+b0d0,
                    ; b0c0-a0d0

mulps       xmm1, xmm1; c1c1, d1d1, c0c0, d0d0
movps       xmm2, xmm1;c1c1, d1d1, c0c0, d0d0
shufps      xmm2, xmm2, b1; d1d1, c1c1, d0d0, c0c0
addps       xmm2, xmm1; c1c1+d1d1, c1c1+d1d1, c0c0+d0d0,
                    ; c0c0+d0d0
```

Example 7-10. Division of Two Pairs of Single-Precision Complex Numbers (Contd.)

```
divps   xmm0, xmm2
shufps  xmm0, xmm0, b1  ; (b1c1-a1d1)/(c1c1+d1d1),
                        ; (a1c1+b1d1)/(c1c1+d1d1),
                        ; (b0c0-a0d0)/( c0c0+d0d0),
                        ; (a0c0+b0d0)/( c0c0+d0d0)
```

In both examples, the complex numbers are stored in arrays of structures. MOVSLDUP, MOVSHDUP, and the asymmetric ADDSUBPS allow performing complex arithmetic on two pairs of single-precision complex numbers simultaneously, without unnecessary swizzling between data elements.

Due to microarchitectural differences, software should implement the multiplication of complex double-precision numbers using SSE3 instructions on processors based on Intel Core microarchitecture. In Intel Core Duo and Intel Core Solo processors, software should use scalar SSE2 instructions to implement double-precision complex multiplication. This is because the data path between SIMD execution units is 128 bits in the Intel Core microarchitecture and 64 in previous microarchitectures. Processors based on the Enhanced Intel Core microarchitecture generally execute SSE3 instruction more efficiently than previous microarchitectures. They also have a 128-bit shuffle unit that will benefit complex arithmetic operations further than the Intel Core microarchitecture.

Example 7-11 shows two equivalent implementations of double-precision complex multiplication of two pairs of complex numbers using vector SSE2 versus SSE3 instructions. Example 7-12 shows the equivalent scalar SSE2 implementation.

Example 7-11. Double-Precision Complex Multiplication of Two Pairs

SSE2 Vector Implementation	SSE3 Vector Implementation
`movapd    xmm0, [rax]      ;y x`	`movapd     xmm0, [rax]      ;y x`
`movapd    xmm1, [rax+16]   ;w z`	`movapd     xmm1, [rax+16]   ;z z`
`unpcklpd  xmm1, xmm1       ;z z`	`movapd     xmm2, xmm1`
`movapd    xmm2, [rax+16]   ;w z`	`unpcklpd   xmm1, xmm1`
`unpckhpd  xmm2, xmm2       ;w w`	`unpckhpd   xmm2, xmm2`
`mulpd     xmm1, xmm0       ;z*y z*x`	`mulpd      xmm1, xmm0       ;z*y z*x`
`mulpd     xmm2, xmm0       ;w*y w*x`	`mulpd      xmm2, xmm0       ;w*y w*x`
`xorpd     xmm2, xmm7       ;-w*y +w*x`	`shufpd     xmm2, xmm2, 1    ;w*x w*y`
`shufpd    xmm2, xmm2,1     ;w*x -w*y`	`addsubpd   xmm1, xmm2       ;w*x+z*y z*x-w*y`
`addpd     xmm2, xmm1       ;z*y+w*x z*x-w*y`	`movapd     [rcx], xmm1`
`movapd    [rcx], xmm2`	

Example 7-12. Double-Precision Complex Multiplication Using Scalar SSE2

```
movsd  xmm0, [rax]      ;x
movsd  xmm5, [rax+8]    ;y
movsd  xmm1, [rax+16]   ;z
movsd  xmm2, [rax+24]   ;w

movsd  xmm3, xmm1       ;z
movsd  xmm4, xmm2       ;w
mulsd  xmm1, xmm0       ;z*x
mulsd  xmm2, xmm0       ;w*x
mulsd  xmm3, xmm5       ;z*y
```

Example 7-12. Double-Precision Complex Multiplication Using Scalar SSE2 (Contd.)

```
mulsd   xmm4, xmm5    ;w*y
subsd   xmm1, xmm4    ;z*x - w*y
addsd   xmm3, xmm2    ;z*y + w*x
movsd   [rcx], xmm1
movsd   [rcx+8], xmm3
```

7.6.1.2 Packed Floating-Point Performance in Intel Core Duo Processor

Most of the packed SIMD floating-point code will speed up on Intel Core Solo processors relative to Pentium M processors. This is due to an improvement in decoding packed SIMD instructions.

The improved packed floating-point performance on the Intel Core Solo processor over the Pentium M processor depends on several factors. Generally, decoder-bound code with a mixture of integer and packed floating-point instructions can expect significant gain. Code that is limited by execution latency and has a "cycles per instructions" ratio greater than one will not benefit from decoder improvement.

When targeting complex arithmetics on Intel Core Solo and Intel Core Duo processors, single-precision SSE3 instructions can deliver higher performance than alternatives. On the other hand, tasks requiring double-precision complex arithmetic may perform better using scalar SSE2 instructions on Intel Core Solo and Intel Core Duo processors. This is because scalar SSE2 instructions can be dispatched through two ports and executed using two separate floating-point units.

Packed horizontal SSE3 instructions (HADDPS and HSUBPS) can simplify the code sequence for some tasks. However, these instructions consist of more than five micro-ops on Intel Core Solo and Intel Core Duo processors. Care must be taken to ensure the latency and decoding penalty of the horizontal instruction does not offset any algorithmic benefits.

7.6.2 Dot Product and Horizontal SIMD Instructions

Sometimes the AOS-type of data organization is more natural in many algebraic formulae. One typical example is the *dot product* operation. The dot product operation can be implemented using SSE/SSE2 instruction sets. SSE3 added a few horizontal add/subtract instructions for applications that rely on the horizontal computation model. SSE4.1 provides additional enhancement with instructions capable of directly evaluating dot product operations of vectors of 2, 3 or 4 components.

Example 7-13. Dot Product of Vector Length 4 Using SSE/SSE2

Using SSE/SSE2 to compute one dot product

```
movaps    xmm0, [rax]        // a4, a3, a2, a1
mulps     xmm0, [rax+16]     // a4*b4, a3*b3, a2*b2, a1*b1
movhlps   xmm1, xmm0         // X, X,  a4*b4, a3*b3, upper half not needed
addps     xmm0, xmm1         // X, X, a2*b2+a4*b4, a1*b1+a3*b3,
pshufd    xmm1, xmm0, 1      // X, X, X, a2*b2+a4*b4
addss     xmm0, xmm1         // a1*b1+a3*b3+a2*b2+a4*b4
movss     [rcx], xmm0
```

Example 7-14. Dot Product of Vector Length 4 Using SSE3

```
Using SSE3 to compute one dot product

movaps    xmm0, [rax]
mulps     xmm0, [rax+16]   // a4*b4, a3*b3, a2*b2, a1*b1
haddps    xmm0, xmm0       // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
movaps    xmm1, xmm0       // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
psrlq     xmm0, 32         // 0, a4*b4+a3*b3, 0, a4*b4+a3*b3
addss     xmm0, xmm1       // -, -, -, a1*b1+a3*b3+a2*b2+a4*b4
movss     [rax], xmm0
```

Example 7-15. Dot Product of Vector Length 4 Using SSE4.1

```
Using SSE4.1 to compute one dot product

movaps    xmm0, [rax]
dpps      xmm0, [rax+16], 0xf1   // 0, 0, 0, a1*b1+a3*b3+a2*b2+a4*b4
movss     [rax], xmm0
```

Example 7-13, Example 7-14, and Example 7-15 compare the basic code sequence to compute one dot-product result for a pair of vectors.

The selection of an optimal sequence in conjunction with an application's memory access patterns may favor different approaches. For example, if each dot product result is immediately consumed by additional computational sequences, it may be more optimal to compare the relative speed of these different approaches. If dot products can be computed for an array of vectors and kept in the cache for subsequent computations, then more optimal choice may depend on the relative throughput of the sequence of instructions.

In Intel Core microarchitecture, Example 7-14 has higher throughput than Example 7-13. Due to the relatively longer latency of HADDPS, the speed of Example 7-14 is slightly slower than Example 7-13.

In Enhanced Intel Core microarchitecture, Example 7-15 is faster in both speed and throughput than Example 7-13 and Example 7-14. Although the latency of DPPS is also relatively long, it is compensated by the reduction of number of instructions in Example 7-15 to do the same amount of work.

Unrolling can further improve the throughput of each of three dot product implementations. Example 7-16 shows two unrolled versions using the basic SSE2 and SSE3 sequences. The SSE4.1 version can also be unrolled and using INSERTPS to pack 4 dot-product results.

Example 7-16. Unrolled Implementation of Four Dot Products

SSE2 Implementation	SSE3 Implementation
`movaps xmm0, [rax]`	`movaps xmm0, [rax]`
`mulps xmm0, [rax+16] ;w0*w1 z0*z1 y0*y1 x0*x1`	`mulps xmm0, [rax+16]`
`movaps xmm2, [rax+32]`	`movaps xmm1, [rax+32]`
`mulps xmm2, [rax+16+32] ;w2*w3 z2*z3 y2*y3 x2*x3`	`mulps xmm1, [rax+16+32]`
`movaps xmm3, [rax+64]`	`movaps xmm2, [rax+64]`
`mulps xmm3, [rax+16+64] ;w4*w5 z4*z5 y4*y5 x4*x5`	`mulps xmm2, [rax+16+64]`
`movaps xmm4, [rax+96]`	`movaps xmm3, [rax+96]`
`mulps xmm4, [rax+16+96] ;w6*w7 z6*z7 y6*y7 x6*x7`	`mulps xmm3, [rax+16+96]`
	`haddps xmm0, xmm1`
	`haddps xmm2, xmm3`
	`haddps xmm0, xmm2`
	`movaps [rcx], xmm0`

Example 7-16. Unrolled Implementation of Four Dot Products (Contd.)

SSE2 Implementation	SSE3 Implementation
```
movaps    xmm1, xmm0
unpcklps  xmm0, xmm2    ; y2*y3 y0*y1 x2*x3 x0*x1
unpckhps  xmm1, xmm2    ; w2*w3 w0*w1 z2*z3 z0*z1
movaps    xmm5, xmm3
unpcklps  xmm3, xmm4    ; y6*y7 y4*y5 x6*x7 x4*x5
unpckhps  xmm5, xmm4    ; w6*w7 w4*w5 z6*z7 z4*z5

addps     xmm0, xmm1
addps     xmm5, xmm3
movaps    xmm1, xmm5
movhlps   xmm1, xmm0
movlhps   xmm0, xmm5
addps     xmm0, xmm1
movaps    [rcx], xmm0
```

7.6.3 Vector Normalization

Normalizing vectors is a common operation in many floating-point applications. Example 7-17 shows an example in C of normalizing an array of (x, y, z) vectors.

Example 7-17. Normalization of an Array of Vectors

```
for (i=0;i<CNT;i++)
{ float size = nodes[i].vec.dot();
    if (size != 0.0)
    { size = 1.0f/sqrtf(size); }
    else
    { size = 0.0; }
    nodes[i].vec.x *= size;
    nodes[i].vec.y *= size;
    nodes[i].vec.z *= size;
}
```

Example 7-18 shows an assembly sequence that normalizes the x, y, z components of a vector.

Example 7-18. Normalize (x, y, z) Components of an Array of Vectors Using SSE2

```
Vec3 *p = &nodes[i].vec;
__asm
{   mov      rax, p
    xorps    xmm2, xmm2
    movups   xmm1, [rax]          // loads the (x, y, z) of input vector plus x of next vector
    movaps   xmm7, xmm1           // save a copy of data from memory (to restore the unnormalized value)
    movaps   xmm5, _mask          // mask to select (x, y, z) values from an xmm register to normalize
    andps    xmm1, xmm5           // mask 1st 3 elements
    movaps   xmm6, xmm1           // save a copy of (x, y, z) to compute normalized vector later
    mulps    xmm1,xmm1            // 0, z*z, y*y, x*x
    pshufd   xmm3, xmm1, 0x1b     // x*x, y*y, z*z, 0
    addps    xmm1, xmm3           // x*x, z*z+y*y, z*z+y*y, x*x
    pshufd   xmm3, xmm1, 0x41     // z*z+y*y, x*x, x*x, z*z+y*y
    addps    xmm1, xmm3           // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd   xmm1, xmm2           // compare size to 0
    jz zero
    movaps   xmm3, xmm4           // preloaded unitary vector (1.0, 1.0, 1.0, 1.0)
    sqrtps   xmm1, xmm1
    divps    xmm3, xmm1
    jmp      store
zero:
    movaps   xmm3, xmm2
store:

    mulps    xmm3, xmm6           //normalize the vector in the lower 3 elements
    andnps   xmm5, xmm7           // mask off the lower 3 elements to keep the un-normalized value
    orps     xmm3, xmm5           // order the un-normalized component after the normalized vector
    movaps   [rax], xmm3          // writes normalized x, y, z; followed by unmodified value
```

Example 7-19 shows an assembly sequence using SSE4.1 to normalizes the x, y, z components of a vector.

Example 7-19. Normalize (x, y, z) Components of an Array of Vectors Using SSE4.1

```
Vec3 *p = &nodes[i].vec;
__asm
{   mov      rax, p
    xorps    xmm2, xmm2
    movups   xmm1, [rax]              // loads the (x, y, z) of input vector plus x of next vector
    movaps   xmm7, xmm1               // save a copy of data from memory
    dpps     xmm1, xmm1, 0x7f         // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd   xmm1, xmm2               // compare size to 0
    jz zero
    movaps   xmm3, xmm4               // preloaded unitary vector (1.0, 1.0, 1.0, 1.0)
    sqrtps   xmm1, xmm1
    divps    xmm3, xmm1
    jmp      store
zero:
    movaps   xmm3, xmm2
store:
    mulps    xmm3, xmm6               //normalize the vector in the lower 3 elements
    blendps  xmm3, xmm7, 0x8          // copy the un-normalized component next to the normalized vector
    movaps   [rax], xmm3
```

In Example 7-18 and Example 7-19, the throughput of these instruction sequences are basically limited by the long-latency instructions of DIVPS and SQRTPS. In Example 7-19, the use of DPPS replaces eight SSE2 instructions to evaluate and broadcast the dot-product result to four elements of an XMM register. This could result in improvement of the relative speed of Example 7-19 over Example 7-18.

7.6.4 Using Horizontal SIMD Instruction Sets and Data Layout

SSE and SSE2 provide packed add/subtract, multiply/divide instructions that are ideal for situations that can take advantage of vertical computation model, such as SOA data layout. SSE3 and SSE4.1 added horizontal SIMD instructions including horizontal add/subtract, dot-product operations. These more recent SIMD extensions provide tools to solve problems involving data layouts or operations that do not conform to the vertical SIMD computation model.

In this section, we consider a vector-matrix multiplication problem and discuss the relevant factors for choosing various horizontal SIMD instructions.

Example 7-20 shows the vector-matrix data layout in AOS, where the input and out vectors are stored as an array of structure.

Example 7-20. Data Organization in Memory for AOS Vector-Matrix Multiplication

```
Matrix M4x4 (pMat):                  M00 M01 M02 M03
                                     M10 M11 M12 M13
                                     M20 M21 M22 M23
                                     M30 M31 M32 M33
4 input vertices V4x1 (pVert):       V0x V0y V0z V0w
                                     V1x V1y V1z V1w
                                     V2x V2y V2z V2w
                                     V3x V3y V3z V3w
Output vertices O4x1 (pOutVert):     O0x O0y O0z O0w
                                     O1x O1y O1z O1w
                                     O2x O2y O2z O2w
                                     O3x O3y O3z O3w
```

Example 7-21 shows an example using HADDPS and MULPS to perform vector-matrix multiplication with data layout in AOS. After three HADDPS completing the summations of each output vector component, the output components are arranged in AOS.

Example 7-21. AOS Vector-Matrix Multiplication with HADDPS

```
    mov       rax, pMat
    mov       rbx, pVert
    mov       rcx, pOutVert
    xor                   rdx, rdx
    movaps  xmm5,[rax+16]         // load row M1?
    movaps  xmm6,[rax+2*16]       // load row M2?
    movaps  xmm7,[rax+3*16]       // load row M3?
lloop:
    movaps  xmm4, [rbx + rdx]     // load input vector
    movaps  xmm0, xmm4
    mulps   xmm0, [rax]           // m03*vw, m02*vz, m01*vy, m00*vx,
    movaps  xmm1, xmm4
    mulps   xmm1, xmm5            // m13*vw, m12*vz, m11*vy, m10*vx,

    movaps  xmm2, xmm4
    mulps   xmm2, xmm6            // m23*vw, m22*vz, m21*vy, m20*vx
    movaps  xmm3, xmm4
    mulps   xmm3, xmm7            // m33*vw, m32*vz, m31*vy, m30*vx,
    haddps  xmm0, xmm1
    haddps  xmm2, xmm3
    haddps  xmm0, xmm2
    movaps  [rcx + rdx], xmm0     // store a vector of length 4
    add     rdx, 16
    cmp     rdx, top
    jb lloop
```

Example 7-22 shows an example using DPPS to perform vector-matrix multiplication in AOS.

Example 7-22. AOS Vector-Matrix Multiplication with DPPS

```
    mov     rax, pMat
    mov     rbx, pVert
    mov     rcx, pOutVert
    xor                     rdx, rdx
    movaps  xmm5,[rax+16]          // load row M1?
    movaps  xmm6,[rax+2*16]        // load row M2?
    movaps  xmm7,[rax+3*16]        // load row M3?
lloop:
    movaps  xmm4, [rbx + rdx]      // load input vector
    movaps  xmm0, xmm4
    dpps    xmm0, [rax], 0xf1      // calculate dot product of length 4, store to lowest dword
    movaps  xmm1, xmm4
    dpps    xmm1, xmm5, 0xf1
    movaps  xmm2, xmm4
    dpps    xmm2, xmm6, 0xf1
    movaps  xmm3, xmm4
    dpps    xmm3, xmm7, 0xf1
    movss   [rcx + rdx + 0*4], xmm0  // store one element of vector length 4
    movss   [rcx + rdx + 1*4], xmm1
    movss   [rcx + rdx + 2*4], xmm2
    movss   [rcx + rdx + 3*4], xmm3
    add     rdx, 16
    cmp     rdx, top
    jb      lloop
```

Example 7-21 and Example 7-22 both work with AOS data layout using different horizontal processing techniques provided by SSE3 and SSE4.1. The effectiveness of either techniques will vary, depending on the degree of exposures of long-latency instruction in the inner loop, the overhead/efficiency of data movement, and the latency of HADDPS vs. DPPS.

On processors that support both HADDPS and DPPS, the choice between either technique may depend on application-specific considerations. If the output vectors are written back to memory directly in a batch situation, Example 7-21 may be preferable over Example 7-22, because the latency of DPPS is long and storing each output vector component individually is less than ideal for storing an array of vectors.

There may be partially-vectorizable situations that the individual output vector component is consumed immediately by other non-vectorizable computations. Then, using DPPS producing individual component may be more suitable than dispersing the packed output vector produced by three HADDPS as in Example 7-21.

7.6.4.1 SOA and Vector Matrix Multiplication

If the native data layout of a problem conforms to SOA, then vector-matrix multiply can be coded using MULPS, ADDPS without using the longer-latency horizontal arithmetic instructions, or packing scalar components into packed format (Example 7-22). To achieve higher throughput with SOA data layout, there are either prerequisite data preparation or swizzling/deswizzling on-the-fly that must be comprehended. For example, an SOA data layout for vector-matrix multiplication is shown in Example 7-23.

Each matrix element is replicated four times to minimize data movement overhead for producing packed results.

Example 7-23. Data Organization in Memory for SOA Vector-Matrix Multiplication

```
Matrix M16x4 (pMat):
          M00 M00 M00 M00 M01 M01 M01 M01 M02 M02 M02 M02 M03 M03 M03 M03
          M10 M10 M10 M10 M11 M11 M11 M11 M12 M12 M12 M12 M13 M13 M13 M13
          M20 M20 M20 M20 M21 M21 M21 M21 M22 M22 M22 M22 M23 M23 M23 M23
          M30 M30 M30 M30 M31 M31 M31 M31 M32 M32 M32 M32 M33 M33 M33 M33
4 input vertices V4x1 (pVert):   V0x V1x V2x V3x
                                 V0y V1y V2y V3y
                                 V0z V1z V2z V3z
                                 V0w V1w V2w V3w
Ouput vertices O4x1 (pOutVert): O0x O1x O2x O3x
                                 O0y O1y O2y O3y
                                 O0z O1z O2z O3z
                                 O0w O1w O2w O3w
```

The corresponding vector-matrix multiply example in SOA (unrolled for four iteration of vectors) is shown in Example 7-24.

Example 7-24. Vector-Matrix Multiplication with Native SOA Data Layout

```
    mov     rbx, pVert
    mov     rcx, pOutVert
    xor     rdx, rdx
    movaps  xmm5,[rax + 16]      // load row M1?
    movaps  xmm6,[rax + 2*16]    // load row M2?
    movaps  xmm7,[rax + 3*16]    // load row M3?
lloop_vert:
    mov     rax, pMat
    xor     edi, edi
    movaps  xmm0, [rbx]          // load V3x, V2x, V1x, V0x
    movaps  xmm1, [rbx]          // load V3y, V2y, V1y, V0y
    movaps  xmm2, [rbx]          // load V3z, V2z, V1z, V0z
    movaps  xmm3, [rbx]          // load V3w, V2w, V1w, V0w
loop_mat:
    movaps  xmm4, [rax]          // m00, m00, m00, m00,
    mulps   xmm4, xmm0           // m00*V3x, m00*V2x, m00*V1x, m00*V0x,
    movaps  xmm4, [rax + 16]     // m01, m01, m01, m01,
    mulps   xmm5, xmm1           // m01*V3y, m01*V2y, m01*V1y, m01*V0y,
    addps   xmm4, xmm5
    movaps  xmm5, [rax + 32]     // m02, m02, m02, m02,
    mulps   xmm5, xmm2           // m02*V3z, m02*V2z, m02*V1z, m02*V0z,
    addps   xmm4, xmm5
    movaps  xmm5, [rax+ 48]      // m03, m03, m03, m03,
    mulps   xmm5, xmm3           // m03*V3w, m03*V2w, m03*V1w, m03*V0w,
    addps   xmm4, xmm5
    movaps  [rcx + rdx], xmm4
    add     rax, 64
    add     rdx, 16
    add     edi, 1
    cmp     edi, 4
    jb lloop_mat
    add     rbx, 64
    cmp     rdx, top
    jb lloop_vert
```

7. Updates to Chapter 10

Change bars and **violet** text show changes to Chapter 10 of the *Intel® 64 and IA-32 Architectures Optimization Resource Manual:* Sub-NUMA Clustering.

Changes to this chapter:

- Section 10.3:
  - Consolidated links and page titles where necessary.
  - Removed dead link to Intel blog.

CHAPTER 10 SUB-NUMA CLUSTERING

Sub-NUMA Clustering (SNC) is a mode for improving average latency from last level cache (LLC) to local memory. It replaces the Cluster-on-Die (COD) implementation which was used in the previous generation of the Intel® Xeon® processor E5 family.

10.1 SUB-NUMA CLUSTERING

SNC can improve the average LLC/memory latency by splitting the LLC into disjoint clusters based on address range, with each cluster bound to a subset of memory controllers in the system.

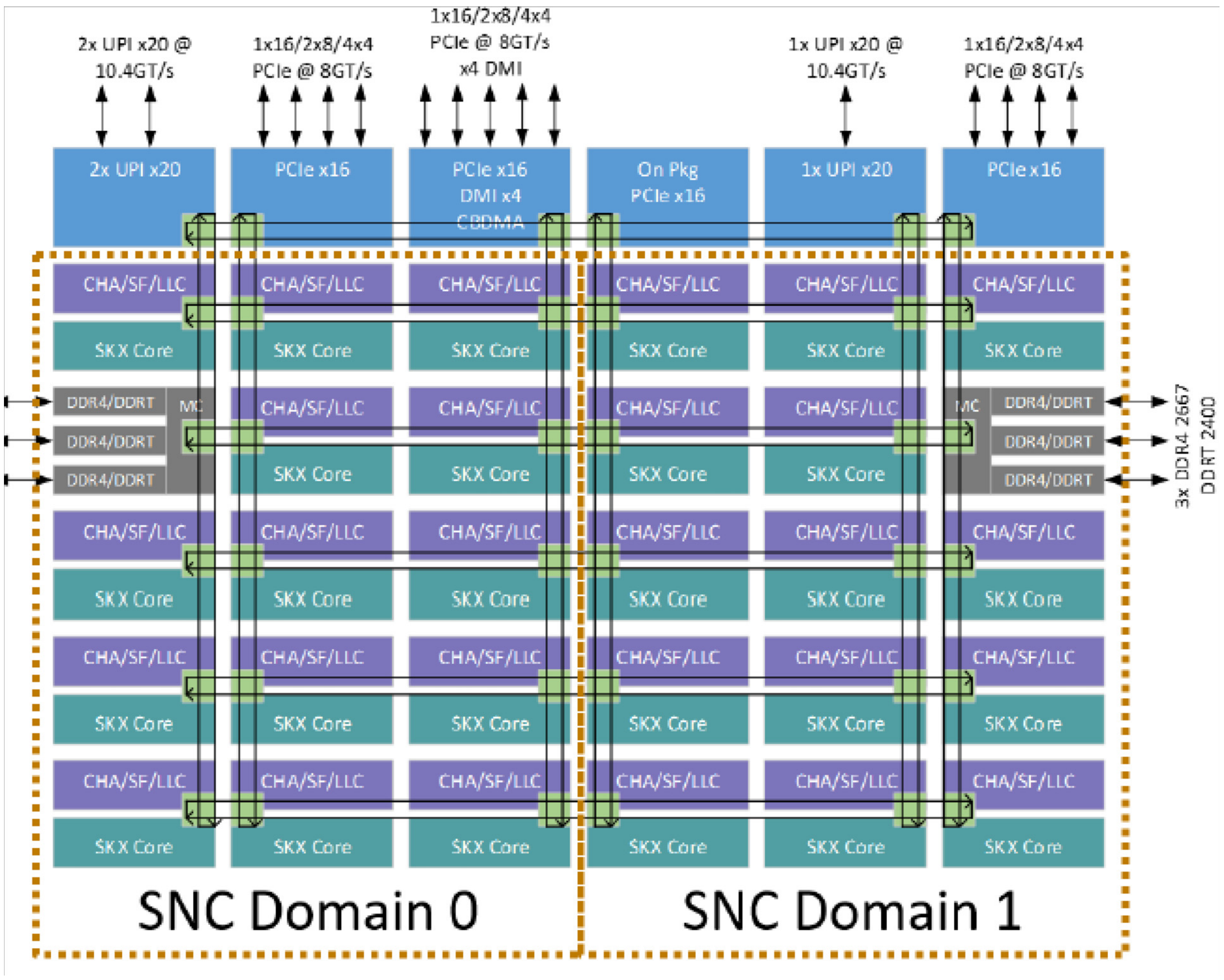


Figure 10-1. Example of SNC Configuration

10.2 COMPARISON WITH CLUSTER-ON-DIE

SNC provides similar localization benefits to those of COD, but without some of COD's disadvantages. Unlike COD, SNC has the following properties.

- Only one Ultra Path Interconnect (UPI) caching agent is required.
- Memory access latency in remote clusters is smaller, as no UPI flow is needed.
- It uses LLC capacity more efficiently as there is no duplication of lines in the LLC.

A disadvantage of SNC is listed below.

- Remote cluster addresses are never cached in local cluster LLC, resulting in larger latency compared to Cluster-on-Die (COD) in some cases.

10.3 SNC USAGE

This section describes the following modes and their BIOS names in brackets (the exact BIOS parameter names may vary depending on the BIOS vendor and version).

- NUMA disabled (NUMA Optimized: Disabled)
- SNC off (Integrated Memory Controller (IMC) Interleaving: auto, NUMA Optimized: Enabled, Sub_NUMA Cluster: Disabled)
- SNC on (IMC Interleaving: 1-way Interleave, NUMA Optimized: Enabled, Sub_NUMA Cluster: Enabled)

The commands that follow were executed on a 2-socket Intel® Xeon® system, 28 cores per a socket, Intel® Hyper-Threading Technology enabled.

10.3.1 How to Check NUMA Configuration

There are additional NUMA nodes in a system with SNC enabled; to get benefits from the SNC feature, a developer should be aware of the NUMA configuration.

This chapter describes different ways to check NUMA system configuration.

libnuma

An application can check NUMA configuration with `libnuma`.

As an example this code uses the `libnuma` library to find the maximum number of NUMA nodes.

```
#include <stdio.h>
#include <stdlib.h>
#include <numa.h>

int main(int argc, char *argv[])
{
    int max_node;

/* Check the system for NUMA support */
    max_node = numa_max_node();
    printf("%d\n", max_node);
    return 0;
}
```

numactl

In Linux* you can check the NUMA configuration with the `numactl` utility (the `numactl-libs`, and `numactl-devel` packages might also be required).

```
$ numactl --hardware
```

NUMA disabled:

```
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
node 0 size: 196045 MB
node 0 free: 190581 MB
node distances:
node   0
  0:  10
```

SNC off:

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83
node 0 size: 96973 MB
node 0 free: 94089 MB
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111
node 1 size: 98304 MB
node 1 free: 95694 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

SNC on:

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 7 8 9 14 15 16 17 21 22 23 56 57 58 59 63 64 65 70
71 72 73 77 78 79
node 0 size: 47821 MB
node 0 free: 45759 MB
node 1 cpus: 4 5 6 10 11 12 13 18 19 20 24 25 26 27 60 61 62 66 67 68 69
74 75 76 80 81 82 83
node 1 size: 49152 MB
node 1 free: 47097 MB
node 2 cpus: 28 29 30 31 35 36 37 42 43 44 45 49 50 51 84 85 86 87 91 92
93 98 99 100 101 105 106 107
node 2 size: 49152 MB
node 2 free: 47617 MB
node 3 cpus: 32 33 34 38 39 40 41 46 47 48 52 53 54 55 88 89 90 94 95 96
97 102 103 104 108 109 110 111
node 3 size: 49152 MB
node 3 free: 47231 MB
node distances:
node   0   1   2   3
  0:  10  11  21  21
  1:  11  10  21  21
  2:  21  21  10  11
  3:  21  21  11  10
```

hwloc

In Linux* you can also check the NUMA configuration with the `lstopo` utility (the `hwloc` package is required). For example:

```
$ lstopo -p --of png --no-io --no-caches > numa_topology.png
```

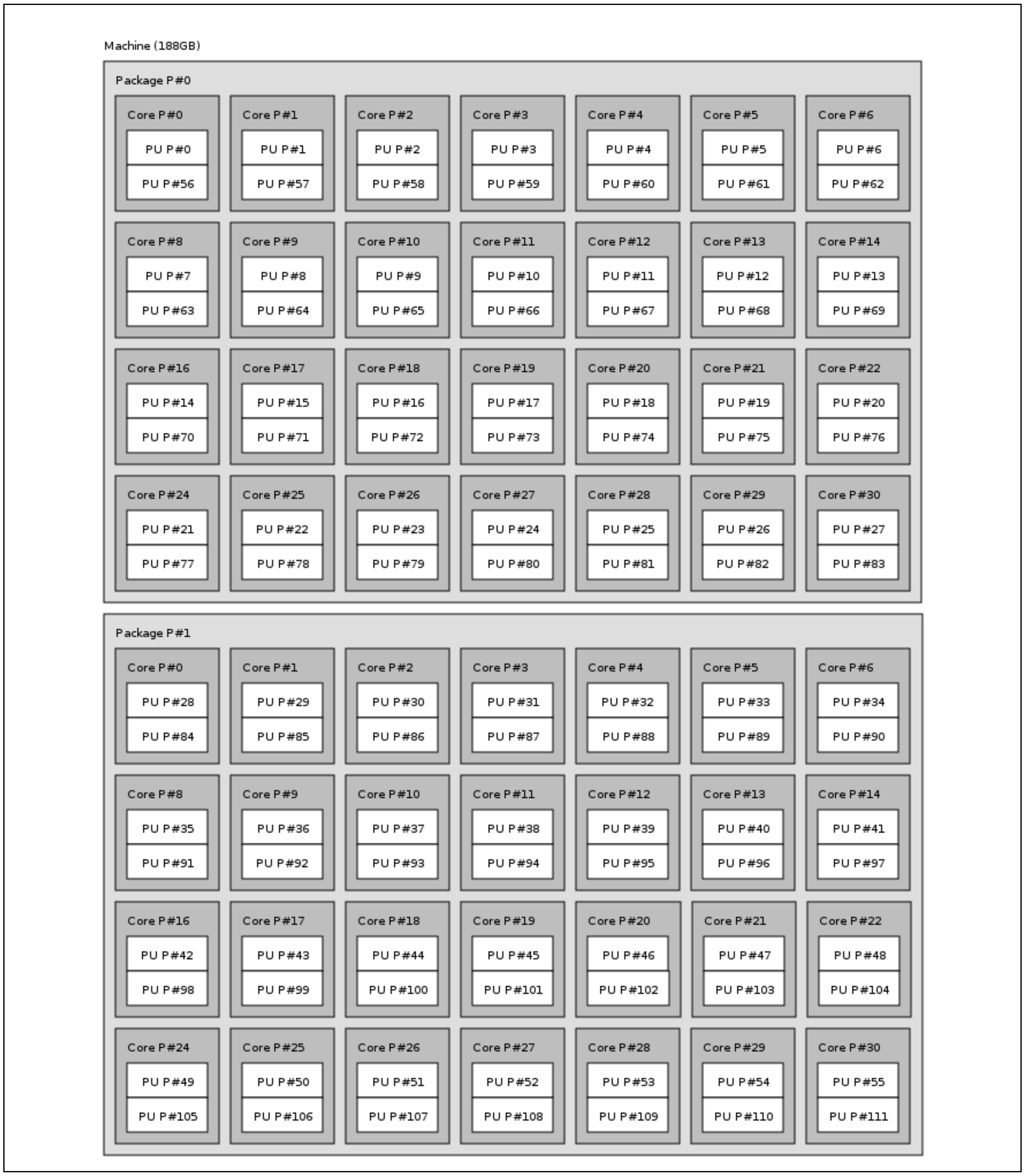


Figure 10-2. NUMA Disabled

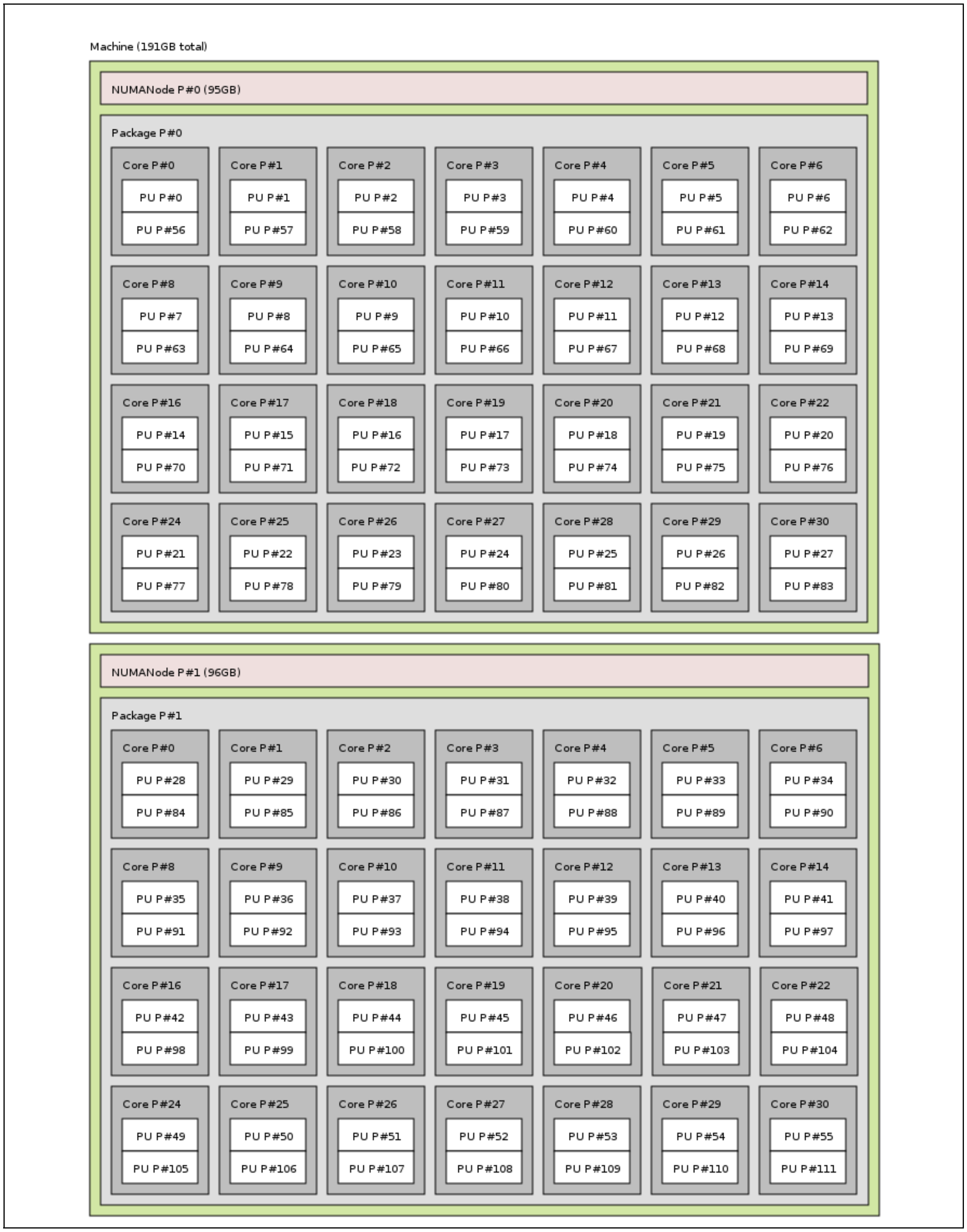


Figure 10-3. SNC Off

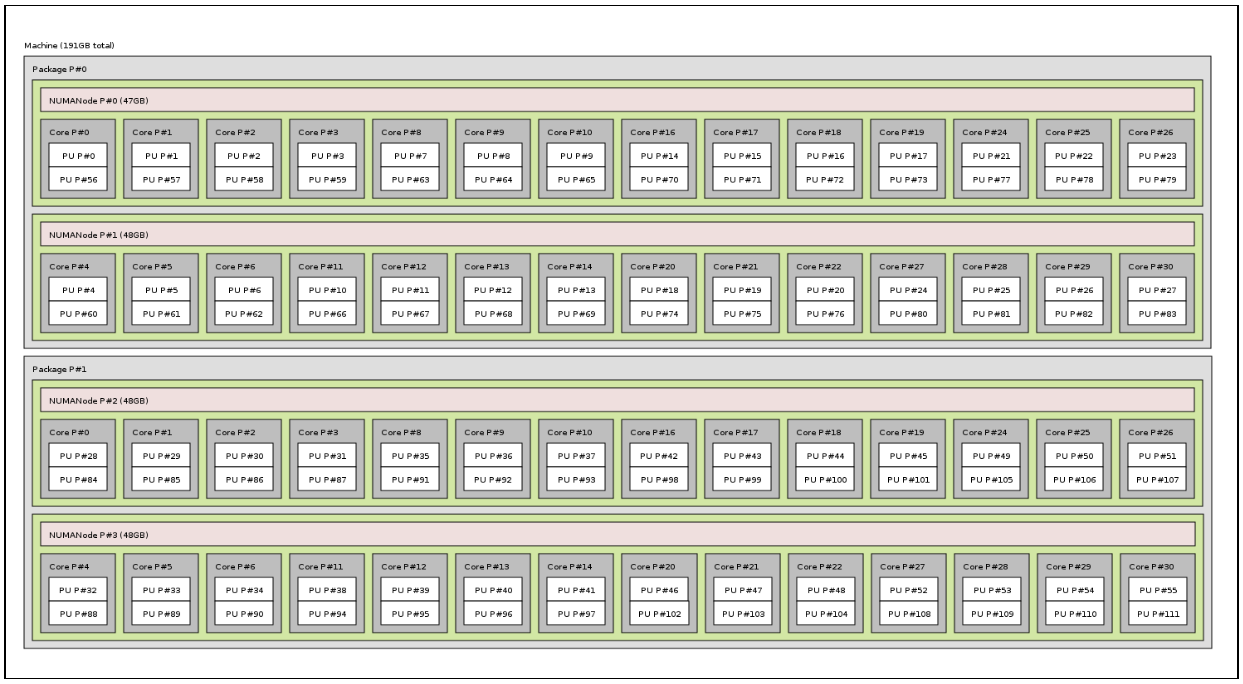

Machine (191GB total)
Package P#0
NUMANode P#0 (47GB)
Core P#0 PU P#0 PU P#56
Core P#1 PU P#1 PU P#57
Core P#2 PU P#2 PU P#58
Core P#3 PU P#3 PU P#59
Core P#8 PU P#7 PU P#63
Core P#9 PU P#8 PU P#64
Core P#10 PU P#9 PU P#65
Core P#16 PU P#14 PU P#70
Core P#17 PU P#15 PU P#71
Core P#18 PU P#16 PU P#72
Core P#19 PU P#17 PU P#73
Core P#24 PU P#21 PU P#77
Core P#25 PU P#22 PU P#78
Core P#26 PU P#23 PU P#79
NUMANode P#1 (48GB)
Core P#4 PU P#4 PU P#60
Core P#5 PU P#5 PU P#61
Core P#6 PU P#6 PU P#62
Core P#11 PU P#10 PU P#66
Core P#12 PU P#11 PU P#67
Core P#13 PU P#12 PU P#68
Core P#14 PU P#13 PU P#69
Core P#20 PU P#18 PU P#74
Core P#21 PU P#19 PU P#75
Core P#22 PU P#20 PU P#76
Core P#27 PU P#24 PU P#80
Core P#28 PU P#25 PU P#81
Core P#29 PU P#26 PU P#82
Core P#30 PU P#27 PU P#83
Package P#1
NUMANode P#2 (48GB)
Core P#0 PU P#28 PU P#84
Core P#1 PU P#29 PU P#85
Core P#2 PU P#30 PU P#86
Core P#3 PU P#31 PU P#87
Core P#8 PU P#35 PU P#91
Core P#9 PU P#36 PU P#92
Core P#10 PU P#37 PU P#93
Core P#16 PU P#42 PU P#98
Core P#17 PU P#43 PU P#99
Core P#18 PU P#44 PU P#100
Core P#19 PU P#45 PU P#101
Core P#24 PU P#49 PU P#105
Core P#25 PU P#50 PU P#106
Core P#26 PU P#51 PU P#107
NUMANode P#3 (48GB)
Core P#4 PU P#32 PU P#88
Core P#5 PU P#33 PU P#89
Core P#6 PU P#34 PU P#90
Core P#11 PU P#38 PU P#94
Core P#12 PU P#39 PU P#95
Core P#13 PU P#40 PU P#96
Core P#14 PU P#41 PU P#97
Core P#20 PU P#46 PU P#102
Core P#21 PU P#47 PU P#103
Core P#22 PU P#48 PU P#104
Core P#27 PU P#52 PU P#108
Core P#28 PU P#53 PU P#109
Core P#29 PU P#54 PU P#110
Core P#30 PU P#55 PU P#111

Figure 10-4. SNC On

10.3.2 MPI Optimizations for SNC

Software needs to be NUMA optimized to benefit from SNC. Running one MPI rank per NUMA region trivially ensures locality-of-access without requiring changes to the code to ensure that it behaves in a NUMA friendly manner. This is a simple way to improve performance through the use of SNC.

The Intel® MPI Library includes some NUMA-related optimizations. The out-of-the-box behavior of the Intel MPI Library should cover most cases, but there are some environment variables available to control NUMA-related features that can improve performance in specific cases.

The relevant environment variables mainly relate to MPI process placement, that is, process pinning/binding – such as the I_MPI_PIN_DOMAIN variable. For more information, see the Intel® MPI Library Developer Reference. This environment variable defines a number of non-overlapping subsets (domains) of logical processors on a node, and a set of rules for how MPI processes are bound to these domains: one MPI process per domain, as illustrated below.

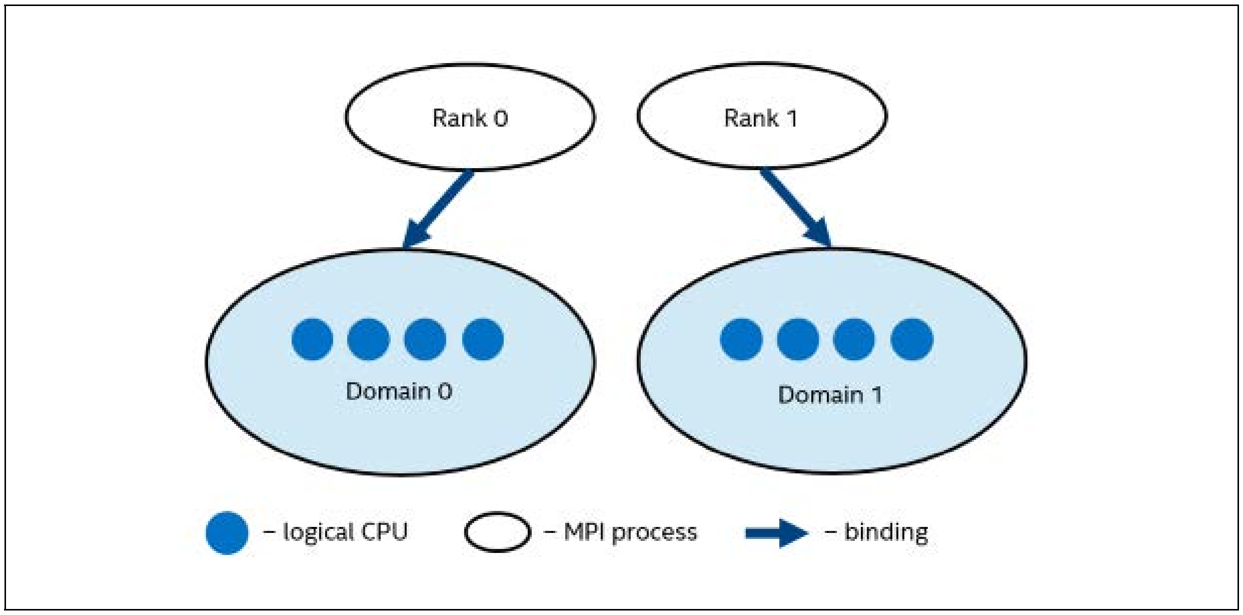


Figure 10-5. Domain Example with One MPI Process Per Domain

Each MPI process can create a number of child threads to run within the corresponding domain. The process' threads can freely migrate from one logical processor to another within the particular domain.

For example, `I_MPI_PIN_DOMAIN=numa` may be a reasonable option for hybrid MPI/OpenMP* applications with SNC mode enabled. In this case, each domain consists of logical processors that share a particular NUMA node. The number of domains on a machine is equal to the number of NUMA nodes on the machine.

Please see the Intel MPI Library documentation for detailed information.

10.3.3 SNC Performance Comparison

This section contains performance data collected with Intel® Memory Latency Checker (Intel® MLC) to demonstrate the variations in performance (latency) between NUMA nodes in different modes.

An important factor in determining application performance is the time required for the application to fetch data from the processor's cache hierarchy and from the memory subsystem. Local memory and cross-socket memory latencies vary significantly in a NUMA-enabled multi-socket system. Bandwidth also plays an important role in determining performance. So measuring these latencies and bandwidths is important when establishing a baseline for the system being tested, and performing performance analysis.

Intel MLC is a tool used to measure memory latencies and bandwidth, and how they change as the load on the system increases. It also provides several options for more fine-grained investigation where bandwidth and latencies from a specific set of cores to caches or memory can be measured as well.

For details, see Intel® Memory Latency Checker v.3.10 (Intel® MLC).

The following command was used to collect the performance data:

```
% mlc_avx512 --latency_matrix
```

This command measures idle memory latency from each socket in the system to every other socket and reports the results in a matrix. The default invocation reports latencies to all of the NUMA nodes in the system. NUMA-level reporting works only on Linux. On Windows, only socket level reporting is supported.

NOTE

It is challenging to measure memory latencies on modern Intel processors accurately as they have sophisticated HW prefetchers. Intel MLC automatically disables these prefetchers while measuring the latencies and restores them to their previous state on completion. The prefetcher control is exposed through an MSR and MSR access requires root level permission. So, Intel MLC needs to be run as `‘root’` on Linux.

The software configuration used for these measurements is Intel MLC v3.3-Beta2, Red Hat* Linux* 7.2.

```
NUMA disabled:
Using buffer size of 2000.000MB
Measuring idle latencies (in ns)...
        Memory node
Socket     0      1
     0   126.5   129.4
     1   123.1   122.6

SNC off:
Using buffer size of 2000.000MB
Measuring idle latencies (in ns)...
                Numa node
Numa node          0       1
       0         81.9   153.1
       1        153.7    82.0

SNC on:
Using buffer size of 2000.000MB
Measuring idle latencies (in ns)...
                Numa node
Numa node          0       1       2       3
       0         81.6    89.4   140.4   153.6
       1         86.5    78.5   144.3   162.8
       2        142.3   153.0    81.6    89.3
       3        144.5   162.8    85.5    77.4
```

6. Updates to Chapter 11

Change bars and **violet** text show changes to Chapter 11 of the *Intel® 64 and IA-32 Architectures Optimization Resource Manual:* Multicore and Hyper-Threading Technology.

Changes to this chapter:

- Section 11.4.1: Updated and consolidated outdated link within document title.
- Section 11.4.2: Removed dead link for: *Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor*.

CHAPTER 11
MULTICORE AND INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

This chapter describes software optimization techniques for multithreaded applications running in an environment using either multiprocessor (MP) systems or processors with hardware-based multi-threading support. Multiprocessor systems are systems with two or more sockets, each mated with a physical processor package. Intel 64 and IA-32 processors that provide hardware multithreading support include dual-core processors, quad-core processors and processors supporting HT Technology[1].

Computational throughput in a multithreading environment can increase as more hardware resources are added to take advantage of thread-level or task-level parallelism. Hardware resources can be added in the form of more than one physical-processor, processor-core-per-package, and/or logical-processor-per-core. Therefore, there are some aspects of multithreading optimization that apply across MP, multi-core, and HT Technology. There are also some specific microarchitectural resources that may be implemented differently in different hardware multithreading configurations (for example: execution resources are not shared across different cores but shared by two logical processors in the same core if HT Technology is enabled). This chapter covers guidelines that apply to these situations.

This chapter covers:

- Performance characteristics and usage models.
- Programming models for multithreaded applications.
- Software optimization techniques in five specific areas.

11.1 PERFORMANCE AND USAGE MODELS

The performance gains of using multiple processors, multicore processors or HT Technology are greatly affected by the usage model and the amount of parallelism in the control flow of the workload. Two common usage models are:

- Multithreaded applications.
- Multitasking using single-threaded applications.

11.1.1 Multithreading

When an application employs multithreading to exploit task-level parallelism in a workload, the control flow of the multi-threaded software can be divided into two parts: parallel tasks and sequential tasks.

Amdahl's law describes an application's performance gain as it relates to the degree of parallelism in the control flow. It is a useful guide for selecting the code modules, functions, or instruction sequences that are most likely to realize the most gains from transforming sequential tasks and control flows into parallel code to take advantage multithreading hardware support.

Figure 11-1 illustrates how performance gains can be realized for any workload according to Amdahl's law. The bar in Figure 11-1 represents an individual task unit or the collective workload of an entire application.

1. The presence of hardware multithreading support in Intel 64 and IA-32 processors can be detected by checking the feature flag CPUID .01H:EDX[28]. A return value of in bit 28 indicates that at least one form of hardware multithreading is present in the physical processor package. The number of logical processors present in each package can also be obtained from CPUID. The application must check how many logical processors are enabled and made available to application at runtime by making the appropriate operating system calls. See the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for information.

In general, the speed-up of running multiple threads on an MP systems with *N* physical processors, over single-threaded execution, can be expressed as:

$$\text{RelativeResponse} = \frac{\text{Tsequential}}{\text{Tparallel}} = \left(1 - P + \frac{P}{N} + O\right)$$

where *P* is the fraction of workload that can be parallelized, and *O* represents the overhead of multi-threading and may vary between different operating systems. In this case, performance gain is the inverse of the relative response.

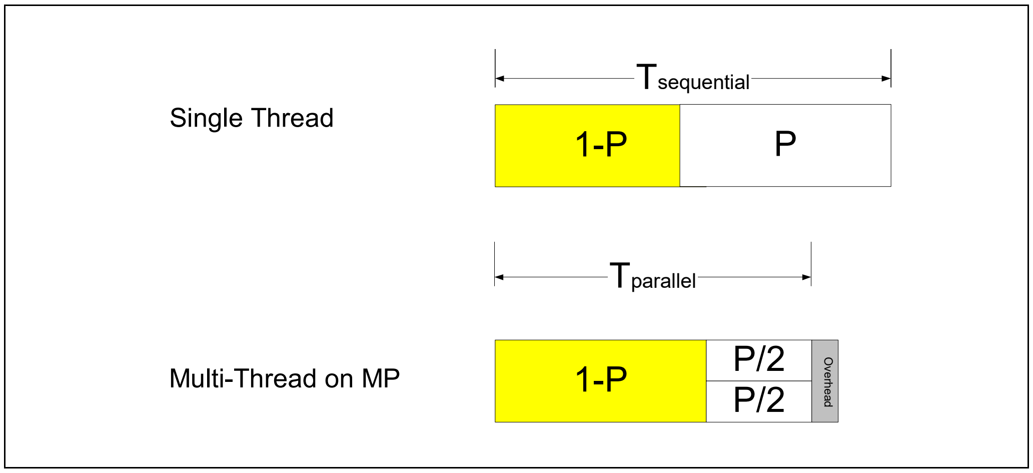


Figure 11-1. Amdahl's Law and MP Speed-up

When optimizing application performance in a multithreaded environment, control flow parallelism is likely to have the largest impact on performance scaling with respect to the number of physical processors and to the number of logical processors per physical processor.

If the control flow of a multi-threaded application contains a workload in which only 50% can be executed in parallel, the maximum performance gain using two physical processors is only 33%, compared to using a single processor. Using four processors can deliver no more than a 60% speed-up over a single processor. Thus, it is critical to maximize the portion of control flow that can take advantage of parallelism. Improper implementation of thread synchronization can significantly increase the proportion of serial control flow and further reduce the application's performance scaling.

In addition to maximizing the parallelism of control flows, interaction between threads in the form of thread synchronization and imbalance of task scheduling can also impact overall processor scaling significantly.

Excessive cache misses are one cause of poor performance scaling. In a multithreaded execution environment, they can occur from:

- Aliased stack accesses by different threads in the same process.
- Thread contentions resulting in cache line evictions.
- False-sharing of cache lines between different processors.

Techniques that address each of these situations (and many other areas) are described in sections in this chapter.

11.1.2 Multitasking Environment

Hardware multithreading capabilities in Intel 64 and IA-32 processors can exploit task-level parallelism when a workload consists of several single-threaded applications and these applications are scheduled to run concurrently under an MP-aware operating system. In this environment, hardware multithreading capabilities can deliver higher throughput for the workload, although the relative performance of a single task (in terms of time of completion relative to the same task when in a single-threaded environment) will vary, depending on how much shared execution resources and memory are utilized.

For development purposes, several popular operating systems (for example Microsoft Windows* XP Professional and Home, Linux* distributions using kernel 2.4.19 or later[1]) include OS kernel code that can manage the task scheduling and the balancing of shared execution resources within each physical processor to maximize the throughput.

Because applications run independently under a multitasking environment, thread synchronization issues are less likely to limit the scaling of throughput. This is because the control flow of the workload is likely to be 100% parallel[2] (if no inter-processor communication is taking place and if there are no system bus constraints).

With a multitasking workload, however, bus activities and cache access patterns are likely to affect the scaling of the throughput. Running two copies of the same application or same suite of applications in a lock-step can expose an artifact in performance measuring methodology. This is because an access pattern to the first level data cache can lead to excessive cache misses and produce skewed performance results. Fix this problem by:

- Including a per-instance offset at the start-up of an application.
- Introducing heterogeneity in the workload by using different datasets with each instance of the application.
- Randomizing the sequence of start-up of applications when running multiple copies of the same suite.

When two applications are employed as part of a multitasking workload, there is little synchronization overhead between these two processes. It is also important to ensure each application has minimal synchronization overhead within itself.

An application that uses lengthy spin loops for intra-process synchronization is less likely to benefit from HT Technology in a multitasking workload. This is because critical resources will be consumed by the long spin loops.

11.2 PROGRAMMING MODELS AND MULTITHREADING

Parallelism is the most important concept in designing a multithreaded application and realizing optimal performance scaling with multiple processors. An optimized multithreaded application is characterized by large degrees of parallelism or minimal dependencies in the following areas:

- Workload.
- Thread interaction.
- Hardware utilization.

The key to maximizing workload parallelism is to identify multiple tasks that have minimal inter-dependencies within an application and to create separate threads for parallel execution of those tasks.

Concurrent execution of independent threads is the essence of deploying a multithreaded application on a multiprocessing system. Managing the interaction between threads to minimize the cost of thread synchronization is also critical to achieving optimal performance scaling with multiple processors.

Efficient use of hardware resources between concurrent threads requires optimization techniques in specific areas to prevent contentions of hardware resources. Coding techniques for optimizing thread synchronization and managing other hardware resources are discussed in subsequent sections.

Parallel programming models are discussed next.

1. This code is included in Red Hat* Linux Enterprise AS 2.1.
2. A software tool that attempts to measure the throughput of a multitasking workload is likely to introduce control flows that are not parallel. Thread synchronization issues must be considered as an integral part of its performance measuring methodology.

11.2.1 Parallel Programming Models

Two common programming models for transforming independent task requirements into application threads are:

- Domain decomposition.
- Functional decomposition.

11.2.1.1 Domain Decomposition

Usually large compute-intensive tasks use data sets that can be divided into a number of small subsets, each having a large degree of computational independence. Examples include:

- Computation of a discrete cosine transformation (DCT) on two-dimensional data by dividing the two-dimensional data into several subsets and creating threads to compute the transform on each subset.
- Matrix multiplication; here, threads can be created to handle the multiplication of half of matrix with the multiplier matrix.

Domain Decomposition is a programming model based on creating identical or similar threads to process smaller pieces of data independently. This model can take advantage of duplicated execution resources present in a traditional multiprocessor system. It can also take advantage of shared execution resources between two logical processors in HT Technology. This is because a data domain thread typically consumes only a fraction of the available on-chip execution resources.

Section 11.3.4, “Key Practices of Execution Resource Optimization,” discusses additional guidelines that can help data domain threads use shared execution resources cooperatively and avoid the pitfalls creating contentions of hardware resources between two threads.

11.2.2 Functional Decomposition

Applications usually process a wide variety of tasks with diverse functions and many unrelated data sets. For example, a video codec needs several different processing functions. These include DCT, motion estimation and color conversion. Using a functional threading model, applications can program separate threads to do motion estimation, color conversion, and other functional tasks.

Functional decomposition will achieve more flexible thread-level parallelism if it is less dependent on the duplication of hardware resources. For example, a thread executing a sorting algorithm and a thread executing a matrix multiplication routine are not likely to require the same execution unit at the same time. A design recognizing this could advantage of traditional multiprocessor systems as well as multiprocessor systems using processors supporting HT Technology.

11.2.3 Specialized Programming Models

Intel Core Duo processor and processors based on Intel Core microarchitecture offer a second-level cache shared by two processor cores in the same physical package. This provides opportunities for two application threads to access some application data while minimizing the overhead of bus traffic.

Multi-threaded applications may need to employ specialized programming models to take advantage of this type of hardware feature. One such scenario is referred to as producer-consumer. In this scenario, one thread writes data into some destination (hopefully in the second-level cache) and another thread executing on the other core in the same physical package subsequently reads data produced by the first thread.

The basic approach for implementing a producer-consumer model is to create two threads; one thread is the producer and the other is the consumer. Typically, the producer and consumer take turns to work on a buffer and inform each other when they are ready to exchange buffers. In a producer-consumer model, there is some thread synchronization overhead when buffers are exchanged between the producer and consumer. To achieve optimal scaling with the number of cores, the synchronization overhead must be kept low. This can be done by ensuring the producer and consumer threads have comparable time constants for completing each incremental task prior to exchanging buffers.

Example 11-1 illustrates the coding structure of single-threaded execution of a sequence of task units, where each task unit (either the producer or consumer) executes serially (shown in Figure 11-2). In the equivalent scenario under multi-threaded execution, each producer-consumer pair is wrapped as a thread function and two threads can be scheduled on available processor resources simultaneously.

Example 11-1. Serial Execution of Producer and Consumer Work Items

```
for (i = 0; i < number_of_iterations; i++) {
    producer (i, buff);  // pass buffer index and buffer address
    consumer (i, buff);
}(
```

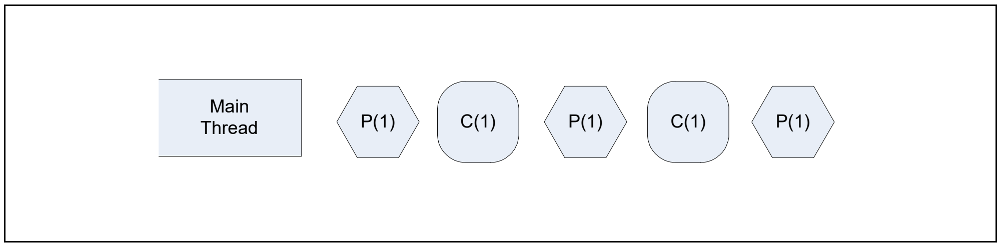


Figure 11-2. Single-threaded Execution of Producer-consumer Threading Model

11.2.3.1 Producer-Consumer Threading Models

Figure 11-3 illustrates the basic scheme of interaction between a pair of producer and consumer threads. The horizontal direction represents time. Each block represents a task unit, processing the buffer assigned to a thread.

The gap between each task represents synchronization overhead. The decimal number in the parenthesis represents a buffer index. On an Intel Core Duo processor, the producer thread can store data in the second-level cache to allow the consumer thread to continue work requiring minimal bus traffic.

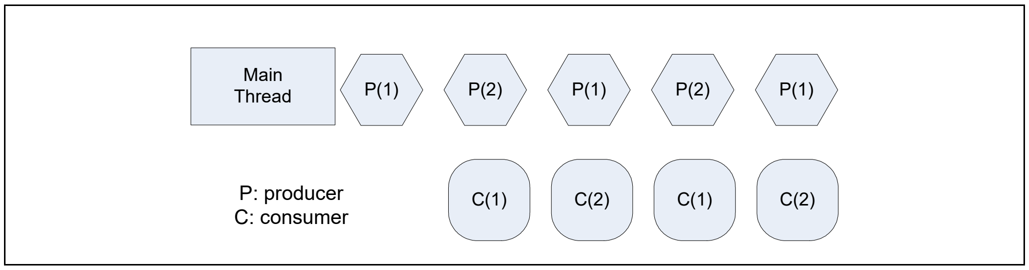


Figure 11-3. Execution of Producer-consumer Threading Model on a Multicore Processor

The basic structure to implement the producer and consumer thread functions with synchronization to communicate buffer index is shown in Example 11-2.

Example 11-2. Basic Structure of Implementing Producer Consumer Threads

```
(a) Basic structure of a producer thread function
void producer_thread()
{       int iter_num = workamount - 1; // make local copy
        int mode1 = 1; // track usage of two buffers via 0 and 1
        produce(buffs[0],count);  // placeholder function
        while (iter_num--) {

                Signal(&signal1,1);  // tell the other thread to commence
                produce(buffs[mode1],count); // placeholder function
                WaitForSignal(&end1);
                mode1 = 1 - mode1;  // switch to the other buffer
        }

}
b) Basic structure of a consumer thread
void consumer_thread()
{       int mode2 = 0;  // first iteration start with buffer 0, than alternate
        int iter_num = workamount - 1;
        while (iter_num--) {

                WaitForSignal(&signal1);
                consume(buffs[mode2],count); // placeholder function
                Signal(&end1,1);
                mode2 = 1 - mode2;
        }
        consume(buffs[mode2],count);
}
```

It is possible to structure the producer-consumer model in an interlaced manner such that it can minimize bus traffic and be effective on multicore processors without shared second-level cache.

In this interlaced variation of the producer-consumer model, each scheduling quanta of an application thread comprises of a producer task and a consumer task. Two identical threads are created to execute in parallel. During each scheduling quanta of a thread, the producer task starts first and the consumer task follows after the completion of the producer task; both tasks work on the same buffer. As each task completes, one thread signals to the other thread notifying its corresponding task to use its designated buffer. Thus, the producer and consumer tasks execute in parallel in two threads. As long as the data generated by the producer reside in either the first or second level cache of the same core, the consumer can access them without incurring bus traffic. The scheduling of the interlaced producer-consumer model is shown in Figure 11-4.

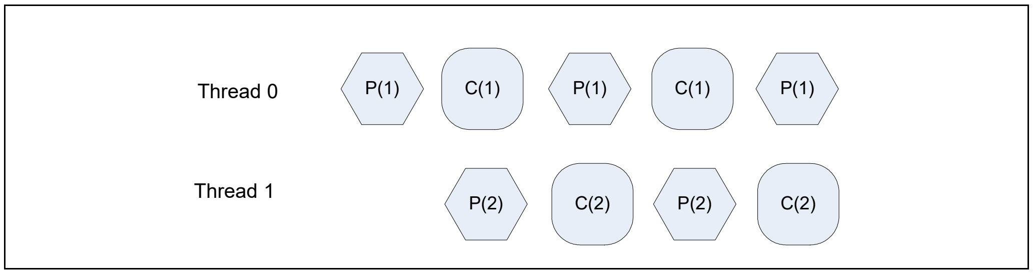


Figure 11-4. Interlaced Variation of the Producer Consumer Model

Example 11-3 shows the basic structure of a thread function that can be used in this interlaced producer-consumer model.

Example 11-3. Thread Function for an Interlaced Producer Consumer Model

```
// master thread starts first iteration, other thread must wait
// one iteration
void producer_consumer_thread(int master)
{
int mode = 1 - master; // track which thread and its designated
                                  // buffer index
unsigned int iter_num = workamount >> 1;
unsigned int i=0;

iter_num += master & workamount & 1;

   if (master)  // master thread starts the first iteration
   {
       produce(buffs[mode],count);
       Signal(sigp[1-mode1],1);  // notify producer task in follower
                                        // thread that it can proceed
    consume(buffs[mode],count);
       Signal(sigc[1-mode],1);
    i = 1;
   }



   for (; i < iter_num; i++)
   {
    WaitForSignal(sigp[mode]);
       produce(buffs[mode],count); // notify the producer task in
                                            // other thread
       Signal(sigp[1-mode],1);

   WaitForSignal(sigc[mode]);
    consume(buffs[mode],count);
       Signal(sigc[1-mode],1);
   }
}
```

11.2.4 Tools for Creating Multithreaded Applications

Programming directly to a multithreading application programming interface (API) is not the only method for creating multithreaded applications. New tools (such as the Intel compiler) have become available with capabilities that make the challenge of creating multithreaded application easier.

Features available in the latest Intel compilers are:

- Generating multithreaded code using OpenMP* directives[1].
- Generating multithreaded code automatically from unmodified high-level code[2].

1. Intel Compiler 5.0 and later supports OpenMP directives. Visit http://software.intel.com for details.
2. Intel Compiler 6.0 supports auto-parallelization.

11.2.4.1 Programming with OpenMP Directives

OpenMP provides a standardized, non-proprietary, portable set of Fortran and C++ compiler directives supporting shared memory parallelism in applications. OpenMP supports directive-based processing. This uses special preprocessors or modified compilers to interpret parallelism expressed in Fortran comments or C/C++ pragmas. Benefits of directive-based processing include:

- The original source can be compiled unmodified.
- It is possible to make incremental code changes. This preserves algorithms in the original code and enables rapid debugging.
- Incremental code changes help programmers maintain serial consistency. When the code is run on one processor, it gives the same result as the unmodified source code.
- Offering directives to fine tune thread scheduling imbalance.
- Intel's implementation of OpenMP runtime can add minimal threading overhead relative to hand-coded multithreading.

11.2.4.2 Automatic Parallelization of Code

While OpenMP directives allow programmers to quickly transform serial applications into parallel applications, programmers must identify specific portions of the application code that contain parallelism and add compiler directives. Intel Compiler 6.0 supports a new (-QPARALLEL) option, which can identify loop structures that contain parallelism. During program compilation, the compiler automatically attempts to decompose the parallelism into threads for parallel processing. No other intervention or programmer is needed.

11.2.4.3 Supporting Development Tools

See Appendix A, "Application Performance Tools" for information on the various tools that Intel provides for software development.

11.3 OPTIMIZATION GUIDELINES

This section summarizes optimization guidelines for tuning multithreaded applications. Five areas are listed (in order of importance):

- Thread synchronization.
- Bus utilization.
- Memory optimization.
- Front end optimization.
- Execution resource optimization.

Practices associated with each area are listed in this section. Guidelines for each area are discussed in greater depth in sections that follow.

Most of the coding recommendations improve performance scaling with processor cores; and scaling due to HT Technology. Techniques that apply to only one environment are noted.

11.3.1 Key Practices of Thread Synchronization

Key practices for minimizing the cost of thread synchronization are summarized below:

- Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.
- Replace a spin-lock that may be acquired by multiple threads with pipelined locks such that no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to acquire a lock.

- Use a thread-blocking API in a long idle loop to free up the processor.
- Prevent "false-sharing" of per-thread-data between two threads.
- Place each synchronization variable alone, separated by 128 bytes or in a separate cache line.

See Section 11.4, "Thread Synchronization," for details.

11.3.2 Key Practices of System Bus Optimization

Managing bus traffic can significantly impact the overall performance of multithreaded software and MP systems. Key practices of system bus optimization for achieving high data throughput and quick response are:

- Improve data and code locality to conserve bus command bandwidth.
- Avoid excessive use of software prefetch instructions and allow the automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.
- Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies.
- Use full write transactions to achieve higher data throughput.

See Section 11.5, "System Bus Optimization," for details.

11.3.3 Key Practices of Memory Optimization

Key practices for optimizing memory operations are summarized below:

- Use cache blocking to improve locality of data access. Target one quarter to one half of cache size when targeting processors supporting HT Technology.
- Minimize the sharing of data between threads that execute on different physical processors sharing a common bus.
- Minimize data access patterns that are offset by multiples of 64-KBytes in each thread.
- Adjust the private stack of each thread in an application so the spacing between these stacks is not offset by multiples of 64 KBytes or 1 MByte (prevents unnecessary cache line evictions) when targeting processors supporting HT Technology.
- Add a per-instance stack offset when two instances of the same application are executing in lock steps to avoid memory accesses that are offset by multiples of 64 KByte or 1 MByte when targeting processors supporting HT Technology.

See Section 11.6, "Memory Optimization," for details.

11.3.4 Key Practices of Execution Resource Optimization

Each physical processor has dedicated execution resources. Logical processors in physical processors supporting HT Technology share specific on-chip execution resources. Key practices for execution resource optimization include:

- Optimize each thread to achieve optimal frequency scaling first.
- Optimize multithreaded applications to achieve optimal scaling with respect to the number of physical processors.
- Use on-chip execution resources cooperatively if two threads are sharing the execution resources in the same physical processor package.
- For each processor supporting HT Technology, consider adding functionally uncorrelated threads to increase the hardware resource utilization of each physical processor package.

See Section 11.8, "Affinities and Managing Shared Platform Resources," for details.

11.3.5 Generality and Performance Impact

The next five sections cover the optimization techniques in detail. Recommendations discussed in each section are ranked by importance in terms of estimated local impact and generality.

Rankings are subjective and approximate. They can vary depending on coding style, application and threading domain. The purpose of including high, medium and low impact ranking with each recommendation is to provide a relative indicator as to the degree of performance gain that can be expected when a recommendation is implemented.

It is not possible to predict the likelihood of a code instance across many applications, so an impact ranking cannot be directly correlated to application-level performance gain. The ranking on generality is also subjective and approximate.

Coding recommendations that do not impact all three scaling factors are typically categorized as medium or lower.

11.4 THREAD SYNCHRONIZATION

Applications with multiple threads use synchronization techniques in order to ensure correct operation. However, thread synchronization that are improperly implemented can significantly reduce performance.

The best practice to reduce the overhead of thread synchronization is to start by reducing the application's requirements for synchronization. Intel Thread Profiler can be used to profile the execution timeline of each thread and detect situations where performance is impacted by frequent occurrences of synchronization overhead.

Several coding techniques and operating system (OS) calls are frequently used for thread synchronization. These include spin-wait loops, spin-locks, critical sections, to name a few. Choosing the optimal OS call for the circumstance and implementing synchronization code with parallelism in mind are critical in minimizing the cost of handling thread synchronization.

SSE3 provides two instructions (MONITOR/MWAIT) to help multithreaded software improve synchronization between multiple agents. In the first implementation of MONITOR and MWAIT, these instructions are available to operating system so that operating system can optimize thread synchronization in different areas. For example, an operating system can use MONITOR and MWAIT in its system idle loop (known as C0 loop) to reduce power consumption. An operating system can also use MONITOR and MWAIT to implement its C1 loop to improve the responsiveness of the C1 loop. See Chapter 9 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

11.4.1 Choice of Synchronization Primitives

Thread synchronization often involves modifying some shared data while protecting the operation using synchronization primitives. There are many primitives to choose from. Guidelines that are useful when selecting synchronization primitives are:

- Favor compiler intrinsics or an OS provided interlocked API for atomic updates of simple data operation, such as increment and compare/exchange. This will be more efficient than other more complicated synchronization primitives with higher overhead.

  For more information on using different synchronization primitives, see the white paper, *Developing Multi-threaded Applications: A Platform Consistent Approach*.
- When choosing between different primitives to implement a synchronization construct, using Intel Thread Checker and Thread Profiler can be very useful in dealing with multithreading functional correctness issue and performance impact under multi-threaded execution. Additional information on the capabilities of Intel Thread Checker and Thread Profiler are described in Appendix A.

Table 11-1 is useful for comparing the properties of three categories of synchronization objects available to multi-threaded applications.

Table 11-1. Properties of Synchronization Objects

Characteristics	Operating System Synchronization Objects	Light Weight User Synchronization	Synchronization Object based on MONITOR/MWAIT
Cycles to acquire and release (if there is a contention)	Thousands or Tens of thousands cycles	Hundreds of cycles	Hundreds of cycles
Power consumption	Saves power by halting the core or logical processor if idle	Some power saving if using PAUSE	Saves more power than PAUSE
Scheduling and context switching	Returns to the OS scheduler if contention exists (can be tuned with earlier spin loop count)	Does not return to OS scheduler voluntarily	Does not return to OS scheduler voluntarily
Ring level	Ring 0	Ring 3	Ring 0
Miscellaneous	Some objects provide intra-process synchronization and some are for inter-process communication	Must lock accesses to synchronization variable if several threads may write to it simultaneously. Otherwise can write without locks.	Same as light weight. Can be used only on systems supporting MONITOR/MWAIT
Recommended use conditions	▪ Number of active threads is greater than number of cores ▪ Waiting thousands of cycles for a signal ▪ Synchronization among processes	▪ Number of active threads is less than or equal to number of cores ▪ Infrequent contention ▪ Need inter process synchronization	▪ Same as light weight objects ▪ MONITOR/MWAIT available

11.4.2 Synchronization for Short Periods

The frequency and duration that a thread needs to synchronize with other threads depends application characteristics. When a synchronization loop needs very fast response, applications may use a spin-wait loop.

A spin-wait loop is typically used when one thread needs to wait a short amount of time for another thread to reach a point of synchronization. A spin-wait loop consists of a loop that compares a synchronization variable with some predefined value. See Example 11-4(a).

On a modern microprocessor with a superscalar speculative execution engine, a loop like this results in the issue of multiple simultaneous read requests from the spinning thread. These requests usually execute out-of-order with each read request being allocated a buffer resource. On detection of a write by a worker thread to a load that is in progress, the processor must guarantee no violations of memory order occur. The necessity of maintaining the order of outstanding memory operations inevitably costs the processor a severe penalty that impacts all threads.

This penalty occurs on the Pentium M processor, the Intel Core Solo and Intel Core Duo processors. However, the penalty on these processors is small compared with penalties suffered on the Pentium 4 and Intel Xeon processors. There the performance penalty for exiting the loop is about 25 times more severe.

On a processor supporting HT Technology, spin-wait loops can consume a significant portion of the execution bandwidth of the processor. One logical processor executing a spin-wait loop can severely impact the performance of the other logical processor.

Example 11-4. Spin-wait Loop and PAUSE Instructions

(a) An un-optimized spin-wait loop experiences performance penalty when exiting the loop. It consumes execution resources without contributing computational work.

```
do {
    // This loop can run faster than the speed of memory access,
    // other worker threads cannot finish modifying sync_var until
    // outstanding loads from the spinning loops are resolved.
} while( sync_var != constant_value);
```

(b) Inserting the PAUSE instruction in a fast spin-wait loop prevents performance-penalty to the spinning thread and the worker thread

```
do {
_asm   pause
    // Ensure this loop is de-pipelined, i.e. preventing more than one
    // load request to sync_var to be outstanding,
    // avoiding performance penalty when the worker thread updates
    // sync_var and the spinning thread exiting the loop.

}
while( sync_var != constant_value);
```

(c) A spin-wait loop using a "test, test-and-set" technique to determine the availability of the synchronization variable. This technique is recommended when writing spin-wait loops to run on Intel 64 and IA-32 architecture processors.

```
Spin_Lock:
    CMP lockvar, 0 ;            // Check if lock is free.
    JE  Get_lock
         PAUSE;                 // Short delay.
         JMP Spin_Lock;
Get_Lock:
         MOV EAX, 1;
         XCHG EAX, lockvar;     // Try to get lock.
         CMP EAX, 0;            // Test if successful.
         JNE Spin_Lock;
Critical_Section:
         <critical section code>
         MOV lockvar, 0;         // Release lock.
```

User/Source Coding Rule 13. (M impact, H generality) *Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.*

The penalty of exiting from a spin-wait loop can be avoided by inserting a PAUSE instruction in the loop. In spite of the name, the PAUSE instruction improves performance by introducing a slight delay in the loop and effectively causing the memory read requests to be issued at a rate that allows immediate detection of any store to the synchronization variable. This prevents the occurrence of a long delay due to memory order violation.

One example of inserting the PAUSE instruction in a simplified spin-wait loop is shown in Example 11-4(b). The PAUSE instruction is compatible with all Intel 64 and IA-32 processors. On IA-32 processors prior to Intel NetBurst microarchitecture, the PAUSE instruction is essentially a NOP instruction. Additional examples of optimizing spin-wait loops using the PAUSE instruction are available in Application note AP-949, Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor.

Inserting the PAUSE instruction has the added benefit of significantly reducing the power consumed during the spin-wait because fewer system resources are used.

11.4.3 Optimization with Spin-Locks

Spin-locks are typically used when several threads needs to modify a synchronization variable and the synchronization variable must be protected by a lock to prevent unintentional overwrites. When the lock is released, however, several threads may compete to acquire it at once. Such thread contention significantly reduces performance scaling with respect to frequency, number of discrete processors, and HT Technology.

To reduce the performance penalty, one approach is to reduce the likelihood of many threads competing to acquire the same lock. Apply a software pipelining technique to handle data that must be shared between multiple threads.

Instead of allowing multiple threads to compete for a given lock, no more than two threads should have write access to a given lock. If an application must use spin-locks, include the PAUSE instruction in the wait loop. Example 11-4(c) shows an example of the "test, test-and-set" technique for determining the availability of the lock in a spin-wait loop.

User/Source Coding Rule 14. (M impact, L generality) *Replace a spin lock that may be acquired by multiple threads with pipelined locks such that no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to use a lock.*

11.4.4 Synchronization for Longer Periods

When using a spin-wait loop not expected to be released quickly, an application should follow these guidelines:

- Keep the duration of the spin-wait loop to a minimum number of repetitions.
- Applications should use an OS service to block the waiting thread; this can release the processor so that other runnable threads can make use of the processor or available execution resources.

On processors supporting HT Technology, operating systems should use the HLT instruction if one logical processor is active and the other is not. HLT will allow an idle logical processor to transition to a halted state; this allows the active logical processor to use all the hardware resources in the physical package. An operating system that does not use this technique must still execute instructions on the idle logical processor that repeatedly check for work. This "idle loop" consumes execution resources that could otherwise be used to make progress on the other active logical processor.

If an application thread must remain idle for a long time, the application should use a thread blocking API or other method to release the idle processor. The techniques discussed here apply to traditional MP system, but they have an even higher impact on processors that support HT Technology.

Typically, an operating system provides timing services, for example Sleep(*dwMilliseconds*)[1]; such variables can be used to prevent frequent checking of a synchronization variable.

Another technique to synchronize between worker threads and a control loop is to use a thread-blocking API provided by the OS. Using a thread-blocking API allows the control thread to use less processor cycles for spinning and waiting. This gives the OS more time quanta to schedule the worker threads on available processors. Furthermore, using a thread-blocking API also benefits from the system idle loop optimization that OS implements using the HLT instruction.

User/Source Coding Rule 15. (H impact, M generality) *Use a thread-blocking API in a long idle loop to free up the processor.*

Using a spin-wait loop in a traditional MP system may be less of an issue when the number of runnable threads is less than the number of processors in the system. If the number of threads in an application is expected to be greater than the number of processors (either one processor or multiple processors), use a thread-blocking API to free up processor resources. A multithreaded application adopting one control thread to synchronize multiple worker threads may consider limiting worker threads to the number of processors in a system and use thread-blocking APIs in the control thread.

1. The Sleep() API is not thread-blocking, because it does not guarantee the processor will be released. Example 11-5(a) shows an example of using Sleep(0), which does not always realize the processor to another thread.

11.4.4.1 Avoid Coding Pitfalls in Thread Synchronization

Synchronization between multiple threads must be designed and implemented with care to achieve good performance scaling with respect to the number of discrete processors and the number of logical processor per physical processor. No single technique is a universal solution for every synchronization situation.

The pseudo-code example in Example 11-5(a) illustrates a polling loop implementation of a control thread. If there is only one runnable worker thread, an attempt to call a timing service API, such as Sleep(0), may be ineffective in minimizing the cost of thread synchronization. Because the control thread still behaves like a fast spinning loop, the only runnable worker thread must share execution resources with the spin-wait loop if both are running on the same physical processor that supports HT Technology. If there are more than one runnable worker threads, then calling a thread blocking API, such as Sleep(0), could still release the processor running the spin-wait loop, allowing the processor to be used by another worker thread instead of the spinning loop.

A control thread waiting for the completion of worker threads can usually implement thread synchronization using a thread-blocking API or a timing service, if the worker threads require significant time to complete. Example 11-5(b) shows an example that reduces the overhead of the control thread in its thread synchronization.

Example 11-5. Coding Pitfall using Spin Wait Loop

```
(a) A spin-wait loop attempts to release the processor incorrectly. It experiences a performance penalty if the only
worker thread and the control thread runs on the same physical processor package.
// Only one worker thread is running,
//  the control loop waits for the worker thread to complete.

ResumeWorkThread(thread_handle);
While (!task_not_done ) {
   Sleep(0)   // Returns immediately back to spin loop.
 ...
}
(b) A polling loop frees up the processor correctly.

// Let a worker thread run and wait for completion.
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
  Sleep(FIVE_MILISEC)

//  This processor is released for some duration, the processor
//  can be used by other threads.
...
}
```

In general, OS function calls should be used with care when synchronizing threads. When using OS-supported thread synchronization objects (critical section, mutex, or semaphore), preference should be given to the OS service that has the least synchronization overhead, such as a critical section.

11.4.5 Prevent Sharing of Modified Data and False-Sharing

Depending on the cache topology relative to processor/core topology and the specific underlying microarchitecture, sharing of modified data can incur some degree of performance penalty when a software thread running on one core tries to read or write data that is currently present in modified state in the local cache of another core. This will cause eviction of the modified cache line back into memory and reading it into the first-level cache of the other core. The latency of such cache line transfer is much higher than using data in the immediate first level cache or second level cache.

False sharing applies to data used by one thread that happens to reside on the same cache line as different data used by another thread. These situations can also incur a performance delay depending on the topology of the logical processors/cores in the platform.

False sharing can experience a performance penalty when the threads are running on logical processors reside on different physical processors or processor cores. For processors that support HT Technology, false-sharing incurs a performance penalty when two threads run on different cores, different physical processors, or on two logical processors in the physical processor package. In the first two cases, the performance penalty is due to cache evictions to maintain cache coherency. In the latter case, performance penalty is due to memory order machine clear conditions.

A generic approach for multi-threaded software to prevent incurring false-sharing penalty is to allocate separate critical data or locks with alignment granularity according to a “false-sharing threshold” size. The following steps will allow software to determine the “false-sharing threshold” across Intel processors:

1. If the processor supports CLFLUSH instruction, i.e. CPUID.01H:EDX.CLFLUSH[bit 19] =1:

   Use the CLFLUSH line size, i.e. the integer value of CPUID.01H:EBX[15:8], as the “false-sharing threshold”.
2. If CLFLUSH line size is not available, use CPUID leaf 4 as described below:

   Determine the “false-sharing threshold” by evaluating the largest system coherency line size among valid cache types that are reported via the sub-leaves of CPUID leaf 4. For each sub-leaf n, its associated system coherency line size is (CPUID.(EAX=4, ECX=n):EBX[11:0] + 1).
3. If neither CLFLUSH line size is available, nor CPUID leaf 4 is available, then software may choose the “false-sharing threshold” from one of the following:
   a. Query the descriptor tables of CPUID leaf 2 and choose from available descriptor entries.
   b. A Family/Model-specific mechanism available in the platform or a Family/Model-specific known value.
   c. Default to a safe value 64 bytes.

User/Source Coding Rule 16. (H impact, M generality) *Beware of false sharing within a cache line or within a sector. Allocate critical data or locks separately using alignment granularity not smaller than the “false-sharing threshold”.*

When a common block of parameters is passed from a parent thread to several worker threads, it is desirable for each work thread to create a private copy (each copy aligned to multiples of the *“false-sharing threshold”)* of frequently accessed data in the parameter block.

11.4.6 Placement of Shared Synchronization Variable

On processors based on Intel NetBurst microarchitecture, bus reads typically fetch 128 bytes into a cache, the optimal spacing to minimize eviction of cached data is 128 bytes. To prevent false-sharing, synchronization variables and system objects (such as a critical section) should be allocated to reside alone in a 128-byte region and aligned to a 128-byte boundary.

Example 11-6 shows a way to minimize the bus traffic required to maintain cache coherency in MP systems. This technique is also applicable to MP systems using processors with or without HT Technology.

Example 11-6. Placement of Synchronization and Regular Variables

```
int regVar;
int padding[32];
int SynVar[32*NUM_SYNC_VARS];
int AnotherVar;
```

On Pentium M, Intel Core Solo, Intel Core Duo processors, and processors based on Intel Core microarchitecture; a synchronization variable should be placed alone and in separate cache line to avoid false-sharing. Software must not allow a synchronization variable to span across page boundary.

User/Source Coding Rule 17. (M impact, ML generality) *Place each synchronization variable alone, separated by 128 bytes or in a separate cache line.*

User/Source Coding Rule 18. (H impact, L generality) *Do not place any spin lock variable to span a cache line boundary.*

At the code level, false sharing is a special concern in the following cases:

- Global data variables and static data variables that are placed in the same cache line and are written by different threads.
- Objects allocated dynamically by different threads may share cache lines. Make sure that the variables used locally by one thread are allocated in a manner to prevent sharing the cache line with other threads.

Another technique to enforce alignment of synchronization variables and to avoid a cacheline being shared is to use compiler directives when declaring data structures. See Example 11-7.

Example 11-7. Declaring Synchronization Variables without Sharing a Cache Line

```
__declspec(align(64)) unsigned __int64 sum;
struct sync_struct {…};
__declspec(align(64)) struct sync_struct sync_var;
```

Other techniques that prevent false-sharing include:

- Organize variables of different types in data structures (because the layout that compilers give to data variables might be different than their placement in the source code).
- When each thread needs to use its own copy of a set of variables, declare the variables with:
  - — Directive threadprivate, when using OpenMP.
  - — Modifier __declspec (thread), when using Microsoft compiler.
- In managed environments that provide automatic object allocation, the object allocators and garbage collectors are responsible for layout of the objects in memory so that false sharing through two objects does not happen.
- Provide classes such that only one thread writes to each object field and close object fields, in order to avoid false sharing.

One should not equate the recommendations discussed in this section as favoring a sparsely populated data layout. The data-layout recommendations should be adopted when necessary and avoid unnecessary bloat in the size of the work set.

11.5 SYSTEM BUS OPTIMIZATION

The system bus services requests from bus agents (e.g. logical processors) to fetch data or code from the memory sub-system. The performance impact due data traffic fetched from memory depends on the characteristics of the workload, and the degree of software optimization on memory access, locality enhancements implemented in the software code. A number of techniques to characterize memory traffic of a workload is discussed in Appendix A. Optimization guidelines on locality enhancement is also discussed in Section 3.6.10, "Locality Enhancement," and Section 9.5.11, "Hardware Prefetching and Cache Blocking Techniques."

The techniques described in Chapter 3 and Chapter 9 benefit application performance in a platform where the bus system is servicing a single-threaded environment. In a multi-threaded environment, the bus system typically services many more logical processors, each of which can issue bus requests independently. Thus, techniques on locality enhancements, conserving bus bandwidth, reducing large-stride-cache-miss-delay can have strong impact on processor scaling performance.

11.5.1 Conserve Bus Bandwidth

In a multithreading environment, bus bandwidth may be shared by memory traffic originated from multiple bus agents (These agents can be several logical processors and/or several processor cores). Preserving the bus bandwidth can improve processor scaling performance. Also, effective bus bandwidth typically will decrease if there are significant large-stride cache-misses. Reducing the amount of large-stride cache misses (or reducing DTLB misses) will alleviate the problem of bandwidth reduction due to large-stride cache misses.

One way for conserving available bus command bandwidth is to improve the locality of code and data. Improving the locality of data reduces the number of cache line evictions and requests to fetch data. This technique also reduces the number of instruction fetches from system memory.

User/Source Coding Rule 19. (M impact, H generality) *Improve data and code locality to conserve bus command bandwidth.*

Using a compiler that supports profiler-guided optimization can improve code locality by keeping frequently used code paths in the cache. This reduces instruction fetches. Loop blocking can also improve the data locality. Other locality enhancement techniques can also be applied in a multithreading environment to conserve bus bandwidth (see Section 9.5, “Memory Optimization Using Prefetch”).

Because the system bus is shared between many bus agents (logical processors or processor cores), software tuning should recognize symptoms of the bus approaching saturation. One useful technique is to examine the queue depth of bus read traffic. When the bus queue depth is high, locality enhancement to improve cache utilization will benefit performance more than other techniques, such as inserting more software prefetches or masking memory latency with overlapping bus reads. An approximate working guideline for software to operate below bus saturation is to check if bus read queue depth is significantly below 5.

Some MP and workstation platforms may have a chipset that provides two system buses, with each bus servicing one or more physical processors. The guidelines for conserving bus bandwidth described above also applies to each bus domain.

11.5.2 Understand the Bus and Cache Interactions

Be careful when parallelizing code sections with data sets that results in the total working set exceeding the second-level cache and /or consumed bandwidth exceeding the capacity of the bus. On an Intel Core Duo processor, if only one thread is using the second-level cache and / or bus, then it is expected to get the maximum benefit of the cache and bus systems because the other core does not interfere with the progress of the first thread. However, if two threads use the second-level cache concurrently, there may be performance degradation if one of the following conditions is true:

- Their combined working set is greater than the second-level cache size.
- Their combined bus usage is greater than the capacity of the bus.
- They both have extensive access to the same set in the second-level cache, and at least one of the threads writes to this cache line.

To avoid these pitfalls, multithreading software should try to investigate parallelism schemes in which only one of the threads access the second-level cache at a time, or where the second-level cache and the bus usage does not exceed their limits.

11.5.3 Avoid Excessive Software Prefetches

Pentium 4 and Intel Xeon Processors have an automatic hardware prefetcher. It can bring data and instructions into the unified second-level cache based on prior reference patterns. In most situations, the hardware prefetcher is likely to reduce system memory latency without explicit intervention from software prefetches. It is also preferable to adjust data access patterns in the code to take advantage of the characteristics of the automatic hardware prefetcher to improve locality or mask memory latency. Processors based on Intel Core microarchitecture also provides several advanced hardware prefetching mechanisms. Data access patterns that can take advantage of earlier generations of hardware prefetch mechanism generally can take advantage of more recent hardware prefetch implementations.

Using software prefetch instructions excessively or indiscriminately will inevitably cause performance penalties. This is because excessively or indiscriminately using software prefetch instructions wastes the command and data bandwidth of the system bus.

Using software prefetches delays the hardware prefetcher from starting to fetch data needed by the processor core. It also consumes critical execution resources and can result in stalled execution. In some cases, it may be fruitful to evaluate the reduction or removal of software prefetches to migrate towards more effective use of hardware prefetch mechanisms. The guidelines for using software prefetch instructions are described in Chapter 3. The techniques for using automatic hardware prefetcher is discussed in Chapter 9.

User/Source Coding Rule 20. (M impact, L generality) *Avoid excessive use of software prefetch instructions and allow automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.*

11.5.4 Improve Effective Latency of Cache Misses

System memory access latency due to cache misses is affected by bus traffic. This is because bus read requests must be arbitrated along with other requests for bus transactions. Reducing the number of outstanding bus transactions helps improve effective memory access latency.

One technique to improve effective latency of memory read transactions is to use multiple overlapping bus reads to reduce the latency of sparse reads. In situations where there is little locality of data or when memory reads need to be arbitrated with other bus transactions, the effective latency of scattered memory reads can be improved by issuing multiple memory reads back-to-back to overlap multiple outstanding memory read transactions. The average latency of back-to-back bus reads is likely to be lower than the average latency of scattered reads interspersed with other bus transactions. This is because only the first memory read needs to wait for the full delay of a cache miss.

User/Source Coding Rule 21. (M impact, M generality) *Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies.*

Another technique to reduce effective memory latency is possible if one can adjust the data access pattern such that the access strides causing successive cache misses in the last-level cache is predominantly less than the trigger threshold distance of the automatic hardware prefetcher. See Section 9.5.3, "Example of Effective Latency Reduction with Hardware Prefetch."

User/Source Coding Rule 22. (M impact, M generality) *Consider adjusting the sequencing of memory references such that the distribution of distances of successive cache misses of the last level cache peaks towards 64 bytes.*

11.5.5 Use Full Write Transactions to Achieve Higher Data Rate

Write transactions across the bus can result in write to physical memory either using the full line size of 64 bytes or less than the full line size. The latter is referred to as a partial write. Typically, writes to write-back (WB) memory addresses are full-size and writes to write-combine (WC) or uncacheable (UC) type memory addresses result in partial writes. Both cached WB store operations and WC store operations utilize a set of six WC buffers (64 bytes wide) to manage the traffic of write transactions. When competing traffic closes a WC buffer before all writes to the buffer are finished, this results in a series of 8-byte partial bus transactions rather than a single 64-byte write transaction.

User/Source Coding Rule 23. (M impact, M generality) *Use full write transactions to achieve higher data throughput.*

Frequently, multiple partial writes to WC memory can be combined into full-sized writes using a software write-combining technique to separate WC store operations from competing with WB store traffic. To implement software write-combining, uncacheable writes to memory with the WC attribute are written to a small, temporary buffer (WB type) that fits in the first level data cache. When the temporary buffer is full, the application copies the content of the temporary buffer to the final WC destination.

When partial-writes are transacted on the bus, the effective data rate to system memory is reduced to only 1/8 of the system bus bandwidth.

11.6 MEMORY OPTIMIZATION

Efficient operation of caches is a critical aspect of memory optimization. Efficient operation of caches needs to address the following:

- Cache blocking.
- Shared memory optimization.
- Eliminating 64-KByte aliased data accesses.
- Preventing excessive evictions in first-level cache.

11.6.1 Cache Blocking Technique

Loop blocking is useful for reducing cache misses and improving memory access performance. The selection of a suitable block size is critical when applying the loop blocking technique. Loop blocking is applicable to single-threaded applications as well as to multithreaded applications running on processors with or without HT Technology. The technique transforms the memory access pattern into blocks that efficiently fit in the target cache size.

When targeting Intel processors supporting HT Technology, the loop blocking technique for a unified cache can select a block size that is no more than one half of the target cache size, if there are two logical processors sharing that cache. The upper limit of the block size for loop blocking should be determined by dividing the target cache size by the number of logical processors available in a physical processor package. Typically, some cache lines are needed to access data that are not part of the source or destination buffers used in cache blocking, so the block size can be chosen between one quarter to one half of the target cache (see Chapter 3, “General Optimization Guidelines”).

Software can use the deterministic cache parameter leaf of CPUID to discover which subset of logical processors are sharing a given cache (see Chapter 9, “Optimizing Cache Usage”). Therefore, guideline above can be extended to allow all the logical processors serviced by a given cache to use the cache simultaneously, by placing an upper limit of the block size as the total size of the cache divided by the number of logical processors serviced by that cache. This technique can also be applied to single-threaded applications that will be used as part of a multitasking workload.

User/Source Coding Rule 24. (H impact, H generality) *Use cache blocking to improve locality of data access. Target one quarter to one half of the cache size when targeting Intel processors supporting HT Technology or target a block size that allow all the logical processors serviced by a cache to share that cache simultaneously.*

11.6.2 Shared-Memory Optimization

Maintaining cache coherency between discrete processors frequently involves moving data across a bus that operates at a clock rate substantially slower that the processor frequency.

11.6.2.1 Minimize Sharing of Data between Physical Processors

When two threads are executing on two physical processors and sharing data, reading from or writing to shared data usually involves several bus transactions (including snooping, request for ownership changes, and sometimes fetching data across the bus). A thread accessing a large amount of shared memory is likely to have poor processor-scaling performance.

User/Source Coding Rule 25. (H impact, M generality) *Minimize the sharing of data between threads that execute on different bus agents sharing a common bus. The situation of a platform consisting of multiple bus domains should also minimize data sharing across bus domains.*

One technique to minimize sharing of data is to copy data to local stack variables if it is to be accessed repeatedly over an extended period. If necessary, results from multiple threads can be combined later by writing them back to a shared memory location. This approach can also minimize time spent to synchronize access to shared data.

11.6.2.2 Batched Producer-Consumer Model

The key benefit of a threaded producer-consumer design, shown in Figure 11-5, is to minimize bus traffic while sharing data between the producer and the consumer using a shared second-level cache. On an Intel Core Duo processor and when the work buffers are small enough to fit within the first-level cache, re-ordering of producer and consumer tasks are necessary to achieve optimal performance. This is because fetching data from L2 to L1 is much faster than having a cache line in one core invalidated and fetched from the bus.

Figure 11-5 illustrates a batched producer-consumer model that can be used to overcome the drawback of using small work buffers in a standard producer-consumer model. In a batched producer-consumer model, each scheduling quanta batches two or more producer tasks, each producer working on a designated buffer. The number of tasks to batch is determined by the criteria that the total working set be greater than the first-level cache but smaller than the second-level cache.

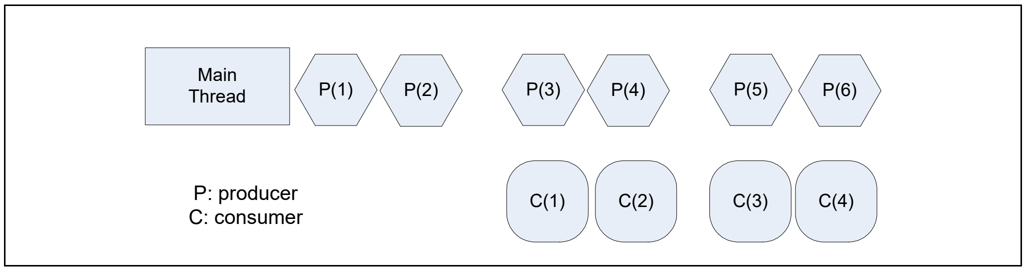


Figure 11-5. Batched Approach of Producer Consumer Model

Example 11-8 shows the batched implementation of the producer and consumer thread functions.

Example 11-8. Batched Implementation of the Producer Consumer Threads

```
void producer_thread()
{    int iter_num = workamount - batchsize;
     int mode1;
for (mode1=0; mode1 < batchsize; mode1++)
{          produce(buffs[mode1],count);  }

     while (iter_num--)
     {     Signal(&signal1,1);
           produce(buffs[mode1],count); // placeholder function
           WaitForSignal(&end1);
           mode1++;
           if (mode1 > batchsize)
                 mode1 = 0;
     }
}
```

Example 11-8. Batched Implementation of the Producer Consumer Threads (Contd.)

```
void consumer_thread()
{    int mode2 = 0;
     int iter_num = workamount - batchsize;
     while (iter_num--)
     {     WaitForSignal(&signal1);
           consume(buffs[mode2],count); // placeholder function
           Signal(&end1,1);
           mode2++;
           if (mode2 > batchsize)
                 mode2 = 0;

     }
     for (i=0;i<batchsize;i++)
     {     consume(buffs[mode2],count);
           mode2++;
           if (mode2 > batchsize)
                 mode2 = 0;
     }
}
```

11.6.3 Eliminate 64-KByte Aliased Data Accesses

The 64-KByte aliasing condition is discussed in Chapter 3. Memory accesses that satisfy the 64-KByte aliasing condition can cause excessive evictions of the first-level data cache. Eliminating 64-KByte aliased data accesses originating from each thread helps improve frequency scaling in general. Furthermore, it enables the first-level data cache to perform efficiently when HT Technology is fully utilized by software applications.

User/Source Coding Rule 26. (H impact, H generality) *Minimize data access patterns that are offset by multiples of 64 KBytes in each thread.*

The presence of 64-KByte aliased data access can be detected using Pentium 4 processor performance monitoring events. Appendix B includes an updated list of Pentium 4 processor performance metrics. These metrics are based on events accessed using the Intel VTune Performance Analyzer.

Performance penalties associated with 64-KByte aliasing are applicable mainly to current processor implementations of HT Technology or Intel NetBurst microarchitecture. The next section discusses memory optimization techniques that are applicable to multithreaded applications running on processors supporting HT Technology.

11.7 FRONT END OPTIMIZATION

For dual-core processors where the second-level unified cache is shared by two processor cores (Intel Core Duo processor and processors based on Intel Core microarchitecture), multi-threaded software should consider the increase in code working set due to two threads fetching code from the unified cache as part of front end and cache optimization. For quad-core processors based on Intel Core microarchitecture, the considerations that applies to Intel Core 2 Duo processors also apply to quad-core processors.

11.7.1 Avoid Excessive Loop Unrolling

Unrolling loops can reduce the number of branches and improve the branch predictability of application code. Loop unrolling is discussed in detail in Chapter 3. Loop unrolling must be used judiciously. Be sure to consider the benefit of improved branch predictability and the cost of under-utilization of the loop stream detector (LSD).

***User/Source Coding Rule 27. (M impact, L generality)** Avoid excessive loop unrolling to ensure the LSD is operating efficiently.*

11.8 AFFINITIES AND MANAGING SHARED PLATFORM RESOURCES

Modern OSes provide either API and/or data constructs (e.g. affinity masks) that allow applications to manage certain shared resources , e.g. logical processors, Non-Uniform Memory Access (NUMA) memory sub-systems.

Before multithreaded software considers using affinity APIs, it should consider the recommendations in Table 11-2.

Table 11-2. Design-Time Resource Management Choices

Runtime Environment	Thread Scheduling/Processor Affinity Consideration	Memory Affinity Consideration
A single-threaded application	Support OS scheduler objectives on system response and throughput by letting OS scheduler manage scheduling. OS provides facilities for end user to optimize runtime specific environment.	Not relevant; let OS do its job.
A multi-threaded application requiring: **i) less than all processor resource in the system,** **ii) share system resource with other concurrent applications,** **iii) other concurrent applications may have higher priority.**	Rely on OS default scheduler policy. Hard-coded affinity-binding will likely harm system response and throughput; and/or in some cases hurting application performance.	Rely on OS default scheduler policy. Use API that could provide transparent NUMA benefit without managing NUMA explicitly.
A multi-threaded application requiring **i) foreground and higher priority,** **ii) uses less than all processor resource in the system,** **iii) share system resource with other concurrent applications,** **iv) but other concurrent applications have lower priority.**	If application-customized thread binding policy is considered, a cooperative approach with OS scheduler should be taken instead of hard-coded thread affinity binding policy. For example, the use of SetThreadIdealProcessor() can provide a floating base to anchor a next-free-core binding policy for locality-optimized application binding policy, and cooperate with default OS policy.	Use API that could provide transparent NUMA benefit without managing NUMA explicitly. Use performance event to diagnose non-local memory access issue if default OS policy cause performance issue.

Table 11-2. Design-Time Resource Management Choices (Contd.)

Runtime Environment	Thread Scheduling/Processor Affinity Consideration	Memory Affinity Consideration
A multithreaded application runs in foreground, requiring all processor resource in the system and not sharing system resource with concurrent applications; multithreading.	Application-customized thread binding policy can be more efficient than default OS policy. Use performance event to help optimize locality and cache transfer opportunities. A multithreaded application that employs its own explicit thread affinity-binding policy should deploy with some form of opt-in choice granted by the end-user or administrator. For example, permission to deploy explicit thread affinity-binding policy can be activated after permission is granted after installation.	Application-customized memory affinity binding policy can be more efficient than default OS policy. Use performance event to diagnose non-local memory access issues related to either OS or custom policy

11.8.1 Topology Enumeration of Shared Resources

Whether multithreaded software ride on OS scheduling policy or need to use affinity APIs for customized resource management, understanding the topology of the shared platform resource is essential. The processor topology of logical processors (SMT), processor cores, and physical processors in the platform can enumerated using information provided by CPUID. This is discussed in Chapter 9, “Multiple-Processor Management” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. A white paper and reference code is also available from Intel.

11.8.2 Non-Uniform Memory Access

Platforms using two or more Intel Xeon processors based on Nehalem microarchitecture support non-uniform memory access (NUMA) topology because each physical processor provides its own local memory controller. NUMA offers system memory bandwidth that can scale with the number of physical processors. System memory latency will exhibit asymmetric behavior depending on the memory transaction occurring locally in the same socket or remotely from another socket. Additionally, OS-specific construct and/or implementation behavior may present additional complexity at the API level that the multi-threaded software may need to pay attention to memory allocation/initialization in a NUMA environment.

Generally, latency sensitive workload would favor memory traffic to stay local over remote. If multiple threads shares a buffer, the programmer will need to pay attention to OS-specific behavior of memory allocation/initialization on a NUMA system.

Bandwidth sensitive workloads will find it convenient to employ a data composition threading model and aggregates application threads executing in each socket to favor local traffic on a per-socket basis to achieve overall bandwidth scalable with the number of physical processors.

The OS construct that provides the programming interface to manage local/remote NUMA traffic is referred to as memory affinity. Because OS manages the mapping between physical address (populated by system RAM) to linear address (accessed by application software); and paging allows dynamic reassignment of a physical page to map to different linear address dynamically, proper use of memory affinity will require a great deal of OS-specific knowledge.

To simplify application programming, OS may implement certain APIs and physical/linear address mapping to take advantage of NUMA characteristics transparently in certain situations. One common technique is for OS to delay commit of physical memory page assignment until the first memory reference on that physical page is accessed in the linear address space by an application thread. This means that the allocation of a memory buffer in the linear address space by an application thread does not necessarily determine which socket will service local memory traffic when the memory allocation API returns to the program. However, the memory allocation API that supports this level of NUMA transparency varies across different OSes. For example, the portable C-language API "malloc" provides some degree of transparency on Linux*, whereas the API "VirtualAlloc" behave similarly on Windows*. Different OSes may also provide memory allocation APIs that require explicit NUMA information, such that the mapping between linear address to local/remote memory traffic are fixed at allocation.

Example 11-9 shows an example that multi-threaded application could undertake the least amount of effort dealing with OS-specific APIs and to take advantage of NUMA hardware capability. This parallel approach to memory buffer initialization is conducive to having each worker thread keep memory traffic local on NUMA systems.

Example 11-9. Parallel Memory Initialization Technique Using OpenMP and NUMA

```
#ifdef _LINUX // Linux implements malloc to commit physical page at first touch/access
    buf1 = (char *) malloc(DIM*(sizeof (double))+1024);
    buf2 = (char *) malloc(DIM*(sizeof (double))+1024);
    buf3 = (char *) malloc(DIM*(sizeof (double))+1024);
#endif
#ifdef windows
// Windows implements malloc to commit physical page at allocation, so use VirtualAlloc
    buf1 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
    buf2 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
    buf3 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
#endif
            (continue)
    a = (double *) buf1;
    b = (double *) buf2;
    c = (double *) buf3;
#pragma omp parallel
{ // use OpenMP threads to execute each iteration of the loop
// number of OpenMP threads can be specified by default or via environment variable
#pragma omp for private(num)
// each loop iteration is dispatched to execute in different OpenMP threads using private iterator
    for(num=0;num<len;num++)
    {// each thread perform first-touches to its own subset of memory address, physical pages
//          mapped to the local memory controller of the respective threads
            a[num]=10.;
            b[num]=10.;
            c[num]=10.;
    }
}
```

Note that the example shown in Example 11-9 implies that the memory buffers will be freed after the worker threads created by OpenMP have ended. This situation avoids a potential issue of repeated use of malloc/free across different application threads. Because if the local memory that was initialized by one thread and subsequently got freed up by another thread, the OS may have difficulty in tracking/re-allocating memory pools in linear address space relative to NUMA topology. In Linux, another API, "numa_local_alloc" may be used.

11.9 OPTIMIZATION OF OTHER SHARED RESOURCES

Resource optimization in multithreaded application depends on the cache topology and execution resources associated within the hierarchy of processor topology. Processor topology and an algorithm for software to identify the processor topology are discussed in Chapter 9, "Multiple-Processor Management" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

In platforms with shared buses, the bus system is shared by multiple agents at the SMT level and at the processor core level of the processor topology. Thus multithreaded application design should start with an approach to manage the bus bandwidth available to multiple processor agents sharing the same bus link in an equitable manner. This can be done by improving the data locality of an individual application thread or allowing two threads to take advantage of a shared second-level cache (where such shared cache topology is available).

In general, optimizing the building blocks of a multithreaded application can start from an individual thread. The guidelines discussed in Chapter 3 through Chapter 13 largely apply to multithreaded optimization.

Tuning Suggestion 2. *Optimize single threaded code to maximize execution throughput first.*

Tuning Suggestion 3. *Employ efficient threading model, leverage available tools (such as Intel Threading Building Block, Intel Thread Checker, Intel Thread Profiler) to achieve optimal processor scaling with respect to the number of physical processors or processor cores.*

11.9.1 Expanded Opportunity for Intel® HT Optimization

The Intel® Hyper-Threading Technology (Intel® HT) implementation in Nehalem microarchitecture differs from previous generations of Intel HT implementations. It offers broader opportunity for multithreaded software to take advantage of Intel HT and achieve higher system throughput over a broader range of application problems. This section provides a few heuristic recommendations and illustrates some of these optimization opportunities.

Chapter 2, "Intel® 64 and IA-32 Architectures" covered some of the microarchitectural capability enhancements in Intel Hyper-Threading Technology. Many of these enhancements center around the basic needs of multi-threaded software in terms of sharing common hardware resources that may be used by more than one thread context.

Different software algorithms and workload characteristics may produce different performance characteristics due to their demands on critical microarchitectural resources that may be shared amongst several logical processors. A brief comparison of the various microarchitectural subsystems that can play a significant role in software tuning for Intel HT is summarized in Table 11-3.

Table 11-3. Microarchitectural Resources Comparisons of Intel® HT Implementations

Microarchitectural Subsystem	Nehalem Microarchitecture 06_1AH	NetBurst Microarchitecture 0F_02H, 0F_03H, 0F_04H, 0F_06H
Issue ports, execution units	Three issue ports (0, 1, 5) distributed to handle ALU, SIMD, and FP computations.	Unbalanced ports, fast ALU SIMD and FP sharing the same port (port 1).
Buffering	More entries in ROB, RS, fill buffers, etc., with moderate pipeline depths.	Less balance between buffer entries and pipeline depths.
Branch Prediction and Misaligned memory access	More robust speculative execution with immediate reclamation after misprediction; efficient handling of cache splits.	More microarchitectural hazards resulting in pipeline cleared for both threads.
Cache hierarchy	Larger and more efficient.	More microarchitectural hazards to work around.

Table 11-3. Microarchitectural Resources Comparisons of Intel® HT Implementations

Microarchitectural Subsystem	Nehalem Microarchitecture 06_1AH	NetBurst Microarchitecture 0F_02H, 0F_03H, 0F_04H, 0F_06H
Memory and bandwidth	NUMA, three channels per socket to DDR3, up to 32GB/s per socket.	SMP, FSB, or dual FSB, up to 12.8 GB/s per FSB.

For compute bound workloads, the Intel HT opportunity in Intel NetBurst microarchitecture tends to favor thread contexts that executes with relatively high CPI (average cycles to retire consecutive instructions). At a hardware level, this is in part due to the issue port imbalance in the microarchitecture, as port 1 is shared by fast ALU, slow ALU (more heavy-duty integer operations), SIMD, and FP computations. At a software level, some of the cause for high CPI and may appear as benign catalyst for providing HT benefit may include: long latency instructions (port 1), some L2 hits, occasional branch mispredictions, etc. But the length of the pipeline in NetBurst microarchitecture often impose additional internal hardware constraints that limits software's ability to take advantage of Intel HT.

The microarchitectural enhancements listed in Table 11-3 are expected to provide broader software optimization opportunities for compute-bound workloads. Whereas contention in the same execution unit by two compute-bound threads might be a concern to choose a functional-decomposition threading model over data-composition threading. Nehalem microarchitecture will likely be more accommodating to support the programmer to choose the optimal threading decomposition models.

Memory intensive workloads can exhibit a wide range of performance characteristics, ranging from completely parallel memory traffic (saturating system memory bandwidth, as in the well-known example of Stream), memory traffic dominated by memory latency, or various mixtures of compute operations and memory traffic of either kind.

The Intel HT implementation in Intel NetBurst microarchitecture may provide benefit to some of the latter two types of workload characteristics. The HT capability in the Nehalem microarchitecture can broaden the operating envelop of the two latter types of workload characteristics to deliver higher system throughput, due to its support for non-uniform memory access (NUMA), more efficient link protocol, and system memory bandwidth that scales with the number of physical processors.

Some cache levels of the cache hierarchy may be shared by multiple logical processors. Using the cache hierarchy is an important means for software to improve the efficiency of memory traffic and avoid saturating the system memory bandwidth. Multi-threaded applications employing cache-blocking technique may wish to partition a target cache level to take advantage of Intel Hyper-Threading Technology. Alternately two logical processors sharing the same L1 and L2, or logical processors sharing the L3 may wish to manage the shared resources according to their relative topological relationship. A white paper on processor topology enumeration and cache topology enumeration with companion reference code has been published (see reference in Chapter 1).

7. Updates to Chapter 15

Change bars and **violet** text show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Optimization Resource Manual:* Optimizations for Intel® AVX, FMA, and AVX2.

Changes to this chapter:

- Typo corrections where necessary.
- Section 15.12 Corrected Cross-Reference.
- Section 15.13, modified reference to Example 15-31 for clarity.

CHAPTER 15

OPTIMIZATIONS FOR INTEL® AVX, INTEL® AVX2, AND INTEL® FMA

Intel® Advanced Vector Extension (Intel® AVX), is a major enhancement to Intel Architecture. It extends the functionality of previous generations of 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) vector instructions and increased the vector register width to support 256-bit operations. The Intel AVX ISA enhancement is focused on float-point instructions. Some 256-bit integer vectors are supported via floating-point to integer and integer to floating-point conversions.

Sandy Bridge microarchitecture implements the Intel AVX instructions, in most cases, on 256-bit hardware. Thus, each core has 256-bit floating-point Add and Multiply units. The Divide and Square-root units are not enhanced to 256-bits. Thus, Intel AVX instructions use the 128-bit hardware in two steps to complete these 256-bit operations.

Prior generations of Intel® SSE instructions generally are two-operand syntax, where one of the operands serves both as source and as destination. Intel AVX instructions are encoded with a VEX prefix, which includes a bit field to encode vector lengths and support three-operand syntax. A typical instruction has two sources and one destination. Four operand instructions such as VBLENDVPS and VBLENDVPD exist as well. The added operand enables non-destructive source (NDS) and it eliminates the need for register duplication using MOVAPS operations.

With the exception of MMX™ instructions, almost all legacy 128-bit Intel SSE instructions have Intel AVX equivalents that support three operand syntax. 256-bit Intel AVX instructions employ three-operand syntax and some with 4-operand syntax.

The 256-bit vector register **YMM** extends the 128-bit **XMM** register to 256 bits. Thus the lower 128-bits of YMM is aliased to the legacy XMM registers.

While 256-bit Intel AVX instructions writes 256 bits of results to YMM, 128-bit Intel AVX instructions writes 128-bits of results into the XMM register and zeros the upper bits above bit 128 of the corresponding YMM. 16 vector registers are available in 64-bit mode. Only the lower 8 vector registers are available in non-64-bit modes.

Software can continue to use any mixture of legacy Intel SSE code, 128-bit Intel AVX code and 256-bit Intel AVX code. Section covers guidelines to deliver optimal performance across mixed-vector-length code modules without experiencing transition delays between legacy Intel SSE and Intel AVX code. There are no transition delays of mixing 128-bit Intel AVX code and 256-bit Intel AVX code.

The optimal memory alignment of an Intel AVX 256-bit vector, stored in memory, is 32 bytes. Some data-movement 256-bit Intel AVX instructions enforce 32-byte alignment and will signal #GP fault if memory operand is not properly aligned. The majority of 256-bit Intel AVX instructions do not require address alignment. These instructions generally combine load and compute operations, so any non-aligned memory address can be used in these instructions.

For best performance, software should pay attention to align the load and store addresses to 32 bytes whenever possible.

The major differences between using Intel AVX instructions and legacy Intel SSE instructions are summarized in Table 15-1.

Table 15-1. Features between 256-bit Intel® AVX, 128-bit Intel® AVX, and Legacy Intel® SSE Extensions

Features	256-bit AVX	128-bit AVX	Legacy SSE-AESNI
Functionality Scope	Floating-point operation, Data Movement.	Matches legacy SIMD ISA (except MMX).	128-bit FP and integer SIMD ISA.
Register Operand	YMM.	XMM.	XMM.
Operand Syntax	Up to 4; non-destructive source.	Up to 4; non-destructive source.	2 operand syntax; destructive source.
Memory alignment	Load-Op semantics do not require alignment.	Load-Op semantics do not require alignment.	Always enforce 16B alignment.
Aligned Move Instructions	32 byte alignment.	16 byte alignment.	16 byte alignment.
Non-destructive source operand	Yes.	Yes.	No.
Register State Handling	Updates bits 255:0.	Updates 127:0; Zeroes bits above 128.	Updates 127:0; Bits above 128 unmodified.
Intrinsic Support	▪ New 256-bit data types. ▪ _mm256 prefix for promoted functionality. ▪ New intrinsics for new functionalities.	▪ Existing data types. ▪ Inherit same prototype for exiting functionalities. ▪ Use "_mm" prefix for new VEX-128 functionalities.	Baseline datatypes and prototype definitions.
128-bit Lanes	Applies to most 256-bit operations.	One 128-bit lane.	One 128-bit lane.
Mixed Code Handling	Use VZEROUPPER to avoid transition penalty.	No transition penalty.	Transition penalty after executing 256-bit AVX code.

15.1 INTEL® AVX INTRINSICS CODING

256-bit Intel AVX instructions have new intrinsics. Specifically, 256-bit Intel AVX instruction that are promoted to 256-bit vector length from existing Intel SSE functionality are generally prototyped with a "_mm256" prefix instead of the "_mm" prefix and using new data types defined for 256-bit operation. New functionality in 256-bit AVX instructions have brand new prototype.

The 128-bit Intel AVX instruction that were promoted from legacy SIMD ISA uses the same prototype as before. Newer functionality common in 256-bit and 128-bit AVX instructions are prototyped with "_mm256" and "_mm" prefixes respectively.

Thus porting from legacy SIMD code written in intrinsic can be ported to 256-bit Intel AVX code with a modest effort.

The following guidelines show how to convert a simple intrinsic from Intel SSE code sequence to Intel AVX:

- Align statically and dynamically allocated buffers to 32-bytes.
- May need to double supplemental buffer size.
- Change __mm_ intrinsic name prefix with __mm256_.
- Change variable data types names from __m128 to __m256.
- Divide by 2 iteration count (or double stride length).

This example below on Cartesian coordinate transformation demonstrates the Intel AVX Instruction format, 32 byte YMM registers, dynamic and static memory allocation with data alignment of 32bytes, and the C data type representing 8 floating-point elements in a YMM register.

Example 15-1. Cartesian Coordinate Transformation with Intrinsics

```
//Use SSE intrinsic
#include "wmmintrin.h"

int main()
{ int len = 3200;
  //Dynamic memory allocation with 16byte
  //alignment
  float* pInVector = (float*) _mm_malloc(len*sizeof(float),
16);
  float* pOutVector = (float*) _mm_malloc(len*sizeof(float),
16);
//init data
for(int i=0; i<len; i++) pInVector[i] = 1;

float cos_theta = 0.8660254037;
  float sin_theta = 0.5;
//Static memory allocation of 4 floats with 16byte
alignment
__declspec(align(16)) float cos_sin_theta_vec[4] =
{cos_theta, sin_theta, cos_theta, sin_theta};

__declspec(align(16)) float sin_cos_theta_vec[4] =
{sin_theta, cos_theta, sin_theta, cos_theta};

//__m128 data type represents an xmm
  //register with 4 float elements
  __m128 Xmm_cos_sin =
_mm_load_ps(cos_sin_theta_vec);

  //SSE 128bit packed single load
  __m128 Xmm_sin_cos =
_mm_load_ps(sin_cos_theta_vec);

  __m128 Xmm0, Xmm1, Xmm2, Xmm3;
//processing 8 elements in an unrolled twice loop

for(int i=0; i<len; i+=8)
  {
    Xmm0 = _mm_load_ps(pInVector+i);
    Xmm1 = _mm_moveldup_ps(Xmm0);
    Xmm2 = _mm_movehdup_ps(Xmm0);
    Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
    Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
    Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
    _mm_store_ps(pOutVector + i, Xmm3);
```

```
// Use Intel AVX intrinsic
#include "immintrin.h"

int main()
{ int len = 3200;
  //Dynamic memory allocation with 32byte
  //alignment
  float* pInVector = (float*) _mm_malloc(len*sizeof(float),
32);
  float* pOutVector = (float*) _mm_malloc(len*sizeof(float),
32);
//init data
for(int i=0; i<len; i++) pInVector[i] = 1;

float cos_theta = 0.8660254037;
  float sin_theta = 0.5;
//Static memory allocation of 8 floats with 32byte
alignment
__declspec(align(32)) float cos_sin_theta_vec[8] =
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
sin_theta, cos_theta, sin_theta};

__declspec(align(32)) float sin_cos_theta_vec[8] =
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
cos_theta, sin_theta, cos_theta };

  //__m256 data type holds 8 float elements
  __m256 Ymm_cos_sin = _mm256_-
load_ps(cos_sin_theta_vec);

  //AVX 256bit packed single load
  __m256 Ymm_sin_cos = _mm256_-
load_ps(sin_cos_theta_vec);

  __m256 Ymm0, Ymm1, Ymm2, Ymm3;

  //processing 8 elements in an unrolled twice loop

for(int i=0; i<len; i+=16)
  {
    Ymm0 = _mm256_load_ps(pInVector+i);
    Ymm1 = _mm256_moveldup_ps(Ymm0);
    Ymm2 = _mm256_movehdup_ps(Ymm0);
    Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
    Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
    Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
    _mm256_store_ps(pOutVector + i, Ymm3);
```

Example 15-1. Cartesian Coordinate Transformation with Intrinsics (Contd.)

```
    Xmm0 = _mm_load_ps(pInVector+i+4);
    Xmm1 = _mm_moveldup_ps(Xmm0);
    Xmm2 = _mm_movehdup_ps(Xmm0);
    Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
    Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
    Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
    _mm_store_ps(pOutVector+i+4, Xmm3);
  }
_mm_free(pInVector);
  _mm_free(pOutVector);
return 0;
}
```

```
    Ymm0 = _mm256_load_ps(pInVector+i+8);
    Ymm1 = _mm256_moveldup_ps(Ymm0);
    Ymm2 = _mm256_movehdup_ps(Ymm0);
    Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
    Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
    Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
    _mm256_store_ps(pOutVector+i+8, Ymm3);
  }
_mm_free(pInVector);
  _mm_free(pOutVector);
return 0;
}
```

15.1.1 Intel® AVX Assembly Coding

Similar to the intrinsic porting guidelines, assembly porting guidelines are listed below.

- Align statically and dynamically allocated buffers to 32-bytes.
- Double the supplemental buffer sizes if needed.
- Add a "v" prefix to instruction names.
- Change register names from xmm to ymm.
- Add destination registers to computational Intel AVX instructions.
- Divide the iteration count by two (or double stride length).

Example 15-2. Cartesian Coordinate Transformation with Assembly

```
//Use SSE Assembly
int main()
{
  int len = 3200;
  //Dynamic memory allocation with 16byte
  //alignment
  float* pInVector = (float*) _mm_malloc(len*sizeof(float),
16);
  float* pOutVector = (float*) _mm_malloc(len*sizeof(float),
16);

//init data
for(int i=0; i<len; i++)
    pInVector[i] = 1;

  //Static memory allocation of 4 floats
  //with 16byte alignment
  float cos_theta = 0.8660254037;
  float sin_theta = 0.5;
  __declspec(align(16)) float cos_sin_theta_vec[4] =
{cos_theta, sin_theta, cos_theta, sin_theta};
```

```
// Use Intel AVX assembly
int main()
{
  int len = 3200;
  //Dynamic memory allocation with 32byte
  //alignment
  float* pInVector = (float*) _mm_malloc(len*sizeof(float),
32);
  float* pOutVector = (float*) _mm_malloc(len*sizeof(float),
32);

//init data
for(int i=0; i<len; i++)
    pInVector[i] = 1;

  //Static memory allocation of 8 floats
  //with 32byte alignment
  float cos_theta = 0.8660254037;
  float sin_theta = 0.5;
  __declspec(align(32)) float cos_sin_theta_vec[8] =
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
sin_theta, cos_theta, sin_theta};
```

Example 15-2. Cartesian Coordinate Transformation with Assembly (Contd.)

```
 __declspec(align(16)) float sin_cos_theta_vec[4] =
{sin_theta, cos_theta, sin_theta, cos_theta};

//processing 8 elements in an unrolled-twice loop
__asm
{
  mov rax, pInVector
  mov rbx, pOutVector
// Load into an xmm register of 16 bytes
  movups xmm3,
        xmmword ptr[cos_sin_theta_vec]
  movups xmm4,
        xmmword ptr[sin_cos_theta_vec]

  mov rdx, len
  shl rdx, 2          //size of input array in bytes
  xor rcx, rcx
loop1:
  movsldup  xmm0, [rax+rcx]
  movshdup  xmm1, [rax+rcx]
//example: mulps has 2 operands
  mulps xmm0, xmm3
  mulps xmm1, xmm4
  addsubps xmm0, xmm1
// 16 byte store from an xmm register
  movaps [rbx+rcx], xmm0

  movsldup  xmm0, [rax+rcx+16]
  movshdup  xmm1, [rax+rcx+16]
  mulps xmm0, xmm3
  mulps xmm1, xmm4
  addsubps xmm0, xmm1
// offset of 16 bytes from previous store
  movaps [rbx+rcx+16], xmm0

// Processed 32bytes in this loop
//(The code is unrolled twice)
  add rcx, 32
  cmp rcx, rdx
  jl loop1
}
  _mm_free(pInVector);
  _mm_free(pOutVector);
  return 0;
}
```

```
 __declspec(align(32)) float sin_cos_theta_vec[8] =
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
cos_theta, sin_theta, cos_theta};

//processing 16 elements in an unrolled-twice loop
__asm
 {
   mov rax, pInVector
   mov rbx, pOutVector
// Load into an ymm register of 32 bytes
   vmovups ymm3,
         ymmword ptr[cos_sin_theta_vec]
   vmovups ymm4,
         ymmword ptr[sin_cos_theta_vec]

   mov rdx, len
   shl rdx, 2        //size of input array in bytes
   xor rcx, rcx
loop1:
   vmovsldup  ymm0, [rax+rcx]
   vmovshdup  ymm1, [rax+rcx]
//example: vmulps has 3 operands
   vmulps ymm0, ymm0, ymm3
   vmulps ymm1, ymm1, ymm4
   vaddsubps ymm0, ymm0, ymm1
// 32 byte store from an ymm register
   vmovaps [rbx+rcx], ymm0

   vmovsldup  ymm0, [rax+rcx+32]
   vmovshdup  ymm1, [rax+rcx+32]
   vmulps ymm0, ymm0, ymm3
   vmulps ymm1, ymm1, ymm4
   vaddsubps ymm0, ymm0, ymm1
// offset of 32 bytes from previous store
   vmovaps [rbx+rcx+32], ymm0

// Processed 64bytes in this loop
//(The code is unrolled twice)
   add rcx, 64
   cmp rcx, rdx
   jl loop1
 }
 _mm_free(pInVector);
 _mm_free(pOutVector);
 return 0;
}
```

15.2 NON-DESTRUCTIVE SOURCE (NDS)

Most Intel AVX instructions have three operands. A typical instruction has two sources and one destination, with both source operands unmodified by the instruction. This section describes how using the NDS feature to save register copies, reduce the amount of instructions, reduce the amount of micro-ops, and improve performance. In this example, the Intel AVX code is more than 2x faster than the Intel SSE code.

The following example uses a vectorized calculation of the polynomial A^3+A^2+A. The polynomial calculation pseudo code is:

```
While (i<len)
{
 B[i] := A[i]^3 + A[i]^2 + A[i]
 i++
}
```

In Example 15-3, the left column shows the vectorized implementation using Intel SSE assembly. In this code, A is copied by an additional load from memory to a register, and A2 is copied using a register to register assignment. The code uses 10 micro-ops to process four elements.

The middle column in this example uses 128-bit Intel AVX instructions and takes advantage of NDS. The additional load and register copies are eliminated. This code uses 8 micro-ops to process four elements and is about 30% faster than the baseline above.

The right column in this example uses 256-bit AVX instructions. It uses 8 micro-ops to process 8 elements. Combining the NDS feature with the doubling of vector width, this speeds up the baseline by more than 2x.

Example 15-3. Direct Polynomial Calculation

SSE Code	128-bit AVX Code	256-bit AVX Code
float* pA = InputBuffer;	float* pA = InputBuffer1;	float* pA = InputBuffer1;
float* pB = OutputBuffer;	float* pB = OutputBuffer1;	float* pB = OutputBuffer1;
int len = miBufferWidth-4;	int len = miBufferWidth-4;	int len = miBufferWidth-8;
__asm	__asm	__asm
{	{	{
mov rax, pA	mov rax, pA	mov rax, pA
mov rbx, pB	mov rbx, pB	mov rbx, pB
movsxd r8, len	movsxd r8, len	movsxd r8, len
loop1:	loop1:	loop1:
//Load A	//Load A	//Load A
movups xmm0, [rax+r8*4]	vmovups xmm0, [rax+r8*4]	vmovups ymm0, [rax+r8*4]
//Copy A		
movups xmm1, [rax+r8*4]		
//A^2	//A^2	//A^2
mulps xmm1, xmm1	vmulps xmm1, xmm0, xmm0	vmulps ymm1, ymm0, ymm0
//Copy A^2		
movupsxmm2, xmm1		
//A^3	//A^3	//A^3
mulps xmm2, xmm0	vmulps xmm2, xmm1, xmm0	vmulps ymm2, ymm1, ymm0
//A + A^2	//A+A^2	//A+A^2
addps xmm0, xmm1	vaddps xmm0, xmm0, xmm1	vaddps ymm0, ymm0, ymm1
//A + A^2 + A^3	//A+A^2+A^3	//A+A^2+A^3
addps xmm0, xmm2	vaddps xmm0, xmm0, xmm2	vaddps ymm0, ymm0, ymm2
//Store result	//Store result	//Store result
movups[rbx+r8*4], xmm0	vmovups[rbx+r8*4], xmm0	vmovups [rbx+r8*4], ymm0

Example 15-3. Direct Polynomial Calculation (Contd.)

SSE Code	128-bit AVX Code	256-bit AVX Code
sub r8, 4 jge loop1 }	sub r8, 4 jge loop1 }	sub r8, 8 jge loop1 }

15.3 MIXING AVX CODE WITH SSE CODE

The Intel AVX architecture allows programmers to port a large code base gradually, resulting in mixed AVX code and SSE code. If your code includes both Intel AVX and Intel SSE, consider the following:

- Recompilation of SSE code with the Intel compiler and the option "/QxAVX" in Windows or "-xAVX" in Linux. This transforms all SSE instructions to 128-bit AVX instructions automatically. This refers to inline assembly and intrinsic code. "GCC -c -mAVX" will generate AVX code, including assembly files. GCC assembler also supports "-msse2avx" switch to generate AVX code from Intel SSE.
- Intel AVX and Intel SSE code can co-exist and execute in the same run. This can happen if your application includes third party libraries with Intel SSE code, a new DLL using Intel AVX code is deployed that calls other modules running Intel SSE code, or you cannot recompile all your application at once. In these cases, the Intel AVX code must use the VZEROUPPER instruction to avoid AVX/SSE transition penalty.

Intel AVX instructions always modify the upper bits of YMM registers and Intel SSE instructions do not modify the upper bits. From a hardware perspective, the upper bits of the YMM register collection can be considered to be in one of three states:

- Clean: All upper bits of YMM are zero. This is the state when the processor starts from RESET.
- Modified and Unsaved (In Table 15-2, this is abbreviated as M/U): The execution of one Intel AVX instruction (either 256-bit or 128-bit) modifies the upper bits of the destination YMM. This is also referred to as dirty upper YMM state. In this state, bits 255:128 and bits 127:0 of a given YMM are related to the most recent 256-bit or 128-bit AVX instruction that operated on that register.
- Preserved/Non_INIT Upper State (In Table 15-2, this is abbreviated as P/N): In this state, the upper YMM state is not zero. The upper 128 bits of a YMM and the lower 128 bits may be unrelated to the last Intel AVX instruction executed in the processor as a result of XRSTOR from a saved image with dirty upper YMM state.

If software inter-mixes Intel AVX and Intel SSE instructions without using VZEROUPPER properly, it can experience an Intel AVX/Intel SSE transition penalty. The situations of executing Intel SSE, Intel AVX, or managing the YMM state using XSAVE/XRSTOR/VZEROUPPER/VZEROALL is illustrated in Figure 15-1. The penalty associated with transitions into or out of the processor state "Modified and Unsaved" is implementation specific, depending on the microarchitecture.

Figure 15-1 depicts the situations that a transition penalty will occur for recent generations of microarchitectures that support Intel AVX, up to and including the Broadwell microarchitecture. The transition penalty of A and B occurs with each instruction execution that would cause the transition. It is largely the cost of copying the entire YMM state to internal storage.

To minimize the occurrence of YMM state transitions related to the "Preserved/Non_INIT Upper State", software that uses XSAVE/XRSTOR family of instructions to save/restore the YMM state should write a "Clean" upper YMM state to the XSAVE region in memory. Restoring a dirty YMM image from memory into the YMM registers can experience a penalty. This is illustrated in Figure 15-1.

The Skylake microarchitecture implements a different state machine than prior generations to manage the YMM state transition associated with mixing Intel SSE and Intel AVX instructions. It no longer saves the entire upper YMM state when executing an Intel SSE instruction when in "Modified and Unsaved" state, but saves the upper bits of individual register. As a result, mixing Intel SSE and Intel AVX instructions will experience a penalty associated with partial register dependency of the destination registers being used and additional blend operation on the upper bits of the destination registers. Figure 15-2 depicts the transition penalty applicable to the Skylake microarchitecture.

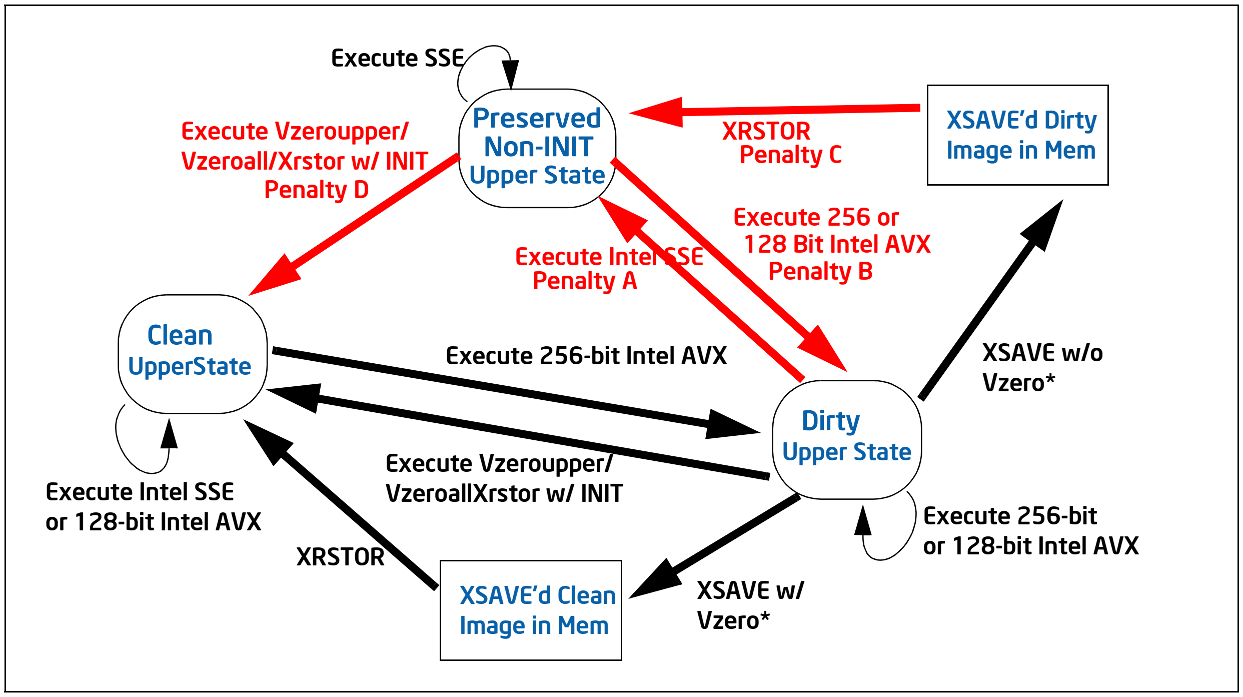

Execute SSE
Preserved Non-INIT Upper State
XSAVE'd Dirty Image in Mem
XRSTOR Penalty C
Execute Vzeroupper/ Vzeroall/Xrstor w/ INIT Penalty D
Execute 256 or 128 Bit Intel AVX Penalty B
Execute Intel SSE Penalty A
Clean UpperState
Execute 256-bit Intel AVX
XSAVE w/o Vzero*
Dirty Upper State
Execute Vzeroupper/ VzeroallXrstor w/ INIT
Execute Intel SSE or 128-bit Intel AVX
Execute 256-bit or 128-bit Intel AVX
XRSTOR
XSAVE'd Clean Image in Mem
XSAVE w/ Vzero*

Figure 15-1. Intel® AVX—Intel® SSE Transitions in the Broadwell, and Prior Generation Microarchitectures

Table 15-2 lists the effect of mixing Intel AVX and Intel SSE code, with the bottom row indicating the types of penalty that might arise depending on the initial YMM state (the row marked 'Begin') and the ending state. Table 15-2 also includes the effect of transition penalty (Type C and D) associated with restoring a dirty YMM state image stored in memory.

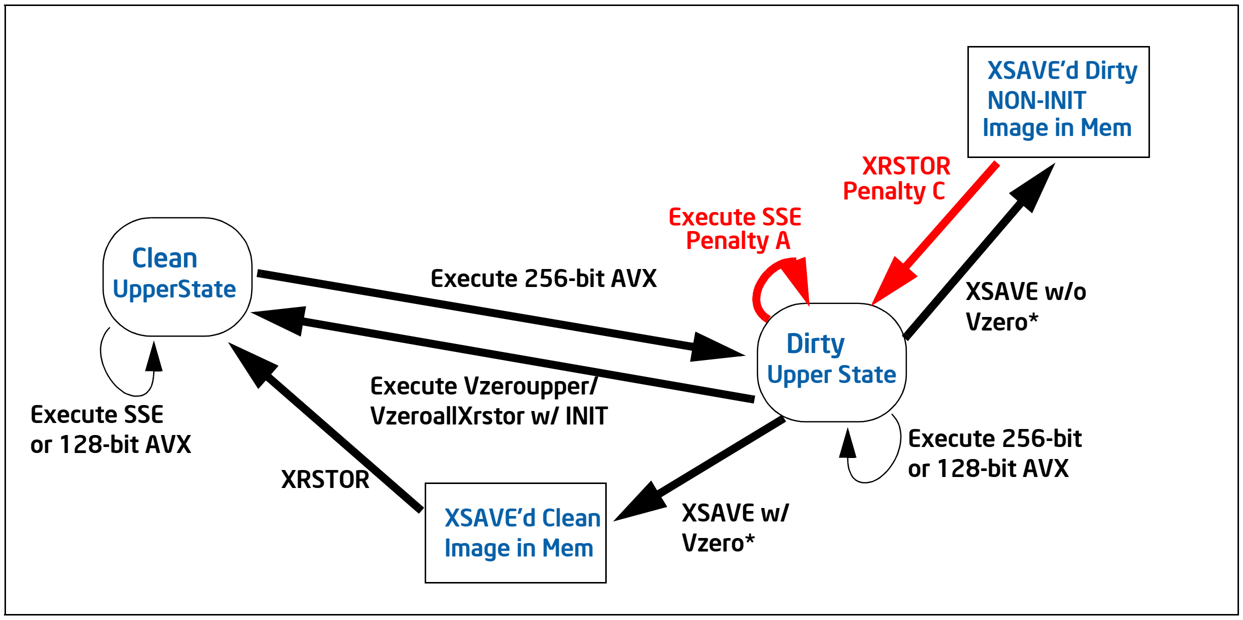

XSAVE'd Dirty NON-INIT Image in Mem
XRSTOR Penalty C
Execute SSE Penalty A
Clean UpperState
Execute 256-bit AVX
XSAVE w/o Vzero*
Dirty Upper State
Execute Vzeroupper/ VzeroallXrstor w/ INIT
Execute SSE or 128-bit AVX
Execute 256-bit or 128-bit AVX
XRSTOR
XSAVE'd Clean Image in Mem
XSAVE w/ Vzero*

Figure 15-2. Intel® AVX- Intel® SSE Transitions in the Skylake Microarchitecture

Table 15-2. State Transitions of Mixing AVX and SSE Code

	Execute SSE			Execute AVX-128			Execute AVX-256			VZeroupper	XRSTOR	
Begin	Clean	M/U	P/N	Clean	M/U	P/S	Clean	M/U	P/N	P/N	Dirty Image	Clean Image
End	Clean	P/N	P/N	Clean	M/U	M/U	M/U	M/U	M/U	Clean	P/N	Clean
Penalty	No	A	No	No	No	B	No	No	B	D	C	No

The magnitude of each type of transition penalty can vary across different microarchitectures. In Skylake microarchitecture, some of the transition penalty is reduced. The transition diagram and associated penalty is depicted in Figure 15-2. Table 15-3 gives approximate order of magnitude of the different transition penalty types across recent microarchitectures.

Table 15-3. Approximate Magnitude of Intel® AVX—Intel® SSE Transition Penalties in Different Microarchitectures

Type	Haswell Microarchitecture	Broadwell Microarchitecture	Skylake Microarchitecture	Ice Lake Client Microarchitecture
A	~XSAVE	~XSAVE	Partial Register Dependency + Blend	~XSAVE
B	~XSAVE	~XSAVE	NA	~XSAVE
C	~Fraction of XSAVE	~Fraction of XSAVE	~XSAVE	~Fraction of XSAVE
D	~XSAVE	~XSAVE	NA	~XSAVE

To enable fast transitions between 256-bit Intel AVX and Intel SSE code blocks, use the VZEROUPPER instruction before and after an AVX code block that would need to switch to execute SSE code. The VZEROUPPER instruction resets the upper 128 bits of all Intel AVX registers. This instruction has zero latency. In addition, the processor changes back to a Clean state, after which execution of SSE instructions or Intel AVX instructions has no transition penalty with prior microarchitectures. In Skylake microarchitecture, the SSE block can executed from a Clean state without the penalty of upper-bits dependency and blend operation.

128-bit Intel AVX instructions zero the upper 128-bits of the destination registers. Therefore, 128-bit and 256-bit Intel AVX instructions can be mixed with no penalty.

Assembly/Compiler Coding Rule 56. (H impact, H generality) *Whenever a 256-bit AVX code block and 128-bit SSE code block might execute in sequence, use the VZEROUPPER instruction to facilitate a transition to a "Clean" state for the next block to execute from.*

15.3.1 Mixing Intel® AVX and Intel SSE in Function Calls

Intel AVX to Intel SSE transitions can occur unexpectedly when calling functions or returning from functions. For example, if a function that uses 256-bit Intel AVX, calls another function, the callee might be using SSE code. Similarly, after a 256-bit Intel AVX function returns, the caller might be executing Intel SSE code.

Assembly/Compiler Coding Rule 57. (H impact, H generality) *Add VZEROUPPER instruction after 256-bit AVX instructions are executed and before any function call that might execute SSE code. Add VZEROUPPER at the end of any function that uses 256-bit AVX instructions.*

Example 15-4. Function Calls and Intel® AVX/Intel® SSE transitions

```
__attribute__((noinline)) void SSE_function()
{
    __asm addps xmm1, xmm2
    __asm xorps xmm3, xmm4
}

__attribute__((noinline)) void AVX_function_no_zeroupper()
{
    __asm vaddps ymm1, ymm2, ymm3
    __asm vxorps ymm4, ymm5, ymm6
}
__attribute__((noinline)) void AVX_function_with_zeroupper()
{
    __asm vaddps ymm1, ymm2, ymm3
    __asm vxorps ymm4, ymm5, ymm6
    //add vzeroupper when returning from an AVX function
    __asm vzeroupper
}
```

```
// Code encounter transition penalty

__asm vaddps ymm1, ymm2, ymm3
..

//penalty
SSE_function();
AVX_function_no_zeroupper();
//penalty
__asm addps xmm1, xmm2
```

```
// Code mitigated transition penalty

__asm vaddps ymm1, ymm2, ymm3
//add vzeroupper before
//calling SSE function from AVX code
__asm vzeroupper   //no penalty
SSE_function();
AVX_function_with_zeroupper();
//no penalty
__asm addps xmm1, xmm2
```

Table 15-2 summarizes a heuristic of the performance impact of using or not using VZEROUPPER to bridge transitions of inter-function calls that changes between AVX code implementation and SSE code.

Table 15-4. Effect of VZEROUPPER with Inter-Function Calls Between AVX and SSE Code

Inter-Function Call	Prior Microarchitectures	Skylake Microarchitecture
With VZEROUPPER	1X (baseline)	~1
No VZEROUPPER	< 0.1X	Fraction of baseline

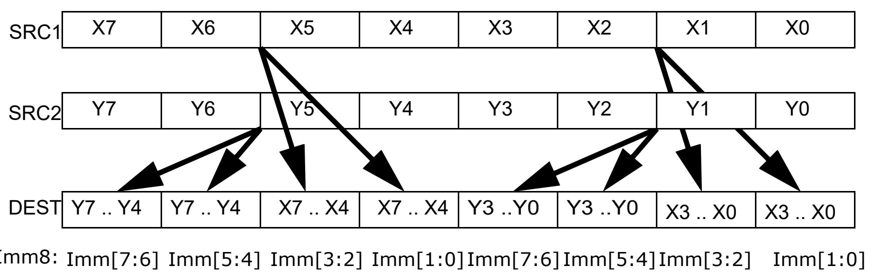
15.4 128-BIT LANE OPERATION AND AVX

256-bit operations in Intel AVX are generally performed in two halves of 128-bit lanes. Most of the 256-bit Intel AVX instructions are defined as in-lane: the destination elements in each lane are calculated using source elements only from the same lane. There are only a few cross-lane instructions, which are described below.

The majority of SSE computational instructions perform computation along vertical slots with each data elements. The 128-bit lanes does not affect porting 128-bit code into 256-bit AVX code. VADDPS is one example of this.

Many 128-bit SSE instruction moves data elements horizontally, e.g. SHUFPS uses an imm8 byte to control the horizontal movement of data elements.

Intel AVX promotes these horizontal 128-bit SIMD instruction in-lane into 256-bit operation by using the same control field within the low 128-bit land and the high 128-bit lane. For example, the 256-bit VSHUFPS instruction uses a control byte containing 4 control values to select the source location of each destination element in a 128-bit lane. This is shown below.



Control Values 00b: X0/Y0 (Low lane), X4/Y4 (high lane)
Control Values 01b: X1/Y1 (Low lane), X5/Y5 (high lane)
Control Values 10b: X2/Y2 (Low lane), X6/Y6 (high lane)
Control Values 11b: X3/Y3 (Low lane), X7/Y7 (high lane)

15.4.1 Programming With the Lane Concept

Using the lane concept, algorithms implemented with SSE instruction set can be easily converted to use 256-bit Intel AVX. An SSE algorithm that executes iterations 0 to n can be converted such that the calculation of iteration i is done in the low lane and the calculation of iteration i+k is done in the high lane. For consecutive iterations k equals one.

Some vectorized algorithms implemented with SSE instructions cannot use a simple conversion described above. For example, shuffles that move elements within 16 bytes cannot be naturally converted to shuffles with 32 byte since 32 byte shuffles can't cross lanes.

You can use the following instructions as building blocks for working with lanes:

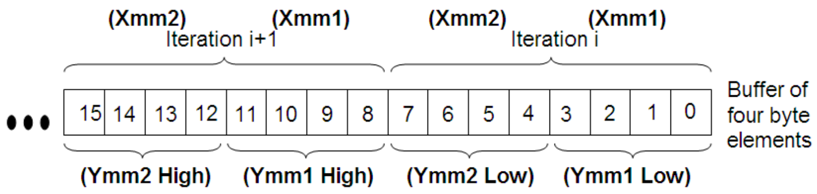
- VINSERTF128 - insert packed floating-point values.
- VEXTRACTF128 - extract packed floating-point values.
- VPERM2F128 - permute floating-point values.
- VBROADCAST - load with broadcast.

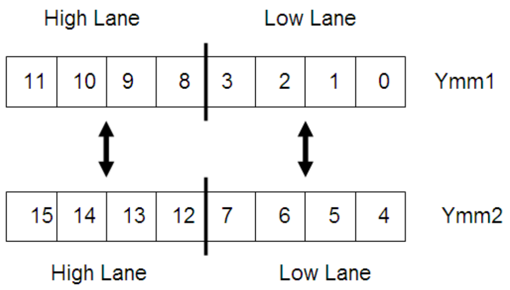
The sections below describe two techniques: the strided loads and the cross register overlap. These methods implement the in lane data arrangement described above and are useful in many algorithms that initially seem to require cross lane calculations.

15.4.2 Strided Load Technique

The strided load technique is a programming method that uses Intel AVX instructions and is useful for algorithms that involve unsupported cross-lane shuffles.

The method describes how to arrange data to avoid cross-lane shuffles. The main idea is to use 128-bit loads in a way that mimics the corresponding Intel SSE algorithm, and enables the 256 Intel AVX instructions to execute iterations i of the loop in the low lanes and the iteration and i+k in the high lanes. In the following example, k equals one.



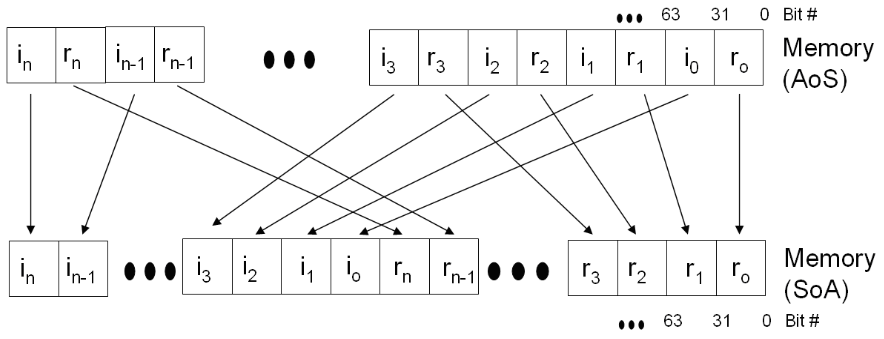


The values in the low lanes of Ymm1 and Ymm2 in the figure above correspond to iteration i in the SSE implementation. Similarly, the values in the high lanes of Ymm1 and Ymm2 correspond to iteration i+1.

The following example demonstrates the strided load method in a conversion of an Array of Structures (AoS) to a Structure of Arrays (SoA). In this example, the input buffer contains complex numbers in an AoS format. Each complex number is made of a real and an imaginary float values. The output buffer is arranged as SoA. All the real components of the complex numbers are located at the first half of the output buffer and all the imaginary components are located at the second half of the buffer. The following pseudo code and figure illustrate the conversion:

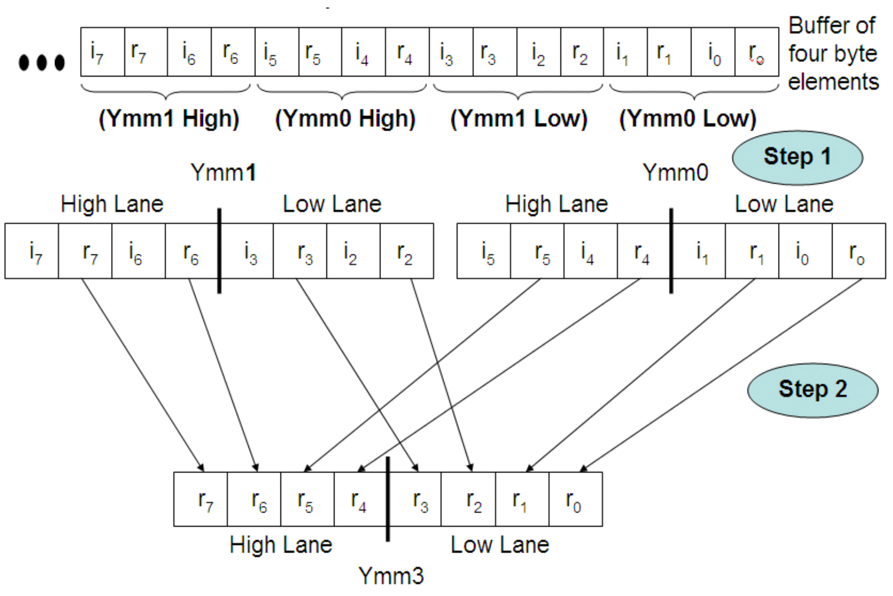
Example 15-5. AoS to SoA Conversion of Complex Numbers in C Code

```
for (i = 0; i < N; i++)
{
  Real[i] = Complex[i].Real;
  Imaginary[i] = Complex[i].Imaginary;
}
```



A simple extension of the Intel SSE algorithm from 16-byte to 32-byte operations would require cross-lane data transition, as shown in the following figure. However, this is not possible with Intel AVX architecture and a different technique is required.

The challenge of cross-lane shuffle can be overcome with Intel AVX for AoS to SoA conversion. Using VINSERTF128 to load 16 bytes to the appropriate lane in the YMM registers obviates the need for cross-lane shuffle. Once the data is organized properly in the YMM registers for step 1, 32-byte VSHUFPS can be used to move the data in lanes, as shown in step 2.



The following code compares the Intel SSE implementation of AoS to SoA with the 256-bit Intel AVX implementation and demonstrates the performance gained.

Example 15-6. Aos to SoA Conversion of Complex Numbers Using Intel® AVX

Intel® SSE Code	Intel® AVX Code

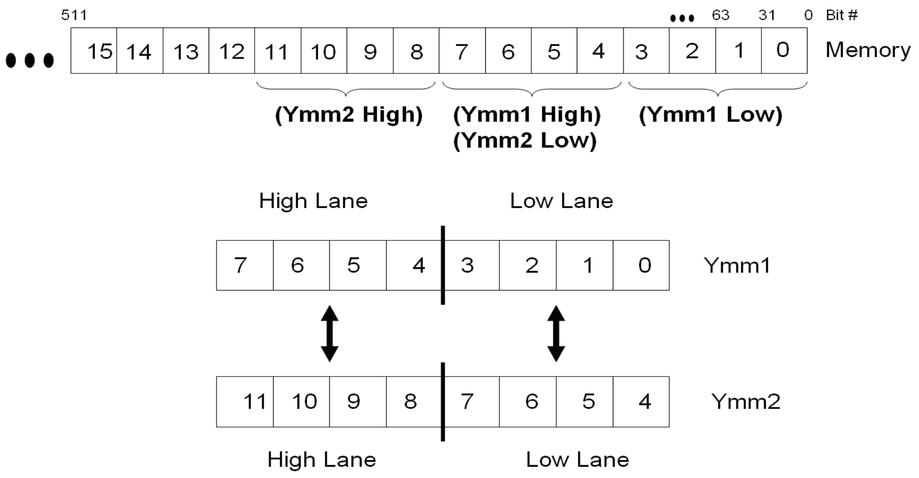
xor rbx, rbx
xor rdx, rdx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr1
mov rax, outPtr2
loop1:
movups xmm0, [rdi+rbx]
//i1 r1 i0 r0
movups xmm1, [rdi+rbx+16]
// i3 r3 i2 r2
movups xmm2, xmm0
 	
shufps xmm0, xmm1, 0xdd
//i3 i2 i1 i0
shufps xmm2, xmm1, 0x88
//r3 r2 r1 r0
	xor rbx, rbx
	xor rdx, rdx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr1
	mov rax, outPtr2
	loop1:
	vmovups xmm0, [rdi+rbx]
	//i1 r1 i0 r0
	vmovups xmm1, [rdi+rbx+16]
	// i3 r3 i2 r2
	vinsertf128 ymm0, ymm0, [rdi+rbx+32] , 1
	//i5 r5 i4 r4; i1 r1 i0 r0
	vinsertf128 ymm1, ymm1, [rdi+rbx+48] , 1
	//i7 r7 i6 r6; i3 r3 i2 r2
	vshufps ymm2, ymm0, ymm1, 0xdd
	//i7 i6 i5 i4; i3 i2 i1 i0
	vshufps ymm3, ymm0, ymm1, 0x88
	//r7 r6 r5 r4; r3 r2 r1 r0

movups [rax+rdx], xmm0 movups [rsi+rdx], xmm2 add rdx, 16 add rbx, 32 cmp rcx, rbx jnz loop1	vmovups [rax+rdx], ymm2 vmovups [rsi+rdx], ymm3 add rdx, 32 add rbx, 64 cmp rcx, rbx jnz loop1

15.4.3 The Register Overlap Technique

The register overlap technique is useful for algorithms that use shuffling. Similar to the strided load technique, the register overlap technique arranges data to avoid cross-lane shuffles.

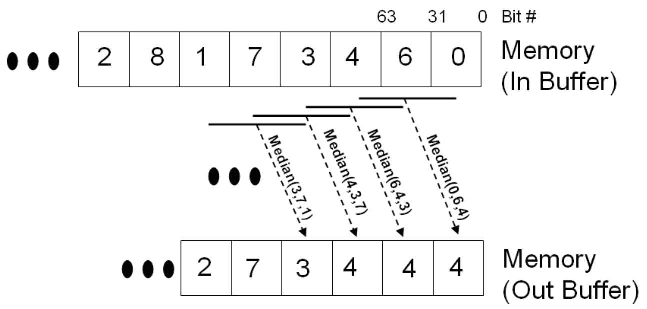
This technique is useful for algorithm that process continues data, which is partially shared by sequential iterations. The following figure illustrates the desired data layout. This is enabled by using overlapping 256-bit loads, or by using the VPERM2F128 instruction.



The Median3 code sample below demonstrates the register overlap technique. The median3 technique calculates the median of every three consecutive elements in a vector.

Y[i] = Median(X[i], X[i+1], X[i+2])

Where Y is the output vector and X is the input vector. The following figure illustrates the calculation done by the median algorithm.



Following are three implementations of the Median3 algorithm:

- Alternative 1 is the Intel SSE implementation.
- Alternatives 2 and 3 implement the register overlap technique in two ways.
  - — Alternative 2 loads the data from the input buffer into the YMM registers using overlapping 256-bit load operations.
  - — Alternative 3 loads the data from the input buffer into the YMM registers using a 256-bit load operation and VPERM2F128.
  - — Alternatives 2 and 3 gain performance by using wider vectors.

Example 15-7. Register Overlap Method for Median of 3 Numbers

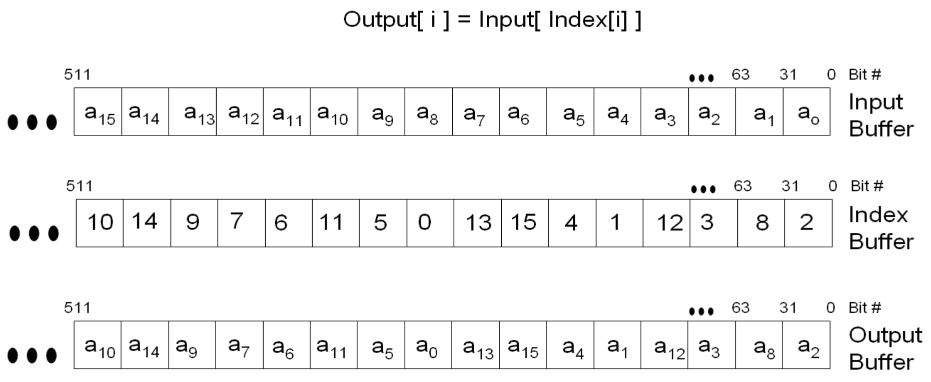
1: SSE Code	2: 256-bit AVX w/ Overlapping Loads	3: 256-bit AVX with VPERM2F128
xor ebx, ebx	xor ebx, ebx	xor ebx, ebx
mov rcx, len	mov rcx, len	mov rcx, len
mov rdi, inPtr	mov rdi, inPtr	mov rdi, inPtr
mov rsi, outPtr	mov rsi, outPtr	mov rsi, outPtr
movaps xmm0, [rdi]	vmovaps ymm0, [rdi]	vmovaps ymm0, [rdi]
loop_start:	loop_start:	loop_start:
movaps xmm4, [rdi+16]	vshufps ymm2, ymm0,	add rdi, 32
movaps xmm2, [rdi]	[rdi+16], 0x4E	vmovaps ymm6, [rdi]
movaps xmm1, [rdi]	vshufps ymm1, ymm0,	vperm2f128 ymm1, ymm0, ymm6, 0x21
movaps xmm3, [rdi]	ymm2, 0x99	vshufps ymm3, ymm0, ymm1, 0x4E
add rdi, 16	add rbx, 8	vshufps ymm2, ymm0, ymm3, 0x99
add rbx, 4	add rdi, 32	add rbx, 8
shufps xmm2, xmm4, 0x4e		vminps ymm5, ymm0, ymm2
shufps xmm1, xmm2, 0x99	vminps ymm4, ymm0, ymm1	vmaxps ymm0, ymm0, ymm2
minps xmm3, xmm1	vmaxps ymm0, ymm0, ymm1	vminps ymm4, ymm0, ymm3
maxps xmm0, xmm1	vminps ymm3, ymm0, ymm2	vmaxps ymm7, ymm4, ymm5
minps xmm0, xmm2	vmaxps ymm5, ymm3, ymm4	vmovaps ymm0, ymm6
maxps xmm0, xmm3	vmovaps [rsi], ymm5	vmovaps [rsi], ymm7
movaps [rsi], xmm0	add rsi, 32	add rsi, 32
movaps xmm0, xmm4	vmovaps ymm0, [rdi]	cmp rbx, rcx
add rsi, 16	cmp rbx, rcx	jl loop_start
cmp rbx, rcx	jl loop_start	
jl loop_start		

15.5 DATA GATHER AND SCATTER

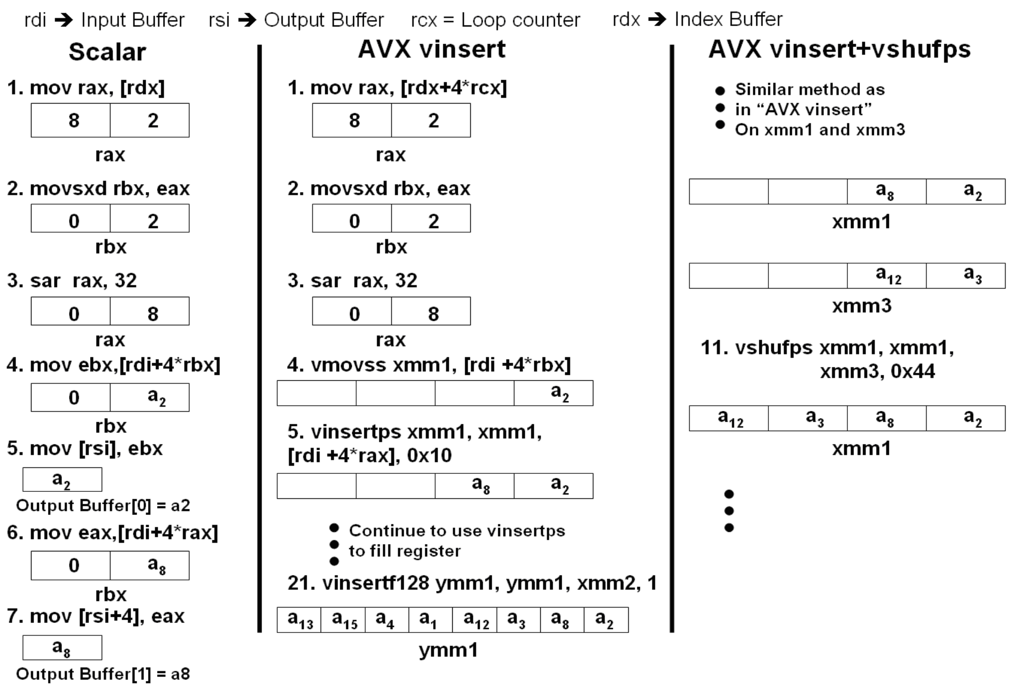
This section describes techniques for implementing data gather and scatter operations using Intel AVX instructions.

15.5.1 Data Gather

The gather operation reads elements from an input buffer based on indexes specified in an index buffer. The gathered elements are written in an output buffer. The following figure illustrates an example for a gather operation.


Output[i] = Input[Index[i]]
511
63 31 0 Bit #
a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
Input Buffer
10 14 9 7 6 11 5 0 13 15 4 1 12 3 8 2
Index Buffer
a10 a14 a9 a7 a6 a11 a5 a0 a13 a15 a4 a1 a12 a3 a8 a2
Output Buffer

Following are 3 implementations for the gather operation from an array of 4 byte elements. Alternative 1 is a scalar implementation using general purpose registers. Alternative 2 and 3 use Intel AVX instructions. The figure below shows code snippets from Example 15-8 assuming that it runs the first iteration on data from the previous figure.



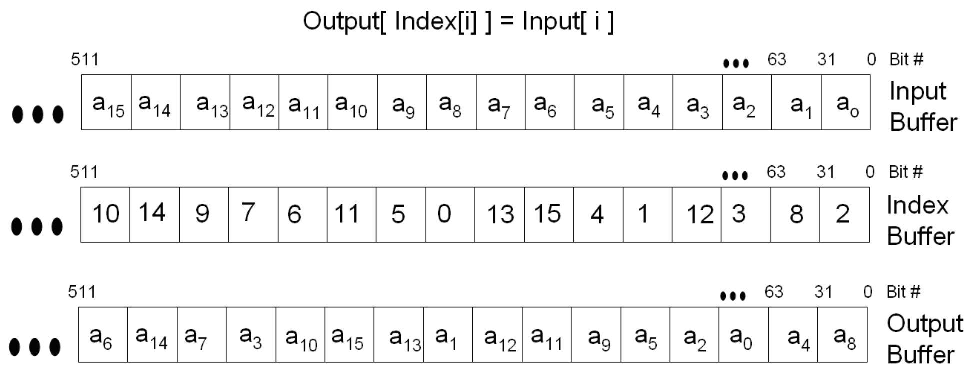
Performance of the Intel AVX examples is similar to the performance of a corresponding Intel SSE implementation. The table below shows the three gather implementations.

Example 15-8. Data Gather - Intel® AVX versus Scalar Code

1: Scalar Code	2: Intel® AVX w/ VINSERTPS	3: VINSERTPS+VSHUFPS
mov rdi, InBuf	mov rdi, InBuf	mov rdi, InBuf
mov rsi, OutBuf	mov rsi, OutBuf	mov rsi, OutBuf
mov rdx, Index	mov rdx, Index	mov rdx, Index
xor rcx, rcx	xor rcx, rcx	xor rcx, rcx
loop1:	loop1:	loop1:
mov rax, [rdx]	mov rax, [rdx + 4*rcx]	mov rax, [rdx + 4*rcx]
movsxd rbx, eax	movsxd rbx, eax	movsxd rbx, eax
sar rax, 32	sar rax, 32	sar rax, 32
mov ebx, [rdi + 4*rbx]	vmovss xmm1, [rdi + 4*rbx]	vmovss xmm1, [rdi + 4*rbx]
mov [rsi], ebx	vinsertps xmm1, xmm1,	vinsertps xmm1, xmm1,
mov eax, [rdi + 4*rax]	[rdi + 4*rax], 0x10	[rdi + 4*rax], 0x10
mov [rsi + 4], eax	mov rax, [rdx + 8 + 4*rcx]	mov rax, [rdx + 8 + 4*rcx]
	movsxd rbx, eax	movsxd rbx, eax
mov rax, [rdx + 8]	sar rax, 32	sar rax, 32
movsxd rbx, eax	vinsertps xmm1, xmm1,	vmovss xmm3, [rdi + 4*rbx]
sar rax, 32	[rdi + 4*rbx], 0x20	vinsertps xmm3, xmm3,
mov ebx, [rdi + 4*rbx]		[rdi + 4*rax], 0x10
mov [rsi + 8], ebx	vinsertps xmm1, xmm1,	
mov eax, [rdi + 4*rax]	[rdi + 4*rax], 0x30	vshufps xmm1, xmm1,
mov [rsi + 12], eax		xmm3, 0x44
mov rax, [rdx + 16]	mov rax, [rdx + 16 + 4*rcx]	mov rax, [rdx + 16 + 4*rcx]
movsxd rbx, eax	movsxd rbx, eax	movsxd rbx, eax
sar rax, 32	sar rax, 32	sar rax, 32
mov ebx, [rdi + 4*rbx]	vmovss xmm2, [rdi + 4*rbx]	vmovss xmm2, [rdi + 4*rbx]
mov [rsi + 16], ebx	vinsertps xmm2, xmm2,	vinsertps xmm2, xmm2,
mov eax, [rdi + 4*rax]	[rdi + 4*rax], 0x10	[rdi + 4*rax], 0x10
mov [rsi + 20], eax		
	mov rax, [rdx + 24 + 4*rcx]	mov rax, [rdx + 24 + 4*rcx]
mov rax, [rdx + 24]	movsxd rbx, eax	movsxd rbx, eax
movsxd rbx, eax	sar rax, 32	sar rax, 32
sar rax, 32	vinsertps xmm2, xmm2,	vmovss xmm4, [rdi + 4*rbx]
mov ebx, [rdi + 4*rbx]	[rdi + 4*rbx], 0x20	vinsertps xmm4, xmm4,
mov [rsi + 24], ebx		[rdi + 4*rax], 0x10
mov eax, [rdi + 4*rax]	vinsertps xmm2, xmm2,	
mov [rsi + 28], eax	[rdi + 4*rax], 0x30	vshufps xmm2, xmm2,
		xmm4, 0x44
add rsi, 32	vinsertf128 ymm1, ymm1,	
add rdx, 32	xmm2, 1	vinsertf128 ymm1, ymm1,
add rcx, 8		xmm2, 1
cmp rcx, len	vmovaps [rsi + 4*rcx], ymm1	
jl loop1	add rcx, 8	vmovaps [rsi + 4*rcx], ymm1
	cmp rcx, len	add rcx, 8
	jl loop1	cmp rcx, len
		jl loop1

15.5.2 Data Scatter

The scatter operation reads elements from an input buffer sequentially. It then writes them to an output buffer based on indexes specified in an index buffer. The following figure illustrates an example for a scatter operation.


Output[Index[i]] = Input[i]
511
63
31
0
Bit #
a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
Input Buffer
10 14 9 7 6 11 5 0 13 15 4 1 12 3 8 2
Index Buffer
a_6 a_{14} a_7 a_3 a_{10} a_{15} a_{13} a_1 a_{12} a_{11} a_9 a_5 a_2 a_0 a_4 a_8
Output Buffer

The following table includes a scalar implementation and an Intel AVX implementation of a scatter sequence. The Intel AVX examples consist mainly of 128-bit Intel AVX instructions. Performance of the Intel AVX examples is similar to the performance of corresponding Intel SSE implementation.

Example 15-9. Scatter Operation Using Intel® AVX

Scalar Code	AVX Code
mov rdi, InBuf mov rsi, OutBuf mov rdx, Index xor rcx, rcx loop1: movsxd rax, [rdx] mov ebx, [rdi] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 4] mov ebx, [rdi + 4] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 8] mov ebx, [rdi + 8] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 12] mov ebx, [rdi + 12] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 16] mov ebx, [rdi + 16] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 20] mov ebx, [rdi + 20] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 24] mov ebx, [rdi + 24]	mov rdi, InBuf mov rsi, OutBuf mov rdx, Index xor rcx, rcx loop1: vmovaps ymm0, [rdi + 4*rcx] movsxd rax, [rdx + 4*rcx] movsxd rbx, [rdx + 4*rcx + 4] vmovss [rsi + 4*rax], xmm0 movsxd rax, [rdx + 4*rcx + 8] vpalignr xmm1, xmm0, xmm0, 4 vmovss [rsi + 4*rbx], xmm1 movsxd rbx, [rdx + 4*rcx + 12] vpalignr xmm2, xmm0, xmm0, 8 vmovss [rsi + 4*rax], xmm2 movsxd rax, [rdx + 4*rcx + 16] vpalignr xmm3, xmm0, xmm0, 12 vmovss [rsi + 4*rbx], xmm3 movsxd rbx, [rdx + 4*rcx + 20] vextractf128 xmm0, ymm0, 1 vmovss [rsi + 4*rax], xmm0 movsxd rax, [rdx + 4*rcx + 24] vpalignr xmm1, xmm0, xmm0, 4 vmovss [rsi + 4*rbx], xmm1 movsxd rbx, [rdx + 4*rcx + 28]

Example 15-9. Scatter Operation Using Intel® AVX (Contd.)

Scalar Code	AVX Code
mov [rsi + 4*rax], ebx movsxd rax, [rdx + 28] mov ebx, [rdi + 28] mov [rsi + 4*rax], ebx add rdi, 32 add rdx, 32 add rcx, 8 cmp rcx, len jl loop1	vpalignr xmm2, xmm0, xmm0, 8 vmovss [rsi + 4*rax], xmm2 vpalignr xmm3, xmm0, xmm0, 12 vmovss [rsi + 4*rbx], xmm3 add rcx, 8 cmp rcx, len jl loop1

15.6 DATA ALIGNMENT FOR INTEL® AVX

This section explains the benefit of aligning data that is used by Intel AVX instructions and proposes some methods to improve performance when such alignment is not possible. Most examples in this section are variations of the SAXPY kernel. SAXPY is the Scalar Alpha * X + Y algorithm.

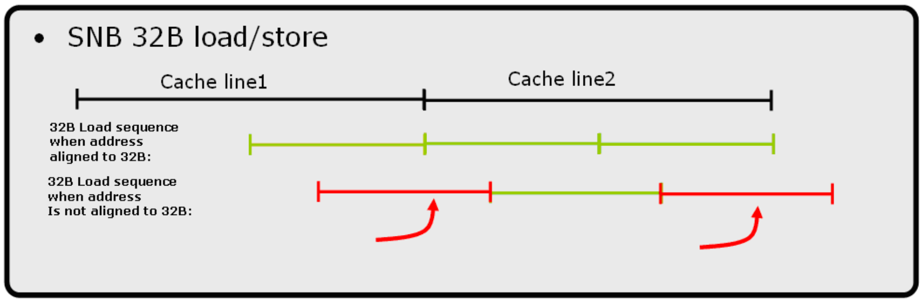
The C code below is a C implementation of SAXPY.

```
for (int i = 0; i < n; i++)
{ c[i] = alpha * a[i] + b[i]; }
```

15.6.1 Align Data to 32 Bytes

Aligning data to vector length is recommended. When using 16-byte SIMD instructions, loaded data should be aligned to 16 bytes. Similarly, for best results when using Intel AVX instructions with 32-byte registers align the data to 32-bytes.

When using Intel AVX with unaligned 32-byte vectors, every second load is a cache-line split, since the cache-line is 64 bytes. This doubles the cache line split rate compared to Intel SSE code that uses 16-byte vectors. Even though split line access penalties have been reduced significantly since Nehalem microarchitecture, a high cache-line split rate in memory-intensive code may cause performance degradation.


• SNB 32B load/store
Cache line1
Cache line2
32B Load sequence when address aligned to 32B:
32B Load sequence when address Is not aligned to 32B:

Example 15-10. SAXPY using Intel® AVX

```
    mov         rax, src1
    mov         rbx, src2
    mov         rcx, dst
    mov         rdx, len
    xor         rdi, rdi
    vbroadcastss        ymm0, alpha


start_loop:
    vmovups     ymm1, [rax + rdi]
    vmulps      ymm1, ymm1, ymm0
    vmovups     ymm2, [rbx + rdi]
    vaddps      ymm1, ymm1, ymm2
    vmovups     [rcx + rdi], ymm1
    vmovups     ymm1, [ rax + rdi + 32]
    vmulps      ymm1, ymm1, ymm0
    vmovups     ymm2, [rbx + rdi + 32]
    vaddps      ymm1, ymm1, ymm2
    vmovups     [rcx + rdi + 32], ymm1

    add         rdi, 64
    cmp         rdi, rdx
    jl          start_loop
```

SAXPY is a memory intensive kernel that emphasizes the importance of data alignment. Optimal performance requires both data source address to be 32-byte aligned and destination address also 32-byte aligned. If only one of the three address is not aligned to 32-byte boundary, the performance may be halved. If all three addresses are mis-aligned relative to 32 byte, the performance degrades further. In some cases, unaligned accesses may result in lower performance for Intel AVX code compared to Intel SSE code. Other Intel AVX kernels typically have more computation which can reduce the effect of the data alignment penalty.

Assembly/Compiler Coding Rule 58. (H impact, M generality) *Align data to 32-byte boundary when possible. Prefer store alignment over load alignment.*

You can use dynamic data alignment using the _mm_malloc intrinsic instruction with the Intel® Compiler, or _aligned_malloc of the Microsoft* Compiler. For example:

//dynamically allocating 32byte aligned buffer with 2048 float elements.

InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 32);

You can use static data alignment using __declspec(align(32)). For example:

//Statically allocating 32byte aligned buffer with 2048 float elements.

__declspec(align(32)) float InputBuffer[2048];

15.6.2 Consider 16-Byte Memory Access when Memory is Unaligned

For best results use Intel AVX 32-byte loads and align data to 32-bytes. However, there are cases where you cannot align the data, or data alignment is unknown. This can happen when you are writing a library function and the input data alignment is unknown. In these cases, using 16-byte memory accesses may be the best alternative. The following method uses 16-byte loads while still benefiting from the 32-byte YMM registers.

NOTE

Beginning with Skylake microarchitecture, this optimization is not necessary. The only case where 16-byte loads may be more efficient is when the data is 16-byte aligned but not 32-byte aligned. In this case 16-byte loads might be preferable as no cache line split memory accesses are issued.

Consider replacing unaligned 32-byte memory accesses using a combination of VMOVUPS, VINSERTF128, and VEXTRACTF128 instructions.

Example 15-11. Using 16-Byte Memory Operations for Unaligned 32-Byte Memory Operation

```
Convert 32-byte loads as follows:
    vmovups      ymm0, mem   ->  vmovups xmm0, mem
                                 vinsertf128   ymm0, ymm0, mem+16, 1
Convert 32-byte stores as follows:
    vmovups mem, ymm0        ->  vmovups mem, xmm0
                                 vextractf128 mem+16, ymm0, 1

The following intrinsics are available to handle unaligned 32-byte memory operating using 16-byte memory accesses:
    _mm256_loadu2_m128 ( float const * addr_hi, float const * addr_lo);
    _mm256_loadu2_m128d ( double const * addr_hi, double const * addr_lo);
    _mm256_loadu2_m128 i( __m128i const * addr_hi, __m128i const * addr_lo);
    _mm256_storeu2_m128 ( float * addr_hi, float * addr_lo, __m256 a);
    _mm256_storeu2_m128d ( double * addr_hi, double * addr_lo, __m256d a);
    _mm256_storeu2_m128 i( __m128i * addr_hi, __m128i * addr_lo, __m256i a);
```

Example 15-12 shows two implementations for SAXPY with unaligned addresses. Alternative 1 uses 32-byte loads and alternative 2 uses 16-byte loads. These code samples are executed with two source buffers, src1, src2, at 4 byte offset from 32-byte alignment, and a destination buffer, DST, that is 32-byte aligned. Using two 16-byte memory operations in lieu of 32-byte memory access performs faster.[1]

Example 15-12. SAXPY Implementations for Unaligned Data Addresses

AVX with 32-byte memory operation	AVX using two 16-byte memory operations
mov rax, src1 mov rbx, src2 mov rcx, dst mov rdx, len xor rdi, rdi vbroadcastss ymm0, alpha start_loop: vmovups ymm1, [rax + rdi] vmulps ymm1, ymm1, ymm0 vmovups ymm2, [rbx + rdi] vaddps ymm1, ymm1, ymm2 vmovups [rcx + rdi], ymm1	mov rax, src1 mov rbx, src2 mov rcx, dst mov rdx, len xor rdi, rdi vbroadcastss ymm0, alpha start_loop: vmovups xmm2, [rax+rdi] vinsertf128 ymm2, ymm2, [rax+rdi+16], 1 vmulps ymm1, ymm0, ymm2 vmovups xmm2, [rbx + rdi] vinsertf128 ymm2, ymm2, [rbx+rdi+16], 1 vaddps ymm1, ymm1, ymm2

1. Beginning with Haswell microarchitecture and onward, it is better to read the entire register: 32-byte register or 64-byte register (with the availability of Intel® AVX-512).

Example 15-12. SAXPY Implementations for Unaligned Data Addresses (Contd.)

AVX with 32-byte memory operation	AVX using two 16-byte memory operations
 vmovups ymm1, [rax+rdi+32] vmulps ymm1, ymm1, ymm0 vmovups ymm2, [rbx+rdi+32] vaddps ymm1, ymm1, ymm2 vmovups [rcx+rdi+32], ymm1 add rdi, 64 cmp rdi, rdx jl start_loop	vmovups [rcx+rdi], ymm1 vmovups xmm2, [rax+rdi+32] vinsertf128 ymm2, ymm2, [rax+rdi+48], 1 vmulps ymm1, ymm0, ymm2 vmovups xmm2, [rbx+rdi+32] vinsertf128 ymm2, ymm2, [rbx+rdi+48], 1 vaddps ymm1, ymm1, ymm2 vmovups [rcx+rdi+32], ymm1 add rdi, 64 cmp rdi, rdx jl start_loop

Assembly/Compiler Coding Rule 59. (M impact, H generality) *Align data to 32-byte boundary when possible. If it is not possible to align both loads and stores, then prefer store alignment over load alignment.*

15.6.3 Prefer Aligned Stores Over Aligned Loads

There are cases where it is possible to align only a subset of the processed data buffers. In these cases, aligning data buffers used for store operations usually yields better performance than aligning data buffers used for load operations.

Unaligned stores are likely to cause greater performance degradation than unaligned loads, since there is a very high penalty on stores to a split cache-line that crosses pages. This penalty is estimated at 150 cycles. Stores that cross a page boundary are executed at retirement. In Example 15-12, unaligned store address can affect SAXPY performance for 3 unaligned addresses to about one quarter of the aligned case.

15.7 L1D CACHE LINE REPLACEMENTS

NOTE

Beginning with Haswell microarchitecture, cache line replacement is no longer a concern .

When a load misses the L1D Cache, a cache line with the requested data is brought from a higher memory hierarchy level. In memory intensive code where the L1D Cache is always active, replacing a cache line in the L1D Cache may delay other loads. In Sandy Bridge and Ivy Bridge microarchitectures, the penalty for 32-Byte loads may be higher than the penalty for 16-Byte loads. Therefore, memory intensive Intel AVX code with 32-Byte loads and with data set larger than the L1D Cache may be slower than similar code with 16-Byte loads.

When Example 15-12 is run with a data set that resides in the L2 Cache, the 16-byte memory access implementation is slightly faster than the 32-byte memory operation.

Be aware that the relative merit of 16-byte memory accesses versus 32-byte memory access is implementation specific across generations of microarchitectures.

In Haswell microarchitecture, the L1D Cache can support two 32-byte fetch each cycle.

15.8 4K ALIASING

4-KByte memory aliasing occurs when the code stores to one memory location and shortly after that it loads from a different memory location with a 4-KByte offset between them. For example, a load to linear address 0x400020 follows a store to linear address 0x401020.

The load and store have the same value for bits 5 - 11 of their addresses and the accessed byte offsets should have partial or complete overlap.

4K aliasing may have a five-cycle penalty on the load latency. This penalty may be significant when 4K aliasing happens repeatedly and the loads are on the critical path. If the load spans two cache lines it might be delayed until the conflicting store is committed to the cache. Therefore 4K aliasing that happens on repeated unaligned Intel AVX loads incurs a higher performance penalty.

To detect 4K aliasing, use the LD_BLOCKS_PARTIAL.ADDRESS_ALIAS event that counts the number of times Intel AVX loads were blocked due to 4K aliasing.

To resolve 4K aliasing, try the following methods in the following order:

- Align data to 32 Bytes.
- Change offsets between input and output buffers if possible.
- Sandy Bridge and Ivy Bridge microarchitectures may benefit from using 16-Byte memory accesses on memory which is not 32-Byte aligned.

15.9 CONDITIONAL SIMD PACKED LOADS AND STORES

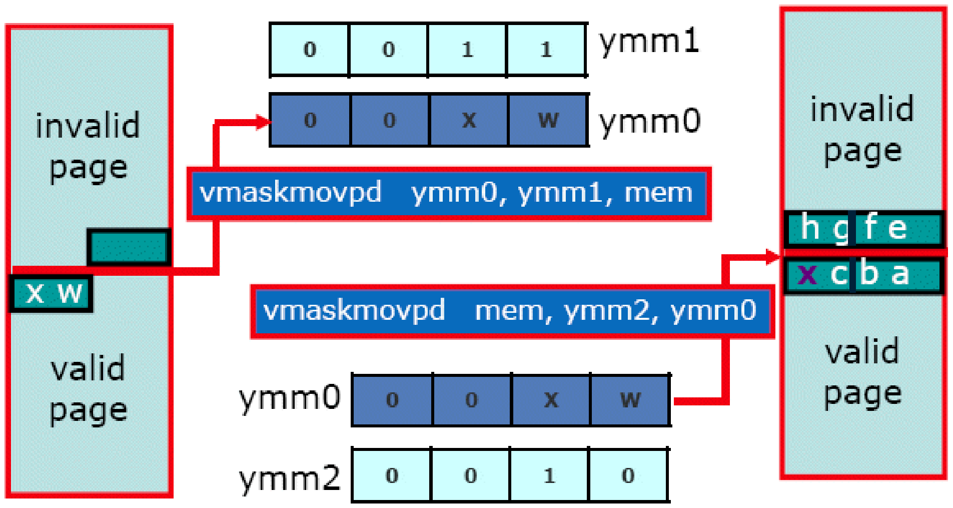
The VMASKMOV instruction conditionally moves packed data elements to/from memory, depending on the mask bits associated with each data element. The mask bit for each data element is the most significant bit of the corresponding element in the mask register.

When performing a mask load, the returned value is 0 for elements which have a corresponding mask value of 0. The mask store instruction writes to memory only the elements with a corresponding mask value of 1, while preserving memory values for elements with a corresponding mask value of 0. Faults can occur only for memory accesses that are required by the mask. Faults do not occur due to referencing any memory location if the corresponding mask bit value for that memory location is zero. For example, no faults are detected if the mask bits are all zero.

The following figure shows an example for a mask load and a mask store which does not cause a fault. In this example, the mask register for the load operation is ymm1 and the mask register for the store operation is ymm2.

When using masked load or store consider the following:

- On processors based on microarchitectures prior to Skylake, the address of a VMASKMOV store is considered as resolved only after the mask is known. Loads that follow a masked store may be blocked, depending on the memory disambiguation prediction, until the mask value is known.
- If the mask is not all 1 or all 0, loads that depend on the masked store have to wait until the store data is written to the cache. If the mask is all 1 the data can be forwarded from the masked store to the dependent loads. If the mask is all 0 the loads do not depend on the masked store.



- Masked loads including an illegal address range do not result in an exception if the range is under a zero mask value. However, the processor may take a multi-hundred-cycle “assist” to determine that no part of the illegal range have a one mask value. This assist may occur even when the mask is “zero” and it seems obvious to the programmer that the load should not be executed.

When using VMASKMOV, consider the following:

- Use VMASKMOV only in cases where VMOVUPS cannot be used.
- Use VMASKMOV on 32Byte aligned addresses if possible.
- If possible use valid address range for masked loads, even if the illegal part is masked with zeros.
- Determine the mask as early as possible.
- Avoid store-forwarding issues by performing loads prior to a VMASKMOV store if possible.
- Be aware of mask values that would cause the VMASKMOV instruction to require assist (if an assist is required, the latency of VMASKMOV to load data will increase dramatically):
  — Load data using VMASKMOV with a mask value selecting 0 elements from an illegal address will require an assist.
  — Load data using VMASKMOV with a mask value selecting 0 elements from a legal address expressed in some addressing form (e.g. [base+index], disp[base+index])will require an assist.

With processors based on the Skylake microarchitecture, the performance characteristics of VMASKMOV instructions have the following notable items:

- Loads that follow a masked store is not longer blocked until the mask value is known.
- Store data using VMASKMOV with a mask value permitting 0 elements to be written to an illegal address will require an assist.

15.9.1 Conditional Loops

VMASKMOV enables vectorization of loops that contain conditional code. There are two main benefits in using VMASKMOV over the scalar implementation in these cases:

- VMASKMOV code is vectorized.
- Branch mispredictions are eliminated.

Below is a conditional loop C code:

Example 15-13. Loop with Conditional Expression

```
for(int i = 0; i < miBufferWidth; i++)
{          if(A[i]>0)
           {                 B[i] = (E[i]*C[i]);
           }
           else
           {                 B[i] = (E[i]*D[i]);
           }
}
```

Example 15-14. Handling Loop Conditional with VMASKMOV

Scalar	AVX using VMASKMOV

float* pA = A;
float* pB = B;
float* pC = C;
float* pD = D;
float* pE = E;
uint64 len = (uint64) (miBuffer-Width)*sizeof(float);
__asm
{
    mov rax, pA
    mov rbx, pB
    mov rcx, pC
    mov rdx, pD
    mov rsi, pE
    mov r8, len
 	
//xmm8 all zeros
    vxorps xmm8, xmm8, xmm8
 	
    xor r9, r9
loop1:
    vmovss xmm1, [rax+r9]
    vcomiss xmm1, xmm8
    jbe a_le
a_gt:
    vmovss xmm4, [rcx+r9]
    jmp mul
a_le:
    vmovss xmm4, [rdx+r9]
mul:
    vmulss xmm4, xmm4, [rsi+r9]
    vmovss [rbx+r9], xmm4
    add r9, 4
    cmp r9, r8
   jl loop1
}
	float* pA = A;
	float* pB = B;
	float* pC = C;
	float* pD = D;
	float* pE = E;
	uint64 len = (uint64) (miBufferWidth)*sizeof(float);
	__asm
	{
	    mov rax, pA
	    mov rbx, pB
	    mov rcx, pC
	    mov rdx, pD
	    mov rsi, pE
	    mov r8, len
	
	//ymm8 all zeros
	  vxorps ymm8, ymm8, ymm8
	    //ymm9 all ones
	  vcmpps ymm9, ymm8, ymm8, 0
	  xor r9, r9
	loop1:
	  vmovups ymm1, [rax+r9]
	  vcmpps ymm2, ymm8, ymm1, 1
	  vmaskmovps ymm4, ymm2, [rcx+r9]
	  vxorps ymm2, ymm2, ymm9
	  vmaskmovps ymm5, ymm2, [rdx+r9]
	  vorps ymm4, ymm4, ymm5
	  vmulps ymm4, ymm4, [rsi+r9]
	  vmovups [rbx+r9], ymm4
	  add r9, 32
	  cmp r9, r8
	  jl loop1
	}

The performance of the left side of Example 15-14 is sensitive to branch mis-predictions and can be an order of magnitude slower than the VMASKMOV example which has no data-dependent branches.

15.10 MIXING INTEGER AND FLOATING-POINT CODE

Integer SIMD functionalities in Intel AVX instructions are limited to 128-bit. There are some algorithm that uses mixed integer SIMD and floating-point SIMD instructions. Therefore, porting such legacy 128-bit code into 256-bit AVX code requires special attention.

For example, PALINGR (Packed Align Right) is an integer SIMD instruction that is useful arranging data elements for integer and floating-point code. But VPALINGR instruction does not have a corresponding 256-bit instruction in AVX.

There are three approaches to consider when porting legacy code consisting of mostly floating-point with some integer operations into 256-bit AVX code:

- Locate a 256-bit AVX alternative to replace the critical128-bit Integer SIMD instructions if such an AVX instructions exist. This is more likely to be true with integer SIMD instruction that rearranges data elements.
- Mix 128-bit AVX and 256-bit AVX instructions.
- Use Intel AVX2 instructions.

The performance gain from these two approaches may vary. Where possible, use method (1), since this method utilizes the full 256-bit vector width.

In case the code is mostly integer, convert the code from 128-bit SSE to 128 bit AVX instructions and gain from the Non destructive Source (NDS) feature.

Example 15-15. Three-Tap Filter in C Code

```
for(int i = 0; i < len -2; i++)
{
    pOut[i] = A[i]*coeff[0]+A[i+1]*coeff[1]+A[i+2]*coeff[2];
}
```

Example 15-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD

```
    xor   ebx, ebx
    mov     rcx, len
    mov     rdi, inPtr
    mov     rsi, outPtr
    mov     r15, coeffs
    movss      xmm2, [r15]          // load coeff 0
    shufps     xmm2, xmm2, 0        // broadcast coeff 0
    movss      xmm1, [r15+4]        // load coeff 1
    shufps     xmm1, xmm1, 0        // broadcast coeff 1
    movss      xmm0, [r15+8]        // coeff 2
    shufps     xmm0, xmm0, 0        // broadcast coeff 2
    movaps     xmm5, [rdi]          // xmm5={A[n+3],A[n+2],A[n+1],A[n]}
```

Example 15-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD (Contd.)

```
loop_start:
    movaps  xmm6, [rdi+16]          // xmm6={A[n+7],A[n+6],A[n+5],A[n+4]}
    movaps  xmm7, xmm6
    movaps  xmm8, xmm6
    add    rdi, 16        // inPtr+=16
    add    rbx, 4         // loop counter
    palignr    xmm7, xmm5, 4        // xmm7={A[n+4],A[n+3],A[n+2],A[n+1]}
    palignr    xmm8, xmm5, 8        // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
    mulps  xmm5, xmm2         //xmm5={C0*A[n+3],C0*A[n+2],C0*A[n+1], C0*A[n]}

    mulps  xmm7, xmm1         // xmm7={C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
    mulps  xmm8, xmm0         // xmm8={C2*A[n+5],C2*A[n+4] C2*A[n+3],C2*A[n+2]}
    addps  xmm7 ,xmm5
    addps  xmm7, xmm8
    movaps  [rsi], xmm7
    movaps  xmm5, xmm6
    add    rsi, 16              // outPtr+=16
    cmp    rbx, rcx
    jl     loop_start
```

Example 15-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS

```
    xor   ebx, ebx
    mov     rcx, len
    mov     rdi, inPtr
    mov     rsi, outPtr
    mov     r15, coeffs
    vbroadcastss       ymm2, [r15]       // load and broadcast coeff 0
    vbroadcastss       ymm1, [r15+4]     // load and broadcast coeff 1
    vbroadcastss       ymm0, [r15+8]     // load and broadcast coeff 2
```

Example 15-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS (Contd.)

```
loop_start:
    vmovaps     ymm5, [rdi]          // Ymm5={A[n+7],A[n+6],A[n+5],A[n+4];
                                     // A[n+3],A[n+2],A[n+1] , A[n]}
    vshufps  ymm6, ymm5, [rdi+16], 0x4e            // ymm6={A[n+9],A[n+8],A[n+7],A[n+6];
                                                   // A[n+5],A[n+4],A[n+3],A[n+2]}
    vshufps  ymm7, ymm5, ymm6, 0x99                // ymm7={A[n+8],A[n+7],A[n+6],A[n+5];
                                                   // A[n+4],A[n+3],A[n+2],A[n+1]}
    vmulps   ymm3, ymm5, ymm2          // ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5],C0*A[n+4];
                                       // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
    vmulps   ymm9, ymm7, ymm1          // ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                                       // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
    vmulps   ymm4, ymm6, ymm0          // ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                                       // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
    vaddps   ymm8, ymm3, ymm4
    vaddps   ymm10, ymm8, ymm9
    vmovaps     [rsi], ymm10
    add         rdi, 32              // inPtr+=32
    add         rbx, 8               // loop counter
    add         rsi, 32              // outPtr+=32
    cmp         rbx, rcx
    jl      loop_start
```

Example 15-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code

```
    xor   ebx, ebx
    mov     rcx, len
    mov     rdi, inPtr
    mov     rsi, outPtr
    mov     r15, coeffs
    vbroadcastss        ymm2, [r15]              // load and broadcast coeff 0
    vbroadcastss        ymm1, [r15+4]            // load and broadcast coeff 1
    vbroadcastss        ymm0, [r15+8]            // load and broadcast coeff 2
    vmovaps        xmm3, [rdi]             // xmm3={A[n+3],A[n+2],A[n+1],A[n]}
loop_start:
    vmovaps        xmm4, [rdi+16]          // xmm4={A[n+7],A[n+6],A[n+5],A[n+4]}
    vmovaps        xmm5, [rdi+32]          // xmm5={A[n+11], A[n+10],A[n+9],A[n+8]}
    vinsertf128    ymm3, ymm3, xmm4, 1            // ymm3={A[n+7],A[n+6],A[n+5],A[n+4];
                                                  // A[n+3], A[n+2],A[n+1],A[n]}
    vpalignr       xmm6, xmm4, xmm3, 4            // xmm6={A[n+4],A[n+3],A[n+2],A[n+1]}
    vpalignr       xmm7, xmm5, xmm4, 4            // xmm7={A[n+8],A[n+7],A[n+6],A[n+5]}
    vinsertf128         ymm6, ymm6, xmm7, 1            // ymm6={A[n+8],A[n+7],A[n+6],A[n+5];
                                                        // A[n+4],A[n+3],A[n+2],A[n+1]}
    vpalignr       xmm8, xmm4, xmm3, 8             // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
    vpalignr       xmm9, xmm5, xmm4, 8            // xmm9={A[n+9],A[n+8],A[n+7],A[n+6]}
    vinsertf128         ymm8, ymm8, xmm9, 1            // ymm8={A[n+9],A[n+8],A[n+7],A[n+6];
                                                        // A[n+5],A[n+4],A[n+3],A[n+2]}
    vmulps         ymm3, ymm3, ymm2          // ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5], C0*A[n+4];
                                             // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
    vmulps         ymm6, ymm6, ymm1          // ymm6={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                                             // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
```

Example 15-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code (Contd.)

```
    vmulps      ymm8, ymm8, ymm0       // ymm8={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                                       // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
    vaddps      ymm3, ymm3, ymm6
    vaddps      ymm3, ymm3, ymm8
    vmovaps     [rsi], ymm3
    vmovaps     xmm3, xmm5
    add    rdi, 32          // inPtr+=32
    add    rbx, 8           // loop counter
    add    rsi, 32          // outPtr+=32
    cmp    rbx, rcx
    jl     loop_start
```

Example 15-17 uses 256-bit VSHUFPS to replace the PALIGNR in 128-bit mixed SSE code. This speeds up almost 70% over the 128-bit mixed SSE code of Example 15-16 and slightly ahead of Example 15-18.

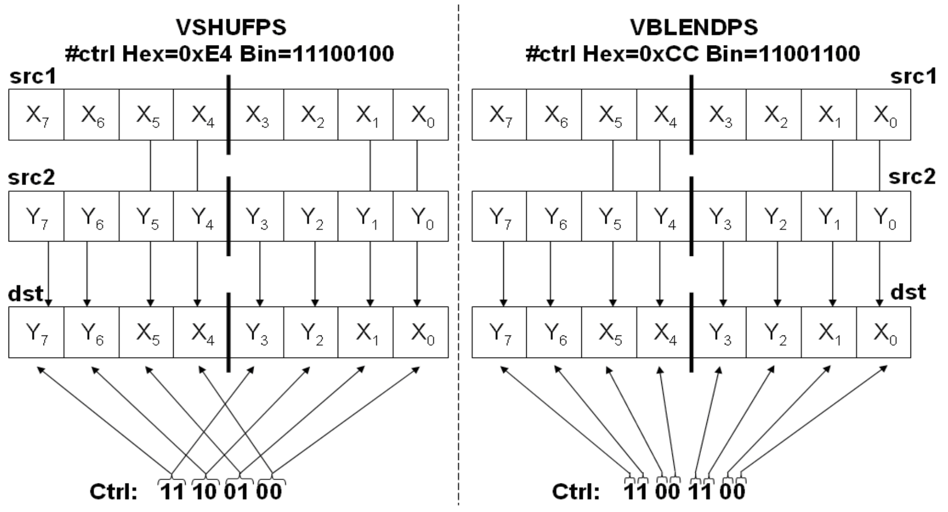
For code that includes integer instructions and is written with 256-bit Intel AVX instructions, replace the integer instruction with floating-point instructions that have similar functionality and performance. If there is no similar floating-point instruction, consider using a 128-bit Intel AVX instruction to perform the required integer operation.

15.11 HANDLING PORT 5 PRESSURE

Port 5 in Sandy Bridge microarchitecture includes shuffle units which frequently become a performance bottleneck. Ice Lake Client microarchitecture has added a restricted, in-lane shuffle unit to port 1 to help reduce some of the pressure. Shuffle operations which can be restructured to operate in-lane will benefit from this unit. Sometimes it is possible to replace shuffle instructions that dispatch only on port 5, with different instructions and improve performance by reducing port 5 pressure. For more information, see Table E-11.

15.11.1 Replace Shuffles with Blends

There are a few cases where shuffles such as VSHUFPS or VPERM2F128 can be replaced by blend instructions. Intel AVX shuffles are executed only on port 5, while blends are also executed on port 0. Therefore, replacing shuffles with blends could reduce port 5 pressure. The following figure shows how a VSHUFPS is implemented using VBLENDPS.



The following example shows two implementations of an 8x8 Matrix transpose. In both cases, the bottleneck is Port 5 pressure. Alternative 1 uses 12 vshufps instructions that are executed only on port 5. Alternative 2 replaces eight of the vshufps instructions with the vblendps instruction which can be executed on Port 0.

Example 15-19. 8x8 Matrix Transpose - Replace Shuffles with Blends

256-bit AVX using VSHUFPS	AVX replacing VSHUFPS with VBLENDPS
mov rcx, inpBuf mov rdx, outBuf mov r10, NumOfLoops loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vshufps ymm10, ymm6, ymm3, 0xE4 vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vshufps ymm11, ymm0, ymm6, 0xE4	mov rcx, inpBuf mov rdx, outBuf mov r10, NumOfLoops loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vblendps ymm10, ymm6, ymm3, 0xCC vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vblendps ymm11, ymm0, ymm6, 0xCC

Example 15-19. 8x8 Matrix Transpose - Replace Shuffles with Blends (Contd.)

256-bit AVX using VSHUFPS	AVX replacing VSHUFPS with VBLENDPS
vshufps ymm12, ymm3, ymm1, 0xE4	vblendps ymm12, ymm3, ymm1, 0xCC
vperm2f128 ymm3, ymm10, ymm11, 0x20	vperm2f128 ymm3, ymm10, ymm11, 0x20
vmovaps [rdx], ymm3	vmovaps [rdx], ymm3
vunpckhps ymm5, ymm13, ymm14	vunpckhps ymm5, ymm13, ymm14
vshufps ymm13, ymm6, ymm9, 0xE4	vblendps ymm13, ymm6, ymm9, 0xCC
vunpckhps ymm4, ymm15, ymm2	vunpckhps ymm4, ymm15, ymm2
vperm2f128 ymm2, ymm12, ymm13, 0x20	vperm2f128 ymm2, ymm12, ymm13, 0x20
vmovaps 32[rdx], ymm2	vmovaps 32[rdx], ymm2
vshufps ymm14, ymm8, ymm7, 0x4E	vshufps ymm14, ymm8, ymm7, 0x4E
vshufps ymm15, ymm14, ymm7, 0xE4	vblendps ymm15, ymm14, ymm7, 0xCC
vshufps ymm7, ymm5, ymm4, 0x4E	vshufps ymm7, ymm5, ymm4, 0x4E
vshufps ymm8, ymm8, ymm14, 0xE4	vblendps ymm8, ymm8, ymm14, 0xCC
vshufps ymm5, ymm5, ymm7, 0xE4	vblendps ymm5, ymm5, ymm7, 0xCC
vperm2f128 ymm6, ymm8, ymm5, 0x20	vperm2f128 ymm6, ymm8, ymm5, 0x20
vmovaps 64[rdx], ymm6	vmovaps 64[rdx], ymm6
vshufps ymm4, ymm7, ymm4, 0xE4	vblendps ymm4, ymm7, ymm4, 0xCC
vperm2f128 ymm7, ymm15, ymm4, 0x20	vperm2f128 ymm7, ymm15, ymm4, 0x20
vmovaps 96[rdx], ymm7	vmovaps 96[rdx], ymm7
vperm2f128 ymm1, ymm10, ymm11, 0x31	vperm2f128 ymm1, ymm10, ymm11, 0x31
vperm2f128 ymm0, ymm12, ymm13, 0x31	vperm2f128 ymm0, ymm12, ymm13, 0x31
vmovaps 128[rdx], ymm1	vmovaps 128[rdx], ymm1
vperm2f128 ymm5, ymm8, ymm5, 0x31	vperm2f128 ymm5, ymm8, ymm5, 0x31
vperm2f128 ymm4, ymm15, ymm4, 0x31	vperm2f128 ymm4, ymm15, ymm4, 0x31
vmovaps 160[rdx], ymm0	vmovaps 160[rdx], ymm0
vmovaps 192[rdx], ymm5	vmovaps 192[rdx], ymm5
vmovaps 224[rdx], ymm4	vmovaps 224[rdx], ymm4
dec r10	dec r10
jnz loop1	jnz loop1

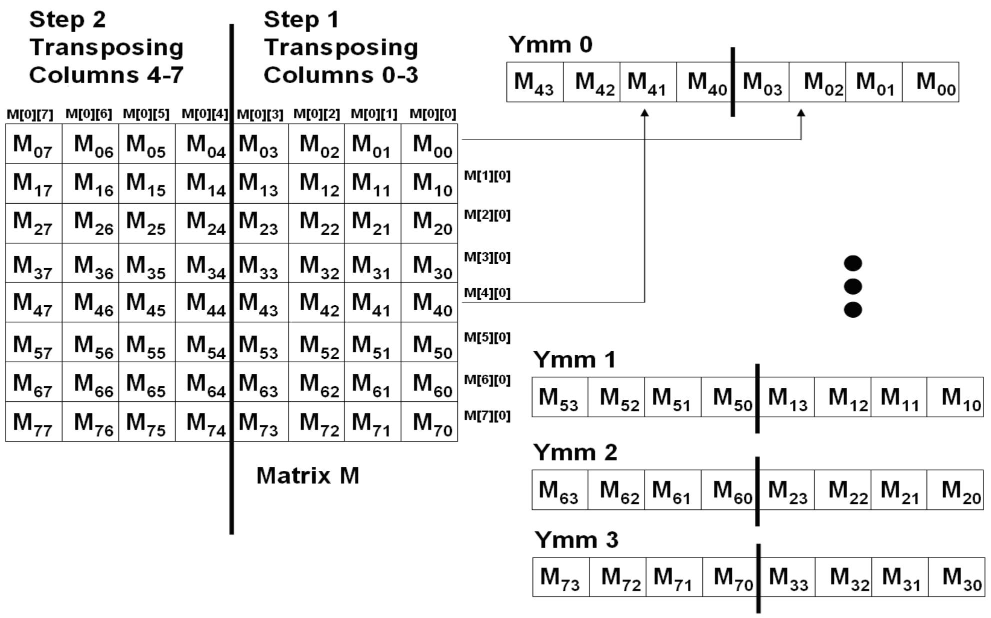
In Example 15-19, replacing VSHUFPS with VBLENDPS relieved port 5 pressure and can gain almost 40% speedup.

Assembly/Compiler Coding Rule 60. (M impact, M generality) *Use Blend instructions in lieu of shuffle instruction in AVX whenever possible.*

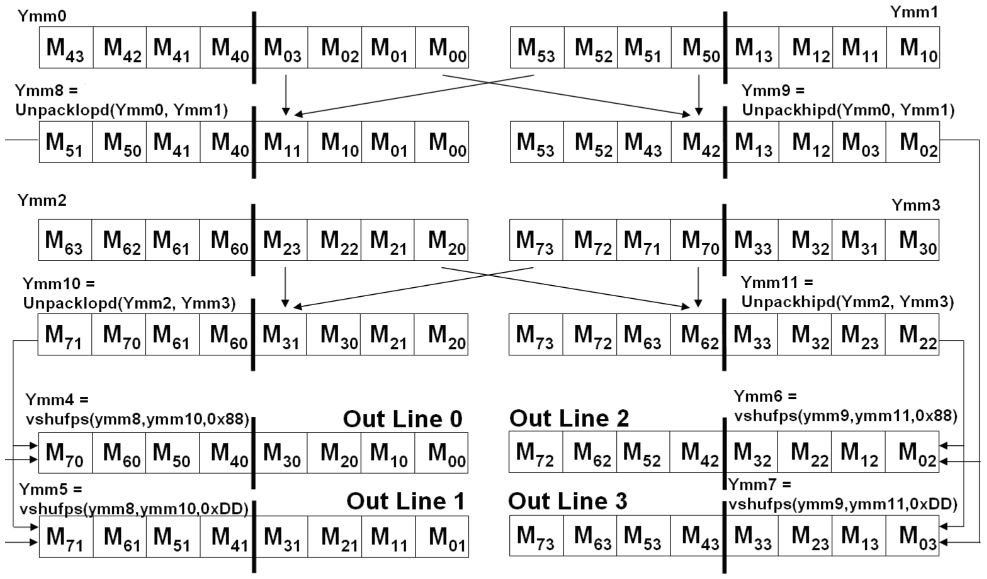
15.11.2 Design Algorithm with Fewer Shuffles

In some cases you can reduce port 5 pressure by changing the algorithm to use less shuffles. The figure below shows that the transpose moved all the elements in rows 0-4 to the low lanes, and all the elements in rows 4-7 to the high lanes. Therefore, using 256-bit loads in the beginning of the algorithm requires using VPERM2F128 in order to swap elements between the lanes. The processor executes the VPERM2F128 instruction only on port 5.

Example 15-19 used eight 256-bit loads and eight VPERM2F128 instructions. You can implement the same 8x8 Matrix Transpose using VINSERTF128 instead of the 256-bit loads and the eight VPERM2F128. Using VINSERTF128 from memory is executed in the load ports and on port 0 or 5. The original method required loads that are performed on the load ports and VPERM2F128 that is only performed on port 5. Therefore redesigning the algorithm to use VINSERTF128 reduces port 5 pressure and improves performance.


Step 2 Transposing Columns 4-7
Step 1 Transposing Columns 0-3
Matrix M
Ymm 0
Ymm 1
Ymm 2
Ymm 3
M[0][7] M[0][6] M[0][5] M[0][4] M[0][3] M[0][2] M[0][1] M[0][0]
M[1][0] M[2][0] M[3][0] M[4][0] M[5][0] M[6][0] M[7][0]

The following figure describes step 1 of the 8x8 matrix transpose with vinsertf128. Step 2 performs the same operations on different columns.


Ymm0
Ymm1
Ymm8 = Unpacklopd(Ymm0, Ymm1)
Ymm9 = Unpackhipd(Ymm0, Ymm1)
Ymm2
Ymm3
Ymm10 = Unpacklopd(Ymm2, Ymm3)
Ymm11 = Unpackhipd(Ymm2, Ymm3)
Ymm4 = vshufps(ymm8,ymm10,0x88)
Out Line 0
Out Line 2
Ymm6 = vshufps(ymm9,ymm11,0x88)
Ymm5 = vshufps(ymm8,ymm10,0xDD)
Out Line 1
Out Line 3
Ymm7 = vshufps(ymm9,ymm11,0xDD)

Example 15-20. 8x8 Matrix Transpose Using VINSERTPS

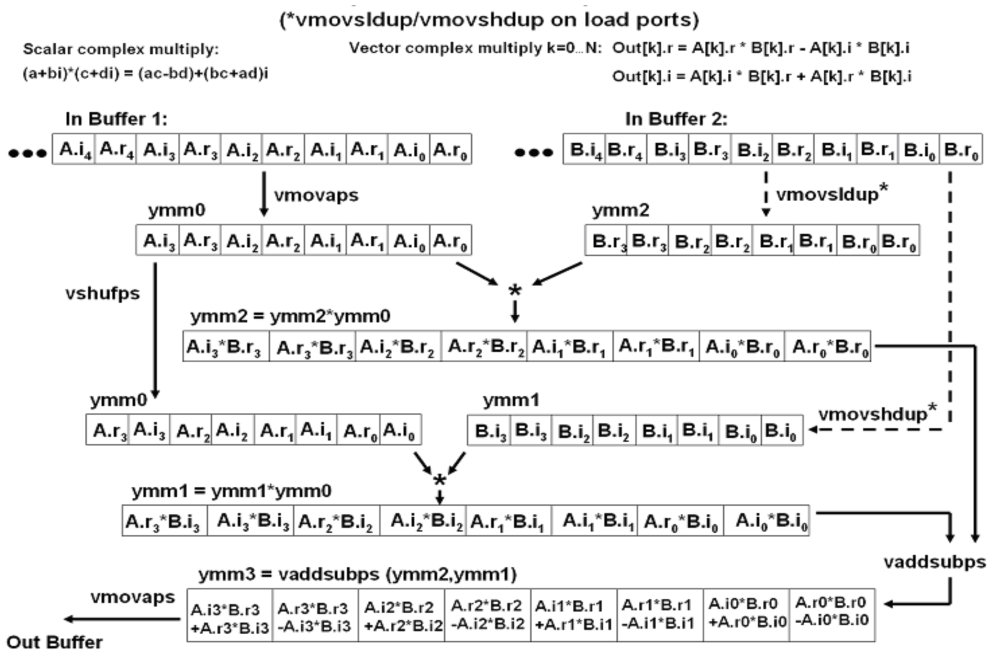
```
    mov         rcx, inpBuf
    mov         rdx, outBuf
    mov         r10, NumOfLoops
loop1:
    vmovaps       xmm0, [rcx]
    vinsertf128   ymm0, ymm0, [rcx + 128], 1
    vmovaps       xmm1, [rcx + 32]
    vinsertf128   ymm1, ymm1, [rcx + 160], 1

    vunpcklpd     ymm8, ymm0, ymm1
    vunpckhpd     ymm9, ymm0, ymm1
    vmovaps       xmm2, [rcx+64]
    vinsertf128   ymm2, ymm2, [rcx + 192], 1
    vmovaps       xmm3, [rcx+96]
    vinsertf128   ymm3, ymm3, [rcx + 224], 1
    vunpcklpd     ymm10, ymm2, ymm3
    vunpckhpd     ymm11, ymm2, ymm3
    vshufps       ymm4, ymm8, ymm10, 0x88
    vmovaps       [rdx], ymm4
    vshufps       ymm5, ymm8, ymm10, 0xDD
    vmovaps       [rdx+32], ymm5
    vshufps       ymm6, ymm9, ymm11, 0x88
    vmovaps       [rdx+64], ymm6
    vshufps       ymm7, ymm9, ymm11, 0xDD
    vmovaps       [rdx+96], ymm7
    vmovaps       xmm0, [rcx+16]
    vinsertf128   ymm0, ymm0, [rcx + 144], 1
    vmovaps       xmm1, [rcx + 48]
    vinsertf128   ymm1, ymm1, [rcx + 176], 1

    vunpcklpd     ymm8, ymm0, ymm1
    vunpckhpd     ymm9, ymm0, ymm1

    vmovaps       xmm2, [rcx+80]
    vinsertf128   ymm2, ymm2, [rcx + 208], 1
    vmovaps       xmm3, [rcx+112]
    vinsertf128   ymm3, ymm3, [rcx + 240], 1

    vunpcklpd     ymm10, ymm2, ymm3
    vunpckhpd     ymm11, ymm2, ymm3
    vshufps       ymm4, ymm8, ymm10, 0x88
    vmovaps       [rdx+128], ymm4
    vshufps       ymm5, ymm8, ymm10, 0xDD
    vmovaps       [rdx+160], ymm5
    vshufps       ymm6, ymm9, ymm11, 0x88
    vmovaps       [rdx+192], ymm6
    vshufps       ymm7, ymm9, ymm11, 0xDD
    vmovaps       [rdx+224], ymm7
    dec           r10
    jnz           loop1
```

In Example 15-20, this reduced port 5 pressure further than the combination of VSHUFPS with VBLENDPS in Example 15-19. It can gain 70% speedup relative to relying on VSHUFPS alone in Example 15-19.

15.11.3 Perform Basic Shuffles on Load Ports

Some shuffles can be executed in the load ports (ports 2, 3) if the source is from memory. The following example shows how moving some shuffles (vmovsldup/vmovshdup) from Port 5 to the load ports improves performance significantly.

The following figure describes an Intel AVX implementation of the complex multiply algorithm with vmovsldup/vmovshdup on the load ports.


(*vmovsldup/vmovshdup on load ports)
Scalar complex multiply:
(a+bi)*(c+di) = (ac-bd)+(bc+ad)i
Vector complex multiply k=0..N: Out[k].r = A[k].r * B[k].r - A[k].i * B[k].i
Out[k].i = A[k].i * B[k].r + A[k].r * B[k].i
In Buffer 1:
In Buffer 2:
vmovaps
vmovsldup*
ymm0
ymm2
vshufps
ymm2 = ymm2*ymm0
ymm0
ymm1
vmovshdup*
ymm1 = ymm1*ymm0
vaddsubps
ymm3 = vaddsubps (ymm2,ymm1)
vmovaps
Out Buffer

Example 15-21 includes two versions of the complex multiply. Both versions are unrolled twice. Alternative 1 shuffles all the data in registers. Alternative 2 shuffles data while it is loaded from memory.

Example 15-21. Port 5 versus Load Port Shuffles

Shuffles data in registers	Shuffling loaded data
mov rax, inPtr1	mov rax, inPtr1
mov rbx, inPtr2	mov rbx, inPtr2
mov rdx, outPtr	mov rdx, outPtr
mov r8, len	mov r8, len
xor rcx, rcx	xor rcx, rcx
loop1:	loop1:
vmovaps ymm0, [rax +8*rcx]	vmovaps ymm0, [rax +8*rcx]
vmovaps ymm4, [rax +8*rcx +32]	vmovaps ymm4, [rax +8*rcx +32]
vmovaps ymm3, [rbx +8*rcx]	
vmovsldup ymm2, ymm3	vmovsldup ymm2, [rbx +8*rcx]
vmulps ymm2, ymm2, ymm0	vmulps ymm2, ymm2, ymm0
vshufps ymm0, ymm0, ymm0, 177	vshufps ymm0, ymm0, ymm0, 177
vmovshdup ymm1, ymm3	vmovshdup ymm1, [rbx +8*rcx]
vmulps ymm1, ymm1, ymm0	vmulps ymm1, ymm1, ymm0
vmovaps ymm7, [rbx +8*rcx +32]	vmovsldup ymm6, [rbx +8*rcx +32]
vmovsldup ymm6, ymm7	vmulps ymm6, ymm6, ymm4
vmulps ymm6, ymm6, ymm4	vaddsubps ymm3, ymm2, ymm1
vaddsubps ymm2, ymm2, ymm1	vmovshdup ymm5, [rbx +8*rcx +32]
vmovshdup ymm5, ymm7	
vmovaps [rdx+8*rcx], ymm2	vmovaps [rdx +8*rcx], ymm3
vshufps ymm4, ymm4, ymm4, 177	vshufps ymm4, ymm4, ymm4, 177
vmulps ymm5, ymm5, ymm4	vmulps ymm5, ymm5, ymm4
vaddsubps ymm6, ymm6, ymm5	vaddsubps ymm7, ymm6, ymm5
vmovaps [rdx+8*rcx+32], ymm6	vmovaps [rdx +8*rcx +32], ymm7
add rcx, 8	add rcx, 8
cmp rcx, r8	cmp rcx, r8
jl loop1	jl loop1

15.12 DIVIDE AND SQUARE ROOT OPERATIONS

In Intel microarchitectures prior to Skylake, the SSE divide and square root instructions DIVPS and SQRTPS have a latency of 14 cycles (or the neighborhood) and they are not pipelined. This means that the throughput of these instructions is one in every 14 cycles. The 256-bit Intel AVX instructions VDIVPS and VSQRTPS execute with 128-bit data path and have a latency of 28 cycles and they are not pipelined as well. Therefore, the performance of the Intel SSE divide and square root instructions is similar to the Intel AVX 256-bit instructions on Sandy Bridge microarchitecture.

With the Skylake microarchitecture, 256-bit and 128-bit version of (V)DIVPS/(V)SQRTPS have the same latency because the 256-bit version can execute with a 256-bit data path. The latency is improved and is pipelined to execute with significantly improved throughput. See Appendix D.3, "Latency and Throughput".

In microarchitectures that provide DIVPS/SQRTPS with high latency and low throughput, it is possible to speed up single-precision divide and square root calculations using the (V)RSQRTPS and (V)RCPPS instructions. For example, with 128-bit RCPPS/RSQRTPS at 5-cycle latency and 1-cycle throughput or with 256-bit implementation of these instructions at 7-cycle latency and 2-cycle throughput, a single Newton-Raphson iteration or Taylor approximation can achieve almost the same precision as the

(V)DIVPS and (V)SQRTPS instructions. See Intel® 64 and IA-32 Architectures Software Developer's Manual for more information on these instructions.

In some cases, when the divide or square root operations are part of a larger algorithm that hides some of the latency of these operations, the approximation with Newton-Raphson can slow down execution, because more micro-ops, coming from the additional instructions, fill the pipe.

With the Skylake microarchitecture, choosing between approximate reciprocal instruction alternative versus DIVPS/SQRTPS for optimal performance of simple algebraic computations depend on a number of factors. Table 15-5 shows several algebraic formula the throughput comparison of implementations of different numeric accuracy tolerances. In each row, 24-bit accurate implementations are IEEE-compliant and using the respective instructions of 128-bit or 256-bit ISA. The columns of 22-bit and 11-bit accurate implementations are using approximate reciprocal instructions of the respective instruction set.

Table 15-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture

Algorithm	Instruction Type	24-bit Accurate	22-bit Accurate	11-bit Accurate
Z = X/Y	SSE	1X	0.9X	1.3X
	256-bit AVX	1X	1.5X	2.6X
$Z = X^{0.5}$	SSE	1X	0.7X	2X
	256-bit AVX	1X	1.4X	3.4X
$Z = X^{-0.5}$	SSE	1X	1.7X	4.3X
	256-bit AVX	1X	3X	7.7X
$Z = (X *Y + Y*Y)^{0.5}$	SSE	1X	0.75X	0.85X
	256-bit AVX	1X	1.1X	1.6X
Z = (X+2Y+3)/(Z-2Y-3)	SSE	1X	0.85X	1X
	256-bit AVX	1X	0.8X	1X

If targeting processors based on the Skylake microarchitecture, Table 15-5 can be summarized as:

- For 256- bit AVX code, Newton-Raphson approximation can be beneficial on Skylake microarchitecture when the algorithm contains only operations executed on the divide unit. However, when single precision divide or square root operations are part of a longer computation, the lower latency of the DIVPS or SQRTPS instructions can lead to better overall performance.
- For SSE or 128-bit AVX implementation, consider use of approximation for divide and square root instructions only for algorithms that do not require precision higher than 11-bit or algorithms that contain multiple operations executed on the divide unit.

Table 15-6 summarizes recommended calculation methods of divisions or square root when using single-precision instructions, based on the desired accuracy level across recent generations of Intel microarchitectures.

Table 15-6. Single-Precision Divide and Square Root Alternatives

Operation	Accuracy Tolerance	Recommendation
Divide	24 bits (IEEE)	DIVPS
	~ 22 bits	Skylake: Consult Table 15-5 Prior uarch: RCPPS + 1 Newton-Raphson Iteration + MULPS
	~ 11 bits	RCPPS + MULPS
Reciprocal square root	24 bits (IEEE)	SQRTPS + DIVPS
	~ 22 bits	RSQRTPS + 1 Newton-Raphson Iteration
	~ 11 bits	RSQRTPS

Table 15-6. Single-Precision Divide and Square Root Alternatives (Contd.)

Operation	Accuracy Tolerance	Recommendation
Square root	24 bits (IEEE)	SQRTPS
	~ 22 bits	Skylake: Consult Table 15-5 Prior uarch: RSQRTPS + 1 Newton-Raphson Iteration + MULPS
	~ 11 bits	RSQRTPS + RCPPS

15.12.1 Single-Precision Divide

To compute:

Z[i]=A[i]/B[i]

On a large vector of single-precision numbers, Z[i] can be calculated by a divide operation, or by multiplying 1/B[i] by A[i].

Denoting B[i] by N, it is possible to calculate 1/N using the (V)RCPPS instruction, achieving approximately 11-bit precision.

For better accuracy you can use the one Newton-Raphson iteration:

```
X_(0 )  ~= 1/N                ; Initial estimation, rcp(N)
X_(0 ) =  1/N*(1-E)
E=1-N*X_0                                ; E ~= 2^(-11)
X_1=X_0*(1+E)=1/N*(1-E^2 )      ; E^2 ~= 2^(-22)
X_1=X_0*(1+1-N*X_0 )= 2 *X_0 - N*X_0^2
```

X_1 is an approximation of 1/N with approximately 22-bit precision.

Example 15-22. Divide Using DIVPS for 24-bit Accuracy

SSE code using DIVPS	Using VDIVPS
mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx loop1: movups xmm0, [rax+rdx*1] movups xmm1, [rbx+rdx*1] divps xmm0, xmm1 movups [rcx+rdx*1], xmm0 add rdx, 16 cmp rdx, rsi jl loop1	mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx loop1: vmovups ymm0, [rax+rdx*1] vmovups ymm1, [rbx+rdx*1] vdivps ymm0, ymm0, ymm1 vmovups [rcx+rdx*1], ymm0 add rdx, 32 cmp rdx, rsi jl loop1

Example 15-23. Divide Using RCPPS 11-bit Approximation

SSE code using RCPPS	Using VRCPPS
mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx loop1: movups xmm0,[rax+rdx*1] movups xmm1,[rbx+rdx*1] rcpps xmm1,xmm1 mulps xmm0,xmm1 movups [rcx+rdx*1],xmm0 add rdx, 16 cmp rdx, rsi jl loop1	mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx loop1: vmovups ymm0, [rax+rdx] vmovups ymm1, [rbx+rdx] vrcpps ymm1, ymm1 vmulps ymm0, ymm0, ymm1 vmovups [rcx+rdx], ymm0 add rdx, 32 cmp rdx, rsi jl loop1

Example 15-24. Divide Using RCPPS and Newton-Raphson Iteration

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy
mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx loop1: movups xmm0, [rax+rdx*1] movups xmm1, [rbx+rdx*1] rcpps xmm3, xmm1 movaps xmm2, xmm3 addps xmm3, xmm2 mulps xmm2, xmm2 mulps xmm2, xmm1 subps xmm3, xmm2 mulps xmm0, xmm3 movups [rcx+rdx*1], xmm0 add rdx, 16 cmp rdx, rsi jl loop1	mov rax, pIn1 mov rbx, pIn2 mov rcx, pOut mov rsi, iLen xor rdx, rdx loop1: vmovups ymm0, [rax+rdx] vmovups ymm1, [rbx+rdx] vrcpps ymm3, ymm1 vaddps ymm2, ymm3, ymm3 vmulps ymm3, ymm3, ymm3 vmulps ymm3, ymm3, ymm1 vsubps ymm2, ymm2, ymm3 vmulps ymm0, ymm0, ymm2 vmovups [rcx+rdx], ymm0 add rdx, 32 cmp rdx, rsi jl loop1

15.12.2 Single-Precision Reciprocal Square Root

To compute Z[i]=1/ (A[i]) ^0.5 on a large vector of single-precision numbers, denoting A[i] by N, it is possible to calculate 1/N using the (V)RSQRTPS instruction.

For better accuracy you can use one Newton-Raphson iteration:

```
X_0 ~=1/N ; Initial estimation RCP(N)
E=1-N*X_0^2
X_0= (1/N)^0.5 * ((1-E)^0.5 )  =  (1/N)^0.5 * (1-E/2) ; E/2~= 2^(-11)
X_1=X_0*(1+E/2)  ~= (1/N)^0.5 * (1-E^2/4)            ; E^2/4?2^(-22)
X_1=X_0*(1+1/2-1/2*N*X_0^2 )= 1/2*X_0*(3-N*X_0^2)
X1 is an approximation of (1/N)^0.5 with approximately 22-bit precision.
```

Example 15-25. Reciprocal Square Root Using DIVPS+SQRTPS for 24-bit Accuracy

Using SQRTPS, DIVPS	Using VSQRTPS, VDIVPS
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, iLen	mov rcx, iLen
xor rdx, rdx	xor rdx, rdx
loop1:	loop1:
movups xmm1, [rax+rdx]	vmovups ymm1, [rax+rdx]
sqrtps xmm0, xmm1	vsqrtps ymm0, ymm1
divps xmm0, xmm1	vdivps ymm0, ymm0, ymm1
movups [rbx+rdx], xmm0	vmovups [rbx+rdx], ymm0
add rdx, 16	add rdx, 32
cmp rdx, rcx	cmp rdx, rcx
jl loop1	jl loop1

Example 15-26. Reciprocal Square Root Using RSQRTPS 11-bit Approximation

SSE code using RSQRTPS	Using VRSQRTPS
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, iLen	mov rcx, iLen
xor rdx, rdx	xor rdx, rdx
loop1:	
rsqrtps xmm0, [rax+rdx]	loop1:
movups [rbx+rdx], xmm0	vrsqrtps ymm0, [rax+rdx]
add rdx, 16	vmovups [rbx+rdx], ymm0
cmp rdx, rcx	add rdx, 32
jl loop1	cmp rdx, rcx
	jl loop1

Example 15-27. Reciprocal Square Root Using RSQRTPS and Newton-Raphson Iteration

RSQRTPS + MULPS ~ 22 bit accuracy	VRSQRTPS + VMULPS ~ 22 bit accuracy
__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};	__declspec(align(32)) float half[8] = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5};
__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};	__declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
__asm	__asm
{	{
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, iLen	mov rcx, iLen
xor rdx, rdx	xor rdx, rdx
movups xmm3, [three]	vmovups ymm3, [three]
movups xmm4, [minus_half]	vmovups ymm4, [half]
loop1:	loop1:
movups xmm5, [rax+rdx]	vmovups ymm5, [rax+rdx]
rsqrtps xmm0, xmm5	vrsqrtps ymm0, ymm5
movaps xmm2, xmm0	
mulps xmm0, xmm0	vmulps ymm2, ymm0, ymm0
mulps xmm0, xmm5	vmulps ymm2, ymm2, ymm5
subps xmm0, xmm3	vsubps ymm2, ymm3, ymm2
mulps xmm0, xmm2	vmulps ymm0, ymm0, ymm2
mulps xmm0, xmm4	vmulps ymm0, ymm0, ymm4
movups [rbx+rdx], xmm0	
add rdx, 16	vmovups [rbx+rdx], ymm0
cmp rdx, rcx	add rdx, 32
jl loop1	cmp rdx, rcx
}	jl loop1
	}

15.12.3 Single-Precision Square Root

To compute Z[i]= (A[i])^0.5 on a large vector of single-precision numbers, denoting A[i] by N, the approximation for N^0.5 is N multiplied by (1/N)^0.5 , where the approximation for (1/N)^0.5 is described in the previous section.

To get approximately 22-bit precision of N^0.5, use the following calculation:

N^0.5 = X_1*N = 1/2*N*X_0*(3-N*X_0^2)

Example 15-28. Square Root Using SQRTPS for 24-bit Accuracy

Using SQRTPS	Using VSQRTPS
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, iLen	mov rcx, iLen
xor rdx, rdx	xor rdx, rdx
loop1:	loop1:
movups xmm1, [rax+rdx]	vmovups ymm1, [rax+rdx]
sqrtps xmm1, xmm1	vsqrtps ymm1,ymm1
movups [rbx+rdx], xmm1	vmovups [rbx+rdx], ymm1
add rdx, 16	add rdx, 32
cmp rdx, rcx	cmp rdx, rcx
jl loop1	jl loop1

Example 15-29. Square Root Using RSQRTPS 11-bit Approximation

SSE code using RSQRTPS	Using VRSQRTPS
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, iLen	mov rcx, iLen
xor rdx, rdx	xor rdx, rdx
loop1:	vxorps ymm8, ymm8, ymm8
movups xmm1, [rax+rdx]	loop1:
xorps xmm8, xmm8	vmovups ymm1, [rax+rdx]
cmpneqps xmm8, xmm1	vcmpneqps ymm9, ymm8, ymm1
rsqrtps xmm1, xmm1	vrsqrtps ymm1, ymm1
rcpps xmm1, xmm1	vrcpps ymm1, ymm1
andps xmm1, xmm8	vandps ymm1, ymm1, ymm9
movups [rbx+rdx], xmm1	vmovups [rbx+rdx], ymm1
add rdx, 16	add rdx, 32
cmp rdx, rcx	cmp rdx, rcx
jl loop1	jl loop1

Example 15-30. Square Root Using RSQRTPS and One Taylor Series Expansion

RSQRTPS + Taylor ~ 22 bit accuracy	VRSQRTPS + Taylor ~ 22 bit accuracy
__declspec(align(16)) float minus_half[4] =	__declspec(align(32)) float three[8] =
{-0.5, -0.5, -0.5, -0.5};	{3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
__declspec(align(16)) float three[4] =	__declspec(align(32)) float minus_half[8] =
{3.0, 3.0, 3.0, 3.0};	{-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5};
__asm	__asm
{	{
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, iLen	mov rcx, iLen
xor rdx, rdx	xor rdx, rdx
movups xmm6, [three]	vmovups ymm6, [three]
movups xmm7, [minus_half]	vmovups ymm7, [minus_half]
loop1:	vxorps ymm8, ymm8, ymm8
movups xmm3, [rax+rdx]	loop1:
rsqrtps xmm1, xmm3	vmovups ymm3, [rax+rdx]
xorps xmm8, xmm8	vrsqrtps ymm4, ymm3
cmpneqps xmm8, xmm3	vcmpneqps ymm9, ymm8, ymm3
andps xmm1, xmm8	vandps ymm4, ymm4, ymm9
movaps xmm4, xmm1	vmulps ymm1, ymm4, ymm3
mulps xmm1, xmm3	vmulps ymm2, ymm1, ymm4
movaps xmm5, xmm1	vsubps ymm2, ymm2, ymm6
mulps xmm1, xmm4	vmulps ymm1, ymm1, ymm2
subps xmm1, xmm6	vmulps ymm1, ymm1, ymm7
mulps xmm1, xmm5	vmovups [rbx+rdx], ymm1
mulps xmm1, xmm7	add rdx, 32
movups [rbx+rdx], xmm1	cmp rdx, rcx
add rdx, 16	jl loop1
cmp rdx, rcx	}
jl loop1	
}	

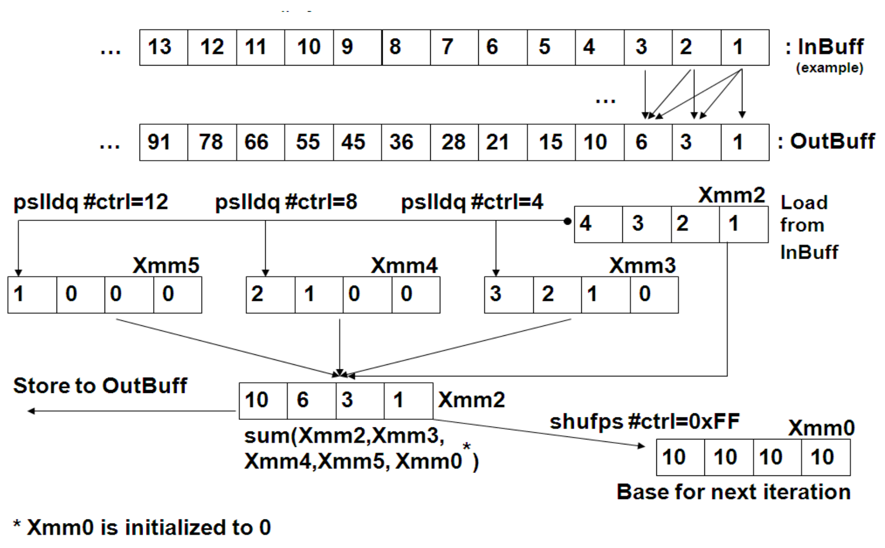
15.13 OPTIMIZATION OF ARRAY SUB SUM EXAMPLE

This section shows the transformation of SSE implementation of Array Sub Sum algorithm to Intel AVX implementation.

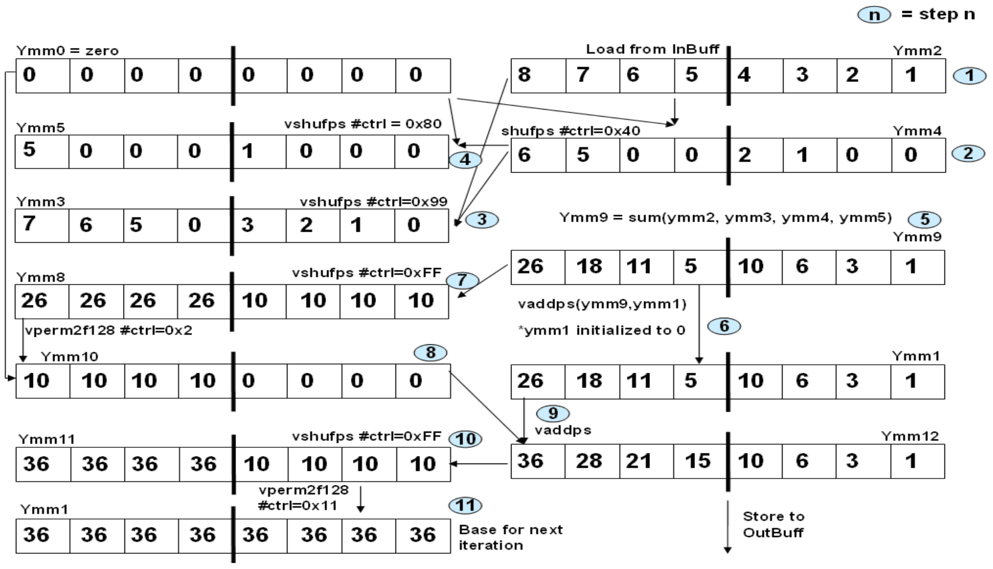
The Array Sub Sum algorithm is:

$Y_{[i]}$ = Sum of k from 0 to i ($X_{[k]}$) = $X_{[0]} + X_{[1]} + .. + X_{[i]}$

The following figure describes the SSE implementation.



The figure below describes the Intel AVX implementation of the Array Sub Sums algorithm. The PSLLDQ is an integer SIMD instruction which does not have an AVX equivalent. It is replaced by VSHUFPS.



Example 15-31. Array Sub Sums Algorithm

SSE code	AVX code
mov rax, InBuff	mov rax, InBuff
mov rbx, OutBuff	mov rbx, OutBuff
mov rdx, len	mov rdx, len
xor rcx, rcx	xor rcx, rcx
xorps xmm0, xmm0	vxorps ymm0, ymm0, ymm0
	vxorps ymm1, ymm1, ymm1
loop1:	loop1:
movaps xmm2, [rax+4*rcx]	vmovaps ymm2, [rax+4*rcx]
movaps xmm3, xmm2	vshufps ymm4, ymm0, ymm2, 0x40
movaps xmm4, xmm2	vshufps ymm3, ymm4, ymm2, 0x99
movaps ymm5, ymm2	vshufps ymm5, ymm0, ymm4, 0x80
pslldq xmm3, 4	vaddps ymm6, ymm2, ymm3
pslldq xmm4, 8	vaddps ymm7, ymm4, ymm5
pslldq xmm5, 12	vaddps ymm9, ymm6, ymm7
addps xmm2, xmm3	vaddps ymm1, ymm9, ymm1
addps xmm4, xmm5	vshufps ymm8, ymm9, ymm9, 0xff
addps ymm2, xmm4	vperm2f128 ymm10, ymm8, ymm0, 0x2
addps xmm2, xmm0	vaddps ymm12, ymm1, ymm10
movaps xmm0, ymm2	vshufps ymm11, ymm12, ymm12, 0xff
shufps xmm0, xmm2, 0xFF	vperm2f128 ymm1, ymm11, ymm11, 0x11
movaps [rbx+4*rcx], xmm2	vmovaps [rbx+4*rcx], ymm12
add rcx, 4	add rcx, 8
cmp rcx, rdx	cmp rcx, rdx
jl loop1	jl loop1

Example 15-31 shows SSE implementation of array sub sum and AVX implementation. The AVX code is about 40% faster, though not on microarchitectures where there are more compute than shuffle ports.

15.14 HALF-PRECISION FLOATING-POINT CONVERSIONS

In applications that use floating-point and require only the dynamic range and precision offered by the 16-bit floating-point format, storing persistent floating-point data encoded in 16-bits has strong advantages in memory footprint and bandwidth conservation. These situations are encountered in some graphics and imaging workloads.

The encoding format of half-precision floating-point numbers can be found in Chapter 4, “Data Types” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Instructions to convert between packed, half-precision floating-point numbers and packed single-precision floating-point numbers is described in Chapter 14, “Programming with Intel® AVX, FMA, and Intel® AVX2” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* and in the reference pages of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

To perform computations on half precision floating-point data, packed 16-bit FP data elements must be converted to single precision format first, and the single-precision results converted back to half precision format, if necessary. These conversions of 8 data elements using 256-bit instructions are very fast and handle the special cases of denormal numbers, infinity, zero and NaNs properly.

15.14.1 Packed Single-Precision to Half-Precision Conversion

To convert the data in single precision floating-point format to half precision format, without special hardware support like VCVTPS2PH, a programmer needs to do the following:

- Correct exponent bias to permitted range for each data element.
- Shift and round the significand of each data element.
- Copy the sign bit to bit 15 of each element.
- Take care of numbers outside the half precision range.
- Pack each data element to a register of half size.

Example 15-32 compares two implementations of floating-point conversion from single precision to half precision. The code on the left uses packed integer shift instructions that is limited to 128-bit SIMD instruction set. The code on right is unrolled twice and uses the VCVTPS2PH instruction.

Example 15-32. Single-Precision to Half-Precision Conversion

AVX-128 code

```
    __asm {
    mov      rax, pIn
    mov      rbx, pOut
    mov      rcx, bufferSize
    add      rcx, rax
    vmovdqu xmm0,SignMask16
    vmovdqu xmm1,ExpBiasFixAndRound
    vmovdqu xmm4,SignMaskNot32
    vmovdqu xmm5,MaxConvertibleFloat
    vmovdqu xmm6,MinFloat
loop:
    vmovdqu      xmm2, [rax]
    vmovdqu      xmm3, [rax+16]
    vpaddd       xmm7, xmm2, xmm1
    vpaddd       xmm9, xmm3, xmm1
    vpand        xmm7, xmm7, xmm4
    vpand        xmm9, xmm9, xmm4
    add          rax, 32
    vminps       xmm7, xmm7, xmm5
    vminps       xmm9, xmm9, xmm5
    vpcmpgtd     xmm8, xmm7, xmm6
    vpcmpgtd     xmm10, xmm9, xmm6
    vpand        xmm7, xmm8, xmm7
    vpand        xmm9, xmm10, xmm9
    vpackssdw    xmm2, xmm3, xmm2
    vpsrad       xmm7, xmm7, 13
    vpsrad       xmm8, xmm9, 13
    vpand        xmm2, xmm2, xmm0
    vpackssdw    xmm3, xmm7, xmm9
    vpaddw       xmm3, xmm3, xmm2
    vmovdqu      [rbx], xmm3
    add          rbx, 16
    cmp          rax, rcx
    jl           loop
```

VCVTPS2PH code

```
    __asm {
    mov          rax, pIn
    mov          rbx, pOut
    mov          rcx, bufferSize
    add          rcx, rax
loop:
    vmovups      ymm0,[rax]
    vmovups      ymm1,[rax+32]
    add          rax, 64
    vcvtps2ph    [rbx],ymm0, roundingCtrl
    vcvtps2ph    [rbx+16],ymm1,roundingCtrl
    add          rbx, 32
    cmp          rax, rcx
    jl           loop
```

The code using VCVTPS2PH is approximately four times faster than the AVX-128 sequence. Although it is possible to load 8 data elements at once with 256-bit AVX, most of the per-element conversion operations require packed integer instructions which do not have 256-bit extensions yet. Using VCVTPS2PH is not only faster but also provides handling of special cases that do not encode to normal half-precision floating-point values.

15.14.2 Packed Half-Precision to Single-Precision Conversion

Example 15-33 compares two implementations using AVX-128 code and with VCVTPH2PS.

Conversion from half precision to single precision floating-point format is easier to implement, yet using VCVTPH2PS instruction performs about 2.5 times faster than the alternative AVX-128 code.

Example 15-33. Half-Precision to Single-Precision Conversion

AVX-128 code	VCVTPS2PH code
__asm {	__asm {
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov rcx, bufferSize	mov rcx, bufferSize
add rcx, rax	add rcx, rax
vmovdqu xmm0,SignMask16	loop:
vmovdqu xmm1,ExpBiasFix16	vcvtph2ps ymm0,[rax]
vmovdqu xmm2,ExpMaskMarker	vcvtph2ps ymm1,[rax+16]
loop:	add rax, 32
vmovdqu xmm3, [rax]	vmovups [rbx], ymm0
add rax, 16	vmovups [rbx+32], ymm1
vpandn xmm4, xmm0, xmm3	add rbx, 64
vpand xmm5, xmm3, xmm0	cmp rax, rcx
vpsrlw xmm4, xmm4, 3	jl loop
vpaddw xmm6, xmm4, xmm1	
vpcmpgtw xmm7, xmm6, xmm2	
vpand xmm6, xmm6, xmm7	
vpand xmm8, xmm3, xmm7	
vpor xmm6, xmm6, xmm5	
vpsllw xmm8, xmm8, 13	
vpunpcklwd xmm3, xmm8, xmm6	
vpunpckhwd xmm4, xmm8, xmm6	
vmovdqu [rbx], xmm3	
vmovdqu [rbx+16], xmm4	
add rbx, 32	
cmp rax, rcx	
jl loop	

15.14.3 Locality Consideration for using Half-Precision FP to Conserve Bandwidth

Example 15-32 and Example 15-33 demonstrate the performance advantage of using FP16C instructions when software needs to convert between half-precision and single-precision data. Half-precision FP format is more compact, consumes less bandwidth than single-precision FP format, but sacrifices dynamic range, precision, and incurs conversion overhead if arithmetic computation is required. Whether it is profitable for software to use half-precision data will be highly dependent on locality considerations of the workload.

This section uses an example based on the horizontal median filtering algorithm, "Median3". The Median3 algorithm calculates the median of every three consecutive elements in a vector:

Y[i] = Median3(X[i], X[i+1], X[i+2])

Where: Y is the output vector, and X is the input vector.

Example 15-34 shows two implementations of the Median3 algorithm; one uses single-precision format without conversion, the other uses half-precision format and requires conversion. Alternative 1 on the left works with single precision format using 256-bit load/store operations, each of which loads/stores eight 32-bit numbers. Alternative 2 uses 128-bit load/store operations to load/store eight 16-bit numbers in half precision format and VCVTPH2PS/VCVTPS2PH instructions to convert it to/from single precision floating-point format.

Example 15-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision

Single-Precision code w/o Conversion	Half-Precision code w/ Conversion
xor rbx, rbx	xor rbx, rbx
mov rcx, len	mov rcx, len
mov rdi, inPtr	mov rdi, inPtr
mov rsi, outPtr	mov rsi, outPtr
vmovaps ymm0, [rdi]	vcvtph2ps ymm0, [rdi]
loop:	loop:
add rdi, 32	add rdi,16
vmovaps ymm6, [rdi]	vcvtph2ps ymm6, [rdi]
vperm2f128 ymm1, ymm0, ymm6, 0x21	vperm2f128 ymm1, ymm0, ymm6, 0x21
vshufps ymm3, ymm0, ymm1, 0x4E	vshufps ymm3, ymm0, ymm1, 0x4E
vshufps ymm2, ymm0, ymm3, 0x99	vshufps ymm2, ymm0, ymm3, 0x99
vminps ymm5, ymm0, ymm2	vminps ymm5, ymm0, ymm2
vmaxps ymm0, ymm0, ymm2	vmaxps ymm0, ymm0, ymm2
vminps ymm4, ymm0, ymm3	vminps ymm4, ymm0, ymm3
vmaxps ymm7, ymm4, ymm5	vmaxps ymm7, ymm4, ymm5
vmovaps ymm0, ymm6	vmovaps ymm0, ymm6
vmovaps [rsi], ymm7	vcvtps2ph [rsi], ymm7, roundingCtrl
add rsi, 32	add rsi, 16
add rbx, 8	add rbx, 8
cmp rbx, rcx	cmp rbx, rcx
jl loop	jl loop

When the locality of the working set resides in memory, using half-precision format with processors based on Ivy Bridge microarchitecture is about 30% faster than single-precision format, despite the conversion overhead. When the locality resides in L3, using half-precision format is still ~15% faster. When the locality resides in L1, using single-precision format is faster because the cache bandwidth of the L1 data cache is much higher than the rest of the cache/memory hierarchy and the overhead of the conversion becomes a performance consideration.

15.15 FUSED MULTIPLY-ADD (FMA) INSTRUCTIONS GUIDELINES

FMA instructions perform vectored operations of "a * b + c" on IEEE-754-2008 floating-point values, where the multiplication operations "a * b" are performed with infinite precision, the final results of the addition are rounded to produced the desired precision. Details of FMA rounding behavior and special case handling can be found in section 2.3 of Intel® Architecture Instruction Set Extensions Programming Reference.

FMA instruction can speed up and improve the accuracy of many FP calculations. Haswell microarchitecture implements FMA instructions with execution units on port 0 and port 1 and 256-bit data paths. Dot product, matrix multiplication and polynomial evaluations are expected to benefit from the use of FMA, 256-bit data path and the independent executions on two ports. The peak throughput of FMA from each processor core are 16 single-precision and 8 double-precision results each cycle.

Algorithms designed to use FMA instruction should take into consideration that non-FMA sequence of MULPD/PS and ADDPD/PS likely will produce slightly different results compared to using FMA. For numerical computations involving a convergence criteria, the difference in the precision of intermediate results must be factored into the numeric formalism to avoid surprise in completion time due to rounding issues.

User/Source Coding Rule 28. *Factor in precision and rounding characteristics of FMA instructions when replacing multiply/add operations executing non-FMA instructions.* FMA improves performance when an algorithm is execution-port throughput limited, like DGEMM.

There may be situations where using FMA might not deliver better performance. Consider the vectored operation of "a * b + c * d" and data are ready at the same time:

In the three-instruction sequence of

```
VADDPS ( VMULPS (a,b) , VMULPS (c,b) );
```

VMULPS can be dispatched in the same cycle and execute in parallel, leaving the latency of VADDPS (3 cycle) exposed. With unrolling the exposure of VADDPS latency may be further amortized.

When using the two-instruction sequence of

```
VFMADD213PS ( c, d, VMULPS (a,b) );
```

The latency of FMA (5 cycle) is exposed for producing each vector result.

User/Source Coding Rule 29. *Factor in result-dependency, latency of FP add vs. FMA instructions when replacing FP add operations with FMA instructions.*

15.15.1 Optimizing Throughput with FMA and Floating-Point Add/MUL

In the Skylake microarchitecture, there are two pipes of executions supporting FMA, vector FP Multiply, and FP ADD instructions. All three categories of instructions have a latency of 4 cycles and can dispatch to either port 0 or port 1 to execute every cycle.

The arrangement of identical latency and number of pipes allows software to increase the performance of situations where floating-point calculations are limited by the floating-point add operations that follow FP multiplies. Consider a situation of vector operation $A_n = C_1 + C_2 * A_{n-1}$:

Example 15-35. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence
mov eax, NumOfIterations	mov eax, NumOfIterations
mov rbx, pA	mov rbx, pA
mov rcx, pC1	mov rcx, pC1
mov rdx, pC2	mov rdx, pC2
vmovups ymm0, ymmword ptr [rbx] // A	vmovups ymm0, ymmword ptr [rbx] // A
vmovups ymm1, ymmword ptr [rcx] // C1	vmovups ymm1, ymmword ptr [rcx] // C1
vmovups ymm2, ymmword ptr [rdx] // C2	vmovups ymm2, ymmword ptr [rdx] // C2
loop:	loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
vaddps ymm0, ymm1, ymm4	dec eax
dec eax	jnz loop
jnz loop	
	vmovups ymmword ptr[rbx], ymm0 // store A
vmovups ymmword ptr[rbx], ymm0 // store A	

Example 15-35. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence
Cost per iteration: ~ fp add latency + fp add latency	Cost per iteration: ~ fma latency

The overall throughput of the code sequence on the LHS is limited by the combined latency of the FP MUL and FP ADD instructions of specific microarchitecture. The overall throughput of the code sequence on the RHS is limited by the throughput of the FMA instruction of the corresponding microarchitecture.

A common situation where the latency of the FP ADD operation dominates performance is the following C code:

```
for ( int 1 = 0; i < arrLenght; i ++) result += arrToSum[i];
```

Example 15-35 shows two implementations with and without unrolling.

Example 15-36. Unrolling to Hide Dependent FP Add Latency

No Unroll	Unroll 8 times

```
    mov eax, arrLength
    mov rbx, arrToSum
    vmovups  ymm0, ymmword ptr [rbx]
    sub eax, 8
loop:
    add rbx, 32
    vaddps ymm0, ymm0, ymmword ptr [rbx]
    sub eax, 8
    jnz loop


    vextractf128 xmm1, ymm0, 1
    vaddps xmm0, xmm0, xmm1
    vpermilps xmm1, xmm0, 0xe
    vaddps xmm0, xmm0, xmm1
    vpermilps xmm1, xmm0, 0x1
    vaddss xmm0, xmm0, xmm1
```
	```
	    mov eax, arrLength
	    mov rbx, arrToSum
	    vmovups  ymm0, ymmword ptr [rbx]
	    vmovups  ymm1, ymmword ptr 32[rbx]
	    vmovups  ymm2, ymmword ptr 64[rbx]
	    vmovups  ymm3, ymmword ptr 96[rbx]
	    vmovups  ymm4, ymmword ptr 128[rbx]
	    vmovups  ymm5, ymmword ptr 160[rbx]
	    vmovups  ymm6, ymmword ptr 192[rbx]
	    vmovups  ymm7, ymmword ptr 224[rbx]

	    sub eax, 64
	loop:
	    add rbx, 256
	    vaddps ymm0, ymm0, ymmword ptr [rbx]
	    vaddps ymm1, ymm1, ymmword ptr 32[rbx]
	    vaddps ymm2, ymm2, ymmword ptr 64[rbx]
	    vaddps ymm3, ymm3, ymmword ptr 96[rbx]
	    vaddps ymm4, ymm4, ymmword ptr 128[rbx]
	    vaddps ymm5, ymm5, ymmword ptr 160[rbx]
	    vaddps ymm6, ymm6, ymmword ptr 192[rbx]
	    vaddps ymm7, ymm7, ymmword ptr 224[rbx]
	    sub eax, 64
	    jnz loop

	    vaddps ymm0, ymm0, ymm1
	    vaddps ymm2, ymm2, ymm3
	    vaddps ymm4, ymm4, ymm5
	    vaddps ymm6, ymm6, ymm7
	    vaddps ymm0, ymm0, ymm2
	    vaddps ymm4, ymm4, ymm6
	    vaddps ymm0, ymm0, ymm4
	```

Example 15-36. Unrolling to Hide Dependent FP Add Latency (Contd.)

No Unroll	Unroll 8 times
movss result, xmm0	vextractf128 xmm1, ymm0, 1 vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0xe vaddps xmm0, xmm0, xmm1 vpermilps xmm1, xmm0, 0x1 vaddss xmm0, xmm0, xmm1 movss result, xmm0

Without unrolling (LHS of Example 15-35), the cost of summing every 8 array elements is about proportional to the latency of the FP ADD instruction, assuming the working set fit in L1. To use unrolling effectively, the number of unrolled operations should be at least "latency of the critical operation" * "number of pipes". The performance gain of optimized unrolling versus no unrolling, for a given microarchitecture, can approach "number of pipes" * "Latency of FP ADD".

User/Source Coding Rule 30. *Consider using unrolling technique for loops containing back-to-back dependent FMA, FP Add or Vector MUL operations, The unrolling factor can be chosen by considering the latency of the critical instruction of the dependency chain and the number of pipes available to execute that instruction.*

15.15.2 Optimizing Throughput with Vector Shifts

In the Skylake microarchitecture, many common vector shift instructions can dispatch into either port 0 or port 1, compared to only one port in prior generations, see Table 2-12 and Table E-2.

A common situation where the latency of the FP ADD operation dominates performance is the following C code, where a, b, and c are integer arrays:

```
for ( int 1 = 0; i < len; i ++) c[i] += 4* a[i] + b[i]/2;
```

Example 15-35 shows two implementations with and without unrolling.

Example 15-37. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence
mov eax, NumOfIterations mov rbx, pA mov rcx, pC1 mov rdx, pC2 vmovups ymm0, ymmword ptr [rbx] // A vmovups ymm1, ymmword ptr [rcx] // C1 vmovups ymm2, ymmword ptr [rdx] // C2 loop: vmulps ymm4, ymm0 ,ymm2 // A * C2 vaddps ymm0, ymm1, ymm4 dec eax jnz loop vmovups ymmword ptr[rbx], ymm0 // store An	mov eax, NumOfIterations mov rbx, pA mov rcx, pC1 mov rdx, pC2 vmovups ymm0, ymmword ptr [rbx] // A vmovups ymm1, ymmword ptr [rcx] // C1 vmovups ymm2, ymmword ptr [rdx] // C2 loop: vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2 dec eax jnz loop vmovups ymmword ptr[rbx], ymm0 // store An
Cost per iteration: ~ fp add latency + fp add latency	Cost per iteration: ~ fma latency

15.16 AVX2 OPTIMIZATION GUIDELINES

AVX2 instructions promotes the great majority of 128-bit SIMD integer instructions to operate on 256-bit YMM registers. AVX2 also adds a rich mix of broadcast/permute/variable-shift instructions to accelerate numerical computations. The 256-bit AVX2 instructions are supported by Haswell microarchitecture, which implements 256-bit data path with low latency and high throughput.

Consider an intra-coding 4x4 block image transformation[1] shown in Figure 15-3.

A 128-bit SIMD implementation can perform this transformation by the following technique:

- Convert 8-bit pixels into 16-bit word elements and fetch two 4x4 image block as 4 row vectors.
- The matrix operation 1/128 * (B x R) can be evaluated with row vectors of the image block and column vectors of the right-hand-side coefficient matrix using a sequence of SIMD instructions of PMADDWD, packed shift and blend instructions.
- The two 4x4 word-granular, intermediate result can be re-arranged into column vectors.
- The left-hand-side coefficient matrix in row vectors and the column vectors of the intermediate block can be calculated (using PMADDWD, shift, blend) and written out.

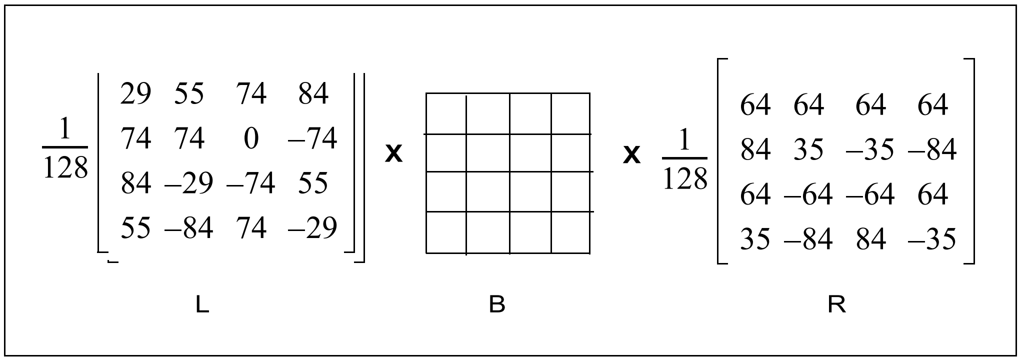


Figure 15-3. 4x4 Image Block Transformation

The same technique can be implemented using AVX2 instructions in a straightforward manner. The AVX2 sequence is illustrated in Example 15-38 and Example 15-39.

Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2

```
// b0: input row vector from 4 consecutive 4x4 image block of word pixels
// rmc0-3: columnar vector coefficient of the RHS matrix, repeated 4X for 256-bit
// min32km1: saturation constant vector to cap intermediate pixel to less than or equal to 32767
// w0: output row vector of garbled intermediate matrix, elements within each block are garbled
// e.g Low 128-bit of row 0 in descending order: y07, y05, y06, y04, y03, y01, y02, y00

#define __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0_256,
rmc1_256, rmc2_256, rmc3_256, min32km1)\
```

1. C. Yeo, Y. H. Tan, Z. Li and S. Rahardja, “Mode-Dependent Fast Separable KLT for Block-based Intra Coding,” JCTVC-B024, Geneva, Switzerland, Jul 2010

Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
  {__m256i  tt0, tt1, tt2, tt3, tttmp;\
  tt0 = _mm256_madd_epi16(b0, (rmc0_256));\
  tt1 = _mm256_madd_epi16(b0, rmc1_256);\
  tt1 = _mm256_hadd_epi32(tt0, tt1);\
  tttmp = _mm256_srai_epi32( tt1, 31);\
  tttmp = _mm256_srli_epi32( tttmp, 25);\
  tt1 = _mm256_add_epi32( tt1, tttmp);\
  tt1 = _mm256_min_epi32(_mm256_srai_epi32( tt1, 7), min32km1);\
  tt1 = _mm256_shuffle_epi32(tt1, 0xd8); \
  tt2 = _mm256_madd_epi16(b0, rmc2_256);\
  tt3 = _mm256_madd_epi16(b0, rmc3_256);\
  tt3 = _mm256_hadd_epi32(tt2, tt3);\
  tttmp = _mm256_srai_epi32( tt3, 31);\
  tttmp = _mm256_srli_epi32( tttmp, 25);\
  tt3 = _mm256_add_epi32( tt3, tttmp);\
  tt3 = _mm256_min_epi32( _mm256_srai_epi32(tt3, 7), min32km1);\
  tt3 = _mm256_shuffle_epi32(tt3, 0xd8);\
  w0 = _mm256_blend_epi16(tt1, _mm256_slli_si256( tt3, 2),  0xaa);\
}
// t0-t3: 256-bit input vectors of un-garbled intermediate matrix 1/128 * (B x R)
// lmr_256: 256-bit vector of one row of LHS coefficient, repeated 4X
// min32km1: saturation constant vector to cap final pixel to less than or equal to 32767
// w0; Output row vector of final result in un-garbled order
#define __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr_256, min32km1)\
   {__m256i tb0, tb1, tb2, tb3, tbtmp;
   tb0 = _mm256_madd_epi16( lmr_256, t0);\
   tb1 = _mm256_madd_epi16( lmr_256, t1);\
   tb1 = _mm256_hadd_epi32(tb0, tb1);\
   tbtmp = _mm256_srai_epi32( tb1, 31);\
   tbtmp = _mm256_srli_epi32( tbtmp, 25);\
   tb1 = _mm256_add_epi32( tb1, tbtmp);\
   tb1 = _mm256_min_epi32( _mm256_srai_epi32( tb1, 7), min32km1);\
   tb1 = _mm256_shuffle_epi32(tb1, 0xd8);\
   tb2 = _mm256_madd_epi16( lmr_256, t2);\
   tb3 = _mm256_madd_epi16( lmr_256, t3);\
   tb3 = _mm256_hadd_epi32(tb2, tb3);\
   tbtmp = _mm256_srai_epi32( tb3, 31);\
   tbtmp = _mm256_srli_epi32( tbtmp, 25);\
   tb3 = _mm256_add_epi32( tb3, tbtmp);\
   tb3 = _mm256_min_epi32( _mm256_srai_epi32( tb3, 7), min32km1);\
   tb3 = _mm256_shuffle_epi32(tb3, 0xd8); \
   tb3 = _mm256_slli_si256( tb3, 2);\
   tb3 = _mm256_blend_epi16(tb1, tb3,  0xaa);\
```

Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
    w0 = _mm256_shuffle_epi8(tb3, _mm256_setr_epi32( 0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a,
0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a));\
}
```

In Example 15-39, matrix multiplication of 1/128 * (B xR) is evaluated first in a 4-wide manner by fetching from 4 consecutive 4x4 image block of word pixels. The first macro shown in Example 15-38 produces an output vector where each intermediate row result is in an garbled sequence between the two middle elements of each 4x4 block. In Example 15-39, undoing the garbled elements and transposing the intermediate row vector into column vectors are implemented using blend primitives instead of shuffle/unpack primitives.

In Haswell microarchitecture, shuffle/pack/unpack primitives rely on the shuffle execution unit dispatched to port 5. In some situations of heavy SIMD sequences, port 5 pressure may become a determining factor in performance.

If 128-bit SIMD code faces port 5 pressure when running on Haswell microarchitecture, porting 128-bit code to use 256-bit AVX2 can improve performance and alleviate port 5 pressure.

Example 15-39. Separable KLT Intra-block Transformation Using AVX2

```
short __declspec(align(16))cst_rmc0[8] = {64, 84, 64, 35, 64, 84, 64, 35};
short __declspec(align(16))cst_rmc1[8] = {64, 35, -64, -84, 64, 35, -64, -84};
short __declspec(align(16))cst_rmc2[8] = {64, -35, -64, 84, 64, -35, -64, 84};
short __declspec(align(16))cst_rmc3[8] = {64, -84, 64, -35, 64, -84, 64, -35};
short __declspec(align(16))cst_lmr0[8] = {29, 55, 74, 84, 29, 55, 74, 84};
short __declspec(align(16))cst_lmr1[8] = {74, 74, 0, -74, 74, 74, 0, -74};
short __declspec(align(16))cst_lmr2[8] = {84, -29, -74, 55, 84, -29, -74, 55};
short __declspec(align(16)) cst_lmr3[8] = {55, -84, 74, -29, 55, -84, 74, -29};


void Klt_256_d(short * Input, short * Output, int iWidth, int iHeight)
{int iX, iY;
__m256i rmc0 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *) &cst_rmc0[0]));
__m256i rmc1 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc1[0]));
__m256i rmc2 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc2[0]));
__m256i rmc3 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc3[0]));
__m256i lmr0 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr0[0]));
__m256i lmr1 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr1[0]));
__m256i lmr2 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr2[0]));
__m256i lmr3 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr3[0]));
__m256i min32km1 = _mm256_broadcastd_epi32(_mm_cvtsi32_si128( _mm_setr_epi32( 0x7fff7fff, 0x7fff7fff,
0x7fff7fff, 0x7fff7fff));
__m256i b0, b1, b2, b3, t0, t1, t2, t3;
__m256i w0, w1, w2, w3;
short* pImage = Input;
short* pOutImage = Output;
int hgt = iHeight, wid= iWidth;


            (continue)
```

Example 15-39. Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
// We implement 1/128 * (Mat_L x (1/128 * (Mat_B x Mat_R))) from the inner most parenthesis
for( iY = 0; iY < hgt; iY+=4) {
  for( iX = 0; iX < wid; iX+=16) {
      //load row 0 of 4 consecutive 4x4 matrix of word pixels
      b0 = _mm256_loadu_si256( (__m256i *) (pImage + iY*wid+ iX)) ;
      // multiply row 0 with columnar vectors of the RHS matrix coefficients
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0, rmc1, rmc2, rmc3, min32km1);
       // low 128-bit of garbled row 0, from hi->lo: y07, y05, y06, y04, y03, y01, y02, y00
      b1 = _mm256_loadu_si256( (__m256i *)  (pImage + (iY+1)*wid+ iX) );
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b1, w1, rmc0, rmc1, rmc2, rmc3, min32km1);
       // hi->lo y17, y15, y16, y14, y13, y11, y12, y10
      b2 = _mm256_loadu_si256( (__m256i *)  (pImage + (iY+2)*wid+ iX) );
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b2, w2, rmc0, rmc1, rmc2, rmc3, min32km1);
      b3 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+3)*wid+ iX) );
      __MyM_KIP_PxRMC_ROW_4x4Wx4(b3, w3, rmc0, rmc1, rmc2, rmc3, min32km1);


      // unscramble garbled middle 2 elements of each 4x4 block, then
      // transpose into columnar vectors: t0 has 4 consecutive column 0 or 4 4x4 intermediate
      t0 = _mm256_blend_epi16( w0, _mm256_slli_epi64(w1, 16), 0x22);
      t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w2, 32), 0x44);
      t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w3, 48), 0x88);
      t1 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 32), _mm256_srli_epi64(w1, 16), 0x22);
      t1 = _mm256_blend_epi16( t1, w2, 0x44);
      t1 = _mm256_blend_epi16( t1, _mm256_slli_epi64(w3, 16), 0x88); // column 1
      t2 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 16), w1, 0x22);
      t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w2, 16), 0x44);
      t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w3, 32), 0x88); // column 2
      t3 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 48), _mm256_srli_epi64(w1, 32), 0x22);
      t3 = _mm256_blend_epi16( t3, _mm256_srli_epi64(w2, 16), 0x44);
      t3 = _mm256_blend_epi16( t3, w3, 0x88);// column 3


      // multiply row 0 of the LHS coefficient with 4 columnar vectors of intermediate blocks
      // final output row are arranged in normal order
      __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr0, min32km1);
      _mm256_store_si256( (__m256i *) (pOutImage+iY*wid+ iX), w0) ;

      __MyM_KIP_LMRxP_ROW_4x4Wx4(w1, t0, t1, t2, t3, lmr1, min32km1);
      _mm256_store_si256( (__m256i *) (pOutImage+(iY+1)*wid+ iX), w1) ;

      __MyM_KIP_LMRxP_ROW_4x4Wx4(w2, t0, t1, t2, t3, lmr2, min32km1);
      _mm256_store_si256( (__m256i *) (pOutImage+(iY+2)*wid+ iX), w2) ;
```

Example 15-39. Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
        __MyM_KIP_LMRxP_ROW_4x4Wx4(w3, t0, t1, t2, t3, lmr3, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+3)*wid+ iX), w3) ;
    }
 }
 }
```

Although 128-bit SIMD implementation is not shown here, it can be easily derived.

When running 128-bit SIMD code of this KLT intra-coding transformation on Sandy Bridge microarchitecture, the port 5 pressure are less because there are two shuffle units, and the effective throughput for each 4x4 image block transformation is around 50 cycles. Its speed-up relative to optimized scalar implementation is about 2.5X.

When the 128-bit SIMD code runs on Haswell microarchitecture, micro-ops issued to port 5 account for slightly less than 50% of all micro-ops, compared to about one third on prior microarchitecture, resulting in about 25% performance regression. On the other hand, AVX2 implementation can deliver effective throughput in less than 35 cycle per 4x4 block.

15.16.1 Multi-Buffering and AVX2

There are many compute-intensive algorithms (e.g. hashing, encryption, etc.) which operate on a stream of data buffers. Very often, the data stream may be partitioned and treated as multiple independent buffer streams to leverage SIMD instruction sets.

Detailed treatment of hashing several buffers in parallel can be found at http://www.scirp.org/journal/PaperInformation.aspx?paperID=23995 and at http://eprint.iacr.org/2012/476.pdf.

With AVX2 providing a full compliment of 256-bit SIMD instructions with rich functionality at multiple width granularities for logical and arithmetic operations. Algorithms that had leveraged XMM registers and prior generations of SSE instruction sets can extend those multi-buffering algorithms to use AVX2 on YMM and deliver even higher throughput. Optimized 256-bit AVX2 implementation may deliver up to 1.9X throughput when compared to 128-bit versions.

The image block transformation example discussed in Section 15.16 can be construed also as a multi-buffering implementation of 4x4 blocks. When the performance baseline is switched from a two-shuffle-port microarchitecture (Sandy Bridge) to single-shuffle-port microarchitecture, the 256-bit wide AVX2 provides a speed up of 1.9X relative to 128-bit SIMD implementation.

Greater details on multi-buffering can be found in the white paper at: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf.

15.16.2 Modular Multiplication and AVX2

Modular multiplication of very large integers are often used to implement efficient modular exponentiation operations which are critical in public key cryptography, such as RSA 2048. Library implementation of modular multiplication is often done with MUL/ADC chain sequences. Typically, a MUL instruction can produce a 128-bit intermediate integer output, and add-carry chains must be used at 64-bit intermediate data granularity.

In AVX2, VPMULUDQ/VPADDQ/VPSRLQ/VPSLLQ/VPBROADCASTQ/VPERMQ allow vectorized approach to implement efficient modular multiplication/exponentiation for key lengths corresponding to RSA1024 and RSA2048. For details of modular exponentiation/multiplication and AVX2 implementation in OpenSSL, see http://rd.springer.com/chapter/10.1007%2F978-3-642-31662-3_9?LI=true.

The basic heuristic starts with reformulating the large integer input operands in 512/1024 bit exponentiation in redundant representations. For example, a 1024-bit integer can be represented using base 2^29 and 36 "**digits**", where each "**digit**" is less than 2^29. A digit in such redundant representation can be placed in a dword slot of a vector register. Such redundant representation of large integer simplifies the requirement to perform carry-add chains across the hardware granularity of the intermediate results of unsigned integer multiplications.

Each VPMULUDQ in AVX2 using the **digits** from a redundant representation can produce 4 separate 64-bit intermediate result with sufficient headroom (e.g. 5 most significant bits are 0 excluding sign bit). Then, VPADDQ is sufficient to implement add-carry chain requirement without needing SIMD versions of equivalent of ADC-like instructions. More details are available in the reference cited in paragraph above, including the cost factor of conversion to redundant representation and effective speedup accounting for parallel output bandwidth of VPMULUDQ/VPADDQ chain.

15.16.3 Data Movement Considerations

Haswell microarchitecture can support up to two 256-bit loads and one 256-bit store micro-ops dispatched each cycle. Most existing binaries with heavy data-movement operation can benefit from this enhancement and the higher bandwidths of the L1 data cache and L2 without re-compilation, if the binary is already optimized for the prior generation microarchitecture. For example, 256-bit SAXPY computation was limited by the number of load/store ports available in the previous microarchitecture generation; it will benefit immediately on Haswell microarchitecture.

In some situations, there may be some intricate interactions between microarchitectural restrictions on the instruction set that is worth some discussion. We consider two commonly used library functions memcpy() and memset() and the optimal choice to implement them on the new microarchitecture.

With memcpy() on Haswell microarchitecture, using REP MOVSB to implement memcpy operation for large copy length can take advantage the 256-bit store data path and deliver throughput of more than 20 bytes per cycle. For copy length that are smaller than a few hundred bytes, REP MOVSB approach is slower than using 128-bit SIMD technique described in Section 15.16.3.1.

With memcpy() on Ice Lake microarchitecture, using in-lined REP MOVSB to implement memcpy is as fast as a 256-bit AVX implementation for copy lengths that are variable and unknown at compile time. For lengths that are known at compile time, REP MOVSB is almost as good as 256-bit AVX for short strings up to 128 bytes (9 cycles vs 3-7 cycles), and better for strings of 2K bytes and longer. For these cases we recommend using inline REP MOVSB. That said, software should still branch away for zero byte copies.

15.16.3.1 SIMD Heuristics to implement Memcpy()

We start with a discussion of the general heuristic to attempt implementing memcpy() with 128-bit SIMD instructions, which revolves around three numeric factors (destination address alignment, source address alignment, bytes to copy) relative to the width of register width of the desired instruction set. The data movement work of memcpy can be separated into the following phases:

- An initial unaligned copy of 16 bytes, allows looping destination address pointer to become 16-byte aligned. Thus subsequent store operations can use as many 16-byte aligned stores.
- The remaining bytes-left-to-copy are decomposed into (a) multiples of unrolled 16-byte copy operations, plus (b) residual count that may include some copy operations of less than 16 bytes. For example, to unroll eight time to amortize loop iteration overhead, the residual count must handle individual cases from 1 to 8x16-1 = 127.
- Inside an 8X16 unrolled main loop, each 16 byte copy operation may need to deal with source pointer address is not aligned to 16-byte boundary and store 16 fresh data to 16B-aligned destination address. When the iterating source pointer is not 16B-aligned, the most efficient technique is a three instruction sequence of:
  - Fetch an 16-byte chunk from an 16-byte-aligned adjusted pointer address and use a portion of this chunk with complementary portion from previous 16-byte-aligned fetch.
  - Use PALIGNR to stitch a portion of the current chunk with the previous chunk.

— Stored stitched 16-byte fresh data to aligned destination address, and repeat this 3 instruction sequence.

This 3-instruction technique allows the fetch:store instruction ratio for each 16-byte copy operation to remain at 1:1.

While the above technique (specifically, the main loop dealing with copying thousands of bytes of data) can achieve throughput of approximately 10 bytes per cycle on Sandy Bridge and Ivy Bridge microarchitectures with 128-bit data path for store operations, an attempt to extend this technique to use wider data path will run into the following restrictions:

- To use 256-bit VPALIGNR with its 2X128-bit lane microarchitecture, stitching of two partial chunks of the current 256-bit 32-byte-aligned fetch requires another 256-bit fetch from an address 16-byte offset from the current 32-byte-aligned 256-bit fetch.
  — The fetch:store ratio for each 32-byte copy operation becomes 2:1.
  — The 32-byte-unaligned fetch (although aligned to 16-byte boundary) will experience a cache-line split penalty, once every 64-bytes of copy operation.

The net of this attempt to use 256-bit ISA to take advantage of the 256-bit store data-path microarchitecture was offset by the 4-instruction sequence and cacheline split penalty.

15.16.3.2 Memcpy() Implementation Using Enhanced REP MOVSB

It is interesting to compare the alternate approach of using enhanced REP MOVSB to implement memcpy(). In Haswell and Ivy Bridge microarchitectures, REP MOVSB is an optimized, hardware provided, micro-op flow.

On Ivy Bridge microarchitecture, a REP MOVSB implementation of memcpy can achieve throughput at slightly better than the 128-bit SIMD implementation when copying thousands of bytes. However, if the size of copy operation is less than a few hundred bytes, the REP MOVSB approach is less efficient than the explicit residual copy technique described in phase 2 of Section 15.16.3.1. This is because handling 1-127 residual copy length (via jump table or switch/case, and is done before the main loop) plus one or two 8x16B iterations incurs less branching overhead than the hardware provided micro-op flows. For the grueling implementation details of 128-bit SIMD implementation of memcpy(), one can look up from the archived sources of open source library such as GLibC.

On Haswell microarchitecture, using REP MOVSB to implement memcpy operation for large copy length can take advantage the 256-bit store data path and deliver throughput of more than 20 bytes per cycle. For copy length that are smaller than a few hundred bytes, REP MOVSB approach is still slower than treating the copy length as the residual phase of Section 15.16.3.1.

15.16.3.3 Memset() Implementation Considerations

The interface of Memset() has one address pointer as destination, which simplifies the complexity of managing address alignment scenarios to use 256-bit aligned store instruction. After an initial unaligned store, and adjusting the destination pointer to be 32-byte aligned, the residual phase follows the same consideration as described in Section 15.16.3.1, which may employ a large jump table to handle each residual value scenario with minimal branching, depending on the amount of unrolled 32B-aligned stores. The main loop is a simple YMM register to 32-byte-aligned store operation, which can deliver close to 30 bytes per cycle for lengths more than a thousand byte. The limiting factor here is due to each 256-bit VMOVDQA store consists of a store_address and a store_data micro-op flow. Only port 4 is available to dispatch the store_data micro-op each cycle.

Using REP STOSB to implement memset() has the code size advantage versus a SIMD implementation, like REP MOVSB for memcpy(). On Haswell microarchitecture, a memset() routine implemented using REP STOSB will also benefit the from the 256-bit data path and increased L1 data cache bandwidth to deliver up to 32 bytes per cycle for large count values.

Comparing the performance of memset() implementations using REP STOSB vs. 256-bit AVX2 requires one to consider the pattern of invocation of memset(). The invocation pattern can lead to the necessity of using different performance measurement techniques. There may be side effects affecting the outcome of each measurement technique.

The most common measurement technique that is often used with a simple routine like memset() is to execute memset() inside a loop with a large iteration count, and wrap the invocation of RDTSC before and after the loop.

A slight variation of this measurement technique can apply to measuring memset() invocation patterns of multiple back-to-back calls to memset() with different count values with no other intervening instruction streams executed between calls to memset().

In both of the above memset() invocation scenarios, branch prediction can play a significant role in affecting the measured total cycles for executing the loop. Thus, measuring AVX2-implemented memset() under a large loop to minimize RDTSC overhead can produce a skewed result with the branch predictor being trained by the large loop iteration count.

In more realistic software stacks, the invocation patterns of memset() will likely have the characteristics that:

- There are intervening instruction streams being executed between invocations of memset(), the state of branch predictor prior to memset() invocation is not pre-trained for the branching sequence inside a memset() implementation.
- Memset() count values are likely to be uncorrected.

The proper measurement technique to compare memset() performance for more realistic memset() invocation scenarios will require a per-invocation technique that wraps two RDTSC around each invocation of memset().

With the per-invocation RDTSC measurement technique, the overhead of RDTSC and be pre-calibrated and post-validated outside of a measurement loop. The per-invocation technique may also consider cache warming effect by using a loop to wrap around the per-invocation measurements.

When the relevant skew factors of measurement techniques are taken into effect, the performance of memset() using REP STOSB, for count values smaller than a few hundred bytes, is generally faster than the AVX2 version for the common memset() invocation scenarios. Only in the extreme scenarios of hundreds of unrolled memset() calls, all using count values less than a few hundred bytes and with no intervening instruction stream between each pair of memset() can the AVX2 version of memset() take advantage of the training effect of the branch predictor.

15.16.3.4 Hoisting Memcpy/Memset Ahead of Consuming Code

There may be situations where the data furnished by a call to memcpy/memset and subsequent instructions consuming the data can be re-arranged:

```
memcpy ( pBuf, pSrc, Cnt); // make a copy of some data with knowledge of Cnt
..... // subsequent instruction sequences are not consuming pBuf immediately
result = compute( pBuf); // memcpy result consumed here
```

When the count is known to be at least a thousand byte or more, using enhanced REP MOVSB/STOSB can provide another advantage to amortize the cost of the non-consuming code. The heuristic can be understood using a value of Cnt = 4096 and memset() as example:

- A 256-bit SIMD implementation of memset() will need to issue/execute retire 128 instances of 32-byte store operation with VMOVDQA, before the non-consuming instruction sequences can make their way to retirement.
- An instance of enhanced REP STOSB with ECX= 4096 is decoded as a long micro-op flow provided by hardware, but retires as one instruction. There are many store_data operation that must complete before the result of memset() can be consumed. Because the completion of store data operation is de-coupled from program-order retirement, a substantial part of the non-consuming code stream can process through the issue/execute and retirement, essentially cost-free if the non-consuming sequence does not compete for store buffer resources.

Software that use enhanced REP MOVSB/STOSB must check its availability by verifying CPUID.(EAX=07H, ECX=0):EBX.[bit 9] reports 1.

15.16.3.5 256-bit Fetch versus Two 128-bit Fetches

On Sandy Bridge and Ivy Bridge microarchitectures, using two 16-byte aligned loads are preferred due to the 128-bit data path limitation in the memory pipeline of the microarchitecture.

To take advantage of Haswell microarchitecture's 256-bit data path microarchitecture, the use of 256-bit loads must consider the alignment implications. Instruction that fetched 256-bit data from memory should pay attention to be 32-byte aligned. If a 32-byte unaligned fetch would span across cache line boundary, it is still preferable to fetch data from two 16-byte aligned address instead.

15.16.3.6 Mixing MULX and AVX2 Instructions

Combining MULX and AVX2 instruction can further improve the performance of some common computation task, e.g. numeric conversion 64-bit integer to ascii format can benefit from the flexibility of MULX register allocation, wider YMM register, and variable packed shift primitive VPSRLVD for parallel moduli/remainder calculations.

Example 15-40 shows a macro sequence of AVX2 instruction to calculate one or two finite range unsigned short integer(s) into respective decimal digits, featuring VPSRLVD in conjunction with Montgomery reduction technique.

Example 15-40. Macros for Parallel Moduli/Remainder Calculation

```
static  short  quoTenThsn_mulplr_d[16] =
{  0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0,  0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static  short  mten_mulplr_d[16] = { -10, 1, -10,  1, -10,  1, -10, 1, -10, 1, -10,  1, -10,  1, -10, 1};


// macro to convert input t5 (a __m256i type) containing quotient (dword 4) and remainder
// (dword 0) into single-digit integer (between 0-9) in output y3 ( a__m256i);
//both dword element "t5" is assume to be less than 10^4, the rest of dword must be 0;
//the output is 8 single-digit integer, located in the low byte of each dword, MS digit in dword 0
#define    __ParMod10to4AVX2dw4_0( y3, t5 ) \
{ __m256i x0, x2;              \
   x0 = _mm256_shuffle_epi32( t5, 0);  \
   x2 = _mm256_mulhi_epu16(x0, _mm256_loadu_si256( (__m256i *)  quoTenThsn_mulplr_d));\
   x2 = _mm256_srlv_epi32(  x2, _mm256_setr_epi32(0x0, 0x4, 0x7, 0xa, 0x0, 0x4, 0x7, 0xa) ); \
   (y3) = _mm256_or_si256(_mm256_slli_si256(x2, 6),   _mm256_slli_si256(t5, 2) );  \
   (y3) = _mm256_or_si256(x2, y3);\
   (y3) = _mm256_madd_epi16(y3, _mm256_loadu_si256( (__m256i *) mten_mulplr_d) ) ;\
}
// parallel conversion of dword integer (< 10^4) to 4 single digit integer in __m128i
#define    __ParMod10to4AVX2dw( x3, dw32 ) \
{ __m128i x0, x2;              \
   x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128(  dw32));  \
   x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
   x2 = _mm_srlv_epi32(  x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) ); \
   (x3) = _mm_or_si128(_mm_slli_si128(x2, 6),   _mm_slli_si128(_mm_cvtsi32_si128(  dw32), 2) );  \
   (x3) = _mm_or_si128(x2, (x3));\
   (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}
```

Example 15-41 shows a helper utility and overall steps to reduce a 64-bit signed integer into a 63-bit unsigned range with reduced-range integer quotient/remainder pairs using MULX. Note that this example relies on Example 15-40 and Example 15-42.

Example 15-41. Signed 64-bit Integer Conversion Utility

```
#define  QWCG10to 80xabcc77118461cefdull

static  int   pr_cg_10to4[8] = { 0x68db8db, 0 , 0, 0, 0x68db8db, 0, 0, 0};
static  int   pr_1_m10to4[8] = { -10000, 0 , 0, 0 , 1, 0 , 0, 0};
                  (continue)
```

```
char * i64toa_avx2i( __int64 xx, char * p)
{int cnt;
  _mm256_zeroupper();
  if( xx < 0) cnt = avx2i_q2a_u63b(-xx, p);
  else   cnt = avx2i_q2a_u63b(xx, p);
  p[cnt] = 0;
  return p;
}

// Convert unsigned short (< 10^4) to ascii
__inline  int ubsAvx2_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
__m128i x0, m0, x2, x3, x4;
  if( x_Lt10k < 10) { *ps = '0' + x_Lt10k; return 1; }
  x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128(  x_Lt10k));
  // calculate quotients of divisors 10, 100, 1000, 10000
  m0 = _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d);
  x2 = _mm_mulhi_epu16(x0, m0);
  // u16/10, u16/100, u16/1000, u16/10000
  x2 = _mm_srlv_epi32(  x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) );
  // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
  x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
  x4 = _mm_or_si128(x2, x3);
  // produce 4 single digits in low byte of each dword
  x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;// add bias for ascii encoding
  x2 = _mm_add_epi32( x4, _mm_set1_epi32( 0x30303030 ) );
  // pack 4 single digit into a dword, start with most significant digit
  x3 = _mm_shuffle_epi8(x2, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
  if (x_Lt10k > 999 )    {*(int *) ps = _mm_cvtsi128_si32(  x3);  return 4;}
```

Example 15-41. Signed 64-bit Integer Conversion Utility (Contd.)

```
    tmp = _mm_cvtsi128_si32(  x3);
    if (x_Lt10k > 99 ) {
       *((short *) (ps)) = (short ) (tmp >>8);
       ps[2] = (char ) (tmp >>24);
       return 3;
    }


   *((short *) ps) = (short ) (tmp>>16); return 2;
   }


   }
```

Example 15-42 shows the steps of numeric conversion of a 63-bit dynamic range into ascii format according to a progressive range reduction technique using a vectorized Montgomery reduction scheme. Note that this example relies on Example 15-40.

Example 15-42. Unsigned 63-bit Integer Conversion Utility

```
unsigned   avx2i_q2a_u63b (unsigned __int64 xx, char *ps)
{ __m128i  v0;
  __m256i  m0, x1, x3, x4, x5 ;
  unsigned __int64 xxi, xx2, lo64, hi64;
__int64  w;
  int j, cnt, abv16, tmp, idx, u;
    // conversion of less than 4 digits
   if ( xx < 10000 ) {
       j = ubsAvx2_Lt10k_2s_i2 ( (unsigned ) xx, ps); return j;
   } else if (xx < 100000000 ) {  // dynamic range of xx is less than 9 digits
    // conversion of 5-8 digits
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)xx));  // broadcast to every dword
    // calculate quotient and remainder, each with reduced range (< 10^4)
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
   // quotient in dw4, remainder in dw0
     m0 = _mm256_add_epi32( _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)xx), 0),
x3);
     __ParMod10to4AVX2dw4_0( x3, m0); // 8 digit in low byte of each dw
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
     x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
           0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );


                      (continue)
```

Example 15-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
// pack 8 single-digit integer into first 8 bytes and set rest to zeros
 x4 = _mm256_permutevar8x32_epi32( x4, _mm256_setr_epi32(0x4, 0x0, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1)  );
 tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
 _BitScanForward((unsigned long *) &idx, tmp);
 cnt = 8 -idx; // actual number non-zero-leading digits to write to output
} else  {  // conversion of 9-12 digits
 lo64 = _mulx_u64(xx, (unsigned __int64) QWCG10to8, &hi64);
 hi64 >>= 26;

 xxi = _mulx_u64(hi64, (unsigned __int64)100000000, &xx2);
 lo64 = (unsigned __int64)xx - xxi;

 if( hi64 < 10000) { // do digist 12-9 first
    __ParMod10to4AVX2dw(v0, (int)hi64);
    v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
   // continue conversion of low 8 digits of a less-than 12-digit value
    x5 = _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)lo64), 0);
    x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)lo64));  // broadcast to every dword
    x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
    x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
    m0 = _mm256_add_epi32( x5, x3); // quotient in dw4, remainder in dw0
    __ParMod10to4AVX2dw4_0( x3, m0);
    x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
    x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
        0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
   x5 = _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_shuffle_epi8(v0,
        _mm_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080)), 0);
    x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x2, 0x4, 0x0, 0x1,
        0x1, 0x1, 0x1, 0x1) );
    tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
    _BitScanForward((unsigned long *) &idx, tmp);
    cnt = 12 -idx;
 } else { // handle greater than 12 digit input value
    cnt = 0;
    if ( hi64 >  100000000) { // case of input value has more than 16 digits
      xxi =  _mulx_u64(hi64, (unsigned __int64) QWCG10to8, &xx2) ;
      abv16 = (int)(xx2 >>26);
      hi64 -= _mulx_u64((unsigned __int64) abv16, (unsigned __int64) 100000000, &xx2);
      __ParMod10to4AVX2dw(v0,  abv16);
      v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
      v0 = _mm_shuffle_epi8(v0, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );


              (continue)
```

Example 15-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
            tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(v0, _mm_set1_epi32( 0x30303030 )) );
            _BitScanForward((unsigned long *) &idx, tmp);
            cnt = 4 -idx;
        }


       // conversion of lower 16 digits
        x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)hi64));  // broadcast to every dword
        x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
        x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
        m0 = _mm256_add_epi32(_mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)hi64),
0), x3);
        __ParMod10to4AVX2dw4_0( x3, m0);
        x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
        x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
             0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
        x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)lo64));  // broadcast to every dword
        x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
        x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
        m0 = _mm256_add_epi32(_mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)lo64),
0), ), x3);
        __ParMod10to4AVX2dw4_0( x3, m0);
        x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
        x5 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080,
             0x80808080, 0x80808080, 0x0004080c, 0x80808080) );
        x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x4, 0x0, 0x6, 0x2,
             0x1,  0x1, 0x1, 0x1)  );
        cnt += 16;
        if (cnt <=  16) {
            tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
            _BitScanForward((unsigned long *) &idx, tmp);
            cnt -= idx;
        }
    }
  }

   w = _mm_cvtsi128_si64( _mm256_castsi256_si128(x4));
   switch(cnt) {
   case 5:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
   break;
   case 6:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
   break;
   case 7:*ps = (char) (w >>8);   *(short *) (&ps[1]) = (short) (w >>16);
     *(unsigned *) (&ps[3]) = (w >>32);


                        (continue)
```

Example 15-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
  break;
  case 8: *(long long *)ps = w;
  break;
  case 9:*ps++ = (char) (w >>24);  *(long long *) (&ps[0]) =  _mm_cvtsi128_si64(
_mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 10:*(short *)ps = (short) (w >>16);
  *(long long *) (&ps[2])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
  *(long long *) (&ps[3])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 12: *(unsigned *)ps = (unsigned int) w; *(long long *) (&ps[4])  = _mm_cvtsi128_si64(
_mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 13:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
  *(long long *) (&ps[4]) =  _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
  break;
  case 14:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
  *(long long *) (&ps[6]) =  _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
  break;
  case 15:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
     *(unsigned *) (&ps[3]) = (w >>32);
  *(long long *) (&ps[7])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
  break;
  case 16: _mm_storeu_si128( (__m128i *) ps, _mm256_castsi256_si128(x4));
  break;

  case 17:u = (int) _mm_cvtsi128_si64(v0);  *ps++ = (char) (u >>24);
  _mm_storeu_si128( (__m128i *) &ps[0], _mm256_castsi256_si128(x4));
  break;
  case 18:u = (int) _mm_cvtsi128_si64(v0);  *(short *)ps = (short) (u >>16);
  _mm_storeu_si128( (__m128i *) &ps[2], _mm256_castsi256_si128(x4));
  break;
  case 19:u = (int) _mm_cvtsi128_si64(v0);  *ps = (char) (u >>8); *(short *) (&ps[1]) = (short) (u >>16);
  _mm_storeu_si128( (__m128i *) &ps[3], _mm256_castsi256_si128(x4));
  break;
  case 20:u = (int) _mm_cvtsi128_si64(v0);  *(unsigned *)ps = (short) (u);
  _mm_storeu_si128( (__m128i *) &ps[4], _mm256_castsi256_si128(x4));
  break;
  }

 return cnt;
}
```

The AVX2 version of numeric conversion across the dynamic range of 3/9/17 output digits are approximately 23/57/54 cycles per input, compared to standard library implement ion's range of 85/260/560 cycles per input.

The techniques illustrated above can be extended to numeric conversion of other library, such as binary-integer-decimal (BID) encoded IEEE-754-2008 Decimal floating-point format. For BID-128 format, Example 15-42 can be adapted by adding another range-reduction stage using a pre-computed 256-bit constant to perform Montgomery reduction at modulus 10^16. The technique to construct the 256-bit constant is covered in Chapter 14, "SSE4.2 and SIMD Programming For Text-Processing/Lexing/Parsing"of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

15.16.4 Considerations for Gather Instructions

VGATHER family of instructions fetch multiple data elements specified by a vector index register containing relative offsets from a base address. Processors based on Haswell microarchitecture is the first implementation of the VGATHER instruction and a single instruction results in multiple micro-ops being executed. In the Broadwell microarchitecture, the throughput of the VGATHER family of instructions have improved significantly; see Table D-5.

Depending on data organization and access patterns, it is possible to create equivalent code sequences without using VGATHER instruction that will execute faster and with fewer micro-ops than a single VGATHER instruction (e.g. see Section 15.5.1). Example 15-43 shows some of the situations where use of VGATHER on Haswell microarchitecture is unlikely to provide performance benefit.

Example 15-43. Access Patterns Favoring Non-VGATHER Techniques

Access Patterns	Recommended Instruction Selection
Sequential elements	Regular SIMD loads (MOVAPS/MOVUPS, MOVDQA/MOVDQU)
Fewer than 4 elements	Regular SIMD load + horizontal data-movement to re-arrange slots
Small Strides	Load all nearby elements + shuffle/permute to collected strided elements: `VMOVUPD YMM0, [sequential elements]` `VPERMQ YMM1, YMM0, 0x08 // the even elements` `VPERMQ YMM2, YMM0, 0x0d // the odd elements`
Transpositions	Regular SIMD loads + shuffle/permute/blend to transpose to columns
Redundant elements	Load once + shuffle/blend/logical to build data vectors in register. In this case, result[i] = x[index[i]] + x[index[i+1]], the technique below may be preferable to using multiple VGATHER: `ymm0 <- VGATHER ( x[index[k] ]); // fetching 8 elements` `ymm1 <- VBLEND( VPERM (ymm0), VBROADCAST ( x[indexx[k+8]]);` `ymm2 <- VPADD( ymm0, ymm1);`

In other cases, using the VGATHER instruction can reduce code size and execute faster with techniques including but not limited to amortizing the latency and throughput of VGATHER, or by hoisting the fetch operations well in advance of consumer code of the destination register of those fetches. Example 15-44 lists some patterns that can benefit from using VGATHER on Haswell microarchitecture.

General tips for using VGATHER:

- Gathering more elements with a VGATHER instruction helps amortize the latency and throughput of VGATHER, and is more likely to provide performance benefit over an equivalent non-VGATHER flow. For example, the latency of 256-bit VGATHER is less than twice the equivalent 128-bit VGATHER and therefore more likely to show gains than two 128-bit equivalent ones. Also, using index size larger than data element size results in only half of the register slots utilized but not a proportional latency reduction. Therefore the dword index form of VGATHER is preferred over qword index if dwords or single-precision values are to be fetched.

- It is advantageous to hoist VGATHER well in advance of the consumer code.
- VGATHER merges the (unmasked) gathered elements with the previous value of the destination. Therefore, in cases where the previous value of the destination doesn't need to be merged (for instance, when no elements is masked off), it can be beneficial to break the dependency of the VGATHER instruction on the previous writer of the destination register (by zeroing out the register with a VXOR instruction).

Example 15-44. Access Patterns Likely to Favor VGATHER Techniques

Access Patterns	Instruction Selection

4 or more elements with unknown masks	Code with conditional element gathers typically either will not vectorize without a VGATHER instruction or provide relatively poor performance due to data-dependent mis-predicted branches.
	C code with data-dependent branches:
	`if (condition[i] > 0) { result[i] = x[index[i]] }`
	AVX2 equivalent sequence:
	`YMM0 <- VPCMPGT (condition, zeros) // compute vector mask`
	`YMM2 <- VGATHER (x[YMM1], YMM0) // addr=x[YMM1], mask=YMM0`

Vectorized index calculation with 8 elements	Vectorized calculations to generate the index synergizes well with the VGATHER instruction functionality.
	C code snippet:
	`x[index1[i] + index2[i]]`
	AVX2 equivalent:
	`YMM0 <- VPADD (index1, index2)     // calc vector index`
	`YMM1 <- VGATHER (x[YMM0], mask)    // addr=x[YMM0]`

Performance of the VGATHER instruction compared to a multi-instruction gather equivalent flow can vary due to (1) differences in the base algorithm, (2) different data organization, and (3) the effectiveness of the equivalent flow. In performance critical applications it is advisable to evaluate both options before choosing one.

The throughput of GATHER instructions continue to improve from Broadwell to Skylake Microarchitecture. This is shown in Figure 15-4.

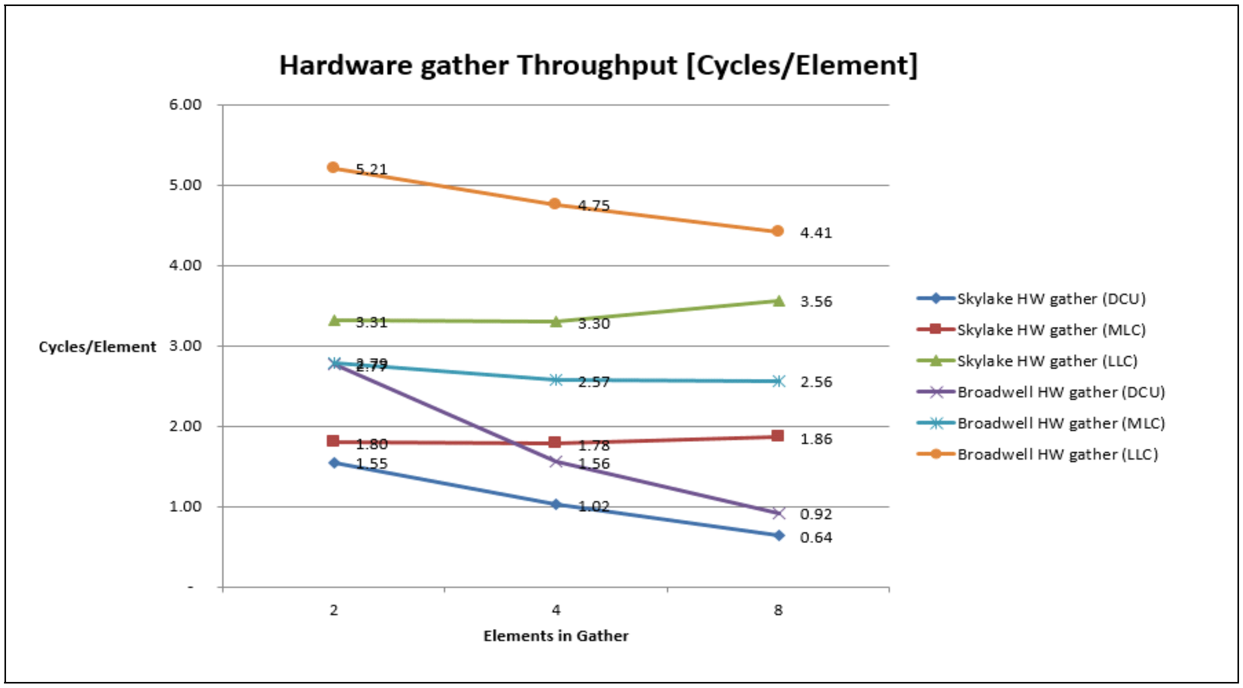


Figure 15-4. Throughput Comparison of Gather Instructions

Example 15-45 gives the asm sequence of software implementation that is equivalent to the VPGATHERD instruction. This can be used to compare the trade-off of using a hardware gather instruction or software gather sequence based on inserting an individual element.

Example 15-45. Software AVX Sequence Equivalent to Full-Mask VPGATHERD

```
mov eax, [rdi]                          // load index0
vmovd xmm0, [rsi+4*rax]                 // load element0
mov eax, [rdi+4]                        // load index1
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x1    // load element1
mov eax, [rdi+8]                        // load index2
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x2    // load element2
mov eax, [rdi+12]                       // load index3
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x3    // load element3
mov eax, [rdi+16]                       // load index4
vmovd xmm1, [rsi+4*rax]                 // load element4
mov eax, [rdi+20]                       // load index5
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x1    // load element5
mov eax, [rdi+24]                       // load index6
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x2    // load element6
mov eax, [rdi+28]                       // load index7
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x3    // load element7
vinserti128 ymm0, ymm0, xmm1, 1         //result in ymm0
```

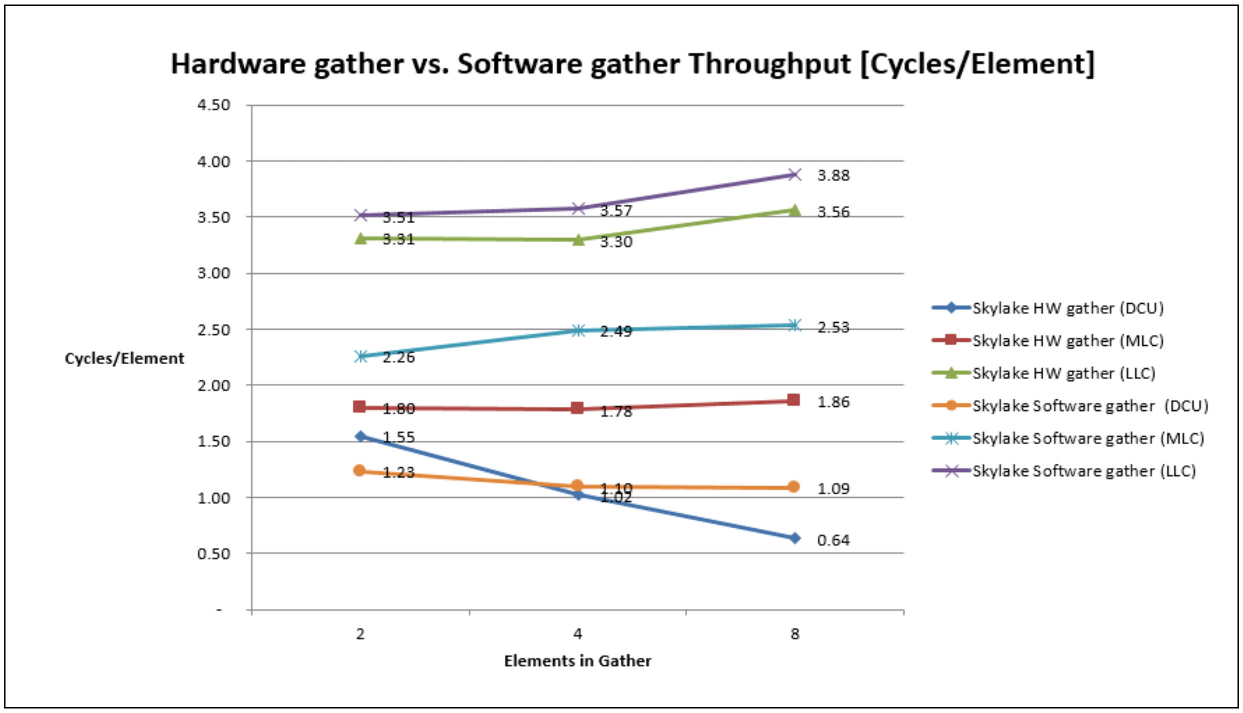


Figure 15-5. Comparison of HW GATHER Versus Software Sequence in Skylake Microarchitecture

Figure 15-5 compares per-element throughput using the VPGATHERD instruction versus a software gather sequence with Skylake microarchitecture as a function of cache locality of data supply. With the exception of using hardware GATHER on two data elements per instruction, the gather instruction out-performs the software sequence on Skylake microarchitecture.

If data supply locality is from memory, software sequences are likely to perform better than the hardware GATHER instruction.

15.16.4.1 Strided Loads

This section compares using the hardware GATHER instruction versus alternative implementations of handling Array of Structures (AOS) to Structure of Arrays (SOA) transformation. The code separates the real and imaginary elements in a complex array into two separate arrays.

C code:

```
for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[i].real;
    Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

Example 15-46. AOS to SOA Transformation Alternatives

1: Scalar Code	2: AVX w/ VINSRT+VSHUFPS	3: AVX2 w/ VPGATHERD
loop: lea eax, [r10+r10*1] movsxd rax, eax inc r10d mov r11d, dword ptr [rsi+rax*8] mov dword ptr [rcx+rax*4], r11d mov r11d, dword ptr [rsi+rax*8+0x4] mov dword ptr [rdx+rax*4], r11d mov r11d, dword ptr [rsi+rax*8+0x8] mov dword ptr [rcx+rax*4+0x4], r11d mov r11d, dword ptr [rsi+rax*8+0xc] mov dword ptr [rdx+rax*4+0x4], r11d cmp r10d, r8d jl loop	loop: vmovdqu xmm0, xmmword ptr [r10+rcx*8] vmovdqu xmm1, xmmword ptr [r10+rcx*8+0x10] vmovdqu xmm4, xmmword ptr [r10+rcx*8+0x40] vmovdqu xmm5, xmmword ptr [r10+rcx*8+0x50] vinserti128 ymm2, ymm0, xmmword ptr [r10+rcx*8+0x20], 0x1 vinserti128 ymm3, ymm1, xmmword ptr [r10+rcx*8+0x30], 0x1 vinserti128 ymm6, ymm4, xmmword ptr [r10+rcx*8+0x60], 0x1 vinserti128 ymm7, ymm5, xmmword ptr [r10+rcx*8+0x70], 0x1 add rcx, 0x10 vshufps ymm0, ymm2, ymm3, 0x88 vshufps ymm1, ymm2, ymm3, 0xdd vshufps ymm4, ymm6, ymm7, 0x88 vshufps ymm5, ymm6, ymm7, 0xdd vmovups ymmword ptr [r9], ymm0 vmovups ymmword ptr [r8], ymm1 vmovups ymmword ptr [r9+0x20], ymm4 vmovups ymmword ptr [r8+0x20], ymm5 add r9, 0x40 add r8, 0x40 cmp rcx, rsi jl loop	loop: lea r11, [r10+rcx*8] vpxor ymm5, ymm5, ymm5 add rcx, 0x8 vpxor ymm6, ymm6, ymm6 vmovdqa ymm3, ymm0 vmovdqa ymm4, ymm0 vpgatherdd ymm5, ymmword ptr [r11+ymm2*4], ymm3 vpgatherdd ymm6, ymmword ptr [r11+ymm1*4], ymm4 vmovdqu ymmword ptr [r9], ymm5 vmovdqu ymmword ptr [r8], ymm6 add r9, 0x20 add r8, 0x20 cmp rcx, rsi jl loop

With strided access patterns, an AVX software sequence can load and shuffle on multiple elements and is the more optimal technique.

Table 15-7. Comparison of AOS to SOA with Strided Access Pattern

Microarchitecture	Scalar	VPGATHERD	AVX VINSRTF128/VSHUFFLEPS
Broadwell	1X	1.7X	4.8X
Skylake	1X	2.7X	4.9X

15.16.4.2 Adjacent Loads

This section compares using the hardware GATHER instruction versus alternative implementations of handling a variant situation of AOS to SOA transformation. In this case, AOS data are not loaded sequentially but via an index array.

C code:

```
for(int i=0;i<len;i++){
    Real_buffer[i] = Complex_buffer[Index_buffer[i]].real;
    Imaginary_buffer[i] = Complex_buffer[Index_buffer[i]].imag;
}
```

Example 15-47. Non-Strided AOS to SOA

AVX2 GATHERPD	AVX VINSRTF128 /UNPACK
loop: vmovdqu ymm1, ymmword ptr [rsi+rdx*4] vpaddd ymm3, ymm1, ymm1 vpaddd ymm14, ymm13, ymm3 vxorpd ymm5, ymm5, ymm5 vmovdqa ymm2, ymm0 vxorpd ymm6, ymm6, ymm6 vmovdqa ymm4, ymm0 vxorpd ymm10, ymm10, ymm10 vmovdqa ymm7, ymm0 vxorpd ymm11, ymm11, ymm11 vmovdqa ymm9, ymm0 vextracti128 xmm12, ymm14, 0x1 vextracti128 xmm8, ymm3, 0x1 vgatherdpd ymm6, ymmword ptr[r8+xmm8*8],ymm4 vgatherdpd ymm5, ymmword ptr[r8+xmm3*8],ymm2 vmovupd ymmword ptr [rcx+rdx*8], ymm5 vmovupd ymmword ptr [rcx+rdx*8+0x20], ymm6 vgatherdpd ymm11, ymmword ptr[r8+xmm12*8],ymm7 vgatherdpd ymm10, ymmword ptr[r8+xmm14*8],ymm9 vmovupd ymmword ptr [rax+rdx*8], ymm10 vmovupd ymmword ptr [rax+rdx*8+0x20], ymm11 add rdx, 0x8 cmp rdx, r11 jb loop	loop: movsxd r10, dword ptr [rdx+rsi*4] shl r10, 0x4 movsxd r11, dword ptr [rdx+rsi*4+0x8] shl r11, 0x4 vmovupd xmm0, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0x4] shl r10, 0x4 vinsertf128 ymm2, ymm0, xmmword ptr [r9+r11*1], 0x1 vmovupd xmm1, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0xc] shl r10, 0x4 vinsertf128 ymm3, ymm1, xmmword ptr [r9+r10*1], 0x1 movsxd r10, dword ptr [rdx+rsi*4+0x10] shl r10, 0x4 vunpcklpd ymm4, ymm2, ymm3 vunpckhpd ymm5, ymm2, ymm3 vmovupd ymmword ptr [rcx], ymm4 vmovupd xmm6, xmmword ptr [r9+r10*1] vmovupd ymmword ptr [rax], ymm5 movsxd r10, dword ptr [rdx+rsi*4+0x18] shl r10, 0x4 vinsertf128 ymm8, ymm6, xmmword ptr [r9+r10*1], 0x1 movsxd r10, dword ptr [rdx+rsi*4+0x14] shl r10, 0x4 vmovupd xmm7, xmmword ptr [r9+r10*1] movsxd r10, dword ptr [rdx+rsi*4+0x1c] add rsi, 0x8 shl r10, 0x4 vinsertf128 ymm9, ymm7, xmmword ptr [r9+r10*1], 0x1 vunpcklpd ymm10, ymm8, ymm9 vunpckhpd ymm11, ymm8, ymm9 vmovupd ymmword ptr [rcx+0x20], ymm10 add rcx, 0x40 vmovupd ymmword ptr [rax+0x20], ymm11 add rax, 0x40 cmp rsi, r8 jl loop

With non-strided, regular access pattern of AOS to SOA, an AVX software sequence that uses VINSERTF128 and interleaved packing of multiple elements can be more optimal.

Table 15-8. Comparison of Indexed AOS to SOA Transformation

Microarchitecture	VPGATHERPD	AVX VINSRTF128/VUNPCK*
Broadwell	1X	1.4X
Skylake	1.3X	1.7X

15.16.5 AVX2 Conversion Remedy to MMX Instruction Throughput Limitation

In processors based on the Skylake microarchitecture, the functionality of the MMX instruction set is unchanged from prior generations. But many MMX instructions are constrained to execute to one port with half the instruction throughput relative to prior microarchitectures. The MMX instructions with throughput constraints include:

- PADDS[B/W], PADDUS[B/W], PSUBS[B/W], PSUBUS[B/W].
- PCMPGT[B/W/D], PCMPEQ[B/W/D].
- PMAX[UB/SW], PMIN[UB/SW].
- PAVG[B/W], PABS[B/W/D], PSIGN[B/W/D].

To overcome the reduction of MMX instruction throughput, conversion of asm and intrinsic code to use AVX2 instruction will provide significant performance improvements. Example 15-48 shows the asm sequence using AVX2 versus MMX equivalent. In Skylake microarchitecture, the MMX code shown in Example 15-48 will execute at approximately half the speed relative to the Broadwell microarchitecture. This is due to PMAXSW/PMINSW throughput being reduced by half with the single-port restriction. When the same task is implemented with the equivalent AVX2 sequence, the performance of the AVX2 code on Skylake microarchitecture will be ~3.9X of the MMX code executing on the Broadwell microarchitecture.

Example 15-48. Conversion to Throughput-Reduced MMX sequence to AVX2 Alternative

MMX Code	AVX2 Code
mov rax, pIn	mov rax, pIn
mov rbx, pOut	mov rbx, pOut
mov r8, len	mov r8, len
mov rcx, 8	mov rcx, 32
movq mm0, [rax]	vmovdqu ymm0, ymmword ptr [rax]
movq mm1, [rax + 8]	vmovdqu ymm1, ymmword ptr [rax + 32]
movq mm2, mm0	vmovdqu ymm2, ymm0
movq mm3, mm1	vmovdqu ymm3, ymm1
cmp rcx, r8	cmp rcx, r8
jge end	jge end
loop:	loop:
movq mm4, [rax + 2*rcx]	vmovdqu ymm4, ymmword ptr [rax + 2*rcx]
movq mm5, [rax + 2*rcx + 8]	vmovdqu ymm5, ymmword ptr [rax + 2*rcx + 32]
	vpmaxsw ymm0, ymm0, ymm4
pmaxsw mm0, mm4	vpmaxsw ymm1, ymm1, ymm5
pmaxsw mm1, mm5	vpminsw ymm2, ymm2, ymm4
pminsw mm2, mm4	vpminsw ymm3, ymm3, ymm5
pminsw mm3, mm5	add rcx, 32
add rcx, 8	cmp rcx, r8
cmp rcx, r8	jl loop
jl loop	end:
	//Reduction
end:	vpmaxsw ymm0, ymm0, ymm1
//Reduction	vextracti128 xmm1, ymm0, 1
pmaxsw mm0, mm1	vpmaxsw xmm0, xmm0, xmm1
pshufw mm1, mm0, 0xE	vpshufd xmm1, xmm0, 0xe
pmaxsw mm0, mm1	vpmaxsw xmm0, xmm0, xmm1
pshufw mm1, mm0, 1	vpshuflw xmm1, xmm0, 0xe
pmaxsw mm0, mm1	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 1
pminsw mm2, mm3	vpmaxsw xmm0, xmm0, xmm1
pshufw mm3, mm2, 0xE	vmovd eax, xmm0
pminsw mm2, mm3	mov word ptr [rbx], ax
pshufw mm3, mm2, 1	vpminsw ymm2, ymm2, ymm3
pminsw mm2, mm3	vextracti128 xmm1, ymm2, 1
	vpminsw xmm2, xmm2, xmm1
movd eax, mm0	vpshufd xmm1, xmm2, 0xe
mov WORD PTR [rbx], ax	vpminsw xmm2, xmm2, xmm1
movd eax, mm2	vpshuflw xmm1, xmm2, 0xe
mov WORD PTR [rbx + 2], ax	vpminsw xmm2, xmm2, xmm1
emms	vpshuflw xmm1, xmm2, 1
	vpminsw xmm2, xmm2, xmm1
	vmovd eax, xmm2
	mov word ptr [rbx + 2], ax

8. Updates to Chapter 18

Change bars and violet text show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual:* Software Optimization for Intel AVX-512 Instructions.

Changes to this chapter:

- Example 18-1: Corrected typos: Teta with theta.
- Example 18-2: Corrected typos: Teta with theta.

CHAPTER 18
SOFTWARE OPTIMIZATION FOR INTEL® AVX-512 INSTRUCTIONS

Intel® Advanced Vector Extensions 512 (Intel® AVX-512) are the following set of 512-bit instruction set extensions supported by recent microarchitectures, beginning with Skylake server microarchitecture, and the Intel® Xeon Phi™ processors based on Knights Landing microarchitecture.

- Intel® AVX-512 Foundation (F)
  - 512-bit vector width.
  - 32 512-bit long vector registers.
  - Data expand and data compress instructions.
  - Ternary logic instruction.
  - 8 new 64-bit long mask registers.
  - Two source cross-lane permute instructions.
  - Scatter instructions.
  - Embedded broadcast/rounding.
  - Transcendental support.
- Intel® AVX-512 Conflict Detection Instructions (CD)
- Intel® AVX-512 Exponential and Reciprocal Instructions (ER)
- Intel® AVX-512 Prefetch Instructions (PF)
- Intel® AVX-512 Byte and Word Instructions (BW)
- Intel® AVX-512 Double Word and Quad Word Instructions (DQ)
  - New QWORD and Compute and Convert Instructions.
- Intel® AVX-512 Vector Length Extensions (VL)

The Venn diagram below shows the different extensions supported by the two processor families.

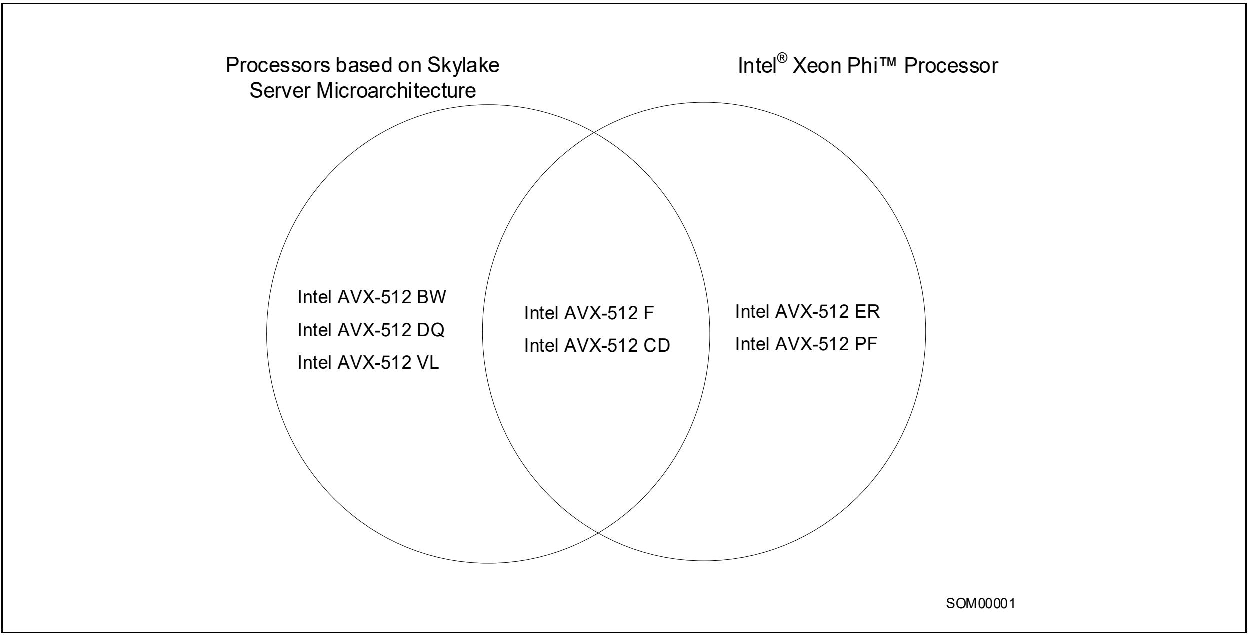


Figure 18-1. Intel® AVX-512 Extensions Supported by Skylake Server Microarchitecture and Knights Landing Microarchitecture

Performance reports in this chapter are based on Data Cache Unit (DCU) resident data measurements on the Skylake Server System with Intel® Turbo-Boost technology disabled, Intel® SpeedStep® Technology disabled, core and uncore frequency set to 1.8GHz, unless otherwise specified. This fixed frequency configuration is used in order to isolate code change impacts from other factors. See Section 2.5.3, "Skylake Server Power Management", to understand the power and frequency impacts of using Intel AVX-512.

18.1 BASIC INTEL® AVX-512 VS. INTEL® AVX2 CODING

In most cases, the main performance driver for Intel AVX-512 will be the 512-bit register width. This section demonstrates the similarity and differences between basic Intel AVX2 and Intel AVX-512 code and explains how to convert code from Intel AVX2 to Intel AVX-512 easily. The first sub section demonstrates the conversion of intrinsic code and the second sub-section of assembly code. The following sections highlight advanced aspects that require consideration and treatment when doing such conversions.

The examples in the following subsections implement a Cartesian coordinate system rotation. A point in a Cartesian coordinate system is described by the pair (x,y). The following picture demonstrates a Cartesian rotation of (x,y) by angle θ to (x',y').

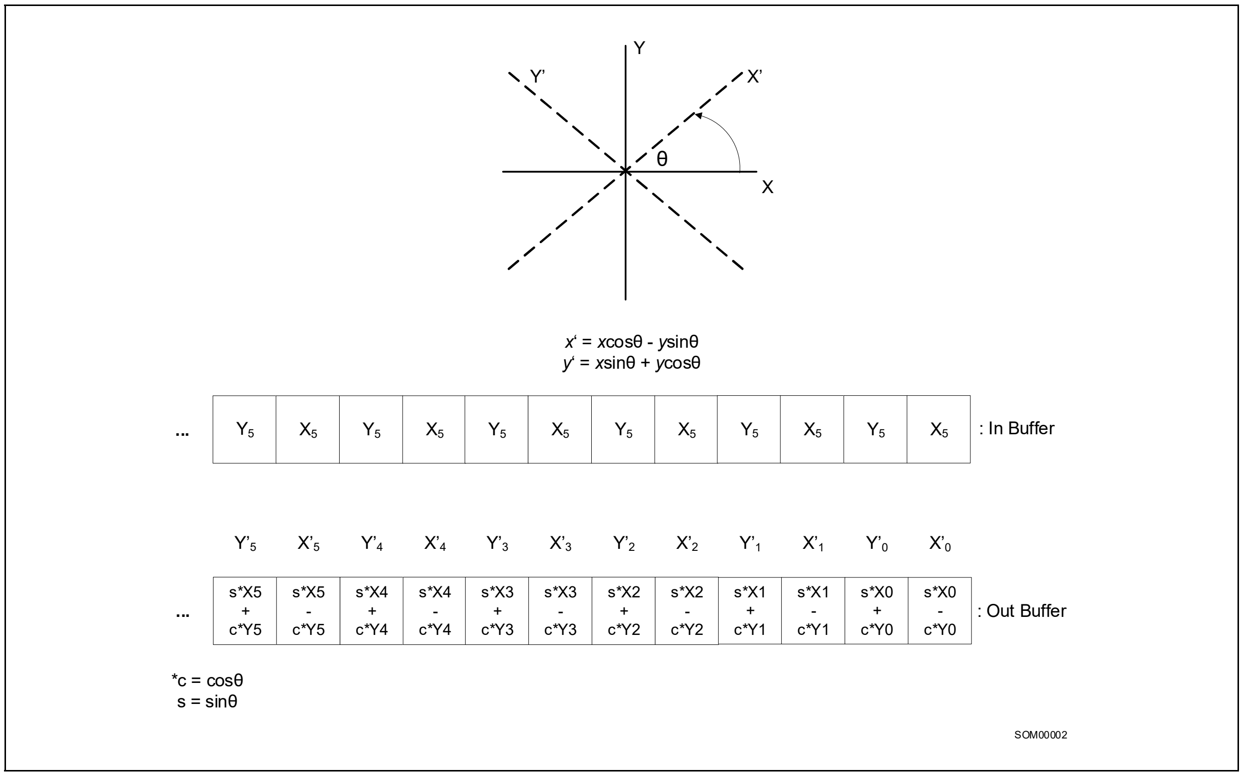

Y
Y'
X'
θ
X
x' = xcosθ - ysinθ
y' = xsinθ + ycosθ
... Y5 X5 Y5 X5 Y5 X5 Y5 X5 Y5 X5 Y5 X5 : In Buffer
Y'5 X'5 Y'4 X'4 Y'3 X'3 Y'2 X'2 Y'1 X'1 Y'0 X'0
... s*X5 + c*Y5 | s*X5 - c*Y5 | s*X4 + c*Y4 | s*X4 - c*Y4 | s*X3 + c*Y3 | s*X3 - c*Y3 | s*X2 + c*Y2 | s*X2 - c*Y2 | s*X1 + c*Y1 | s*X1 - c*Y1 | s*X0 + c*Y0 | s*X0 - c*Y0 : Out Buffer
*c = cosθ
s = sinθ
SOM00002

Figure 18-2. Cartesian Rotation

18.1.1 Intrinsic Coding

The following comparison of Intel AVX2 and Intel AVX-512 shows how to convert a simple intrinsic Intel AVX2 code sequence to Intel AVX-512. This example demonstrates the Intel AVX Instruction format, 64 byte ZMM registers, dynamic and static memory allocation with data alignment of 64bytes, and the C data type representing 16 floating point elements in a ZMM register. Follow these guidelines when doing this transformation.

- Align statically and dynamically allocated buffers to 64-bytes.
- Use a double supplemental buffer size for constants.
- Change __mm256_ intrinsic name prefix with __mm512_.
- Change variable data types names from __m256 to __m512.
- Divide by 2 iteration count (double stride length).

Example 18-1. Cartesian Coordinate System Rotation with Intrinsics

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code
#include <immintrin.h>	#include <immintrin.h>
int main()	int main()
{	{
int len = 3200;	int len = 3200;
//Dynamic memory allocation with 32byte	//Dynamic memory allocation with 64byte
//alignment	//alignment
float* pInVector = (float *)	float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);	_mm_malloc(len*sizeof(float),64);
float* pOutVector = (float *)	float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);	_mm_malloc(len*sizeof(float),64);
//init data	//init data
for (int i=0; i<len; i++)	for (int i=0; i<len; i++)
pInVector[i] = 1;	pInVector[i] = 1;
float cos_theta = 0.8660254037;	float cos_theta = 0.8660254037;
float sin_theta = 0.5;	float sin_theta = 0.5;
//Static memory allocation of 8 floats with 32byte align-ments	//Static memory allocation of 16 floats with 64byte align-ments
__declspec(align(32)) float cos_sin_theta_vec[8] =	__declspec(align(64)) float cos_sin_theta_vec[16] =
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,	{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
sin_theta, cos_theta, sin_theta};	sin_theta, cos_theta, sin_theta cos_theta, sin_theta,
	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta};
__declspec(align(32)) float sin_cos_theta_vec[8] =	__declspec(align(64)) float sin_cos_theta_vec[16] =
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,	{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
cos_theta, sin_theta, cos_theta};	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta};
//__m256 data type represents a Ymm	//__m512 data type represents a Zmm
// register with 8 float elements	// register with 16 float elements
__m256 Ymm_cos_sin = _mm256_-load_ps(cos_sin_theta_vec);	__m512 Zmm_cos_sin = _mm512_-load_ps(cos_sin_theta_vec);

Example 18-1. Cartesian Coordinate System Rotation with Intrinsics (Contd.)

//Intel® AVX2 256bit packed single load
__m256 Ymm_sin_cos = _mm256_-
load_ps(sin_cos_theta_vec);
 	
__m256 Ymm0, Ymm1, Ymm2, Ymm3;
//processing 16 elements in an unrolled
//twice loop
for(int i=0; i<len; i+=16)
{
Ymm0 = _mm256_load_ps(pInVector+i);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector + i,Ymm3);
 	
Ymm0 = _mm256_load_ps(pInVector+i+8);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2, Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector+i+8,Ymm3);
}
 	
_mm_free(pInVector);
_mm_free(pOutVector);
 	
return 0;
}
	//Intel® AVX-512 512bit packed single load
	__m512 Zmm_sin_cos = _mm512_-
	load_ps(sin_cos_theta_vec);
	__m512 Zmm0, Zmm1, Zmm2, Zmm3;
	//processing 32 elements in an unrolled
	//twice loop
	for(int i=0; i<len; i+=32)
	{
	Zmm0 = _mm512_load_ps(pInVector+i);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2,Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector + i,Zmm3);
	
	Zmm0 = _mm512_load_ps(pInVector+i+16);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2, Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector+i+16,Zmm3);
	}
	_mm_free(pInVector);
	_mm_free(pOutVector);
	
	return 0;
	}

Baseline	Speedup: 1.95x

18.1.2 Assembly Coding

Similar to the intrinsic porting guidelines, assembly porting guidelines are listed below:

- Align statically and dynamically allocated buffers to 64-bytes.
- Double the supplemental buffer sizes if needed.
- Add a “v” prefix to instruction names.
- Change register names from ymm to zmm.
- Divide the iteration count by two (or double stride length).

Example 18-2. Cartesian Coordinate System Rotation with Assembly

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code
#include <immintrin.h>	#include <immintrin.h>
int main()	int main()
{	{
int len = 3200;	int len = 3200;
//Dynamic memory allocation with 32byte alignment	//Dynamic memory allocation with 64byte alignment
float* pInVector = (float *)	float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);	_mm_malloc(len*sizeof(float),64);
float* pOutVector = (float *)	float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);	_mm_malloc(len*sizeof(float),64);
//init data	//init data
for (int i=0; i<len; i++)	for (int i=0; i<len; i++)
pInVector[i] = 1;	pInVector[i] = 1;
float cos_theta = 0.8660254037;	float cos_theta = 0.8660254037;
float sin_theta = 0.5;	float sin_theta = 0.5;
//Static memory allocation of 8 floats with 32byte align-	//Static memory allocation of 16 floats with 64byte align-
ments	ments
__declspec(align(32)) float cos_sin_theta_vec[8] =	__declspec(align(64)) float cos_sin_theta_vec[16] =
{cos_theta, sin_theta,	{cos_theta,
cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
sin_theta};	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta};
	__declspec(align(64)) float sin_cos_theta_vec[16] =
__declspec(align(32)) float sin_cos_theta_vec[8] =	{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,	cos_theta, sin_theta cos_theta, sin_theta, cos_theta,
cos_theta, sin_theta, cos_theta};	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta};
__asm	__asm
{	{
mov rax,pInVector	mov rax,pInVector
mov r8,pOutVector	mov r8,pOutVector
// Load into a ymm register of 32 bytes	// Load into a zmm register of 64 bytes
vmovups ymm3, ymmword ptr[cos_sin_theta_vec]	vmovups zmm3, zmmword ptr[cos_sin_theta_vec]
vmovups ymm4, ymmword ptr[sin_cos_theta_vec]	vmovups zmm4, zmmword ptr[sin_cos_theta_vec]
mov edx, len	mov edx, len
shl edx, 2	shl edx, 2
xor ecx, ecx	xor ecx, ecx
loop1:	loop1:
vmovsldup ymm0, [rax+rcx]	vmovsldup zmm0, [rax+rcx]
vmovshdup ymm1, [rax+rcx]	vmovshdup zmm1, [rax+rcx]
vmulps ymm1, ymm1, ymm4	vmulps zmm1, zmm1, zmm4
vfmaddsub213ps ymm0, ymm3, ymm1	vfmaddsub213ps zmm0, zmm3, zmm1
// 32 byte store from a ymm register	// 64 byte store from a zmm register
vmovaps [r8+rcx], ymm0	vmovaps [r8+rcx], zmm0

Example 18-2. Cartesian Coordinate System Rotation with Assembly (Contd.)

vmovsldup ymm0, [rax+rcx+32]
 vmovshdup ymm1, [rax+rcx+32]
 vmulps ymm1, ymm1, ymm4
 vfmaddsub213ps ymm0, ymm3, ymm1
 // offset 32 bytes from previous store
 vmovaps [r8+rcx+32], ymm0
 	
 // Processed 64bytes in this loop
 // (the code is unrolled twice)
 add ecx, 64
 cmp ecx, edx
 jl loop1
 }
 	
 _mm_free(pInVector);
 _mm_free(pOutVector);
 	
 return 0;
}
	vmovsldup zmm0, [rax+rcx+64]
	 vmovshdup zmm1, [rax+rcx+64]
	 vmulps zmm1, zmm1, zmm4
	 vfmaddsub213ps zmm0, zmm3, zmm1
	 // offset 64 bytes from previous store
	 vmovaps [r8+rcx+64], zmm0
	
	 // Processed 128bytes in this loop
	 // (the code is unrolled twice)
	 add ecx, 128
	 cmp ecx, edx
	 jl loop1
	 }
	
	 _mm_free(pInVector);
	 _mm_free(pOutVector);
	
	 return 0;
	}

Baseline	Speedup: 1.95x

18.2 MASKING

Intel AVX-512 instructions which use the Extended VEX coding scheme (EVEX) encode a predicate operand to conditionally control per-element computational operation and update the result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers, 64 bits each. From this set of 8 architectural registers, only k1 through k7 can be addressed as the predicate operand; k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

A predicate operand can be used to enable memory fault-suppression for some instructions with a memory source operand.

As a predicate operand, the opmask registers contain one bit to govern the operation / update of each data element of a vector register. Masking is supported on Skylake microarchitecture for instructions with all data sizes: byte (int8), word (int16), single precision floating-point (float32), integer doubleword (int32), double precision floating-point (float64), integer quadword (int64). Therefore, a vector register holds either 8, 16, 32 or 64 elements; accordingly, the length of a vector mask register is 64 bits. Masking on Skylake microarchitecture is also enabled for all vector length values: 128-bit, 256-bit and 512-bit. Each instruction accesses only the number of least significant mask bits needed based on its data type and vector length. For example, Intel AVX-512 instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 (i.e., 512/64) least significant bits of the opmask register.

An opmask register affects an Intel AVX-512 instruction at per-element granularity. So, any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in Intel AVX-512 has the following properties:

- The instruction's operation is only performed for an element if the corresponding opmask bit is set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value may be preserved (merging-masking) or zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a powerful construct to implement control-flow predication, since the mask provides a merging behavior for Intel AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior removes the implicit dependency on the old value when it is not needed.

Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory, only accept merging-masking.

The per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- Generated as a result of a vector instruction (CMP, FPCLASS, etc.).
- Loaded from memory.
- Loaded from GPR register.
- Modified by mask-to-mask operations.

18.2.1 Masking Example

The masked instructions conditionally operate with packed data elements, depending on the mask bits associated with each data element. The mask bit for each data element is the corresponding bit in the mask register.

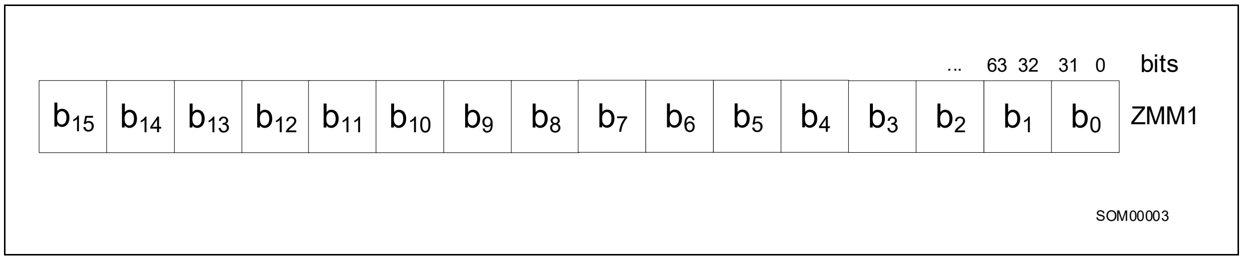
When performing a mask instruction, the returned value is 0 for elements which have a corresponding mask value of 0. The corresponding value in the destination register depends on the zeroing flag:

- If the flag is set, the memory location is filled with zeros.
- If the flag is not set, the values in memory location can are preserved.

The following figures show an example for a mask move from one register to another when using merging masking.

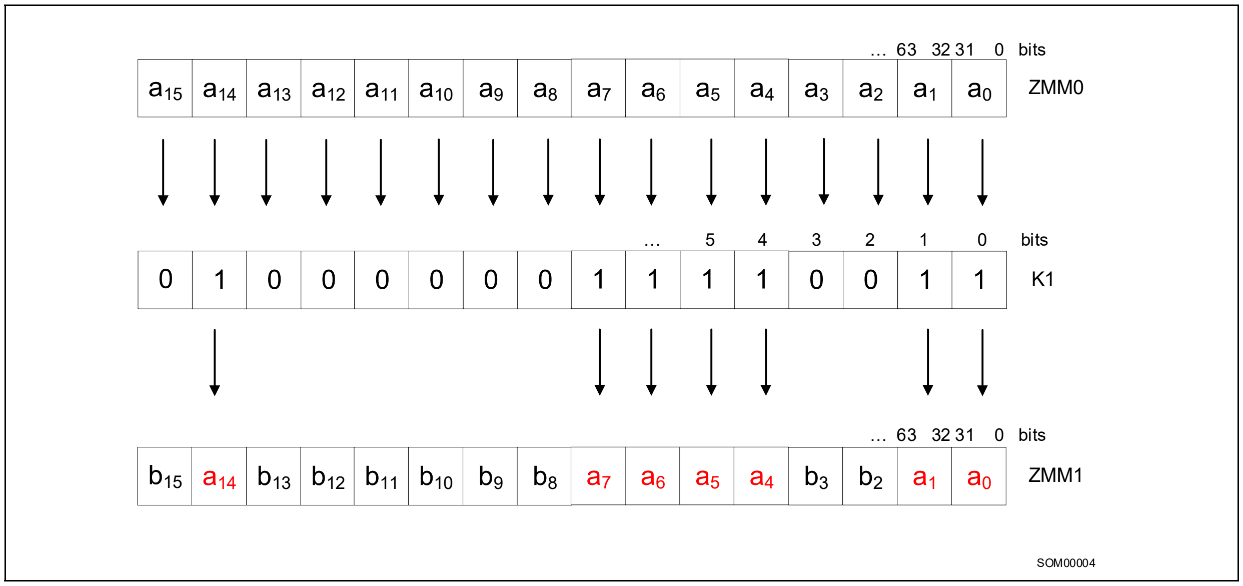
```
vmovaps  zmm1 {k1},  zmm0
```

The destination register before instruction execution is shown below.



... 63 32 31 0 bits
b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
ZMM1
SOM00003

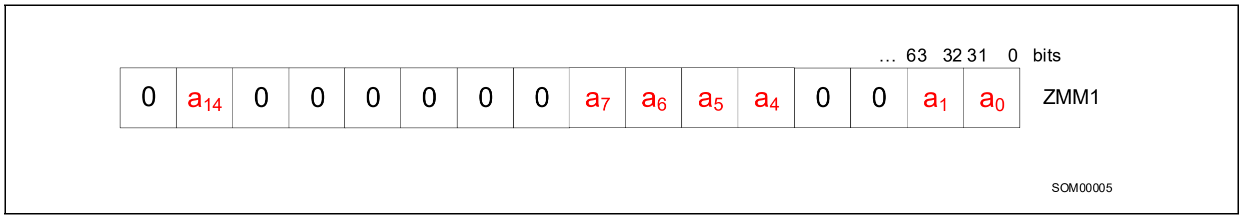
Operation is as follows.


… 63 32 31 0 bits
a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
ZMM0
… 5 4 3 2 1 0 bits
0 1 0 0 0 0 0 0 1 1 1 1 0 0 1 1
K1
… 63 32 31 0 bits
b_{15} a_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 a_7 a_6 a_5 a_4 b_3 b_2 a_1 a_0
ZMM1
SOM00004

The result of the execution with zeroing masking is (notice the {z} in the instruction):

```
vmovaps zmm1 {k1}{z}, zmm0
```

.


… 63 32 31 0 bits
0 a_{14} 0 0 0 0 0 0 a_7 a_6 a_5 a_4 0 0 a_1 a_0
ZMM1
SOM00005

Notice that merging masking operations has a dependency on the destination, but zeroing masking is free of such dependency.

The following example shows how masking could be done with Intel AVX-512 in contrast to Intel AVX2.

C Code:

```
const int N = miBufferWidth;
const double* restrict a = A;
const double* restrict b = B;
double* restrict c = Cref;

for (int i = 0; i < N; i++){
      double res = b[i];
      if(a[i] > 1.0){
              res = res * a[i];
     }
      c[i] = res;
}
```

Example 18-3. Masking with Intrinsics

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code

for (int i = 0; i < N; i+=32){
__m256d aa, bb, mask;
#pragma unroll(8)
for (int j = 0; j < 8; j++){
aa = _mm256_loadu_pd(a+i+j*4);
bb = _mm256_loadu_pd(b+i+j*4);
mask = _mm256_c-
mp_pd(_mm256_set1_pd(1.0), aa, 1);
aa = _mm256_and_pd(aa, mask); // zero the
false values
aa = _mm256_mul_pd(aa, bb);
bb = _mm256_blendv_pd(bb, aa, mask);
_mm256_storeu_pd(c+4*j, bb);
}
 	
c += 32;
}
	for (int i = 0; i < N; i+=32){
	__m512d aa, bb;
	__mmask8 mask;
	#pragma unroll(4)
	for (int j = 0; j < 4; j++){
	aa = _mm512_loadu_pd(a+i+j*8);
	bb = _mm512_loadu_pd(b+i+j*8);
	mask = _mm512_cmp_p-
	d_mask(_mm512_set1_pd(1.0), aa, 1);
	bb = _mm512_mask_mul_pd(bb, mask, aa,
	bb);
	_mm512_storeu_pd(c+8*j, bb);
	}
	
	c += 32;
	}

Baseline	Speedup: 2.9x

Example 18-4. Masking with Assembly

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code
mov rax, a mov r11, b mov r8, N shr r8, 5 mov rsi, c xor rcx, rcx xor r9, r9 loop: vmovupd ymm1, ymmword ptr [rax+rcx*8] inc r9d vmovupd ymm6, ymmword ptr [rax+rcx*8+0x20] vmovupd ymm2, ymmword ptr [r11+rcx*8] vmovupd ymm7, ymmword ptr [r11+rcx*8+0x20] vmovupd ymm11, ymmword ptr [rax+rcx*8+0x40] vmovupd ymm12, ymmword ptr [r11+rcx*8+0x40] vcmppd ymm4, ymm0, ymm1, 0x1 vcmppd ymm9, ymm0, ymm6, 0x1 vcmppd ymm14, ymm0, ymm11, 0x1 vandpd ymm16, ymm1, ymm4 vandpd ymm17, ymm6, ymm9	mov rax, a mov r11, b mov r8, N shr r8, 5 mov rsi, c xor rcx, rcx xor r9, r9 mov rdi, 1 cvtsi2sd xmm8, rdi vbroadcastsd zmm8, xmm8 loop: vmovups zmm0, zmmword ptr [rax+rcx*8] inc r9d vmovups zmm2, zmmword ptr [rax+rcx*8+0x40] vmovups zmm4, zmmword ptr [rax+rcx*8+0x80] vmovups zmm6, zmmword ptr [rax+rcx*8+0xc0] vmovups zmm1, zmmword ptr [r11+rcx*8] vmovups zmm3, zmmword ptr [r11+rcx*8+0x40] vmovups zmm5, zmmword ptr [r11+rcx*8+0x80] vmovups zmm7, zmmword ptr [r11+rcx*8+0xc0]

Example 18-4. Masking with Assembly (Contd.)

vmulpd ymm3, ymm16, ymm2 vmulpd ymm8, ymm17, ymm7 vmovupd ymm1, ymmword ptr [rax+rcx*8+0x60] vmovupd ymm6, ymmword ptr [rax+rcx*8+0x80] vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymm2, ymmword ptr [r11+rcx*8+0x60] vmovupd ymm7, ymmword ptr [r11+rcx*8+0x80] vmovupd ymmword ptr [rsi], ymm5 vmovupd ymmword ptr [rsi+0x20], ymm10 vcmppd ymm4, ymm0, ymm1, 0x1 vcmppd ymm9, ymm0, ymm6, 0x1 vandpd ymm18, ymm11, ymm14 vandpd ymm19, ymm1, ymm4 vandpd ymm20, ymm6, ymm9 vmulpd ymm13, ymm18, ymm12 vmulpd ymm3, ymm19, ymm2 vmulpd ymm8, ymm20, ymm7 vmovupd ymm11, ymmword ptr [rax+rcx*8+0xa0] vmovupd ymm1, ymmword ptr [rax+rcx*8+0xc0] vmovupd ymm6, ymmword ptr [rax+rcx*8+0xe0] vblendvpd ymm15, ymm12, ymm13, ymm14 vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymm12, ymmword ptr [r11+rcx*8+0xa0] vmovupd ymm2, ymmword ptr [r11+rcx*8+0xc0] vmovupd ymm7, ymmword ptr [r11+rcx*8+0xe0] vmovupd ymmword ptr [rsi+0x40], ymm15 vmovupd ymmword ptr [rsi+0x60], ymm5 vmovupd ymmword ptr [rsi+0x80], ymm10 vcmppd ymm14, ymm0, ymm11, 0x1 vcmppd ymm4, ymm0, ymm1, 0x1 vcmppd ymm9, ymm0, ymm6, 0x1 vandpd ymm21, ymm11, ymm14 add rcx, 0x20 vandpd ymm22, ymm1, ymm4 vandpd ymm23, ymm6, ymm9 vmulpd ymm13, ymm21, ymm12 vmulpd ymm3, ymm22, ymm2 vmulpd ymm8, ymm23, ymm7 vblendvpd ymm15, ymm12, ymm13, ymm14 vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymmword ptr [rsi+0xa0], ymm15 vmovupd ymmword ptr [rsi+0xc0], ymm5 vmovupd ymmword ptr [rsi+0xe0], ymm10 add rsi, 0x100 cmp r9d, r8d jb loop	vcmppd k1, zmm8, zmm0, 0x1 vcmppd k2, zmm8, zmm2, 0x1 vcmppd k3, zmm8, zmm4, 0x1 vcmppd k4, zmm8, zmm6, 0x1 vmulpd zmm1{k1}, zmm0, zmm1 vmulpd zmm3{k2}, zmm2, zmm3 vmulpd zmm5{k3}, zmm4, zmm5 vmulpd zmm7{k4}, zmm6, zmm7 vmovups zmmword ptr [rsi], zmm1 vmovups zmmword ptr [rsi+0x40], zmm3 vmovups zmmword ptr [rsi+0x80], zmm5 vmovups zmmword ptr [rsi+0xc0], zmm7 add rcx, 0x20 add rsi, 0x100 cmp r9d, r8d jb loop
Baseline	Speedup: 2.9x

18.2.2 Masking Cost

Using masking may result in lower performance than the corresponding non-masked code. This may be caused by one of the following situations:

- An additional blend operation on each load.
- Dependency on the destination when using merge masking. This dependency does not exist when using zero masking.
- More restrictive masking forwarding rules (see Forwarding and Memory Masking for more information).

The following example shows how using merge masking creates a dependency on the destination register.

Example 18-5. Masking Example

No Masking	Merge Masking	Zero Masking
mov rbx, iter loop: vmulps zmm0, zmm9, zmm8 vmulps zmm1, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}, zmm9, zmm8 vmulps zmm1{k1}, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}{z}, zmm9, zmm8 vmulps zmm1{k1}{z}, zmm9, zmm8 dec rbx jnle loop
Baseline	Slowdown: 4x	Slowdown: Equal to baseline.

With no masking, the processor executes 2 multiplies per cycle on a 2 FMA server.

With merge masking, the processor executes 2 multiplies every 4 cycles as the multiplies in iteration N depend on the output of the multiplies in iteration N-1.

Zero masking does not have a dependency on the destination register and therefore can execute 2 multiplies per cycle on a 2 FMA server.

Recommendation: *Masking has a cost, so use it only when necessary. When possible, use zero masking rather than merge masking.*

18.2.3 Masking vs. Blending

This section discusses the advantages and disadvantages of using blending vs. masking for conditional code.

Consider the following code:

```
for ( i=0; i<SIZE; i++ )
{
     if ( a[i] > 0 )
    {
          b[i] *= 2;
    }
   else
    {
          b[i] /= 2;
    }
}
```

The example below shows two possible compilation alternatives of the code.

- Alternative 1 uses masked code and straight-forward arithmetic processing of data.
- Alternative 2 splits code to two independent unmasked flows that are processed one after another, and then a masked move (blending), just before storing to memory.

Example 18-6. Masking vs. Blending Example 1

Alternative 1	Alternative 2

```
    mov rax, pImage
    mov rbx, pImage1
    mov rcx, pOutImage
    mov rdx, len
    vpxord zmm0, zmm0, zmm0
mainloop:
    vmovdqa32 zmm2, [rax+rdx*4-0x40]
    vmovdqa32 zmm1, [rbx+rdx*4-0x40]
    vpcmpgtd k1, zmm1, zmm0
    knotw k2, k1
(1) vpslld zmm2 {k1}, zmm2, 1
(2) vpsrld zmm2 {k2}, zmm2, 1
(3) vmovdqa32 [rcx+rdx*4-0x40], zmm2
    sub rdx, 16
    jne mainloop
```
	```
	    mov rax, pImage
	    mov rbx, pImage1
	    mov rcx, pOutImage
	    mov rdx, len
	    vpxord zmm0, zmm0, zmm0
	mainloop:
	    vmovdqa32 zmm2, [rax+rdx*4-0x40]
	    vmovdqa32 zmm1, [rbx+rdx*4-0x40]
	    vpcmpgtd k1, zmm1, zmm0
	    vmovdqa32 zmm3, zmm2
	    vpslld zmm2, zmm2, 1
	    vpsrld zmm3, zmm3, 1
	(1) vmovdqa32 zmm3 {k1}, zmm2
	(2) vmovdqa32 [rcx+rdx*4-0x40], zmm3
	    sub rdx, 16
	    jne mainloop
	```

Baseline cycles 1x Baseline instructions 1x	Speedup: 1.23x Instructions: 1.11x

In Alternative 1, there is a dependency between instructions (1) and (2), and (2) and (3). That means that instruction (2) has to wait for the result of the blending of instruction (1), before starting execution, and instruction (3) needs to wait for instruction (2).

In Alternative 2, there is only one such dependency because each branch of conditional code is executed in parallel on all the data, and a mask is used for blending back to one register only before writing data back to the memory.

Blending is faster, but it does not mask exceptions, which may occur on the unmasked data.

Alternative 2 executes 11% more instructions; it provides 23% speedup in overall execution. Alternative 2 uses an extra register (zmm3). This extra register usage may cause extra latency in case of register pressure (freeing register to memory and loading it afterwards).

The following code is another example of masking vs. blending.

```
for (int i = 0;i<len;i++){
      if (a[i] > b[i]){
          a[i] += b[i];
     }
}
```

Example 18-7. Masking vs. Blending Example 2

Alternative 1	Alternative 2
mov rax,a mov rbx,b mov rdx,size2 loop1: vmovdqa32 zmm1,[rax +rdx*4 -0x40] vmovdqa32 zmm2,[rbx +rdx*4 -0x40] (1) vpcmpgtd k1,zmm1,zmm2 (2) vmovdqa32 zmm3{k1}{z},zmm2 (3) vpaddd zmm1,zmm1,zmm3 vmovdqa32 [rax +rdx*4 -0x40],zmm1 sub rdx,16 jne loop1	mov rax,a mov rbx,b mov rdx,size2 loop1: vmovdqa32 zmm1,[rax +rdx*4 -0x40] vmovdqa32 zmm2,[rbx +rdx*4 -0x40] (1)vpcmpgtd k1,zmm1,zmm2 (2)vpaddd zmm1{k1},zmm1,zmm2 vmovdqa32 [rax +rdx*4 -0x40],zmm1 sub rdx,16 jne loop1
Baseline cycles 1x Baseline instructions 1x	Speedup: 1.05x Instructions: 0.87x

In Alternative 1, there is a dependency between instructions (1) and (2), and (2) and (3).

In Alternative 2, there are only 2 instructions in the dependency chain: (1) and (2).

18.2.4 Nested Conditions / Mask Aggregation

Intel AVX-512 contains a set of instructions for mask operation, which enable executing all bitwise logical operators on a mask register, facilitating implementation of nested and/or multiply conditions.

In the following example, logical and (&&) is executed using a *kandw* instruction.

```
for(int iX = 0; iX < iBufferWidth; iX++)
{
     if ((*pInImage)>0 && ((*pInImage)&3)==3)
    {
            *pRefImage =  (*pInImage)+5;
    }
    else
    {
            *pRefImage = (*pInImage);
    }

    pRefImage++;
    pInImage++;
}
```

Example 18-8. Multiple Condition Execution

Scalar	Intel® AVX2	Intel® AVX-512
mov rsi, pImage mov rdi, pOutImage mov rbx, len xor rax, rax mainloop: mov r8d, dword ptr [rsi+rax*4] mov r9d, r8d cmp r8d, 0 jle label1 and r9d, 0x3 cmp r9d, 3 jne label1 add r8d, 5 label1: mov dword ptr [rdi+rax*4], r8d add rax, 1 cmp rax, rbx jne mainloop	mov rsi, pImage mov rdi, pOutImage mov rbx, len xor rax, rax vpbroadcastd ymm1, [five] vpbroadcastd ymm7, [three] vpxor ymm3, ymm3, ymm3 mainloop: vmovdqa ymm0, [rsi+rax*4] vmovaps ymm6, ymm0 vpcmpgtd ymm5, ymm0, ymm3 vpand ymm6, ymm6, ymm7 vpcmpeqd ymm6, ymm6, ymm7 vpand ymm5, ymm5, ymm6 vpaddd ymm4, ymm0, ymm1 vblendvps ymm4, ymm0, ymm4, ymm5 vmovdqa [rdi+rax*4], ymm4 add rax, 8 cmp rax, rbx jne mainloop	mov rsi, pImage mov rdi, pOutImage mov rbx, len xor rax, rax vpbroadcastd zmm1, [five] vpbroadcastd zmm5, [three] vpxord zmm3, zmm3, zmm3 mainloop: vmovdqa32 zmm0, [rsi+rax*4] vpcmpgtd k1, zmm0, zmm3 vpandd zmm6, zmm5, zmm0 vpcmpeqd k2, zmm6, zmm5 kandw k1, k2, k1 vpaddd zmm0 {k1}, zmm0, zmm1 vmovdqa32 [rdi+rax*4], zmm0 add rax, 16 cmp rax, rbx jne mainloop
Baseline 1x	Speedup: 5x	Speedup: 11x

18.2.5 Memory Masking Microarchitecture Improvements

Masking improvements since Broadwell microarchitecture are detailed below.

Table 18-1. Cache Comparison Between Skylake Server Microarchitecture and Broadwell Microarchitecture

Item	Broadwell Microarchitecture	Skylake Server Microarchitecture
1	The address of a vmaskmov store is considered as resolved only after the mask is known. Loads that follow a masked store may be blocked, depending on the memory disambiguation predictor, until the mask value is known.	This issue is resolved. The address of a vmaskmov store can be resolved before the mask is known.
2	If the mask is not all 1 or all 0, loads that depend on the masked store must wait until the store data is written to the cache. If the mask is all 1 the data can be forwarded from the masked store to the dependent loads. If the mask is all 0 the loads do not depend on the masked store.	If the mask is not all 1 or all 0, loads that depend on the masked store must wait until the store data is written to the cache. If the mask is all 1 the data can be forwarded from the masked store to the dependent loads. If the mask is all 0 the loads do not depend on the masked store.
3	When including an illegal memory address range with masked loads (using the vmaskmov instruction), the processor might take a multi-cycle “assist” to determine if any part of the illegal range has a one mask value. This assist might occur even when the mask was ”all-zero” and it seemed obvious to the programmer that the load should not be executed.	For Intel AVX-512 masking, if the mask is all-zeros then memory faults will be ignored and no assist will be issued.

18.2.6 Peeling and Remainder Masking

Accessing cache line aligned data gives better performance than accessing non-aligned data. In many cases, the address is not known in compile time, or known and not-aligned. In these cases a peeling algorithm may be proposed, to process first elements in masked mode, up to first aligned address, and then process unmasked body and masked remainder. This method increases code size, but improves data processing overall.

The following code is an example of peeling and remainder masking.

```
for (size_t i = 0; i < len; i++)
    pOutImage[i] = (pInImage[i] * alfa) + add_value;
```

The table below shows the difference in implementation and execution speed of two versions of the code, both working on unaligned output data array.

Example 18-9. Peeling and Remainder Masking

No peeling, unmasked body, masked remainder	Peeling, unmasked body, masked remainder
mov rbx, pOutImage // Output	mov rax, pImage // Input
mov rax, pImage // Input	mov rbx, pOutImage // Output
mov rcx, len	mov rcx, len
mov edx, addValue	movss xmm0, addValue
vpbroadcastd zmm0, edx	vpbroadcastd zmm0, xmm0
mov edx, alfa	movss xmm1, alfa
vpbroadcastd zmm3, edx	vpbroadcastd zmm3, xmm1
mov rdx, rcx	xor r8, r8
sar rdx, 4 // 16 elements per iteration, RDX - number of full iterations	xor r9, r9
	vmovups zmm10, [indices]
jz remainder // no full iterations	vpbroadcastd zmm12, ecx
xor r8, r8	
vmovups zmm10, [indices]	peeling:
	mov rdx, rbx
mainloop:	and rdx, 0x3F
vmovups zmm1, [rax + r8]	jz endofpeeling //nothing to peel
vfmadd213ps zmm1, zmm3, zmm0	neg rdx
vmovups [rbx + r8], zmm1	add rdx, 64 // 64 - X
add r8, 0x40	// now rdx contains the number of bytes to the closest alignment
sub rdx, 1	
jne mainloop	mov r9, rdx
	sar r9, 2 // now r9 contains number of elements in peeling
remainder:	
// produce mask for remainder	vpbroadcastd zmm12, r9d
and rcx, 0xF // number of elements in remainder	vpcmpd k2, zmm10, zmm12, 1 //compare lower to produce mask for peeling
jz end // no elements in remainder	
vpbroadcastd zmm2, ecx	
vpcmpd k2, zmm10, zmm2, 1 //compare lower	vmovups zmm1 {k2}{z}, [rax]
	vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
vmovups zmm1 {k2}{z}, [rax + r8]	vmovups [rbx] {k2}, zmm1 //unaligned store
vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0	
vmovups [rbx + r8] {k2}, zmm1	endofpeeling:
end:	sub rcx, r9
	mov r8, rcx
	sar r8, 4 //number of full iterations
	jz remainder //no full iterations

Example 18-9. Peeling and Remainder Masking (Contd.)

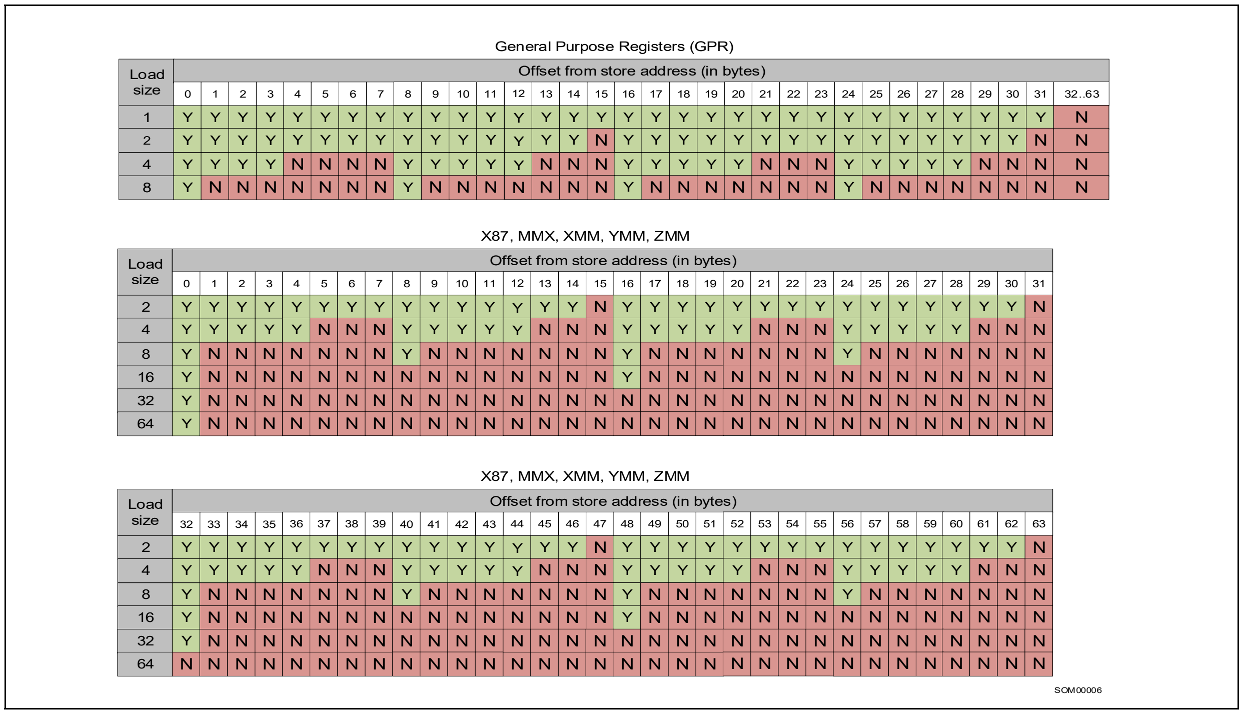
```
mainloop:
    vmovups zmm1, [rax + rdx]
    vfmadd213ps  zmm1, zmm3, zmm0
    vmovaps [rbx + rdx], zmm1 // aligned store is safe here
!!
    add rdx, 0x40
    sub r8, 1
    jne mainloop
remainder:
    // produce mask for remainder
    and rcx, 0xF // number of elements in remainder
    jz end // no elements in remainder
    vpbroadcastd zmm2, ecx
     vpcmpd k2, zmm10, zmm2, 1 //compare lower
    vmovups zmm1 {k2}{z}, [rax + rdx]
    vfmadd213ps  zmm1 {k2}{z}, zmm3, zmm0
    vmovaps [rbx + rdx] {k2}, zmm1 //aligned
end:
```

Baseline 1x	Speedup: 1.04x

18.3 FORWARDING AND UNMASKED OPERATIONS

When using an unmasked store instruction, and load instruction after it, data forwarding depends on load type, size and address offset from store address, and does not depend on the store address itself (i.e., the store address does not have to be aligned to or fit into cache line, forwarding will occur for non-aligned and even line-split stores).

The figure below describes all possible cases when data forwarding will occur.



General Purpose Registers (GPR)

Load size	Offset from store address (in bytes)																															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32..63
1	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N
4	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N

X87, MMX, XMM, YMM, ZMM

Load size	Offset from store address (in bytes)																														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
4	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
16	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
64	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

X87, MMX, XMM, YMM, ZMM

Load size	Offset from store address (in bytes)																														
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
4	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
16	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
64	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

SOM00006

Figure 18-3. Data Forwarding Cases

There are two important points to be considered when using data forwarding.

1. Data forwarding to GPR is possible only from the lower 256 bits of store instruction. Note this when loading GPR with data that has recently been written.
2. Do not use masks, as forwarding is supported only for certain masks.

18.4 FORWARDING AND MEMORY MASKING

When using masked store and load, consider the following:

- When the mask is not all-ones or all-zeroes, the load operation, following the masked store operation from the same address is blocked, until the data is written to the cache.
- Unlike GPR forwarding rules, vector loads whether or not they are masked, do not forward unless load and store addresses are exactly the same.
  - st_mask = 10101010, ld_mask = 01010101, can forward: no, should block: yes
  - st_mask = 00001111, ld_mask = 00000011, can forward: no, should block: yes
- When the mask is all-ones, blocking does not occur, because the data may be forwarded to the load operation.
  - st_mask = 11111111, ld_mask = don't care, can forward: yes, should block: no
- When mask is all-zeroes, blocking does not occur, though neither does forwarding.
  - st_mask = 00000000, ld_mask = don't care, can forward: no, should block: no

In summary, a masked store should be used carefully, for example, if the remainder size is known at compile time to be 1, and there is a load operation from the same cache line after it (or there is an overlap in addresses + vector lengths), it may be better to use scalar remainder processing, rather than a masked remainder block.

18.5 DATA COMPRESS

The data compress operation reads elements from an input buffer on indices specified by mask register 1's bits. The elements which have been read, are then written to the destination buffer. If the number of elements is less than the destination register size, the rest of the space is filled with zeroes.

The following figure describes the data compress operation.

```
if (k[i] == 1)
{
        dest[a] = src[i];
        a++;
}
```

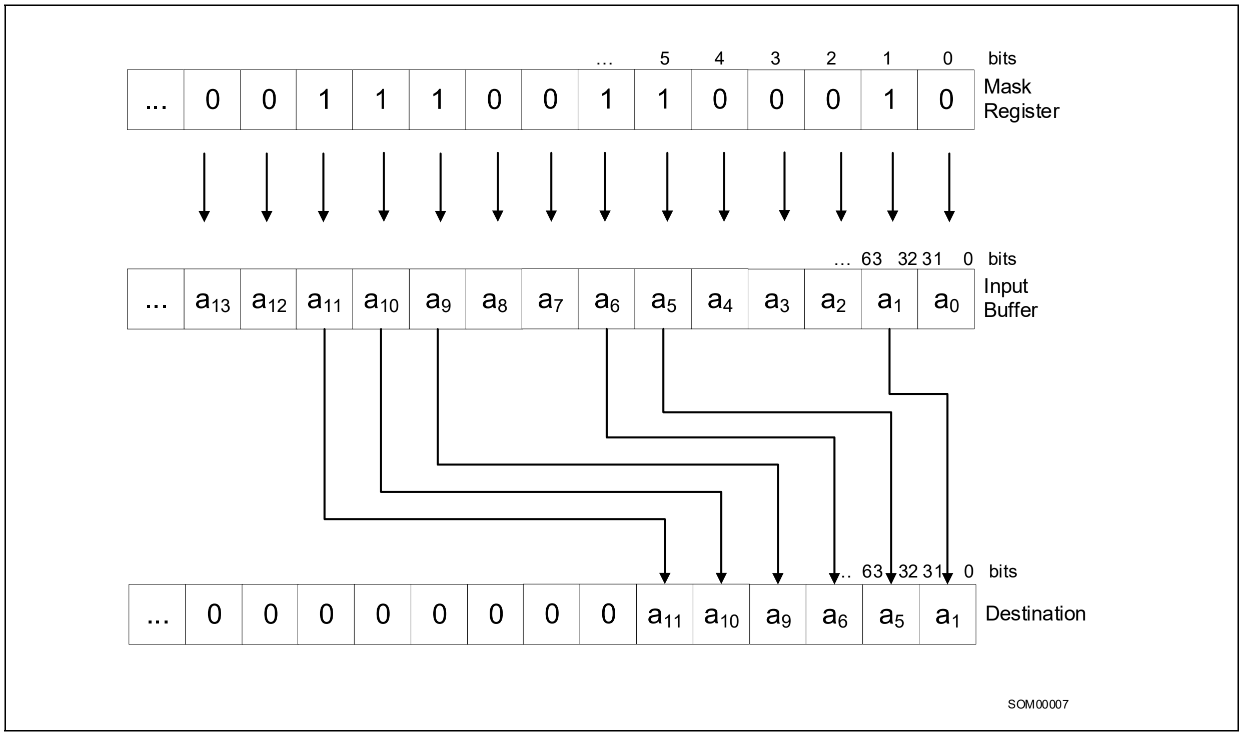


Figure 18-4. Data Compress Operation

18.5.1 Data Compress Example

The following snippet shows collection of all positive elements from one array to another array.

```
for (int i=0; i<SIZE; i++)
{
    if ( a[i] > 0 )
      b[j++] = a[i];
}
```

Following are four implementations for the compress operation from an array of dword elements.

- Alternative 1 uses scalar data access and checks each element separately. If it is greater than 0 it is written to the destination array.
- Alternative 2 is Intel AVX code that uses a shuffle instruction together with the pre-allocated and pre-initialized table with shuffle keys. The compare instruction provides the entry point number to the shuffle-key table. Then the key is loaded and the original array is shuffled according to the keys. Four elements are processed in each iteration.
- Alternative 3 uses the same algorithm as in Alternative 2, but uses Intel AVX2 256-bit registers, and a permutation on the dword instruction instead of using byte shuffle. Eight elements are processed in each iteration.
- Alternative 4 is an Intel AVX-512 algorithm, which uses the *vpcompress* instruction together with the mask register as a compress key. 16 elements are processed in each iteration.

Example 18-10. Comparing Intel® AVX-512 Data Compress with Other Alternatives

Alternative 1: Scalar

```
        mov rsi, source
        mov rdi, dest
        mov r9, len

        xor r8, r8
        xor r10, r10
mainloop:
        mov r11d, dword ptr [rsi+r8*4]
        test r11d, r11d
        jle  m1
        mov dword ptr [rdi+r10*4], r11d
        inc r10
m1:
        inc r8
        cmp r8, r9
        jne mainloop
```

Baseline 1x

Example 18-10. Comparing Intel® AVX-512 Data Compress with Other Alternatives (Contd.)

Alternative 2: Intel® AVX

```
        mov rsi, source
        mov rdi, dest
        mov r14, shuffle_LUT
        mov r15, write_mask
        mov r9, len

        xor r8, r8
        xor r11, r11
        vpxor xmm0, xmm0, xmm0
mainloop:
        vmovdqa xmm1, [rsi+r8*4]
        vpcmpgtd xmm2, xmm1, xmm0
        mov r10, 4
        vmovmskps r13, xmm2
        shl r13, 4
        vmovdqu xmm3, [r14+r13]
        vpshufb xmm2, xmm1, xmm3
        popcnt r13, r13
        sub r10, r13
        vmovdqu xmm3, [r15+r10*4]
        vmaskmovps [rdi+r11*4], xmm3, xmm2
        add r11, r13
        add r8, 4
        cmp r8, r9
        jne mainloop

shuffle_LUT:
   .int 0x80808080, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x80808080, 0x80808080, 0x80808080
   .int 0x07060504, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x80808080, 0x80808080
   .int 0x0b0A0908, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x0b0A0908, 0x80808080, 0x80808080
   .int 0x07060504, 0x0b0A0908, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x0b0A0908, 0x80808080
   .int 0x0F0E0D0C, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x07060504, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x0F0E0D0C, 0x80808080
   .int 0x0b0A0908, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x03020100, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
   .int 0x07060504, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
   .int 0x03020100, 0x07060504, 0x0b0A0908, 0x0F0E0D0C

write_mask:
   .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
   .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

Speedup: 2.87x

Example 18-10. Comparing Intel® AVX-512 Data Compress with Other Alternatives (Contd.)

Alternative 3: Intel® AVX2

```
            mov rsi, source
            mov rdi, dest
            mov r14, shuffle_LUT
            mov r15, write_mask
            mov r9, len

            xor r8, r8
            xor r11, r11
            vpxor ymm0, ymm0, ymm0
mainloop:
            vmovdqa ymm1, [rsi+r8*4]
            vpcmpgtd ymm2, ymm1, ymm0
            mov r10, 8
            vmovmskps r13, ymm2
            shl r13, 5
            vmovdqu ymm3, [r14+r13]
            vpermd ymm2, ymm3, ymm1
            popcnt r13, r13
            sub r10, r13
            vmovdqu ymm3, [r15+r10*4]
            vmaskmovps [rdi+r11*4], ymm3, ymm2
            add r11, r13
            add r8, 8
            cmp r8, r9
            jne mainloop

// The lookup table is too large to reproduce in the document. It consists of 256 rows of 8 32 bit integers.
//The first 8 and the last 8 rows are shown below.

shuffle_LUT:
.int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0
// Skipping 240 lines
.int 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0, 0x0
.int 0x0, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x0, 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x0, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7
```

Example 18-10. Comparing Intel® AVX-512 Data Compress with Other Alternatives (Contd.)

```
write_mask:
.int 0x80000000, 0x80000000, 0x80000000, 0x80000000
.int 0x80000000, 0x80000000, 0x80000000, 0x80000000
.int 0x00000000, 0x00000000, 0x00000000, 0x00000000
.int 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

Speedup: 5.27x

Alternative 4: Intel® AVX-512

```
        mov rsi, source
        mov rdi, dest
        mov r9, len

        xor r8, r8
        xor r10, r10
        vpxord zmm0, zmm0, zmm0
mainloop:
        vmovdqa32 zmm1, [rsi+r8*4]
        vpcmpgtd k1, zmm1, zmm0
        vpcompressd zmm2 {k1}, zmm1
        vmovdqu32 [rdi+r10*4], zmm2
        kmovd r11d, k1
        popcnt r12, r11
        add r8, 16
        add r10, r12
        cmp r8, r9
        jne mainloop
```

Speedup: 11.9x

18.6 DATA EXPAND

Data expand operations read elements from the source array (register) and put them in the destination register in the places indicated by enabled bits in the mask register. If the number of enabled bits is less than destination register size, the extra values are ignored.

```
if (k[i] == 1)
{
          dest[i] = src[a];
          a++;
}
```

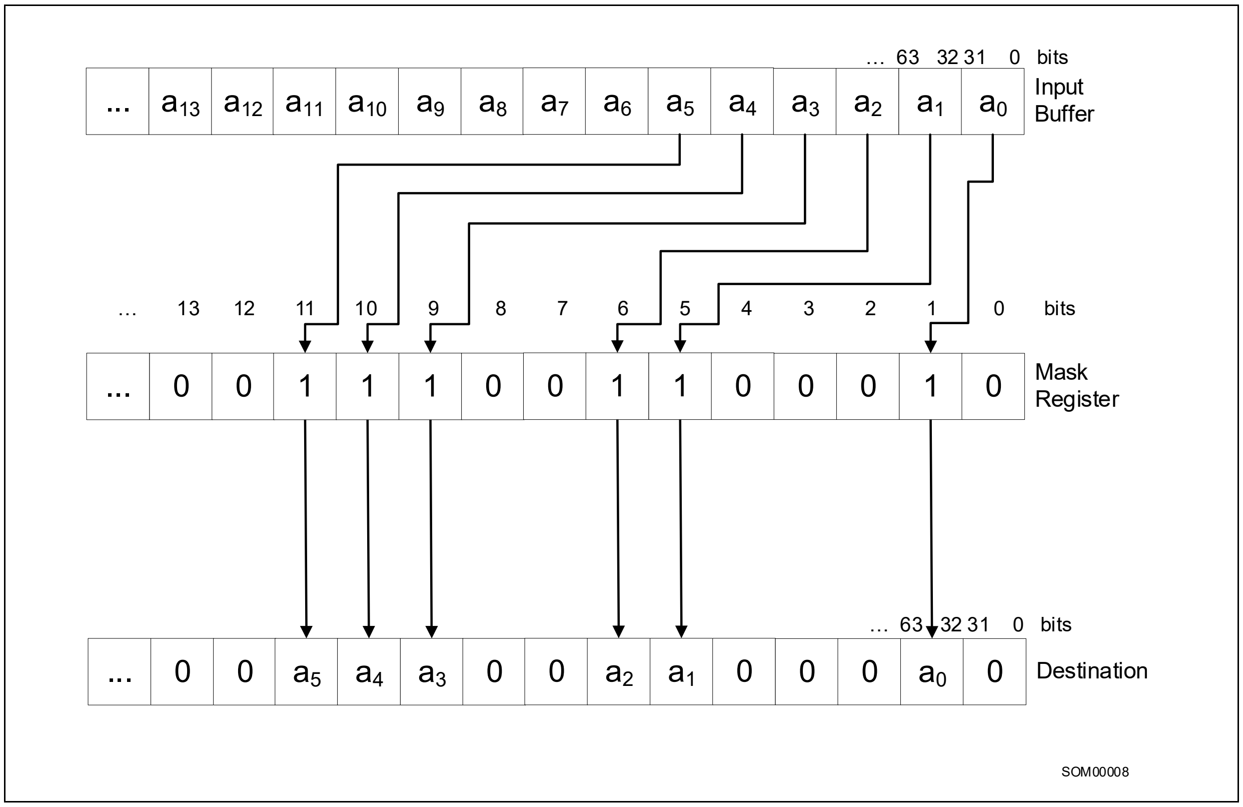

... 63 32 31 0 bits
...
a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
Input Buffer
... 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bits
... 0 0 1 1 1 0 0 1 1 0 0 0 1 0
Mask Register
... 63 32 31 0 bits
... 0 0 a_5 a_4 a_3 0 0 a_2 a_1 0 0 0 a_0 0
Destination
SOM00008

Figure 18-5. Data Expand Operation

18.6.1 Data Expand Example

The following snippet shows an example of using the expand operation. For every positive number in an array, the code sets its consecutive number among positives.

```
for (int i=0; i<SIZE; i++)
{
    if (a[i] > 0)
        dest[i] = a[count++];
    else
        dest[i] = 0;
}
```

Here are three implementations for the expand operation from an array of 16 dword elements.

- Alternative 1 uses scalar data access and checks each element separately. If it is greater than 0 then the corresponding element in the destination array is rewritten with the value from source value at index count, and the counter is incremented.
- Alternative 2 shows Intel AVX2 code that uses a shuffle instruction together with the pre-allocated and pre-initialized table with shuffle keys. The compare instruction provides the entry point number to the shuffle-key table. Then the key is loaded and the original array is shuffled according to the keys. Four elements are processed in each iteration.
- Alternative 3 shows Intel AVX-512 code, which uses the *vpexpandd* instruction together with the mask register as an expand key. 16 elements are processed in each iteration.

Example 18-11. Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives

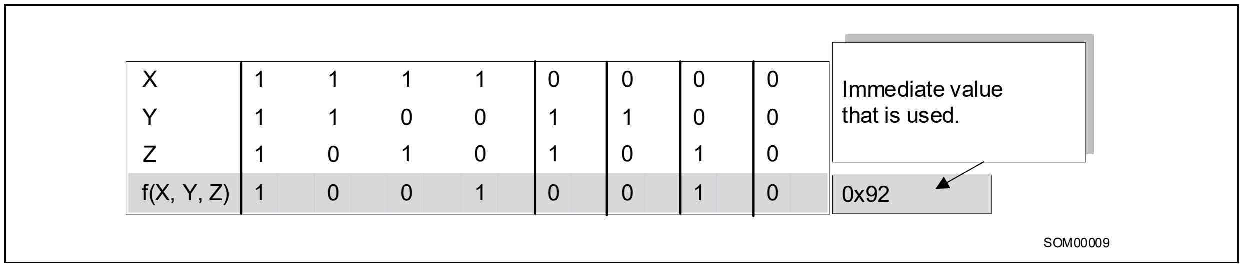
Alternative 1: Scalar	Alternative 2: Intel® AVX2 Code	Alternative 3: Intel® AVX-512 Code
mov rsi, input	mov rsi, input	vpxord zmm0, zmm0, zmm0
mov rdi, output	mov rdi, output	mainloop:
mov r9, len	mov r9, len	vmovdqa32 zmm1, [rsi+r8*4]
xor r8, r8	xor r8, r8	vpcmpgtd k1, zmm1, zmm0
xor r10, r10	xor r10, r10	vmovdqu32 zmm1,
mainloop:	vpxor ymm0, ymm0, ymm0	[rsi+r10*4]
mov r11d, dword ptr	mov r14, shuf2	vpexpandd zmm2 {k1}{z},
[rsi+r8*4]	mainloop:	zmm1
test r11d, r11d	vmovdqa ymm1, [rsi+r8*4]	vmovdqu32 [rdi+r8*4], zmm2
jle m1	vpxor ymm4, ymm4, ymm4	add r8, 16
mov r11d, dword ptr	vpcmpgtd ymm2, ymm1, ymm0	kmovd r11d, k1
[rsi+r10*4]	vmovdqu ymm1, [rsi+r10*4]	popcnt r12, r11
mov dword ptr [rdi+r8*4],	vmovmskps r13, ymm2	add r10, r12
r11d	shl r13, 5	cmp r8, r9
inc r10	vmovdqa ymm3, [r14+r13]	jne mainloop
m1:	vpermd ymm4, ymm3, ymm1	
inc r8	popcnt r13, r13	
cmp r8, r9	add r10, r13	
jne mainloop	vmaskmovps [rdi+r8*4], ymm2,	
	ymm4	
	add r8, 8	
	cmp r8, r9	
	jne mainloop	
	// The lookup table is too large to	
	// reproduce in the document. It consists	
	// of 256 rows of 8 32-bit integers. The	
	// first 8 and the last 8 rows are shown	
	// below. The table needs to be 32-byte	
	// aligned.	
	shuf2:	
	.int 0, 0, 0, 0, 0, 0, 0, 0	
	.int 0, 0, 0, 0, 0, 0, 0, 0	
	.int 0, 0, 0, 0, 0, 0, 0, 0	
	.int 0, 1, 0, 0, 0, 0, 0, 0	
	.int 0, 0, 0, 0, 0, 0, 0, 0	
	.int 0, 0, 1, 0, 0, 0, 0, 0	
	.int 0, 0, 1, 0, 0, 0, 0, 0	
	.int 0, 1, 2, 0, 0, 0, 0, 0	
	// Skipping 240 lines	
	.int 0, 0, 0, 0, 1, 2, 3, 4	
	.int 0, 0, 0, 1, 2, 3, 4, 5	
	.int 0, 0, 0, 1, 2, 3, 4, 5	
	.int 0, 1, 0, 2, 3, 4, 5, 6	
	.int 0, 0, 0, 1, 2, 3, 4, 5	
	.int 0, 0, 1, 2, 3, 4, 5, 6	
	.int 0, 0, 1, 2, 3, 4, 5, 6	
	.int 0, 1, 2, 3, 4, 5, 6, 7	
Baseline 1x	Speedup: 4.23x	Speedup: 8.58x

18.7 TERNARY LOGIC

A ternary logic *vpternlog* operation executes any bitwise logical function between three operands in one instruction. The instruction requires three operands and an immediate value, which is the truth table of this logical expression. The first operand is used as destination, and, therefore, destroyed after the execution.

18.7.1 Ternary Logic Example 1

The following example shows a bitwise logic function of three variables. The function in this example is defined by the following truth table.



X	1	1	1	1	0	0	0	0
Y	1	1	0	0	1	1	0	0
Z	1	0	1	0	1	0	1	0
f(X, Y, Z)	1	0	0	1	0	0	1	0

Immediate value that is used.
0x92
SOM00009

Figure 18-6. Ternary Logic Example 1 Truth Table

Using Karnaugh maps on this truth table, we can define the function as:

f(X,Y,Z) = $\bar{x}\,\bar{y}\,z \lor xyz \lor x\,\bar{y}\,\bar{z}$

or, in shorter notation, using fewer binary operations:

f(X,Y,Z) = $\bar{y}(z \oplus x) \lor xyz$

The C code for the function above is as follows:

```
for (int i=0; i<SIZE; i++)
{
      Dst[i] = ((~Src2[i]) & (Src1[i] ^ Src3[i])) | (Src1[i] & Src2[i] & Src3[i]);
}
```

The value of the function for each combination of X, Y and Z gives an immediate value that is used in the instruction.

Here are three implementations for this logical function applied to all values in X, Y and Z arrays.

- Alternative 1 is an Intel AVX2 256-bit vector computation, using bitwise logical functions available in Intel AVX2.
- Alternative 2 is a 512-bit vector computation, using bitwise logical functions available in Intel AVX-512, without using the *vpternlog* instruction.
- Alternative 3 is an Intel AVX-512 512-bit vector computation, using the *vpternlog* instruction.

All alternatives in the table are unrolled by factor 2.

Example 18-12. Comparing Ternary Logic to Other Alternatives

Alternative 1: Intel® AVX2

```
    mov rax, src1
    mov rbx, src2
    mov rcx, src3
    mov r11, dst
    mov r8, len
    xor r10, r10
mainloop:
    vmovdqu ymm1, ymmword ptr [rax+r10*4]
    vmovdqu ymm3, ymmword ptr [rdx+r10*4]
    vmovdqu ymm2, ymmword ptr [rcx+r10*4]
    vmovdqu ymm10, ymmword ptr [rcx+r10*4+0x20]
    vpand ymm0, ymm1, ymm3
    vpxor ymm4, ymm1, ymm2
    vpand ymm5, ymm0, ymm2
    vpandn ymm6, ymm3, ymm4
    vpor ymm7, ymm5, ymm6
    vmovdqu ymmword ptr [r11+r10*4], ymm7
    vmovdqu ymm9, ymmword ptr [rax+r10*4+0x20]
    vmovdqu ymm11, ymmword ptr [rdx+r10*4+0x20]
    vpxor ymm12, ymm9, ymm10
    vpand ymm8, ymm9, ymm11
    vpandn ymm14, ymm11, ymm12
    vpand ymm13, ymm8, ymm10
    vpor ymm15, ymm13, ymm14
    vmovdqu ymmword ptr [r11+r10*4+0x20], ymm15
```

```
    add r10, 0x10
    cmp r10, r8
    jb mainloop
```

Baseline 1x

Example 18-12. Comparing Ternary Logic to Other Alternatives (Contd.)

Alternative 2: Intel® AVX-512 Logic Instructions	Alternative 3: Intel® AVX-512 using *vpternlog* Instruction
mov rdi, src1 mov rsi, src2 mov rdx, src3 mov r11, dst mov r8, len xor r10, r10 mainloop: vmovups zmm2, zmmword ptr [rdi+r10*4] vmovups zmm4, zmmword ptr [rdi+r10*4+0x40] vmovups zmm6, zmmword ptr [rsi+r10*4] vmovups zmm8, zmmword ptr [rsi+r10*4+0x40] vmovups zmm3, zmmword ptr [rdx+r10*4] vmovups zmm5, zmmword ptr [rdx+r10*4+0x40] vpandd zmm0, zmm2, zmm6 vpandd zmm1, zmm4, zmm8 vpxord zmm7, zmm2, zmm3 vpxord zmm9, zmm4, zmm5 vpandd zmm10, zmm0, zmm3 vpandd zmm12, zmm1, zmm5 vpandnd zmm11, zmm6, zmm7 vpandnd zmm13, zmm8, zmm9 vpord zmm14, zmm10, zmm11 vpord zmm15, zmm12, zmm13 vmovups zmmword ptr [r11+r10*4], zmm14 vmovups zmmword ptr [r11+r10*4+0x40], zmm15 add r10, 0x20 cmp r10, r9 jb mainloop	mov r9, src1 mov r8, src2 mov r10, src3 mov r11, dst mov rsi, len xor rax rax mainloop: vmovaps zmm1, [r8+rax*4] vmovaps zmm0, [r9+rax*4] vpternlogd zmm0,zmm1,[r10], 0x92 vmovaps [r11], zmm0 vmovaps zmm1, [r8+rax*4+0x40] vmovaps zmm0, [r9+rax*4+0x40] vpternlogd zmm0,zmm1, [r10+0x40], 0x92 vmovaps [r11+0x40], zmm0 add rax, 32 add r10, 0x80 add r11, 0x80 cmp rax, rsi jne mainloop
Speedup: 1.94x	Speedup: 2.36x (1.22x vs Intel® AVX-512 with logic instructions)

18.7.2 Ternary Logic Example 2

The next example is a sign change operation, frequently used in Fortran. Consider the following code, running on two arrays of floating point numbers.

```
for (int i=0; i<SIZE; i++)
{
      b[i] = a[i] > 0 ? b[i] : -b[i];
}
```

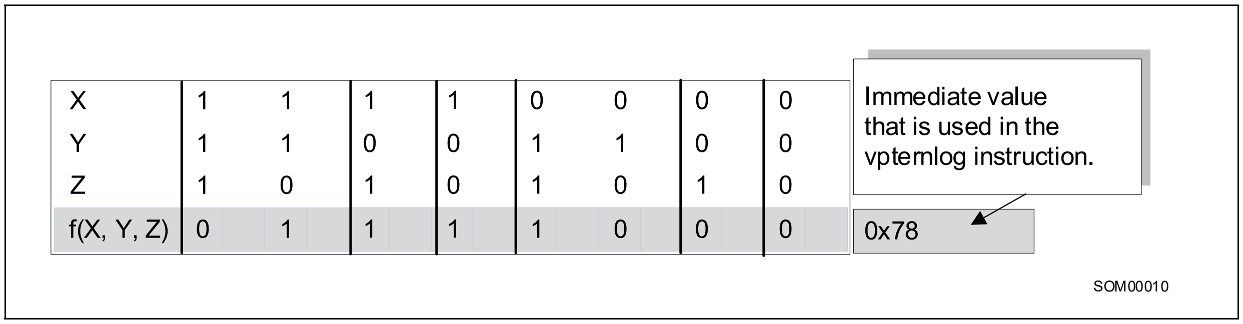
This code is equivalent to:

```
for (int i=0; i<SIZE; i++)
{
        b[i] = ( a[i] & 0x80000000 ) ^ b[i];
}
```

Or, in other words:

$x = (y \wedge z) \oplus x$

This logic expression gives the following truth table.



X	1	1	1	1	0	0	0	0
Y	1	1	0	0	1	1	0	0
Z	1	0	1	0	1	0	1	0
f(X, Y, Z)	0	1	1	1	1	0	0	0

Immediate value that is used in the vpternlog instruction.

0x78

Figure 18-7. Ternary Logic Example 2 Truth Table

Therefore one *vpternlog* instruction can be used instead of using two logic instructions (*vpand* and *vpxor*):

```
vpternlog x,y,z,0x78
```

18.8 NEW SHUFFLE INSTRUCTIONS

Intel AVX-512 added 3 new shuffle operations.

- vpermw: a new single source any-to-any word permute.
- permt2[w/d/q/ps/pd]: a new any to any 2 source permute (overriding src register).
- permi2[w/d/q/ps/pd]: a new any to any 2 source permute (overriding control register).

The following figure shows how *vpermi2ps* is used. Notice that in the following example zmm0 is the shuffle control but also the output register (the control register is overridden).

```
vpermi2ps zmm0, zmm1, zmm2
```

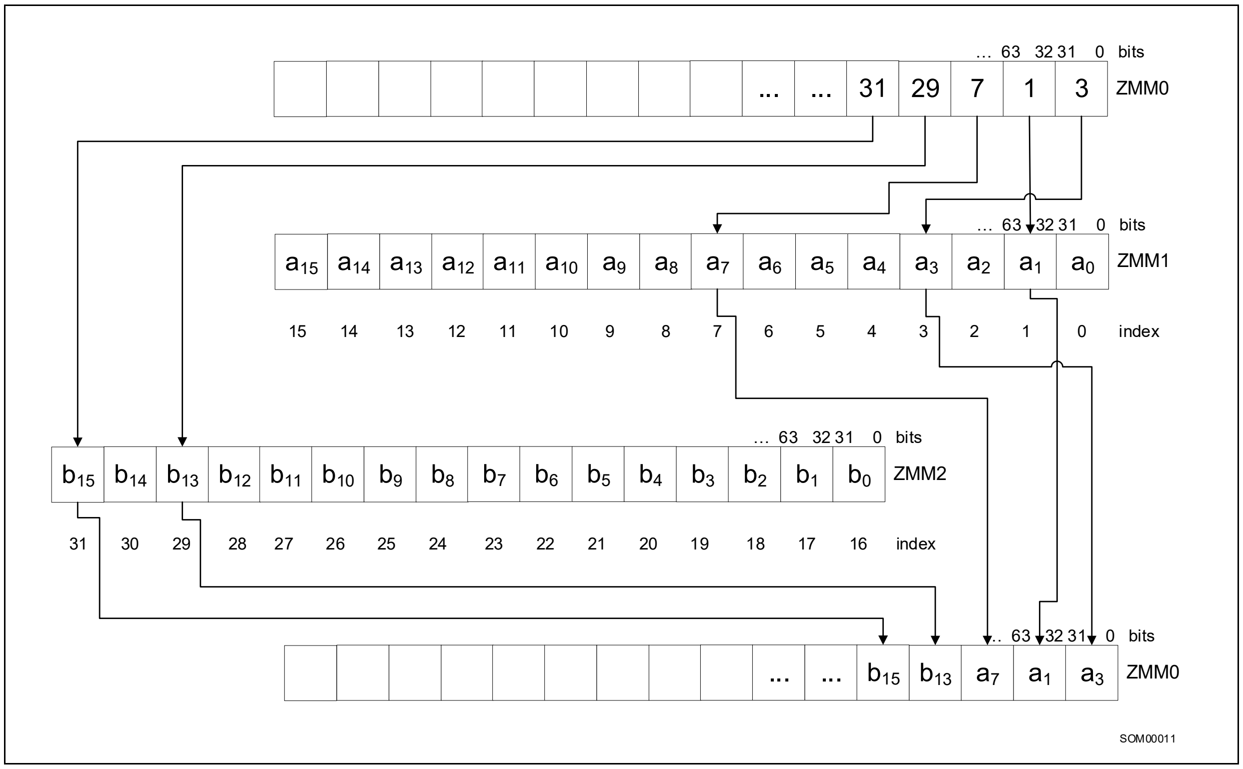

… 63 32 31 0 bits
… … 31 29 7 1 3
ZMM0
a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
ZMM1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 index
b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
ZMM2
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 index
… … b15 b13 a7 a1 a3
ZMM0
SOM00011

Figure 18-8. VPERMI2PS Instruction Operation

Note that the index register values must have the same resolution as the instruction and source registers (word when working on words, dword when working on dwords, etc.).

18.8.1 Two Source Permute Example

In this example we will show the use of the two source permute instructions in a matrix transpose operation. The matrix we want to transpose is square 8x8 matrix of word elements.

$$\begin{bmatrix} a_{00} & \cdots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \cdots & a_{77} \end{bmatrix}^{T} \rightarrow \begin{bmatrix} a_{00} & \cdots & a_{71} \\ \vdots & \ddots & \vdots \\ a_{17} & \cdots & a_{77} \end{bmatrix}$$

The corresponding C code is as follows (assuming each matrix occupies a continuous block of 8*8*2 = 128 bytes):

```
 for(int iY = 0; iY < 8; iY++)
{
        for(int iX = 0; iX < 8; iX++)
       {
               trasposedMatrix[iY*8+iX] = originalMatrix[iX*8+iY];
       }
}
```

Here are three implementations for this matrix transpose.

- Alternative 1 is scalar code, which accesses each element of the source matrix and puts it to the corresponding place in the destination matrix. This code does 64 (8x8) iterations per 1 matrix.
- Alternative 2 is Intel AVX2 code, which uses Intel AVX2 permutation and shuffle (unpack) instructions. Only 1 iteration per 8x8 matrix is required.
- Alternative 3 Intel AVX-512 code which uses the Two Source Permutation instructions. Note that this code first loads permutation masks, and then matrix data. The mask used to perform the permutation is stored in the following array:

```
short permMaskBuffer [8*8] = { 0, 8, 16, 24, 32, 40, 48, 56,
                           1, 9, 17, 25, 33, 41, 49, 57,
                           2, 10, 18, 26, 34, 42, 50, 58,
                           3, 11, 19, 27, 35, 43, 51, 59,
                            4, 12, 20, 28, 36, 44, 52, 60,
                            5, 13, 21, 29, 37, 45, 53, 61,
                            6, 14, 22, 30, 38, 46, 54, 62,
                            7, 15, 23, 31, 39, 47, 55, 63 };
```

Each alternative transposes 50 matrixes, 8x8 2-byte elements each.

Example 18-13. Matrix Transpose Alternatives

Alternative 1: Scalar code	**Alternative 2: Intel® AVX2 Code**	**Alternative 3: Intel® AVX-512 Code**
mov rsi, pImage	mov rsi, pImage	mov rax, permMaskBuffer
mov rdi, pOutImage	mov rdi, pOutImage	vmovdqa32 zmm10, [rax]
xor rdx, rdx	xor rdx, rdx	vmovdqa32 zmm11, [rax+0x40]
matrix_loop:	matrix_loop:	mov rsi, pImage
xor rax, rax	vmovdqa xmm0, [rsi]	mov rdi, pOutImage
outerloop:	vmovdqa xmm1, [rsi+0x10]	xor rdx, rdx
xor rbx, rbx	vmovdqa xmm2, [rsi+0x20]	matrix_loop:
innerloop:	vmovdqa xmm3, [rsi+0x30]	vmovdqa32 zmm2, [rsi]
mov rcx, rax		vmovdqa32 zmm3, [rsi+0x40]
shl rcx, 3	vinserti128 ymm0, ymm0, [rsi+0x40], 0x1	vmovdqa32 zmm0, zmm10
add rcx, rbx	vinserti128 ymm1, ymm1, [rsi+0x50], 0x1	vmovdqa32 zmm1, zmm11
mov r8w, word ptr [rsi+rcx*2]	vinserti128 ymm2, ymm2, [rsi+0x60], 0x1	vpermi2w zmm0, zmm2, zmm3
mov rcx, rbx	vinserti128 ymm3, ymm3, [rsi+0x70], 0x1	vpermi2w zmm1, zmm2, zmm3
shl rcx, 3		vmovdqa32 [rdi], zmm0
add rcx, rax	vpunpcklwd ymm4, ymm0, ymm1	vmovdqa32 [rdi+0x40], zmm1
mov word ptr [rdi+rcx*2], r8w	vpunpckhwd ymm5, ymm0, ymm1	
add rbx, 1	vpunpcklwd ymm6, ymm2, ymm3	add rdx, 1
cmp rbx, 8	vpunpckhwd ymm7, ymm2, ymm3	add rsi, 64*2
jne innerloop		add rdi, 64*2
add rax, 1	vpunpckldq ymm0, ymm4, ymm6	cmp rdx, 50
cmp rax, 8	vpunpckhdq ymm1, ymm4, ymm6	jne matrix_loop
jne outerloop	vpunpckldq ymm2, ymm5, ymm7	
add rdx, 1	vpunpckhdq ymm3, ymm5, ymm7	
add rsi, 64*2		
add rdi, 64*2	vpermq ymm0, ymm0, 0xD8	
cmp rdx, 50	vpermq ymm1, ymm1, 0xD8	
jne matrix_loop	vpermq ymm2, ymm2, 0xD8	
	vpermq ymm3, ymm3, 0xD8	
	vmovdqa [rdi], ymm0	
	vmovdqa [rdi+0x20], ymm1	
	vmovdqa [rdi+0x40], ymm2	
	vmovdqa [rdi+0x60], ymm3	
	add rdx, 1	
	add rsi, 64*2	
	add rdi, 64*2	
	cmp rdx, 50	
	jne matrix_loop	
Baseline 1x	Speedup: 13.7x	Speedup: 37.3x (2.7x vs Intel® AVX2 code)

18.9 BROADCAST

18.9.1 Embedded Broadcast

Intel AVX-512 introduces embedded broadcast operations, in which a broadcast operation is implied within the syntax of a non-broadcast instruction. A source from memory can be broadcast, that is, repeated, across all the elements of the effective source operand, up to 16 times for a 32-bit data element, and up to 8 times for a 64-bit data element, without using an additional source register. This is useful when we want to reuse the same scalar operand for all the operations in a vector instruction.

Embedded broadcast is only enabled on instructions with an element size of 32 or 64 bits; however, new FP16 instructions allow embedded broadcast. Please see Section 19.4.7, "FP16 Conversions to and from Other Data Types" for more information. In the case of older technologies, byte and word element broadcasts do not support embedded broadcast. Use a broadcast instruction, rather than embedded broadcast, to broadcast a byte or word.

Using embedded broadcast can reduce the number of registers used in the code, which may be helpful when register pressure exists.

In addition, when using embedded broadcast the load micro-op is in the same instruction as the operation micro-op, and therefore can benefit from micro fusion.

For example, replace the following code:

```
vbroadcastss zmm3, [rax]

vmulps zmm1, zmm2, zmm3
```

with:

```
vmulps zmm1, zmm2, [rax] {1to16}
```

The `{1to16}` primitive does the following:

1. Loads one float32 (single precision) element from memory.
2. Replicates it 16 times to form a vector of 16 32-bit floating point elements.

Intel AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

18.9.2 Broadcast Executed on Load Ports

In Skylake Server microarchitecture, a broadcast instruction with a memory operand of 32 bits or above is executed on the load ports; it is not executed on port 5 as other shuffles are. Alternative 2 in the following example shows how executing the broadcast on the load ports reduces the workload on port 5 and increases performance. Alternative 3 shows how embedded broadcast benefits from both executing the broadcast on the load ports and micro fusion.

Example 18-14. Broadcast Executed on Load Ports Alternatives

Alternative 1: 32-bit Load and Register Broadcast	Alternative 2: Broadcast with a 32-bit Memory Operand	Alternative 3: 32-bit Embedded Broadcast
loop: vmovd xmm0, [rax] vpbroadcastd zmm0, xmm0 vpaddd zmm2, zmm1, zmm0 vpermd zmm2, zmm3, zmm2 add rax, 0x4 sub rdx, 0x1 jnz loop	loop: vpbroadcastd zmm0, [rax] vpaddd zmm2, zmm1, zmm0 vpermd zmm2, zmm3, zmm2 add rax, 0x4 sub rdx, 0x1 jnz loop	loop: vpaddd zmm2, zmm1, [rax]{1to16} vpermd zmm2, zmm3, zmm2 add rax, 0x4 sub rdx, 0x1 jnz loop

Example 18-14. Broadcast Executed on Load Ports Alternatives (Contd.)

Alternative 1: 32-bit Load and Register Broadcast	Alternative 2: Broadcast with a 32-bit Memory Operand	Alternative 3: 32-bit Embedded Broadcast
Baseline 1x	Speedup: 1.57x	Speedup: 1.9x

The following example shows that on Skylake Server microarchitecture, 16-bit broadcast is executed on port 5 and therefore does not gain from the memory operand broadcast.

Example 18-15. 16-bit Broadcast Executed on Port 5

Alternative 1: 16-bit Load and Register Broadcast	Alternative 2: Broadcast with a 16-bit Memory Operand
loop: vmovd xmm0, [rax] vpbroadcastw zmm0, xmm0 vpaddw zmm2, zmm1, zmm0 vpermw zmm2, zmm3, zmm2 add rax, 0x4 sub rdx, 0x1 jnz loop	loop: vpbroadcastw zmm0, [rax] vpaddw zmm2, zmm1, zmm0 vpermw zmm2, zmm3, zmm2 add rax, 0x2 sub rdx, 0x1 jnz loop
Baseline 1x	Speedup: equal to baseline

Notice that embedded broadcast is not supported for 16-bit memory operands.

18.10 EMBEDDED ROUNDING

By default, the Rounding Mode is set by bits 13:14 of the MXCSR register.

Intel AVX-512 introduces a new instruction attribute called Static (per instruction) Rounding Mode (RM) or Rounding Mode override. This attribute allows a specific arithmetic rounding mode to be applied, ignoring the value of the RM bits in the MXCSR. In combination with the rounding-mode, Intel AVX-512 also has an SAE ("suppress-all-exceptions") attribute, to disable reporting any floating-point exception flag in the MXCSR. SAE is always implied when rounding-mode is enabled.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In this case, vector length is assumed to be the maximal possible vector length (512-bit in case of Intel AVX-512). The table below summarizes the possible static rounding-mode assignments in Intel AVX-512. Note that some instructions already allow the rounding mode to be statically specified via immediate bits. In such case, the immediate bits take precedence over the embedded rounding mode in the same way as they take precedence over the bits in MXCSR.RM

18.10.1 Static Rounding Mode

Static rounding mode functions and descriptions are listed below.

Table 18-2. Static Rounding Mode Functions

Function	Description
{rn-sae}	Round to nearest (even) + SAE
{rd-sae}	Round down (toward -infinity) + SAE
{ru-sae}	Round up (toward +infinity) + SAE
{rz-sae}	Round toward zero (Truncate) + SAE

The following code snippet shows a usage example.

Example 18-16. Embedded vs Non-embedded Rounding

Using Embedded Rounding	Without Embedded Rounding
vaddps zmm7 {k6}, zmm2, zmm4, {ru-sae}	;rax & rcx point to temporary dword values in memory used to load and save (for restoring) MXCSR value vstmxcsr [rax] ;load mxcsr value to memory mov ebx, [rax] ;move to register and ebx, 0xFFFF9FFF ;zero RM bits or ebx, 0x5F80 ;put {ru} to RM bits and suppress all exceptions mov [rcx], ebx ;move new value to the memory vldmxcsr [rcx] ;save to MXCSR vaddps zmm7 {k6}, zmm2, zmm4 ;operation itself vldmxcsr [rax] ;restore previous MXCSR value

This piece of code would perform the single-precision floating point addition of vectors zmm2 and zmm4 with round-towards-plus-infinity, leaving the result in vector zmm7 using k6 as a conditional writemask. Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

The following are examples of instructions instances where the static rounding-mode is not allowed.

```
; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2 {rd}, 0x00


; instructions with memory operands
vmulps zmm7 {k6}, zmm2, [rax] {rd}


; instructions with vector length different than maximal vector length (512-bit)
vaddps ymm7 {k6}, ymm2, ymm4 {rd}


; non-floating point instructions
vpaddd zmm7 {k6}, zmm2, zmm4 {rd}
```

18.11 SCATTER INSTRUCTION

This instruction performs a non-continuous store of data (scatter). Given a base address, a set of signed offsets and a data item, the instruction writes each element in the data register to the memory location computed from the base address and corresponding offset. The instruction stores up to 16 elements (8 elements for qword indices) in a doubleword vector or 8 elements in a quadword vector, to the memory locations pointed to by the base address and index vector. Elements are stored only if their corresponding mask bit is one. The figure below describes the following operation.

```
vscatterdpd   [rax + zmm0]{k1} , zmm1
```

In this example, `rax` contains the base address, `zmm0` contains a set of offsets, and `zmm1` contains data to be written.

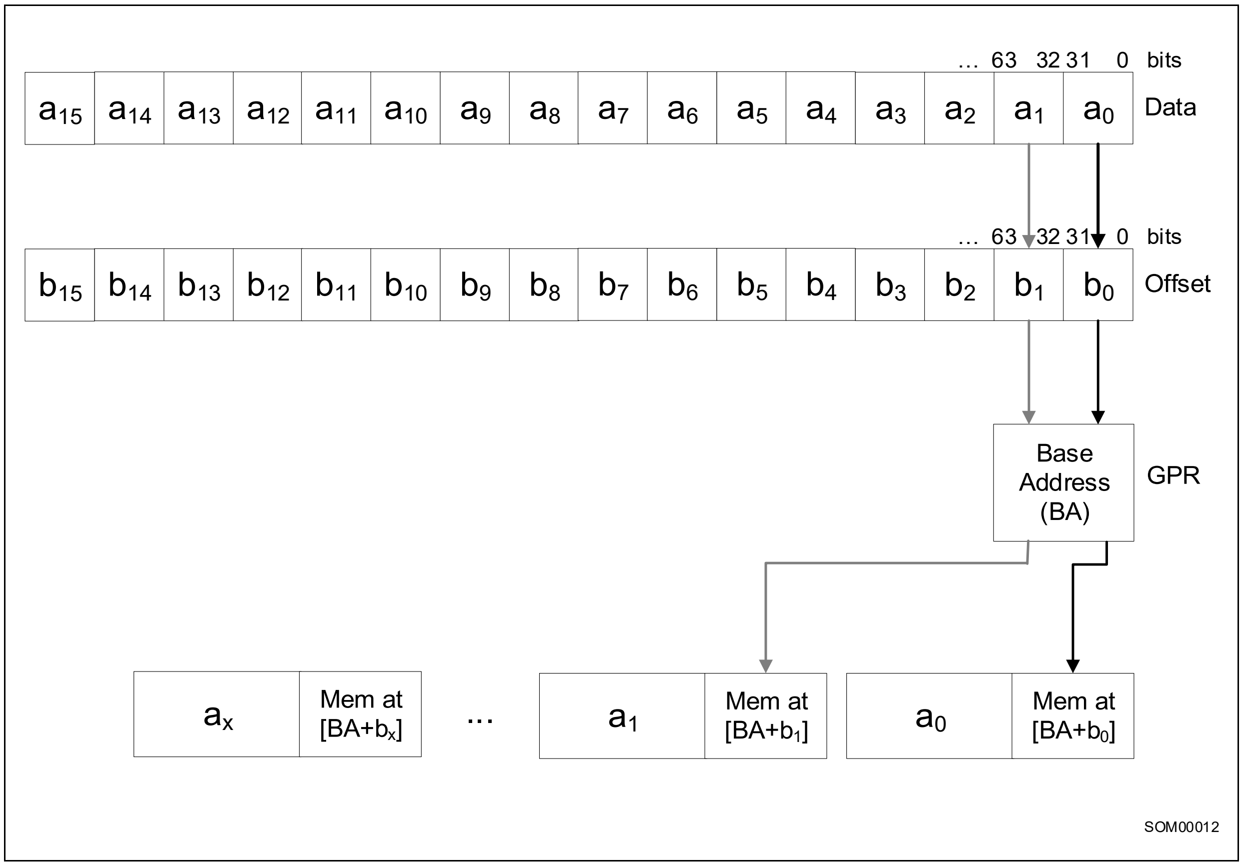


Figure 18-9. VSCATTERDPD Instruction Operation

18.11.1 Data Scatter Example

Given an array of unique indexes, ranging from 0 to N, we want to sort the array of N values, according to the corresponding index, while converting the values from long long integers (64 bits) to floating point numbers (32 bits).

```
for ( int i=0; i < N; i++ )
{
      dst[ ind [i] ] = (float)src[i];
}
```

Here are three implementations of the code above.

- Alternative 1 is pure scalar code.
- Alternative 2 is a software sequence for scatter.
- Alternative 3 is a hardware scatter.

NOTE

A hardware Scatter operation issues as many store operations, as the number of elements in the vector. Do not use a scatter operation to store sequential elements, which can be stored with one vmov instruction.

.

Example 18-17. Scatter

Scalar	
```
    mov rax, pImage    //input
    mov rcx, pOutImage //output
    mov rbx, pIndex    //indexes
    mov rdx, len       //length
    xor r9, r9
mainloop:
    mov r9d, [rbx+rdx-0x4]
    vcvtsi2ss xmm0, xmm0, qword ptr [rax+rdx*2-0x8]
    vmovss [rcx+r9*4], xmm0
    sub rdx, 4
    jnz mainloop
```
Baseline 1x	
Software Sequence	**Hardware Scatter**

Example 18-17. Scatter

```
shufMaskP:
    .quad 0x0000000200000001
    .quad 0x0000000400000003
    .quad 0x0000000600000005
    .quad 0x0000000800000007

mov rax, pImage   //input
   mov rcx, pOutImage //output
   mov rbx, pIndex   //indexes
   mov rdx, len      //length
   mov r9, shufMaskP
   vmovaps ymm2, [r9]
mainloop:
   vmovaps zmm1, [rax + rdx*2 - 0x80] //load data
   vcvtuqq2ps  ymm0, zmm1  //convert to float
   movsxd r9, [rbx + rdx - 0x40]  //load 8th index
   vmovss [rcx + 4*r9], xmm0
   vpermd ymm0, ymm2, ymm0
   movsxd r9, [rbx + rdx - 0x3c]  //load 7th index
   vmovss [rcx + 4*r9], xmm0
   vpermd ymm0, ymm2, ymm0
   movsxd r9, [rbx + rdx - 0x38]  //load 6th index
   vmovss [rcx + 4*r9], xmm0
   vpermd ymm0, ymm2, ymm0
   movsxd r9, [rbx + rdx - 0x34]  //load 5th index
   vmovss [rcx + 4*r9], xmm0
   vpermd ymm0, ymm2, ymm0
   movsxd r9, [rbx + rdx - 0x30]  //load 4th index
   vmovss [rcx + 4*r9], xmm0
   vpermd ymm0, ymm2, ymm0
   movsxd r9, [rbx + rdx - 0x2c]  //load 3rd index
   vmovss [rcx + 4*r9], xmm0
   vpermd ymm0, ymm2, ymm0
```

```
   mov rax, pImage   //input
   mov rcx, pOutImage //output
   mov rbx, pIndex   //indexes
   mov rdx, len      //length
mainloop:
   vmovdqa32 zmm0, [rbx+rdx-0x40]
   vmovdqa32   zmm1, [rax+rdx*2-0x80]
   vcvtuqq2ps  ymm1, zmm1
  vmovdqa32   zmm2, [rax+rdx*2-0x40]
   vcvtuqq2ps  ymm2, zmm2
  vshuff32x4 zmm1, zmm1, zmm2, 0x44
   kxnorw     k1,k1,k1
   vscatterdps [rcx+4*zmm0] {k1}, zmm1
   sub rdx, 0x40
   jnz mainloop
```

Example 18-17. Scatter

movsxd r9, [rbx + rdx - 0x28] //load 2nd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x24] //load 1st index vmovss [rcx + 4*r9], xmm0 vmovaps zmm1, [rax + rdx*2 - 0x40] //load data vcvtuqq2ps ymm0, zmm1 //convert to float movsxd r9, [rbx + rdx - 0x20] //load 8th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x1c] //load 7th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x18] //load 6th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x14] //load 5th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x10] //load 4th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0xc] //load 3rd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x8] //load 2nd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x4] //load 1st index vmovss [rcx + 4*r9], xmm0 sub rdx, 0x40 jnz mainloop	
Speedup: 1.48x	Speedup: 1.53x

18.12 STATIC ROUNDING MODES, SUPPRESS-ALL-EXCEPTIONS (SAE)

The Suppress-all-exceptions (SAE) feature was added to Intel AVX-512 floating-point instructions. This feature is helpful when spurious flag settings are undesirable. Although current implementations of vector math functions usually allow spurious flag settings, they can cause problems for applications that run with exceptions enabled. Standard-compliant code does not allow spurious flag settings.

In addition to standard-mandated uses (IEEE, OpenCL), static rounding modes have applications in math libraries that operate under the default rounding mode (which can be dynamically set).

18.13 QWORD INSTRUCTION SUPPORT

Intel AVX-512 extends QWORD support to many instructions introduced in Intel AVX and Intel AVX2. QWORD support was added to the instructions as detailed in the following sections.

18.13.1 QUADWORD Support in Arithmetic Instructions

Intel AVX-512 adds new quadword extension to *vpmaxsq, vpmaxuq, vpminsq, vpminuq, and vpmullq*.

The following example will store to array c the max value between the sum and the multiply of two 64bit numbers.

```
const int N = miBufferWidth;
const __int64* restrict a = A;
const __int64* restrict b = B;
__int64* restrict c = Cref;


for (int i = 0; i < N; i++){
          __int64 sum = a[i] + b[i];
          __int64 mul = a[i] * b[i];
          c[i] = mul > sum ? mul : sum;
}
```

The code below shows how the new support reduces instruction count from 118 in Intel AVX2 to 30 in Intel AVX-512 and results in a 3.1x speedup.

Example 18-18. QWORD Example, Intel® AVX2 vs. Intel® AVX-512

Intel® AVX2 Intrinsics	Intel® AVX-512 Intrinsics

for (int i = 0; i < N; i+= 32){
__m256i aa, bb, aah, bbh, mul, sum;
#pragma unroll(8)
for (int j = 0; j < 8; j++){
aa = _mm256_loadu_si256((const __m256i*)(a+i+4*j));
bb = _mm256_loadu_si256((const __m256i*)(b+i+4*j));
sum = _mm256_add_epi64(aa, bb);
mul = _mm256_mul_epu32(aa, bb);
aah = _mm256_srli_epi64(aa, 32);
bbh = _mm256_srli_epi64(bb, 32);
aah = _mm256_mul_epu32(aah, bb);
bbh = _mm256_mul_epu32(bbh, aa);
aah = _mm256_add_epi32(aah, bbh);
aah = _mm256_slli_epi64(aah, 32);
mul = _mm256_add_epi64(mul, aah);
aah = _mm256_cmpgt_epi64(mul, sum);
aa = _mm256_castpd_si256 (_mm256_blendv_pd(_mm256_castsi256_pd (sum), _mm256_castsi256_pd(mul), _mm256_castsi256_pd(aah)));
_mm256_storeu_si256((__m256i*)(c+4*j), aa);
}
c += 32;
}
	for (int i = 0; i < N; i+= 32){
	__m512i aa, bb, mul, sum;
	#pragma unroll(4)
	for (int j = 0; j < 4; j++){
	aa = _mm512_loadu_si512((const __m512i*)(a+i+8*j));
	bb = _mm512_loadu_si512((const __m512i*)(b+i+8*j));
	sum = _mm512_add_epi64(aa, bb);
	mul = _mm512_mullo_epi64(aa, bb);
	aa = _mm512_max_epi64(sum, mul);
	_mm512_storeu_si512((__m512i*)(c+8*j), aa);
	
	}
	
	c += 32;
	}

Baseline 1x	Speedup: 3.1x

Example 18-18. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 (Contd.)

Intel® AVX2 Assembly	Intel® AVX-512 Assembly
loop: vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0x20] inc r9d vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0x20] vmovdqu32 ymm17, ymmword ptr [r11+rcx*8] vmovdqu32 ymm19, ymmword ptr [rax+rcx*8] vmovdqu ymm13, ymmword ptr [rax+rcx*8+0x40] vmovdqu ymm11, ymmword ptr [r11+rcx*8+0x40] vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm27, ymm26, 0x20 vpsrlq ymm16, ymm19, 0x20 vpsrlq ymm18, ymm17, 0x20 vpaddq ymm6, ymm28, ymm26 vpsrlq ymm10, ymm13, 0x20 vpsrlq ymm12, ymm11, 0x20 vpaddq ymm0, ymm19, ymm17 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpaddd ymm31, ymm29, ymm30 vmovdqu32 ymm29, ymmword ptr [r11+rcx*8+0x80] vpsllq ymm5, ymm31, 0x20 vmovdqu32 ymm31, ymmword ptr [rax+rcx*8+0x80] vpsrlq ymm30, ymm29, 0x20 vpmuludq ymm20, ymm16, ymm17 vpmuludq ymm21, ymm18, ymm19 vpmuludq ymm4, ymm28, ymm26 vpaddd ymm22, ymm20, ymm21 vpaddq ymm7, ymm4, ymm5 vpsrlq ymm28, ymm31, 0x20 vmovdqu32 ymm20, ymmword ptr [r11+rcx*8+0x60] vpsllq ymm24, ymm22, 0x20 vmovdqu32 ymm22, ymmword ptr [rax+rcx*8+0x60] vpsrlq ymm21, ymm20, 0x20 vpaddq ymm4, ymm22, ymm20 vpcmpgtq ymm8, ymm7, ymm6 vblendvpd ymm9, ymm6, ymm7, ymm8 vmovups ymmword ptr [rsi+0x20], ymm9 vpmuludq ymm14, ymm10, ymm11 vpmuludq ymm15, ymm12, ymm13 vpmuludq ymm8, ymm28, ymm29 vpmuludq ymm9, ymm30, ymm31 vpmuludq ymm23, ymm19, ymm17 vpaddd ymm16, ymm14, ymm15 vpsrlq ymm19, ymm22, 0x20 vpaddd ymm10, ymm8, ymm9 vpaddq ymm1, ymm23, ymm24	loop: vmovups zmm0, zmmword ptr [rax+rcx*8] inc r9d vmovups zmm5, zmmword ptr [rax+rcx*8+0x40] vmovups zmm10, zmmword ptr [rax+rcx*8+0x80] vmovups zmm15, zmmword ptr [rax+rcx*8+0xc0] vmovups zmm1, zmmword ptr [r11+rcx*8] vmovups zmm6, zmmword ptr [r11+rcx*8+0x40] vmovups zmm11, zmmword ptr [r11+rcx*8+0x80] vmovups zmm16, zmmword ptr [r11+rcx*8+0xc0] vpaddq zmm2, zmm0, zmm1 vpmullq zmm3, zmm0, zmm1 vpaddq zmm7, zmm5, zmm6 vpmullq zmm8, zmm5, zmm6 vpaddq zmm12, zmm10, zmm11 vpmullq zmm13, zmm10, zmm11 vpaddq zmm17, zmm15, zmm16 vpmullq zmm18, zmm15, zmm16 vpmaxsq zmm4, zmm2, zmm3 vpmaxsq zmm9, zmm7, zmm8 vpmaxsq zmm14, zmm12, zmm13 vpmaxsq zmm19, zmm17, zmm18 vmovups zmmword ptr [rsi], zmm4 vmovups zmmword ptr [rsi+0x40], zmm9 vmovups zmmword ptr [rsi+0x80], zmm14 vmovups zmmword ptr [rsi+0xc0], zmm19 add rcx, 0x20 add rsi, 0x100 cmp r9d, r8d jb loop

Example 18-18. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 (Contd.)

Intel® AVX2 Assembly	Intel® AVX-512 Assembly
vpsllq ymm18, ymm16, 0x20 vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0xc0] vpsllq ymm12, ymm10, 0x20 vpmuludq ymm23, ymm19, ymm20 vpmuludq ymm24, ymm21, ymm22 vpaddd ymm25, ymm23, ymm24 vmovdqu32 ymm19, ymmword ptr [rax+rcx*8+0xa0] vpsllq ymm27, ymm25, 0x20 vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm16, ymm19, 0x20 vpcmpgtq ymm2, ymm1, ymm0 vblendvpd ymm3, ymm0, ymm1, ymm2 vpaddq ymm0, ymm13, ymm11 vmovups ymmword ptr [rsi], ymm3 vpmuludq ymm17, ymm13, ymm11 vpmuludq ymm11, ymm31, ymm29 vpaddq ymm1, ymm17, ymm18 vpaddq ymm13, ymm31, ymm29 vpaddq ymm14, ymm11, ymm12 vmovdqu32 ymm17, ymmword ptr [r11+rcx*8+0xa0] vmovdqu ymm12, ymmword ptr [r11+rcx*8+0xe0] vpsrlq ymm18, ymm17, 0x20 vpcmpgtq ymm2, ymm1, ymm0 vpmuludq ymm26, ymm22, ymm20 vpcmpgtq ymm15, ymm14, ymm13 vblendvpd ymm3, ymm0, ymm1, ymm2 vblendvpd ymm0, ymm13, ymm14, ymm15 vmovdqu ymm14, ymmword ptr [rax+rcx*8+0xe0] vmovups ymmword ptr [rsi+0x40], ymm3 vmovups ymmword ptr [rsi+0x80], ymm0 vpaddq ymm5, ymm26, ymm27 vpsrlq ymm11, ymm14, 0x20 vpsrlq ymm13, ymm12, 0x20 vpaddq ymm1, ymm19, ymm17 vpaddq ymm0, ymm14, ymm12 vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0xc0] vpmuludq ymm20, ymm16, ymm17 add rcx, 0x20 vpmuludq ymm21, ymm18, ymm19 vpaddd ymm22, ymm20, ymm21 vpsrlq ymm27, ymm26, 0x20 vpsllq ymm24, ymm22, 0x20 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpmuludq ymm15, ymm11, ymm12 vpmuludq ymm16, ymm13, ymm14 vpmuludq ymm23, ymm19, ymm17 vpaddd ymm31, ymm29, ymm30 vpaddd ymm17, ymm15, ymm16	

Example 18-18. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 (Contd.)

vpaddq ymm2, ymm23, ymm24 vpsllq ymm19, ymm17, 0x20 vpcmpgtq ymm6, ymm5, ymm4 vblendvpd ymm7, ymm4, ymm5, ymm6 vpsllq ymm6, ymm31, 0x20 vmovups ymmword ptr [rsi+0x60], ymm7 vpaddq ymm7, ymm28, ymm26 vpcmpgtq ymm3, ymm2, ymm1 vpmuludq ymm5, ymm28, ymm26 vpmuludq ymm18, ymm14, ymm12 vblendvpd ymm4, ymm1, ymm2, ymm3 vpaddq ymm8, ymm5, ymm6 vpaddq ymm1, ymm18, ymm19 vmovups ymmword ptr [rsi+0xa0], ymm4 vpcmpgtq ymm9, ymm8, ymm7 vpcmpgtq ymm2, ymm1, ymm0 vblendvpd ymm10, ymm7, ymm8, ymm9 vblendvpd ymm3, ymm0, ymm1, ymm2 vmovups ymmword ptr [rsi+0xc0], ymm10 vmovups ymmword ptr [rsi+0xe0], ymm3 add rsi, 0x100 cmp r9d, r8d jb loop	
Baseline 1x	Speedup: 3.1x

18.13.2 QUADWORD Support in Convert Instructions

The following tables demonstrate the new quadword extension in convert instructions.

Table 18-3. Vector Quadword Extensions

From / To	Vector SP	Vector DP	Vector int64	Vector uint64
Vector SP	-		vcvtps2qq	vcvtps2uqq
Vector DP		-	vcvtpd2qq	vcvtpd2qq
Vector int64	vcvtqq2ps	vcvtqq2pd	-	
Vector uint64	vcvtqq2ps	vcvtuqq2pd		-

Table 18-4. Scalar Quadword Extensions

From / To	Scalar SP	Scalar DP	Scalar int64	Scalar uint64
Scalar SP	-		vcvtss2si	vcvtss2usi
Scalar DP		-	vcvtsd2si	vcvtsd2usi
Scalar int64	vcvtsi2sd	vcvtsi2sd	-	
Scalar uint64	vcvtusi2sd	vcvtusi2sd		-

18.13.3 QUADWORD Support for Convert with Truncation Instructions

The following tables demonstrate the new quadword extension in convert with truncate instructions.

Table 18-5. Vector Quadword Extensions

From / To	Vector int64	Vector uint64
Vector SP	vcvttps2qq	vcvttps2uqq
Vector DP	vcvttpd2qq	vcvttpd2qq

Table 18-6. Scalar Quadword Extensions

From / To	Scalar int64	Scalar uint64
Scalar SP	vcvttss2si	vcvttss2usi
Scalar DP	vcvttsd2si	vcvttsd2usi

18.14 VECTOR LENGTH ORTHOGONALITY

All Intel AVX-512 instructions, in processors that support Vector Length Extensions (VL), can operate at three vector lengths: 128-bit, 256-bit and 512-bit. All of these vector lengths are supported by all Intel AVX-512 instructions, except instructions with Embedded Rounding.

In the instruction encoding, the same two bits are used for encoding vector length and embedded rounding control, therefore when embedded rounding is used, the vector length is automatically assumed to be 512 bits (maximum vector length in Intel AVX-512).

See also Section 18.10, “Embedded Rounding”.

18.15 INTEL® AVX-512 INSTRUCTIONS FOR TRANSCENDENTAL SUPPORT

This section lists and describes the new instructions introduced by Intel AVX-512 for transcendental support.

18.15.1 VRCP14, VRSQRT14 - Software Sequences for 1/x, x/y, sqrt(x)

Syntax:

VRCP14PD/PS dest, src

VRSQRT14PD/PS dest, src

18.15.1.1 Application Examples

There are software sequences for Reciprocal, Division, Square Root, and Inverse Square Root instructions.

Software sequences for 1/x, x/y, sqrt(x) are beneficial for throughput (not so much for latency, unless the accuracy is quite low). They are typically implemented via Newton-Raphson approximations, or polynomial approximations.

One advantage of VRCP14 and VRSQRT14 is the improved accuracy, compared with the legacy RCPPS, RSQRTPS. This helps shorten the computation, in particular for double precision (which requires two instead of three Newton-Raphson iterations for a 50-52 bit approximation).

Another advantage of these instructions is that they have double-precision versions (while the legacy RCP/RSQRT instructions did not). This further boosts double-precision performance. On Skylake Server microarchitecture, double precision reciprocal and square root software sequences have significantly better throughput than the VDIV and VSQRT instructions in 512-bit vector mode Double Precision Transcendental Argument Reductions (e.g., log, cbrt).

In functions such as log() or the cube root (cbrt), a rounded VRCP14PD result can be used in place of an expensive reciprocal table lookup. The same technique could be used before via RCPPS, but was less efficient for double-precision.

See Section 18.15.3, "VRNDSCALE - Vector Round Scale" for a log() argument reduction example.

18.15.2 VGETMANT VGETEXP - Vector Get Mantissa and Vector Get Exponent

Syntax:

VGETMANTPD/PS dest_mant, src, imm

VGETEXPPD/PS dest_exp, src

18.15.2.1 Application Examples

Logarithm Function

```
log2(x) = VGETEXP(x) + log2(VGETMANT(x,8))
```

```
log(x) = VGETEXP(x)*log(2.0) + log(VGETMANT(x,8))
```

As seen above, the computation is reduced to computing log(VGETMANT(x,8)), where VGETMANT(x,8) is guaranteed to be in [1,2) for all valid function inputs, and NaN for invalid inputs (x<0).

A variety of algorithms can be applied to compute the logarithm of the mantissa. The selection of a particular algorithm may depend on the desired accuracy, on optimization goals (latency or throughput optimized), or on specifics of the microarchitecture. Some algorithms may use other normalization options for the mantissa: [0.5, 1) or [0.75, 1.5); however, the basic identity underlying the computation is shown above.

See Section 18.15.5, "VSCALEF - Vector Scale" for details on X^{alpha} (constant alpha) and division.

18.15.3 VRNDSCALE - Vector Round Scale

Syntax:

VRNDSCALEPD/PS dest, src, imm

18.15.3.1 Application Examples

Lookup tables are frequently used in transcendental function implementations. The table index is most often based on a few leading bits of the input. VRNDSCALE can be used as part of the argument reduction process, to form the floating-point input value corresponding to the table index. The following example implements the argument reduction for log(x), where $1 \leq x < 2$:

```
y = RCP14(x);             // y is in (0.5, 1]

y0=RNDSCALE(y, k*16);      // y0 has k mantissa bits (leading 1

                                                              // included)

R = x?y0 - 1;             // |R|<2-14+2-k.
```

Therefore log(x) = -log(y0) + log(1+R).

log(1+R)can be computed via a polynomial, and log(y0) can be retrieved from a lookup table of 2k-1+1 elements, or 2k-1 elements, at the expense of an additional check.

18.15.4 VREDUCE - Vector Reduce

Syntax:

VREDUCEPD/PS dest, src, imm

18.15.4.1 Application Examples

The most significant benefit of VREDUCE is latency reduction in common transcendental operations such as exp2 and pow (which includes an exp2 operation). Uses in other transcendental functions such as atan() are also possible.

See Section 18.15.5, "VSCALEF - Vector Scale".

18.15.5 VSCALEF - Vector Scale

Syntax:

VSCALEFPD/PS dest, src1, src2

18.15.5.1 Application Examples

```
exp2 (2x)
```

```
exp2(x) = VSCALEF( 2VREDUCE(x, RD_mode), x)
```

R(x) = VREDUCE(x, RD_mode) = x - floor(x) is in [0, 1). 2R(x) is computed by other means, such as polynomial approximation, or table lookup with polynomial approximation. VSCALEF correctly handles overflow and underflow. It is also defined to handle exp() special cases correctly (such as when the input is an Infinity), so there is no need for special paths in a vector implementation. In the absence of VSCALEF, inputs that are very large in magnitude require a separate path.

Since explicit exponent manipulation is no longer needed, VSCALEF also helps improve throughput.

```
Exp(x)
```

```
Exp(x) = VSCALEF( 2R(x), x*(1/log(2.0)),
```

where,

R(x) = x - log(2.0)*floor(x*(1/log(2.0));

R(x) is accurately computed by using a sufficiently long log(2.0) approximation (longer than the native floating-point format).

As with exp2(), the advantages of using VSCALEF are better throughput and elimination of secondary branches.

x^{alpha} (constant alpha)

For example, alpha=1/3 (the cube root function, cbrt).

The basic reduction for this computation is:

x^{alpha} = VSCALEF((VGETMANT(x, imm))alpha?2VREDUCE (VGETEXP(x)*alpha, RD_mode), VGETEXP(x)*alpha)

selecting the immediate (imm) is based on the value of the alpha constant.

Division:

```
a/b = VSCALEF(VGETMANT(a,0)/VGETMANT(b,0), VGETEXP(a)-VGETEXP(b))
```

This reduction allows for a branch-free implementation of divide, that covers overflow, underflow, and special inputs (zeroes, Infinities, or denormals).

|VGETMANT(x,0)| is in [1,2) for all non-NaN inputs.

VGETMANT(a,0)/VGETMANT(b,0) can be computed to the desired accuracy.

The suppress-all-exceptions (SAE) feature available in Intel AVX-512 can help ensure spurious flag settings do not occur. Flags can be set correctly as part of the computation (except for divide-by-zero, which requires an additional step).

For high accuracy or IEEE compliance, the hardware instruction typically provides better performance, especially in terms of latency.

18.15.6 VFPCLASS - Vector Floating Point Class

Syntax:

VFPCLASSPD/PS dest_mask, src, imm

18.15.6.1 Application Examples

The VFPCLASS instruction is used to detect special cases so they can be directed to a special path, or alternatively, handled with masked operations in the main path. See two examples below.

Reciprocal Sequence, Square Root Sequence:

The reduced argument for the 1/x computation is e=1-x*RCP14(x). This expression evaluates to NaN when x is ±0 or ±Inf, as RCP14 returns the correct result for these special cases. VFPCLASS enables you to set mask=1 for x=±0 or ±Inf, and mask=0 for all other x. This mask can then be used to select between the RCP14 output (result for special cases), or the result of a reciprocal refinement computation starting with RCP14 (for typical inputs).

In a similar manner, a square root computation based on RSQRT14 can use the VFPCLASS instruction to create a mask for =±0 or x=+Inf.

Pow(x,y) function:

The main path of pow(x,y)=2y*log2(x) does not operate on x?0, x=Inf/NaN, or y=Inf/NaN. One VFPCLASS op can be used to set special_x_mask=1 for x?0 or x=Inf/NaN. A second VFPCLASS op would be used to set special_y_mask=1 for y=Inf/NaN. A branch to a secondary path is taken if either mask is set.

18.15.7 VPERM, VPERMI2, VPERMT2 - Small Table Lookup Implementation

18.15.7.1 Application Examples

Math library functions are frequently implemented using table lookups. In vector mode, large table lookups would use vector gather. Small table lookups can be implemented via the VPERM* instructions, which are significantly faster.

Examples of common transcendental functions that achieved very significant speedup using VPERM* for table lookups: exp(), log(), pow() - both single and double precision.

18.16 CONFLICT DETECTION

The Intel AVX-512 Conflict Detection instructions are instructions that, together with Intel AVX-512 Foundation instructions, enable efficient vectorization of loops with possible vector dependencies (i.e., conflicts) through memory. VPCONFLICT performs horizontal comparisons of elements within a single vector register. VPCONFLICT compares each element of a vector register with all previous elements in that register, and outputs the results of all of the comparisons. These horizontal comparisons can be used for other purposes.

Other conflict detection instructions allow for efficient manipulation of the comparison results. The VPLZCNT instruction lets us generate controls for in-register permute operations used to combine vector elements with matching values.

18.16.1 Vectorization with Conflict Detection

The Intel AVX-512CD instructions allow efficient vectorization of loops with reads and writes through an array of pointers (e.g., *ptr[i] += val[i]) or an indirectly addressed array (e.g., A[B[i]] += val[i]).

Consider the following histogram computation:

```
for(int i = 0; i < num_inputs; i++)
   {
          histogram[input[i] & (num_bins - 1)]++;
    }
```

If input[0] = input[1] = 3, we will get an incorrect answer if we use SIMD instructions to read histogram[input[0]] and histogram[input[1]] into a register (with a gather), increment them, and then write them back (with a scatter). After this sequence, the value in histogram[3] will be 1, when it should be 2.

The problem occurs because we have duplicate indices; this creates a dependence between the write to the histogram in iteration 0 and the read from the histogram in iteration 1 - the read should get the value of the previous write.

To detect this scenario, look for duplicate indices (or pointer values), using the VPCONFLICT instruction. This instruction compares each element of a vector register with all previous elements in that register.

Example:

```
vpconflictd zmm0, zmm1
```

The figure below is an example that shows the execution of a VPCONFLICTD instruction. The input, ZMM1, contains 16 integers, shown in the light grey boxes. ZMM1 is at the top of the figure, and also visually transposed along the left-hand side of the figure. The white boxes show the equality comparisons that the hardware performs between different elements of ZMM1, and the outcome of each comparison (0 = not equal, 1 = equal). Each comparison output is a single bit in the output of the instruction. Comparisons that are not performed (i.e., the dark grey boxes) produce a single '0' bit in the output. Finally, the output register, ZMM0, is shown at the bottom of the figure. Each element is shown as a decimal representation of the bits above it.

Use VPCONFLICT in different ways to help vectorize loops.

The simplest option is to check for any duplicate indices in a given SIMD register. If there are none, SIMD instructions can be used to compute all elements simultaneously. If conflicts are present, execute a scalar loop for that group of elements.

Branching to a scalar version of the loop on any duplicate indices can work well if duplicates are extremely rare. However, if the chance of getting even one duplicate in a given iteration of the vectorized loop is large enough, then it is better to use SIMD as much as possible, to exploit as much parallelism as possible.

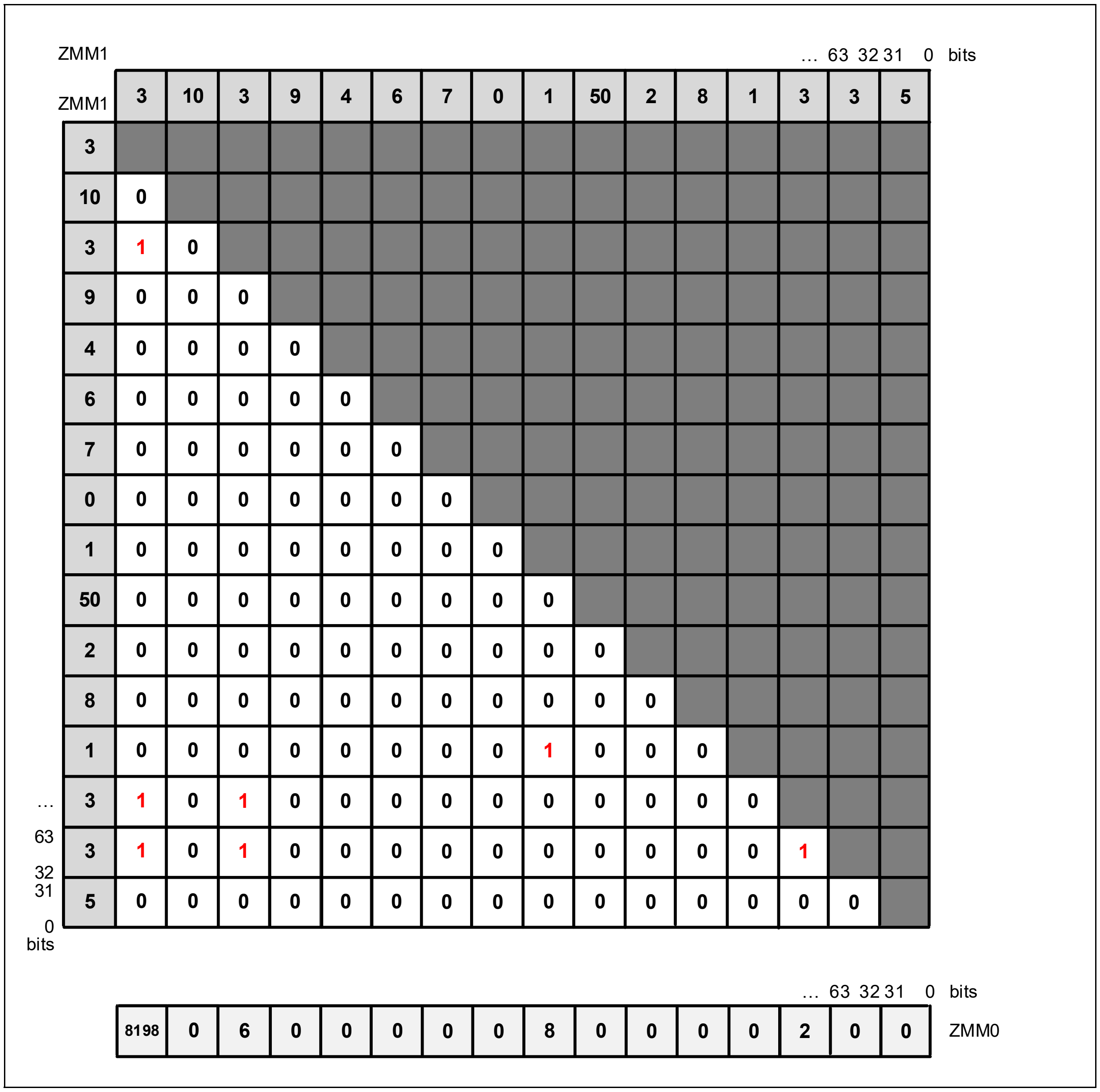


Figure 18-10. VPCONFLICTD Instruction Execution

For loops performing updates to memory locations, such as in the histogram example, minimize store-load forwarding by merging the updates to each distinct index while the data is in registers, and only perform a single write to each memory location. Further, the merge can be performed in a parallel fashion.

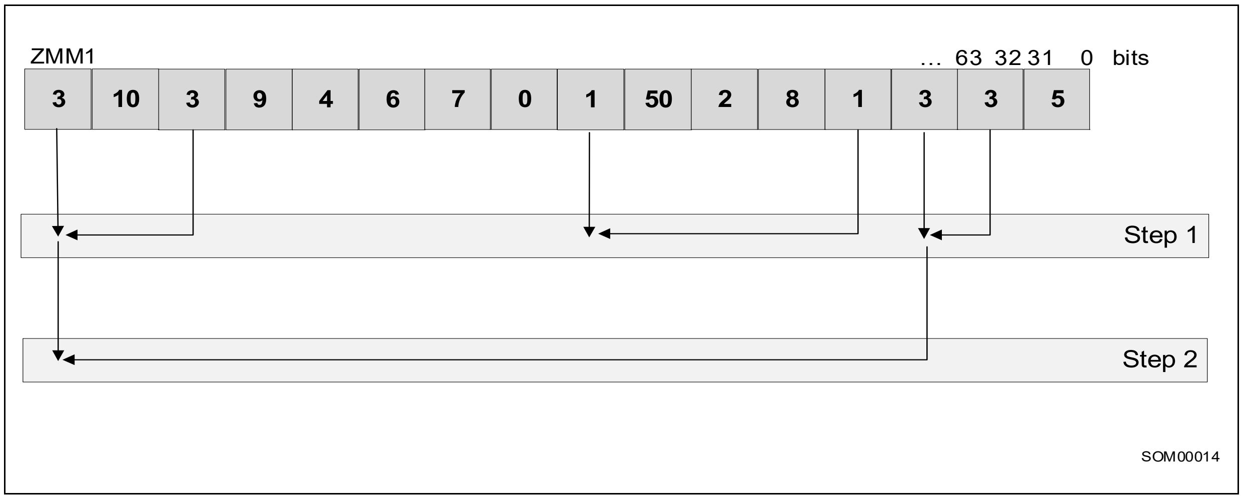


Figure 18-11. VPCONFLICTD Merging Process

The figure above shows the merging process for the example set of indices. While the figure shows only the indices, it actually merges the values. Most of the indices are unique, and thus require no merging. Step 1 combines three pairs of indices: two pairs of '3's and one pair of '1's. Step 2 combines the intermediate results for the '3's from step 1, so that there is now a single value for each distinct index. Notice that in only two steps, the four elements with an index value of 3 are merged, because we performed a tree reduction; we merged pairs of results or intermediate results at each step.

The merging (combining or reduction) process shown above is done with a set of permute operations. The initial permute control is generated with a VPLZCNT+VPSUB sequence. VPLZCNT provides the number of leading zeros for each vector element (i.e., contiguous zeros in the most significant bit positions). Subtracting the results of VPLZCNT from the number of bits in each vector element, minus one, provides the bit position of the most significant '1' bit in the result of the VPCONFLICT instruction, or results in a '-1' for an element if it has no conflicts. In the example above this sequence results in the following permute control.

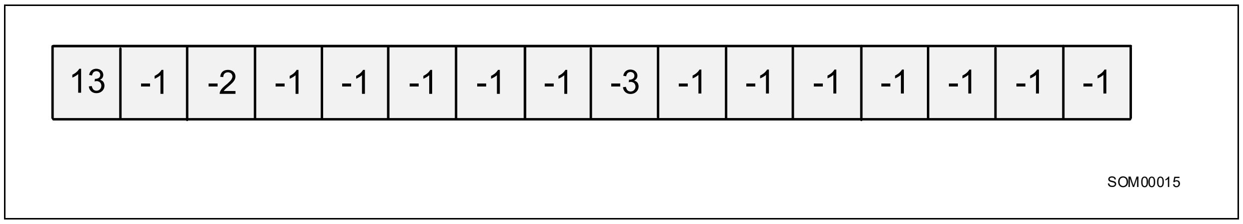


Figure 18-12. VPCONFLICTD Permute Control

The permute loop for merging matching indices and generating the next set of permute indices repeats until all values in the permute control become equal to '-1'.

The assembly code below shows both the scalar version of a histogram loop, and the vectorized version with a tree reduction. Speedups are modest because the loop contains little computation; the SIMD benefit comes almost entirely from vectorizing just the logical AND operation and the increment. SIMD speedups can be much higher for loops containing more vectorizable computation.

Example 18-19. Scatter Implementation Alternatives

Scalar Code (Unrolled Two Times)	Intel® AVX-512 Code

```
    mov r9d, bins_minus_1
    mov ebx, num_inputs
    mov r10, pInput
    mov r15, pHistogram
    xor rax, rax
histogram_loop:
    lea ecx, [rax + rax]
    inc eax
    movsxd rcx, ecx
    mov esi, [r10+rcx*4]
    and esi, r9d
    mov r8d, [r10+rcx*4+4]
    movsxd rsi, esi
    and r8d, r9d
    movsxd r8, r8d
    inc dword ptr [r15+rsi*4]
    inc dword ptr [r15+r8*4]
    cmp eax, ebx
    jb histogram_loop
```
	```
	    vmovaps zmm4, all_1 // {1, 1, ..., 1}
	    vmovaps zmm5, all_negative_1
	    vmovaps zmm6, all_31
	    vmovaps zmm7, all_bins_minus_1
	    mov ebx, num_inputs
	    mov r10, pInput
	    mov r15, pHistogram
	    xor rcx, rcx
	histogram_loop:
	    vpandd zmm3, zmm7, [r10+rcx*4]
	    vpconflictd zmm0, zmm3
	    kxnorw k1, k1, k1
	    vmovaps zmm2, zmm4
	    vpxord zmm1, zmm1, zmm1
	    vpgatherdd zmm1{k1}, [r15+zmm3*4]
	    vptestmd  k1, zmm0, zmm0
	    kortestw  k1, k1
	    je update

	    vplzcntd zmm0, zmm0
	    vpsubd zmm0, zmm6, zmm0

	conflict_loop:
	    vpermd zmm8{k1}{z}, zmm0, zmm2
	    vpermd zmm0{k1}, zmm0, zmm0
	    vpaddd zmm2{k1}, zmm2, zmm8
	    vpcmpned k1, zmm5, zmm0
	    kortestw  k1, k1
	    jne conflict_loop

	update:
	    vpaddd zmm0, zmm2, zmm1
	    kxnorw k1, k1, k1
	    add rcx, 16
	    vpscatterdd [r15+zmm3*4]{k1}, zmm0
	    cmp ecx, ebx
	    jb histogram_loop
	```

Scalar, Baseline, 1x	Speedup: 1.11x (random inputs); 1.34x (input values identical)

Notice that the end result of the conflict loop (i.e., the resulting vector after all merging is done, ZMM2 in the above sequence) holds the complete set of partial sums. That is, for each element, the result contains the value of that element merged with all earlier elements with the same index value. Using the earlier example values, ZMM2 contains the result shown in Figure 18-13.

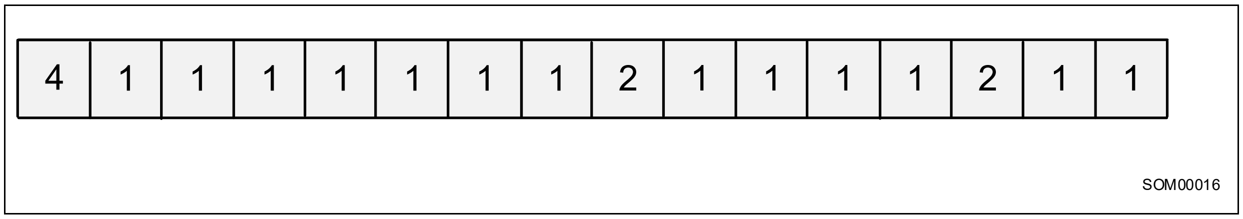


Figure 18-13. VPCONFLICTD ZMM2 Result

While the above sequence does not take advantage of this, other use cases might.

18.16.2 Sparse Dot Product with VPCONFLICT

A sparse vector may be stored as a pair of arrays: one containing non-zero values, and one containing the original locations of those values in the vector. Note that the indices are sorted in increasing order.

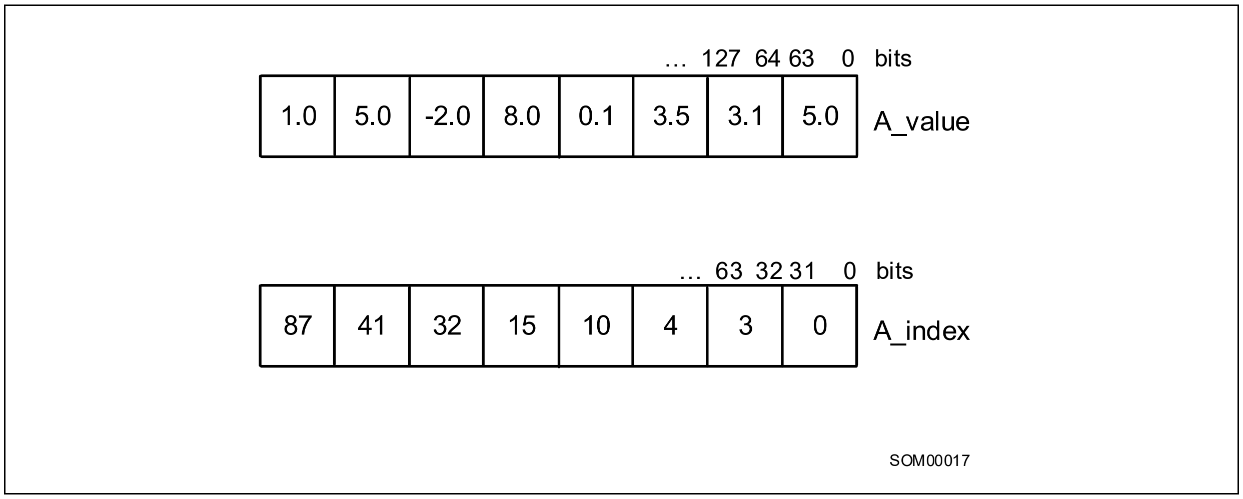


Figure 18-14. Sparse Vector Example

To perform a dot product of two sparse vectors efficiently, we need to find elements with matching indices; those are the only ones on which we should perform the multiply and accumulation. The scalar method for doing this is to start at the beginning of the two index arrays, compare those indices, and if there is a match, do the multiply and accumulate, then advance the indices of both vectors. If there is no match, we advance the index of the lagging vector.

```
A_offset = 0; B_offset = 0; sum = 0;
while ((A_offset < A_length) && (B_offset < B_length))
  {
      if (A_index[A_offset] == B_index[B_offset]) // match
        {
               sum += A_value[A_offset] * B_value[B_offset];
               A_offset++;
               B_offset++;
        }
      else if (A_index[A_offset] < B_index[B_offset])
        {
               A_offset++;
        }
      else
        {
               B_offset++;
        }
  }
```

The Intel AVX-512CD instructions provide an efficient way to vectorize this loop. Instead of comparing one index from each vector at a time, we can compare eight of them. First we combine eight indices from each vector into a single vector register. Then, the VPCONFLICT instruction compares the indices. We use the output to create a mask of elements in vector A that have a match, and also to create permute controls to move the corresponding values of B to the same location, so that we can use a vector FMA instruction.

Example 18-20 shows the assembly code for both the scalar and vector versions of a single comparison and FMA. For brevity, the offset updates and looping are omitted.

Example 18-20. Scalar vs. Vector Update Using AVX-512CD

Scalar Code	Intel® AVX-512 Code

mov rdx, A_index
mov rcx, A_offset
mov rax, A_value
mov r12, B_index
mov r13, B_offset
mov rbx, B_value
 	
mov r10d, [rdx+rcx*4]
mov r11d, [r12+r13*4]
cmp r10d, r11d
jne skip_fma
 	
// do the fma on a match
movsd xmm5, [rbx+r13*8]
mulsd xmm5, [rax+rcx*8]
addsd xmm4, xmm5
skip_fma:
	mov rdx, A_index
	mov rcx, A_offset
	mov rax, A_value
	mov r12, B_index
	mov r13, B_offset
	mov rbx, B_value
	mov r14, all_31s // array of {31, 31, ...}
	vmovaps zmm2, [r14]
	mov r15, upconvert_control // array of {0, 7, 0, 6, 0, 5, 0, 4, 0, 3, 0, 2, 0, 1, 0, 0}
	vmovaps zmm1, [r15]
	vpternlogd zmm0, zmm0, zmm0, 255
	movl esi, 21845
	kmovw k1, esi // odd bits set
	
	// read 8 indices for A
	vmovdqu ymm5, [rdx+rcx*4]
	// read 8 indices for B, and put
	// them in the high part of zmm6
	vinserti64x4 zmm6, zmm5, [r12+r13*4], 1
	vpconflictd zmm7, zmm6
	// extract A vs. B comparisons
	vextracti64x4 ymm8, zmm7, 1
	// convert comparison results to
	// permute control
	vplzcntd zmm9, zmm8
	vptestmd k2, zmm8, zmm0
	vpsubd zmm10, zmm2, zmm9
	// upconvert permute controls from
	// 32b to 64b, since data is 64b
	vpermd zmm11{k1}, zmm1, zmm10
	// Move A values to corresponding
	// B values, and do FMA
	vpermpd zmm12{k2}{z}, zmm11, [rax+rcx*8]
	vfmadd231pd zmm4, zmm12, [rbx+r13*8]

Baseline, 1x	Speedup, 4.4x

18.17 INTEL® AVX-512 VECTOR BYTE MANIPULATION INSTRUCTIONS (VBMI)

Intel® AVX-512 VBMI instructions are a set of 512-bit instructions that are designed to speed up bit manipulation operations. The following sections describe the new instructions and show simple usage examples. See the *Intel® 64 and IA-32 Architectures Software Developer's Manual* for complete instruction definitions. Processors that provide VBMI1 and VBMI2 are enumerated by the CPUID feature flags CPUID:(EAX=07H, ECX=0):ECX[bit 01] = 1 and CPUID:(EAX=07H, ECX=0):ECX[bit 06] = 1, respectively.

18.17.1 Permute Packet Bytes Elements Across Lanes (VPERMB)

The VPERMB instruction is a single source, any-to-any byte permute instruction. The following figure shows a VPERMB instruction operation example.

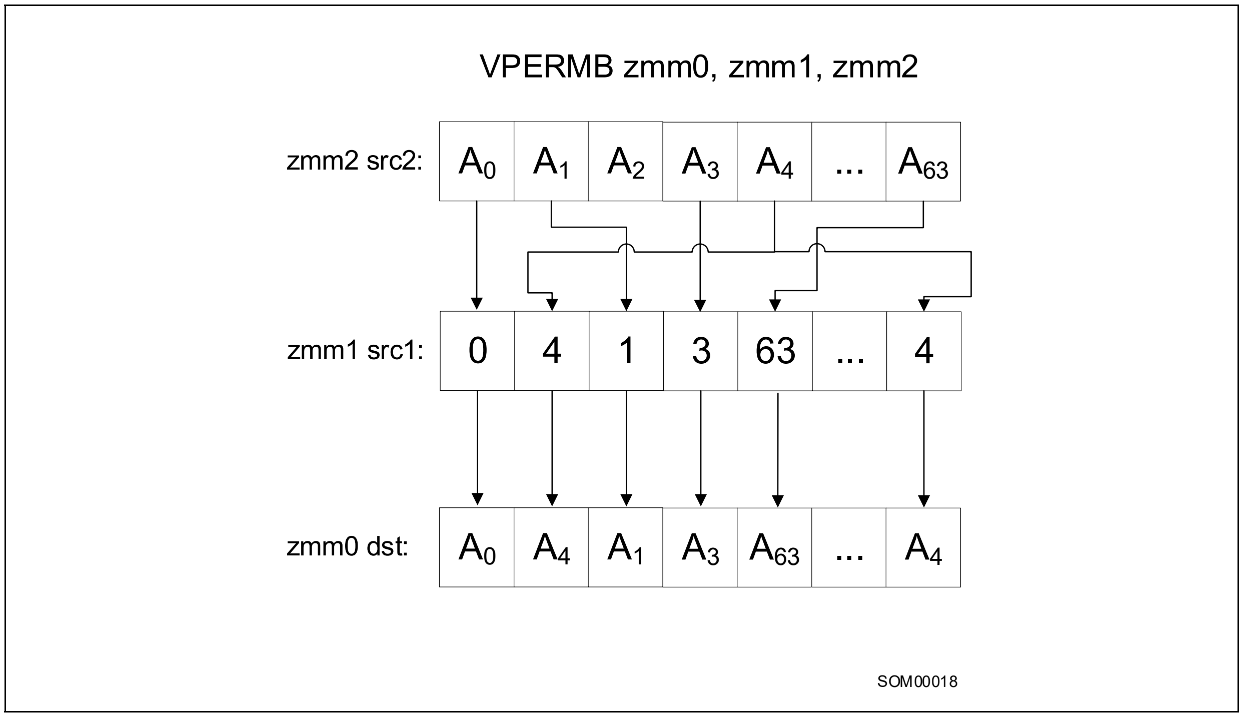

VPERMB zmm0, zmm1, zmm2
zmm2 src2:
A_0 A_1 A_2 A_3 A_4 ... A_{63}
zmm1 src1:
0 4 1 3 63 ... 4
zmm0 dst:
A_0 A_4 A_1 A_3 A_{63} ... A_4
SOM00018

Figure 18-15. VPERMB Instruction Operation

VPERMB Operation:

```
// vpermb zmm Dst {k1}, zmm Src1, zmm Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;

for(int i=0;i<64;i++){
    if(k1[i]){
        Dst[i]= Src2[Src1[i]];
    }else{
        Dst[i]= zero_masking? 0 : Dst[i];
    }
}
```

The following example shows a 64-byte lookup table implementation.

Scalar code:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char* dictionary_bytes, int numOfElements){
    for(int i = 0; i < numOfElements; i++) {
            out_bytes[i] = dictionary_bytes[in_bytes[i] & 63];
      }
}
```

Example 18-21. Improvement with VPERMB Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPERMB Implementation
mov rsi, dictionary_bytes mov r11, in_bytes mov rax, out_bytes mov r9d, numOfElements xor r8, r8 vpmovzxbw zmm3, [rsi] vpmovzxbw zmm4, [rsi+32] loop: vpmovzxbw zmm1, [r11+r8*1] vpmovzxbw zmm2, [r11+r8*1+32] vpermi2w zmm1, zmm3, zmm4 vpermi2w zmm2, zmm3, zmm4 vpmovwb [rax+r8*1], zmm1 vpmovwb [rax+r8*1+32], zmm2 add r8, 64 cmp r8, r9 jl loop	mov rsi, dictionary_bytes mov r11, in_bytes mov rax, out_bytes mov r9d, numOfElements xor r8, r8 vmovdqu32 zmm2, [rsi] loop: vmovdqu32 zmm1, [r11+r8*1] vpermb zmm1, zmm1, zmm2 vmovdqu32 [rax+r8*1], zmm1 add r8, 64 cmp r8, r9 jl loop
Base Measurement: 1x	Speedup: 6.5x

18.17.2 Two-Source Byte Permute Across Lanes (VPERMI2B, VPERMT2B)

The VPERMI2B and VPERMT2B instructions are two-source byte, permute instructions. The destination is also an operation source; in VPERMI2B the destination is the operation index, and in VPERMT2B the destination is one of the data sources.

The following figure shows a VPERMI2B instruction operation example.

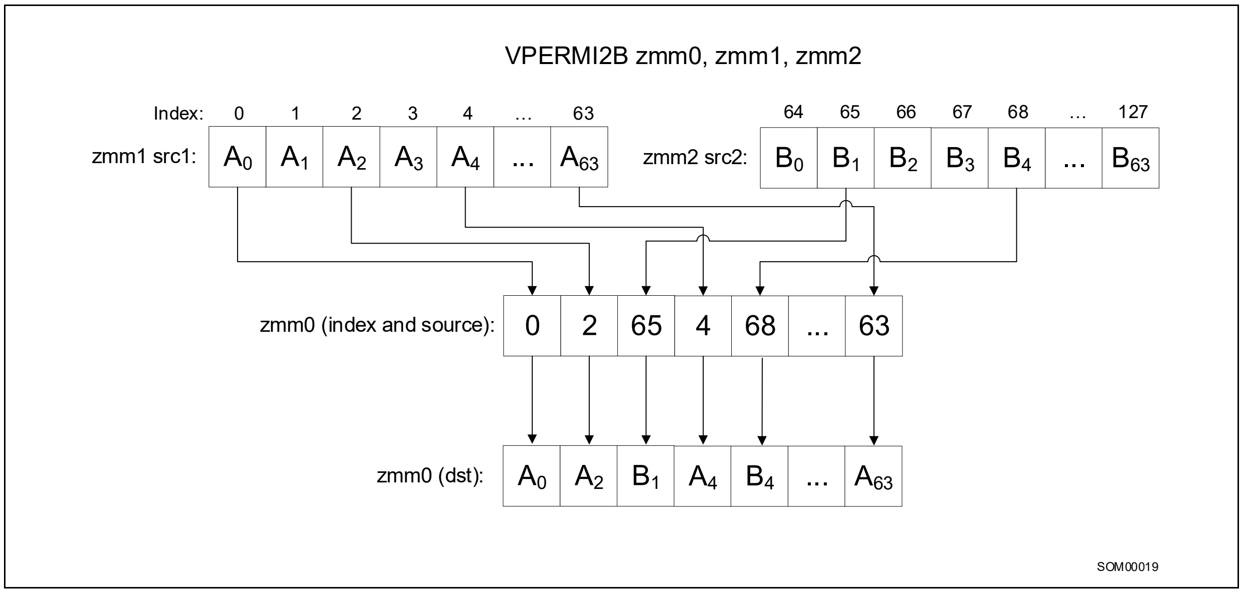

VPERMI2B zmm0, zmm1, zmm2
Index: 0 1 2 3 4 ... 63
zmm1 src1: A_0 A_1 A_2 A_3 A_4 ... A_{63}
64 65 66 67 68 ... 127
zmm2 src2: B_0 B_1 B_2 B_3 B_4 ... B_{63}
zmm0 (index and source): 0 2 65 4 68 ... 63
zmm0 (dst): A_0 A_2 B_1 A_4 B_4 ... A_{63}
SOM00019

Figure 18-16. VPERMI2B Instruction Operation

VPERMI2B Operation:

```
/// vpermi2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;
for(int i=0;i<64;i++){
    if(k1[i]){
        Dst[i]= Dst [i]>63 ? Src1[Dst [i] & 63] : Src2[Dst [i] & 63] ;
    }else{
        Dst[i]= zero_masking? 0 : Dst[i];
    }
}
```

The following figure shows a VPERMT2B instruction operation example.

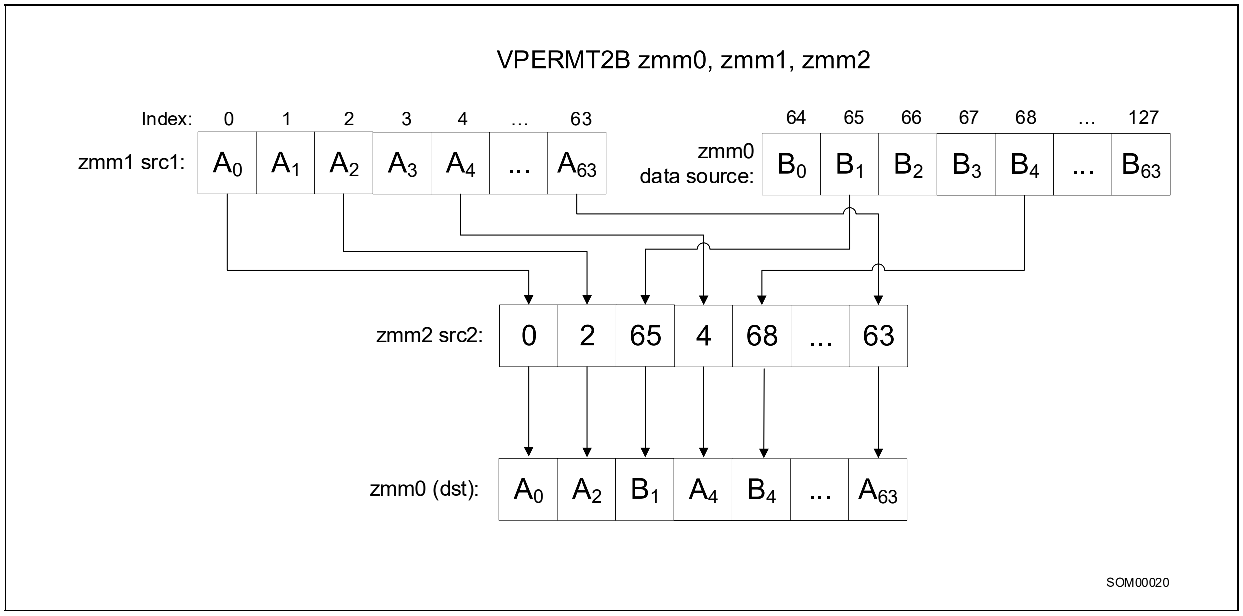

VPERMT2B zmm0, zmm1, zmm2
Index: 0 1 2 3 4 … 63
zmm1 src1: A0 A1 A2 A3 A4 … A63
64 65 66 67 68 … 127
zmm0 data source: B0 B1 B2 B3 B4 … B63
zmm2 src2: 0 2 65 4 68 … 63
zmm0 (dst): A0 A2 B1 A4 B4 … A63
SOM00020

Figure 18-17. VPERMT2B Instruction Operation

VPERMT2B Operation:

```
// vpermt2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, * Src2;
data2= copy(Dst);
for(int i=0;i<64;i++){
    if(k1[i]){
        Dst[i]= Src2[i]>63 ? Src1[Src2 [i] & 63] : Dst[Src2[i] & 63] ;
    }else{
        Dst[i]= zero_masking? 0 : Dst[i];
    }
}
```

The following example shows a 128-byte lookup table implementation.

C Code:
```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char* dictionary_bytes, int numOfElements){
    for(int i = 0; i < numOfElements; i++) {
        out_bytes[i] = dictionary_bytes[in_bytes[i] & 127];
    }
}
```

Example 18-22. Improvement with VPERMI2B Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPERMI2B Implementation

//get data sent to function
mov rsi, dictionary_bytes
mov r11, in_bytes
mov rax, out_bytes
mov r9d, numOfElements
xor r8, r8
//Reorganize dictionary
vpmovzxbw zmm10, [rsi]
vpmovzxbw zmm15, [rsi+64]
vpsllw zmm15, zmm15, 8
vpord zmm10, zmm15, zmm10
vpmovzxbw zmm11, [rsi+32]
vpmovzxbw zmm15, [rsi+96]
vpsllw zmm15, zmm15, 8
vpord zmm11, zmm15, zmm11
//initialize constants
mov r10, 0x00400040
vpbroadcastw zmm12, r10d
mov r10, 0
vpbroadcastd zmm13, r10d
mov r10, 0x00ff00ff
vpbroadcastd zmm14, r10d
//start iterations
loop:
vpmovzxbw zmm1, [r11+r8*1]
vpandd zmm2, zmm1, zmm12
vpcmpw k1, zmm2, zmm13, 4
vpermi2w zmm1, zmm10, zmm11
vpsrlw zmm1{k1}, zmm1, 8
vpandd zmm1, zmm1, zmm14
vpmovwb [rax+r8*1], zmm1
add r8, 32
cmp r8, r9
jl loop
	mov rsi, dictionary_bytes
	mov r11, in_bytes
	mov rax, out_bytes
	mov r9d, numOfElements
	xor r8, r8
	vmovdqu32 zmm2, [rsi]
	vmovdqu32 zmm3, [rsi+64]
	loop:
	vmovdqu32 zmm1, [r11+r8*1]
	vpermi2b zmm1, zmm2, zmm3
	vmovdqu32 [rax+r8*1], zmm1
	add r8, 64
	cmp r8, r9
	jl loop

Base Measurement: 1x	Speedup: 5.3x

18.17.3 Select Packed Unaligned Bytes from Quadword Sources (VPMULTISHIFTQB)

The VPMULTISHIFTQB instruction selects eight unaligned bytes from each input qword element of the second source operand and writes eight assembled bytes for each qword element in the destination operand.

The following figure shows a VPMULTISHIFTQB instruction operation example.

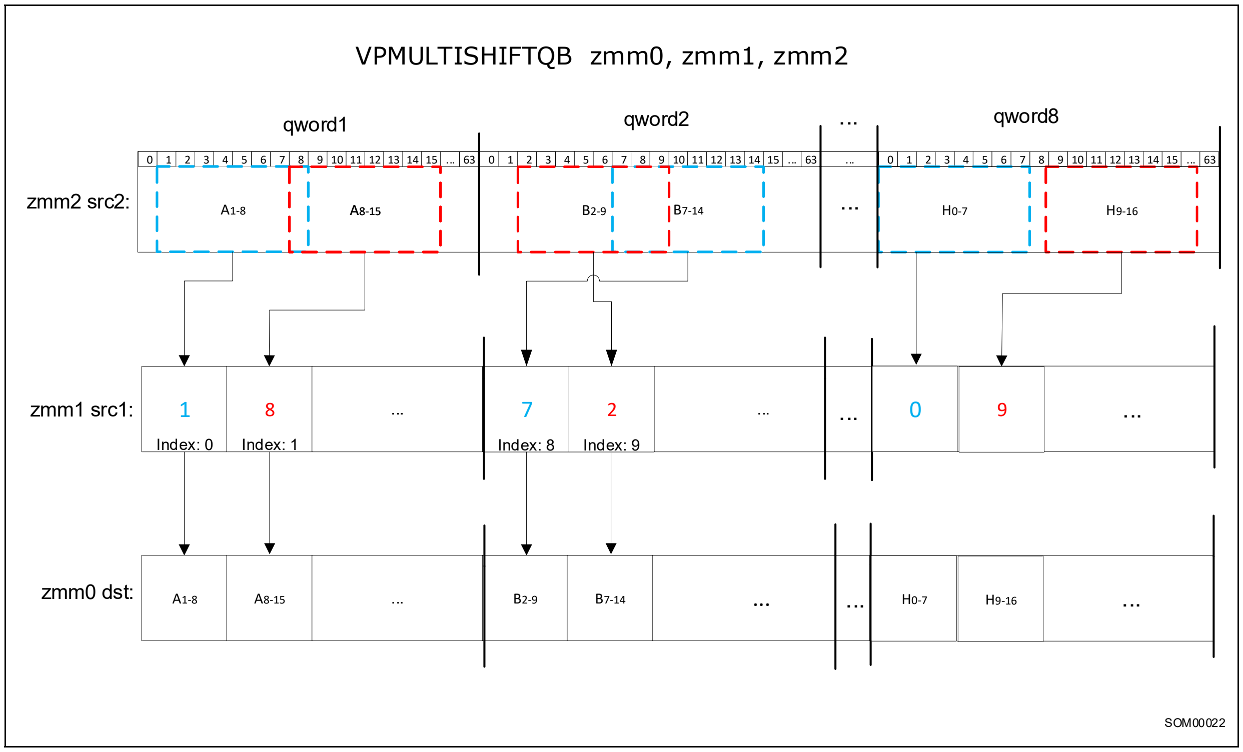


Figure 18-18. VPMULTISHIFTQB Instruction Operation

VPMULTISHIFTQB Operation:

```
// vpmultishiftqb Dst{k1},Src1,Src2
bool zero_masking=false;
unsigned char *Dst, * Src1;
unsigned __int64 *Src2;
bit * k1;
for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        if(k1[i*8 +j]){
            Dst[i*8 +j]= (src2[i]>> Src1[i*8 +j]) &0xFF ;
        }else{
            Dst[i*8 +j]= zero_masking? 0 : Dst[i*8 +j];
        }
    }
}
```

The following example converts a 5-bit unsigned integer array to a 1-byte unsigned integer array.

C code:

```
void decompress (unsigned char* compressedData, unsigned char* decompressedData, int numOfElements){
    for(int i = 0; i < numOfElements; i += 8){
        unsigned __int64 * data = (unsigned __int64 * )compressedData;
        decompressedData[i+0] = * data & 0x1f;
        decompressedData[i+1] = (*data >> 5 ) & 0x1f;
        decompressedData[i+2] = (*data >> 10 ) & 0x1f;
        decompressedData[i+3] = (*data >> 15 ) & 0x1f;
        decompressedData[i+4] = (*data >> 20 ) & 0x1f;
        decompressedData[i+5] = (*data >> 25 ) & 0x1f;
        decompressedData[i+6] = (*data >> 30 ) & 0x1f;
        decompressedData[i+7] = (*data >> 35 ) & 0x1f;
        compressedData += 5;
    }
}
```

Example 18-23. Improvement with VPMULTISHIFTQB Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPMULTISHIFTQB Implementation

mov rdx, compressedData
mov r9, decompressedData
mov eax, numOfElements
shr eax,3
xor rsi, rsi
loop:
mov rcx, qword ptr [rdx]
mov r10, rcx
and r10, 0x1f
mov r11, rcx
mov byte ptr [r9+rsi*8], r10b
mov r10, rcx
shr r10, 0xa
add rdx, 0x5
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x2], r10b
mov r10, rcx
shr r10, 0xf
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x3], r10b
mov r10, rcx
shr r10, 0x14
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x4], r10b
mov r10, rcx
shr r10, 0x19
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x5], r10b
mov r10, rcx
shr r11, 0x5
shr r10, 0x1e
	//constants :
	__declspec (align(64)) const unsigned __int8
	permute_ctrl[64] = {
	    0, 1, 2, 3, 4, 0, 0, 0
	    5, 6, 7, 8, 9, 0, 0, 0
	    10, 11, 12, 13, 14, 0, 0, 0
	    15, 16, 17, 18, 19, 0, 0, 0
	    20, 21, 22, 23, 24, 0, 0, 0
	    25, 26, 27, 28, 29, 0, 0, 0
	    30, 31, 32, 33, 34, 0, 0, 0
	    35, 36, 37, 38, 39, 0, 0, 0
	};
	__declspec (align(64)) const unsigned __int8
	multishift_ctrl[64] = {
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	    0, 5, 10, 15, 20, 25, 30, 35
	};
	//asm:
	mov rsi, compressedData
	mov rdi, decompressedData
	mov r8d, numOfElements
	lea r8, [rdi+r8]
	mov r9, 0x1F1F1F1F
	vpbroadcastd zmm12, r9d
	vmovdqu32 zmm10, permute_ctrl
	vmovdqu32 zmm11, multishift_ctrl

Example 18-23. Improvement with VPMULTISHIFTQB Implementation (Contd.)

and r11, 0x1f shr rcx, 0x23 and r10, 0x1f and rcx, 0x1f mov byte ptr [r9+rsi*8+0x1], r11b mov byte ptr [r9+rsi*8+0x6], r10b mov byte ptr [r9+rsi*8+0x7], cl inc rsi cmp rsi, rax jb loop	loop: vmovdqu32 zmm1, [rsi] vpermb zmm2, zmm10, zmm1 vpmultishiftqb zmm2, zmm11, zmm2 vpandq zmm2, zmm12, zmm2 vmovdqu32 [rdi], zmm2 add rdi, 64 add rsi, 40 cmp rdi, r8 jl loop
Base Measurement: 1x	Speedup: 26x

18.18 FMA LATENCY

When executing in 512-bit register port scheme, Port 0 FMA has a latency of 4 cycles, and Port 5 FMA has a latency of 6 cycles. Bypass can have a -2 (fast bypass) to +1 cycle delay. Therefore, instructions that execute on the Skylake microarchitecture FMA have a latency of 4-7 cycles.

The instructions are divided into the following two groups.

- Group A Instructions: vadd*; vfmadd*; vfnmsub*; vfnmadd*; vfnmsub*; vmax*; vmin*; vmul*; vscalef*; vsub*; vcvt*; vgetexp*; vfixupimm*; vrange*; vgetmant*; vreduce*; vcmp*, vcomi*, vdpp*, vhadd*, vhsub*, vrndscale*, vround*
- Group B Instructions: vpmaddubsw; vpmaddwd; vpmuldq; vpmulhrsw; vpmulhuw; vpmulhw; vpmullw; vpmuludq

The FMA unit supports fast bypass when all instruction sources come from the FMA unit. In this case Group A has a latency of 4 cycles for both ports 0 and 5, and Group B has a latency of 5 cycles for both ports 0 and 5.

The figure below explains fast bypass when all sources come from the FMA unit.

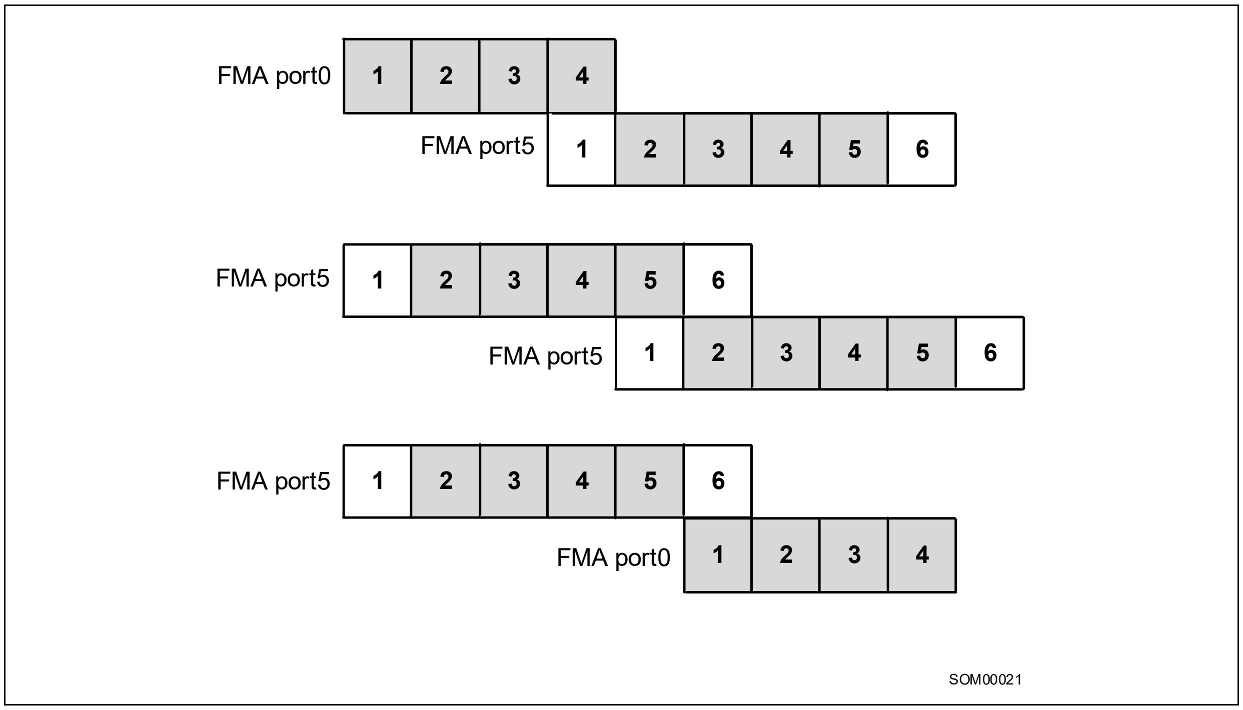


Figure 18-19. Fast Bypass When All Sources Come from FMA Unit

The grey boxes represent compute cycles. The white boxes represent data transfer for the port5 FMA unit.

If fast bypass is not used, that is, when not all sources come from the FMA unit, group A instructions have a latency of 4 cycles on Port0 and 6 cycles on port5, while group B instructions have an additional cycle and hence have a latency of 5 cycles on Port0 and 7 cycles on port5.

The following table summarizes the FMA unit latency for the various options.

Table 18-7. FMA Unit Latency

Instruction Group	**Fast Bypass (FMA Data Reuse)**		**No Fast Bypass (No FMA Data Reuse)**	
	Port 0	**Port 5**	**Port 0**	**Port 5**
Group A	4	4	4	6
Group B	5	5	5	7

18.19 MIXING INTEL® AVX OR INTEL® AVX-512 EXTENSIONS WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE) CODE

There are two main instruction groups that affect the processor states:

- Group A: Instruction types that either set bits 128-511 of vector registers 0-15 to zero, or do not modify them at all.
  - Intel SSE instructions.
  - 128-bit Intel AVX instructions, 128-bit Intel AVX-512 instructions.
  - 256-bit (ymm16-ymm31) Intel AVX-512 instructions.
  - 512-bit (zmm16-zmm31) Intel AVX-512 instructions.
  - AVX-512 instructions that write to mask registers k0-k7.
  - GPR instructions.
- Group B: Instructions types that modify bits 128-511 of vector registers 0-15.
  - 256-bit (ymm0-ymm15) Intel AVX instructions, Intel AVX-512 instructions.
  - 512-bit (zmm0-zmm15) Intel AVX-512 instructions.

The following figure illustrates Skylake Server microarchitecture's model for mixing Intel AVX instructions or Intel AVX-512 instructions with Intel SSE instructions.

The implementation is similar to Skylake client microarchitecture, where every Intel SSE instruction executed in Dirty Upper State (2) needs to preserve bits 128-511 of the destination register, and therefore the operation has an additional dependency on the destination register and a blend operation with bits 128-511.

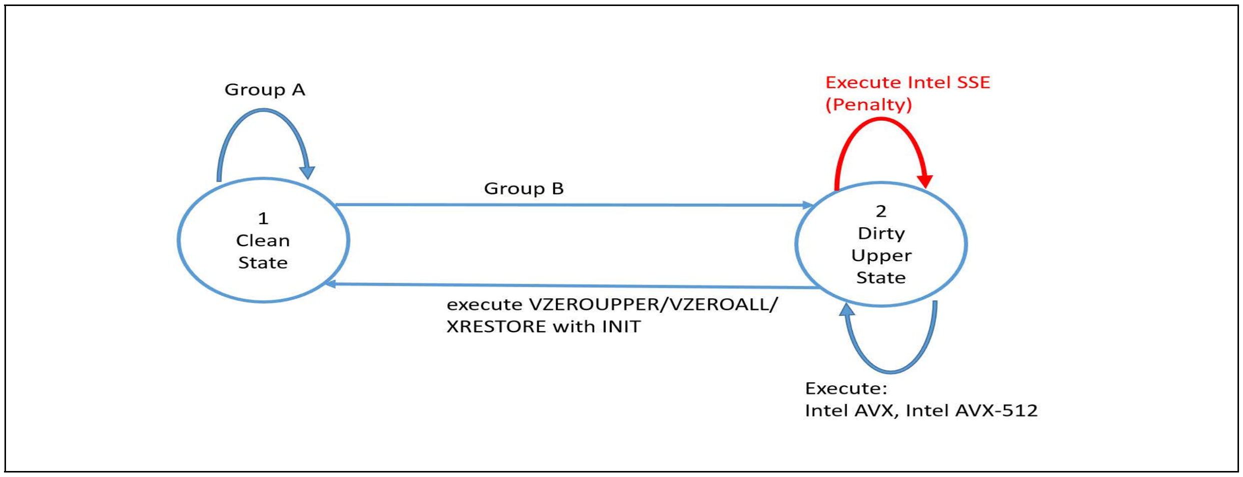

Group A
Execute Intel SSE (Penalty)
Group B
1 Clean State
2 Dirty Upper State
execute VZEROUPPER/VZEROALL/ XRESTORE with INIT
Execute: Intel AVX, Intel AVX-512

Figure 18-20. Mixing Intel AVX Instructions or Intel AVX-512 Instructions with Intel SSE Instructions

Recommendations:

- When mixing group B instructions with Intel SSE instructions, or suspecting that such a mixture might occur, use the VZEROUPPER instruction whenever a transition is expected.
- Add VZEROUPPER after group B instructions were executed and before any function call that might lead to an Intel SSE instruction execution.
- Add VZEROUPPER at the end of any function that uses group B instructions.
- Add VZEROUPPER before thread creation if not already in a clean state so that the thread does not inherit a Dirty Upper State.

18.20 MIXING ZMM VECTOR CODE WITH XMM/YMM

Skylake microarchitecture has two port schemes, one for using 256-bit or less registers, and another for using 512-bit registers.

When using registers up to or including 256 bits, FMA operations dispatch to ports 0 and 1 and SIMD operations dispatch to ports 0, 1 and 5. When using 512-bit register operations, both FMA and SIMD operations dispatch to ports 0 and 5.

The maximum register width in the reservation station (RS) determines the 256 or 512 port scheme.

Notice that when using AVX-512 encoded instructions with YMM registers, the instructions are considered to be 256-bit wide.

The result of the 512-bit port scheme is that XMM or YMM code dispatches to 2 ports (0 and 5) instead of 3 ports (0, 1, and 5) and may have lower throughput and longer latency compared to the 256-bit port scheme.

Example 18-24. 256-bit Code vs. 256-bit Code Mixed with 512-bit Code

256-bit Code Only	256-bit Code Mixed with 512-bit Code

```
Loop:
vpbroadcastd    ymm0, dword ptr [rsp]
vfmadd213ps     ymm7, ymm7, ymm7
vfmadd213ps     ymm8, ymm8, ymm8
vfmadd213ps     ymm9, ymm9, ymm9
vfmadd213ps     ymm10, ymm10, ymm10
vfmadd213ps     ymm11, ymm11, ymm11
vfmadd213ps     ymm12, ymm12, ymm12
vfmadd213ps     ymm13, ymm13, ymm13
vfmadd213ps     ymm14, ymm14, ymm14
vfmadd213ps     ymm15, ymm15, ymm15
vfmadd213ps     ymm16, ymm16, ymm16
vfmadd213ps     ymm17, ymm17, ymm17
vfmadd213ps     ymm18, ymm18, ymm18
vpermd          ymm1, ymm1, ymm1
vpermd          ymm2, ymm2, ymm2
vpermd          ymm3, ymm3, ymm3
vpermd          ymm4, ymm4, ymm4
vpermd          ymm5, ymm5, ymm5
vpermd          ymm6, ymm6, ymm6
dec rdx
jnle Loop
```
	```
	Loop:
	vpbroadcastd    zmm0, dword ptr [rsp]
	vfmadd213ps     ymm7, ymm7, ymm7
	vfmadd213ps     ymm8, ymm8, ymm8
	vfmadd213ps     ymm9, ymm9, ymm9
	vfmadd213ps     ymm10, ymm10, ymm10
	vfmadd213ps     ymm11, ymm11, ymm11
	vfmadd213ps     ymm12, ymm12, ymm12
	vfmadd213ps     ymm13, ymm13, ymm13
	vfmadd213ps     ymm14, ymm14, ymm14
	vfmadd213ps     ymm15, ymm15, ymm15
	vfmadd213ps     ymm16, ymm16, ymm16
	vfmadd213ps     ymm17, ymm17, ymm17
	vfmadd213ps     ymm18, ymm18, ymm18
	vpermd          ymm1, ymm1, ymm1
	vpermd          ymm2, ymm2, ymm2
	vpermd          ymm3, ymm3, ymm3
	vpermd          ymm4, ymm4, ymm4
	vpermd          ymm5, ymm5, ymm5
	vpermd          ymm6, ymm6, ymm6
	dec rdx
	jnle Loop
	```

Baseline 1x	Slowdown: 1.3x

In the 256-bit code only example, the FMAs are dispatched to ports 0 and 1, and *permd* is dispatched to port 5 as the broadcast instruction is 256 bits wide. In the 256-bit and 512-bit mixed code example, the broadcast is 512 bits wide; therefore, the processor uses the 512-bit port scheme where the FMAs dispatch to ports 0 and 5 and *permd* to port 5, thus increasing the pressure on port 5.

18.21 SERVERS WITH A SINGLE FMA UNIT

Some processors based on Skylake microarchitecture have two Intel AVX-512 FMA units, on ports 0 and 5, while other processors based on Skylake microarchitecture have a single Intel AVX-512 FMA unit, which is located on port 0.

Code that is optimized to run on a processor with two FMA units might not be optimal when run on a processor with one FMA unit.

The following example code shows how to detect whether a system has one or two Intel AVX-512 FMA units. It includes the following:

- An Intel AVX-512 warmup.
- A function that executes only FMA instructions.
- A function that executes both FMA and shuffle instructions.
- Code that, based on the results of these two tests, identifies whether the processor has one or two FMA units.

Notice that each test is executed three times to improve test accuracy.

In order to reduce the program overhead, it is highly recommended not to execute this test in every function call, but as part of installation, or once at startup.

The differentiation between the two processors is based on the ratio between the two throughput tests. Processors with two FMA units are able to run the FMA-only test twice as fast as the FMA and shuffle test. However, a processor with one FMA unit will run both tests at the same speed.

Example 18-25. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
#include <string.h>
#include <stdlib.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdint.h>

static uint64_t rdtsc(void) {
  unsigned int ax, dx;

  __asm__ __volatile__ ("rdtsc" : "=a"(ax), "=d"(dx));

  return ((((uint64_t)dx) << 32) | ax);
}

uint64_t fma_shuffle_tpt(uint64_t loop_cnt){
      uint64_t loops =  loop_cnt;
      __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
      __declspec(align(64)) int shuf_vec[16] = {0, 1, 2, 3,4, 5, 6, 7,8, 9, 10, 11,12, 13, 14, 15};
       __asm
       {
             vmovups zmm0, [one_vec]
             vmovups zmm1, [one_vec]
             vmovups zmm2, [one_vec]
```

Example 18-25. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture (Contd.)

```
        vmovups zmm3, [one_vec]
        vmovups zmm4, [one_vec]
        vmovups zmm5, [one_vec]
        vmovups zmm6, [one_vec]
        vmovups zmm7, [one_vec]
        vmovups zmm8, [one_vec]
        vmovups zmm9, [one_vec]
        vmovups zmm10, [one_vec]
        vmovups zmm11, [one_vec]
        vmovups zmm12, [shuf_vec]
        vmovups zmm13, [shuf_vec]
        vmovups zmm14, [shuf_vec]
        vmovups zmm15, [shuf_vec]
        vmovups zmm16, [shuf_vec]
        vmovups zmm17, [shuf_vec]
        vmovups zmm18, [shuf_vec]
        vmovups zmm19, [shuf_vec]
        vmovups zmm20, [shuf_vec]
        vmovups zmm21, [shuf_vec]
        vmovups zmm22, [shuf_vec]
        vmovups zmm23, [shuf_vec]
        vmovups zmm30, [shuf_vec]
        mov rdx, loops
    loop1:
        vfmadd231pd zmm0, zmm0, zmm0
        vfmadd231pd zmm1, zmm1, zmm1
        vfmadd231pd zmm2, zmm2, zmm2
        vfmadd231pd zmm3, zmm3, zmm3
        vfmadd231pd zmm4, zmm4, zmm4
        vfmadd231pd zmm5, zmm5, zmm5
        vfmadd231pd zmm6, zmm6, zmm6
        vfmadd231pd zmm7, zmm7, zmm7
        vfmadd231pd zmm8, zmm8, zmm8
        vfmadd231pd zmm9, zmm9, zmm9
        vfmadd231pd zmm10, zmm10, zmm10
        vfmadd231pd zmm11, zmm11, zmm11
        vpermd zmm12, zmm30, zmm30
        vpermd zmm13, zmm30, zmm30
        vpermd zmm14, zmm30, zmm30
        vpermd zmm15, zmm30, zmm30
        vpermd zmm16, zmm30, zmm30
        vpermd zmm17, zmm30, zmm30
        vpermd zmm18, zmm30, zmm30
        vpermd zmm19, zmm30, zmm30
        vpermd zmm20, zmm30, zmm30
        vpermd zmm21, zmm30, zmm30
        vpermd zmm22, zmm30, zmm30
        vpermd zmm23, zmm30, zmm30
        dec rdx
        jg loop1
     }
}
```

Example 18-25. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture (Contd.)

```
uint64_t fma_only_tpt(int loop_cnt){
    uint64_t loops =  loop_cnt;
    __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
     __asm
     {
         vmovups zmm0, [one_vec]
         vmovups zmm1, [one_vec]
         vmovups zmm2, [one_vec]
         vmovups zmm3, [one_vec]
         vmovups zmm4, [one_vec]
         vmovups zmm5, [one_vec]
         vmovups zmm6, [one_vec]
         vmovups zmm7, [one_vec]
         vmovups zmm8, [one_vec]
         vmovups zmm9, [one_vec]
         vmovups zmm10, [one_vec]
         vmovups zmm11, [one_vec]
         mov rdx, loops
    loop1:
         vfmadd231pd zmm0, zmm0, zmm0
         vfmadd231pd zmm1, zmm1, zmm1
         vfmadd231pd zmm2, zmm2, zmm2
         vfmadd231pd zmm3, zmm3, zmm3
         vfmadd231pd zmm4, zmm4, zmm4
         vfmadd231pd zmm5, zmm5, zmm5
         vfmadd231pd zmm6, zmm6, zmm6
         vfmadd231pd zmm7, zmm7, zmm7
         vfmadd231pd zmm8, zmm8, zmm8
         vfmadd231pd zmm9, zmm9, zmm9
         vfmadd231pd zmm10, zmm10, zmm10
         vfmadd231pd zmm11, zmm11, zmm11
         dec rdx
         jg loop1
     }
}

int main()
{
    int i;
    uint64_t fma_shuf_tpt_test[3];
    uint64_t fma_shuf_tpt_test_min;
    uint64_t fma_only_tpt_test[3];
    uint64_t fma_only_tpt_test_min;
    uint64_t start = 0;
    uint64_t number_of_fma_units_per_core = 2;
```

Example 18-25. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture (Contd.)

```
    /*********************************************************/
    /* Step 1: Warmup */
    /*********************************************************/
    fma_only_tpt(100000);

    /*********************************************************/
    /* Step 2: Execute FMA and Shuffle TPT Test */
    /*********************************************************/

    for(i = 0; i < 3; i++){
        start = rdtsc();
        fma_shuffle_tpt(1000);
        fma_shuf_tpt_test[i] = rdtsc() - start;
    }


    /*********************************************************/
    /* Step 3: Execute FMA only TPT Test  */
    /*********************************************************/
    for(i = 0; i < 3; i++){
        start = rdtsc();
        fma_only_tpt(1000);
        fma_only_tpt_test[i] = rdtsc() - start;
    }

    /*********************************************************/
    /* Step 4: Decide if 1 FMA server or 2 FMA server */
    /*********************************************************/
    fma_shuf_tpt_test_min = fma_shuf_tpt_test[0];
    fma_only_tpt_test_min = fma_only_tpt_test[0];
    for(i = 1; i < 3; i++){
        if ((int)fma_shuf_tpt_test[i] < (int)fma_shuf_tpt_test_min) fma_shuf_tpt_test_min = fma_shuf_tpt_test[i];
        if ((int)fma_only_tpt_test[i] < (int)fma_only_tpt_test_min) fma_only_tpt_test_min = fma_only_tpt_test[i];
    }

    if(((double)fma_shuf_tpt_test_min/(double)fma_only_tpt_test_min) < 1.5){
        number_of_fma_units_per_core = 1;
    }

    printf(”%d FMA server\n”, number_of_fma_units_per_core);
    return 0;
}
```

18.22 GATHER/SCATTER TO SHUFFLE (G2S/STS)

18.22.1 Gather to Shuffle in Strided Loads

In cases where there is data locality between gathered elements in memory, performance can be improved by replacing the gather instruction with a software sequence.

This section discusses the very common strided load pattern. Strided loads are sets of loads where the offset in memory between two consecutive loads is constant.

The following examples show three different code variations performing an Array of Structures (AOS) to Structure of Arrays (SOA) transformation. The code separates the real and imaginary elements in a complex array into two separate arrays.

Consider the following C code:

```
for(int i=0;i<len;i++){

     Real_buffer[i] = Complex_buffer[i].real;

     Imaginary_buffer[i] = Complex_buffer[i].imag;

}
```

Example 18-26. Gather to Shuffle in Strided Loads Example

Alternative 1: Intel® AVX-512 vpgatherdd	**Alternative 2: G2S Using Intel® AVX-512 vpermi2d**
loop: vpcmpeqb k1, xmm0, xmm0 vpcmpeqb k2, xmm0, xmm0 movsxd rdx, edx movsxd rdi, esi inc esi shl rdi, 0x7 vpxord zmm2, zmm2, zmm2 lea rax, [r8+rdx*8] add edx, 0x20 vpgatherdd zmm2{k1}, [rax+zmm1*4] vpxord zmm3, zmm3, zmm3 vpxord zmm4, zmm4, zmm4 vpxord zmm5, zmm5, zmm5 vpgatherdd zmm3{k2}, [rax+zmm0*4] vpcmpeqb k3, xmm0, xmm0 vpcmpeqb k4, xmm0, xmm0 vmovups [r9+rdi*1], zmm2 vmovups [rcx+rdi*1], zmm3 vpgatherdd zmm4{k3}, [rax+zmm1*4+0x80] vpgatherdd zmm5{k4}, [rax+zmm0*4+0x80] vmovups [r9+rdi*1+0x40], zmm4 vmovups [rcx+rdi*1+0x40], zmm5 cmp esi, r14d jb loop	vmovups zmm4, [rdx+r9*8] vmovups zmm0, [rdx+r9*8+0x40] vmovups zmm5, [rdx+r9*8+0x80] vmovups zmm1, [rdx+r9*8+0xc0] vmovaps zmm2, zmm7 vmovaps zmm3, zmm7 vpermi2d zmm2, zmm4, zmm0 vpermt2d zmm4, zmm6, zmm0 vpermi2d zmm3, zmm5, zmm1 vpermt2d zmm5, zmm6, zmm1 vmovdqu32 [rcx+r9*4], zmm2 vmovdqu32 [rcx+r9*4+0x40], zmm3 vmovdqu32 [r8+r9*4], zmm4 vmovdqu32 [r8+r9*4+0x40], zmm5 add r9, 0x20 cmp r9, r10 jb loop
Baseline 1x	Speedup: 4.8x

The following constants were loaded into zmm registers and used as gather and permute indices:

Zmm0 (Alternative 1), zmm6 (Alternative 2)

__declspec (align(64)) const __int32 gather_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

Zmm1 (Alternative 1), zmm7 (Alternative 2)

__declspec (align(64)) const __int32 gather_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

Recommendation: For best performance, replace strided loads where the stride is short, with a sequence of loads and permutes.

18.22.2 Scatter to Shuffle in Strided Stores

The following is an Scatter to Shuffle example that replaces scatter with permute and store instructions

Consider the following C code:

```
for(int i=0;i<len;i++){
     Complex_buffer[i].real = Real_buffer[i];
     Complex_buffer[i].imag = Imaginary_buffer[i];
}
```

Example 18-27. Gather to Shuffle in Strided Stores Example

Alternative 1: Intel® AVX-512 vscatterdps	Alternative 2: S2S using Intel® AVX-512 vpermi2d
loop: vpcmpeqb k1, xmm0, xmm0 lea r11, [r8+rcx*4] vpcmpeqb k2, xmm0, xmm0 vmovups zmm2, [rax+rsi*4] vmovups zmm3, [r9+rsi*4] vscatterdps [r11+zmm1*4]{k1}, zmm2 vscatterdps [r11+zmm0*4]{k2}, zmm3 add rsi, 0x10 add rcx, 0x20 cmp rsi, r10 jl loop	loop: vmovups zmm4, [rax+r8*4] vmovups zmm2, [r10+r8*4] vmovaps zmm3, zmm1 add r8, 0x10 vpermi2d zmm3, zmm4, zmm2 vpermt2d zmm4, zmm0, zmm2 vmovups [r9+rsi*4], zmm3 vmovups [r9+rsi*4+0x40], zmm4 add rsi, 0x20 cmp r8, r11 jl loop
Baseline 1x	Speedup: 4.4x

The following constants were used as scatter indices:

Zmm1:

__declspec (align(64)) const __int32 scatter_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

Zmm0:

__declspec (align(64)) const __int32 scatter_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

The following constants were used as permute indices:

Zmm1:

__declspec (align(64)) const __int32 first_half[16] = {0, 16, 1, 17, 2, 18, 3, 19, 4, 20, 5, 21, 6, 22, 7, 23};

Zmm0:

__declspec (align(64)) const __int32 second_half[16] = {8, 24, 9, 25, 10, 26, 11, 27, 12, 28, 13, 29, 14, 30, 15, 31};

18.22.3 Gather to Shuffle in Adjacent Loads

In cases where the gathered elements are grouped into adjacent sequences, the gather instruction can be replaced by a software sequence to improve performance.

The following example shows how to load vectors when elements are adjacent.

Notice that in this case the order of the elements in the arrays is set according to an index buffer and therefore the software optimization discussed in Section 18.22.1, “Gather to Shuffle in Strided Loads” is not applicable in this case.

Consider the following C code:

```
typedef struct{
        double var[4];
} ElemStruct;

const int* indices = Indices;
const ElemStruct *in = (const ElemStruct*) InputBuffer;
double* restrict out = OutputBuffer;

for (int i = 0; i < width; i++){
      for (int j = 0; j < 4; j++){
           out[i*4+j] = in[indices[i]].var[j];
     }
}
```

Example 18-28. Gather to Shuffle in Adjacent Loads Example

Alternative 1: vgatherdpd Implementation	Alternative 2: Load and Masked broadcast
loop: vpbroadcastd ymm3, [r9+rsi*4] mov r15d, esi vpbroadcastd xmm2, [r9+rsi*4+0x4] add rsi, 0x2 vpbroadcastd ymm3{k1}, xmm2 vpmulld ymm4, ymm3, ymm1 vpaddd ymm5, ymm4, ymm0 vpcmpeqb k2, xmm0, xmm0 shl r15d, 0x2 movsxd r15, r15d vpxord zmm6, zmm6, zmm6 vgatherdpd zmm6{k2}, [r10+ymm5*1] vmovups [r11+r15*8], zmm6 cmp rsi, rdi jl loop	loop: movsxd r11, [r10+rcx*4] shl r11, 0x5 vmovupd ymm0, [r9+r11*1] movsxd r11, [r10+rcx*4+0x4] shl r11, 0x5 vbroadcastf64x4 zmm0{k1}, [r9+r11*1] mov r11d, ecx shl r11d, 0x2 add rcx, 0x2 movsxd r11, r11d vmovups [r8+r11*8], zmm0 cmp rcx, rsi jl loop
Baseline 1x	Speedup: 2.2x

The following constants were used in the vgatherdpd implementation:

ymm0:

 __declspec (align(64)) const __int32 index_inc[8] = {0, 8, 16, 24, 0, 8, 16, 24};

ymm1:

__declspec (align(64)) const __int32 index_scale[8] = {32, 32, 32, 32, 32, 32, 32, 32};

K1 register value is 0xF0.

18.23 DATA ALIGNMENT

This section explains the benefit of aligning data when using the Intel AVX-512 instructions and proposes some methods to improve performance when such alignment is not possible. Most examples in this section are variations of the SAXPY kernel. SAXPY is the Scalar Alpha * X + Y algorithm.

The C code below is a C implementation of SAXPY.

```
for (int i = 0; i < n; i++)
{
c[i] = alpha * a[i] + b[i];
}
```

18.23.1 Align Data to 64 Bytes

Aligning data to vector length is recommended. For best results, when using Intel AVX-512 instructions, align data to 64-bytes.

When doing a 64-byte Intel AVX-512 unaligned load/store, every load/store is a cache-line split, since the cache-line is 64 bytes. This is double the cache line split rate of Intel AVX2 code that uses 32-byte registers. A high cache-line split rate in memory-intensive code can cause poor performance.

The following table shows how the performance of the memory intensive SAXPY code is affected by misaligning input and output buffers. The data in the table is based on the following code.

Example 18-29. Data Alignment

```
__asm {
        mov rax, src1
        mov rbx, src2
        mov rcx, dst
        mov rdx, len
        xor rdi, rdi
        vbroadcastss zmm0, alpha
mainloop:
        vmovups zmm1, [rax]
        vfmadd213ps zmm1, zmm0, [rbx]
        vmovups [rcx], zmm1

        vmovups zmm1, [rax+0x40]
        vfmadd213ps zmm1, zmm0, [rbx+0x40]
        vmovups [rcx+0x40], zmm1

        vmovups zmm1, [rax+0x80]
        vfmadd213ps zmm1, zmm0, [rbx+0x80]
        vmovups [rcx+0x80], zmm1

        vmovups zmm1, [rax+0xC0]
        vfmadd213ps zmm1, zmm0, [rbx+0xC0]
        vmovups [rcx+0xC0], zmm1

        add rax, 256
        add rbx, 256
        add rcx, 256
        add rdi, 64
        cmp rdi, rdx
        jl mainloop
}
```

The following table summarizes the data alignment effects on SAXPY performance with speedup values for the various options.

Table 18-8. Data Alignment Effects on SAXPY Performance vs. Speedup Value

Data Alignment Effects on SAXPY Performance	Speedup
Alternative 1: Both sources and the destination are 64-byte aligned.	Baseline, 1.0
Alternative 2: Both sources are 64-byte aligned, destination has a 4 byte offset from the alignment.	0.66x
Alternative 3: Both sources and the destinations have 4 bytes offset from the alignment.	0.59x
Alternative 4: One source has a 4 byte offset from the alignment, the other source and the destination are 64-byte aligned.	0.77x

18.24 DYNAMIC MEMORY ALLOCATION AND MEMORY ALIGNMENT

Consider the following structure:

```
float3_SOA {
  __declspec(align(64)) float x[16];
  __declspec(align(64)) float y[16];
 };
```

The memory allocated for the structure is aligned to 64 bytes if you use this structure as follows:

```
float3_SOA f;
```

However, if you use dynamic memory allocation as follows, the declspec directive is ignored and the 64-byte memory alignment is not guaranteed:

```
float3_SOA* stPtr = new float3_SOA();
```

In this case, you should use dynamic aligned memory allocation and/or redefine operator *new*.

Recommendation: Align data to 64 bytes, when possible, using the following guidelines.

- Use dynamic data alignment using the _mm_malloc intrinsic instruction with the Intel® Compiler, or _aligned_malloc of the Microsoft* Compiler. For example:

  ```
  //dynamically allocating 64byte aligned buffer with 2048 float elements.
  InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 64);
  ```

- Use static data alignment using __declspec(align(64)). For example:

  ```
  //Statically allocating 64byte aligned buffer with 2048 float elements.
  __declspec(align(64)) float InputBuffer[2048];
  ```

18.25 DIVISION AND SQUARE ROOT OPERATIONS

It is possible to speed up single-precision divide and square root calculations using the VRSQRT14PS/VRSQRT14PD and VRCP14PS/VRCP14PD instructions. These instructions yield an approximation (with 14 bits accuracy) of the Reciprocal Square Roots / Reciprocal Divide of their input.

The Intel AVX-512 implementation of these instructions is pipelined and has:

- For 256-bit vectors: latency of 4 cycles with a throughput of one instruction every cycle.
- For 512-bit vectors: latency of 6 cycles with a throughput of one instruction every 2 cycles.

Skylake microarchitecture introduces the packed-double (PD) variants of reciprocal square-root and reciprocal divide: VRSQRT14PD and VRCP14PD (respectively).

The VRSQRT14PS/VRSQRT14PD and VRCP14PS/VRCP14PD instructions can be used with a single Newton-Raphson iteration or other polynomial approximation to achieve almost the same precision as the VDIVPS and VSQRTPS instructions (see the Intel® 64 and IA-32 Architectures Software Developer's Manuals for more information on these instructions), and may yield a much higher throughput.

If the full precision (IEEE) must be maintained, a low latency and high throughput can be achieved due to the significant performance improvement of the Skylake microarchitecture to DIVPS and SQRTPS, comparing to their performance on previous microarchitectures. This is illustrated in Figure 18-11.

NOTE

In some cases, when the divide or square root operations are part of a larger algorithm that hides some of the latency of these operations, the approximation with Newton-Raphson can slow down execution, because more micro-ops, coming from the additional instructions, fill the pipe.

The following sections show the operations with recommended calculation methods depending on the desired accuracy level.

NOTE

There are two definitions for approximation error of a value and it's approximation ν_{approx}:

Absolute error = $|\nu - \nu_{approx}|$

Relative error = $|\nu - \nu_{approx}| / |\nu|$

In this chapter, the "number of bits" error is relative, and not the error of absolute values.

The value ν to which we compare our approximation should be as accurate as possible, better double accuracy.

18.25.1 Divide and Square Root Approximation Methods

Table 18-9. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Single Precision)

Operation	Accuracy	Recommended Method
Divide	24 bits (IEEE)	DIVPS
	23 bits	RCP14PS + MULPS + 1 Newton-Raphson iteration
	14 bits	RCP14PS + MULPS
Reciprocal Square Root	22 bits	SQRTPS + DIVPS
	23 bits	RSQRT14PS + 1 Newton-Raphson iteration
	14 bits	RSQRT14PS
Square Root	24 bits (IEEE)	SQRTPS
	23 bits	RSQRT14PS + MULPS + 1 Newton-Raphson iteration
	14 bits	RSQRT14PS + MULPS

Table 18-10. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Double Precision)

Operation	Accuracy	Recommended Method
Divide	53 bits (IEEE)	DIVPD
	52 bits	RCP14PD + MULPD + 2 Newton-Raphson iterations
	26 bits	RCP14PD + MULPD + 1 Newton-Raphson iterations
	14 bits	RCP14PD + MULPD

Table 18-10. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Double Precision)

Reciprocal Square Root	53 bits (IEEE)	SQRTPD + DIVPD
	52 bits	RSQRT14PD+2 N-R + error correction or SQRTPD + DIVPD
	50 bits	RSQRT14PD + Polynomial approximation
	26 bits	RSQRT14PD+1 N-R
	14 bits	RSQRT14PD
Square Root	51 bits (IEEE)	SQRTPD
	52 bits	RSQRT14PD + MULPD + Polynomial approximation
	26 bits	RSQRT14PD + MULPD + 1 N-R
	14 bits	RSQRT14PD + MULPD

18.25.2 Divide and Square Root Performance

Performance of vector divide and square root operations on Broadwell and Skylake microarchitectures is shown below.

Table 18-11. 256-bit Intel AVX2 Divide and Square Root Instruction Performance

Broadwell Microarchitecture	**DIVPS**	**SQRTPS**	**DIVPD**	**SQRTPD**
Latency	17	21	23	35
Throughput	10	14	16	28
Skylake Microarchitecture	**DIVPS**	**SQRTPS**	**DIVPD**	**SQRTPD**
Latency	11	12	14	18
Throughput	5	6	8	12

Table 18-12. 512-bit Intel AVX-512 Divide and Square Root Instruction Performance

Skylake Microarchitecture	**DIVPS**	**SQRTPS**	**DIVPD**	**SQRTPD**
Latency	17	19	23	31
Throughput	10	12	16	24

18.25.3 Approximation Latencies

This section shows the latency and throughput for the approximation methods, and DIV and SQRT instructions. The tables below show that in most cases the throughput gain of the approximation methods is (at least) double that of their IEEE counterparts, in simple loops that compute division or square root.

The throughput benefits of approximation sequences are diminished when the loop iterations contain a lot of other computation (besides divide or square root).

As a rule of thumb, approximations of near-IEEE accuracy are recommended when the loop iteration contains no more than 8-10 additional single precision operations, or no more than 12-15 additional double precision operations. The tables below show that these accurate approximations are beneficial for throughput optimizations only. The less accurate approximations can help with latency, as well as throughput.

It should also be mentioned that Newton-Raphson approximations do not handle the following special cases correctly: denormal inputs, zeroes, or Infinities. Some sequences also lose accuracy for near-denormal inputs, due to underflow in intermediate steps. While zero and Infinity inputs are relatively easy to fix with a few additional operations (as done in some of the sequences below), denormal divisors cannot be addressed without significant performance impact. The approximation sequences work best for “middle-of-the-range” inputs that are not close to overflow or underflow thresholds.

The table below shows the latency and throughput of single precision Intel AVX-512 divide and square root instructions, compared to the approximation methods on Skylake microarchitecture.

Table 18-13. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Single Precision

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Divide (a/b)	DIVPS	24 bits (IEEE)	5	11	10	17
	RCP14PS + MULPS + 1 Newton-Raphson Iteration	23 bits	2	16	3	20
	RCP14PS + MULPS	14 bits	1	8	2	10-12
Square root	SQRTPS	24 bits (IEEE)	6	12	12	19
	RSQRT14PS + MULPS + 1 Newton-Raphson Iteration	23 bits	3	16	5	20
	RSQRT14PS + MULPS	14 bits	2	9	3	12
Reciprocal square root	SQRTPS + DIVPS	22 bits	11	23	22	36
	RSQRT14PS + 1 Newton-Raphson Iteration	23 bits	3.67	20	4.89	25
	RSQRT14PS	14 bits	1	4	2	6

Table 18-14. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Divide (a/b)	DIVPD	53 bits (IEEE)	8	14	16	23
	RCP14PD + MULPD + 2 Newton-Raphson Iterations	22 bits	3.2	27	4.7	28.4
	RCP14PD + MULPD + 1 Newton-Raphson Iteration	26 bits	2	16	3	20
	RCP14PD + MULPD	14 bits	1	8	2	10-12
Square root	SQRTPD	53 bits (IEEE)	12	18	24	31
	RSQRT14PD + MULPD + Polynomial Approximation	22 bits	4.82	24.54[1]	6.4	28.48[1]
	RSQRT14PD + MULPD + 1 N-R	26 bits	3.76	17	5	20
	RSQRT14PD + MULPD	14 bits	2	9	3	12
Reciprocal square root	SQRTPD + DIVPD	51 bits	20	32	40	53
	RSQRT14PD + 2-NR + error correction	52 bits	5	29.38	6.53	34
	RSQRT14PD+2 N-R	50 bits	3.79	25.73	5.51	30
	RSQRT14PD+1 N-R	26 bits	2.7	18	4.5	21.67
	RSQRT14PD	14 bits	1	4	2	6

NOTES:

1. These numbers are not rounded because their code sequence contains several FMA (Fused-multiply-add) instructions, which have a varying latency of 4/6. Therefore the latency for these sequences is not necessarily fixed.

18.25.4 Code Snippets

Example 18-30. Vectorized 32-bit Float Division

Single Precision, Divide, 24 Bits (IEEE)

```
float a = 10;
float b = 5;

__asm {
    vbroadcastss zmm0, a            // fill zmm0 with 16 elements of a
    vbroadcastss zmm1, b            // fill zmm1 with 16 elements of b
    vdivps zmm2, zmm0, zmm1         // zmm2 = 16 elements of a/b
}
```

Single Precision, Divide, 23 Bits

```
/*  Input:
        zmm0 = vector of a's
        zmm1 = vector of b's
    Output:
        zmm3 = vector of a/b
*/

__asm {
    vrcp14ps zmm2, zmm1
    vmulps zmm3, zmm0, zmm2
    vmovaps zmm4, zmm0
    vfnmadd231ps zmm4, zmm3, zmm1
    vfmadd231ps zmm3, zmm4, zmm2
}
```

Single Precision, Divide, 14 Bits

```
/*  Input:
        zmm0 = vector of a's
        zmm1 = vector of b's
    Output:
        zmm2 = vector of a/b
*/

__asm {
    vrcp14ps zmm2, zmm1
    vmulps zmm2, zmm0, zmm2
}
```

Example 18-31. Reciprocal Square Root

Single Precision, Reciprocal Square Root, 22 Bits

```
/*  Input:
        zmm0 = vector of a's
        zmm1 = vector of 1's
    Output:
        zmm2 = vector of 1/sqrt (a)
*/

float one = 1.0;

__asm {
    vbroadcastss zmm1, one      // zmm1 = vector of 16 1's
    vsqrtps zmm2, zmm0
    vdivps zmm2, zmm1, zmm2
}
```

Single Precision, Reciprocal Square Root, 23 Bits

```
/*  Input:
        zmm0 = vector of a's
    Output:
        zmm2 = vector of 1/sqrt (a)
*/

float half = 0.5;

__asm {
    vbroadcastss zmm1, half     // zmm1 = vector of 16 0.5's
    vrsqrt14ps zmm2, zmm0
    vmulps zmm3, zmm0, zmm2
    vmulps zmm4, zmm1, zmm2
    vfnmadd231ps zmm1, zmm3, zmm4
    vfmsub231ps zmm3, zmm0, zmm2
    vfnmadd231ps zmm1, zmm4, zmm3
    vfmadd231ps zmm2, zmm2, zmm1
}
```

Single Precision, Reciprocal Square Root, 14 Bits

```
/*  Input:
        zmm0 = vector of a's
    Output:
        zmm2 = vector of 1/sqrt (a)
*/

__asm {
    vrsqrt14ps zmm2, zmm0
}
```

Example 18-32. Square Root

Single Precision, Square Root, 24 Bits (IEEE)

/* Input:
        zmm0 = vector of a's
    Output:
        zmm2 = vector of sqrt (a)
*/
 	
__asm {
    vsqrtps zmm2, zmm0
}

Single Precision, Square Root, 23 Bits	**Single Precision, Square Root, 14 Bits**

/* Input:
        zmm0 = vector of a's
    Output:
        zmm0 = vector of sqrt (a)
*/
 	
float half = 0.5;
 	
__asm {
    vbroadcastss zmm3, half
    vrsqrt14ps zmm1, zmm0
    vfpclassps k2, zmm0, 0xe
    vmulps zmm2, zmm0, zmm1, {rn-sae}
    vmulps zmm1, zmm1, zmm3
    knotw k3, k2
    vfnmadd231ps zmm0{k3}, zmm2, zmm2
    vfmadd213ps zmm0{k3}, zmm1, zmm2
}
	/* Input:
	        zmm0 = vector of a's
	    Output:
	        zmm0 = vector of sqrt (a)
	*/
	
	__asm {
	    vrsqrt14ps zmm1, zmm0
	    vfpclassps k2, zmm0, 0xe
	    knotw k3, k2
	    vmulps zmm0{k3}, zmm0, zmm1
	}

Example 18-33. Dividing Packed Doubles

Double Precision, Divide, 53 Bits (IEEE)

```
/*  Input:
        zmm0 = vector of a's
        zmm1 = vector of b's
    Output:
        zmm2 = vector of a/b
*/

__asm {
    vdivpd zmm2, zmm0, zmm1
}
```

Double Precision, Divide, 52 Bits

```
/*  Input:
        zmm15 = vector of a's
        zmm0 = vector of b's
    Output:
        zmm0 = vector of a/b
*/

double One = 1.0;

__asm {
    vrcp14pd zmm1, zmm0
    vmovapd zmm4, zmm0
    vbroadcastsd zmm2, one
    vfnmadd213pd zmm0, zmm1, zmm2, {rn-sae}
    vfpclasspd k2, zmm1, 0x1e
    vfmadd213pd zmm0, zmm1, zmm1, {rn-sae}}
    knotw k3, k2
    vfnmadd213pd zmm4, zmm0, zmm2, {rn-sae}
    vblendmpd zmm0 {k2}, zmm0, zmm1
    vfmadd213pd zmm0 {k3}, zmm4, zmm0, {rn-sae}
    vmulpd zmm0, zmm0, zmm15
}
```

Double Precision, Divide, 26 Bits

```
/*  Input:
        zmm0 = vector of a's
        zmm1 = vector of b's
    Output:
        zmm3 = vector of a/b
*/

__asm {
    vrcp14pd zmm2, zmm1
    vmulpd zmm3, zmm0, zmm2
    vmovapd zmm4, zmm0
    vfnmadd231pd zmm4, zmm3, zmm1
    vfmadd231pd zmm3, zmm4, zmm2
}
```

Double Precision, Divide, 14 Bits

```
/*  Input:
        zmm0 = vector of a's
        zmm1 = vector of b's
    Output:
        zmm2 = vector of a/b
*/

__asm {
    vrcp14pd zmm2, zmm1
    vmulpd zmm2, zmm0, zmm2
}
```

Example 18-34. Reciprocal Square Root of Doubles

Double Precision, Reciprocal Square Root, 51 Bits

```
/*  Input:
          zmm0 = vector of a's
          zmm1 = vector of 1's
    Output:
          zmm0 = vector of 1/sqrt (a)
*/
__asm {
    vsqrtpd zmm0, zmm0
    vdivpd zmm0, zmm1, zmm0
}
```

Double Precision, Reciprocal Square Root, 52 Bits

```
/*  Input:
          zmm4 = vector of a's
    Output:
          zmm0 = vector of 1/sqrt (a)
*/
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 one[ ] =
{DUP8_DECL(0x3FF0000000000000)};
ALIGNTO(64) __int64 dc1[ ] =
{DUP8_DECL(0x3FE0000000000000)};
ALIGNTO(64) __int64 dc2[ ] =
{DUP8_DECL(0x3FD8000004600001)};
ALIGNTO(64) __int64 dc3[ ] =
{DUP8_DECL(0x3FD4000005E80001)};
__asm {
    vbroadcastsd zmm4, big_num
    vmovapd zmm0, one
    vmovapd zmm5, dc1
    vmovapd zmm6, dc2
    vmovapd zmm7, dc3

    vrsqrt14pd zmm3, zmm4
    vfpclasspd k1, zmm4, 0x5e
    vmulpd zmm1, zmm3, zmm4, {rn-sae}
    vfnmadd231pd zmm0, zmm3, zmm1
    vfmsub231pd zmm1, zmm3, zmm4, {rn-sae}
    vfnmadd213pd zmm1, zmm3, zmm0
    vmovups zmm0, zmm7
    vmulpd zmm2, zmm3, zmm1
    vfmadd213pd zmm0, zmm1, zmm6
    vfmadd213pd zmm0, zmm1, zmm5
    vfmadd213pd zmm0, zmm2, zmm3
    vorpd zmm0{k1}, zmm3, zmm3
}
```

Double Precision, Reciprocal Square Root, 50 Bits

```
/*  Input:
          zmm3 = vector of a's
    Output:
          zmm4 = vector of 1/sqrt (a)
*/
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 one[ ] =
{DUP8_DECL(0x3FF0000000000000)};
ALIGNTO(64) __int64 dc1[ ] =
{DUP8_DECL(0x3FE0000000000000)};
ALIGNTO(64) __int64 dc2[ ] =
{DUP8_DECL(0x3FD8000004600001)};
ALIGNTO(64) __int64 dc3[ ] =
{DUP8_DECL(0x3FD4000005E80001)};
__asm {
    vmovapd zmm5, one
    vmovapd zmm6, dc1
    vmovapd zmm8, dc3
    vmovapd zmm7, dc2

    vrsqrt14pd zmm2, zmm3
    vfpclasspd k1, zmm3, 0x5e
    vmulpd zmm0, zmm2, zmm3, {rn-sae}
    vfnmadd231pd zmm0, zmm2, zmm5
    vmulpd zmm1, zmm2, zmm0
    vmovapd zmm4, zmm8
    vfmadd213pd zmm4, zmm0, zmm7
    vfmadd213pd zmm4, zmm0, zmm6
    vfmadd213pd zmm4, zmm1, zmm2
    vorpd zmm4{k1}, zmm2, zmm2
}
```

Example 18-34. Reciprocal Square Root of Doubles (Contd.)

Double Precision, Reciprocal Square Root, 26 Bits	Double Precision, Reciprocal Square Root, 14 Bits

/* Input:
zmm0 = vector of a's
Output:
zmm1 = vector of 1/sqrt (a)
*/
 	
double half = 0.5;
 	
__asm {
vrsqrt14pd zmm1, zmm0
vmulpd zmm0, zmm0, zmm1
vbroadcastsd zmm3, half
vmulpd zmm2, zmm1, zmm3
vfnmadd213pd zmm2, zmm0, zmm3
vfmadd213pd zmm1, zmm2, zmm1
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm2 = vector of 1/sqrt (a)
	*/
	
	__asm {
	vrsqrt14pd zmm2, zmm0
	}

Example 18-35. Square Root of Packed Doubles

Double Precision, Square Root, 53 Bits (IEEE)	Double Precision, Square Root, 52 Bits

/* Input:
zmm0 = vector of a's
Output:
zmm2 = vector of sqrt (a)
*/
 	
__asm {
vsqrtpd zmm2, zmm0
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm0 = vector of sqrt (a)
	*/
	
	double half = 0.5;
	
	__asm {
	vbroadcastsd zmm4, half
	vrsqrt14pd zmm1, zmm0
	vfpclasspd k2, zmm0, 0xe
	vmulpd zmm2, zmm0, zmm1, {rn-sae}
	vmulpd zmm1, zmm1, zmm4
	knotw k3, k2
	vmovapd zmm3, zmm4
	vfnmadd231pd zmm3, zmm1, zmm2, {rn-sae}
	vfmadd213pd zmm2, zmm3, zmm2, {rn-sae}
	vfmadd213pd zmm1, zmm3, zmm1, {rn-sae}
	vfnmadd231pd zmm0 {k3}, zmm2, zmm2, {rn-sae}
	vfmadd213pd zmm0 {k3}, zmm1, zmm2
	}

Example 18-35. Square Root of Packed Doubles (Contd.)

Double Precision, Square Root, 26 Bits	Double Precision, Square Root, 14 Bits

`/* Input:`
`        zmm0 = vector of a's`
`    Output:`
`        zmm0 = vector of sqrt (a)`
`*/`
 	
`// duplicates x eight times`
`#define DUP8_DECL(x) x, x, x, x, x, x, x, x`
 	
`// used for aligning data structures to n bytes`
`#define ALIGNTO(n) __declspec(align(n))`
 	
`ALIGNTO(64) __int64 OneHalf[ ] =`
`{DUP8_DECL(0X3FE0000000000000)};`
 	
`__asm {`
`    vrsqrt14pd zmm1, zmm0`
`    vfpclasspd k2, zmm0, 0xe`
`    knotw k3, k2`
`    vmulpd zmm0 {k3}, zmm0, zmm1`
`    vmulpd zmm1, zmm1, ZMMWORD PTR [OneHalf]`
`    vfnmadd213pd zmm1, zmm0, ZMMWORD PTR [OneHalf]`
`    vfmadd213pd zmm0 {k3}, zmm1, zmm0`
`}`
	`/* Input:`
	`        zmm0 = vector of a's`
	`    Output:`
	`        zmm0 = vector of sqrt (a)`
	`*/`
	
	`__asm {`
	`    vrsqrt14pd zmm1, zmm0`
	`    vfpclasspd k2, zmm0, 0xe`
	`    knotw k3, k2`
	`    vmulpd zmm0 {k3}, zmm0, zmm1`
	`}`

18.26 CLDEMOTE

Using the CLDEMOTE instruction, a processor puts a cache line into the last shared level of the cache hierarchy so that other CPU cores 'find' the same cache line in the last shared level and expensive cross-core snoop is avoided. The most significant advantage of CLDEMOTE is that multiple consumers can access the shared cache line amortizing each snoop request portion.

18.26.1 Producer-Consumer Communication in Software

In a multiprocessor environment, data sharing between the producers and consumers is an undisputed event. A cache hierarchy solves the major problem of accessing the line from the main memory resulting in faster data transfers. Typical cache hierarchy contains:

- Private L1 data and L1 instruction cache.
- A shared L2 cache for sibling hardware thread.
- A common L3 cache for all the CPU cores.

When a producer consumes data from the I/O or produces it, it is brought into the producer's L1 cache. Consumers read the data by initiating read requests, translating it into cross-core snoops, request, and response events. Consumers report L3 cache miss events and producer cores responding to the consumer core's snoop request. Multiplexing these cross-cores requests and responses when dealing with multiple consumers is detrimental.

18.27 TIPS ON COMPILER USAGE

This section explains some of the important compiler options that can be used with the Intel compiler to derive the best performance on a Skylake server. For complete information on the compiler options and tuning tips, see the main product documentation at: https://software.intel.com/en-us/intel-software-technical-documentation. For example, the Intel® C++ Compiler 17.0 Developer Guide and Reference can be found here: https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide.

Many options have names that are the same on Linux* and Windows*, except that the Windows* form starts with an initial Q. Within text, such option names are shown as [Q]option-name.

The default optimization level is O2 (unless -g is specified, in which case the default is O0). Level O2 enables many compiler optimizations including vectorization. Optimization level O3 is recommended for loop-intensive and HPC applications, as it enables more aggressive loop and memory-access optimizations, such as loop fusion and loop blocking to allow more efficient use of the caches.

For best performance on Skylake server microarchitecture, applications should be compiled with the processor-specific option [Q]xCORE-AVX512. Note that an executable compiled with these options will not run on non-Intel processors or on Intel processors that support only lower instruction sets.

For users who want to generate a common binary that can be executed on Skylake server microarchitecture and the Intel® Xeon Phi™ processors based on Knights Landing microarchitecture, use the option [Q]xCOMMON-AVX512. Note that this option has a performance cost on both Skylake server microarchitecture and Intel® Xeon Phi™ processors compared with executables generated with the target-specific options [Q]xCORE-AVX512 on Skylake server, and [Q]xMIC-AVX512 on Intel® Xeon Phi™ processors.

In addition, users can tune the zmm code generation done by the compiler for Skylake server microarchitecture using the additional option -qopt-zmm-usage=low|high (/Qopt-zmm-usage:low|high on Windows). The argument value of low provides a smooth transition experience from AVX2 ISA to AVX512 ISA on a Skylake server microarchitecture target, such as for enterprise applications. Tuning for ZMM instruction use via explicit vector syntax such as #pragma omp simd simdlen() is recommended. The argument value of high is recommended for applications, such as HPC codes, that are bounded by vector computation to achieve more compute per instruction through use of the wider vector operations. The default value is low for Skylake server microarchitecture-family compilation targets, such as [Q]xCORE-AVX512 and high for CORE/MIC AVX512 combined compilation targets such as [Q]xCOMMON-AVX512.

It is also possible to generate a fat binary that supports multiple instruction sets by using the [Q]axtarget option. For example, if the application is compiled with [Q]axCORE-AVX512,CORE-AVX2 the compiler might generate specialized code for the Skylake server microarchitecture and AVX2 targets, while also generating a default code path that will run on any Intel or compatible, non-Intel processor that supports at least Intel® Streaming SIMD Extensions 2 (Intel® SSE2). At runtime, the application automatically detects whether it is running on an Intel processor. If so, it selects the most appropriate code path for Intel processors; if not, the default code path is selected. It is also important to note that irrespective of the options used, the compiler might insert calls into specialized library routines, such as optimized versions of memset/memcpy, that will dispatch to the appropriate codepath at runtime based on processor detection.

The option -qopt-report[n] (/Qopt-report[:n] on Windows) generates a report on the optimizations performed by the compiler, by default it is written to a file with a .optrpt file extension. n specifies the level of detail, from 0 (no report) to 5 (maximum detail). The option -qopt-report-phase (/Qopt-report-phase on Windows) controls report generation from various compiler phases, but it is recommended to use the default setting where the report is generated for all compiler phases. The report is a useful tool to gain insight into the performance optimizations performed, or not performed, by the compiler, and also to understand the interactions between multiple optimizations such as inlining, OpenMP* parallelization, loop optimizations (such as loop distribution or loop unrolling) and vectorization. The report is based on static compiler analysis. Hence the reports are most useful when correlated with dynamic performance analysis tools, such as Intel® VTune™ Amplifier or Vectorization Advisor (part of Intel® Advisor XE), that do hotspot analysis and provide other dynamic information. Once this information is available, the optimization information can be studied for hotspots (functions/loopnests) in compiler reports. It is important to note that the compiler can generate multiple versions of loop-nests, so it is useful to correlate the analysis with the version actually executed at runtime. The phase ordering of the compiler loop optimizations is intended to enable optimal vectorization. Often, understanding the loop optimization parameters helps to further tune performance. In many cases, finer control of these loop optimizations is available via pragmas, directives, and options.

If the application contains OpenMP pragmas or directives, it can be compiled with -qopenmp (/Qopenmp on Windows) to enable full OpenMP based multi-threading and vectorization. Alternatively, the SIMD vectorization features of OpenMP alone can be enabled by using the option -qopenmp-simd (/Qopenmp-simd on Windows).

For doing studies where compiler-based vectorization has to be turned off completely, use the options

-no-vec -no-simd -qno-openmp-simd (/Qvec- /Qsimd- /Qopenmp-simd- on Windows).

Data alignment plays an important role in improving the efficiency of vectorization. This usually involves two distinct steps from the user or application:

- Align the data.

  When compiling a Fortran program, it is possible to use the option -align array64byte (/align:array64byte on Windows) to align the start of most arrays at a memory address that is divisible by 64. For C/C++ programs, data allocation can be done using routines such as _mm_malloc(…, 64) to align the return-value pointer at 64 bytes. For more information on data alignment, see https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.
- Convey the alignment information to the compiler using appropriate clauses, pragmas, and directives.

Compiler-based software data prefetching can be enabled with the options -O3 -xcore-avx512 -qopt-prefetch[=n] (-O3 /QxCORE-AVX512 /Qopt-prefetch[=n] on Windows), for n=0 (no prefetching) to 5 (maximal prefetching). Using a value of n=5 enables aggressive compiler prefetching, disregarding any hardware prefetching, for strided loads/stores and indexed loads/stores which appear inside loops. Using a value of n=2 reduces the amount of compiler prefetching and restricts it only to direct memory accesses where the compiler heuristics determine that the hardware prefetcher may not be able to handle well. It is recommended to try values of n=2 to 5 to determine the best prefetching strategy for a particular application. It is also possible to use the -qopt-prefetch-distance=n1[,n2] (/Qopt-prefetch-distance=n1[,n2] on Windows) option to fine-tune application performance.

- Useful values to try for n1: 0,4,8,16,32,64.
- Useful values to try for n2: 0,1,2,4,8.

Loop-nests that have a relatively low trip-count value at runtime in hotspots can sometimes lead to sub-optimal AVX-512 performance unless the trip-count is conveyed to the compiler. In many such cases, the compiler will be able to generate better code and deliver better performance if values of loop trip-counts, loop-strides, and array extents (such as for Fortran multi-dimensional arrays) are all known to the compiler. If that is not possible, it may be useful to add appropriate loop_count pragmas to such loops.

Interprocedural optimization (IPO) is enabled using the option -ipo (/Qipo on Windows). This option can be enabled on all the source-files of the application or it can be applied selectively to the source files containing the application hot-spots. IPO permits inlining and other inter-procedural optimizations to happen across these multiple source files. In some cases, this option can significantly increase compile time and code size. Using the option -inline-factor=n (/Qinline-factor:n on Windows) controls the amount of inlining done by the compiler. The default value of n is 100, indicating 100%, or a scale factor of 1. For example, if a value of 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2, thus enabling more inlining than the default.

Profile-guided optimizations (PGO) are enabled using the options -prof-gen and -prof-use (/Qprof-gen and /Qprof-use on Windows). Typically, using PGO increases the effectiveness of using IPO.

The option -fp-model name (/fp:name on Windows) controls tradeoffs between performance, accuracy and reproducibility of floating-point results at a high level. The default value for name is fast=1. Changing it to fast=2 enables more aggressive optimizations at a slight cost in accuracy or reproducibility. Using the value precise for name disallows optimizations that might produce slight variations in floating-point results. When name is double, extended or source, intermediate results are computed in the corresponding precision. In most situations where enhanced floating-point consistency and reproducibility are needed -fp-model precise -fp-model source (/fp:precise /fp:source on Windows) are recommended.

The option -fimf-precision=name (/Qimf-precision=name on Windows) is used to set the accuracy for math library functions. The default is OFF, which means that the compiler uses its own default heuristics. Possible values of name are high, medium, and low. Reduced precision might lead to increased performance and vice versa, particularly for vectorized code. The options -[no-]prec-div and -[no-]prec-sqrt improve[reduce] precision of floating-point divides and square root computations. This may slightly degrade [improve] performance. For more details on floating-point options, see https://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler.

The option -[no-]ansi-alias (/Qansi-alias[-] on Windows) enables [disables] ANSI and ISO C Standard aliasing rules. By default, this option is enabled on Linux, but disabled on Windows. On Windows, especially for C++ programs, adding /Qansi-alias to the compilation options enable the compiler to perform additional optimizations, particularly taking advantage of the type-based disambiguation rules of the ANSI Standard, which says for example, that pointer and float variables do not overlap.

If the optimization report specifies that compiler optimizations may have been disabled to reduce compile-time, use the option -qoverride-limits to override such disabling in the compiler and ensure optimization is applied. This can sometimes be important for applications, especially ones with functions that have big bodies. Note that using this additional option may increase compile time and compiler memory usage significantly in some cases.

The list below shows a sampling of loop-level controls available for fine-tuning optimizations - including a way to turn off a particular transformation reported by the compiler.

- #pragma simd reduction(+:sum)

  The loop is transformed as is, no other loop-optimizations will change the simd-loop.
- #pragma loop_count min(220) avg (300) max (380)

  Fortran syntax: !dir$ loop count(16)
- #pragma vector aligned nontemporal
- #pragma novector // to suppress vectorization
- #pragma unroll(4)
- #pragma unroll(0) // to suppress loop unrolling
- #pragma unroll_and_jam(2) // before an outer loop
- #pragma nofusion
- #pragma distribute_point

  If placed as the first statement right after the for-loop, distribution will be suppressed for that loop.

  Fortran syntax: !dir$ distribute point
- #pragma prefetch *:<hint>:<distance>

  Apply uniform prefetch distance for all arrays in a loop.
- #pragma prefetch <var>:<hint>:<distance>

  Fine-grained control for each array
- #pragma noprefetch [<var>]

  Turns off prefetching [for a particular array]
- #pragma forceinline (recursive)

If placed before a call, this is a hint to the compiler to recursively inline the entire call-chain.

9. Updates to Chapter 20

Change bars and **violet** text show changes to Chapter 20 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual:* Intel® Advanced Matrix Extensions (Intel® AMX).

Changes to this chapter:

- Branding corrected throughout chapter.
- Links and cross-references corrected throughout chapter.
- Updated typos throughout chapter.
- Heading Title Case Corrected throughout chapter.
- Consolidated links throughout chapter.
- Added and changed figure, table and example descriptions:
  - — Table 20-6
  - — Figure 20-10
  - — Figure 20-11
  - — Example 20-17
  - — Example 20-18
  - — Example 20-19
  - — Example 20-21
  - — Example 20-23
  - — Example 20-24
  - — Example 20-28
  - — Example 20-29
- Section 20.1
  - — Updated last paragraph with new links and phrasing.
  - — Updated location of CPUID instruction references.
- Section 20.5.5.3:
  - — Example 20-8: added space between int and m in line 2.
  - — Example 20-10:
    - Split between two pages
    - Changed
      - 17 _amx_interleaved_gemm_ass:
      - 18 amx_interleaved_gemm_ass:
      - to
      - 17
      - 18 __asm {
- Section 20.11
  - — Added text for clarity.
- Section 20.16.2
  - — Added section about optimizing the performance of Intel® Hyper-Threading technology (Intel® HT).

CHAPTER 20 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

This chapter aims to help low-level DL programmers optimally code to the metal on Intel® Xeon® Processors based on Sapphire Rapids SP microarchitecture. It extends the public documentation on Optimizing DL code with DL Boost instructions in Section 20.8.

It explains how to detect processor support in Intel® Advanced Matrix Extensions (Intel® AMX) Architecture (Section 20.1). It provides an overview of Intel AMX architecture (Section 20.2) and presents Intel AMX instruction throughput and latency (Section 20.3). It also discusses software optimization opportunities for Intel AMX (Section 20.5 through Section 20.17), TileConfig/TileRelease and compiler ABI (Section 20.18), Intel AMX state management and system software aspects (Section 20.19), and the use of Intel AMX for higher precision GEMMs (Section 20.20).

Table 20-1. Intel® AMX-Related Links

Description	URL
Intel® AMX architecture definitions in the Intel® 64 and IA-32 Architecture Software Developer's Manual	https://www.intel.com/sdm
Buildable and executable templates of code examples for this chapter.	https://github.com/intel/optimization-manual
Open VINO™ Optimization Guide	https://docs.openvino.ai/latest/openvino_docs_optimization_guide_dldt_optimization_guide.html
oneDNN GitHub	https://github.com/oneapi-src/oneDNN
oneDNN documentation	https://oneapi-src.github.io/oneDNN/
Intel® Optimization TensorFlow Installation Guide	https://www.intel.com/content/www/us/en/developer/articles/guide/optimization-for-tensorflow-installation-guide.html
PyTorch Landing Page	https://pytorch.org/
PyTorch GitHub	https://github.com/pytorch/pytorch
Intel® Neural Compressor (INC) GitHub	https://github.com/intel/neural-compressor
Tips for measuring the performance of matrix multiplication using Intel® MKL	https://www.intel.com/content/www/us/en/developer/articles/technical/a-simple-example-to-measure-the-performance-of-an-intel-mkl-function.html
Intel® AMX ABI	https://gitlab.com/x86-psABIs/x86-64-ABI/-/wikis/home
GitHub Repository	https://github.com/intel/optimization-manual
Using dynamically enabled XSTATE features in Linux user space applications	https://www.kernel.org/doc/html/latest/x86/xstate.html

Table 20-1. Intel® AMX-Related Links

Description	URL
Using dynamically enabled XSTATE features in Windows user space applications	https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getenabledxstatefeatures
	https://docs.microsoft.com/es-es/windows/win32/api/winbase/nf-winbase-enableprocessoptionalxstatefeatures
	https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getthreadenabledxstatefeaturesv
	https://docs.microsoft.com/en-us/windows/win32/api/process-threadsapi/nf-processthreadsapi-updateprocthreadattribute

20.1 DETECTING INTEL® AMX SUPPORT

Use the CPUID instruction described in Chapter 3.3 of the Intel® 64 and IA-32 Architecture Software Developer's Manual to find out whether the processor you are executing on supports Intel AMX at the hardware level.

Specifically, when issuing the CPUID instruction with EAX register set to 7 and ECX register set to 0, the instruction returns in the EDX register an indication on Intel AMX support of bits 22, 24, 25. They are all set to 0 if Intel AMX is not supported and all set to 1 if it is supported by the processor.

Next step is check whether the OS has enabled Intel AMX state. For that you first need to issue the CPUID instruction again to check whether the OS supports the XGETBV instruction, then use it to check whether the OS has enabled the Intel AMX state save/restore.

When issuing the CPUID instruction with EAX register set to 1, the instruction returns an indication of XGETBV support in bit 26 of the ECX register. If bit 26 is set, when issuing the XGETBV instruction with ECX register set to 0, the instruction returns an indication on OS support in saving and restoring Intel AMX state in bits 17 and 18 of the EAX register. Both bits should be set in order to use the Intel AMX instructions. For additional CPUID information about Intel AMX, see Chapter 3.3 of the Intel® 64 and IA-32 Architecture Software Developer's Manual

Operating systems may require calling an OS API to allocate Intel AMX state. Visit LinuxAPI and Windows APIs for more detailed information. Please see Section 20.19 for more information about Intel AMX state management.

20.2 INTEL® AMX MICROARCHITECTURE OVERVIEW

General Intel AMX microarchitecture overview is available in Chapter 18 of Volume 1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual.

20.2.1 INTEL® AMX FREQUENCIES

Discussion on the connection between max frequency, frequency license, and Instruction Set Architecture covering Intel AVX technologies up to Intel® AVX-512 Instruction Set, is available in Section 2.5.3. Intel AMX adds yet another license level whose max frequency is usually lower than that of the Intel AVX-512 license.

When the Intel AMX unit utilization is lower than 15%, the processor may exceed the nominal max frequency associated with Intel AMX license.

20.3 INTEL® AMX INSTRUCTIONS THROUGHPUT AND LATENCY

Several Intel AMX instructions are available. Two instructions (TileLoad*) load data from the memory hierarchy into the tile registers and one instruction (TileStore) stores the contents of a tile register into the DCU (Data Cache Unit–first level cache). Other instructions (TDP*) execute the matrix multiplication, operating on two input tile registers and writing the result into a third tile register. Additionally, there are some less-frequently used instructions. The following table provides the instruction throughput and latency counted in cycles.

Table 20-2. Intel® AMX Instruction Throughput and Latency

Instruction	Throughput	Latency
LDTILECFG		204
STTILECFG		19
TILETRELEASE		13
TDP/*	16	52
TILELOADD	8	45
TILELOADDT1	33	48
TILESTORED	16	
TILEZERO	0	16

NOTE

Due to the high latency of the LDTILECFG instruction we recommend issuing a single pair of LDTILECFG and TILERELEASE operations per Intel AMX-based DL layer implementation.

20.4 DATA STRUCTURE ALIGNMENT

GEMM and Convolution input/output data structures must be 64-byte aligned for optimal performance but should not be aligned to 128-byte, 256-byte, etc. For more details, see Tip 6 in *Tips for Measuring the Performance of Matrix Multiplication Using Intel® MKL*.

20.5 GEMMS / CONVOLUTIONS

20.5.1 NOTATION

The following notation is used for the matrices (A, B, C) and the dimensions (M, K, N) in matrix multiplication (GEMM).

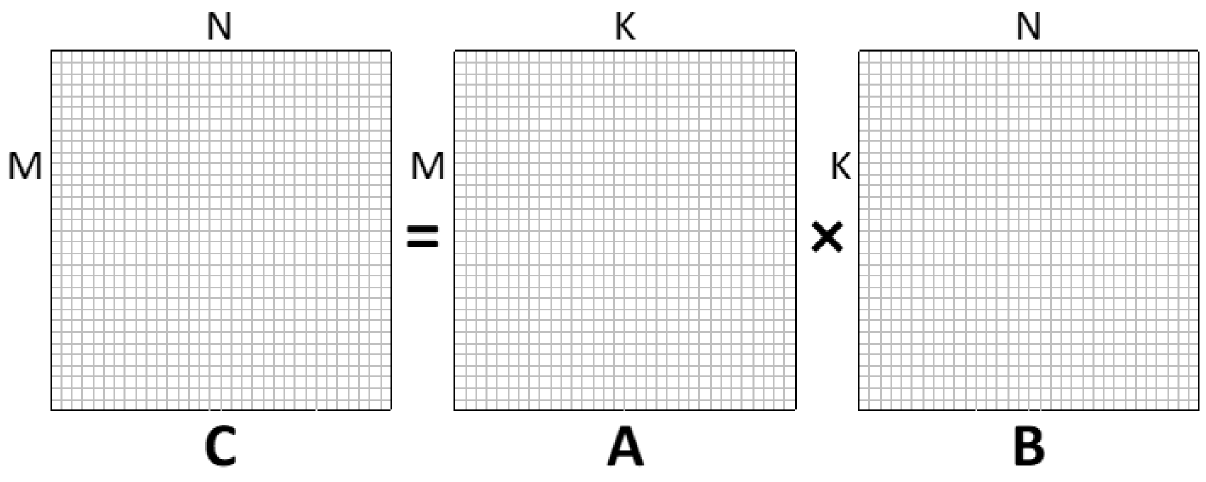


Figure 20-1. Matrix Notation

20.5.2 TILES IN THE INTEL® AMX ARCHITECTURE

The Intel AMX instruction set operates on tiles: large two-dimensional registers with configurable dimensions. The configuration is dependent on the type of tile.

- A-tiles can have between 1-16 rows and 1-MAX_TILE_K columns.
- B-tiles can have between 1-MAX_TILE_K rows and 1–16 columns.
- C-tiles can have between 1-16 rows and 1–16 columns.

MAX_TILE_K=64/sizeof(type_t), and type_t is the type of the data being operated on. Therefore, MAX_TILE_K=64 for (u)int8 data, and MAX_TILE_K=32 for bfloat16 data. The dimensions here are mathematical/logical. For mapping to tile register configuration parameters, see the Intel® Architecture Instruction Set Extensions Programming Reference referenced in Section 20.2.

The type of data residing in the tiles also varies depending on the type of tile.

A tiles and B tiles contain data of `type_t`, which can be (u)int8 or bfloat16.

- C tiles contain data of type res_type_t:
- int32 if type_t=(u)int8
- float if type_t=bfloat16

Thus, a maximum-sized tile multiplication operation for (u)int8 data type looks this way:

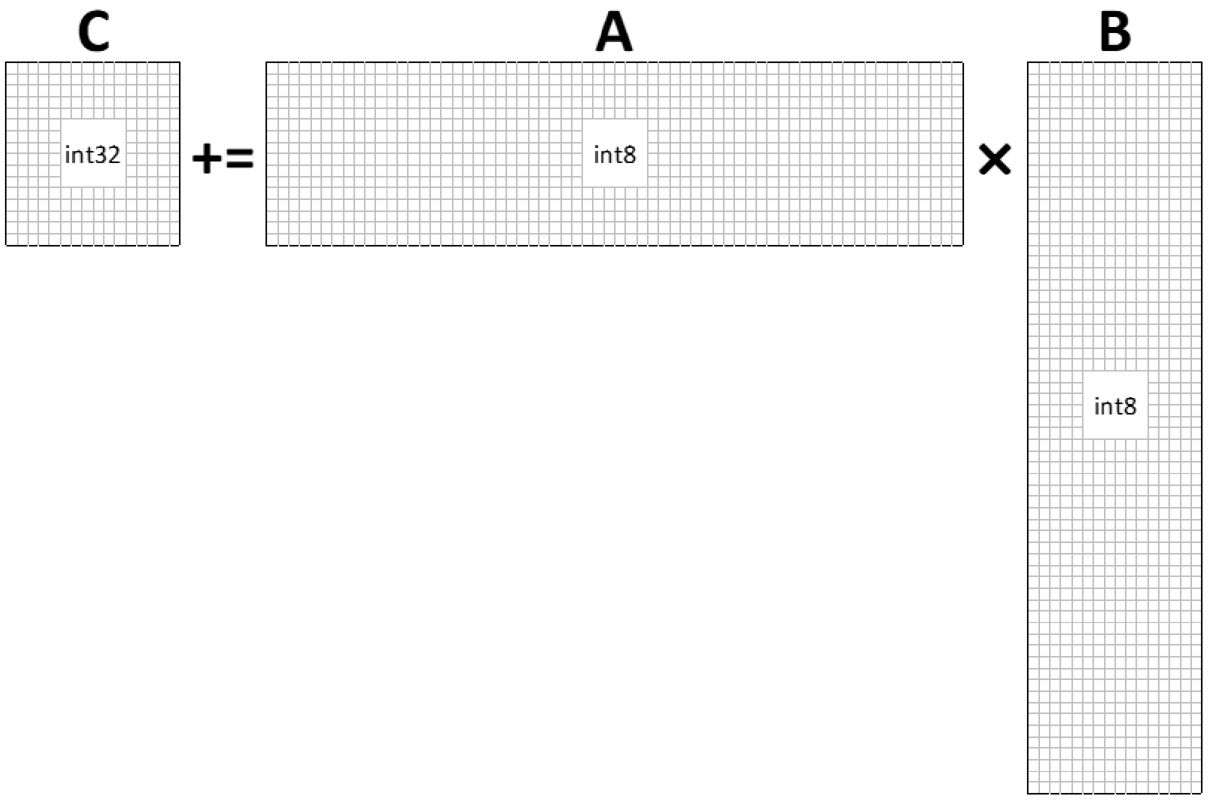


Figure 20-2. Intel® AMX Multiplication with Max-sized int8 Tiles

TileLoad and TileStore Instructions

The tiles are loaded from memory with the TileLoad instruction and stored to memory with a TileStore instruction. The TileLoad/TileStore instructions receive the following parameters:

- The destination/source tile of the TileLoad/TileStore.
- The source/destination location in memory for the TileLoad/TileStore.
- The stride (bytes) in memory between subsequent rows of the tile.

Lines 6—10 in Example 20-1 illustrate how a tile is loaded from memory.

Example 20-1. Pseudo-Code for the Tilezero, TileLoad, and TileStore Instructions

```
template<size_t rows, size_t bytes_cols> class tile {
public:
 friend void tilezero(tile& t) {
  memset(t.v, 0, sizeof(v));
 }
 friend void tileload(tile& t, void* src, size_t bytes_stride) {
  for (size_t row = 0; row < rows; ++row)
   for (size_t bcol = 0; bcol < bytes_cols; ++bcol)
    t.v[row][bcol] = static_cast<int8_t*>(src)[row*bytes_stride + bcol];
 }
friend void tilestore(tile& t, void* dst, size_t bytes_stride) {
  for (size_t row = 0; row < rows; ++row)
   for (size_t bcol = 0; bcol < bytes_cols; ++bcol)
    static_cast<int8_t*>(dst)[row*bytes_stride + bcol] = t.v[row][bcol];
 }
template <class TC, class TA, class TB>
friend void tdp(TC &tC, TA &tA, TB &tB);
private:
 int8_t v[rows][bytes_cols];
};

// clang-format on

template <class TC, class TA, class TB> void tdp(TC &tC, TA &tA, TB &tB)
}
```

For the sake of readability, a tile template class abstraction is introduced. The number of rows in the tile and the number of column bytes per row parametrizes the abstraction.

20.5.3 B MATRIX LAYOUT

Like the Intel® DL Boost use case, the B matrix must undergo a re-layout before it can be used within the corresponding Intel AMX multiply instruction. The re-layout procedure is as follows:

Example 20-2. B Matrix Re-Layout Procedure

```
#define KPACK (4/sizeof(type_t))                    // Vertical K packing into Dword

type_t B_mem_orig[K][N];                            // Original B matrix
type_t B_mem[K/KPACK][N][KPACK];                    // Re-laid B matrix

for (int k = 0; k < K; ++k)
 for (int n = 0; n < N; ++n)
  B_mem[k/KPACK][n][k%KPACK] = B_mem_orig[k][n];
```

The following figures illustrate the data re-layout process for a 64x16 int8 B matrix and a 32x16 bfloat16 B matrix (corresponding to the maximum-sized B-tile):

Figure 20-3. Re-layout of 64x16 int8 B Matrix

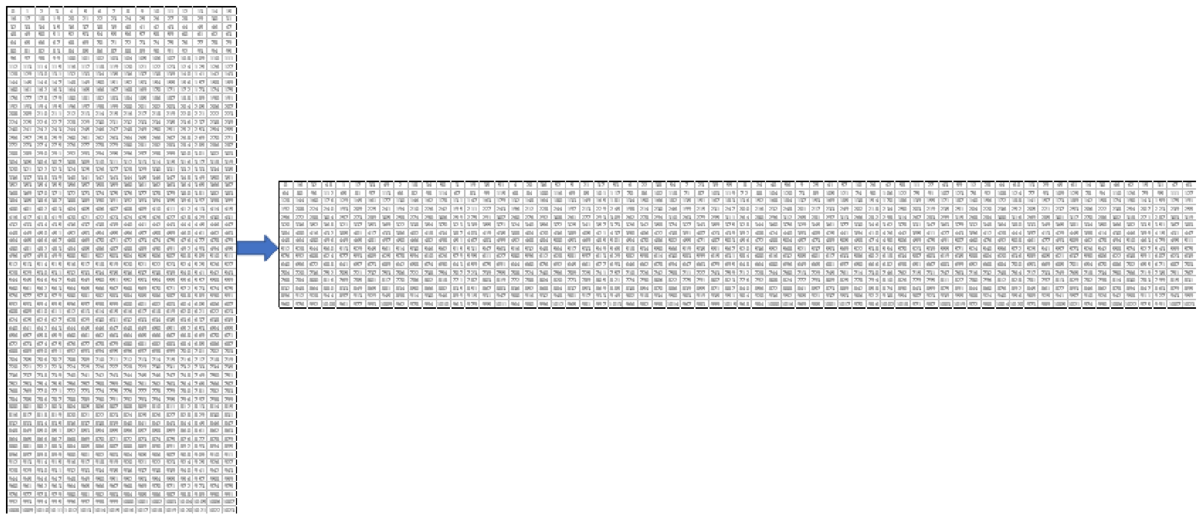
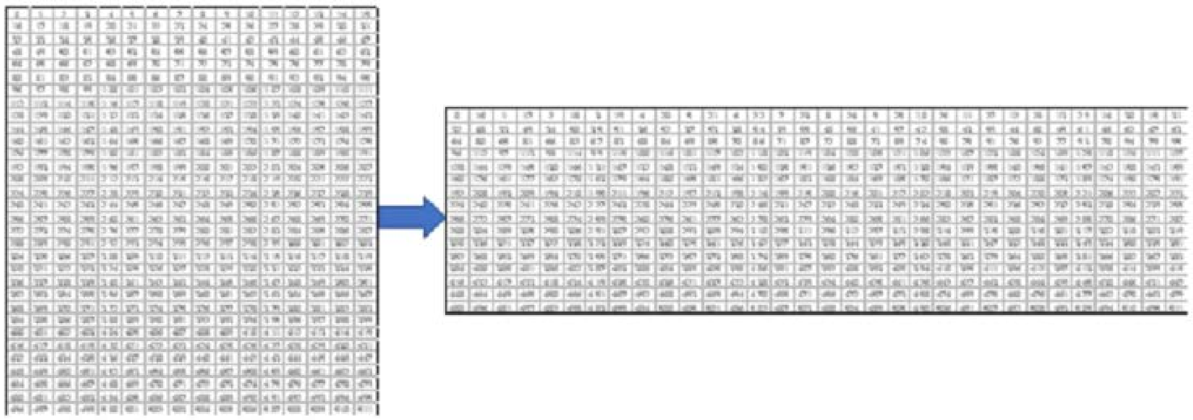


Figure 20-4. Re-layout of 32x16 bfloat16 B Matrix



20.5.4 STRAIGHTFORWARD GEMM IMPLEMENTATION

This is GEMM reference code. Its performance is sub-optimal. Please refer to Section 20.5.5.3 for optimal GEMM code. Begin implementation by defining the following:

Example 20-3. Common Defines

```
#define M ...                           // Number of rows in the A or C matrices
#define K ...                           // Number of columns in the A or rows in the B matrices
#define N ...                           // Number of columns in the B or C matrices
#define M_ACC ...                       // Number of C accumulators spanning the M dimension
#define N_ACC ...                       // Number of C accumulators spanning the N dimension
#define TILE_M ...                      // Number of rows in an A or C tile
#define TILE_K ...                      // Number of columns in an A tile or rows in a B tile
#define TILE_N ...                      // Number of columns in a B or C tile

typedef ... type_t;                     // The type of data being operated on
typedef ... res_type_t;                 // The data type of the result

#define KPACK (4/sizeof(type_t))        // Vertical K packing into Dword

type_t A_mem[M][K];                     // A matrix
type_t B_mem[K/KPACK][N][KPACK];        // B matrix
res_type_t C_mem[M][N];                 // C matrix

template<size_t rows, size_t bytes_cols> class tile;
template<class T> void tilezero (T& t);
template<class T> void tileload (T& t, void* src, size_t stride);
template<class T> void tilestore(T& t, void* dst, size_t stride);
template <class TC, class TA, class TB> void tdp(TC &tC, TA &tA, TB &tB) {
    int32_t v;
    for (size_t m = 0; m < TILE_M; m++) {
        for (size_t k = 0; k < TILE_K / KPACK; k++) {
            for (size_t n = 0; n < TILE_N; n++) {
                memcpy(&v, &tC.v[m][n * 4], sizeof(v));
                v += tA.v[m][k * 4] * tB.v[k][n * 4];
                v += tA.v[m][k * 4 + 1] * tB.v[k][n * 4 + 1];
                v += tA.v[m][k * 4 + 2] * tB.v[k][n * 4 + 2];
                v += tA.v[m][k * 4 + 3] * tB.v[k][n * 4 + 3];
                memcpy(&tC.v[m][n * 4], &v, sizeof(v));
            }
        }
    }
}
```

Data type_t is the type being operated upon, i.e., signed/unsigned int8 or bfloat16. For the description of KPACK, see Section 20.5.5. The tile template class and the three functions that operate on it are the same as the ones introduced in Example 20-3. tilezero (t) resets the contents of tile t to 0, tileload(t, src, stride) and loads tile t with the contents of data at src with a stride of stride between consecutive rows. tilestore(t, dst, stride) stores the contents of tile t to dst with a stride of stride between consecutive rows. Additionally, tdp(tC,tA,tB) performs a matrix multiplication equivalent of tC=tC+tA×tB. In reality, tiles are defined by known compile-time integers, and the actual code operating on tiles looks slightly different. Please visit the GitHub Repository for proper usage.

The following is a simple implementation of GEMM of the matrices stored in A_mem and B_mem.

Example 20-4. Reference GEMM Implementation

```
for (int n = 0; n < N; n += N_ACC*TILE_N) {
  for (int m = 0; m < M; m += M_ACC*TILE_M) {
    tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
      for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
        tilezero(tC[m_acc][n_acc]);

    for (int k = 0; k < K; k += TILE_K) {
      for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
        tile<TILE_K/KPACK, TILE_N*KPACK> tB;
        tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
        for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
          tile<TILE_M, TILE_K*sizeof(type_t)> tA;
          tileload(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
          tdp(tC[m_acc][n_acc], tA, tB);
        }
      }
    }
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
      for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
        int mc = m + m_acc*TILE_M, nc = n + n_acc*TILE_N;
        tilestore(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
      }
    }
  }
}
```

This implementation is the reference point in the following discussions.

20.5.5 OPTIMIZATIONS

20.5.5.1 Minimizing Tile Loads

Redundant tile loads may severely impact performance due to the large size of the data loaded into the tiles, unnecessary cache evictions, etc. To minimize tile loads, it is essential to utilize the data as completely as possible once it has been loaded into the tile.

Location of the K Loop: Outside of the M_ACC and N_ACC Loops

The three loops in lines 8–18 of Example 20-4 could also have been written this way:

Example 20-5. K-Dimension Loop as Innermost Loop–A, a Highly Inefficient Approach

```
for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
  tile<TILE_K/KPACK, TILE_N*KPACK> tB;
  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
    tile<TILE_M, TILE_K*sizeof(type_t)> tA;
    for (int k = 0; k < K; k += TILE_K) {
      tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
      tileload(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
      tdp(tC[m_acc][n_acc], tA, tB);
    }
  }
}
```

While both approaches yield correct results, there are K/TILE_K×N_ACC B tile loads in the reference implementation. Additionally, K/TILE_K×N_ACC×M_ACC B tile loads in the implementation presented in this section. The number of A tile loads is identical.

This approach is also characterized by excessive pressure on the memory along with an increased number of tile loads.

Suppose the B_mem data resides in main memory. In the reference implementation, a new chunk of TILE_K×TILE_N B data is read every M_ACC iteration of the inner loop. The inner loop then reuses the read data. In the current implementation, when n_acc == m_acc == 0, a new chunk of TILE_K×TILE_N B data is read every iteration of the inner loop. Then the same data is read (presumably from caches) on subsequent iterations of n_acc, m_acc. This burst access pattern of reads from main memory results in increased data latency and decreased performance.

Hence, keeping the K-dimension loop outside the M_ACC and N_ACC loops is recommended.

Pre-Loading Innermost Loop Tiles

Consider the following replacement code for the code in lines 8–18 of Example 20-4:

Example 20-6. Innermost Loop Tile Pre-Loading

```
1    for (int k = 0; k < K; k += TILE_K) {
2      tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
3      for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
4        tileload(tA[m_acc], &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
5      for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
6       tile<TILE_K/KPACK, TILE_N*KPACK> tB;
7        tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
8        for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
9          tdp(tC[m_acc][n_acc], tA[m_acc], tB);
10       }
11     }
12   }
```

The A-tile has been extended to an array of A-tiles (line 2) and pre-read the A tiles for the current K-loop iteration (lines 3–4). A pre-read A-tile is used in the tile multiplication (line 9). There were

K/TILE_K×N_ACC×M_ACC A-tile reads in the reference implementation, while there are only K/TILE_K×M_ACC A-tile reads in the current implementation.

Hence, preallocation and pre-reading the tiles of the innermost loop (tA[M_ACC] in this case) is recommended. The maximum number of tiles used at any given time in this scenario is N_ACC×M_ACC+M_ACC+1 as opposed to N_ACC×M_ACC+2 in the reference implementation. Since this optimization requires preallocation of an additional M_ACC-1 tiles, and since tiles are a scarce resource, if N_ACC<M_ACC, it might prove beneficial to switch the order of the N_ACC and M_ACC loops. This way, it is possible to allocate N_ACC-1<M_ACC-1 additional tiles:

Example 20-7. Switched Order of M_ACC and N_ACC Loops

```
for (int k = 0; k < K; k += TILE_K) {
  tile<TILE_K/KPACK, TILE_N*KPACK> tB[N_ACC];
  for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
    tileload(tB[n_acc], B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
    tile<TILE_M, TILE_K*sizeof(type_t)> tA;
    tileload(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
      tdp(tC[m_acc][n_acc], tA, tB[n_acc]);
    }
  }
}
```

2D Accumulator Array vs. 1D Accumulator Array

Consider Example 20-6 with the following scenarios:

- `N_ACC=2,M_ACC=2`
- `N_ACC=4,M_ACC=1`

As stated before, the number of A tile loads in lines 3–11 is M_ACC, and the number of B tile loads is N_ACC. Thus, the total number of tile loads (M_ACC+N_ACC) is 4 in the first scenario vs. 5 in the second one (an increase of 25%), even though both scenarios perform the same amount of work.

Hence, using 2D accumulator arrays is recommended. Selecting dimensions close to square is particularly recommended (since x=y minimizes f(x,y)=x+y under the constraint x×y=const).

20.5.5.2 Software Pipelining of Tile Loads and Stores

It is a best practice to interleave instructions using different resources so they may be executed in parallel, preventing a bottleneck involving a specific resource. Therefore, preventing sequential TileLoads and TileStores (see lines 19–23 of Example 20-4 and lines 3–4 of Example 20-6) is recommended. Instead, interleave them with the tdp instructions (see Example 20-8).

20.5.5.3 Optimized GEMM Implementation

Below is the original code from Example 20-4, augmented with the insights from Example 20-6, with tile loads and stores interleaved with tdps:

Example 20-8. Optimized GEMM Implementation

```
1 for (int n = 0; n < N; n += N_ACC*TILE_N) {
2   for (int m = 0; m < M; m += M_ACC*TILE_M) {
3     tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
4     tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
5     tile<TILE_K/KPACK, TILE_N*KPACK> tB;
6
7     for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
8       for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
9         tilezero(tC[m_acc][n_acc]);
10
11     for (int k = 0; k < K; k += TILE_K) {
12       for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
13         tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
14         for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
15           if (n_acc == 0)
16             tileload(tA[m_acc], &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
17           tdp(tC[m_acc][n_acc], tA[m_acc], tB);
18           if (k == K - TILE_K) {
19             int mc = m + m_acc*TILE_M, nc = n + n_acc*TILE_N;
20             tilestore(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
21           }
22         }
23       }
24     }
25   }
26}
```

While placing the tile loads and stores under conditions inside the main loop (lines 13, 16, 20), conditions can be eliminated by sufficiently unrolling the loops.

The rest of this section presents a specific example of GEMM, implemented in low-level Intel AMX instructions. This is to show a full performance potential from using Intel AMX extensions.

Example 20-9. Dimension of Matrices, Data Types, and Tile Sizes

```
#define M 32
#define K 128
#define N 32
#define M_ACC 2
#define N_ACC 2
#define TILE_M 16
#define TILE_K 64
#define TILE_N 64

typedef int8_t type_t
typedef int32_t res_type_t
```

The following code is a specific example of the algorithm outlined in Example 20-8.

Example 20-10. Optimized GEMM Assembly Language Implementation

```
/*1 of 2*/
typedef struct {
        uint8_t palette_id;
        uint8_t startRow;
        uint8_t reserved[14];
        uint16_t cols[16];
        uint8_t rows[16];
} __attribute__ ((__packed__)) tileconfig_t;

static const tileconfig_t tc = {
        1,                                                      // palette_id
        0,                                                      // startRow
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,               // reserved - must be
        64, 64, 64, 64, 64, 64, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // calls for 7 tiles used
       16, 16, 16, 16, 16, 16, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0    // rows for 7 tiles used
};


_asm {
ldtilecfg tc                            # Load tile config
mov r8, A_mem                           # Initialize register for A
mov r9, B_mem                           # Initialize register for B
mov r10, C_mem                          # Initialize register for C

mov r11, 128                            #  Initialize register for strides
tileloadd tmm6, [r9 + r11*1]            #  Load B for n_acc = 0, k_acc = 0
tileloadd tmm4, [r8 + r11*1]            #  Load A for m_acc = 0, k_acc = 0
tilezero tmm0                           #  Zero accumulator tile
tdpbssd tmm0, tmm4, tmm6                #  Multiply-add tmm0 += tmm4 * tmm6
tileloadd tmm5, [r8 + r11*1 + 2048]     #  Load A for m_acc = 1, k_acc = 0
tilezero tmm1                           #  Zero accumulator tile
/*2 of 2*/
31   tdpbssd tmm1, tmm5, tmm6                  # Multiply-add tmm1 += tmm5 * tmm6
32   tileloadd tmm6, [r9 + r11*1 + 64 ]        # Load B for n_acc = 1, k_acc = 0
33   tilezero tmm2                             # Zero accumulator tile
34   tdpbssd tmm2, tmm4, tmm6                  # Multiply-add tmm2 += tmm4 * tmm6
35   tilezero tmm3                             # Zero accumulator tile
36   tdpbssd tmm3, tmm5, tmm6                  # Multiply-add tmm3 += tmm5 * tmm6
37   tileloadd tmm6, [r9 + r11*1 + 2048]       # Load B for n_acc = 0, k_acc = 1
38   tileloadd tmm4, [r8 + r11*1 + 64]         # Load A for m_acc = 0, k_acc = 1
39   tdpbssd tmm0, tmm4, tmm6                  # Multiply-add tmm0 += tmm4 * tmm6
40   tilestored [r10 + r11*1], tmm0            # Store C for m_acc = 0, n_acc = 0
41   tileloadd tmm5, [r8 + r11*1 + 2112]       # Load A for m_acc = 1, k_acc = 1
42   tdpbssd tmm1, tmm5, tmm6                  # Multiply-add tmm1 += tmm5 * tmm6
43   tilestored [r10 + r11*1 + 2048], tmm1     # Store C for m_acc = 1, n_acc = 0
44   tileloadd tmm6, [r9 + r11*1 + 2112]       # Load B for n_acc = 1, k_acc = 1
45   tdpbssd tmm2, tmm4, tmm6                  # Multiply-add tmm2 += tmm4 * tmm6
46   tilestored [r10 + r11*1 + 64], tmm2       # Store C for m_acc = 0, n_acc = 1
47   tdpbssd tmm3, tmm5, tmm6                  # Multiply-add tmm3 += tmm5 * tmm6
48   tilestored [r10 + r11*1 + 2112], tmm3     # Store C for m_acc = 1, n_acc = 1
49   }
```

Lines 1-12 in Example 20-10 define the tile configuration for this example, and contain information about tile sizes. Tile configuration should be loaded prior to any execution of Intel AMX instructions (line 16). Tile sizes are defined by the configuration at the load time and can't be changed dynamically (unless TileRelease is called). The 'palette_id' field in the configuration specifies the number of logical tiles available for use; palette_id == 1 means 8 logical tiles are available, named tmm0 through tmm7. This particular example uses 7 logical tiles (tmm4, tmm5 for A, tmm6 for B, tmm0-tmm3 for C).

According to the dimensions specified, K-loop consists of 2 iterations (cf. code listing 8.1, line 11) according to the dimensions specified in the example. Lines 23-34 implement the first iteration and lines 35-46 the second iteration. Note the interleaving of tdp and TileStore instructions to hide the high cost of TileStore operation.

Variable Input Dimensions

The code in Example 20-8 and 20-10 process an entire matrix of inputs of size MxK. Sometimes, only part of the input is significant, so it is beneficial to adapt the computation to the actual input size. Often, topologies that use self-attention it is enough to process only the first m rows of the input that are significant, where m < M. For example, taking the GEMM dimensions described above with the choice of a 1D accumulator array of N_ACC=2,M_ACC=1, when accepting data as input with at most sixteen significant rows, we can degenerate the m loop (line 2 in Example 20-8) so as to effectively reduce the computation by half.

It is worth noting that in variable M dimension use cases there is an advantage to 1D accumulators. Up to N_ACC=6, M_ACC=1 dimensions are possible if N is 96 or larger, one tile for A, one tile for B and six tiles for the accumulator.

20.5.5.4 Direct Convolution with Intel® AMX

Direct convolution is performed directly on the input data; no data replication is required. However, there are some layout considerations.

Activations Layout

Similar to the Intel DL Boost use case, the activations are laid out in a layout obtained from the original layout by the following procedure:

Example 20-11. Activations Layout Procedure

```
#define K C                              // K-dimension of the A matrix = channels
#define M H*W                            // M-dimension of the A matrix = spatial
type_t A_mem_orig[C][H][W];              // Original activations tensor
type_t A_mem[H][W][K];                   // Re-laid A matrix7

for (int c = 0; c < C; ++c)
  for (int h = 0; h < H; ++h)
    for (int w = 0; w < W; ++w)
      A_mem[h][w][c] = A_mem_orig[c][h][w];
```

This procedure on the left side of the diagram below shows the conversion of a 3-dimensional tensor into a 2-dimensional matrix:

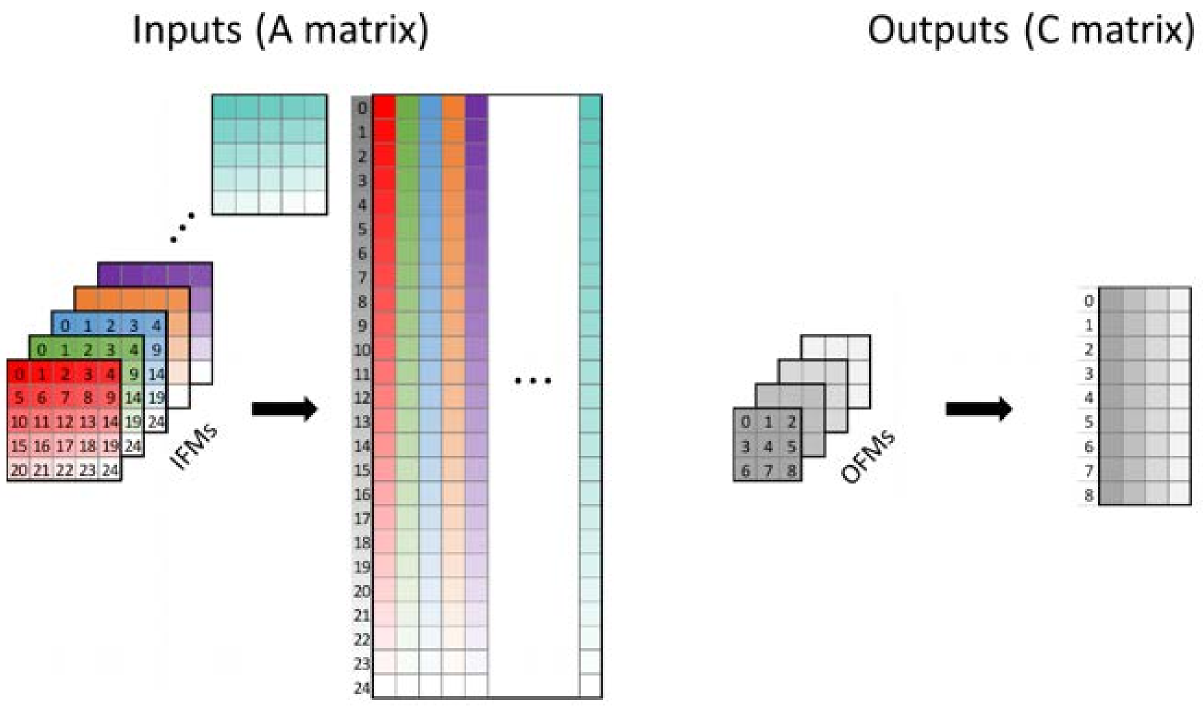


Figure 20-5. Activations layout

The procedure shown on the right is identical for the outputs, e.g., the activations of the next layer in the topology).

Weights Layout

Similar to the Intel DL Boost use case, the weights are re-laid by the following procedure:

Example 20-12. Weights Re-Layout Procedure

```
#define KH ...                              // Vertical dimension of the weights
#define KW ...                              // Horizontal dimension of the weights
#define KPACK (4/sizeof(type_t))            // Vertical K packing into Dword

type_t B_mem_orig[K][N][KH][KW];            // Original weights
type_t B_mem[KH][KW][K/KPACK][N][KPACK];    // Re-laid B matrices

for (int kh = 0; kh < KH; ++kh)
 for (int kw = 0; kw < KW; ++kw)
  for (int k = 0; k < K; ++k)
   for (int n = 0; n < N; ++n)
    B_mem[kh][kw][k/KPACK][n][k%KPACK] = B_mem_orig[k][n][kh][kw];
```

The procedure transforms the original 4-dimensional tensor into a series of 2-dimensional matrices (a single matrix is highlighted in orange in Example 20-12) as illustrated in the following diagram for `KH=KW=3`, resulting in a series of 9 B-matrices:

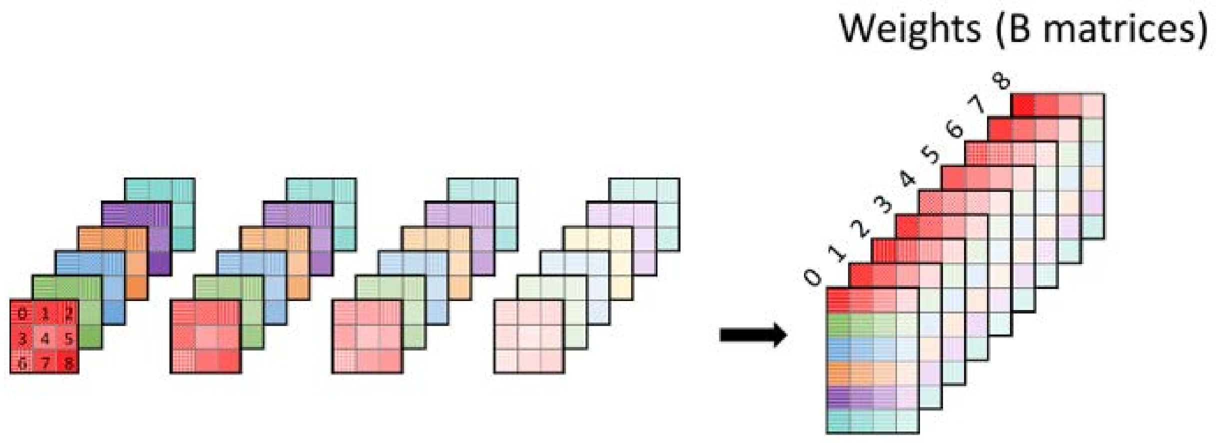


Figure 20-6. Weights Re-Layout

20.5.5.5 Convolution - Matrix-like Multiplications and Summations Equivalence

Figure 20-7 illustrates the equivalence between convolution and summation of a series of matrix-like multiplications between subsets of the 2-dimensional A-matrix representing the 3-dimensional activations tensor. The 2-dimensional B-matrices correspond to the various spatial elements of the weights filter.

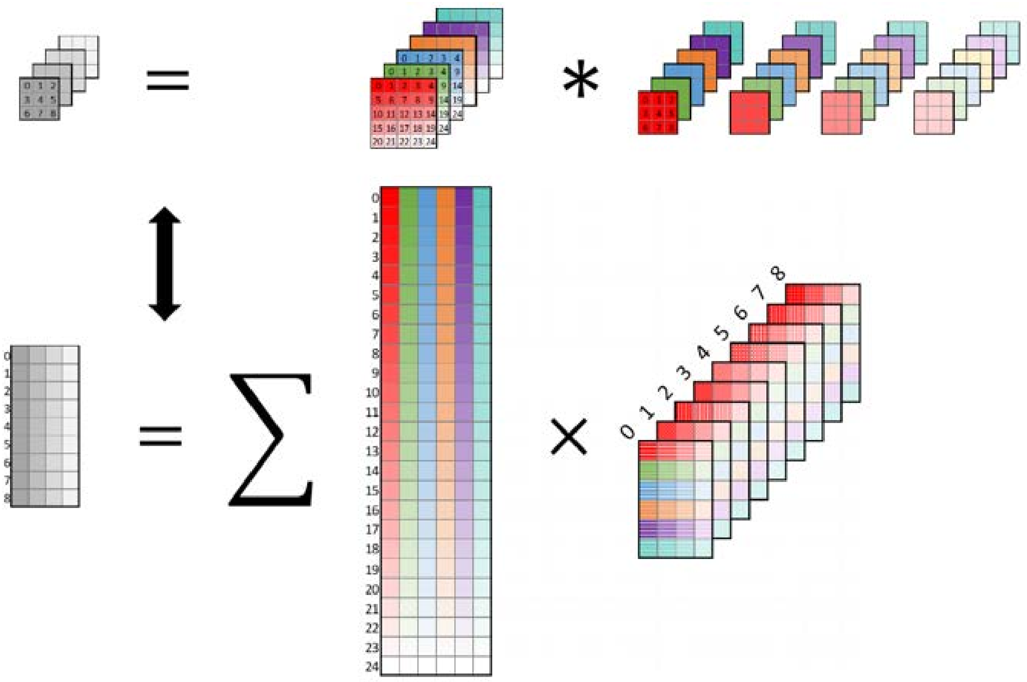


Figure 20-7. Convolution-Matrix Multiplication and Summation Equivalence

The A-matrix subset participating in the matrix-like multiplication depends on the spatial weight element in question (i.e., the kh,kw coordinates, or the index in the range 0–8 in the previous example). For each weight element, the A-matrix's participating rows will interact with the weight element when the filter is slid over the activations. For example, when sliding the filter over the activations in the previous example, weight element 0 will only interact with activation elements 0, 1, 2, 5, 6, 7, 10, 11, and 12. For example, it will not interact with activation element four because when the filter is applied in such a manner (i.e., weight element 0 interacts with activation element 4), weight elements 2, 5, and 8 leave the activation frame entirely. The A-matrix subsets for several weight elements are illustrated in the following figure.

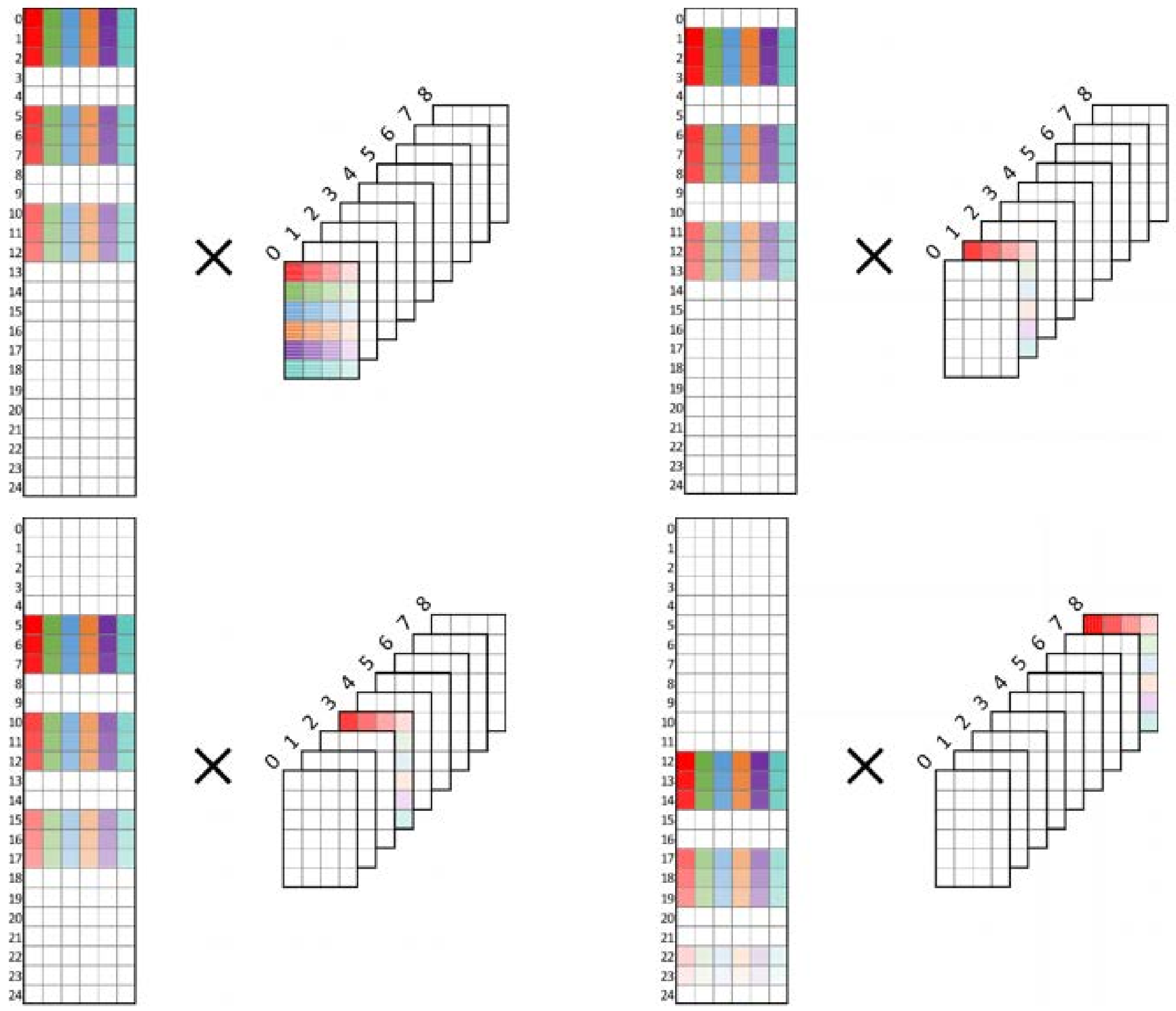


Figure 20-8. Matrix-Like Multiplications Part of a Convolution

20.5.5.6 Optimized Convolution Implementation

Replace the common defines in Example 20-3 with the following:

Example 20-13. Common Defines for Convolution

```
#define H ...                   // The height of the activation frame
#define W ...                   // The width of the activation frame
#define MA (H*W)                // The M dimension (rows) of the A matrix
#define K ...                   // Number of activation channels
#define N ...                   // Number of output channels
#define KH ...                  // The height of the weights kernel
#define KW ...                  // The width of the weights kernel
#define SH ...                  // The vertical stride of the convolution
#define SW ...                  // The horizontal stride of the convolution
#define M_ACC ...               // Number of C accumulators spanning the M dimension
#define N_ACC ...               // Number of C accumulators spanning the N dimension
#define TILE_M ...              // Number of rows in an A or C tile
#define TILE_K ...              // Number of columns in an A tile or rows in a B tile
#define TILE_N ...              // Number of columns in a B or C tile

#define HC ((H-KH)/SH+1)        // The height of the output frame
#define WC ((W-KW)/SW+1)        // The width of the output frame
#define MC (HC*WC)              // The M dimension (rows) of the C matrix

typedef ... type_t;             // The type of the data being operated on
typedef... res_type_t;          // The data type of the result

#define KPACK (4/sizeof(type_t))                // Vertical K packing into Dword

type_t A_mem[H][W][K];                          // A matrix (equivalent to A_mem[H*W][K])
type_t B_mem[KH][KW][K/KPACK][N][KPACK];        // B matrices
res_type_t C_mem[MC][N];                        // C matrix

template<size_t rows, size_t cols> class tile;

template<class T> void tilezero (T& t);
template<class T> void tileload (T& t, void* src, size_t stride);
template<class T> void tilestore(T& t, void* dst, size_t stride);
template<class TC, class TA, class TB> void tdp(TC& tC, TA& tA, TB& tB);

int mc_to_ha(int mc) {return mc / HC * SH;}     // C matrix M -> A tensor h coord
int mc_to_wa(int mc) {return mc % HC * SW;}     // C matrix M -> A tensor w coord
```

Replace the implementation in Example 20-8 with the following:

Example 20-14. Optimized Direct Convolution Implementation

```
1 for (int n = 0; n < N; n += N_ACC*TILE_N) {
2  for (int m = 0; m < MC; m += M_ACC*TILE_M) {
3   tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
4   tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
5   tile<TILE_K/KPACK, TILE_N*KPACK> tB;
6
7   for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
8    for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
9     tilezero(tC[m_acc][n_acc]);
10
11   for (int k = 0; k < K; k += TILE_K) {
12    for (int kh = 0; kh < KH; ++kh) {
13     for (int kw = 0; kw < KW; ++kw) {
14      for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
15       int nc = n + n_acc*TILE_N;
16       tileload(tB, B_mem[kh][kw][k/KPACK][nc], N*sizeof(type_t)*KPACK);
17       for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
18        int mc = m + m_acc*TILE_M;
19        if (n_acc == 0) {
20         int ha = mc_to_ha(mc)+kh, wa = mc_to_wa(mc)+kw;
21         tileload(tA[m_acc], &A_mem[ha][wa][k], K*SW*sizeof(type_t));
22        }
23        tdp(tC[m_acc][n_acc], tA[m_acc], tB);
24        if (k + kh + kw == K - TILE_K + KH + KW - 2)
25         tilestore(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
26       }
27      }
28     }
29    }
30   }
31  }
32 }
```

The divergences highlighted in yellow in Example 20-8 include:

- The loop over M-dimension (line 2) references the M-dimension of the C-matrix (since the M-dimensions of A and C no longer have to be the same). To get the corresponding A-matrix m index from a C-matrix m index, one must employ the conversion functions mc_to_ha() and mc_to_wa() (line 20).
- There are additional loops over the weights kernel dimensions KH and KW (lines 12–13), which define the B-matrix to be used (line 16), enter into the condition for accumulator tile storing (line 24) and computation of A-matrix coordinates (line 20).
- The stride of the A tile load must account for the convolutional horizontal stride (line 21).

Note that care should be taken to define TILE_M*M_ACC in such a way that it cleanly divides WC (the width of the output frame), i.e., WC%(TILE_M*M_ACC)==0. Otherwise, some tiles will end up loading data that should not be multiplied by the corresponding weight element (see Figure 20-8). Possible mitigations of this issue:

- An M_ACC loop with a dynamic upper limit depending on the current position in A.

- Use different sized A tiles (and correspondingly C tiles) depending on the current position in A (if there are enough free tiles, performing TileConfig during the convolution is highly discouraged).
- Define TILE_M without consideration for WC and remove/disregard the "junk" data from the results at the post-processing stage (code not shown). Care should be taken in this case concerning the advancement of the m index (line 2) since the current assumption is that every row of every tile is valid (corresponds to a row in the C matrix). If "junk" data is loaded, this is no longer the case: a C-tile will have less than TILE_M rows of C.

Location of the KH, KW Loops

As shown in Example 20-5, it is ill-advised to put the loop over the K-dimension inside an inner M_ACC or N_ACC loop. The same considerations hold in the case of the kh,kw loops. While there is no functional obstacle precluding the positioning of the kh,kw loops further up (before lines 12-13), it is recommended to keep them under the K loop and above the M_ACC, N_ACC loops because, during the traversal of kh,kw with the same k value, the TileLoad of A-data (line 21) will have much overlap with A-data loaded for previous values of kh,kw (with the same k value). This data will likely reside in the lowest-level cache. Moving the kh,kw loops upward will reduce that likelihood.

20.6 CACHE BLOCKING

Data movement costs vary greatly depending on where the data lies in the cache hierarchy. When the matrices involved in a GEMM or convolution are larger than the available cache, computations must proceed in such a manner as to optimize data reuse from the cache. Here a simple cache-blocking scheme is implemented to simultaneously process partial blocks of the A, B, and C matrices.

20.6.1 OPTIMIZED CONVOLUTION IMPLEMENTATION WITH CACHE BLOCKING

In the following example, the focus is on implementing cache blocking for the optimized convolution implementation described in the Optimized Convolution Implementation <XREF> section. However, note that similar changes can also be made to the optimized GEMM implementation. Alternatively, the GEMM implementation can be derived as a special case of convolution with KH=KW=1 and SH=SW=1.

In addition to the common defines in Example 20-13, add the following:

Example 20-15. Additional Defines for Convolution with Cache Blocking

```
#define MC_CACHE ...                          // Extent of cache block along the M dimension of the C matrix
#define K_CACHE ...                           // Extent of cache block along the K dimension
#define N_CACHE ...                           // Extent of cache block along the N dimension
typedef ... acc_type_t;                       // The accumulation data type (either int32 or float)
acc_type_t aC_mem[M_ACC][N_ACC][TILE_M][TILE_N];          // Accumulator buffers of C
```

Replace the implementation in Example 20-14 with the following:

Example 20-16. Optimized Convolution Implementation with Cache Blocking

```
for (int nb = 0; nb < N; nb += N_CACHE) {
  for (int mb = 0; mb < MC; mb += MC_CACHE) {
    for (int kb = 0; kb < K; kb += K_CACHE) {
      for (int n = nb; n < nb + N_CACHE; n += N_ACC*TILE_N) {
        for (int m = mb; m < mb + MC_CACHE; m += M_ACC*TILE_M) {
          tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
          tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
          tile<TILE_K/KPACK, TILE_N*KPACK> tB;

          for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
            for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
              if (kb == 0)
                tilezero(tC[m_acc][n_acc]);
              else {
                int m_aC = (m - mb) / TILE_M + m_acc;
                int n_aC = (n - nb) / TILE_N + n_acc;
                tileload(tC[m_acc][n_acc], &aC_mem[m_aC][n_aC],
                       TILE_N*sizeof(acc_type_t));
              }

          for (int k = kb; k < kb + K_CACHE; k += TILE_K) {
            for (int kh = 0; kh < KH; ++kh) {
              for (int kw = 0; kw < KW; ++kw) {
                for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
                  int nc = n + n_acc*TILE_N;
                  tileload(tB, B_mem[kh][kw][k/KPACK][nc], N*sizeof(type_t)*KPACK);
                  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
                    int mc = m + m_acc*TILE_M;
                    if (n_acc == 0) {
                      int ha = mc_to_ha(mc)+kh, wa = mc_to_wa(mc)+kw;
                      tileload(tA[m_acc], &A_mem[ha][wa][k], K*SW*sizeof(type_t));
                    }
                    tdp(tC[m_acc][n_acc], tA[m_acc], tB);
                    if (k + kh + kw == K - TILE_K + KH + KW - 2)
                      tilestore(tC[m_acc][n_acc], &C_mem[mc][nc],
                               N*sizeof(res_type_t));
                    else if (k + kh + kw == kb + K_CACHE - TILE_K + KH + KW - 2) {
                      int m_aC = (m - mb) / TILE_M + m_acc;
                      int n_aC = (n - nb) / TILE_N + n_acc;
                      tilestore(tC[m_acc][n_acc], &aC_mem[m_aC][n_aC],
                               TILE_N*sizeof(acc_type_t));
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The loops over the N, MC, and K dimensions are replaced by loops over cache blocks of N, MC, and K.

Additional loops over the entire N, MC, and K-dimensions are added at the outermost level. These loops have a step size equal to the cache blocks of N, MC, and K.

In the case of cache blocking along the K-dimension, additional calls to `TileLoad` and TileStore are required to load and store intermediate accumulation results. Note that this adds additional memory traffic, especially for int8 output data types (as Accumulation data type is either int32_t or float). For this reason, it is generally not advisable to block along the K dimension.

For simplicity, assume the following relationships:

- N is an integer multiple of N_CACHE: an integer multiple of N_ACC*TILE_N.
- MC is an integer multiple of MC_CACHE: an integer multiple of M_ACC*TILE_M. As before, the condition WC%(TILE_M*M_ACC)==0 still holds.
- K is an integer multiple of K_CACHE: an integer multiple of TILE_K.

Define the following set of operations as the compute kernel of the optimized convolution implementation. First, initialize the accumulation tiles to zero (line 13) for an M_ACC*TILE_M x N_ACC*TILE_N chunk of the C-matrix. Next, for each of the KH*KW B-matrices, the matrix multiplication of the corresponding M_ACC*TILE_M x K chunk of the A-matrix by a K x N_ACC*TILE_N chunk of the B-matrix is performed, each time accumulating to the same set of accumulation tiles (lines 18–30). Finally, the results are stored in the C-matrix (line 32).

Continue with the computation **of a full cache block** of C-matrix, ignoring any blocking along the K-dimension. First, the kernel is performed for the first chunks of the A, B, and C cache blocks. Next, the chunks of A and C advance along the M dimension, and the kernel is repeated with the same chunk set of the B-matrices. The above step is repeated until the last chunks of A and C in the current cache block have been accessed. Next, the chunks of B and C are advanced along the N-dimension by N_ACC*TILE_N and the chunk of A returns to the beginning of its cache block.

Observe the following from the above description of the computation of a **full cach**e block of the C-matrix:

- For each kernel iteration, it is better if the current chunk of matrix A (roughly KH*M_ACC*TILE_M*K*sizeof(type_t)) fits into the DCU. This allows for maximal data reuse between the partially overlapping regions of A that need to be accessed by the different B matrices.
- Advancing from one chunk of matrix A to the next, it is better if the current chunk set of the B matrices (in total, KH*KW*K*N_ACC*TILE_N*sizeof(type_t)) fits into the DCU.
- Advancing from one chunk set of the B matrices to the next, it is better if the current cache block of matrix A fits into the MLC.
- Advancing from one cache block of matrix A to the next, it is better if the current cache block of the B matrices (in total, KH*KW*K*N_CACHE*sizeof(type_t)) fits into the MLC.

From these observations, a general cache blocking strategy is choosing `MC_CACHE` and `N_CACHE` to be as large as possible while keeping the A, B, and C cache blocks in the MLC.

Intel® AMX-Specific Considerations

A specific feature of Intel AMX-accelerated kernels to keep in mind when applying the previous cache-blocking recommendations is any post-processing of results from the Intel AMX unit (e.g., adding bias, dequantizing, converting between data types) must occur by way of vector registers. Thus, a buffer is needed to store results from the accumulation tiles and load them into vector registers for post-processing. Note that if acc_type_t is the same as res_type_t, the C matrix itself can be used to store intermediate results. However, the buffer is small (at most 4KB for the accumulation strategies described in vSection) and easily fits into the DCU. While it should still be considered when determining the optimal cache block partitioning, it is unlikely to influence kernel performance strongly.

20.7 MINI-BATCHING IN LARGE BATCH INFERENCE

Layers have different sizes and shapes, which require different cache and memory-blocking strategies. There are layers with a small spatial dimension (M) and relatively larger shared dimension (K) and SIMD dimension (N). In such layers, the weights are significantly larger than the inputs. Therefore, most of the load operations are weights matrix loads whose cost is high when the weights reside in memory or the last level cache.

Running a large batch allows employing an optimization that amortizes the cost of loading the weight matrix. The idea is to use the same weights for multiple inputs, e.g., execute the same layer with multiple images. This optimization is highly applicable in CNNs where the inputs of the first layers are large while the weights are relatively small but end with small input images and large weight matrices. Optimal execution of the topology starts in the instance or image affinity, where a single input goes through one layer after another before the next input is retrieved. At some point, the topology execution switches to layer affinity, where the same layer processes several inputs (mini-batch) before moving forward to the next layer.

For example, in ResNet-50, the conv-1 to conv-4 layers have relatively large IFMs and smaller weight matrices. However, many weight matrices are larger than MLC size (mid-level cache) in the conv-5 layers. The switchover point from image affinity to layer affinity on a 4th Generation Intel® Xeon® Processor microarchitecture is the first layer of conv-5.

The diagram below illustrates six layers with four instances per thread (mini-batch of four). Boxes with identical colors identify the same layers in each column. Arrows flowing downward through each column's layers represent the data flow of a particular instance. Translucent red arrows identify the execution order of layers with corresponding instances. The first four layers of the diagram have instance (aka image) affinity, and the last two have layer affinity.

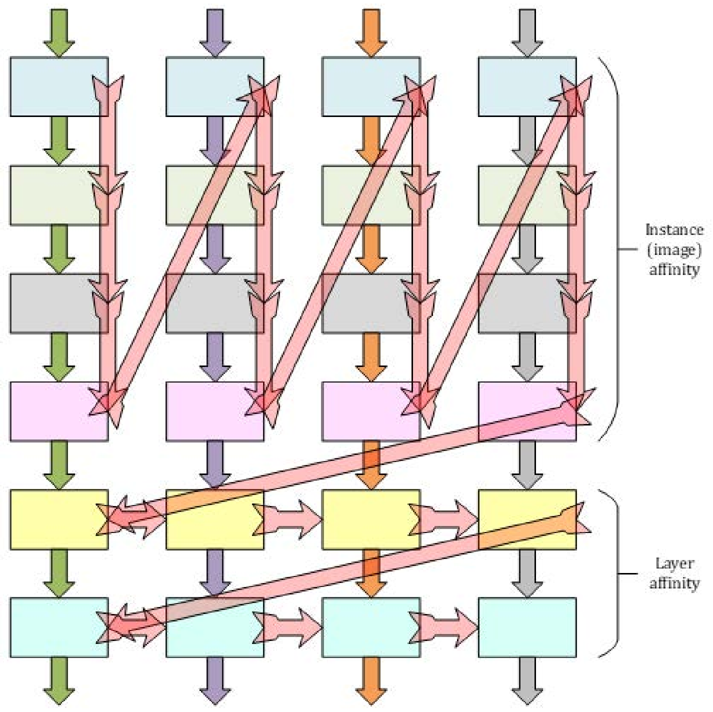


Figure 20-9. Batching Execution Using Six Layers with Four Instances Per Thread

On Resnet-50, this optimization can yield a 17% performance gain.

20.8 NON-TEMPORAL TILE LOADS

When a regular tile load is issued, the data for the tile are placed in L2, L1, and then in the tile register (DRAM/L3->L2->L1->tile register), as with any other register load. This has the well-known benefit of reduced data read latency due to data proximity when recently accessed data are reaccessed after a short time. However, indiscriminate application of this approach might sometimes prove detrimental.

Consider the code in Example 20-4, referring to the unoptimized, unblocked implementation for simplicity. The five loops in the code listing alongside the total input (A) matrix data and weights (B) matrix data accessed at each loop level is shown in the following table. The original row in the code listing is provided for convenience:

Table 20-3. Five Loops in Example 20-4

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	M×K	K×N
2	m	[0:M:M_ACC×TILE_M]	M×K	K×N_ACC×TILE_N
8	k	[0:K:TILE_K]	MC_CACHE×K	K×N_ACC×TILE_N
9	n acc	[0:N_ACC:1]	M_ACC×TILE_M×TILE_K	TILE_K×N_ACC×TILE_N
12	m ac	[0:M_ACC:1]	M_ACC×TILE_M×TILE_K	TILE_K×N_ACC×TILE_N

20.8.1 PRIORITY INVERSION SCENARIOS WITH TEMPORAL LOADS

For the following discussion, assume:

- The data type is int8 (i.e., each element in the table above takes 1 byte).
- TILE_M=16, TILE_K=64, TILE_N=16 (i.e., all tiles are of size 1kB).
- L1 cache size is 32kB.
- M_AC=N_ACC=2.

Scenario One:

Consider the following scenario, including M=256, K=1024, and N=256.

Table 20-4 illustrates accessed data sizes:

Table 20-4. Accessed Data Sizes: Scenario One

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	256kB	256kB
2	m	[0:M:M_ACC×TILE_M]	256kB	32kB
8	k	[0:K:TILE_K]	32kB	32kB
9	n acc	[0:N_ACC:1]	32kB	2kB
12	m ac	[0:M_ACC:1]	32kB	2kB

At the k loop level, the combined sizes of A and B accessed data will overflow the L1 cache by a factor of two. Proceeding to the m-level since m is progressing, new A-data are constantly read (a total of 256kB-32kB=224kB new A data), while the same 32kB of B data are being accessed repeatedly. Thus, a priority inversion occurs: new A-data placed in the L1 cache repeatedly are accessed only once. They evict the 32kB of B data that are accessed eight times. Placement of A data in the L1 cache is not beneficial: the next time the same data are accessed will be in the n loop after 256kB (x8 L1 cache size) of A data has been read. Additionally, it is detrimental because it causes repeated eviction of 32kB of B data that could have been read from the L1 cache eight times.

Scenario Two:

Consider the following scenario, including M=32, K=1024, and N=256. Here, the M-dimension is covered in the m_acc loop, and the loop over m is redundant. The priority inversion is: as n advances, new B-data (accessed only once) repeatedly evict 32kB of A-data that could have been read (8 times) from the L1 cache had it not been pushed out by B-data.

Here, the M-dimension is covered in the m_acc loop, and the loop over m is redundant. The priority inversion is: as n advances, new B-data (accessed only once) repeatedly evict 32kB of A-data that could have been read (8 times) from the L1 cache had it not been pushed out by B-data.

Table 20-5. Accessed Data Sizes: Scenario Two

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	32kB	256kB
2	m	[0:M:M_ACC×TILE_M]		32kB
8	k	[0:K:TILE_K]		
9	n acc	[0:N_ACC:1]	2kB	2kB
12	m ac	[0:M_ACC:1]		

These two basic scenarios can be readily extended to the blocked code in Example 20-16.

Table 20-6. Accessed Data Sizes Extended to Blocked Code

Row	Var	Variable Range	A Data Size	B Data Size
1	nb	[0:N:N_CACHE]	M×K	
2	mb	[0:MC:MC_CACHE]	M×K	
3	kb	[0:K:K_CACHE]	MC_CACHE×K	
4	n	[nb:nb+N_CACHE:N_ACC×TILE_N]	MC_CACHE×K_CACHE	K_CACHE×KH×KW×N_ACC×TILE_N
5	m	[mb:mb+MC_CACHE:M_ACC×TILE_M]		
18	k	[kb:kb+K_CACHE:TILE_K]		
19	kh	[0:KH:1]	/*/*	TILE_K×KH×KW×N_ACC×TILE_N
20	kw	[0:KW:1]	M_ACC×TILE_M×TILE_K	
21	n acc	[0:N_ACC:1]		TILE_K×N_ACC×TILE_N
24	m ac	[0:M_ACC:1]		

NOTE

Due to the nature of convolution, the loops over kh, kw reuse most of the A-data.

The innermost loops m_acc, n_acc, kh,kw will access at most M_ACC kB of A data and KH×KW×N_ACC kB of B-data, which, in some cases (e.g., KH=KW=3, N_ACC=4) might already overflow the L1 cache size. Thus, several opportunities for priority inversions exist in this more complex loop structure, depending on the parameters in the table above:

- B-data evicting reusable A-data at the kh,kw loops level.
- A-data evicting reusable B-data at the m loop level.
- B-data evicting reusable A-data at the n loop level.
- A-data evicting reusable B-data at the mb loop level.
- B-data evicting reusable A-data at the nb loop level.

Solution to Priority Inversions: Non-Temporal Loads

Intel AMX architecture introduces a way to load tile registers bypassing the L1 cache via non-temporal tile loads (TILELOADDT1). This allows the user to deal with priority inversions such as those described above by loading the large, non-reusable data chunk with non-temporal loads. Thus, the larger chunk is prevented from evicting the smaller, frequently used data chunk. In Table 20-4, the A-tiles are loaded with non-temporal loads while loading B-tiles with temporal loads. This ensures the B-tile loads at the m loop level will all come from the L1 cache. In Table 20-5, the B-tiles are loaded with non-temporal loads while loading A-tiles with temporal loads, thus ensuring that the A-tile loads at the n loop level will all come from the SL1 cache.

20.9 USING LARGE TILES IN SMALL CONVOLUTIONS TO MAXIMIZE DATA REUSE

A convolution with a small-sized input frame can make the Intel AMX computation inefficient.

Consider the following example: a 7x7 input frame, with padding of 1 (size including padding is 9x9), convolved with a 3x3 filter to produce a 7x7 output frame.

Figure 20-10 shows the pieces participating in the convolution (in yellow) interacting with the khaki=0,0 weight element.

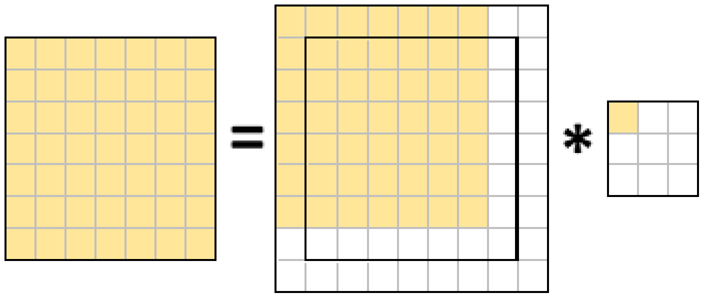


Figure 20-10. A Convolution Example

Thus, the yellow parts of the input frame are the only ones that should be loaded into A-tiles when processing weight element kh,kw=0,0. The white parts of the input frame should be ignored. This requires the number of tile rows to be set at seven, utilizing less than half of the A-tile, reducing B (weights) data reuse by a factor of two. Each A-tile is now half the size, and seven tiles are required to cover the spatial dimension. Because there are not seven tiles, B-tiles must be loaded twice as many times, potentially leading to significant performance degradation, depending on the size of the weights. This is usually inversely proportional to the spatial size of the input frame).

Figure 20-11 shows three A-tiles with sixteen rows and one tile with seven rows to cover the entire spatial dimension of the convolution.

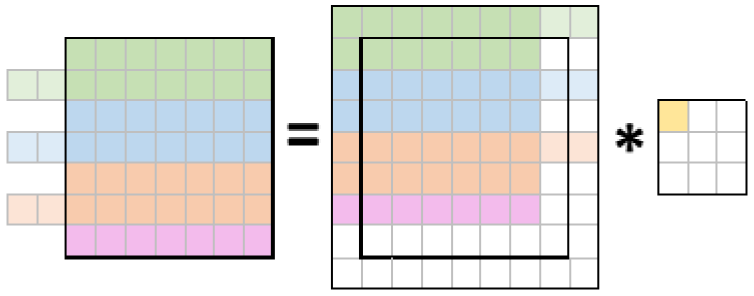


Figure 20-11. A Convolution Example with Large Tiles.

Each tile is highlighted differently. The green, blue, and orange tiles now load those two "extra" pieces previously ignored. Those pieces will waste compute resources and take up two rows in the accumulator tiles. The user may choose to ignore those rows in subsequent computations (e.g., int8-quantization, RELU, etc.), complicating the implementation. The potential benefit of increased B-data reuse could be dramatic, however.

20.10 HANDLING INCONVENIENTLY-SIZED ACTIVATIONS

Occasionally, the spatial dimensions of an activation might be ill-suited for efficient tiling with tiles. Consider a GEMM with activations' M=100. This poses a challenge: while the M dimension can be neatly tiled by ten tiles, each with ten rows, this approach is inefficient since a larger M dimension of 112 requires only seven tiles with sixteen rows. This means that the data reuse for M=100 is 30% worse than for M=112.

The following solutions will be useful:

1. Define two types of A- and C-tiles – tiles with 16 rows and one tile with four. Use tiles of the first type for M=0..9 and the second type tile for M=96..99.
2. Allocate extra space in A and C buffers, as if M=112, and use tiles with 16 rows exclusively. The extra space need not be zeroed out or otherwise prepared in any way. In this case, the last (seventh) tile will load four meaningful rows (M=96..99) and twelve "garbage" rows (M=100..111). At the output, tile C will have four meaningful rows (M=96..99) and twelve "garbage" rows (M=100..111) which the user can then ignore.

The first solution does not require tampering with the A and C buffers and computes 100 tile rows, producing a clean result. Still, it requires additional A- and C-tiles. unused throughout the computation except at the very end. Since only eight tiles are available, this requirement can be costly and might **reduce** the data reuse (e.g., to use a 2D accumulator array, you would need three x2 C-tiles, two A-tiles, and two B-tiles, equaling ten tiles). The second solution avoids this requirement by complicating buffer handling and paying with additional loads, compute, and storing (it loads, computes, and stores 112 tile rows).

20.11 POST-CONVOLUTION OPTIMIZATIONS

Most Intel AMX-friendly applications are from the Deep Learning domain, where the data flows through multiple layers. It is often necessary to process the convolution output before passing it as an input to the next layer (processing operations depend on a specific application). This stage is called **post-convolution**.

20.11.1 POST-CONVOLUTION FUSION

As with Intel AVX-512 code, a critical optimization is the "fusion" of post-convolutional operations to the convolutional data they operate upon. Fusion reduces the memory hierarchy thrashing. Additionally, fusing the quantization step gains x2 (for bfloat16 data type) or x4 (for int8 data type) compute bandwidth, and reduces memory bandwidth by x2 or x4, respectively.

Consider the code in Example 20-8. Lines 7-24 contain the entire GEMM operation for any M, N coordinate in the output. Thus, the optimal location to post-process the data computed in lines 7-24 is right before line 24 while it is still in the low-level cache.

In Example 20-17, blue code illustrates a fully unrolled example from line 7 through 24, for int8 GEMM with K=192, N_ACC=M_ACC=2, TILE_M=2, TILE_K=64, TILE_N=16. The convolution code is fused with post-convolution code (blue) that quantizes the output and ReLU. To keep the post-convolution code in the example short, an unrealistically low value of TILE_M=2 was chosen.

In that example, an additional buffer, temporary_C, contains the convolutional results of M_ACCxN_ACC tiles. The results are stored at the end of the convolutional part and loaded during the post-convolutional part. A temporary buffer is required because the size of the post-processed data is four times smaller. Hence, the convolutional output cannot be written directly to the output buffer.

The GPRs r8, r9, r10, r11, and r14 point to the current location in the A, B, C, temporary_C, and q_bias (which holds the quantization factors and biases) buffers, respectively.

The macros A_OFFSET(m,k), B_OFFSET(k,n), C_OFFSET(m,n), C_TMP_OFFSET(m,n), Q_OFFSET(n), and BIAS_OFFSET(n) receive as arguments m,k,n tile indices and return the offset of the data from r8,r9,r10, r11, and r14, respectively.

Example 20-17. Convolution Code Fused with Post-Convolution Code

```
/*1 of 2*/
#define TILE_N_B                 (N)
#define A_OFFSET(m,k)            ((m)*K*TILE_M + (k)*TILE_K)
#define B_OFFSET(k,n)            ((k)*N*TILE_N*4 + (n)*TILE_N*4)
#define C_OFFSET(m,n)            ((m)*N*TILE_M + (n)*TILE_N)
#define C_TMP_OFFSET(m,n)        ((m)*N*TILE_M*4 + (n)*TILE_N*4)
#define Q_OFFSET(n)              ((n)*TILE_N*4)
#define BIAS_OFFSET(n)           ((n)*TILE_N*4 + N*4)

static const tileconfig_t tc = {
  1,                                                     // Palette ID
  0,                                                     // Start row
     0,  0,  0,  0,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0,    // Reserved – must be 0
  64, 64, 64, 64, 64, 64, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0, // Cols for 7 tiles used
  2, 2, 2, 2, 2, 16, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0       // Rows for tiles used: 2 for A, C,
                                                         // 16 for B
};

ldtilecfg tc                                             // Load tile config
mov          r12, 192                                    // A stride
mov          r13, 128                                    // B, C_TMP stride
tileloadd    tmm5, [r9 + r13*1 + B_OFFSET(0,0)]          // Load B [k,n] = [0,0]
tileloadd    tmm4, [r8 + r12*1 + A_OFFSET(0,0)]          // Load A [m,k] = [0,0]
tilezero     tmm0                                        // Zero acc [m,n] = [0,0]
tdpbusd      tmm0, tmm4, tmm5
tileloadd    tmm6, [r9 + r13*1 + B_OFFSET(0,1)]          // Load B [k,n] = [0,1]
tilezero     tmm2                                        // Zero acc [m,n] = [0,1]
tdpbusd      tmm2, tmm4, tmm6
tileloadd    tmm4, [r8 + r12*1 + A_OFFSET(1,0)]          // Load A [m,k] = [1,0]
tilezero     tmm1                                        // Zero acc [m,n] = [1,0]
tdpbusd      tmm1, tmm4, tmm5
tilezero     tmm3                                        // Zero acc [m,n] = [1,1]
tdpbusd      tmm3, tmm4, tmm6
tileloadd    tmm5, [r9 + r13*1 + B_OFFSET(1,0)]          // Load B [k,n] = [1,0]
tileloadd    tmm4, [r8 + r12*1 + A_OFFSET(0,1)]          // Load A [m,k] = [0,1]
tdpbusd      tmm0, tmm4, tmm5
tileloadd    tmm6, [r9 + r13*1 + B_OFFSET(1,1)]          // Load B [k,n] = [1,1]
tdpbusd      tmm2, tmm4, tmm6
tileloadd    tmm4, [r8 + r12*1 + A_OFFSET(1,1)]          // Load A [m,k] = [1,1]
tdpbusd      tmm1, tmm4, tmm5
tdpbusd      tmm3, tmm4, tmm6
tileloadd    tmm5, [r9 + r13*1 + B_OFFSET(2,0)]          // Load B [k,n] = [2,0]
tileloadd    tmm4, [r8 + r12*1 + A_OFFSET(0,2)]          // Load A [m,k] = [0,2]
tdpbusd      tmm0, tmm4, tmm5
tilestored   [r11 + r13*1 + C_TMP_OFFSET(0,0)], tmm0     // Store C tmp [m,n] = [0,0]
tileloadd    tmm6, [r9 + r13*1 + B_OFFSET(2,1)]          // Load B [k,n] = [2,1]
```

```
/*2 of 2*/
tdpbusd        tmm2, tmm4, tmm6
tilestored     [r11 + r13*1 + C_TMP_OFFSET(0,1)], tmm2          // Store C tmp [m,n] = [0,1]
tileloadd      tmm4, [r8 + r12*1 + A_OFFSET(1,2)]               // Load A [m,k] = [1,2]
tdpbusd        tmm1, tmm4, tmm5
tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,0)], tmm1          // Store C tmp [m,n] = [1,0]
tdpbusd        tmm3, tmm4, tmm6
tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,1)], tmm3          // Store C tmp [m,n] = [1,1]

vcvtdq2ps      zmm0 , [r11 + C_TMP_OFFSET(0,0) + 0*TILE_N_B]    // int32 -> float
vmovups        zmm1 , [r14 + Q_OFFSET(0)]                       // q-factors for N=0
vmovups        zmm2 , [r14 + BIAS_OFFSET(0)]                    // biases    for N=0
vfmadd213ps    zmm0 , zmm1 , zmm2                               // zmm0  = zmm0  * q + b
vcvtps2dq      zmm0 , zmm0                                      // float -> int32
vpxord         zmm3 , zmm3 , zmm3                               // Prepare zero ZMM
vpmaxsd        zmm0 , zmm0 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,0) + 0*TILE_N_B], zmm0         // uint32 -> uint8
vcvtdq2ps      zmm4 , [r11 + C_TMP_OFFSET(0,0) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm4 , zmm1 , zmm2                               // zmm4  = zmm4  * q + b
vcvtps2dq      zmm4 , zmm4                                      // float -> int32
vpmaxsd        zmm4 , zmm4 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,0) + 1*TILE_N_B], zmm4         // uint32 -> uint8
vcvtdq2ps      zmm5 , [r11 + C_TMP_OFFSET(1,0) + 0*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm5 , zmm1 , zmm2                               // zmm5  = zmm5  * q + b
vcvtps2dq      zmm5 , zmm5                                      // float -> int32
vpmaxsd        zmm5 , zmm5 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,0) + 0*TILE_N_B], zmm5         // uint32 -> uint8
vcvtdq2ps      zmm6 , [r11 + C_TMP_OFFSET(1,0) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm6 , zmm1 , zmm2                               // zmm6  = zmm6  * q + b
vcvtps2dq      zmm6 , zmm6                                      // float -> int32
vpmaxsd        zmm6 , zmm6 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,0) + 1*TILE_N_B], zmm6         // uint32 -> uint8
vcvtdq2ps      zmm7 , [r11 + C_TMP_OFFSET(0,1) + 0*TILE_N_B]    // int32 -> float
vmovups        zmm8 , [r14 + Q_OFFSET(1)]                       // q-factors for N=1
vmovups        zmm9 , [r14 + BIAS_OFFSET(1)]                    // biases    for N=1
vfmadd213ps    zmm7 , zmm8 , zmm9                               // zmm7  = zmm7  * q + b
vcvtps2dq      zmm7 , zmm7                                      // float -> int32
vpmaxsd        zmm7 , zmm7 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,1) + 0*TILE_N_B], zmm7         // uint32 -> uint8
vcvtdq2ps      zmm10, [r11 + C_TMP_OFFSET(0,1) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm10, zmm8 , zmm9                               // zmm10 = zmm10 * q + b
vcvtps2dq      zmm10, zmm10                                     // float -> int32
vpmaxsd        zmm10, zmm10, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,1) + 1*TILE_N_B], zmm10        // uint32 -> uint8
vcvtdq2ps      zmm11, [r11 + C_TMP_OFFSET(1,1) + 0*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm11, zmm8 , zmm9                               // zmm11 = zmm11 * q + b
vcvtps2dq      zmm11, zmm11                                     // float -> int32
vpmaxsd        zmm11, zmm11, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,1) + 0*TILE_N_B], zmm11        // uint32 -> uint8
vcvtdq2ps      zmm12, [r11 + C_TMP_OFFSET(1,1) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm12, zmm8 , zmm9                               // zmm12 = zmm12 * q + b
vcvtps2dq      zmm12, zmm12                                     // float -> int32
vpmaxsd        zmm12, zmm12, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,1) + 1*TILE_N_B], zmm12        // uint32 -> uint8
```

20.11.2 INTEL® AMX AND INTEL® AVX-512 INTERLEAVING (SW PIPELINING)

A modern CPU has multiple functional units that can execute different instructions simultaneously. For example, a load instruction and an arithmetic instruction can execute in parallel. A commonly used approach for maximizing the utilization of various resources in parallel is the out-of-order execution, where the CPU might alter the order of the instructions to achieve higher resource utilization.

Intel AMX compute instructions are prime candidates for optimization because they utilize resources very lightly (1/2 of the available ALU ports, 1/TILE_M of the time).

The blue post-convolutional code of one iteration could, theoretically, execute in parallel to the **Bold** code in lines 3 through 25 (before the first TileStore) of the next iteration, where iteration is the execution of the code in Example 20-17. Unfortunately, this cannot be done automatically and efficiently by the CPU: since the convolution (**Bold**) and post-convolution (blue) parts of the code tend to be sizable, the CPU can only overlap small portions of them efficiently before it runs out of resources in the out-of-order machine. Thus, a manual (SW) solution is required.

As previously written, the blue code before the first TileStore can be run in parallel with the green code of the next iteration. This would overwrite temporary_C memory, which the post-convolution code reads from. To remove this dependency and maximize parallel execution, use double-buffering on temporary_C. Temporary_C would thus contain two buffers, interchanged every iteration.

In Example 20-28, the content deviates from the previous example by interleaving the current iteration's convolutional code with the previous iteration's post-convolutional code. Temporary_C is double-buffered, with r11 pointing to the buffer of the current iteration and r12 pointing to the previous iteration's buffer. They are exchanged at the end of the iteration.

Example 20-18. An Example of a Short GEMM Fused and Pipelined with Quantization and ReLU

```
/*1 of 3*/
1   ldtilecfg     tc                                          // Load tile config
2   mov           r15, 192                                    // A stride
3   mov           r13, 128                                    // B, C_TMP stride
4   tileloadd     tmm5, [r9 + r13*1 + B_OFFSET(0,0)]          // Load B [k,n] = [0,0]
5   tileloadd     tmm4, [r8 + r15*1 + A_OFFSET(0,0)]          // Load A [m,k] = [0,0]
6   tilezero      tmm0                                        // Zero acc [m,n] = [0,0]
7   vcvtdq2ps     zmm0, [r12 + C_TMP_OFFSET(0,0) + 0*TILE_N_B] // int32 -> float
8   vmovups       zmm1, [r14 + Q_OFFSET(0)]                   // q-factors for N=0
9   vmovups       zmm2, [r14 + BIAS_OFFSET(0)]                // biases    for N=0
10  vfmadd213ps   zmm0, zmm1, zmm2                            // zmm0 = zmm0 * q + b
11  vcvtps2dq     zmm0, zmm0                                  // float -> int32
12  vpxord        zmm3, zmm3, zmm3                            // Prepare zero ZMM
13  vpmaxsd       zmm0, zmm0, zmm3                            // RELU (int32)
14  tdpbusd       tmm0, tmm4, tmm5
15  tileloadd     tmm6, [r9 + r13*1 + B_OFFSET(0,1)]          // Load B [k,n] = [0,1]
16  tilezero      tmm2                                        // Zero acc [m,n] = [0,1]
17  vpmovusdb     [r10 + C_OFFSET(0,0) + 0*TILE_N_B], zmm0    // uint32 -> uint8
18  vcvtdq2ps     zmm4 , [r12 + C_TMP_OFFSET(0,0) + 4*TILE_N_B] // int32 -> float
19  vfmadd213ps   zmm4 , zmm1 , zmm2                          // zmm4  = zmm4  * q + b
20  tdpbusd       tmm2, tmm4, tmm6
21  tileloadd     tmm4, [r8 + r15*1 + A_OFFSET(1,0)]          // Load A [m,k] = [1,0]
22  tilezero      tmm1                                        // Zero acc [m,n] = [1,0]
23  vcvtps2dq     zmm4 , zmm4                                 // float -> int32
24  vpmaxsd       zmm4 , zmm4 , zmm3                          // RELU (int32)
25  vpmovusdb     [r10 + C_OFFSET(0,0) + 1*TILE_N_B], zmm4    // uint32 -> uint8
26  tdpbusd       tmm1, tmm4, tmm5
27  tilezero      tmm3                                        // Zero acc [m,n] = [1,1]
```

```
/*2 of 3*/
vcvtdq2ps       zmm5 , [r12 + C_TMP_OFFSET(1,0) + 0*TILE_N_B]     // int32 -> float
vfmadd213ps     zmm5 , zmm1 , zmm2                                // zmm5  = zmm5  * q + b
vcvtps2dq       zmm5 , zmm5                                       // float -> int32
vpmaxsd         zmm5 , zmm5 , zmm3                                // RELU (int32)
tdpbusd         tmm3, tmm4, tmm6
tileloadd       tmm5 , [r9 + r13*1 + B_OFFSET(1,0)]               // Load B [k,n] = [1,0]
tileloadd       tmm4 , [r8 + r15*1 + A_OFFSET(0,1)]               // Load A [m,k] = [0,1]
vpmovusdb       [r10 + C_OFFSET(1,0) + 0*TILE_N_B], zmm5          // uint32 -> uint8
vcvtdq2ps       zmm6 , [r12 + C_TMP_OFFSET(1,0) + 4*TILE_N_B]     // int32 -> float
vfmadd213ps     zmm6 , zmm1 , zmm2                                // zmm6  = zmm6  * q + b
tdpbusd         tmm0, tmm4, tmm5
tileloadd       tmm6, [r9 + r13*1 + B_OFFSET(1,1)]                // Load B [k,n] = [1,1]
vcvtps2dq       zmm6 , zmm6                                        // float -> int32
vpmaxsd         zmm6 , zmm6 , zmm3                                // RELU (int32)
vpmovusdb       [r10 + C_OFFSET(1,0) + 1*TILE_N_B], zmm6          // uint32 -> uint8
tdpbusd         tmm2 , tmm4, tmm6
tileloadd       tmm4 , [r8 + r15*1 + A_OFFSET(1,1)]               // Load A [m,k] = [1,1]
vcvtdq2ps       zmm7 , [r12 + C_TMP_OFFSET(0,1) + 0*TILE_N_B]     // int32 -> float
vmovups         zmm8 , [r14 + Q_OFFSET(1)]                        // q-factors for N=1
vmovups         zmm9 , [r14 + BIAS_OFFSET(1)]                     // biases    for N=1
vfmadd213ps     zmm7 , zmm8 , zmm9                                // zmm7  = zmm7  * q + b
vcvtps2dq       zmm7 , zmm7                                       // float -> int32
vpmaxsd         zmm7 , zmm7 , zmm3                                // RELU (int32)
tdpbusd         tmm1 , tmm4, tmm5
vpmovusdb       [r10 + C_OFFSET(0,1) + 0*TILE_N_B], zmm7          // uint32 -> uint8
vcvtdq2ps       zmm10 , [r12 + C_TMP_OFFSET(0,1) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps     zmm10 , zmm8 , zmm9                               // zmm10 = zmm10 * q + b
tdpbusd         tmm3 , tmm4, tmm6
tileloadd       tmm5 , [r9 + r13*1 + B_OFFSET(2,0)]               // Load B [k,n] = [2,0]
tileloadd       tmm4 , [r8 + r15*1 + A_OFFSET(0,2)]               // Load A [m,k] = [0,2]
vcvtps2dq       zmm10 , zmm10                                      // float -> int32
vpmaxsd         zmm10 , zmm10, zmm3                               // RELU (int32)
vpmovusdb       [r10 + C_OFFSET(0,1) + 1*TILE_N_B], zmm10         // uint32 -> uint8
tdpbusd         tmm0, tmm4, tmm5
tilestored      [r11 + r13*1 + C_TMP_OFFSET(0,0)], tmm0           // Store C tmp [m,n] = [0,0]
tileloadd       tmm6, [r9 + r13*1 + B_OFFSET(2,1)]                // Load B [k,n] = [2,1]
vcvtdq2ps       zmm11, [r12 + C_TMP_OFFSET(1,1) + 0*TILE_N_B]     // int32 -> float
vfmadd213ps     zmm11, zmm8 , zmm9                                // zmm11 = zmm11 * q + b
vcvtps2dq       zmm11, zmm11                                      // float -> int32
vpmaxsd         zmm11, zmm11, zmm3                                // RELU (int32)
tdpbusd         tmm2, tmm4, tmm6
tilestored      [r11 + r13*1 + C_TMP_OFFSET(0,1)], tmm2           // Store C tmp [m,n] = [0,1]
tileloadd       tmm4, [r8 + r15*1 + A_OFFSET(1,2)]                // Load A [m,k] = [1,2]
vpmovusdb       [r10 + C_OFFSET(1,1) + 0*TILE_N_B], zmm11         // uint32 -> uint8
vcvtdq2ps       zmm12, [r12 + C_TMP_OFFSET(1,1) + 4*TILE_N_B]     // int32 -> float
vfmadd213ps     zmm12, zmm8 , zmm9                                // zmm12 = zmm12 * q + b
tdpbusd         tmm1, tmm4, tmm5
tilestored      [r11 + r13*1 + C_TMP_OFFSET(1,0)], tmm1           // Store C tmp [m,n] = [1,0]
vcvtps2dq       zmm12, zmm12                                      // float -> int32
vpmaxsd         zmm12, zmm12, zmm3                                // RELU (int32)
vpmovusdb       [r10 + C_OFFSET(1,1) + 1*TILE_N_B], zmm12         // uint32 -> uint8
tdpbusd         tmm3, tmm4, tmm6
```

```
/*3 of 3*/
tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,1)], tmm3          // Store C tmp [m,n] = [1,1]

xchg           r11, r12                                         // Swap buffers for current/next iter
```

With the exception of a larger TILE_M (N_ACC=M_ACC=2, TILE_M=16, TILE_K=64, TILE_N=16) on a [256x192] x [192x256] GEMM, application of this algorithm with the parameters laid out in section Section 20.8.1 yielded an 18.5% improvement in running time vs. the non-interleaved code described in Section 20.11.1.

20.11.3 AVOIDING THE H/W OVERHEAD OF FREQUENT OPEN/CLOSE OPERATIONS IN PORT FIVE

When the processor executes Intel AMX compute instructions (TDP*), it usually closes port five (one of the two Intel AVX-512 FMA ports) to conserve power. When the processor senses no more Intel AMX compute instructions in the pipeline, it opens port five. This open/close operation stalls the pipeline for a few cycles. Up to 20% performance degradation may be observed when the Intel AVX-512 instruction block contains 100 to 300 Intel AVX-512 instructions.

We recommend adding one or two TileZero instructions in the middle of the green block, roughly one hundred Intel AVX-512 instructions apart. Such an addition ensures that port five remains closed during blocks of up to three hundred Intel AVX-512 instructions. For longer blocks, it is preferable not to insert TileZero since longer blocks execute faster on two open FMA ports. The processor does not open port five for blocks shorter than one hundred Intel AVX-512 instructions, so no special handling is necessary.

NOTE

The TileZero instruction is considered an Intel AMX compute instruction for that matter.

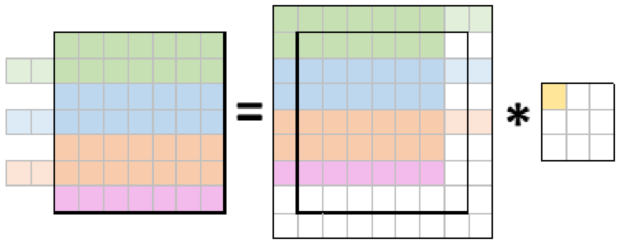


Figure 20-12. Using TileZero to Solve Performance Degradation

20.11.4 POST-CONVOLUTION MULTIPLE OFM ACCUMULATION AND EFFICIENT DOWN-CONVERSION

An important question arises concerning fused post-convolution optimization. What is the optimal block of accumulators processed in a single post-convolution iteration? As a post-processing unit, it is convenient to consider the M_ACC * N_ACC block of tiles accumulated in loops starting at lines 7-8 and 10-11 in Example 20-14 and Example 20-16, respectively. For simplicity, consider only multiples of these accumulation blocks. There is a trade-off between using smaller and larger post-convolution blocks:

Using small post-convolution blocks may have a negative impact by interrupting the convolution flow too often. Conversely, using big post-convolution blocks may also negatively impact by evicting part of the accumulated tiles out of DCU.

The optimal size, therefore, depends very much on the DL network topology and convolution-blocking parameters. Performance studies show that the number of iterations of M_ACC * N_ACC blocks before proceeding to post-convolution iteration may vary from 1 to 7.

As AMX instructions generate a higher precision output (32-bit integers or 32-bit floats) from lower precision inputs (8-bit integers or 16-bit bfloats, respectively), there is a need to convert 32-bit outputs to 8- or 16-bit inputs to be fed to the next DL network layer.

Suppose a single high-precision cache line (512-bit) is processed for conversion at a time. In that case, there will be two or four rounds of processing until a single low-precision cache line is generated for 8- or 16-bit inputs. Potential problems include:

- the number of loads and stores of the same cache line increases 4X or 2X, respectively.
- the next round of processing of the same cache line may occur after this cache line is evicted from DCU.

One of the optimizations mitigating these performance issues is to collect enough high-precision outputs to convert the full low-precision cache line in a single round.

The following drawing shows the conversion flow of 32-bit integers to 8-bit integers. Each colored block at the top represents a single **full** TILE output. The horizontal dimension is OFMs the vertical dimension is spatial).

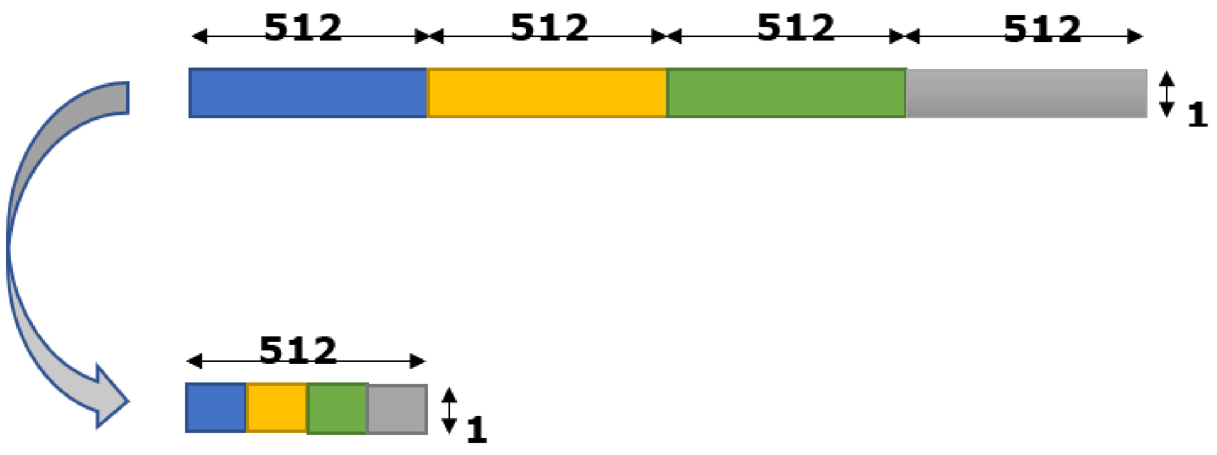


Figure 20-13. A Conversion Flow of 32-bit Integers to 8-bit Integers

To generate full 512-bit cache lines of 8-bit inputs (bottom), a multiple of 64 OFMs should be collected before conversion. Accordingly, to generate full cache lines with 16-bit inputs, a multiple of 32 OFMs should be collected. This often produces better performance results, though it may be viewed as a restriction to convolution blocking parameters (in particular, N_ACC).

Example 20-19 shows the conversion code for two blocks of sixteen cache lines of 32-bit floats converted to a single block of sixteen cache lines of 16-bit bfloats. TMUL outputs are assumed to be placed into a scratchpad *spad,* and the conversion result is placed in the next_inputs buffer.

Example 20-19. Two Blocks of 16 Cache Lines of 32-bit Floats Converted to One Block of 16 Cache Lines of 16-bit BFloat

```
float* spad;
bfloat_16* next_inputs;
inline unsigned inputs_spatial_dim( void ) {
   return /* number of pixels in map */
}
for (int i = 0; i < 16; i++)
{
__m512 f32_0 = _mm512_load_ps(spad);
       __m512 f32_1 = _mm512_load_ps(spad + 16*16);
       __m512 bf16 = _mm512_castsi512_ps(_mm512_cvtne2ps_pbh(f32_1, f32_0));
       _mm512_store_ps(next_inputs, bf16);

     spad += 16; /* Next TILE row */
     next_inputs += 32 * inputs_spatial_dim();
}
```

Example 20-20. Using Unsigned Saturation

```
const int32_t db_sel[16] = { 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15 };
inline __m512i Pack_DwordsToBytes(__m512i dwords[4])
{
   const __m512i sel_reg     = _mm512_load_si512(db_sel);
   const __m512i words_0  = _mm512_packs_epi32(dwords[0], dwords[1]);
   const __m512i words_1  = _mm512_packs_epi32(dwords[2], dwords[3]);
   __m512i bytes             = _mm512_packus_epi16(words_0, words_1);
   bytes                        = _mm512_permutexvar_epi32(sel_reg, bytes);

   return bytes;
}
```

20.12 INPUT AND OUTPUT BUFFERS REUSE (DOUBLE BUFFERING)

Due to the significant computational speedup achieved by the Intel AMX instructions, the performance bottleneck of Intel AMX-enabled applications is usually memory access. The most straightforward way to improve memory utilization is to reduce an application's memory footprint. An application with a smaller memory footprint will keep more of its essential data in the caches while reducing the number of costly cache evictions. This usually improves performance.

In Deep Learning (DL), a simple, efficient way to reduce the memory footprint is to reuse the input and output buffers of various layers in the topology.

The following simple topology illustrates where the previous layer feeds the next layer (left):

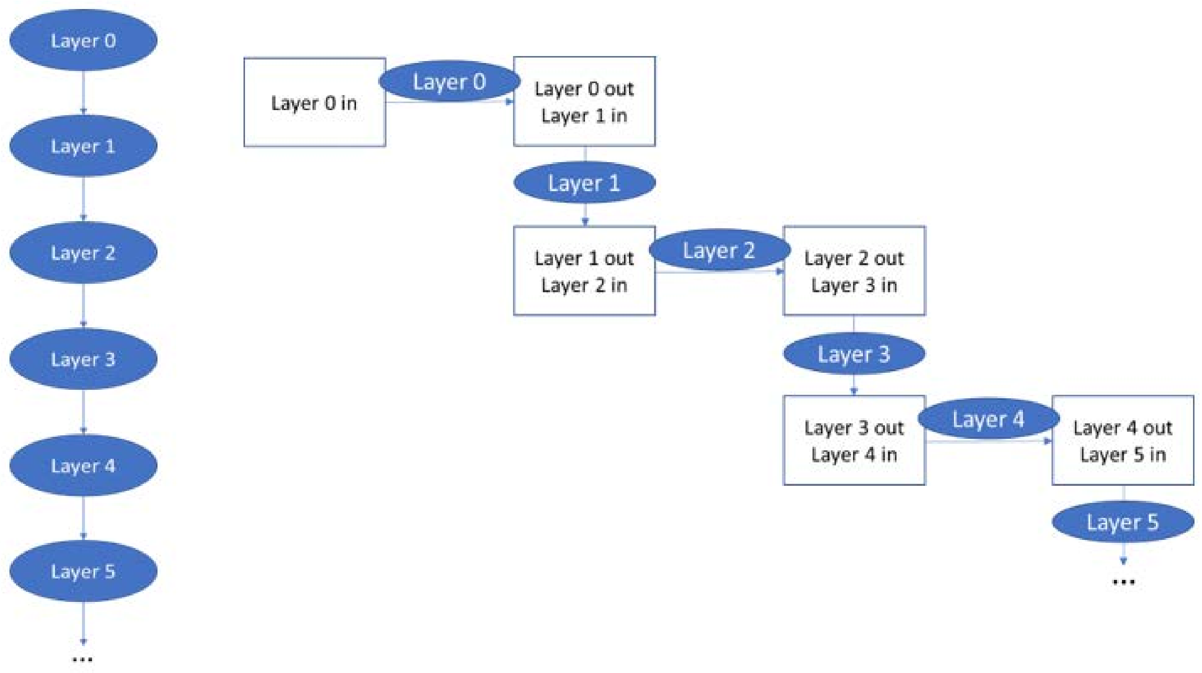


Figure 20-14. Trivial Deep Learning Topology with Naive Buffer Allocation

A straightforward buffer allocation scheme is illustrated on the in Figure 20-14, in which the output of layer N is placed into a dedicated memory buffer which is then consumed as input by layer N+1. In this scheme, such topology with L-layers would require L+1 memory buffers, of which only the last is valuable (containing the final results). The rest of the L memory buffers are single-use and disposable, significantly increasing the application's memory footprint.

The allocation scheme in Figure 20-15 offers an improved scheme whereby the entire topology only requires two reusable memory buffers.

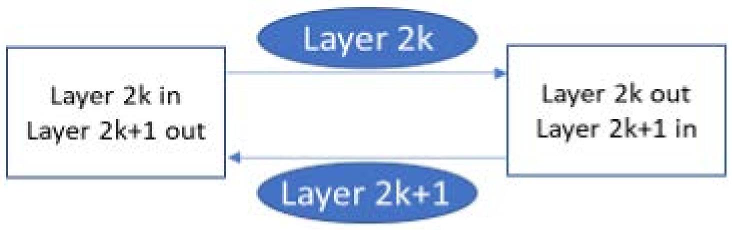


Figure 20-15. Minimal Memory Footprint Buffer Allocation Scheme for Trivial Deep Learning Topology

A more complex topology would require more reusable buffers, but this number is significantly smaller than the naïve approach. ResNet-50, for example, requires only three reusable buffers (instead of 55). Inception-ResNet-V2 requires only five reusable buffers (instead of over 250). This optimization resulted in a 25% improved performance on the int8 end-to-end large batch throughput run of Resnet50 v1.5.

20.13 SOFTWARE PREFETCHES

The CPU employs sophisticated HW prefetchers that predict future access and provide relevant data. This works best when most memory accesses are sequential. For more details on processor hardware prefetchers, see Section 20.13.1.2.

20.13.1 SOFTWARE PREFETCH FOR CONVOLUTION AND GEMM LAYERS

Since the Conv/GEMM kernel is centered around loops over the M, K, and N dimensions of the involved matrices, the access will often be sequential. However, memory blocking, also recommended in this guide, causes the CPU to re-use the same block in the A or B matrices (or both) multiple times during the kernel execution. This means that sometimes the HW prefetcher cannot predict the subsequent access correctly. This opens the opportunity for an SW prefetch algorithm tightly integrated into the Conv/GEMM kernel and can bring in cache lines from future blocks based on the blocking strategy.

While the SW prefetch instruction enables selecting the target cache hierarchy level for the prefetch, this document assumes that the prefetch will go to the MLC. The DCU is too small to prevent the prefetched lines from being evicted before they can be used, and prefetching to LLC may not yield significant improvement.

20.13.1.1 The Prefetch Strategy

The prefetch strategy is highly dependent on the Conv/GEMM kernel method of operation. Assuming the "loops and blocking" design discussed earlier, the kernel operation can probably be split into multiple phases where each phase manages a different part of the matrices (corner, middle, etc.). The developer is encouraged to reduce the program's size by reusing sections for repeatable matrix patterns to avoid overflowing the instruction cache. This can be done by having each section work on relative addresses. The SW prefetch instruction can be integrated into these sections and work on relative addresses. This means that while one section of the code loads addresses for its use, it also prefetches memory for a future section. The future section can be determined by looking at the future indices of any of the M/K/N loop levels.

20.13.1.2 Prefetch Distance

One of the most important decisions when using SW prefetching is the distance between the current and prefetched addresses. Supposing some blocking strategy is employed, it is more complex than adding an offset to the current address. The prefetched address must be set based on the target block of the matrix. If the target block is too close, the prefetched memory might still be in transit when the memory is required, and the CPU will stall, waiting for it to arrive. The data might be evicted if prefetched memory is too distant before it is used. The developer must tune the distance based on the layer/blocking parameters.

As an example heuristic:

- One or two loads for each TMUL operation.
- Where one matrix is already in a register.
- When two registers must be loaded.
- The recommended range between the prefetch time and the consumption time is between 100 and 500 TMUL operations.
- 100 TMUL operations should take about 1600 cycles.
- The maximum number of bytes loaded between prefetch and consumption is 1MB (500 TMUL ops /* 2 loads per ops /* 1K per tile).
- The optimum is probably closer to 100 TMUL ops. At any rate, the developer must check the current CPU architecture and make sure that the MLC will not overflow.

20.13.1.3 To Prefetch A or Prefetch B?

Whether to prefetch A, B, or both depends on the order of layer execution.

In general, the following approaches are available:

- Image affinity.
- Execute the next layer of the same image.
- Complete a single image end-to-end before continuing to the next image in the same mini-batch.

Layer affinity:

- Execute the same layer of the following image.
- Complete a layer for all images in the mini-batch before continuing to the next layer.

The activations (the result of the previous layer) in the CPU caches are seen when image affinity is used. The weights in the caches are found when layer affinity is used. Generally, image affinity is recommended when sizeof(A)>sizeof(B) and layer affinity otherwise. To maximize performance, the developer should tune the switch point between the two methods. The choice between these two methods is also affected by the target matrix for prefetching. If the developer is confident that one of the matrices will already be present in the cache when the Conv/GEMM kernel begins execution, the potential benefit of SW prefetching decreases dramatically.

The size of the A-matrix, B-matrix, and cache.

The developer should sum up the memory requirements of the Conv/GEMM kernel for the current layer and compare it to the size of the cache (MLC). Combined with the previous step, it can indicate whether SW prefetching can yield any performance benefit. When large matrices are involved, there is a greater chance for improvement when prefetching the A- and the B-matrices.

20.13.1.4 To Prefetch or Not to Prefetch C?

It is not the C-matrix we might want to prefetch but rather the final output matrix of the layer, after its post-convolution or post-GEMM phase, including quantization to a lower precision data type. Generally, prefetch those cache lines ahead of time since, with double buffering, these might have been read by previous layers, possibly executed in other cores.

Use the PREFETCHW instruction to read those cache lines into the DCU just in time for the store operations to find them in the DCU ready to be written, avoiding Read For Ownership latency that otherwise delays store completion. The exact timing of issuing the PREFETCHW instruction depends on multiple factors and requires careful tuning to get it right.

20.13.2 SOFTWARE PREFETCH FOR EMBEDDING LAYER

When the memory access pattern is semi-random, it is often impossible for the HW prefetcher to predict since it is based on application logic. In this case, the application may benefit from “proactive” prefetching using the SW prefetch instructions of addresses the application knows it will access soon.

An excellent example is Deep Learning, wherein the recommendation systems often use the embedding layer. The core loop of the embedding algorithm loads indices from an index buffer, and for each index, it loads the corresponding row from the embedding table for further processing. While the index buffer may contain duplicate indices that benefit from CPU caching, the pattern is often considered random or semi-random. This can make the HW prefetcher less efficient. Since the entire content of the index buffer is already known, rows soon to be encountered can be prefetched to the DCU.

Example 20-21. Prefetching Rows to the DCU

```
void prefetched_embedding(uint32_t *a, float *e, float *c, size_t num_indices,
                float scale, float bias, size_t lookahead)
{
    __m512 s = _mm512_set1_ps(scale);
    __m512 b = _mm512_set1_ps(bias);

    for (size_t i = 0; i < num_indices; i++) {
        _mm_prefetch(
            (char const *)&e[a[i + lookahead] * FLOATS_PER_CACHE_LINE],
            _MM_HINT_T0);
        __m512 ereg =
          _mm512_load_ps(&e[((size_t)a[i]) * FLOATS_PER_CACHE_LINE]);
        __m512 creg = _mm512_fmadd_ps(ereg, s, b);
        _mm512_store_ps(&c[i * FLOATS_PER_CACHE_LINE], creg);
    }
}
```

20.14 STORE TO LOAD FORWARDING

Before it gets written to the DCU (first-level cache), store instructions copy data from general purpose, vector, or tile registers into store buffers. All load instructions, other than TileLoad, can load the data they are looking for from the store buffers and memory hierarchy.

The TileLoad instruction can't load data from store buffers. It can only detect that the data is there and must wait until it is written to the memory hierarchy. Once written, TileLoad can read it from the memory hierarchy. This incurs a significant slowdown.

TileStore forwarding to non-TileLoad instructions via store buffers is supported under one restriction: they must both be of cache line size (64 bytes).

Forwarding is generally not advised because this mechanism has outliers. To avoid store-to-load forwarding, ensure enough distance between those two operations in the order of 10s of cycles in time.

20.15 MATRIX TRANSPOSE

This section describes the best-known SW implementations for several matrix transformations of BF16 data.

In this context, **flat format** means:

- Normal (i.e., non-VNNI).
- Unblocked rows (rows of matrices occupy a consecutive region in memory).

The first condition is essential. The second could be relaxed by changing the code in Example 20-22 accordingly. **VNNI format** implies only the second condition (non-blocking of rows). It is important to note that the MxN matrix in flat format will be represented by a (M/2)x(N/*2) matrix in VNNI format.

20.15.1 FLAT-TO-FLAT TRANSPOSE OF BF16 DATA

The primitive block transposed in this algorithm is 32x8 (i.e., 32 rows, eight BF16 numbers each), which is transformed into an 8x32 block (i.e., eight rows of 32 BF16 numbers each).

The implementation uses sixteen ZMM registers and three mask registers.

Input parameters: MxN, sizes of the rectangular block to be transposed. Assuming M is a multiple of 32, and N is a multiple of eight, we may also assume in Example 20-22:

- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains starting address of the input matrix.
- r9 contains starting address of the output buffer.

Example 20-22. BF16 Matrix Transpose (32x8 to 8x32)

```
/*1 of 2 */
mov          r10,        0xf0
kmovd        k1,         r10d
mov          r10,        0xf00
kmovd        k2,         r10d
mov          r10,        0xf000
kmovd        k3,         r10d
mov          rax,        N / 8
L.N:
mov          rdx,        M / 32
L.M:
vbroadcasti32x4 zmm0,                xmmword ptr [r8]
vbroadcasti32x4 zmm0{k1},        xmmword ptr [r8+I_STRIDE*8]
vbroadcasti32x4 zmm0{k2},        xmmword ptr [r8+I_STRIDE*16]
vbroadcasti32x4 zmm0{k3},        xmmword ptr [r8+I_STRIDE*24]
vbroadcasti32x4 zmm1,            xmmword ptr [r8+I_STRIDE*1]
vbroadcasti32x4 zmm1{k1},        xmmword ptr [r8+I_STRIDE*9]
vbroadcasti32x4 zmm1{k2},        xmmword ptr [r8+I_STRIDE*17]
vbroadcasti32x4 zmm1{k3},        xmmword ptr [r8+I_STRIDE*25]
vbroadcasti32x4 zmm2,            xmmword ptr [r8+I_STRIDE*2]
vbroadcasti32x4 zmm2{k1},        xmmword ptr [r8+I_STRIDE*10]
vbroadcasti32x4 zmm2{k2},        xmmword ptr [r8+I_STRIDE*18]
vbroadcasti32x4 zmm2{k3},        xmmword ptr [r8+I_STRIDE*26]
vbroadcasti32x4 zmm3,            xmmword ptr [r8+I_STRIDE*3]
vbroadcasti32x4 zmm3{k1},        xmmword ptr [r8+I_STRIDE*11]
vbroadcasti32x4 zmm3{k2},        xmmword ptr [r8+I_STRIDE*19]
vbroadcasti32x4 zmm3{k3},        xmmword ptr [r8+I_STRIDE*27]
vbroadcasti32x4 zmm4,            xmmword ptr [r8+I_STRIDE*4]
vbroadcasti32x4 zmm4{k1},        xmmword ptr [r8+I_STRIDE*12]
vbroadcasti32x4 zmm4{k2},        xmmword ptr [r8+I_STRIDE*20]
vbroadcasti32x4 zmm4{k3},        xmmword ptr [r8+I_STRIDE*28]
vbroadcasti32x4 zmm5,            xmmword ptr [r8+I_STRIDE*5]
vbroadcasti32x4 zmm5{k1},        xmmword ptr [r8+I_STRIDE*13]
vbroadcasti32x4 zmm5{k2},        xmmword ptr [r8+I_STRIDE*21]
vbroadcasti32x4 zmm5{k3},        xmmword ptr [r8+I_STRIDE*29]
vbroadcasti32x4 zmm6,            xmmword ptr [r8+I_STRIDE*6]
```

```
/*2 of 2 */
vbroadcasti32x4 zmm6{k1},             xmmword ptr [r8+I_STRIDE*14]
vbroadcasti32x4 zmm6{k2},             xmmword ptr [r8+I_STRIDE*22]
vbroadcasti32x4 zmm6{k3},             xmmword ptr [r8+I_STRIDE*30]
vbroadcasti32x4 zmm7,                 xmmword ptr [r8+I_STRIDE*7]
vbroadcasti32x4 zmm7{k1},             xmmword ptr [r8+I_STRIDE*15]
vbroadcasti32x4 zmm7{k2},             xmmword ptr [r8+I_STRIDE*23]
vbroadcasti32x4 zmm7{k3},             xmmword ptr [r8+I_STRIDE*31]
vpunpcklwd      zmm8,     zmm0,       zmm1
vpunpckhwd      zmm9,     zmm0,       zmm1
vpunpcklwd      zmm10,    zmm2,       zmm3
vpunpckhwd      zmm11,    zmm2,       zmm3
vpunpcklwd      zmm12,    zmm4,       zmm5
vpunpckhwd      zmm13,    zmm4,       zmm5
vpunpcklwd      zmm14,    zmm6,       zmm7
vpunpckhwd      zmm15,    zmm6,       zmm7
vpunpckldq      zmm0,     zmm8,       zmm10
vpunpckhdq      zmm1,     zmm8,       zmm10
vpunpckldq      zmm2,     zmm9,       zmm11
vpunpckhdq      zmm3,     zmm9,       zmm11
vpunpckldq      zmm4,     zmm12,      zmm14
vpunpckhdq      zmm5,     zmm12,      zmm14
vpunpckldq      zmm6,     zmm13,      zmm15
vpunpckhdq      zmm7,     zmm13,      zmm15
vpunpcklqdq     zmm8,     zmm0,       zmm4
vpunpckhqdq     zmm9,     zmm0,       zmm4
vpunpcklqdq     zmm10,    zmm1,       zmm5
vpunpckhqdq     zmm11,    zmm1,       zmm5
vpunpcklqdq     zmm12,    zmm2,       zmm6
vpunpckhqdq     zmm13,    zmm2,       zmm6
vpunpcklqdq     zmm14,    zmm3,       zmm7
vpunpckhqdq     zmm15,    zmm3,       zmm7
vmovdqu16       zmmword ptr [r9],                 zmm8
vmovdqu16       zmmword ptr [r9+O_STRIDE],        zmm9
vmovdqu16       zmmword ptr [r9+O_STRIDE*2],      zmm10
vmovdqu16       zmmword ptr [r9+O_STRIDE*3],      zmm11
vmovdqu16       zmmword ptr [r9+O_STRIDE*4],      zmm12
vmovdqu16       zmmword ptr [r9+O_STRIDE*5],      zmm13
vmovdqu16       zmmword ptr [r9+O_STRIDE*6],      zmm14
vmovdqu16       zmmword ptr [r9+O_STRIDE*7],      zmm15

add             r9, 0x40
add             r8, I_STRIDE*32
dec             rdx
jnz             L.M

add             r9, (O_STRIDE*8 — (M/32) * 0X40)
sub             r8, (I_STRIDE*M-0x10)
dec             rax
jnz             L.N
```

Implementation discussion:

- Lines 1-6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9-72 implement the transpose of a primitive block 32x8. It uses 16 ZMM registers (zmm0-zmm15).
- Lines 9-40 implement loading 32 quarter-cache lines into 8 ZMM registers, according to the following picture (numbers are in **bytes**):

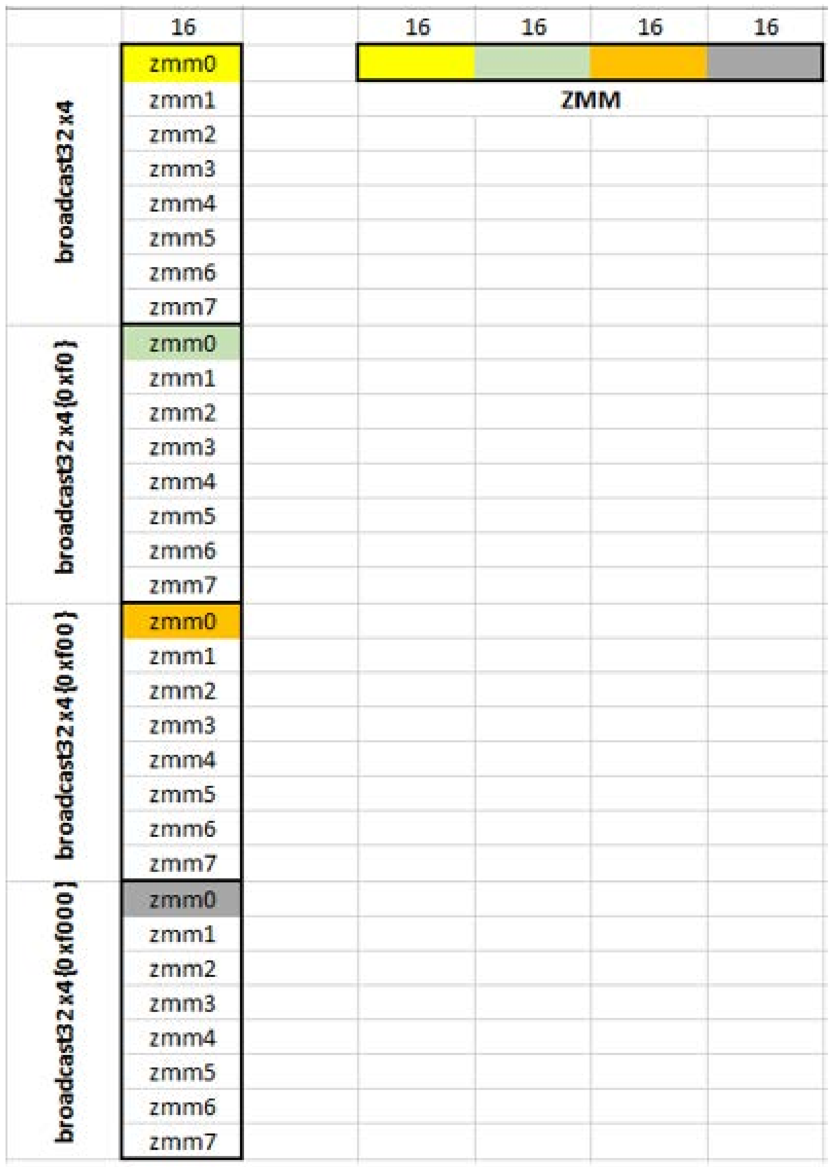


Figure 20-16. Loading 32 Quarter-Cache Lines into 8 ZMM Registers

- Lines 41-64 are transpose flow proper. It creates a transposed block 8x32 in registers zmm8-zmm15.
- Lines 65-72 store transposed block 8x32 to the output buffer.

20.15.2 VNNI-TO-VNNI TRANSPOSE

The primitive block transposed in this algorithm is 8x8 (i.e., eight rows, eight BF16 numbers each), which is transformed into a2x32 block (i.e., two rows of 32 BF16 numbers each).

The implementation uses five ZMM registers and three mask registers.

Input parameters:

- MxN, sizes of the rectangular block to be transposed (*in VNNI format*); it is assumed that M, N are multiples of eight.
- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains starting address to the input matrix.
- r9 contains starting address to the output buffer.
- zmm31 is preloading with four copies of the following constant: unsigned int shuflle_cntrl[4] = {0x05040100, 0x07060302, 0x0d0c0908, 0x0f0e0b0a};

Example 20-23. BF16 VNNI-to-VNNI Transpose (8x8 to 2x32)

```
mov r10, 0xf0
kmovd k1, r10d
mov r10, 0xf00
kmovd k2, r10d
mov r10, 0xf000
kmovd k3, r10d
mov rax, N / 8
L.N:
mov rdx, M / 8
L.M:
vbroadcasti32x4 zmm0, xmmword ptr [r8]
vbroadcasti32x4 zmm0{k1}, xmmword ptr [r8+I_STRIDE*2]
vbroadcasti32x4 zmm0{k2}, xmmword ptr [r8+I_STRIDE*4]
vbroadcasti32x4 zmm0{k3}, xmmword ptr [r8+I_STRIDE*6]
vbroadcasti32x4 zmm1, xmmword ptr [r8+I_STRIDE*1]
vbroadcasti32x4 zmm1{k1}, xmmword ptr [r8+I_STRIDE*3]
vbroadcasti32x4 zmm1{k2}, xmmword ptr [r8+I_STRIDE*5]
vbroadcasti32x4 zmm1{k3}, xmmword ptr [r8+I_STRIDE*7]

vpshufb zmm2, zmm0, zmm31
vpshufb zmm3, zmm1, zmm31
vpunpcklqdq zmm0, zmm2, zmm3
vpunpckhqdq zmm1, zmm2, zmm3

vmovdqu16 zmmword ptr [r9], zmm0
vmovdqu16 zmmword ptr [r9+O_STRIDE], zmm1

add r9, 0x40
add r8, I_STRIDE*8
dec rdx
jnz L.M

add r9, (O_STRIDE*2 - (M/8) * 0x40)
sub r8, (I_STRIDE*M-0x10)
dec rax
jnz L.N
```

BF16 VNNI-to-VNNI Transpose Implementation Discussion

- Lines 1–6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9–22 implement the transpose of a primitive block 32x8. It uses five ZMM registers (zmm0-zmm3, zmm31).
- Lines 9–16 implement loading eight quarter-cache lines into two ZMM registers, according to Figure 20-17 (numbers are in bytes):

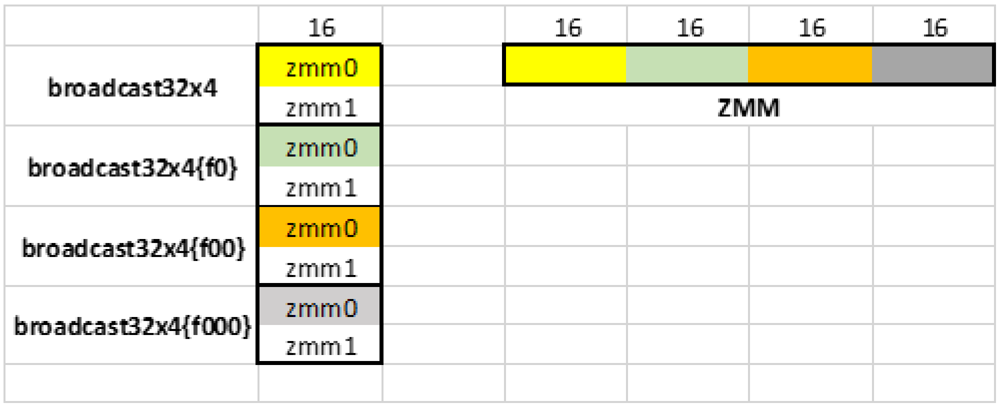


Figure 20-17. Loading Eight Quarter-Cache Lines into Two ZMM Registers

- Lines 17–20 implement simultaneous transpose of four 2x2 blocks of QWORDs (i.e., 2x8 blocks of BF16). It creates a transposed block 2x32 in registers zmm2-zmm3.
- Lines 21–22 store transposed block 2x32 to the output buffer.

20.15.3 FLAT-TO-VNNI TRANSPOSE

The algorithm below is based on: Flat-to-VNNI transpose of WORDs is equivalent to Flat-to-Flat transpose of DWORDs. This is illustrated below (the header numbers are bytes):

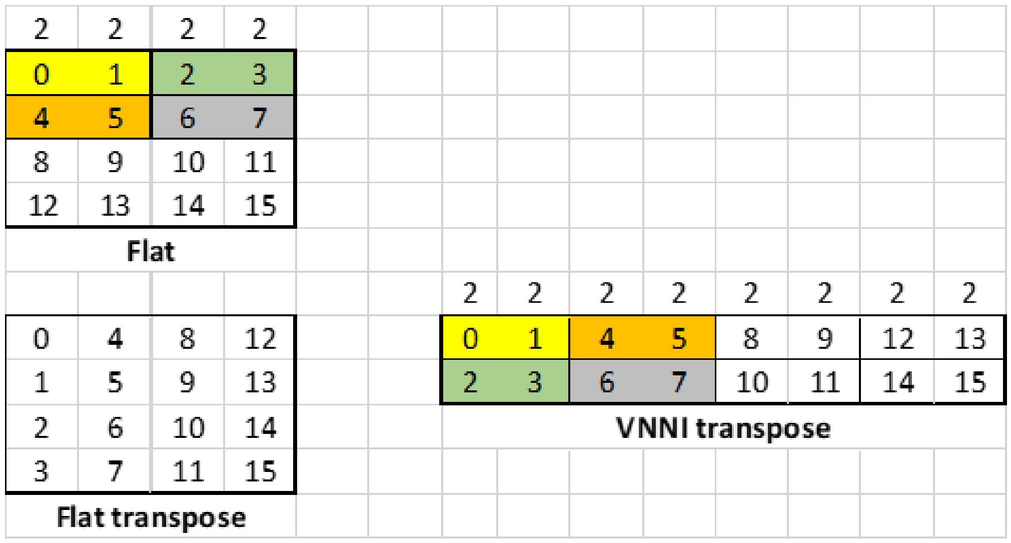


Figure 20-18. Flat-to-VNNI Transpose of WORDs Equivalence to Flat-to-Flat Transpose of DWORDs

The primitive block transposed in this algorithm is 16x8 (i.e., 16 rows, 8 BF16 numbers each), which is transformed into a 4x32 block (i.e., four rows of 32 BF16 numbers each).

The implementation uses eight ZMM registers and three mask registers.

Input parameters:

- MxN, sizes of the rectangular block to be transposed; it is assumed that M multiple of 16, N multiple of eight.
- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains the starting address for the input matrix.
- r9 contains the starting address for the output buffer.

Example 20-24. BF16 Flat-to-VNNI Transpose (16x8 to 4x32)

```
mov r10, 0xf0
kmovd k1, r10d
mov r10, 0xf00
kmovd k2, r10d
mov r10, 0xf000
kmovd k3, r10d
mov rax, N / 8
L.N:
mov rdx, M / 16
L.M:
vbroadcasti32x4 zmm0, xmmword ptr [r8]
vbroadcasti32x4 zmm0{k1}, xmmword ptr [r8+I_STRIDE*4]
vbroadcasti32x4 zmm0{k2}, xmmword ptr [r8+I_STRIDE*8]
vbroadcasti32x4 zmm0{k3}, xmmword ptr [r8+I_STRIDE*12]
vbroadcasti32x4 zmm1, xmmword ptr [r8+I_STRIDE*1]
vbroadcasti32x4 zmm1{k1}, xmmword ptr [r8+I_STRIDE*5]
vbroadcasti32x4 zmm1{k2}, xmmword ptr [r8+I_STRIDE*9]
vbroadcasti32x4 zmm1{k3}, xmmword ptr [r8+I_STRIDE*13]
vbroadcasti32x4 zmm2, xmmword ptr [r8+I_STRIDE*2]
vbroadcasti32x4 zmm2{k1}, xmmword ptr [r8+I_STRIDE*6]
vbroadcasti32x4 zmm2{k2}, xmmword ptr [r8+I_STRIDE*10]
vbroadcasti32x4 zmm2{k3}, xmmword ptr [r8+I_STRIDE*14]
vbroadcasti32x4 zmm3, xmmword ptr [r8+I_STRIDE*3]
vbroadcasti32x4 zmm3{k1}, xmmword ptr [r8+I_STRIDE*7]
vbroadcasti32x4 zmm3{k2}, xmmword ptr [r8+ I_STRIDE*11]
vbroadcasti32x4 zmm3{k3}, xmmword ptr [r8+ I_STRIDE*15]

vpunpckldq zmm4, zmm0, zmm1
vpunpckhdq zmm5, zmm0, zmm1
vpunpckldq zmm6, zmm2, zmm3
vpunpckhdq zmm7, zmm2, zmm3
vpunpcklqdq zmm0, zmm4, zmm6
vpunpckhqdq zmm1, zmm4, zmm6
vpunpcklqdq zmm2, zmm5, zmm7
vpunpckhqdq zmm3, zmm5, zmm7

vmovups zmmword ptr [r9], zmm0
vmovups zmmword ptr [r9+O_STRIDE], zmm1
vmovups zmmword ptr [r9+O_STRIDE*2], zmm2
vmovups zmmword ptr [r9+O_STRIDE*3], zmm3

add r9, 0x40
add r8, I_STRIDE*16
dec rdx
jnz L.M

add r9, (O_STRIDE*4 - (M/16)*0x40)
sub r8, (I_STRIDE*M-0x10)
dec rax
jnz L.N
```

Implementation Discussion

- Lines 1–6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9–36 implement the transpose of a primitive block 16x8. It uses eight ZMM registers (zmm0–zmm7).
- Lines 9–24 implement loading 16 quarter-cache lines into four ZMM registers zmm0-zmm3, according to Figure 20-19 (numbers are in bytes):

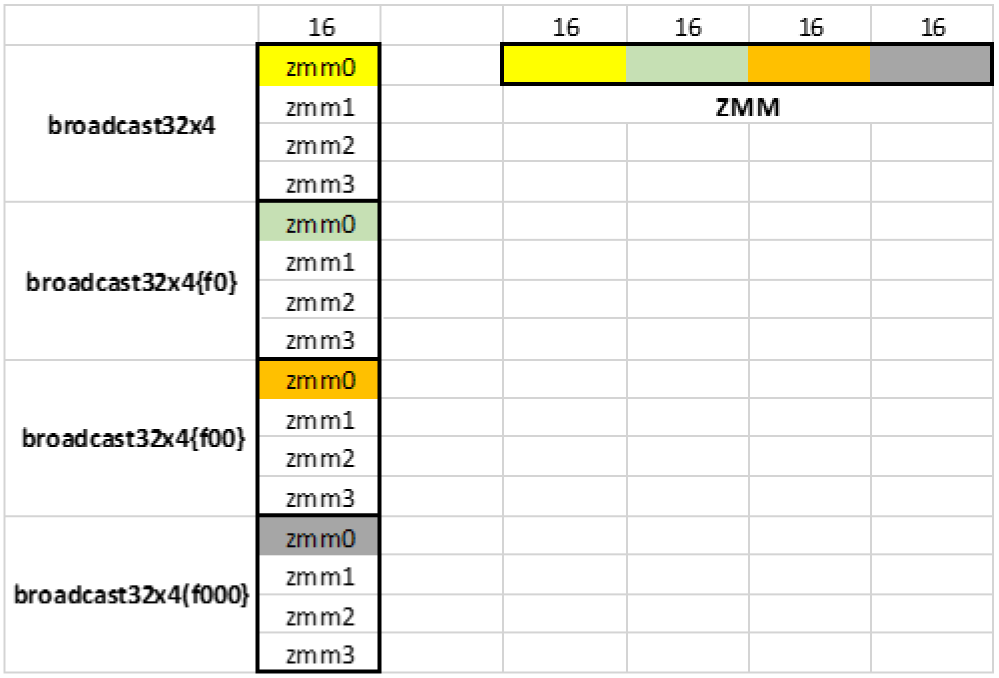


Figure 20-19. BF16 Flat-to-VNNI Transpose

- Lines 25–32 are the transpose flow proper. It creates a transposed block 4x32 in registers zmm0–zmm3.
- Lines 33–36 store transposed block 4x32 to the output buffer.

20.15.4 FLAT-TO-VNNI RE-LAYOUT

The primitive block which is being re-layout in this algorithm is 2x32 (i.e., 2 rows, 32 BF16 numbers each), which is transformed into a 1x64 block (i.e., 1 rows of 64 BF16 numbers each).

The implementation uses **5 ZMM registers and no mask registers**.

Input parameters:

- MxN, sizes of the rectangular block to be transposed; it is assumed that **M multiple of 2, N multiple of 32.**
- I_STRIDE is the row size of input matrix in **bytes**.
- O_STRIDE is the row size of output buffer in **bytes**.
- r8 contains starting address to input matrix.
- r9 contains starting address to output buffer.
- zmm30, zmm31 are preloaded with following constants, respectively:

- const short perm_cntl_1[32] = {0x00, 0x20, 0x01, 0x21, 0x02, 0x22, 0x03, 0x23, 0x04, 0x24, 0x05, 0x25, 0x06, 0x26, 0x07, 0x27, 0x08, 0x28, 0x09, 0x29, 0x0a, 0x2a, 0x0b, 0x2b, 0x0c, 0x2c, 0x0d, 0x2d, 0x0e, 0x2e, 0x0f, 0x2f};
- const short perm_cntl_2[32] = {0x30, 0x10, 0x31, 0x11, 0x32, 0x12, 0x33, 0x13, 0x34, 0x14, 0x35, 0x15, 0x36, 0x16, 0x37, 0x17, 0x38, 0x18, 0x39, 0x19, 0x3a, 0x1a, 0x3b, 0x1b, 0x3c, 0x1c, 0x3d, 0x1d, 0x3e, 0x1e, 0x3f, 0x1f};

Example 20-25. BF16 Flat-to-VNNI Re-Layout

```
1     mov rdx, M / 2
L.M:
 2    mov rax, N / 32
L.N:
 3    vmovups zmm0, zmmword ptr [r8]
 4    vmovups zmm1, zmmword ptr [r8+I_STRIDE]

 5    vmovups zmm2, zmm0
 6    vpermt2w zmm2, zmm30, zmm1
 7    vpermt2w zmm1, zmm31, zmm0

 8    vmovups zmmword ptr [r9], zmm2
 9    vmovups zmmword ptr [r9+0x40], zmm1

10    add r9, 0x80
11    add r8, 0x40
12    dec rax
13    jnz L.N

14    add r9, (O_STRIDE - (N/32)*0x80)
15    add r8, (I_STRIDE*2 – (N/32)*0x40)
16    dec rdx
17    jnz L.M
```

BF16 Flat-to-VNNI Re-Layout Implementation Discussion

- Lines 1, 2 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 3, 4 implement loading two full cache lines into two ZMM registers zmm0-zmm1, from consecutive rows of the input matrix.
- Lines 5 through 7 implement the re-layout of a primitive block 2x32. It uses five ZMM registers (zmm0–zmm2, zmm30-zmm31).
- Lines 8, 9 implement storing two full cache lines in two ZMM registers zmm1-zmm2, into consecutive columns of the output matrix.

20.16 MULTI-THREADING CONSIDERATIONS

20.16.1 THREAD AFFINITY

As with Intel AVX-512 code, it is advised to fully define thread affinity and object affinity to process a single object in the same physical core, thus keeping the activations in core caches (unless larger than the size of the caches). This advice becomes imperative with Intel AMX code since those applications are more sensitive to memory-related issues.

20.16.2 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

Running more than one Intel AMX thread on the same physical core may result in overall performance loss due to the two threads competing for the same hardware resources. Scheduling a non-Intel AMX thread next to an Intel AMX thread on the same core may decrease the thread performance more than one expects due to normal competition for resources.

For optimum performance, please choose one of the following options in priority order:

1. Schedule one Intel AMX thread per physical core on one of its logical processors, while leaving the other logical processors idle.
2. Affintize a software thread that executes an endless TPAUSE CO.2 loop next to the Intel AMX thread.
    a. This prevents other threads from being scheduled on that logical processor.
        1) All hardware resources within the physical core are therefore allocated to the Intel AMX thread.
        2) This endless loop thread must terminate when the Intel AMX thread is about to terminate.
3. Code pause loops of thread pool threads that are waiting for the next task to be assigned to them with UMWAIT or TPAUSE C0.2 rather than with PAUSE, TPAUSE C0.1, or a non-pausing spin.

20.16.3 WORK PARTITIONING BETWEEN CORES

Deep Learning (DL) applications must often adhere to latency requirements that cannot be fulfilled within a single core. In these cases, a single object's processing must be partitioned between multiple cores.

Additionally, often the output of one layer is the input of the next layer. Due to the nature of the computations in DL applications, partitioning over different dimensions (N, M, K) will have different implications for the data locality in the core's caches. Minimize importing data from a different core's caches if possible as this can hamper performance.

20.16.3.1 Partitioning Over M

Partitioning a DL layer over the M dimension has the advantage of nearly complete data locality. The layer's output is also partitioned by M between the cores and is, therefore, already in the cache of the corresponding core at the beginning of the next layer. Figure 20-20 shows this schematically.

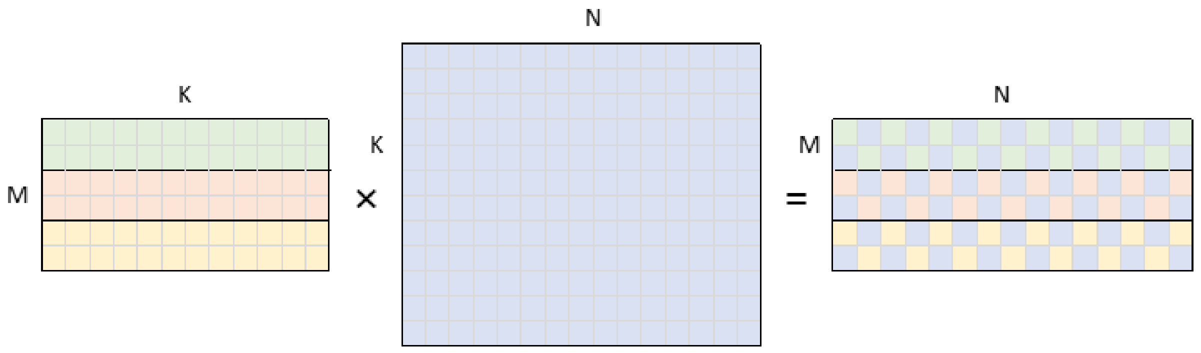


Figure 20-20. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the M-Dimension

Here the data read and written by each of the three cores is bound by a black rectangle.
It should be noted that in the case of convolutions, limited overlap in the M-dimension of the activations occurs between neighboring cores. Due to the convolutions, a finite-sized filter is slid over the activations. Thus, the M-dimension overlaps (KH-1)/*W (refer to Example 20-13) between the two neighboring cores.

- **Advantages:** When multiple layers in a chain are partitioned by the M-dimension between the same number of cores, each core has its data in its local cache.
- **Disadvantages:** All the cores read the B-matrix (or weights in convolutions) entirely, which might pose a bandwidth problem if the B-matrix is large.

20.16.3.2 Partitioning Over N

Partitioning a DL layer over the N-dimension reduces the read bandwidth in GEMMs with large B-matrices or large weights in convolutions. Each core reads a portion of the B-matrix in this scenario:

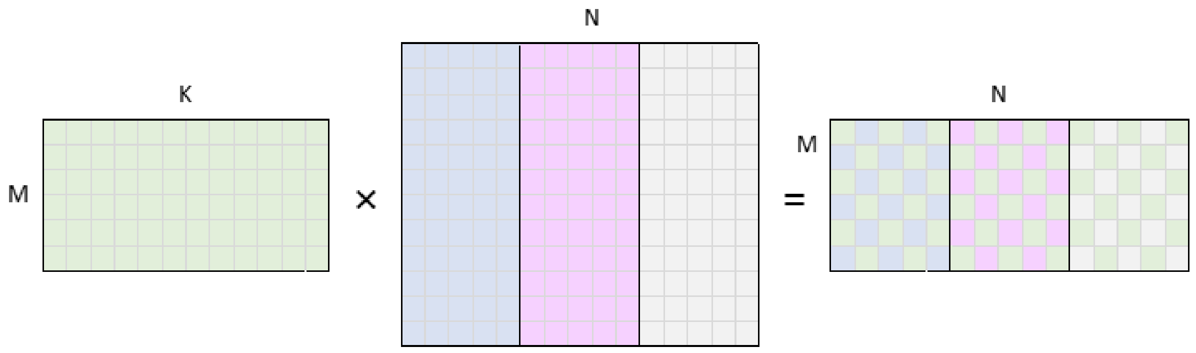


Figure 20-21. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the N-Dimension

Unfortunately, the output of the layer is also partitioned by the N-dimension between the cores, which is incompatible with M and N partitioning of the subsequent layer. For visualization, compare the right side of Figure 20-21 to the left side of Figures 20-20 and 20-21. In this scenario, a core in the subsequent layer is guaranteed to have most of its data from outside its local caches. This is not the case in K-dimension partitioning (see Section 20.16.3.3), but it also comes at a price.

- **Advantages:** It may reduce read bandwidth significantly in case of large B / large weights.
- **Disadvantages:** If the next layer is partitioned by M or by N, most of the activations in the next layer will not reside in the local caches of the corresponding cores.

20.16.3.3 Partitioning Over K

Partitioning a DL layer over the K-dimension reduces the read bandwidth in GEMMs with large K-dimensions by reducing the amount of data being read from the A- and B-matrices (activations and weights in convolutions). Each core reads a portion of the matrices in this scenario, as illustrated in Figure 20-22.

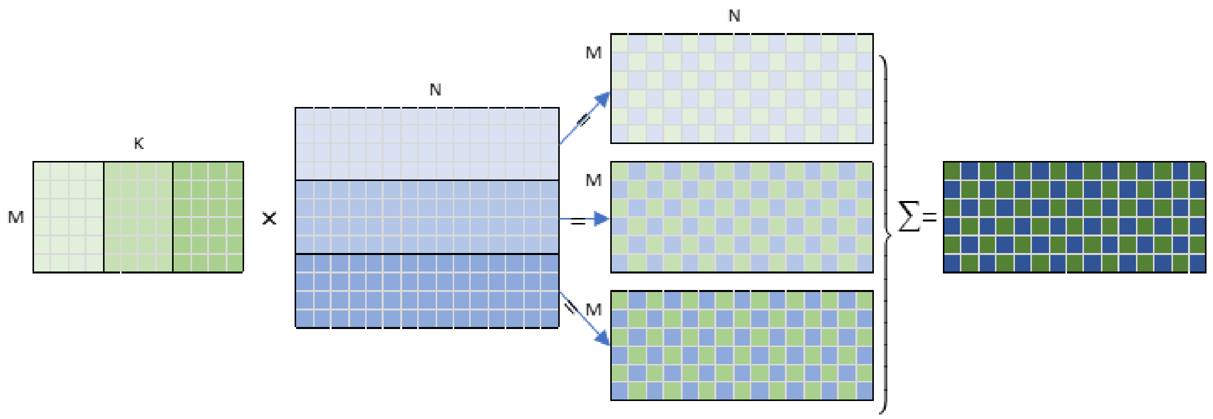


Figure 20-22. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the K-Dimension

Additionally, if a layer is partitioned by the N-dimension and the subsequent layer is partitioned by the K-dimension, the activation data will reside in the local caches of the cores in layer partitioned by the K-dimension. For visualization, compare the right side of Figure 20-21 with the left side of Figure 20-22. Unfortunately, this comes at a price: each core prepares partial results of the entire C-matrix. To obtain final results, either a mutex (or several mutexes) is required to guard the write operations into the C-matrix, or a reduction operation is needed at the end of the layer. The mutex solution is not advised because threads will be blocked for a significant time. A reduction runs the risk of being costly since it entails the following:

- A synchronization barrier is required before the reduction.
- Reading a potentially large amount of data during the reduction:
  - There are T copies of the C-matrix, where T is the number of threads (the example has three).
  - The size of the matrices before the reduction is x2 (in case of a bfloat16 datatype) or x4 (in case of int8 datatype) times larger than the output C-matrix.
  - During the reduction, most of the cores' data will come outside their local cache hierarchy.

20.16.3.4 Memory Bandwidth Implications of Work Partitioning Over Multiple Dimensions

OpenMP offers a convenient interface for nested loop parallelization. For example, one could partition the N, M, and K dimensions can be partitioned automatically between threads using Example 20-26.

Example 20-26. GEMM Parallelized with omp Parallel for Collapse

```
#pragma omp parallel for collapse(2)

for (int n = 0; n < N; n += N_ACC*TILE_N) {
  for (int m = 0; m < M; m += M_ACC*TILE_M) {
    ...
  }
}
```

The collapse clause specifies how many loops within a nested loop should be collapsed into a single iteration space and divided between the threads. The order of the iterations in the collapsed iteration space is the same as though they were executed sequentially.

If there is no specified schedule, OpenMP automatically uses schedule(static,1), resulting in the sequential assignment of loop iterations to threads.

If we assume `N=4*N_ACC*TILE_N` and `M=4*M_ACC*TILE_M` wherein the K-dimension is deliberately excluded from consideration due to its problematic nature, there would be 4*4=16 iterations in the two nested loops. Now assume the division of iterations between three threads. As shown in Table 20-7, the code in Example 20-26 would result in a partition of the iterations between threads.

Table 20-7. A Simple Partition of Work Between Three Threads

							A	B	C
Thread 0:	0.0	0.3	1.2	2.1	3.0	3.3	100%	100%	38%
Thread 1:	0.1	1.0	1.3	2.2	3.1		100%	100%	100%
Thread 2:	0.2	1.1	2.0	2.3	3.2		100%	100%	100%

Where every cell of the form n’,m’ contains the n’=n/N_ACC*TILE_N and m’=m/M_ACC*TILE_M indices from the loops in Example 20-19.

It is clear from Table 20-7 that each of the three threads executes at least one iteration with n’=0,1,2,3 and at least one iteration with m’=0,1,2,3. This means that every thread reads all of A and all of B.

By rearranging the work between threads in the following partitioning, the size of the B read is reduced by each thread by 50%, which might be significant in workloads where B is large. Similarly, the size of A can be reduced by 50% by swapping m’ and n’ indices for workloads with a large A.

Table 20-8. An Optimized Partition of Work Between Three Threads

							A	B	C
Thread 0:	0.0	0.1	0.2	0.3	3.0	3.1	100%	50%	38%
Thread 1:	1.0	1.1	1.2	1.3	3.2	3.3	100%	50%	38%
Thread 2:	2.0	2.1	2.2	2.3			100%	25%	25%

20.16.4 RECOMMENDATION SYSTEM EXAMPLE

Many recommendation systems are built from a few GEMM layers that follow each other, an Embedding layer, and a layer connecting them. They are generally split into four distinct tasks:

1. Bottom GEMMs (MLPs).
2. Embedding.
3. Bottom MLP + Embedding Concat, GEMM, and Reshape.
4. Top GEMMs (MLPs).

The first two are independent so that they can execute in parallel. Their output feeds into the third task, whose output, in turn, feeds into the fourth task.

A few notes:

- Recommendation systems usually use a large batch to rank a reasonably large set of options.
- The GEMM layers are usually compute- or cache-bandwidth limited, whereas the Embedding layer is memory-bandwidth limited.
- Recommendation systems are real-time and therefore limited to a specific latency.

When the latency requirement is a few milliseconds, the recommendation system topology has to be multi-threaded across several cores. The previous section discussed GEMM partitioning across multiple cores. This section deals with work partition between the four different tasks.

Figure 20-23 proposes a way to split the three tasks across machine cores. The block sizes in the chart are for illustration purposes only and do not represent any specific recommendation system.

Those three tasks can then be split into two tasks due to Bottom MLPs and Embedding independence. Those two tasks feed the other tasks: Bottom MLP + Embedding Concat, GEMM, Reshape, and Top MLPs. The latter tasks are merged into a single task. Choosing the number of cores for each task is a trial-and-error exercise. It may involve a phase for analyzing time required to execute each task across different cores.

Because of a dependency between the Bottom MLPs, Embedding tasks, and the third task, a barrier exists between them, implying a potential wait-time immediately following the faster layers.

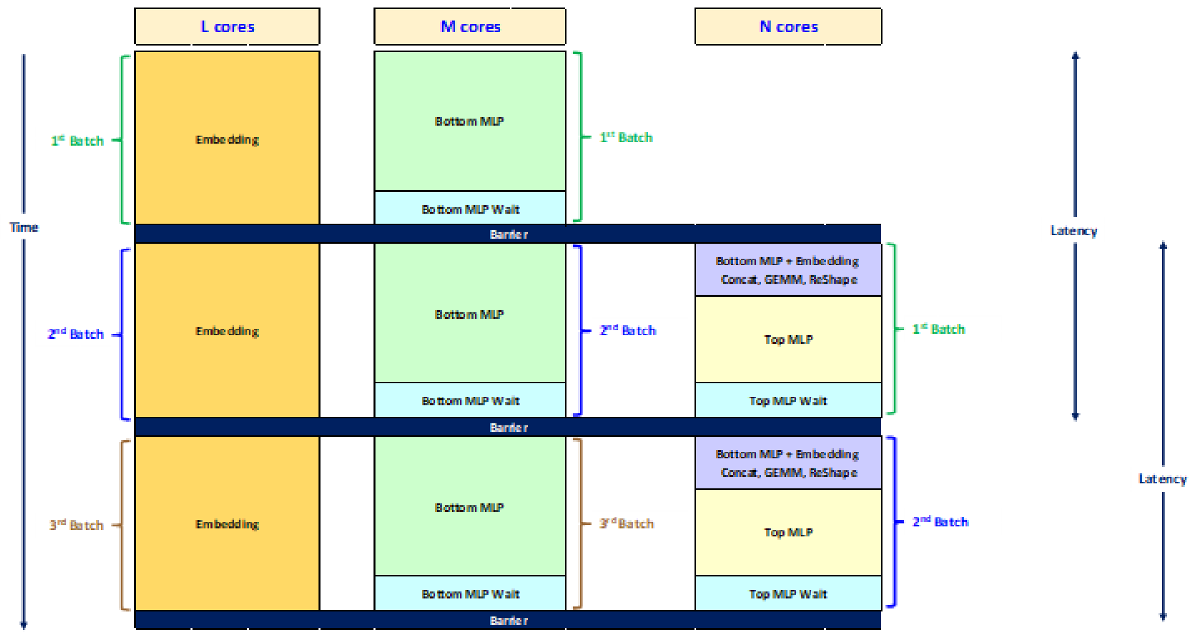


Figure 20-23. A Recommendation System Multi-Threading Model

20.17 SPARSITY OPTIMIZATIONS FOR INTEL® AMX

This section describes how Intel AMX can be further optimized for operations on sparse matrices. An example use case can be the inference of sparse neural networks, where the sparse weights are known to initially reside in DRAM due to the "online" usage model or large model capacity. In those cases, the primary performance bottleneck would be bringing the weights from DRAM. A helpful optimization technique for this case is to get the weights from DRAM in a compressed format, decompress them into the local caches using Intel AVX-512, and perform Intel AMX computations on the decompressed data.

The compressed matrix format can consist of the following components:

- **compressed[]:** an array of non-zero matrix entries.
- **mask[]:** a bit-per-element array for the full matrix. 0 signifies the corresponding element is 0. 1 signifies a non-zero value that exists in the **compressed[]** array mentioned above.

The compressed format can be computed off-line. The sparsity bitmask **mask[]** can be generated using the Intel AVX-512 *VPTESTMB* instruction on the sparse data. The **compressed[]** array can be generated using the Intel AVX-512 *VPCOMPRESS* instruction on the sparse data using the sparsity bitmask.

The code in Example 20-27 uses Intel AVX-512 to generate ***num*** rows of decompressed data, assuming 8-bit elements and 64 elements per tile row.

Example 20-27. Byte Decompression Code with Intel® AVX-512 Intrinsics

```
// uint8_t* compressed_ptr is a pointer to compressed data array
// __mmask64* compression_masks_ptr is a pointer to bitmask array
// uint8_t* decompressed_ptr is a pointer to decompressed data array

for (int i=0; i < num ; i++) {
  __m512i compressed = _mm512_loadu_epi32(compressed_ptr);
  __mmask64 mask = _load_mask64(compression_masks_ptr);
  __m512i decompressed_vec = _mm512_maskz_expand_epi8(mask, compressed);
  _mm512_store_epi32(decompressed_ptr, decompressed_vec);
  decompressed_ptr += 64; // 64 bytes per decompressed row
  compressed_ptr += _mm_countbits_64(mask); // advance compressed pointer by number of non-zero elements
  compression_masks_ptr ++; //64 bitmask bits per decompressed row
}
```

The matrix multiplication code will load the decompressed matrix to tiles from **decompressed[]**, an array containing the decompressed matrix data.

The decompression code makes use of the Intel AVX-512 date expand operation is shown in Figure 20-24.

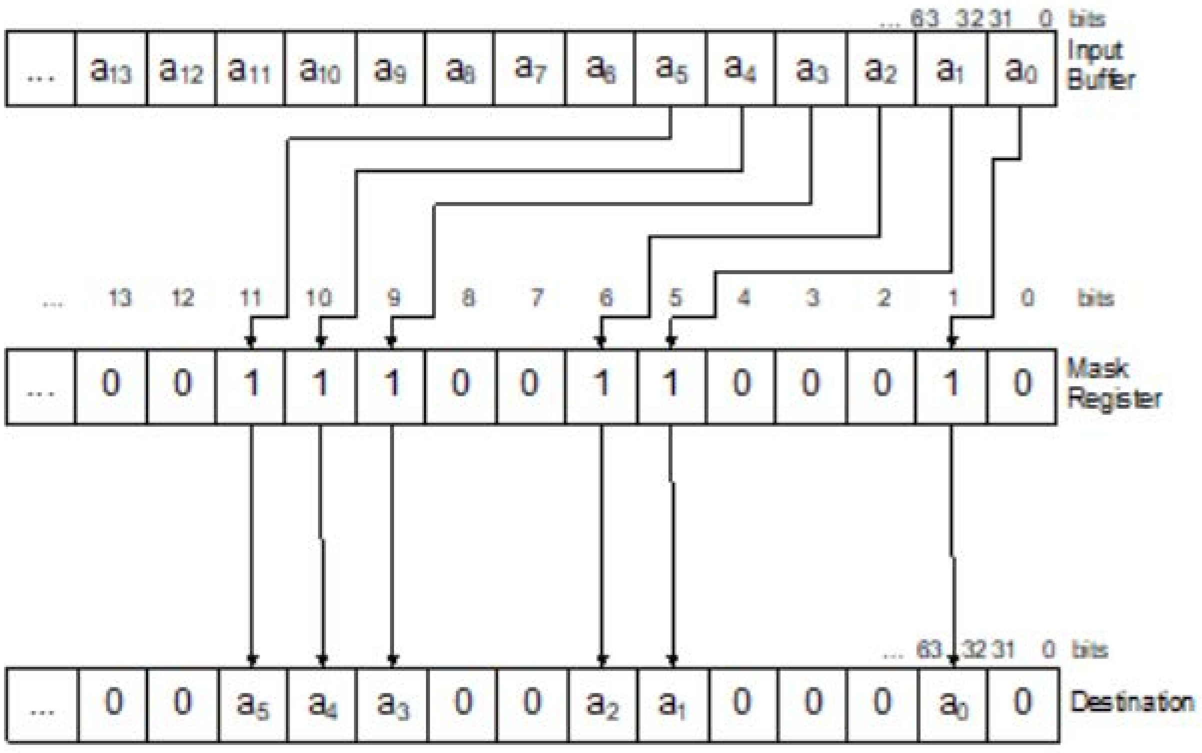

... 63 32 31 0 bits
Input Buffer
... a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
... 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bits
Mask Register
... 0 0 1 1 1 0 0 1 1 0 0 0 1 0
... 63 32 31 0 bits
Destination
... 0 0 a_5 a_4 a_3 0 0 a_2 a_1 0 0 0 a_0 0

Figure 20-24. Data Expand Operation

Decompression code for 16-byte elements can be designed in the same way.

For the best performance, apply the following optimizations:

- **Interleaving:** Fine-grained interleaving of decompression code and matrix multiplication to overlap Intel AVX-512 decompression with Intel AMX computation.
- **Decompress Early:** Prepare the decompressed buffer before immediate Intel AMX use to avoid store forwarding issues.
- **Buffer Reuse:** Decompressing the full sparse matrix could overflow the CPU caches. For best cache reuse, it is recommended to have a decompressed buffer that can hold two decompressed panels of the sparse matrix. While matrix is multiplying with one panel, decompress the next panel for the subsequent iteration. In the subsequent iteration, decompress the next panel into the first half of the decompressed buffer that is no longer used, and so on.
- **Decompress Once:** Coordinate the matrix multiplication blocking and loop structure with the decompression scheme to minimize the number of times the same portion of the sparse matrix is decompressed. For example, if the B-matrix is sparse, traversing the entire vertical M-dimension will compress every vertical panel of the B-matrix only once.

20.18 TILECONFIG/TILERELEASE, CORE C-STATE, AND COMPILER ABI

For a function to use tile registers, it needs to configure them. For the LDTILECFG instruction definition, see Section 20.2. LDTILECFG creates an Intel AMX state which is kept valid until the TILERELEASE instruction is issued. TILERELEASE resets the Intel AMX state back to INIT. When the Intel AMX state is valid, and the OS issues the MWAIT instruction trying to move the physical processor, it executes on to Core C6 State. The 4th Generation Intel® Xeon® Scalable processor based on the Sapphire Rapids microarchitecture will not enter Core C6 even if the sibling logical processor is idle. This is because it lacks the dedicated backing store to keep the Intel AMX state until waking up. The Core C-State is demoted to C1 instead.

This is not an issue in Linux and Windows, where only the idle process issues the MWAIT instruction. The Idle Process in both operating systems does not use the Intel AMX ISA, so its Intel AMX tile state is always invalid (INIT). If still valid, the Intel AMX tile state will have previously been saved in an OS-defined area in memory while context-switching between a thread that uses Intel AMX and the Idle Process thread.

20.18.1 ABI

The tile data registers (tmm0 – tmm7) are volatile. Their contents are passed back and forth between functions through memory. No tile register is saved and restored by the callee. Tile configuration is also volatile. The compiler saves and restores tile configuration and tile register contents if the register(s) need to live across the function call. The compiler eliminates the save instruction because its content remains the same on the stack. The compiler reuses the configured content saved on the stack before the call. All functions need to configure the tile registers themselves; however, tile registers may not be configured across functions.

Please download the System V Application Binary Interface: Intel386 Architecture Processor Supplement, Version1.0.

20.18.2 INTRINSICS

Example 20-28. Identification of Tile Shape Using Parameter m, n, k

```
typedef int _tile1024i __attribute__((__vector_size__(1024), __aligned__(64)));
_tile1024i _tile_loadd_internal(unsigned short m, unsigned short n, const void*base, __SIZE_TYPE__ stride);
_tile1024i _tile_loaddt1_internal(unsigned short m, uunsigned short n, const void*base, __SIZE_TYPE__ stride);
_tile1024i _tile_dpbssd_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbsud_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbusd_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbuud_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbf16ps_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
void_tile_stored_internal(unsigned short m, unsigned short n, void*base, __SIZE_TYPE__ stride, _tile1024i tile);
```

The parameter m, n, k identifies the shape of the tile.

20.18.3 USER INTERFACE

Example 20-29. Intel® AMX Intrinsics Header File

```
/* 1 of 2 */
typedef struct __tile1024i_str {
  const unsigned short row;
const unsigned short col;
  _tile1024i tile;
} __tile1024i;

/// Load tile rows from memory specified by "base" address and "stride" into
/// destination tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILELOADD </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be loaded in memory.
void __tile_loadd(__tile1024i *dst, const void *base, __SIZE_TYPE__ stride);
/// Load tile rows from memory specified by "base" address and "stride" into
/// destination tile "dst". This intrinsic provides a hint to the implementation
/// that the data will likely not be reused in the near future and the data
/// caching can be optimized accordingly.
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILELOADDT1 </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be loaded in memory.
void __tile_stream_loadd(__tile1024i* dst, const void* base, __SIZE_TYPE__ stride);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of signed 8-bit integers in src0 with
/// corresponding signed 8-bit integers in src1, producing 4 intermediate 32-bit
/// results. Sum these 4 results with the corresponding 32-bit integer in "dst",
/// and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
```

```
/* 2 of 3 */
/// This intrinsic corresponds to the <c> TDPBSSD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbssd(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of signed 8-bit integers in src0 with
/// corresponding unsigned 8-bit integers in src1, producing 4 intermediate
/// 32-bit results. Sum these 4 results with the corresponding 32-bit integer
/// in "dst", and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBSUD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbsud(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of unsigned 8-bit integers in src0 with
/// corresponding signed 8-bit integers in src1, producing 4 intermediate 32-bit
/// results. Sum these 4 results with the corresponding 32-bit integer in "dst",
/// and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBUUD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbuud(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Zero the tile specified by "dst".
///
/// \headerfile <immintrin.h>
///
```

```
/* 2of 2 */
/// This intrinsic corresponds to the <c> TILEZERO </c> instruction.
///
/// \param dst
///    The destination tile to be zero. Max size is 1024 Bytes.
void __tile_zero(__tile1024i* dst);
/// Compute dot-product of BF16 (16-bit) floating-point pairs in tiles src0 and
/// src1, accumulating the intermediate single-precision (32-bit) floating-point
/// elements with elements in "dst", and store the 32-bit result back to tile
/// "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBF16PS </c> instruction.
////// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbf16ps(__tile1024i* dst, __tile1024i src0, __tile1024i src1);
/// Store the tile specified by "src" to memory specified by "base" address and
/// "stride".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILESTORED </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be stored in memory.
void __tile_stored(void *base, __SIZE_TYPE__ stride, __tile1024i src);
```

20.18.4 INTEL® AMX INTRINSICS EXAMPLE

In Example 20-30, function foo is called in line 18, and the tile variable 'a' written in line 17 needs to live up to line 21 across the function call. The compiler needs to save the tile data register allocated to 'a' before calling foo, then restore the tile configure register and tile data registers after calling foo. Lines 39, 42, and 46 in Example 20-31 are the save/restore code. Since the configure register doesn't change, the configure register in the stack does not require saving.

Example 20-30. Intel® AMX Intrinsics Usage

```
1 #include <immintrin.h>
2
3 char buf[1024];
4 #define STRIDE 32
5
6 int count = 0;
7 __attribute__((noinline))
8 void foo() {
9   count++;
10 }
11
12 void test_api(int cond, unsigned short row, unsigned short col) {
13   __tile1024i a = {row, col};
14   __tile1024i b = {row, col};
15   __tile1024i c = {row, col};
16
17   __tile_loadd(&a, buf, STRIDE);
18   foo();
19   __tile_loadd(&b, buf, STRIDE);
20   __tile_loadd(&c, buf, STRIDE);
21   __tile_dpbssd(&c, a, b);
22   __tile_stored(buf, STRIDE, c);
23 }
```

clang -O2 -S amx-across-func.c -mamx-int8 -mavx512f -fno-asynchronous-unwind-tables.

Notice the ldtilecfg instruction at the beginning of the function (line 34 in Example 20-31), which sets the Intel AMX registers configuration within the CPU and the TileRelease instruction towards the end of the function. This placement ensures that the Intel AMX state is initialized, thus avoiding the expensive Intel AMX state save/restore in case of a software thread context-switch outside of the Intel AMX function.

Example 20-31. Compiler-Generated Assembly-Level Code from Example 20-30

```
test_api:                       # @test_api
# %bb.0:                           # %entry
        pushq   %rbp
        pushq   %r15
        pushq   %r14
        pushq   %rbx
        subq    $1096, %rsp                 # imm = 0x448
        movl    %edx, %ebx
        movl    %esi, %ebp
        vpxord  %zmm0, %zmm0, %zmm0
        vmovdqu64       %zmm0, (%rsp)
        movb    $1, (%rsp)
        movw    %bx, 20(%rsp)
        movb    %bpl, 50(%rsp)
        movw    %bx, 18(%rsp)
        movb    %bpl, 49(%rsp)
        movw    %bx, 16(%rsp)
        movb    %bpl, 48(%rsp)
        ldtilecfg       (%rsp)
        movl    $buf, %r14d
        movl    $32, %r15d
        tileloadd       (%r14,%r15), %tmm0
        movabsq $64, %rax
        tilestored      %tmm0, 64(%rsp,%rax)    # 1024-byte Folded Spill
        vzeroupper
        callq   foo
        ldtilecfg       (%rsp)
        tileloadd       (%r14,%r15), %tmm0
        tileloadd       (%r14,%r15), %tmm1
        movabsq $64, %rax
        tileloadd       64(%rsp,%rax), %tmm2    # 1024-byte Folded Reload
        tdpbssd %tmm0, %tmm2, %tmm1
        tilestored      %tmm1, (%r14,%r15)
        addq    $1096, %rsp                 # imm = 0x448
        popq    %rbx
        popq    %r14
        popq    %r15
        popq    %rbp
        tilerelease
        retq
```

20.18.5 COMPILATION OPTION

The save/restore is sometimes unnecessary, e.g., when foo does not clobber any tile register. To avoid unnecessary save/restore, compile with "-mllvm -enable-ipra", which does an IPO analysis to get the information on what physical registers are clobbered during the function call. Example 20-32 shows no tile register save/restore across calling foo.

clang -O2 -S amx-across-func.c -mamx-int8 -mavx512f -fno-asynchronous-unwind-tables -mllvm -enable-ipra

Example 20-32. Compiler-Generated Assembly-Level Code Where Tile Register Save/Restore is Optimized Away

```
        .type   test_api,@function
test_api:                               # @test_api
# %bb.0:                                # %entry
        subq    $72, %rsp
        vpxord  %zmm0, %zmm0, %zmm0
        vmovdqu64       %zmm0, 8(%rsp)
        movb    $1, 8(%rsp)
        movw    %dx, 28(%rsp)
        movb    %sil, 58(%rsp)
        movw    %dx, 26(%rsp)
        movb    %sil, 57(%rsp)
        movw    %dx, 24(%rsp)
        movb    %sil, 56(%rsp)
        ldtilecfg       8(%rsp)
        movl    $buf, %eax
        movl    $32, %ecx
        tileloadd       (%rax,%rcx), %tmm0
        callq   foo
        tileloadd       (%rax,%rcx), %tmm1
        tileloadd       (%rax,%rcx), %tmm2
        tdpbssd %tmm1, %tmm0, %tmm2
        tilestored      %tmm2, (%rax,%rcx)
        addq    $72, %rsp
        tilerelease
        vzeroupper
        retq
.Lfunc_end1:
        .size   test_api, .Lfunc_end1-test_api
```

20.19 INTEL® AMX STATE MANAGEMENT

Intel AMX is XSAVE supported, meaning that it defines processor registers that can be saved and restored using instructions of the XSAVE feature set. Intel AMX is also XSAVE enabled, meaning that system software must enable it before it can be used.

The XSAVE feature set operates on state components, each a discrete set of processor registers (or parts of registers). Intel AMX is associated with two state components, XTILECFG and XTILEDATA. The XSAVE feature set organizes state components using state-component bitmaps. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. Intel AMX defines bits 18:17 for its state components (collectively, these are called AMX state):

- State component 17 is used for the 64-byte TILECFG register (XTILECFG state).
- State component 18 is used for the 8192 bytes of tile data (XTILEDATA state).

These are both user-state components, meaning the entire XSAVE feature set can manage them. In addition, it implies that setting bits 18:17 of extended control register XCR0 by system software enables Intel AMX. If those bits are zero, an Intel AMX instruction execution results in an invalid-opcode exception (#UD).

About the XSAVE feature set's INIT optimization, the Intel AMX state is in its initial configuration if the TILECFG register is zero and all tile data are zero.

Enumeration and feature-enabling documentation can be found in Section 20.2.

An execution of XRSTOR or XRSTORS initializes the TILECFG register (resulting in TILES_CONFIGURED = 0) in response to an attempt to load it with an illegal value. Moreover, an execution of XRSTOR or XRSTORS that is not directed to load XTILEDATA leaves it unmodified, even if the execution is loading XTILECFG.

It is highly recommended that developers execute TILERELEASE to initialize the tiles at the end of the Intel AMX instructions code region. More on this is in Section 20.18.

If the system software does not initialize the Intel AMX state first (by executing TILERELEASE, for example), it may disable Intel AMX by clearing XCR0[18:17], by clearing CR4.OSXSAVE, or by setting IA32_XFD[18].

20.19.1 EXTENDED FEATURE DISABLE (XFD)

The XTILEDATA state component size is 8 KBytes, and an operating system may, by default, prefer not to allocate memory for the XTILEDATA state for every user thread. An operating system that enables Intel AMX might select a fault when user threads use the feature. That way, it can allocate a large enough state save area only for the user threads using the feature. An operating system may offer an API for the user threads to declare their intention to use Intel AMX and allow the OS to preallocate the state and avoid an exception when Intel AMX is used for the first time.

See Linux API and Windows API for more details.

Extended feature disable (XFD) is added to the XSAVE feature set to support such usage. See the Intel® AMX Architecture Definition for XFD documentation.

20.19.2 ALTERNATE SIGNAL HANDLER STACK IN LINUX OPERATING SYSTEM

When programs use an alternate signal handler stack, the stack size should be adjusted to accommodate the additional Intel AMX state. See Using XSTATE Features in User-Space Applications for more details.

20.20 USING INTEL® AMX TO EMULATE HIGHER PRECISION GEMMS

Intel AMX/TMUL has instructions that enable matrix-matrix operations such as multiplication on small precision elements. This section considers how to use the low-precision Intel AMX instructions to approximate the answers to matrix-matrix multiplication of higher-precision terms. Even if low-precision inputs are Bfloat16 or Integer8, one can still combine the transforms to approximate matrix-matrix multiplication in higher precisions.

Pay attention to the exponent range and mantissa bits when approximating higher precisions. There are IEEE-754 double precision numbers (FP64) that aren’t representable as single precision (FP32) or lower precisions. These are typically range-based issues in the exponent bits. FP64 has more exponent bits than FP32. However, scaling factors can overcome most range-based problems. If A is a matrix of FP64 values, then A (as a sum of Bfloat16 matrices) cannot generally be represented. Scaling factors can, however, be used to get around most issues. The A-matrix as s1*A1 + s2*A2 + … + sn*An can be written where each matrix A_i is lower precision, and each si is a constant scaling factor.

For Bfloat16 decomposition of FP32, consider the following:

- Let A be a matrix of FP32 values.
- Let A1 = bfloat16(A), a matrix containing RNE-rounded Bfloat16 conversions of A.
- Let A2 = bfloat16(A – fp32(A1)).
- Let A3 = bfloat16(A – fp32(A1) – fp32(A2)).
- Now A is approximately A1 + A2 + A3.

Once one has written two matrices as a sum of lower precision matrices, one can run AMX/TMUL on the product to approximate the higher precision. But to do this effectively, one needs to have higher precision accumulation. There are tricks in the literature for doing higher precision all in a lower precision, such as works on so-called double-double arithmetic. Still, these tend to vary too much from standard matrix-matrix multiplication to be helpful with TMUL. In the case of Bfloat16, having 32-bit accumulation in the product allows one to use Bfloat16 to approximate FP32 accuracy.

Therefore, if A = s1*A1 + s2*A2 + s3*A3, and B = t1*B1 + t2*B2 + t3*B3, then A*B can be computed using AMX/TMUL on the projects Ai*Bj for 1<=i,j<=3, assuming scaling is done carefully to avoid denormals. Assuming FP32 accumulation, the FP32 approximation of A*B can be made by writing out these lower precision multiplies. Scaling factors can be chosen to avoid denormals at times, but they can also be picked in a way that simplifies further steps in the algorithm. In some cases, scaling factors can be chosen to be a power of two, for instance, without significantly reducing the accuracy of the resulting matrix-matrix multiply.

The number of matrices for A or B are picked depending on the mantissa range to cover. If trying to emulate FP32 which has 24 bits of mantissa (including the implicit mantissa bit), it is possible with three Bfloat16 matrices (because each of the triples has 8 bits of mantissa, including the implicit bit.). Here the range is less important because Bfloat16 and FP32 have the same exponent range. Use three Bfloat16 matrices to approximate FP32 precision by BF16x3. Range issues may still come up for BF16x3 cases where A has values close to the maximum or minimum exponent for FP32, but that too can be circumvented by scaling constants. Scaling factors of 2^24 or 2^(-24) suffice to push it far enough away from the boundary to make the computation feasible again. This is dependent upon the closest end of the spectrum.

A few terms from an expansion can also be dropped. For instance, in the BF16x3 case, where there are three As and three Bs, nine products may result. That is:

A*B = (A1+A2+A3)*(B1+B2+B3) = (A1*B1) + (A1*B2 + A2*B1) + (A1*B3 + A2*B2 + A3*B1) + (A2*B3 + A3*B2) +(A3*B3).

The parentheses in the last equation are intentionally derived so that all entries in the same "bin" are put together, and there are nine entries of the form Ai*Bj. This example has five bins, each with its own set of parentheses. In the Bfloat16 case, |Ai| <= |A_i-1}| / 256. This shows the last two bins (with A2*B3,A3*B2,A3*B3) are too small to contribute significantly to the answer, which is why if there are Y terms on each side of A*B, only (Y+1)*Y/2 multiplies are required, not Y*Y multiplies. In this case, dropping the last three (also the difference between Y*Y – (Y+1)*Y/2 when Y=3.) from the nine multiplies. The last three multiplies in the last two bins have terms less than 2^(-24) as big as the first term. So, A*B can be approximated (ignoring the scaling terms for now) as the sum of the first three most significant bins: A1*B1 + (A1*B2+A2*B1)+(A1*B3+A2*B2*A3*B1). In this case, adding from the least significant bin to the most significant bin (A1*B1) is recommended.

Whenever A and B are each expanded out to Y-terms, computing only Y*(Y+1)/2 products works under the condition that each term has the same number of mantissa bits. If some terms have a different number of bits, then this guideline no longer applies. But for BF16x3, each term covers eight mantissa bits and Y=3, so six products are needed.

Regarding accuracy, the worst-case relative error for BF16x3 may be worse than FP32. However, BF16x3 tends to cover a larger mantissa range due to implicit bits, which can be more accurate in many cases. Nevertheless, accuracy is not offered by matrix-matrix multiplication. Even FP64 or FP128 can be bad for component-wise relative errors. Take A = [1, -1] and B = [1; 1]. A*B is zero. Let eps be a small perturbation to A and/or B. The solution may now be arbitrarily bad in terms of relative error. In general, assume that the same mantissa range and exponent range is covered as a given higher-precision floating point format, and the accumulation is at least as good as the higher-precision format. With such an assumption, the answer will be approximately the same as the higher-precision floating point format. It may or may not be identical. Performing the same operation in the higher precision format but changing the order of the computations could yield slightly different results. In terms of matrix-matrix multiplication, it could yield vast differences in relative error.

Things get slightly more complicated if low precision is used to approximate matrix-matrix at FP64 accuracy or FP128 precision. Here the scalars aren't just for avoiding denormals but are necessary to do the initial matrix conversion. Nevertheless, converting to an integer is recommended in this case because the FP32-rounded errors in each of the seven or fewer bins may introduce too many errors. An integer is easier to get right because there are no floating-point errors in each bin.

Conversion to Integer functions in the same way as all of the previous Bfloat16 examples. The quantization literature explains how to map floating point numbers into integers. The only difference is that these integers are further broken down into 8-bit pieces for the use of AMX. Constant factors are still needed, but in this case they are primarily defined in the conversion itself.

One difficulty with quantization to integers is the notion of a shared exponent. All the numbers quantized together with shared exponents must share the same range. The assumption is that all of A shares a joint exponent range. Since this will also be true for B, each row of A and column of B can be quantized separately.

Assuming that there is Integer32 accumulation with the Integer8 multiplies, a matrix may be broken down into far more bits than required. This may significantly reduce the inaccuracy impact of picking a shared exponent. Because Integer32 arithmetic will be precise, modulo overflow/underflow concerns, then one can break up A or B into a huge number of 8-bit integer matrices, then do all the matrix-matrix work with AMX, and then convert back all the results to even get accuracies up to quad-precision.

Considering an extreme case of trying to get over 100-bits of accuracy in a matrix-matrix multiply. All A-values can be quantified into 128-bit integers. The same holds true with B. Once broken down into 8-bit quantities, this will have a significant expansion like: A = s1*A1 + s2*A2 + … + s14*A14 for when attempting 112-bits of mantissa. The same can be done with B = t1*B1 + t2*B2 + … + t14*B14. A*B is potentially 14*14=196 products, but only 105 products are needed because the last few products may have scaling factors less than 2^(-112) times the most important terms. Each product term should be added separately and computing into C from the least significant bits forward.

C15 = (s1*t14)*A1*B14 + (s2*t13)*A2*B13 + … + (s14*t1)*A14*B1

C14 = (s1*t13)*A1*B13 + (s2*t12)*A2*B12 + … + (s13*t1)*A13*B1

C13 = (s1*t12)*A1*B12 + (s2*t11)*A2*B11 + … + (s12*t1)*A12*B1

…

C02 = (s1*t1)*A1*B1

Sometimes choosing scalers is possible such that all the products in a given row can be computed with the same scratch array. The converted sum of C02 gives the final product through C15, where terms like C15 should be computed first.

Writing matrix-matrix multiplies in terms of an expansion like (A1+A2+A3)*(B1+B2+B3) is referred to as "cascading GEMM." Performance will vary depending on the TMUL/AMX specification, and may vary from generation to generation. Note that some computations may become bandwidth-bound. Since there is no quad floating-point precision in hardware for Intel Architecture, the above algorithm may be competitive performance-wise with other approaches like doing software double-double optimizations or software-based quad precision.