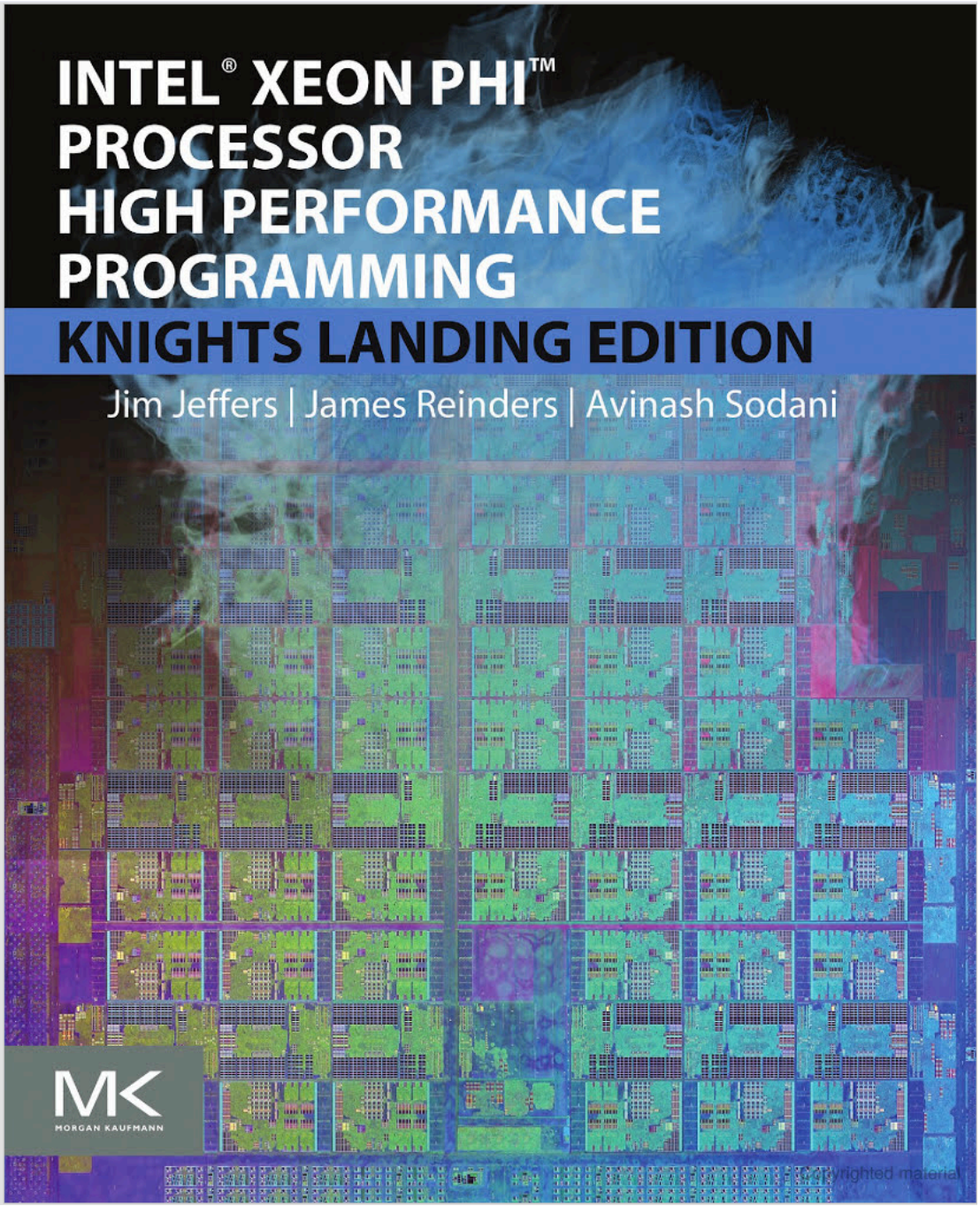
INTEL® XEON PHI™ PROCESSOR HIGH PERFORMANCE PROGRAMMING

KNIGHTS LANDING EDITION

Jim Jeffers | James Reinders | Avinash Sodani



MORGAN KAUFMANN

Contents

SECTION

Knights Landing I

This section focuses on Knights Landing itself, diving into the architecture, the high bandwidth memory, the cluster modes, and the integrated fabric. Like any computer, really understanding it gives us the best application tuning opportunities to utilize it well. This section builds that deep understanding. However, for first-time, many-core programmers, Chapters 2–6 may be an excessive amount of information and could be skipped after reading Chapter 1 by proceeding directly to read Sections II and III of this book. Ultimately, this section contains the Knights Landing details that any expert programmer desiring optimal performance will want to understand.

Chapter 1 describes the motivation for many-core and provides an overview of many-core programming. Chapter 2 gives an introductory overview of the Knights Landing architecture. Chapter 3 discusses application usage of the high bandwidth memory (MCDRAM) and cluster modes. Chapter 4 is an in-depth look at the Knight Landing tile architecture, the mesh interconnect that connects the tiles, and the related memory and cluster modes. Chapter 5 details the Intel® Omni-Path fabric architecture. This section finishes with a chapter dedicated to low-level optimization tips that are most tied to Knights Landing itself. While most of the book focuses on optimizations that are generally universally useful for parallel programming, Chapter 6 (μarch Optimization Advice) provides deeper insight into the workings of Knights Landing micro-architecture and the "do's and don'ts" for performance that expert programmers will enjoy.

This book has three sections: I. Knights Landing, II. Parallel Programming, and III. Pearls. The book also has an extensive Glossary and Index to facilitate jumping around the book.

CHAPTER

1 Introduction

What is new with Knights Landing in this chapter?
Knights Landing is a processor, with a coprocessor option, and uses hyper-threading. Knights Landing also offers high bandwidth memory and fabric support on package. Knights Corner was only available as a coprocessor and used multithreading. Otherwise, the core message of the chapter is the same as for any processor: tune for parallelism.

In this book, we bring together the essentials to high performance programming for an Intel® Xeon Phi™ processor (or coprocessor). The architecture for the Second-Generation Intel Xeon Phi product is commonly known by the Intel code name *Knights Landing*. We will say "Knights Landing" throughout the book instead of the very long phrase "Intel Xeon Phi processor and/or coprocessor (or card)." In general, everything stated about "Knights Landing" applies to both "Intel Xeon Phi *processors*" and "Intel Xeon Phi *coprocessors*." Coprocessor and offload-specific topics are the focus of Chapter 18.

This book is organized into three sections, which cover the architecture specifics, parallel programming models, and real-world examples (pearls), respectively. Together, these sections emphasize essential knowledge to get the most out of Knights Landing.

Programming for Knights Landing is almost entirely about programming in the same way as we would for an Intel® Xeon® processor-based system, but with extra attention on exploiting lots of parallelism because of the uniquely high degree of scaling possible with Knights Landing. This extra attention pays off for processor-based systems as well due to an effect we have dubbed, the "dual-tuning" effect that comes from the compatibility that Knights Landing offers. This "dual-tuning" advantage is a key aspect of why programming for Knights Landing is particularly rewarding and helps protect investments in programming.

Knights Landing is both generally programmable and tailored to tackle highly parallel problems. As such, it is ready to accelerate very demanding parallel applications. We explain how to make sure an application is constructed to take advantage of such a highly parallel capability. As a natural side effect, these techniques generally improve performance on less parallel machines and prepare applications better for computers of the future as well. The overall concept can be thought of as "Portable High-Performance Programming."

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00001-6

INTRODUCTION TO MANY-CORE PROGRAMMING

If you are completely new to many-core programming, we encourage you to read http://lotsofcores.com/sportscar. Sports cars are not designed for a superior experience driving around on slow-moving congested highways. In our introductory tutorial for many-core programming, the similarities between Knights Landing and a sports car give us opportunities to step by step expose you to key techniques for scalable high-performance programming.

The freely downloadable http://lotsofcores.com/sportscar is an adaptation, for Knights Landing, of the first three chapters of our original book on programming for the Intel Xeon Phi coprocessor, titled *Intel Xeon Phi Coprocessor High-Performance Programming*. We had many complements on these chapters as a gentle and readable introduction to many-core programming; these chapters were used in many introductory classes. In the intervening three years, many programmers have learned many-core programming. We offer these three chapters on the web to help introduce more people to many-core programming, with the added bonus that this frees up space in this book for including more in-depth material. The sports car does achieve Knight Landing levels of performance in our updated material.

TREND: MORE PARALLELISM

To squeeze more performance out of new designs, computer designers rely on the strategy of adding more transistors to do multiple things at once. This represents a shift away from relying on higher instruction speeds, which demanded higher power consumption, to a more power-efficient parallel approach. Hardware performance derived from parallel hardware is more disruptive for software designs than speeding up the hardware because it benefits parallel applications to the exclusion of non-parallel programs.

It is interesting to look at a few graphs that quantify the factors behind this trend. Fig. 1.1 shows the end of the "era of higher processor speeds," which gives way to the "era of higher processor parallelism" shown by the trends graphed in Figs. 1.2 and 1.3. This switch is possible because, while both eras required a steady rise in the number of transistors available for a computer design, trends in transistor density continue to follow Moore's law as shown in Fig. 1.4. A continued rise in transistor density, perhaps at a reduced pace, will drive more parallelism in computer designs and result in more performance for applications using parallel programming.

Figs. 1.1–1.4 plot information regarding x86 (and x86-64) processor or coprocessor products made by Intel thus far. The trends of other processor architectures and manufacturers would reveal a similar halt to clock rate increases and a rise in parallelism. Plotting a single long running architecture from a single manufacturer helps us show these trends without other variables at play.

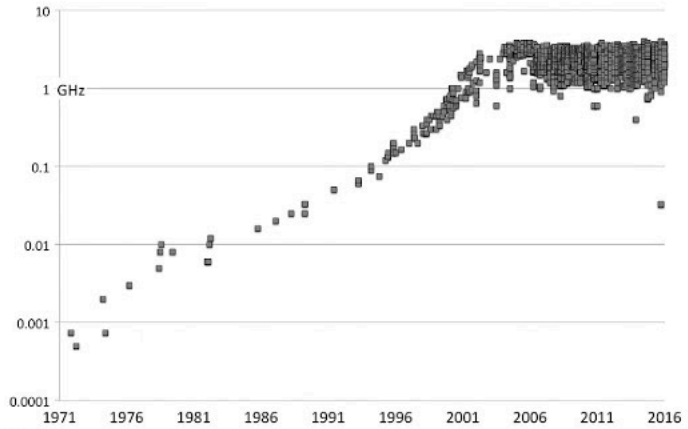


FIG. 1.1

Processor/coprocessor speed era (log scale).

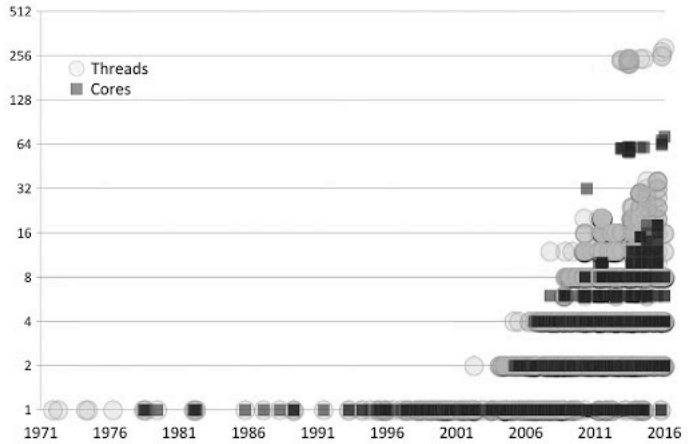


FIG. 1.2

Processor/coprocessor core/thread parallelism (log scale).

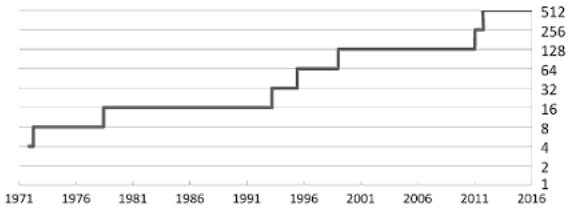


FIG. 1.3

Processor/coprocessor vector parallelism (width in bits) (log scale).

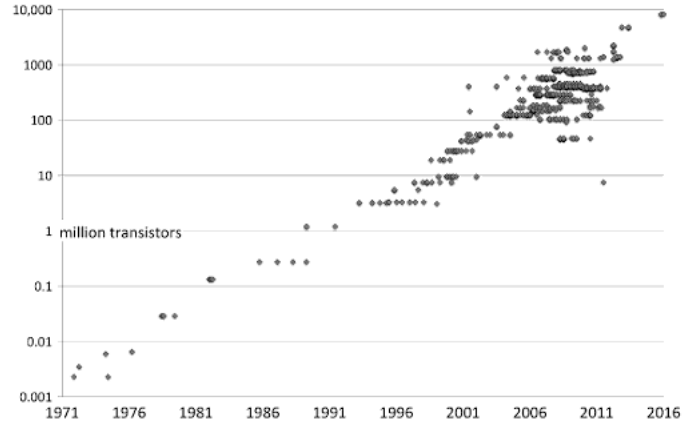


FIG. 1.4

Moore's law continues, processor/coprocessor transistor count (log scale).

WHY INTEL® XEON PHI™ PROCESSORS ARE NEEDED

Knights Landing is designed to extend the reach of applications that have demonstrated the ability to fully utilize the scaling capabilities of Intel Xeon processor-based systems and fully exploit available processor vector capabilities or memory bandwidth. For such applications, Knights Landing offers additional power-efficient scaling, vector support, and local memory bandwidth, while maintaining the programmability and support associated with Intel Xeon processors.

Many applications in the world have not been structured to fully exploit parallelism. This leaves a wealth of capabilities untapped on nearly every computer system.

Such applications can be extended in performance by a highly parallel device only when the application expresses a need for parallelism through parallel programming. This has given rise to calls for "code modernization" to move code forward to exploit the power of newer parallel processors including Knights Landing.

Advice for successful parallel programming can be summarized as "Program with lots of threads that use data organized for vectors with your preferred programming languages and parallelism models." Since most applications have not yet been structured to take advantage of the full magnitude of parallelism available in any processor, understanding how to restructure to expose more parallelism is critically important to enable the best performance for any processor including Knights Landing. This restructuring itself will generally yield benefits on most general-purpose computing systems, a bonus due to the emphasis on common programming languages, models, and tools. We refer to this bonus as the dual-tuning advantage.

It has been said that a single picture can speak a thousand words; for understanding Knights Landing (or any highly parallel device), it is Fig. 1.5 that speaks a thousand words. We used a graph like this in our programming book for Knights Corner, and in our classes to highlight that the Intel Xeon Phi products are designed to deliver unequaled performance on highly parallel code while Intel Xeon processors offer performance across a larger range of applications.

For Knights Corner, a graph like Fig. 1.5 is applied regardless of "vector intensity" (the density of floating point operations — see more in the vectorization chapters). However, assuming full utilization of the floating point math capabilities, Knights

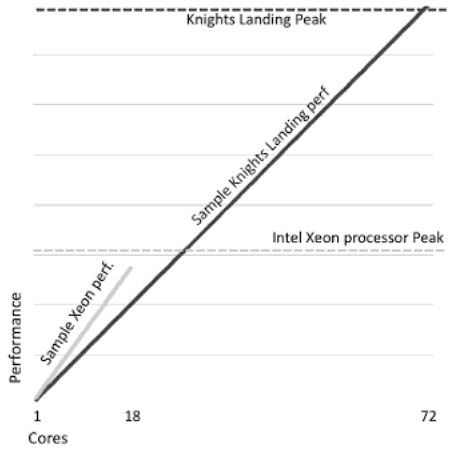


FIG. 1.5

This picture speaks a thousand words.

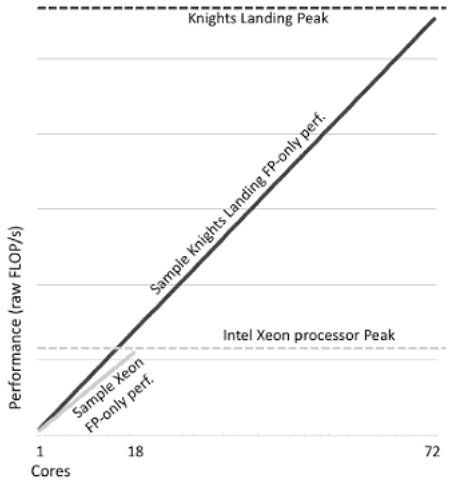


FIG. 1.6

Another thousand words based on Floating-point (FP) arithmetic performance only.

Landing yields Fig. 1.6. This is because, unlike a Knights Corner core, the Knights Landing cores have a higher FLOP/sec/core than contemporary Intel Xeon processors (Haswell architecture) on an FLOP/sec/core basis. Despite lower clock rates, Knights Landing features 512-bit wide vector instructions versus 256-bit wide vector instructions for the Haswell architecture. Knights Landing also has more cores.

Figs. 1.5 and 1.6 speak to this principle: Knights Landing offers the ability to make a system that can potentially offer exceptional performance while still being buildable and power efficient. Intel Xeon processors deliver performance for a broad range of applications but do have a lower limit on peak performance. In exchange for our investments in parallel programming, we may reach otherwise unobtainable performance while remaining portable. The "dual-tuning" double advantage of these Intel products comes from using the same parallelism models, programming languages, and familiar tools to greatly enhance preservation of programming investments.

PROCESSORS VERSUS COPROCESSOR

The original Intel Xeon Phi products, code named *Knights Corner*, were only available as coprocessors. Knights Corner was, in essence, a processor trapped in a coprocessor body. While this provided an excellent entry into the market and helped power the world's faster computer for years, limitations of the coprocessor

model had many people looking forward to an Intel Xeon Phi *processor*. A processor implementation is freed from limitations including limited memory size and PCIe transfers back and forth with a host processor.

The second-generation Intel Xeon Phi product, code named *Knights Landing*, is available as either a processor or a coprocessor. It is accurate to think of it as a processor that can also be packaged as a coprocessor. Most often in this book, we will refer to the first-generation Intel Xeon Phi products by their code name *Knights Corner*, and the second-generation Intel Xeon Phi products by their code name *Knights Landing*. We found this more readable than saying "Intel Xeon Phi (co)processors" or "Intel Xeon Phi processors and coprocessors." In Chapter 18, we address information specific to using Knights Landing as a coprocessor and in offload mode. We do refer to it as a Knight Landing *coprocessor*, in particular, when used in its coprocessor packaging. Likewise, in the few instances where we need to speak about the processor versions specifically, we will refer to it as a Knight Landing *processor*.

MEASURING READINESS FOR HIGHLY PARALLEL EXECUTION

To know if an application is utilizing parallelism well, we should examine how an application scales, uses vectors, and uses memory. We can get some impression of where we are with regard to scaling and vectorization by doing a few simple tests.

To check scaling, we can create a simple graph of performance of various runs each with various numbers of threads (from one up to the number of cores, with attention to thread affinity). This can be done with settings for OpenMP (i.e., OMP_NUM_THREADS for OpenMP) or Intel® Threading Building Blocks (TBB). If the performance graph indicates any significant trailing off of performance, we have tuning opportunities.

To check vectorization, we can compile an application with and without vectorization. For instance, using Intel compilers auto-vectorization: disable vectorization via compiler switches: `-no-vec -no-simd`, use at least `-O2 -xhost` for vectorization. Compare the performance you see. If the performance difference is insufficient, you should examine opportunities to increase vectorization. If you are using libraries, like the Intel Math Kernel Library, you should consider that math routines would remain vectorized no matter how you compile the application itself. Therefore, time spent in the math routines may be considered as vector time. Unless your application is bandwidth limited, the most effective use of Knights Landing will be when most cycles executing are vector instructions doing vector operations. While some may tell you that "most cycles" need to be over 90%, we have found this number to vary widely based on the application. If a program is bandwidth limited, the potential upside of using Knights Landing for its superior bandwidth may dominate instead of vectorization for determining the peak performance.

When using Message Passing Interface (MPI), it is desirable to see a communication versus computation ratio that is not excessively high in terms of communication. The ratio of computation to communication is a key factor in deciding between using offload

versus native model for coprocessor programming. Programs are also most effective using a strategy of overlapping communication and I/O with computation.

There are a number of tools available to assist in profiling and tuning at the system and node levels. These same tools support Knights Landing as well as other processors. Profiling and tuning are covered in detail in Chapter 14, while programming in general is the subject of Section II in this book.

WHAT ABOUT GPUS?

While graphics processing units (GPUs) cannot offer the programmability of Knight Landing, they accelerate some applications through scaling combined with vectorization or bandwidth. Applications that show positive results with GPUs should always benefit from Knights Landing because the same fundamentals of vectorization or bandwidth must be present in an application, although they may need to be expressed differently. The opposite is not true. The flexibility of Knights Landing includes support for applications that cannot run on GPUs. For instance, Knights Landing supports all the features of C, C++, Fortran, OpenMP, TBB, MPI, etc., while GPUs are restricted to special GPU-specific models (like NVidia CUDA) or subsets of standards like OpenMP. This is one reason that a system using Knights Landing will have broader applicability than a system using GPUs and offer greater portability and performance portability. Additionally, tuning for GPU is generally too different from a processor to have the "dual-tuning" benefit we see in programming for Knights Landing. This can lead to a substantial rise in investments to be portable across many machines now and into the future and restrict performance portability in a way that requires GPU-specific retuning generation to generation.

ENJOY THE LACK OF PORTING NEEDED BUT STILL TUNE!

Because Knights Landing is a compatible x86 SMP-on-a-chip, there is no Knights Landing-specific porting needed to run an application. However, the high degree of parallelism in Knights Landing is best suited to applications that are structured to use that parallelism. Almost all applications, which have not been tuned for many-core, will benefit from some tuning beyond the initial base performance to achieve maximum performance on such a highly parallel device. This can range from minor work to major restructuring to expose and exploit parallelism through multiple tasks and use of vectors. The experiences of users of Intel Xeon Phi products and the "forgiving nature" of this approach are generally promising but point out one challenge: the temptation to stop tuning before the best performance is reached. This can be a good thing if the return on investment of further tuning is insufficient and the results are good enough. It can be a bad thing if expectations were that working code would always be high performance. *There ain't no such thing as a free lunch*! The hidden bonus is the "dual-tuning," double advantage of programming investments for Intel

Xeon Phi products that generally applies directly to any general-purpose processor as well. This greatly enhances the preservation of any investment to tune working code by applying to other processors and offering more *forward scaling* to future systems.

TRANSFORMATION FOR PERFORMANCE

There are a number of possible user-level optimizations that have been found effective for ultimate performance. These advanced techniques are not essential. They are possible ways to extract additional performance for your application. The "forgiving nature" of Knights Landing makes transformations optional but should be kept in mind when looking for the highest performance. It is unlikely that peak performance will be achieved without considering some of these optimizations (Section II of this book revisits these in more depth):

- Memory access and loop transformations (e.g., cache blocking, loop unrolling, prefetching, tiling, loop interchange, alignment, affinity).
- Vectorization works best on unit-stride vectors (the data being consumed is contiguous in memory). Data structure transformations can increase the amount of data accessed with unit-strides (such as Array of Structures to Structure of Arrays transformations or recoding to use packed arrays instead of indirect accesses).
- Use of full (not partial) vectors is best, and data transformations to accomplish this should be considered.
- Vectorization is best with properly aligned data.
- Large page considerations (we recommend the widely used Linux libhugetlbfs library).
- Algorithm selection (change) to favor those that are parallelization and vectorization friendly.

HYPER-THREADING VERSUS MULTITHREADING

Knights Landing, like Intel Xeon processors, utilizes hyper-threading. In general, hyper-threading on Knights Landing will prove useful for increasing performance more often than on other prior processors. Knights Corner used a simpler multithreading. Knights Corner always required more than one thread running per core to reach maximal performance. Therefore, Knights Corner reached maximum performance with two, three, or four threads running per core. Knights Landing can reach maximum performance with one, two, or four threads per core.

Programs should parameterize the number of cores and the number of threads per core in order to easily run well on a variety of current and future processors and coprocessors. Therefore, the number of threads per core is usually set outside the application and we tune by checking the performance of using one, two, or four

threads per core on Knights Landing. While three threads per core is possible with Knights Landing, it is unlikely to match the performance of other choices. The architecture behind this is explained in more detail in Chapter 2.

PROGRAMMING MODELS

The most popular parallel programming model for a multimode system is known as MPI+X. This most often means MPI+OpenMP, and for many C++ programmers it means OpenMP+TBB. Section II of this book has chapters discussing all the key programming models in more detail. Programming using Partitioned Global Address Space (PGAS) models is on the rise, and well worth understanding. We discuss PGAS models in Chapter 16. The concept of programming with an "offload" model was popularized by GPUs and the Knights Corner coprocessor. The "offload" model is supported for Knights Landing coprocessors and Knights Landing processors as well. Programming via offloading is discussed in Chapter 18.

No popular programming language was designed for parallelism. In many ways, Fortran has done the best job adding new features, such as `DO CONCURRENT`, to address parallel programming needs. Fortran and C benefits from OpenMP. C++ users have embraced Intel TBB. C++ developers may use OpenMP as well. Section II of this book has chapters discussing the programming models in more detail; we offer some brief advice here.

Knights Landing offers the full capability to use the same tools, programming languages, and programming models as an Intel Xeon processor. However, because Knights Landing is designed for high degrees of parallelism, some models are more interesting than others.

In a way, it is quite simple: an application needs to deal with having lots of tasks, and deal with vector data efficiently (also known as vectorization). There are some recommendations we can make based on what has been working well for programming thus far. For Fortran programmers, use OpenMP, `DO CONCURRENT`, and MPI. For C++ programmers, use TBB, OpenMP, and MPI. TBB is a C++ template library that offers excellent support for task-oriented load balancing. While TBB does not offer vectorization solutions, it does not interfere with any choice of solution for vectorization. TBB is open source and available on a wide variety of platforms supporting most operating systems and processors. For C programmers, use OpenMP and MPI.

WHY WE COULD SKIP TO SECTION II NOW

Parallel programming is the key to Knights Landing. As such, until we have dealt with the critical aspects of parallel programming: scaling, vectorization, and locality; worrying about the rest of Section I in this book could be a distraction.

Of course, learning details about Knights Landing can be fun and very interesting. The rest of Section I (through Chapter 6) dives into just such fun. Chapters 2 and 3 supply an excellent overview for programmers of the keys to Knights Landing architecture and application usage of the on-package high-bandwidth memory, known throughout this book as MultiChannel Dynamic Random Access Memory (MCDRAM), and cluster modes. Chapters 4-6 dive deeply into the Knights Landing design and implications.

We would be remiss if we did not emphasize that the best starting point is to let the MCDRAM act as a cache (memory mode = cache) and leave the cluster mode set to its default (cluster mode = quadrant, in most systems). We would only encourage looking at other modes when fine-tuning an already tuned parallel application. When we have a tuned parallel application, then reading and acting on the rest of Section I will be more important and rewarding.

FOR MORE INFORMATION

We hope you read the Preface to this book. We discuss some history as well as the organization of this book in the Preface.

Some additional reading worth considering includes:

- Introduction to Many-core Programming with our Sport Car driving analogies (adopted from the original Intel Xeon Phi programming book, adapted for Knights Landing). http://lotsofcores.com/sportscar.
- Satish, N., C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. "Can traditional programming bridge the Ninja performance gap for parallel computing applications?," Computer Architecture (ISCA), 2012 39th Annual International Symposium on, vol., no., pp. 440–451, 9–13 June 2012.
- Code modernization resources: http://software.intel.com/moderncode.
- *Structured Parallel Programming: Patterns for Efficient Computation*, Michael McCool, Arch Robinson, James Reinders, Morgan Kaufmann Publishers, 2012, ISBN 9780123914439, http://parallelbook.com.

This page intentionally left blank

CHAPTER

Knights Landing overview

2

In this chapter, we describe the overall architecture of the Second-Generation Intel® Xeon Phi™ product code named *Knights Landing*. We follow with a look at how applications can best use the high-bandwidth MCDRAM and cluster modes in Chapter 3. We dive in-depth into the Knights Landing architecture in Chapter 4, and then the Knights Landing companion Omni-Path fabric architecture in Chapter 5. Programming optimizations specific to the Knights Landing microarchitecture are covered in Chapter 6. Sections II and III (Chapter 7 and beyond) focus entirely on parallel-programming tips, techniques, and examples targeting the extensive parallelism provided by Knights Landing.

What is new with Knights Landing in this chapter?
This entire chapter is about things that are new with Knights Landing.

OVERVIEW

Knights Landing is a many-core processor that delivers massive thread and data parallelism with high memory bandwidth. It is designed to deliver high performance on parallel workloads. Knights Landing provides many innovations and improvements over the prior-generation Intel Xeon Phi coprocessor code named Knights Corner. Knights Landing is a standard Intel Architecture standalone processor that can boot stock operating systems and connect to a network directly via prevalent interconnects such as Infiniband, Ethernet, or Intel® Omni-Path Fabric (Chapter 5). This is a significant advancement over Knights Corner, which is restricted as a PCIe-connected device and, therefore, Knights Corner could only be used when connected to a separate host processor. Knights Landing introduces a more modern, power efficient core that triples (3×) both scalar and vector performance compared with Knights Corner. Knights Landing offers over three TFLOP/s of double precision and six TFLOP/s of single precision peak floating point performance. It introduces a new memory architecture utilizing two types of memory (MCDRAM and DDR) to provide both high memory bandwidth and large memory capacity. It is binary compatible with prior Intel® processors. This means that it can run the same binary executable programs that run on

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00002-8

current x86 or x86-64 processors without requiring any modification to the binary executable programs. Knights Landing introduces the 512-bit Advanced Vector Extensions known as Intel® AVX-512. These new 512-bit vector instructions are architecturally consistent with the previously introduced 256-bit AVX and AVX2 vector instructions. The AVX-512 instructions will be supported in future Intel® Xeon® processors as well. Knights Landing introduces optional on-package integration of the Intel® Omni-Path Architecture enabling direct network connection with improved efficiency compared to an external fabric chip or card.

INSTRUCTION SET

The instruction set architecture (ISA) for Knights Landing is similar to any other x86 processor; it layers recently added new instruction groups on top of support for baseline x86/x86-64 instructions equivalent to recent Intel Xeon processors. Fig. 2.1 shows how the Knights Landing ISA compares with recent Intel Xeon processors. Knights Landing supports all legacy instructions including 128-bit SSE and SSE4.2, and 256-bit AVX and AVX2 technologies.

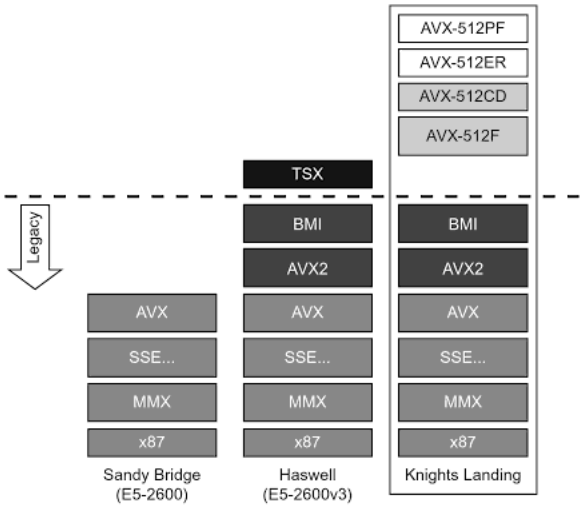


FIG. 2.1

Knights Landing ISA as compared to Intel Xeon processors.

Knights Landing introduces Intel® AVX-512 instructions. AVX-512 provides 512-bit SIMD support, 32 logical registers, 8 new mask registers for vector predication, and gather and scatter instructions to support loading and storing sparse data. Each AVX-512 instruction can perform eight double-precision multiply-add operations (16 FLOPs) or sixteen single-precision multiply-add operations (32 FLOPs), with an optional operand specifying a memory location that can either be a scalar or a vector.

Intel AVX-512 instructions consist of four categories. AVX-512 Foundation instructions (AVX-512 F) are the base 512-bit extensions as described previously. Three other categories are: Intel® AVX-512 Conflict Detection Instructions (CDI), Intel® AVX-512 Exponential and Reciprocal Instructions (ERI), and Intel® AVX-512 Prefetch Instructions (PFI). These capabilities provide efficient conflict detection for improved vectorization (e.g., for histogram updates), fast exponential and reciprocal operations for speeding up transcendental functions, and prefetch capabilities for sparse data, respectively.

AVX-512 has been defined with programmability in mind. Most AVX-512 programming occurs in high-level languages, such as C/C++ and Fortran, through vectorizing compilers and pragmas to guide the compilers or via libraries with optimized instruction sequences, which may use intrinsics.

Knights Landing does not implement Intel® Transactional Synchronization Extensions (TSX). As with other extensions, software is expected to confirm hardware support using the applicable CPUID feature bit when needed to ensure the ability to run on machines with and without these extensions.

ARCHITECTURE OVERVIEW

Fig. 2.2 shows a block diagram of the Knights Landing processor. The Knights Landing design has a concept of a tile, which is the basic unit for replication. Each tile, as shown in Fig. 2.3, consists of two cores, two vector-processing units (VPUs) per core, and a 1 M L2 cache shared between the two cores. These tiles are physically replicated 38 times on Knights Landing, however at most 36 of them are active; the extra two tiles are present to improve manufacturing chip yield and are disabled by the time the processor leaves the factory. Disabled cores consume essentially no power. In total, with 36 tiles, Knights Landing has 72 cores and 144 VPUs on the chip. Versions of Knights Landing, with fewer active tiles or less memory may be manufactured as well.

TILE

The core is a newly designed two-wide, out-of-order core derived from the Intel® Atom processor code named Silvermont. Knights Landing includes significant modifications to the Silvermont microarchitecture to incorporate many features targeting high-performance computing. These features include support for four hyperthreads

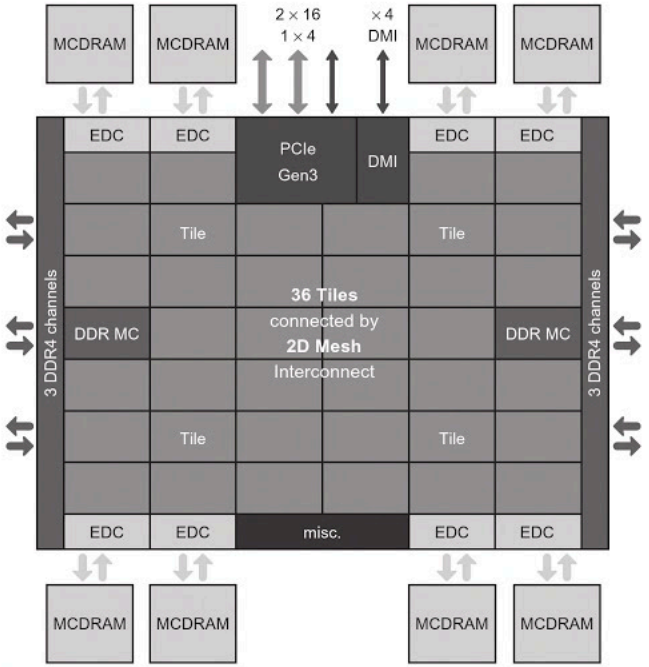


FIG. 2.2

Block diagram showing overview of Knights Landing Architecture.

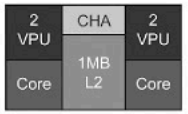


FIG. 2.3

Block diagram of a tile.

per core, deeper out-of-order buffers, higher L1 and L2 cache bandwidths, adding AVX-512 vector instructions, improved reliability features, larger TLBs, and a larger L1 cache. The new core supports all legacy x86 and x86-64 instructions, making Knights Landing completely binary compatible with prior Intel® Architecture processors.

MESH: ON-DIE INTERCONNECT

The cores within a single tile have access to their local L2 cache, and their local processing capabilities, including their local VPUs, all within the single tile. However, in order to access data stored in other L2 caches, memory (MCDRAM or DDR), or perform I/O (PCIe), it is necessary to communicate with other parts of the chip via the on-die interconnect. The tiles are interconnected by a cache-coherent, two-dimensional (2D) *mesh* interconnect. Given its 2D nature, the mesh interconnect provides a more scalable way to connect the tiles by providing higher bandwidth and lower latency compared to the 1D ring interconnect on Knights Corner. The mesh interconnect employs an MESIF cache-coherent protocol — where a cache line can be in (M)odified, (E)xclusive, (S)hared, (I)nvalid, or (F)orward state — to keep the L2 caches in all tiles coherent with each other. The lines present in tile L1 and L2 caches are tracked using a distributed tag directory structure. Each tile holds a portion of this distributed tag directory structure in the box (i.e., hardware design unit) called CHA (caching/home agent). The CHA also serves as the point where the tile connects with the mesh.

The mesh on-die interconnect architecture is based on a ring architecture that has been widely utilized in the current Intel Xeon processor family. The mesh features four parallel networks, each for delivering different types of packets. The mesh has been optimized for the Knights Landing traffic flows and protocols and is capable of delivering greater than 700 GB/s of total aggregate bandwidth.

The mesh is organized into rows and columns of "half" rings that fold upon themselves at the endpoints (e.g., that means that traffic sent off the edge from an edge tile is connected as the input on the same edge of the same tile; this is *not* a torus configuration which would require long connections across the chip resulting in inconsistent delays tile to tile). The mesh enforces a "*YX* routing" rule. This means that a transaction always travels vertically first until it hits the target row, makes a turn, and then travels horizontally until it reaches its destination. Messages arbitrate with the existing traffic on the mesh at injection points as well as when making a turn. The existing traffic on the mesh takes higher priority. The static *YX* routing simplifies the protocol by helping reduce deadlock cases. A single hop on the mesh takes two clocks in the *X*-direction and only one clock in the *Y*-direction.

Cluster modes

The mesh supports three modes of clustered operation, which provide different levels of address affinity to improve overall performance. These are (i) all-to-all mode, (ii) quadrant mode, and (iii) sub-NUMA clustering (SNC) mode. For most systems, quadrant mode will be the default. In some rare configurations, such as when memory capacity is not uniform across all channels, all-to-all mode will be required. Switching from quadrant mode to an SNC mode may boost performance for some applications, generally those using more than one MPI rank on a single Knights Landing. These modes, and the programming implications, are discussed in Chapter 3 with a deeper look in Chapter 4.

MCDRAM (HIGH-BANDWIDTH MEMORY) AND DDR (DDR4)

Knights Landing processor has two types of memory: MCDRAM and DDR. These two types of memory together provide both high bandwidth and large capacity required to support bandwidth hungry applications as a standalone, bootable processor.

MCDRAM is the high-bandwidth memory integrated on-package. There are eight MCDRAM devices integrated, each is of 2 GB capacity, for a total of 16 GB of high-bandwidth memory. These devices are connected to their own memory controller (EDC — named such for historical reasons) via a proprietary on-package I/O (OPIO). Each device has a separate read and write bus connecting it to its EDC. The aggregate Streams Triad bandwidth from all the eight MCDRAMs is over 450 GB/s.

The memory can be configured at boot time in one of the three modes: (i) cache mode — where MCDRAM is a cache for DDR, (ii) flat mode — where MCDRAM is treated like standard memory in the same address space as DDR, and (iii) hybrid mode — where a portion of MCDRAM is cache and the remaining is flat.

The high-bandwidth memory, MCDRAM, is explained in more detail in Chapter 3 along with details on programming including the different use case implications for applications. Much more about the architecture of MCDRAM support and memory modes comes in Chapter 4 including a detailed explanation of its interactions with cluster modes.

DDR offers high-capacity memory that is external to the Knights Landing package. There are two DDR4 memory controllers, on opposite sides of the chip, each controlling three channels. Each channel supports one DIMM per channel for speeds up to 2400 MHz. The total memory capacity will depend on the capacities of the DIMM used, but the maximum total capacity is 384 GB, assuming with 64 GB DIMMs per channel. The aggregate Streams Triad bandwidth from all six DDR4 channels is around 90 GB/s.

I/O (PCIE GEN3)

Knights Landing supports a total of 36 lanes of PCIe Gen3 for I/O, divided into two x16 and one x4 lanes. These lanes are PCIe Root Ports serving as masters on PCIe bus, as opposed to End Points; these are the same as PCIe Root Ports in any other self-hosted processor. The PCIe controller is housed in a box called integrated I/O (IIO) in Fig. 2.2. The IIO also contains four lanes of proprietary direct media interface that is used to connect to the server chipset that provides the platform level legacy functions required to boot a standard operating system.

Knights Landing also integrates the Intel® Omni-Path Fabric on-package in some product versions. The fabric is connected via the 2 x16 lanes of PCIe to the Knights Landing die (leaving x4 lanes for external devices) and provides two 100 Gb/s ports out of the package as shown in Fig. 2.4. The architecture of Intel Omni-Path is the subject of Chapter 5.

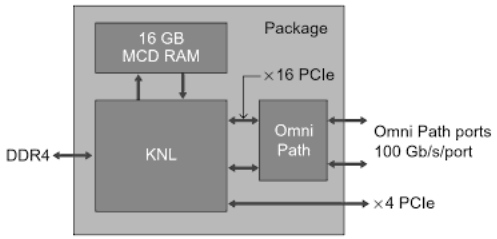


FIG. 2.4

Knights Landing block diagram with Intel Omni-Path Fabric.

MOTIVATION: OUR VISION AND PURPOSE

Having provided an overview, we will share our vision and purpose behind some of the important architectural choices that resulted in the Knights Landing architecture just discussed. Knight Landing started with our desire to remove the overhead of offloading computation from the host processor to a co-processor card connected across the PCIe bus as is present in the first generation (Knights Corner) of Intel Xeon Phi products. The latency and bandwidth of transferring data to and from the coprocessor through a software driver layer, and over a PCIe bus, is a considerable bottleneck. Any performance advantage from faster execution on the card is limited by such transfer overhead for any PCIe compute device. To eliminate transfer overhead, we decided to make Knights Landing a standalone *processor* that can boot off-the-shelf operating systems thereby connecting to the network directly instead of having additional data hops across the PCIe bus. This way, data is not transferred anywhere else for computation; data can stay in main memory while being operated on by the Knights Landing cores. It is no different than any standard processor in that regard; there are many additional benefits to being a host processor.

Once we decided that Knights Landing would be a standalone processor, several other decisions followed from there including the choice of core design. As a standalone processor, Knights Landing would run the entire application, not just its parallel portions. This meant that it needed to be capable of running even single threaded, non-parallel, parts of the application well, while still being significantly more power efficient when executing the parallel portions. This motivated the decision to create a new Knights Landing core that significantly improves single thread performance over the fully parallel-focused Knights Corner — by over 3× — while maintaining power efficiency when running parallel and vectorized code.

Our second big decision was about the ISA. With Knights Landing as a standalone processor, we needed to maximize support for all types of software including standard operating systems, debuggers, infrastructure management, tools, and legacy libraries that would run on it. Allowing software to run without requiring

recompilation removes a major overhead for usage. This motivated us to make Knights Landing binary compatible with the mainline Intel ISA, enabling all legacy code that run on prior Intel processors to run on Knights Landing as well, without modification and without requiring recompilation.

Our third major decision was on the memory architecture. The memory architecture in Knights Landing was designed to support its large compute capability and its standalone processor status. Being a standalone processor, Knights Landing would run the operating system and all parts of the software stack. This called for a capability to support large memory capacity. Users have requirements for memory capacity numbers such as 2-4 GB per core, and in some cases even higher. To provide enough memory capacity for standard software to run on Knights Landing, we decided to provide DDR memory channels. While DDR memory would provide the required capacity, they could not provide the needed bandwidth to feed the massive compute capability of Knights Landing. For this purpose, we incorporated up to 16 GB of high-bandwidth memory, called MCDRAM, in the Knights Landing package. The MCDRAM can provide over 450 GB/s of bandwidth directly to Knights Landing. Together, the two types of memory provide both large capacity and high bandwidth needed for a high-performance many-core Knights Landing processor.

Finally, we had to decide on the on-die fabric to interconnect all the tiles and other components on the chip. The 2D mesh interconnect was chosen for this purpose. It was designed to provide an on-die network that can easily move in excess of 450 GB/s of bandwidth delivered from the memory across the chip. It also provided lower latency connections, because there are fewer hops required to go from one point to another on the chip. A 1D ring interconnect, as used in Knights Corner, would not have scaled efficiently to the levels of bandwidth provisioned in Knights Landing.

PERFORMANCE

One of the objectives of Knights Landing was to provide a significant boost in performance (and performance per watt) for applications that are optimized to exploit its high compute and bandwidth capabilities, while still providing reasonable good performance for unoptimized code that are run "out-of-box." As users optimize their code on Knights Landing processor, we did not want them to start from a point of significant performance deficit and then have to work hard to merely overcome that deficit before realizing any performance upside from the new processor. Instead, we wanted applications to at least realize "at-par" performance for the unoptimized code on Knights Landing immediately and then reap much higher performance benefits for their code optimization efforts. Fig. 2.5 demonstrates this point with some early performance and performance per watt numbers obtained on pre-production versions of Knights Landing (A0 stepping, the first manufactured parts). The graph shows the performance and relative performance/watt of Knights Landing processor relative to single socket of an Intel Xeon E5-2697-v3 (code named *Haswell*) processor. The workloads are shown on *x*-axis. They range from benchmarks, such as Linpack

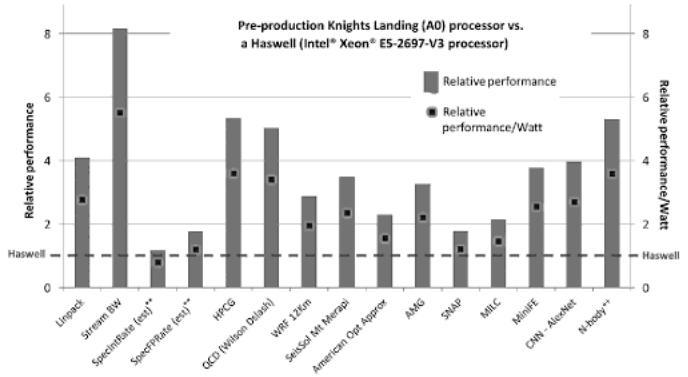


FIG. 2.5

Early performance measurements Knights Landing versus Haswell. Measurements on a pre-production Knights Landing (A0) processor. Results subject to change on production parts.

**Early A0 processor runs, estimated SPEC (not fully compliant nor submitted runs).

++Code and measurement done by colfaxresearch.com.

Data courtesy of Intel Corporation.

and Streams, to various scientific and engineering applications, to general "out-of-box" throughput benchmarks, such as SpecInt Rate and SpecFP Rate. As the early results show, Knights Landing provides at-par performance and performance per watt versus one socket of Haswell on unoptimized, "out-of-box," SpecInt and SpecFP Rate benchmarks (approximately 1.0× to 1.8× on performance and approximately 0.9× to 1.2× on performance per watt) while providing significant improvements (approximately 2× to 8×) on other workloads that have varying degrees of optimizations whether compute bound or memory bound.

SUMMARY

In this chapter, we provided an architectural overview of Knights Landing, the second generation of Intel Xeon Phi products. A defining design choice for Knights Landing is that it is architected to be a standalone bootable processor that runs off the shelf operating systems, middleware and applications, putting it on-par with other Intel Architecture family processors. As a highly parallel oriented processor, Knights Landing enables the growing application base of modernized parallel codes to take advantage of its new high-performance features to reach new performance levels to more rapidly advance scientific discovery, big data analytics, machine learning, and

other high-performance needs. The next few chapters will delve more deeply into these high-performance features and explain their benefits and uses in detail. At this point, if your interest is primarily in parallel application development on Knights Landing, you may want to skip right to Section II now, or after reading Chapter 3, and return to Section I for later reference.

FOR MORE INFORMATION

- *Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals*, Intel Corporation, June 2015, https://software.intel.com/isa-extensions.
- *Knights Landing: 2nd Generation Intel® Xeon Phi™ Product*, IEEE Micro Magazine, Hot Chips Special Issue, Volume 36, Issue 2, March–April 2016.
- *Intel® Architecture Instruction Set Extensions Programming Reference*, Intel Corporation, August 2015, https://software.intel.com/isa-extensions.

CHAPTER

Programming MCDRAM and Cluster modes 3

> **What is new with Knights Landing in this chapter?**
>
> This entire chapter is about application usage of MCDRAM and cluster modes, which are new with Knights Landing.

Knights Landing offers an unprecedented variety of configurations which have traditionally been available only as hardwired and unchangeable design decisions. Specifically, the choices realized by the cluster modes and the memory modes. This wide ranging support allows Knights Landing to act like very different machines based on the configuration used to initialize Knights Landing, the operating system, and then the applications.

Fortunately, these different modes match to different programming styles that are already popular. In that sense, Knights Landing can take on a personality to match an application instead of the other way around. Of course, applications can be modified to take advantage of this variety as well. We will describe these features from both vantage points.

Memory types (MCDRAM and DDR), memory modes (cache, flat, hybrid), and cluster modes (All-to-All, Quadrant, Hemisphere, SNC-2, and SNC-4) were briefly introduced in Chapter 2. In this chapter, we cover the essentials of programming to utilize the high-bandwidth memory known as the MCDRAM and to utilize cluster modes. Chapter 4 dives much deeper into the hardware architecture; this chapter is all about the programming interfaces and implications for applications.

The usage models available are dependent upon which memory and cluster modes were set at boot time. While there is nothing an application can do directly to control which memory mode or which cluster mode is selected (aside from changing a BIOS setting and rebooting the system), there are numerous usage choices remaining to discuss.

MPI+X (e.g., MPI+OpenMP) applications may run faster with *cluster mode=SNC-4* instead of the default *cluster mode=quadrant*. Cache friendly applications will benefit from the default of *memory mode=cache*. Beating the performance of cache mode is not easy for most applications. However, an important class of applications is known to be not cache-friendly, and those will likely benefit the most from other memory modes. Because most applications have found ways to be relatively cache friendly, we could guess that most applications will run well enough with the defaults of *cluster mode=quadrant* and *memory mode=cache*, and we would probably be right. However, those are "fighting words" to some. Such

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00003-X

questions are certainly open to debate by those intent on squeezing every last bit of performance out of Knights Landing. Fortunately, the debate can continue because these decisions were not hardwired into the design.

USE CACHE MODE AND DEFAULT CLUSTER MODE (AT FIRST)

Before we explain memory modes and cluster modes, we have a simple point: initially most applications can gain more from fine tuning for parallelism than from fiddling with memory and cluster modes. The default cluster mode of quadrant and default memory mode of cache will likely work well for most applications. We emphasize this because covering MCDRAM programming so early in the book as Chapter 3, in Section I of this book, was done to group it with things new or unique in Knights Landing. Being so early in the book is not a statement about it being more important than tuning for general parallel programming, which is covered in Section II of this book.

Now that we have that out of the way, we do want to introduce and explain memory modes and cluster modes from a programmer's point of view. Chapters 4 and 6 supply additional information for those inclined to understand the underlying hardware in greater detail. This chapter focuses on the key interfaces for programmers to build a firm foundation for using memory and cluster modes.

PROGRAMMING FOR CLUSTER MODES

We are going to say very few words about cluster modes in this chapter because they are fairly simple to explain. It is very important to know that, regardless of the cluster mode selected, all of memory, DDR and MCDRAM, is always available to every core. Also, all memory is fully cache coherent. This means that every core has the same view of what data is in every location. What differs between the modes is whether the view of a particular memory (MCDRAM or DDR) from the cores is UMA (Uniform Memory Access) or NUMA (Non-Uniform Memory Access). In any case, even if MCDRAM and DDR are each used as UMA memory, the combination is NUMA.

There are five cluster modes, which are often thought of as three modes with two variations: all-to-all, quadrant (variation: hemisphere), and SNC-4 (variation: SNC-2).

The default cluster mode is quadrant, unless you have a machine where the DDR is not evenly populated. If the DDR DIMMs connected to the Knights Landing are not equal in capacity, then the only cluster mode available is the all-to-all mode. When the DDR DIMMs are equally populated, we will likely choose to not run in all-to-all mode. The cluster modes, other than all-to-all, optimize internal tables within Knights Landing assuming evenly distributed DDR DIMMs. Programs written for all-to-all, quadrant, and hemisphere are unlikely to be different. Most programmers will simply consider these modes to be the same from an application standpoint.

Therefore, for programming—the key issues arise from all-to-all/quadrant/hemisphere versus SNC-4/SNC-2 cluster modes. The key difference is whether each memory type (MCDRAM and DDR) is UMA or NUMA.

In quadrant mode, each memory type is UMA meaning that the latency from any given core to any memory location within the same memory type (MCDRAM or DDR) is essentially the same. In a strictly technical sense, the actual latencies will vary a little across the mesh. However, we cannot distinguish between memory regions by different latencies. While nothing is completely uniform in a strictly technical sense, we program to it as UMA because we cannot deterministically change the effective latency solely through the choice of memory locations.

In SNC-4, each memory type is NUMA meaning that the cores and memory are divided into (four) quadrants with lower latency for "near" memory accesses (within the same quadrant) and higher latency for "far" (within a different quadrant) memory accesses. SNC-4 is well suited for MPI applications that utilize four, or a multiple of four, ranks per Knights Landing. If an application is not NUMA aware, and is not divided into *multiple* MPI ranks, then quadrant mode is almost certainly the right choice. For instance, an application written with one MPI rank per node, which uses OpenMP for all the cores on the node, may not be the best fit for SNC. Such a case is at odds with the purpose of SNC, to isolate and localize traffic to distinct lower latency memory partitions.

The other two cluster modes, that are often called out as variations on *quadrant* and *SNC-4*, are *hemisphere* and *SNC-2*. These two modes are identical to quadrant and SNC-4, except they logically divide the cores and memory (by types) into halves instead of quarters. Compared with their corresponding four-way modes, these two-way modes have higher latency that leads to reduced bandwidth as a result. They may be a better fit for the way a particular application is structured, or if the number of active tiles in Knights Landing does not divide by four. For example, a 68-core Knights Landing has 34 tiles (two cores per tile) that divide into halves as 34 and 34 cores, but into quarters as 16, 16, 18, and 18 cores, because tiles do not divide. In such configurations, SNC-2 may be a better choice for a balanced MPI program running multiple ranks per Knights Landing.

PROGRAMMING FOR MEMORY MODES

Knights Landing features high-bandwidth on-package memory known as MCDRAM. This on-package memory is not faster for a single data access than main memory (DDR), but it can support much high bandwidth (more simultaneous data accesses) than main memory. Therefore, when used as a cache, or used directly to store the data most used within an application, substantial speed-ups are possible.

All of the MCDRAM can be used as program allocatable memory in what is known as *flat mode* on Knights Landing. In flat mode, the entirety of DDR space and MCDRAM space is visible to the operating system and applications. It is exposed as separate NUMA nodes.

Flat mode is always available to choose, and it is the only option available on the Knights Landing coprocessor (see Chapter 18) since Knights Landing coprocessors have only MCDRAM and no DDR support. Most often a system, using Knights Landing processors, also has a main memory consisting of DDR attached to a Knights

Landing in addition to the MCDRAM on-package. In cases where there is both MCDRAM *and* DDR memories, all or some of the MCDRAM (25%, 50%, or 100%) can be used as a cache for data in the DDR. In this case, some or all of the MCDRAM acts as a memory-side cache. When 100% of the MCDRAM is used as cache, we call this *cache mode*. In the case where only 25% or 50% of the MCDRAM is used as a cache, we call this *hybrid mode* because the remaining MCDRAM is available as memory as in flat mode but smaller in size. Summaries of these modes are shown in Fig. 3.1. These modes are set at boot time based upon the current BIOS settings. Changes to these BIOS settings, depending on the BIOS used, may be possible through interactive menus at boot time, or tools after boot time, or both.

While memory modes and cluster modes are largely orthogonal, concepts of "near" and "far" come into play. The cluster modes called SNC-4 and SNC-2 can be used to effectively subdivide Knight Landing into smaller grouping of cores (four parts, or two parts, respectively. They also divide the MCDRAM and DDR to align with the grouping of cores. In order to get the full value of these SNC modes, applications should use "near" memory (MCDRAM or DDR) instead of using all of the memories ("near" and "far"). When we focus on "near" only, the size of MCDRAM and DDR can be half or a quarter of the otherwise expected "flat" or "hybrid" memory sizes. This division of MCDRAM is discussed in depth in Chapter 4 in a section titled *Interactions of Cluster and Memory Modes*. Our programming advice is simple: if we choose to utilize MCDRAM directly, we should parameterize our applications to be able to deal with differing sizes of MCDRAM, and avoid naming NUMA nodes by number explicitly (see a later section *How to Not Hard Code the NUMA Node Numbers*).

Memory (MCDRAM and DDR) do not change sizes under SNC-4 and SNC-4 cluster modes versus the other cluster modes. However, the benefit of using SNC cluster modes depends in part on focusing data usage to be in "near" memories. The SNC cluster modes have "near" memory because they affinitize a portion of each memory type to the quarter (SNC-4) or half (SNC-2) of the Knight Landing tiles.

Memory Mode	MCDRAM (HBM)	DDR (DDR4)
Cache	100% caches (caches DDR data)	Memory (*DDR is required in this mode*)
Flat	All used as memory (0% as cache)	Memory (*if present*)
Hybrid	Partitioned as cache (25% or 50%) and memory (75% or 50%)	Memory (*DDR is required in this mode*)

FIG. 3.1

Summary of memory modes.

MEMORY USAGE MODELS

Applications generally will do nothing special when running in cache mode. Of course, applications which maintain a working set to match the MCDRAM size (16 GB) or otherwise block their data usage to maximize reuse knowing the MCDRAM size will benefit from better caching in a manner similar to what you would expect from any cache. In practice, for high-performance applications the cache mode behavior is generally quite good and offers the majority of performance boost available from MCDRAM. Nevertheless, for applications with larger data sets, the non-cache memory modes do offer some performance upside when used. Applications with "cache unfriendly" data are candidates for using memory modes other than cache.

When using some or all of the MCDRAM as memory (flat or hybrid mode), we have summarized the options in Fig. 3.2.

Option	Summary
Do nothing	A program that is well blocked into L2, may be happy with DDR alone. Some experiments to verify may be wise. Cache-friendly applications may still benefit from being in cache mode. Programs which do poorly with caches could conceivably benefit from putting MCDRAM in flat mode and leaving unused.
Cache mode	Trivial to try; no source code changes. Knight Landing can dedicate 100%, 50% or 25% of the MCDRAM as a memory-side cache.
Remaining options are only useful *flat* and *hybrid* memory modes to access the MCDRAM as memory.	
numactl (NUMA Control utility)	No code changes: use numactl to have all data allocations (including the data segments and stack) come from MCDRAM. This works regardless of programming language.
autohbw (comes with memkind package)	No code changes: use the autohbw library (part of the memkind project) to have allocations, of a certain size range, come from MCDRAM. This definitely works for Fortran, C and C++ because their allocation routines ultimately utilize allocation routines associated the standard C library, e.g. glibc. This can work for all programming languages to the extent that their allocations routines also rest on top of the standard C library routines; autohbw could be extended to intercept most any allocation routine. Like memkind, data segments and stacks are not allocated from MCDRAM.
memkind (includes hbw_ routines)	Change code: use the memkind library, or the FASTMEM directives in Intel Fortran, to have specific allocations be mapped to MCDRAM. Applications in any language can find ways to use these explicit APIs. In a later Section, *Approaches to Determining What to Put in MCDRAM*, we discuss some approaches to figuring out what to allocate using memkind. These approaches could guide autohbw usage as well.

FIG. 3.2

Options for using MCDRAM, or *not*.

WHAT IS THE `MEMKIND` LIBRARY (AND `HBWMALLOC`)?

Because we are going to start mentioning the `memkind` library, we need to explain it now. The `memkind` library is a user extensible heap manager, built on top of `jemalloc`, which enables control of memory characteristics and a partitioning of the heap between kinds of memory. This combination of `jemalloc and memkind` starts to address a long-standing software need; there has simply been no good solution for managing multiple kinds of memory (e.g., DRAM, MMIO, RDMA slabs, symmetric heaps) in an operating system and runtime agnostic way. The advocates for the memkind project believe they have helped move us all forward.

The memkind library delivers two interfaces defined in the header files hbwmalloc.h and memkind.h. The developers themselves describe hbwmalloc as the more stable of the two and the API recommended for high-bandwidth memory (MCDRAM) use cases. They describe memkind as more of a work in progress to develop a very generic API for a broad range of memory types. The hbwmalloc API is built on top of memkind as a convenience for users of high-bandwidth memories.

> The `hbwmalloc` APIs are specifically for high-bandwidth memory; the `memkind` APIs are a more generic interface that can support additional memory types in the future, as well as explicit DDR allocations.

The advantage of the `hbwmalloc` interfaces is that they minimize the code changes needed—just add a prefix of "`hbw_`" for popular heap allocation library calls (i.e.,`malloc(), calloc(), realloc(), posix_memalign(), and free()`). The advantage of the `memkind` interfaces is that they are more flexible for additional memory types. The memkind library is also extensible to support user-defined memory types as well.

Linux backs virtual addresses with 4 KB physical pages by default, though some distributions now have enabled transparent huge pages that can give you automatic 2M page allocation. The memkind library supports explicit allocation of large pages by allowing an application to select which data structures should be allocated with large pages; see the function `hbw_posix_memalign_psize` for more details.

Both interfaces should be considered part of a `memkind` "convenience library." All this functionality is built upon existing memory allocation APIs, albeit with a little more work. Since `memkind` is open-source, it is also possible to customize as needed, or simply adopt the techniques for use in custom memory allocators.

MAXIMIZING PERFORMANCE WITH MEMORY USAGE MODELS

We wrote a very simple application to repeatedly read and write memory, from 64 cores in parallel, and we call it *Hello MCDRAM*. The key routine is shown in Fig. 3.3; the full source code is available from our website—see *For More Information* at the

```
#define CACHE 16
void exercisememory(int *start,
                    int *endplus1,
                    unsigned count) {
  unsigned i;
  int s, *a0=start, *a;
  int mythreadid = omp_get_thread_num();
  for (a=a0;a<(endplus1+CACHE*4);*a++=0);
  a=a0;
  for (i=0;i<count*3;i++) {
    s = *a;
    s -= ++*(a+=CACHE);
    s += --*(a+=CACHE);
    s -= ++*(a+=CACHE);
    if (a >= (endplus1-CACHE*4)) a = a0;
    /* dear compiler: we needed to compute s */
    start[0] += s;
  }
}
```

FIG. 3.3

The key to Hello MCDRAM—the exercising of a consistent number of memory locations (count is always the same), but varying run-to-run the range of memory touched (working set—determined by endplus1-start).

end of this chapter. No single application can possibly be representative of every workload, and our *Hello MCDRAM* is no exception. Our toy application is very deliberate at focusing on a single chunk of memory to exercise and it is a short running application. Nevertheless, it helps illustrate the challenges that we face if we try to do better through completely explicit usage of MCDRAM versus a number of other options available to us without modifying our application.

Our *Hello MCDRAM* application always does the same number of memory accesses in any run of the application, but it varies how large the section of memory is that it exercises (ranging from only 4 bytes, to 4 MB of memory, touched per core). In our graphs, we show how many bytes per core are exercised by the application. We then compare *memory mode = cache* with four ways to utilize MCDRAM in flat mode by recording the amount of time for the application to complete. Fig. 3.3 shows very clearly that anything is better than letting the MCDRAM go unused. The performance of our toy application is over 5 × worse for larger data sets than any other option. Cache mode sets a pretty high bar for performance. Measuring performance in "flat mode," but not using MCDRAM at all as we have effectively done in Fig. 3.3, helps give a perspective of the value of MCDRAM. If use of cache mode had not lifted performance significantly, we could conclude that prescriptive uses of MCDRAM (such as autohbw or memkind/FASTMEM) may be able to perform better. In pursuit of performance, MCDRAM should not be left unused. Any application that makes a choice to manage the MCDRAM explicitly is competing with the other usage models including the cache mode.

Cache mode sets a high bar in terms of performance. The true baseline is not using MCDRAM at all; measuring performance in "flat mode" but not using MCDRAM at all gives the true baseline. In pursuit of performance, MCDRAM should not be left unused. If use of cache mode did not lift performance significantly, that may signal opportunities for more prescriptive uses of MCDRAM. Most often, any application which makes a choice to manage the MCDRAM explicitly faces tough competition from the cache mode to reach better performance. Fig. 3.3 shows very clearly that bandwidth hungry applications should not let the MCDRAM go unused.

The effects of 16K L1 caches per core, and 1M L2 caches shared between pairs of cores (giving ~512K/core effectively for our application), are evident in Fig. 3.4. After the working sets exceed the L2 caches, the effects of the MCDRAM govern all the options. The "unused flat" line helps us see that the effects of MCDRAM do indeed

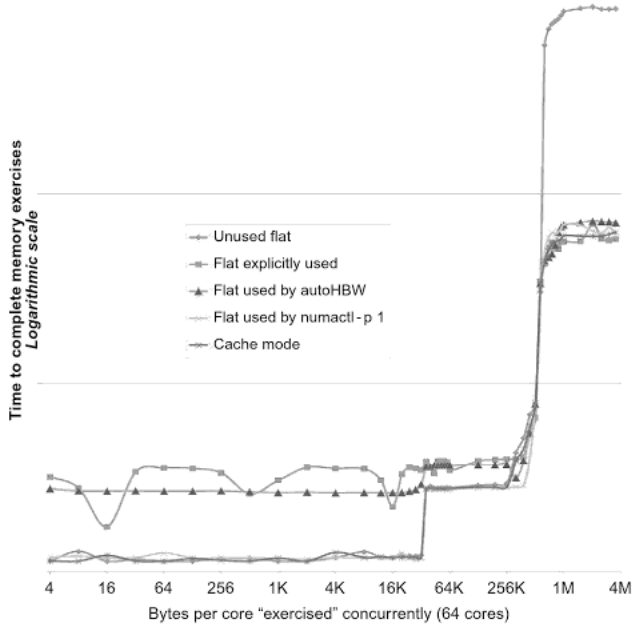


FIG. 3.4

The worst way to use MCDRAM: ignore it (see line marked *unused flat*). The best way to use MCDRAM: we have four choices to match a particular application needs. Three of them need no changes to an application.

consistently and profoundly affect performance for the better, note that the graph is logarithmic on the scales, the difference is very substantial!

In Fig. 3.4, we also see that the less prescriptive methods (cache mode, and using numactl to move all allocations to MCDRAM) yield noticeably better results than the more explicit methods (autohbw and explicit memkind/FASTMEM usage) in which we are prescribing to the system which allocations to place in MCDRAM. The lesson is clear: if explicit usage is going to leave much of the MCDRAM idle, other methods that fully utilize the MCDRAM, even for seemingly less important uses, will very often offer the best performance.

Choosing to explicitly manage the MCDRAM in an application is unlikely to offer the best performance if any significant portion of the MCDRAM is underutilized.

Critical review for our Hello MCDRAM

Our "Hello MCDRAM" program is both relatively cache-friendly and bandwidth hungry. Therefore, Fig. 3.4 reflects those effects. There are some applications, perhaps small in number but important, that are known to be cache-unfriendly. Such applications are much more likely to be rewarded by using the flat memory mode instead of the cache memory mode. Also, there are applications that block into L2 with such high reuse that MCDRAM is not needed. Unless such application would benefit from blocking into the larger capacity of MCDRAM, the existence of MCDRAM may not matter.

numactl -H; Learning NUMA Node Numbering

A key command to know is "numactl -H." We have a few sample outputs, to illustrate actual usage, from a 64-core Knights Landing system, with 16 GB of MCDRAM memory and 64 GB of DDR memory. Fig. 3.5 shows the output when booted in cluster mode = quadrant and memory mode = cache. In such a case, there is only one NUMA memory node, numbered zero (0). The output of "numactl -H" for cluster modes *all-to-all* and *hemisphere* is identical to those for *quadrant* because the division of memory and MCDRAM is always identical.

Fig. 3.6 shows the two NUMA nodes created for cluster mode = *quadrant* with memory mode = *flat*. In this case, NUMA node one (1) is the MCDRAM. Since the NUMA nodes would be identical, the output shown in Fig. 3.6 is also exactly the output when using all-to-all and hemisphere cluster modes with the same memory mode. The DDR node is listed before MCDRAM because the distance to MCDRAM is purposefully higher to avoid making it become the default for all allocations by the operating system. The distances are discussed in detail in Chapter 4.

Fig. 3.7 shows the most complex example, which is cluster mode = *SNC-4* and memory mode = *flat*. In this case, there are eight NUMA nodes created. Note that available DDR and MCDRAM are divided into multiple NUMA nodes when dividing Knights Landing up using a NUMA cluster mode (i.e., SNC-4 or SNC-2). The


```
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
103 104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141
142 143 144 145 146 147 148 149 150 151 152 153 154
155 156 157 158 159 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193
194 195 196 197 198 199 200 201 202 203 204 205 206
207 208 209 210 211 212 213 214 215 216 217 218 219
220 221 222 223 224 225 226 227 228 229 230 231 232
233 234 235 236 237 238 239 240 241 242 243 244 245
246 247 248 249 250 251 252 253 254 255
node 0 size: 65432 MB
node 0 free: 62887 MB
node distances:
node   0
  0:  10
```

FIG. 3.5

numactl output for system in Quadrant + Cache modes.

```
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 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
103 104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141
142 143 144 145 146 147 148 149 150 151 152 153 154
155 156 157 158 159 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193
194 195 196 197 198 199 200 201 202 203 204 205 206
207 208 209 210 211 212 213 214 215 216 217 218 219
220 221 222 223 224 225 226 227 228 229 230 231 232
233 234 235 236 237 238 239 240 241 242 243 244 245
246 247 248 249 250 251 252 253 254 255
node 0 size: 65432 MB
node 0 free: 60405 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15917 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

FIG. 3.6

numactl output for system in Quadrant + Flat modes.

```
     available: 8 nodes (0-7)
     node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
128 129 130 131 132 133 134 135 136 137 138 139 140
141 142 143 192 193 194 195 196 197 198 199 200 201
202 203 204 205 206 207
     node 0 size: 16280 MB
     node 0 free: 15413 MB
     node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 144 145 146 147 148 149 150 151 152 153 154 155
156 157 158 159 208 209 210 211 212 213 214 215 216
217 218 219 220 221 222 223
     node 1 size: 16384 MB
     node 1 free: 15818 MB
     node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 96 97 98 99 100 101 102 103 104 105 106
107 108 109 110 111 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 224 225 226 227 228
229 230 231 232 233 234 235 236 237 238 239
     node 2 size: 16384 MB
     node 2 free: 15617 MB
     node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 112 113 114 115 116 117 118 119 120 121
122 123 124 125 126 127 176 177 178 179 180 181 182
183 184 185 186 187 188 189 190 191 240 241 242 243
244 245 246 247 248 249 250 251 252 253 254 255
     node 3 size: 16384 MB
     node 3 free: 15886 MB
     node 4 cpus:
     node 4 size: 4096 MB
     node 4 free: 3983 MB
     node 5 cpus:
     node 5 size: 4096 MB
     node 5 free: 3982 MB
     node 6 cpus:
     node 6 size: 4096 MB
     node 6 free: 3982 MB
     node 7 cpus:
     node 7 size: 4096 MB
     node 7 free: 3979 MB
     node distances:
     node   0   1   2   3   4   5   6   7
       0:  10  21  21  21  31  41  41  41
       1:  21  10  21  21  41  31  41  41
       2:  21  21  10  21  41  41  31  41
       3:  21  21  21  10  41  41  41  31
       4:  31  41  41  41  10  41  41  41
       5:  41  31  41  41  41  10  41  41
       6:  41  41  31  41  41  41  10  41
       7:  41  41  41  31  41  41  41  10
```

FIG. 3.7

numactl output for system in SNC-4 + Flat modes.

DDR nodes are listed first, and the MCDRAM nodes are listed last. The distances reflect the affinitization of DDR and MCDRAM to the divisions of Knights Landing in this mode. No matter which core you run on, with SNC modes there is a "near" MCDRAM and a "near" DDR. With SNC-4, there are also three (one on SNC-2) "far" MCDRAM nodes and three (one on SNC-2) "far" DDR. Because the number of NUMA nodes change, the actual "node" numbers for MCDRAM are not consistent between SNC modes or the other non-SNC mode. We have a later section *How to Not Hard Code the NUMA Node Numbers* to address how to stay independent of node numbering.

If you find "numactl -H" captivating, you really need to look at `/proc/<pid>/numa_maps`!

WAYS TO OBSERVE MCDRAM ALLOCATIONS

If looking at numactl -H output fascinates you, we encourage you to look at `/proc/<pid>/numa_maps`. These files contain information about each memory area used by a given process. This allows us to determine which NUMA nodes were used for the pages while running. Knowing about numa_map files can be a useful for debugging use of NUMA nodes and huge pages, or just verifying we have set up our program correctly. We do need to remember that allocations occur on first touch, and the NUMA maps will reflect this.

A read operation on numa_maps file will cause the kernel to scan the memory area of a process to determine how memory is used. One line is displayed for each memory area of the process. The numa_maps includes information about the memory policy currently in effect for particular memory areas as well.

Other ways to observe details about allocations include:

- numastat -m (includes huge page info too!)
- numastat -p *{pid,exec_name}*
- *cool idea:* watch -n 1 -p numastat *exec_name*
- cat/sys/devices/system/node/node*/meminfo
- cat/proc/meminfo
- *memkind_install_dir*/bin/memkind-hbw-nodes
- *numactl -hardware*
- lscpu

NUMACTL: MOVE ALL ALLOCATIONS TO MCDRAM

When we can size our application to fit all or most data within the MCDRAM, we can leave an application relatively unmodified but benefit from data being placed into MCDRAM. In an extreme case, if the entire application and all its data fit in MCDRAM, there would be no performance benefit in changing the application to do anything explicit with memory. The "numactl" (NUMA ConTroL) utility can be installed easily on Linux systems with commands such as "yum install numactl."

In order to run an application, with all the allocations, including the stack, going to MCDRAM, we need to use the "-m" option with the NUMA node(s) specified after the option. For quadrant/flat, we would use (see Fig. 3.5 to understand the node number "1"):

```
numactl -m 1 myprogram
```

For SNC-4/flat, we could use (see Fig. 3.6 to understand the node numbers "4 through 7"):

```
numactl -m 4-7 myprogram
```

In the SNC-4 case, we have not taken advantage of "near" versus "far" MCDRAM by lumping them all together as "4-7." In this case, allocations would first come from the "near" MCDRAM and then the other MCDRAM sections. Since `numactl` operates at a program level, there is no simple way to say "use near MCDRAM and then DDR" because the node number for "near" differs from core to core.

In both cases, we suggest always using "-p" instead of "-m" because it states a preference for MCDRAM instead of a requirement. If MCDRAM is exhausted, the application will still run if "-p" is used because it can also allocate from regular memory (DDR).

Combining these thoughts, it is not pretty, but we can do something like this:

```
mpirun -perhost 16 \
numactl -p 4 a.out : \
numactl -p 5 a.out : \
numactl -p 6 a.out : \
numactl -p 7 a.out ...
```

It is reasonable to assume that special support in each MPI implementation will offer this capability since this style of `numactl` command would not be a practical way to invoke thousands of nodes! The Intel MPI product team is planning support that will be controlled by an environment variable `I_MPI_NUMA_POLICY`. Consulting MPI documentation, looking for such support, is highly recommended to get the best behavior while running in SNC cluster modes.

OVERSUBCRIPTION OF MCDRAM: A KILLER OR AN OPPORTUNITY?

A simple "`grabmemory`" application, shown in Fig. 3.8, repeatedly calls `malloc()` for as many gigabytes of data as we specify. Fig. 3.9 shows that we can use "`numactl -p 1 grabmemory 50 touch`" to have our application allocate and use (touch) 50 GB of memory. However, if we require MCDRAM by using "`numactl -m1 grabmemory 50 touch`," as shown in Fig. 3.10, we see the application is aborted by the operating system. Note that the `malloc()` was successful, and the application did not print the failure message it was designed to print. This is because a memory allocation does not fail until a memory page (a page is a "chunk" of memory, generally 4 KB in size but larger sizes exist too) within the allocation is used and the required memory is not actually available. Such failures cannot be trapped by the application; the application must be aborted. In our example, that happens when the `memset()` call is made. Fig. 3.11 makes this very clear by showing that "`numactl -m 1 grabmemory 100`" is able to allocate 100 GB of MCDRAM even though our system has only 16 GB of MCDRAM. Since we did not actually touch that memory, it is allocated but using too much will definitely cause the application to be aborted. Fig. 3.9 serves as a warning that actually ensuring use of MCDRAM is *not* a matter of first-come-first-served as measured by calling `malloc()`. If an application allocates more space than can fit in MCDRAM, and does it with a preference for MCDRAM and not a requirement, the first usage grants the MCDRAM while


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SZALLOC (1024*1024*1024)
#define MAXTRY 25000
/* usage: grabmemory <#gigabytes> [touch] */
int main(int argc, char *argv[])
{
  int *ptr[MAXTRY];
  int i, maxtry=20, touch = (argc > 2);
  if (argc > 1) maxtry = atoi(argv[1]);
  if (maxtry > MAXTRY) maxtry = MAXTRY;
  for (i=0;i<maxtry;i++) {
    ptr[i]=malloc(SZALLOC);
    if (!ptr[i]) {
      printf("Failure after %dGB were "
          "successfully allocated.\n",i);
      return 1;
    }
    printf("Allocated         %dGB         ",i+1);
fflush(stdout);
    if (argc>2) {
      memset(ptr[i],i&255,SZALLOC);
      printf("and touched successfully.\n");
    } else
      printf("no touch\n");
    fflush(stdout);
  }

  printf("Allocated %s%dGB without problems.\n",
      (argc>2) ? "and touched " : "",
      maxtry);
  return 0;
}
```

FIG. 3.8

Application "grabmemory.c" to illustrate effects of numactl.

```
Allocated 1GB and touched successfully.
Allocated 2GB and touched successfully.
… output lines ommitted for space …
Allocated 49GB and touched successfully.
Allocated 50GB and touched successfully.
Allocated and touched 50GB without problems.
```

FIG. 3.9

Output of "numactl -p 1 grabmemory 50 touch." When usage exceeds MCDRAM capacity, pages will be assigned from regular memory when assignments have filled MCDRAM.

```
Allocated 1GB and touched successfully.
Allocated 2GB and touched successfully.
… output lines ommitted for space …
Allocated 14GB and touched successfully.
Allocated 15GB and touched successfully.
Allocated 16GB Killed
```

FIG. 3.10

Output of "numactl -m 1 grabmemory 50 touch." The application is KILLED by the operating system when it tried to *use* too much.

```
Allocated 1GB no touch
Allocated 2GB no touch
… output lines ommitted for space …
Allocated 98GB no touch
Allocated 99GB no touch
Allocated 100GB without problems.
```

FIG. 3.11

Output of "numactl -m 1 grabmemory 100." The allocations far exceed the size of MCDRAM, but the application did not fail because it did not *use* too much.

it is still available. Therefore, oversubscribing of MCDRAM may be a technique to consider for some applications.

Applications that request too much MCDRAM do not fail at `malloc()` time. Such applications are aborted when the number of "touched" memory pages, which are marked as "MCDRAM required," exceed the available MCDRAM. This also means that actually getting MCDRAM is NOT a matter of first-come-first-served as measured by calling malloc(). The allocation on usage can be a surprise if we were not expecting it. It also offers an opportunity to oversubscribe MCDRAM, via allocations with a preference for MCDRAMs, and defer to actual first usage to grant the MCDRAM, if available, or fall back to regular memory, if not.

AUTOHBW: MOVE SELECTED ALLOCATIONS TO MCDRAM

The autohbw library is an interposer library that redirects an application's use of standard heap allocators (i.e., `malloc()`, `calloc()`, `realloc()`, `posix_memalign()`, and `free()`) to utilize MCDRAM. The selection is done purely based on size for specific allocations within our application. Fig. 3.12 shows how to run an application such that allocations, between 250K and 6M in size, will try to allocate from MCDRAM.

Our example assumes we are in a non-SNC mode (e.g., quadrant) with a memory mode other than cache mode. We make this assumption and hard code NUMA node 1 in the setting. This is a bad idea for anything meant to last; we encourage you to learn more later in this chapter in the section *How to Not Hard Code the NUMA Node Numbers*.

When MCDRAM runs out, the application will allocate from regular memory. All allocation smaller than 250K in size, or larger than 6M in size, will not be diverted for special allocations from MCDRAM. In order to use with applications using MPI, we would need to put the LD_PRELOAD in a shell script and then invoke the `mpirun` command on the shell script.

```
export MEMKIND_HBW_NODES=1
export AUTO_HBW_SIZE=250K:6M
LD_PRELOAD=libautohbw.so:libmemkind.so myapplication
```

FIG. 3.12

Example of using autohbw.

In order for `autohbw` to work, the application must be dynamically linked with the memory allocation library that it normally uses. This is default behavior normally, so using `autohbw` commonly does not require worrying about this.

Also, the `autohbw` and `memkind` libraries need to be on the library path (LD_LIBRARY_PATH). Applications using deprecated `valloc()` or `memalign()` cannot be supported and will be aborted if those functions are called (a message will be printed suggesting the use of `posix_memalign()` instead).

The `autohbw` library is open-source, like `memkind`, so it is possible to contribute by extending for additional memory allocation routines if a special need arises beyond the `malloc()`, `calloc()`, `realloc()`, `posix_memalign()`, and `free()` routines.

MEMKIND/FASTMEM: EXPLICIT USAGE OF MCDRAM

Explicit memory usage in C/C++: memkind

When faced with an application with data needs in excess of the size of MCDRAM, we can explicitly choose which data is allocated into MCDRAM. We do this with a set of language constructs or library calls to direct allocations to be from the MCDRAM part of memory. Later in this chapter, we will discuss some tools and techniques to help determine what data will benefit most from using MCDRAM. For now, we will show how to explicitly allocate from MCDRAM assuming we know what we want.

C programmers have MCDRAM versions of the standard heap allocation routines via the functions `hbw_malloc()`, `hbw_calloc()`, `hbw_realloc()`, `hbw_free()`, `hbw_posix_memalign()`, and `hbw_posix_memalign_psize()`. Each of these functions mimic the behavior of the standard routines (with the same names less the "`hbw_`" prefix).

The default policy is for these routines to "prefer" MCDRAM and not require it. This means that when the allocated pages are eventually used, they will be assigned to pages from MCDRAM. In a prior section on numactl, we demonstrated the difference between allocations and page allocations on usage. The same applies for `memkind`, and the allocation does not fail based on MCDRAM availability. It is possible to set a different policy using `hbw_set_policy(HBW_POLICY_BIND)`. That can be done once in an application before allocations are done. Attempts to set policy more than once will fail.

C++ notes

The memkind library ships with some excellent examples, one of which shows how to override `new` to create types that allocate into MCDRAM. Look in the examples directory for memkind_allocated_example.cpp and memkind_allocated_example.hpp.

Explicit memory usage in Fortran: FASTMEM

Fortran programmers can redirect `ALLOCATE()` to use MCDRAM. A summary of the extensions to support this is shown in Fig. 3.13.

Feature	Description
!dir$ ATTRIBUTES FASTMEM *allocatable-variable/ allocatable-field*	Variable or field name tagged so that ALLOCATE for it will be from MCDRAM subject to the current FASTMEM policy.
!dir$ FASTMEM	Lexically next ALLOCATE directed to be from MCDRAM subject to the current FASTMEM policy.
!dir$ NOFASTMEM	Lexically next ALLOCATE directed to be from DDR (not MCDRAM).
FOR_GET_HBW_AVAILABILITY	ifcore module routine to tell if either the memkind library was not linked in or that MCDRAM is unavailable (return values discussed in the text of this chapter)
FOR_SET_FASTMEM_POLICY(*X*)	ifcore module routine to set FASTMEM policy for MCDRAM allocation. Parameter *X* can be: FOR_K_FASTMEM_INFO: return the current FASTMEM policy value. FOR_K_FASTMEM_NORETRY: issue an error (check ALLOCATE stat variable) if MCDRAM is not available. FOR_K_FASTMEM_RETRY_WARN: warn if MCDRAM is not available, then use DDR. FOR_K_FASTMEM_RETRY: if MCDRAM is not available, then automatically (and silently) use DDR.

FIG. 3.13

Summary of Fortran extensions to support MCDRAM usage.

These are extensions to Fortran and currently are unique to the Intel Fortran compiler on Linux. The initial directives were originally introduced in 2015 with the 16.0 compiler. The more recent 17.0 compiler (beta and release in 2016) offers additional control over policies centering on the concepts of "BIND" (require) versus "PREFER" (preferred) allocations. A hot area of debate is failure mode—in other words, what to do if MCDRAM is oversubscribed and how to fail. After we introduce how to use the extensions for MCDRAM, we have a section titled *Fortran FASTMEM failure modes* to discuss options for controlling MCDRAM over-allocation policies.

Extending Fortran ALLOCATE, for MCDRAM, is an emerging capability subject to change based on experiences and customer feedback. To augment our introduction here, our web site has the latest information on this topic—lotsofcores.com/ALLOCATE.

The ATTRIBUTES FASTMEM directive enables MCDRAM memory allocation for an allocated object. The Fortran documentation, like memkind, calls this "HBW" memory. The directive syntax is:

```
!DIR$ ATTRIBUTES FASTMEM :: object
```

The *object* must be an allocatable array. It may be a component of a derived type. It cannot be a local variable, a common block, or an array that will be allocated on the stack.

`FASTMEM` enables MCDRAM memory allocation for an allocated object at runtime. When the specified object is allocated using the `ALLOCATE` statement at runtime, the Fortran Run-Time Library (RTL) allocates the memory in MCDRAM. When the specified object is deallocated using the `DEALLOCATE` statement at runtime, the Fortran RTL deallocates the memory from MCDRAM.

When we use this directive in an application, we must specify the option `-lmemkind` to the compiler/linker. If the libraries required for MCDRAM support are not linked successfully, no warning will be given at compile time and the allocations will fail at runtime.

The Fortran compiler will link executables correctly even if we forgot to link in libmemkind. There is a Fortran runtime called `for_get_hbw_availability()` in the ifcore module to tell us that either the memkind library was not linked in or that MCDRAM is unavailable. The return value represents one of the following values in the ifcore module:

- `FOR_K_HBW_AVAILABLE`—MCDRAM is available.
- `FOR_K_HBW_NO_ROUTINES`—The routines needed for MCDRAM memory allocation are not linked-in (i.e., no libmemkind).
- `FOR_K_HBW_NOT_AVAILABLE`—MCDRAM memory is not available.

We can use ATTRIBUTES option ALIGN with FASTMEM; for example:

```
!DIR$ ATTRIBUTES FASTMEM, ALIGN:64 :: A
```

We can specify MCDRAM usage on each ALLOCATE instead of relying on a directive for the declaration which affects all ALLOCATE statements for affect variables. This gives us more control. We use a directive before the ALLOCATE statement which we want to go to MCDRAM as shown in Figs. 3.14 and 3.15. The FASTMEM directive has a counterpart NOFASTMEM to add clarity or for

```
real, allocatable :: x (:)
! …code…
!dir$ FASTMEM      ! FASTMEM just for the following
ALLOCATE
allocate (x (10), stat = istat)
```

FIG. 3.14

FASTMEM directive used for a specific ALLOCATE.

```
TYPE T
REAL, ALLOCATABLE :: x (:)
END TYPE T
TYPE (T) :: r
…
!dir$ FASTMEM
ALLOCATE (r%x (10000))   ! allocates in MCDRAM
```

FIG. 3.15

FASTMEM directive used to affect allocation of a field.

overriding a FASTMEM directive used on the original declaration. It is best to use NOFASTMEM on an ALLOCATE used to recover from a FASTMEM ALLOCATE.

Fortran FASTMEM failure modes

In this section, we discuss options for controlling MCDRAM over-allocation policies. This section is only important if our program might ever request an ALLOCATION for MCDRAM when sufficient MCDRAM was not available.

Extending Fortran ALLOCATE, for MCDRAM, is an emerging capability subject to change based on experiences, customer feedback, and eventually the standardization processes. To augment our introduction here, our web site has the latest information on this topic, see *For More Information* at the end of this chapter.

We can envision these policies for MCDRAM over-subscription:

1. *ALLOCATE prefers MCDRAM* but will allocate from DDR if MCDRAM is not sufficiently available. It does not fail due to "near" MCDRAM oversubscription, leaving physical pages to be allocated either at ALLOCATE or at first touch from MCDRAM is available but use DDR if MCDRAM is not available. If a page of MCDRAM is not available to allocate, then allocations should be from the DDR. If this fails, the program would be aborted as would happen for other out-of-memory errors. Linux NUMA software calls this a "PREFERRED" policy.
2. *ALLOCATE requires MCDRAM* and will fail if MCDRAM is not sufficiently available. One challenging question is whether the failure will be reported by ALLOCATE, or will happen at "first touch" triggering an out-of-memory program abort. The latter is the behavior in C/C++ program. The former, the desire to have ALLOCATE fail, may be an experimental feature of the Intel beta 17.0 compiler for feedback from the user community. Linux NUMA software calls this a "BIND" policy.

Two side-note notes (not critical to learning how to use MCDRAM): In all cases, we say "MCDRAM" but really mean "near MCDRAM." The "near" concept only matters for SNC-2 and SNC-4 modes; otherwise MCDRAM is always "near." Linux supports an "INTERLEAVE" policy also, intended for users that have a NUMA system but prefer to spread allocations out to present a mix of near/far allocations to even out

as if on an UMA machine. Even though Knights Landing can be configured in such a mode, we do not anticipate INTERLEAVE policy being used on Knights Landing.

ALLOCATE prefers MCDRAM

A Fortran program may set a policy to cause ALLOCATE to use DDR when MCDRAM is not otherwise available using an ifcore.mod function:

```
FOR_SET_FASTMEM_POLICY (FOR_K_FASTMEM_RETRY)
```

This function should be called early in the program and only once. It should be called before the first ALLOCATE statement. Failure to adhere to those restrictions will probably not provide the desired results. There are additional options as shown in Fig. 3.13.

ALLOCATE requires MCDRAM

A Fortran program may set a policy to force ALLOCATE to use only MCDRAM using an ifcore.mod function:

```
FOR_SET_FASTMEM_POLICY (FOR_K_FASTMEM_NORETRY)
```

This function should be called early in the program and only once. It should be called before the first ALLOCATE statement. Failure to adhere to those restrictions will probably not provide the desired results. There are additional options as shown in Fig. 3.15.

There is considerable disagreement about which policy should be used. Some believe the best policy would be to have ALLOCATE fail at allocation time if MCDRAM is over subscribed. With this policy, if insufficient MCDRAM is available to allocate the object to MCDRAM, the allocation may fail, leaving the status in the variable specified in the `STAT=` clause.

Combined with the FASTMEM directive for a specific ALLOCATE statement, we show how to handle a failure at an ALLOCATE statement in Fig. 3.16. The FASTMEM directive has a counterpart NOFASTMEM that is useful for clarity and for overriding any directive that may have been used on the original declaration. It is best to use NOFASTMEM on an ALLOCATE used to recover from a FASTMEM ALLOCATE.

```
real, allocatable :: x (:)
! …code…
!dir$ FASTMEM        ! FASTMEM just for the following ALLOCATE
allocate (x (10), stat = istat)
if (istat .ne. 0) then
  !dir$  nofastmem    ! we'll take any kind of memory for X
  allocate (x (10), stat = istat)
  if (istat .ne. 0) then …      ! no standard memory either
```

FIG. 3.16

Handing an ALLOCATE failure (experimental Fortran extension).

QUERY MEMORY MODE AND MCDRAM AVAILABLE

Applications designed to use MCDRAM in flat or hybrid mode should be sensitive to the amount of MCDRAM available. Assuming a 16-GB MCDRAM on a Knights Landing, the memory mode and cluster mode selected can both influence the amount of memory available. The memory mode may make 100%, 75%, or 50% of the MCDRAM available as flat memory (16, 12, or 8 GB). Of course, if booted in cache mode, the amount of high-bandwidth memory available will be 0. Furthermore, as we will discuss in Chapter 4 (section titled *Interactions of Cluster and Memory Modes*), the Knight Landing may be subdivided into halves or quarters. In such a case, 16 GB can become 16, 12, or 8 GB. Likewise, 12 GB can become 12, 9, or 6 GB and 8 GB can become 8, 6, or 4 GB of capacity available to the application per subdivision of the Knights Landing. Therefore, applications should not be written to assume a certain memory capacity.

The memkind library can be used to check for high-bandwidth memory starting with a `memkind_check_available(MEMKIND_HBW)` or `hbw_check_available()` to check for existence at all. A nonzero return would indicate no MCDRAM is available, which would happen on Knights Landing when booted in cache mode, or on processors with no MCDRAM at all. The function `memkind_get_size(MEMKIND_HBW,&total,&free)` is currently very limited. It will return an amount of memory that is available for use, but it has limitations that prevent it from reporting memory available in the free pool. It also reports memory available for use and is unaffected by allocations of memory without any usage occurring.

These functions are not efficient and therefore should not be called repeatedly in an application especially in any performance-sensitive code. It is best to call once and cache the value for an application. Fig. 3.17 offers a quick example of using these functions in C. Fig. 3.18 shows the output when run on the same 64-core Knights Landing which we have used previously in this chapter (equipped with 64 GB of memory plus 16 GB of MCDRAM).

SNC PERFORMANCE IMPLICATIONS OF ALLOCATION AND THREADING

For the best performance, allocating memory for threads should be done carefully in order to ensure memory is allocated "near" to the work. We can imagine a master thread in an application that allocates and initializes all the memory for the process and then lets the thread pools work on that memory. If the master thread and the worker threads reside in different NUMA nodes, then they will have suboptimal performance. Another variation is that you have multiple thread pools in different NUMA domains, but the first thread pool did the initial "touch," which allocated the physical space behind the virtual allocation. Careful planning and understanding of these problems will be helpful to achieve maximum performance.


```
#include <stdio.h>
#include <memkind_hbw.h>
#include <hbwmalloc.h>

int main(int argc, char *argv[]) {
 size_t total, free;
 int policyis, w, int *mem;
 char *hbw = getenv("MEMKIND_HBW_NODES");
 if (hbw)
  printf("Environment variable "
         "'MEMKIND_HBW_NODES' = '%s'\n",hbw);
 printf("memkind_check_available(MEMKIND_HBW) ="
        "%d (zero means yes)\n",
        memkind_check_available(MEMKIND_HBW));
 printf("memkind_get_size(MEMKIND_HBW) = %d\n",
        memkind_get_size(MEMKIND_HBW,&total,&free));
 printf("hbw_check_available() = %d (zero means yes)\n",
        hbw_check_available());
 printf("total = %15.6fM; free = %15.6fM\n",
        total/1024/1024.0,free/1024/1024.0);
 printf("total = 0x%012lx; free = 0x%012lx\n",
        total,free);
 policyis = hbw_get_policy();
 printf("policy=%d\nHBW_POLICY_BIND=%d\n"
        "HBW_POLICY_PREFERRED=%d\n",
        policyis,
        HBW_POLICY_BIND==policyis,
        HBW_POLICY_PREFERRED==policyis);
 mem = hbw_malloc((size_t)512);
 if (mem<0) printf("Error: ");
 printf("hbw_malloc = %012lx\n",mem[w]);
 return 0;
}
```

FIG. 3.17

Simple "Hello memkind" application.

```
memkind_check_available(MEMKIND_HBW) =0 (zero means yes)
memkind_get_size(MEMKIND_HBW) = 0
hbw_check_available() = 0 (zero means yes)
total =     16384.000000M; free =     15926.421875M
total = 0x000400000000; free = 0x0003e366c000
policy=2
HBW_POLICY_BIND=0
HBW_POLICY_PREFERRED=1
hbw_malloc = 7fcbf740e200
```

FIG. 3.18

Output of our simple "Hello memkind" application.

ALLOCATION WITH SNC

Since SNC is software visible, it requires some special considerations to utilize well in the context of memory allocators. Because modern operating systems use an allocation-on-demand strategy to assign physical pages to virtual addresses only when accessed, some considerations will help performance. For optimal performance data should be allocated and first touched by code running on the NUMA node which will use the particular data the most. This warrants extra thought if memory is allocated, used, freed and reallocated by a central memory allocator. While Linux does support migration of memory from one NUMA node to another, this is an expensive operation, just like returning memory to the operating system, and is worth avoiding if we can.

The right NUMA node for a given NUMA allocation request depends upon the thread asking for it, since SNC subdivides memory and the cores/threads. By default software processes and threads can migrate across the operating system-managed hardware threads. If a calling software thread allocates memory from one node, it may be moved to another node, making the access slower. Intel MPI and OpenMP implementations allow us to affinitize their threads, typically through environment variables so that the migration problem does not happen.

HOW TO NOT HARD CODE THE NUMA NODE NUMBERS

While writing this chapter, we heard the terse summary: "If the system administrator has installed memkind correctly, `MEMKIND_HBW_NODES=1` is not required at all." We will go further: We should avoid the `MEMKIND_HBW_NODES` environment variable all together. This short section is about how to do that.

Avoid using the `MEMKIND_HBW_NODES` environment variable all together.

The Platform Memory Topology Table (PMTT) is a relative newcomer to the ACPI specification (ACPI 5.0, Dec. 6, 2011) and seems likely to be expanded or displaced in the future by other more comprehensive tables. No application should try using the PMTT directly, but the memkind library does so for us. The PMTT contains both bandwidth and latency information that is critical in the reliable identification of MCDRAM. Its existence allows MCDRAM allocation support to automatically select the appropriate MCDRAM from which to allocate.

It may be the case that some systems lack PMTT support, or the memkind library was not installed with PMTT support. For such cases, the environment variable `MEMKIND_HBW_NODES` exists to specify the NUMA node to utilize. This environment variable will override the PMTT table, and we strongly advise against using it if a PMTT table exists. We recommend figuring out how to get PMTT supported on every system!

Assuming the PMTT exists and memkind is properly installed to take advantage of it, the memkind will allocate MCDRAM using the "near" MCDRAM and then, if the request was a preference instead of a requirement, use the "near" DDR. This is essential in the SNC cluster modes because a single value for `MEMKIND_HBW_NODES` would not get the best results.

Since the PMTT is not readable by users, there are specific things done by the installation of memkind that manage access to the information. This is an installation issue, and the details are documented in the memkind package.

The `memkind_hbw_check_available()` call will return a value of `MEMKIND_UNAVAILABLE` if the PMTT table information does not exist, or if it does not indicate the existence of MCDRAM (which does happen when memory mode = cache).

With the PMTT and MCDRAM both in place, `hbw_malloc` and other memkind functions that request MCDRAM, will allocate from "near" MCDRAM and then fall back on DDR allocations ("near" DDR first). This should eliminate our temptation to control with `MEMKIND_HBW_NODES`.

APPROACHES TO DETERMINING WHAT TO PUT IN MCDRAM

Hopefully, earlier sections illustrated that managing MCDRAM explicitly is up against some stiff competition for high performance without resorting to modification of an application. Nevertheless, as programmers we can "know best" and get better performance for our applications if the application has a working set larger than the MCDRAM. Of course, if all our data fits inside the MCDRAM, we have little hope of performance upside! In fact, it does not have to be literally all the data; if the working set fits into MCDRAM, then we will see benefits essentially identical to having everything fit in MCDRAM. There is a strong attraction to simply scaling-out an application to use more Knights Landings so that the working set per Knights Landing fits in MCDRAM.

Assuming we decide to try explicit management of MCDRAM (in flat or hybrid memory modes) using the `memkind` library or `FASTMEM` directives, we need to decide what data to place into MCDRAM.

This section examines two approaches for identifying which application data structures should be explicitly allocated within MCDRAM in order to maximize performance.

> Approach 1: Timings with select data items allocated fast/slow. The best method to do this is using Knights Landing (Approach 1a), but we also show how to approximate it with a dual-socket system based on Intel® Xeon® processors (Approach 1b).
> Approach 2: Use the "memory profiling" capability of the Intel® VTune™ Amplifier tool.

Approach 1a and 1b require code changes. Approach 1b utilizes the two memory regions present on a dual-socket Intel Xeon system (a NUMA environment) to emulate a Knights Landing node containing both MCDRAM and DDR memory, while

Approach 2 utilizes the Intel VTune memory profiling technology to identify the most suitable candidates on the program without requiring coding changes.

We will demonstrate the utilization of these techniques with a real-world hydrodynamics application. This application, called CloverLeaf, is widely utilized for research purposes. Although CloverLeaf is a proxy application (mini-app), the code-base is sufficiently representative to allow meaningful conclusions to be formed regarding the performance of these techniques within larger, more complex applications. Performance results using a Knights Landing, obtained through the application of each of these approaches, are presented in Fig. 3.22. For further information on CloverLeaf, see *For More Information* at the end of this chapter.

APPROACH 1: OBSERVING OR EMULATING MCDRAM EFFECTS

These approaches start by identifying candidate data structures, and running timing tests to determine the benefits of moving into MCDRAM. We can utilize an automated trial-and-improvement-based methodology to examine the performance of an application when executed with alternative setup parameters.

We offer Approach 1a using actual runs on Knights Landing machines, and slightly less accurate Approach 1b using a NUMA system and forcing fast/slow through memory allocations to "near" and "far" memories.

The Approach 1b execution environment is configured such that only one Intel Xeon processor within the system is utilized to actually execute the application. The MCDRAM targeted memory allocations, using the `hbw_malloc` function, within the application are placed within the memory subsystem of the local processor. The standard DDR memory allocations, however, are placed within the memory subsystem of the "remote" processor ("far" memory). Applications will therefore experience higher latency and lower bandwidth for the memory accesses which are serviced from the "remote" memory subsystem, relative to those allocated from within the "local" memory domain. This is due to the fact that these accesses need to traverse the QPI links between the two processor sockets. Using the two different memory regions in this manner effectively emulates the different memory performance

```
#ifdef mcdram_array1
!DIR$ ATTRIBUTES FASTMEM :: array1
#endif

And for C/C++ codes modify the allocations as follows:
float *array1
#ifdef mcdram_array1
array1 = (float *) hbw_malloc( sizeof(float) * N);
#else
array1 = (float *) malloc( sizeof(float) * N);
#endif
```

FIG. 3.19

Examples of changes needed for Approach 1.

characteristics which an application experiences when executing on a Knights Landing node.

It is *not* recommended, however, that applications should be optimized for Approach 1b, as it is not a perfectly accurate representative of the bandwidth and latency characteristics found within Knights Landing. This approach does, however, represent a reasonable, low-effort way of determining which application data structures are most bandwidth sensitive and therefore should be preferably allocated within the MCDRAM memory domain in order to maximize overall performance.

Prerequisites:

1. Obtain access to a Knights Landing or a dual-socket Intel Xeon processor-based system running Linux
2. Install the memkind library

Stage 1: Code modification

Modify the application such that the allocation of its most significant data structures can be individually controlled via preprocessor macros. For Fortran codes, include FASTMEM directives. In the CloverLeaf codebase, our changes can be found in the `definitions.f90` file. For C/C++ codes, switch explicit allocations to utilize the memkind /`hbwmalloc` interfaces. Examples are shown in Fig. 3.19.

Stage 2: Manual Execution

1. Manually build multiple versions of the application, each of which places a different data structure into the memory region emulating the Knights Landing "MCDRAM," all other allocations should continue to be allocated within the memory region emulating the Knights Landing "DDR" resources.
   To activate our sample code in Fig. 3.20, one of the compilations would include `-Dmcdram_array1`. This could also be an opportunity to print out, or otherwise capture, the allocation sizes as input into our later thinking of how to use this information. In the optional "Stage 3," we suggest ways to automate this for larger applications.
2. Ensure MCDRAM allocations will be located where we need them.
   a. For Approach 1a—we want them actually in the MCDRAM, set the `MEMKIND_HBW_NODES` environment variable with a command such as this bash shell command (node 1 assumes Quadrant/Flat configuration):
      `$ export MEMKIND_HBW_NODES = 1`
   b. For Approach 1b—we want them located within the local "near" memory domain, set the `MEMKIND_HBW_NODES` environment variable with a command such as this bash shell command:
      `$ export MEMKIND_HBW_NODES=0`
3. Configure the execution environment (e.g., `OMP_NUM_THREADS`, etc.) such that the application does not require computational resources beyond those available within one processor of the target platform.

```
#!/bin/bash

export OMP_NUM_THREADS=<# cores on one processor>
export MEMKIND_HBW_NODES=0

compile_and_run() {
    make clean 1>/dev/null 2>&1
    make COMPILER=INTEL \
      MPI_COMPILER=mpiifort \
      C_MPI_COMPILER=mpiicc \
      FASTMEM=$1  1>/dev/null 2>&1
    mv clover_leaf clover_leaf_$1
    numactl --membind=1 \
      --cpunodebind=0 \
      ./clover_leaf_$1 1>clover_output_$1 2>&1
    mv clover.out clover.out_$1
    rm clover.in.tmp
}

compile_and_run "density0"
compile_and_run "density1"
…(repeat for each data array)…
```

FIG. 3.20

Pseudocode (based on the bash shell) for use with the CloverLeaf application.

4. Execute each instance of the application. For Approach 1b (only), we want to use a numactl command that binds memory allocations by default to the "far memory":

   `$ numactl -m 1 -N 0 ./app <app options>`
5. Record the runtimes for each execution of the application and sort these with execution times from the lowest to the highest.
6. Identify the data structures which achieved the lowest overall execution times.
7. Within the final application source code/binary, allocate the top *N* data structures within the high-bandwidth memory, up to the capacity limitations of the MCDRAM memory resources available within the Knights Landing nodes.

Revisiting steps 1–5 with multiple "-D" definitions active at once might be helpful, making sure the allocations either fit within the MCDRAM size we expect. Also, we can consider the order that we do allocations into MCDRAM both from the perspective of priority and packing within different MCDRAM sizes. Recall that when using an SNC cluster mode, we will most likely consider available MCDRAM size to be reduced by focusing on the "near" portion of MCDRAM instead of all of it.

Stage 3: Autotuning Configuration (Optional)

For large complex applications, it may not be practical to manually build and execute each different variant of the application. In this scenario, we recommend utilizing your favorite autotuning framework (or scripting language) such that the application can be repeatedly built, using different preprocessor definitions, and executed on the

target platform using the approach outlined in stage 2. Our example pseudocode for use with the CloverLeaf application, to implement Approach 1b, is shown in Fig. 3.20.

APPROACH 2: USING INTEL VTune TO DETERMINE MCDRAM CANDIDATE DATA STRUCTURES

In this approach, the memory profiling facilities of the Intel VTune™ Amplifier (Chapter 14, although we do not duplicate a discussion of "memory profiling" in that chapter) are utilized to identify which application data structures should be explicitly allocated within MCDRAM in order to maximize performance.

Prerequisite:

- Obtain access to a Knights Landing or an Intel Xeon processor-based system, with the Intel Compiler suite and VTune profiler technology installed.

Stage 1: Profiling Data Collection

1. Build the application such that the Intel compiler generates full debugging information within the resulting object files (e.g., by employing the -g compiler option on Linux or the/Z7 compiler option on Windows).
2. Configure the execution environment such that the application will execute across all of the available processing resources within the target node (i.e., set `OMP_NUM_THREADS`, etc., as required).
3. Execute the application under VTune such that the profiler conducts a "memory analysis." The following command demonstrates how to achieve this using the command line version of VTune on a Linux-based system:

```
$ amplxe-cl -collect memory-access -data-limit=0 -knob analyze-mem-objects=true -knob mem-object-size-min-thres=1024 -r <insert results directory name> -app-working-dir /path/to/working_directory--/path/to/working_directory/application binary
```

Stage 2: Profiling Data Analysis

1. After the profiling analysis has completed, load the graphical version of the VTune profiler.
2. On the "Welcome" screen, select "Open Result" and navigate to the directory in which you saved the profiling result directory generated in Stage 1. Open the resulting `<name>.amplxe` file.
3. The "Memory Analysis" profiling results will be displayed and ensure that the "Memory Usage" viewpoint is selected.
4. Navigate to the "Bottom-up" tab.
5. Select the "Bandwidth Domain/Bandwidth Utilization Type/Memory Object/Allocation Stack" grouping.

6. In the resulting profiling data, expand the "DRAM GB/sec" and then the "High" entries.
7. Order the results based on the "LLC Miss Count" field.
8. This will provide an ordering for the data structures within the application that are most memory bandwidth sensitive.

Stage 3: Code Modification

- Modify the application code such that the highest priority data structures (as identified from the list derived from Stage 2) are allocated within the MCDRAM available on the target Knights Landing. Recall that available MCDRAM sizes are reduced when dividing Knights Landing up using a NUMA cluster mode.

RESULTS ANALYSIS OF THE TWO APPROACHES

Using the approaches described above, several experiments were conducted to identify which application data structures within the 3D version of CloverLeaf should be explicitly allocated within MCDRAM in order to deliver the largest speedup in application execution time. The priority orderings for the data structures obtained through each of the above methods are presented in Fig. 3.21, together with results in Fig. 3.22 obtained from experiments on Knights Landing using the autotuning methodology outlined previously in "Approach 1."

The results show that both approaches deliver a preference ordering, for the allocation of application data structures within MCDRAM, which closely matches the performance results observed on a Knights Landing.

Additionally, Fig. 3.22 shows the speedup achieved in the execution time of the CloverLeaf 3D application by explicitly allocating data structures within the MCDRAM resources, relative to only utilizing standard DDR memory. In these experiments, the 600^3 cell benchmarking simulation problem was utilized and executed for 87 time-steps. The eight highest priority data structures, as identified by each of the previous approaches, were able to be explicitly allocated within the MCDRAM (up to the capacity limits of this memory region running in Quadrant/Flat—16 GB).

SUMMARY OF TWO APPROACHES TO "WHAT GOES IN MCDRAM"

Overall the profiling tool-based methodology described in "Approach 2" represents the easiest and least time consuming method for conducting this analysis. This "memory profiling" capability in VTune is relatively new, and is being refined and extended. We highly recommend it. The auto-tuning/trial-and-improvement-based methodology described in "Approach 1" requires significantly more effort and potentially requires substantial source code modifications. It also takes significantly longer to complete due to the multiple application builds and executions which it requires. Although the experimental results show that for this application, the autotuning-based approach can deliver improved performance results and that the

MCDRAM Allocation Preference Order	Approach 1a: Code changes to explore actual MCDRAM performance on Knights Landing	Approach 1b: Code changes to get predictions on MCDRAMeffects using dual-socket Xeon (NUMA system)	Approach 2: No code changes needed –use VTune Memory Profiling feature
1	**work_array1**	work_array1	work_array2
2	**work_array5**	work_array5	work_array3
3	**work_array7**	work_array3	work_array5
4	**work_array3**	work_array7	work_array1
5	**work_array4**	work_array 2	vol_flux_z
6	**density1**	work_array4	volume
7	**work_array6**	work_array6	yvel1
8	**work_array2**	density1	zvel1
9	vol_flux_z	vol_flux_z	xvel1
10	energy1	zvel1	pressure
11	volume	energy1	density1
12	xvel1	yvel1	energy0
13	zvel1	pressure	vol_flux_y
14	pressure	volume	vol_flux_x
15	yvel1	xvel0	density0
16	xvel0	xvel1	viscosity
17	viscosity	zvel0	zarea
18	yvel0	yvel0	yarea
19	vol_flux_x	vol_flux_x	energy1
20	vol_flux_y	viscosity	xarea
21	zvel0	yarea	xvel0
22	xarea	xarea	mass_flux_z
23	density0	vol_flux_y	yvel0
24	zarea	zarea	zvel0
25	yarea	density0	mass_flux_y
26	mass_flux_y	soundspeed	mass_flux_x
27	mass_flux_x	energy0	work_array7
28	mass_flux_z	mass_flux_z	soundspeed
29	soundspeed	mass_flux_y	celldx
30	energy0	mass_flux_x	celldy
31	cellx	celldz	celldz
32	vertexdx	vertexdx	vertexdx
33	vertexdz	vertexx	vertexdy
34	celldy	vertexdy	vertexdz
35	celly	celldx	work_array4
36	cellz	vertexz	work_array6
37	vertexdy	vertexy	vertexz
38	vertexy	celldy	cellx
39	vertexz	vertexdz	cellz
40	vertexx	cellx	vertexy
41	celldx	cellz	celly

FIG. 3.21

Preference ordering for the allocation of data structures within MCDRAM found empirically and by the various approaches. The top eight on the list do fit into MCDRAM in our case.

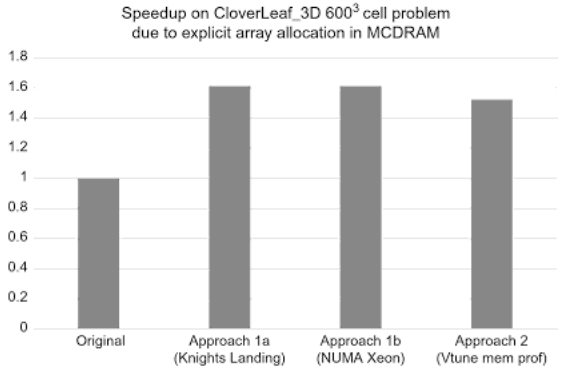


FIG. 3.22

Speedups achieved by explicitly allocation into MCDRAM based on recommendations.

array allocation preference ordering which it identifies is closer to that observed on actual Knights Landing silicon.

Utilizing the approach which employs dual-socket Intel Xeon processors to emulate a Knights Landing node (Approach 1) also facilitates implementation of more advanced explicit memory management strategies using the `memkind` library, and to examine their performance without needing actual Knights Landing-based systems. These may, for example, include the implementation of solutions in which data structures are explicitly staged into the MCDRAM and then, following the completion of computational operations, staged back to the DDR Knights Landing memory regions.

Before modifying applications to explicitly allocate data structures within MCDRAM, we would encourage the use of one or both approaches to gather useful information if the utilization of the autotuning-based approach proves to be infeasible for particular applications. The speedup results obtained through the application of the VTune profiling tool-based methodology (Approach 2) show that this approach can deliver performance improvements which are comparable to those achievable using the substantially more time-consuming autotuning-based method (Approach 1).

WHY REBOOTING IS REQUIRED TO CHANGE MODES

A common question that is asked about Knights Landing is "Why is a reboot necessary to change modes?"

By designing Knights Landing to offer MCDRAM as cache or memory, and to allow the cores to be configured in five different cluster configurations which span NUMA and UMA cluster designs, we have a single-chip solution that can change things about the design of a machine at a level that have traditionally been unchangeable. Operating systems and applications are not prepared to have NUMA distances change or go away, key memories come and go, and caching structures change. It is clear that when modes change that applications need to be stopped, the operating system needs to rebuild structures around the new shape of the machine, and the processor needs to purge most internal tables associated with memory tags, routing, and caching. The way to do that is a reboot of the system, which reinitializes all the information of the transformed Knights Landing when a cluster or memory mode is changed at all levels: applications, operating systems, and the processor itself.

As we will see, the ability to change the parameters for the next boot, coupled with efforts to accelerate reboots, is the best solution available for this unusual ability to pack multiple system designs in a single processor.

BIOS

The memory and cluster modes of Knights Landings are set during the boot process by the system BIOS. An obvious way to affect these settings is to boot into the BIOS, change the settings, exit, and reboot (Figs. 3.23–3.25). Since this requires two reboots to change modes, system vendors offer utilities to change BIOS settings from the command line so that new settings will take affect on the next boot.

We will describe how to do this on systems with Intel motherboards and an Intel-supplied BIOS. We will gladly add information for other vendors if we are contacted with pointers to the information. We have reserved a URL for us to share information that we mention in *For More Information* at the end of this chapter.

SAVE/RESTORE/CHANGE ALL BIOS SETTING

We can save BIOS settings to a file using the command:

```
sudo /bin/syscfg/syscfg /s BIOSQcache.ini
```

And we can restore them using the command:

```
sudo /bin/syscfg/syscfg /r BIOSQcache.ini/b
```

But more interestingly, we can change the three mode settings with individual commands. Figs. 3.26–3.28 show how to display the mode settings from a Linux command line (Intel offers Windows support as well).

FIG. 3.23

Example BIOS settings for the Cluster Modes.

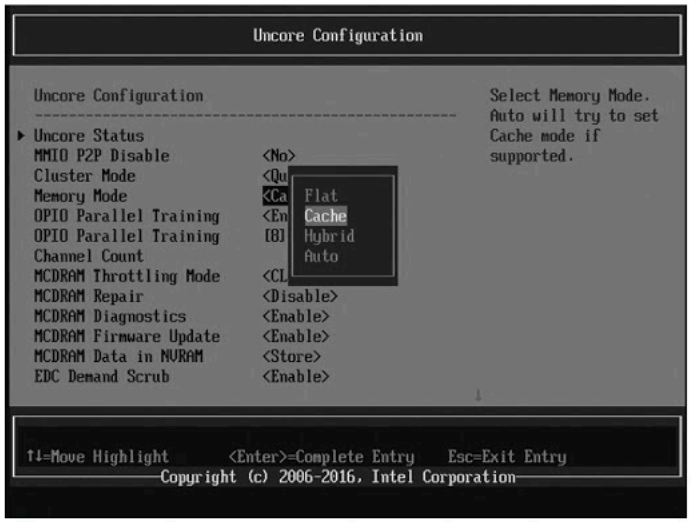


FIG. 3.24

Example BIOS settings for the Memory Modes.

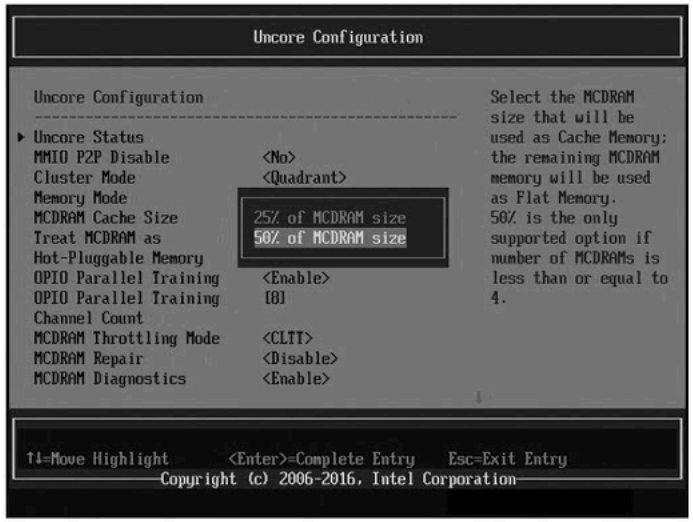


FIG. 3.25

Example BIOS settings for when “Hybrid” memory mode is used.

```
sudo /bin/syscfg/syscfg /d biossettings "Cluster Mode"

Cluster Mode
============
Current Value : Quadrant
------------------------
Possible Values
---------------
All2All : 00
SNC-2 : 01
SNC-4 : 02
Hemisphere : 03
Quadrant : 04
Auto : 05
```

FIG. 3.26

Displaying the current Cluster Mode setting using the Intel syscfg tool.

```
sudo /bin/syscfg/syscfg /d biossettings "Memory Mode"

Memory Mode
===========
Current Value : Cache
---------------------
Possible Values
---------------
Cache : 00
Flat : 01
Hybrid : 02
Auto : 03
```

FIG. 3.27

Displaying the current Memory Mode setting using the Intel syscfg tool.

```
sudo /bin/syscfg/syscfg /d biossettings "MCDRAM Cache Size"

MCDRAM Cache Size
=================
Current Value : 25% of MCDRAM size
----------------------------------
Possible Values
---------------
25% of MCDRAM size : 01
50% of MCDRAM size : 02
```

FIG. 3.28

Displaying the current MCDRAM Cache Size setting using the Intel syscfg tool.

```
SET: sudo /bin/syscfg/syscfg /bcs "" "Cluster Mode" 4
Successfully Completed
SET: sudo /bin/syscfg/syscfg /bcs "" "Memory Mode" 2
Successfully Completed
SET: sudo /bin/syscfg/syscfg /bcs "" "MCDRAM Cache Size" 1
Successfully Completed
```

FIG. 3.29

Setting the three modes using the Intel syscfg tool.

```
#!/bin/bash -f
if [ $# -lt 2 ]; then
  echo "usage: $0 ClusterModeCode MemoryModeCode [MCDRAMCacheSizeCode]"
fi
HYBRID=`sudo /bin/syscfg/syscfg /d biossettings "Memory Mode" | \
      grep -c "Current Value : Hybrid"`
if [ $# -gt 1 ]; then
  sudo /bin/syscfg/syscfg /bcs "" "Cluster Mode" $1
  sudo /bin/syscfg/syscfg /bcs "" "Memory Mode" $2
  HYBRID=`sudo /bin/syscfg/syscfg /d biossettings "Memory Mode" | \
       grep -c "Current Value : Hybrid"`
  if [ $# -gt 2 -a $HYBRID -gt 0 ]; then
    sudo /bin/syscfg/syscfg /bcs "" "MCDRAM Cache Size" $3
  fi
  echo "New values:"
else
  echo "Current values:"
fi

sudo /bin/syscfg/syscfg /d biossettings "Cluster Mode"
sudo /bin/syscfg/syscfg /d biossettings "Memory Mode"
if [ $HYBRID -gt 0 ]; then
  sudo /bin/syscfg/syscfg /d biossettings "MCDRAM Cache Size"
fi
echo " "
```

FIG. 3.30

Our "`setKNLmodes`" script to set and/or display current mode settings.

BIOS utilities can display the current BIOS setting. Since the settings can be changed and they only take affect at reboot time, the displayed value does not necessarily represent the mode the machine is currently in.

Fig. 3.29 shows how to set SNC-4 cluster mode with MCDRAM set to 25% cache. We wrote a little script to that can do this with a single command "`setKNLmodes 4 2 1`" which we show in Fig. 3.30. The script carefully avoids referring to the "MCDRAM Cache Size" parameter if the memory mode is not previously set to "Hybrid" since the Intel syscfg imposes that restriction.

An administrator password can be created for the BIOS, which would impact all the commands we have been shown. The documentation for the BIOS utilities (titled the *Intel® System Configuration Utility*) shows that we have to add "`/bap <BIOS administrator password>`" to the restore command line in such a case, or add the password where we show an empty string ("") in the single-setting-at-a-time `/bcs` commands.

SUMMARY

Key design choices have traditionally been made by hardware designers and software has had to adapt. Knights Landing turns this around and offers an unprecedented flexibility to leave that decision to the user of the machine. This ability to configure to suit an application is exciting but finding the optimal fit requires some understanding of the options. The appeal of directly programming for the "software controlled cache" (flat/hybrid memory modes) is obvious, but it needs to be tempered by the high-performance Knights Landing offers programs without modification through cache modes, numactl and autohbw. We should avoid using the environment variable `MEMKIND_HBW_NODES` and other methods to hard code NUMA numbers in our applications and scripts. Also, with careful usage the more sophisticated SNC cluster modes may benefit programs, especially MPI+X (e.g., MPI+OpenMP) style programming.

Ask your system provider for utilities to set BIOS options so as to avoid needing to boot into the BIOS to interactively change BIOS options related to cluster and memory modes. This will greatly reduce the time to leave a running system, reconfigure it, and be up and running again.

Such flexibility is new, and it is reasonable to assume more techniques and software support will emerge. We welcome feedback, and we will post errata and additional insights on our book website (http://lotsofcores.com).

FOR MORE INFORMATION

- Fortran allocation for MCDRAM is an emerging capability subject to change based on experiences and customer feedback. Therefore, we created a URL where we willl have pointers to the best information on this topic of which we are aware. http://lotsofcores.com/ALLOCATE.
- Memkind library information and github, https://github.com/memkind, clone via: "git clone https://github.com/memkind/memkind.git."

- Knights Landing BIOS configuration article on our website, which we will happily update with any pointers submitted to us on this topic, http://lotsofcores.com/KnightsLandingBIOS.
- Christopher Cantalupo, Vishwanath Venkatesan, Jeff R. Hammond, Krzysztof Czurylo, and Simon Hammond, *User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*, Mar. 18, 2015, http://tinyurl.com/memkind-arch.
- Knights Landing "numactl" listings for all 20 modes, from one particular system. While your results may vary, it is interesting can be interesting to review. http://lotsofcores.com/KnightsLandingNUMActl.
- Overcommit settings in Linux are changeable, see http://tinyurl.com/overcommit-accounting.
- Configuring the out-of-memory killer in Linux: http://tinyurl.com/oom-killer.
- Further information on CloverLeaf, as well as the actual source code, can be found at: https://github.com/UK-MAC.
- Information about PMTT, and other aspects of ACPI tables, http://acpi.info.
- Download code from this chapter, and other chapters, at http://lotsofcores.com.

This page intentionally left blank

CHAPTER

Knights Landing architecture

4

In this chapter, we provide further details on the Knights Landing architecture that we introduced in Chapter 2. We describe the tile and core architecture in detail. We dive deeper into the cluster modes and memory modes supported by Knights Landing, explain how they interact, and mention important programming considerations when using them.

What is new with Knights Landing in this chapter?
This entire chapter is about Knights Landing.

TILE ARCHITECTURE

The Knights Landing tile (Fig. 4.1) is the basic unit that gets replicated across the chip. It consists of two cores, each core is connected to two vector processing units (VPUs), and shared 1 MB L2.

The Knights Landing core is a new core design. As described in Chapter 2, it was derived from a low power core, code named Silvermont originally designed for an Intel® Atom processor, but with heavy modifications to make it suitable for High-Performance Computing. It is a 2-wide, *out-of-order* core that supports up to four threads. This new core is one of the primary reasons for the significant improvement in single thread performance (>3×) in Knights Landing over the prior generation product in the Intel® Xeon Phi family code named *Knights Corner*.

The VPU is the heart of computation on Knights Landing. It is home to all the floating-point (FP) compute capability, ranging from legacy instructions like SSE and x87 through AVX and AVX2 to the latest AVX-512 vector instructions. Each core is connected to two VPU units and hence can execute two 512-bit vector multiply-add instructions per cycle. Thus, each core can deliver 32 dual-precision (DP) or 64 single-precision (SP) FP operations each cycle.

The L2 cache is 1 MB in capacity with 16-way associativity with 64B cache lines that is shared between both cores in the same tile. The L2 cache supports a bandwidth of 1 cache line read and ½ cache line write per cycle, which is shared by both cores as depicted in Fig. 4.2. The bus interface unit (BIU) controls the L2 cache and coherency between the two cores in the tile. One core can use all the available read and write bandwidth if the other core is not accessing the L2. The L2 cache is private to each

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00004-1

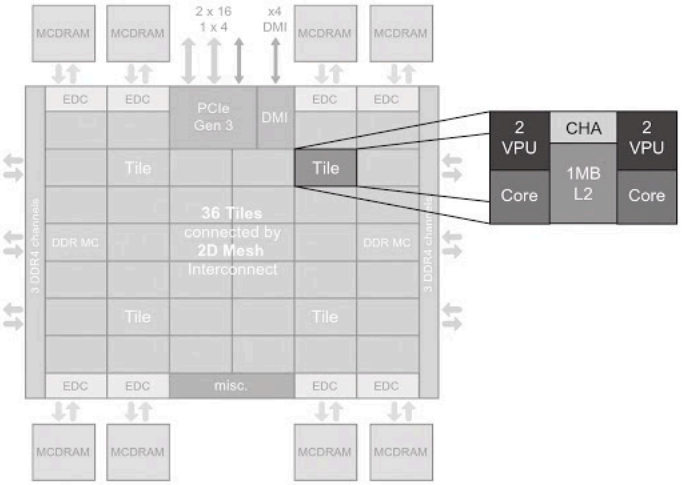


FIG. 4.1

Knights Landing tile within the larger processor die.

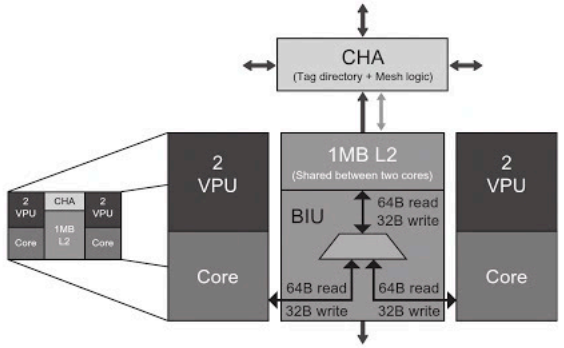


FIG. 4.2

Connections within a tile between cores, BIU, L2, and CHA.

tile, which means multiple tiles reading the same cache line will have their own private copy in their respective L2. However, all L2 caches are coherent; a write operation to a cache line from one tile will invalidate all other copies of that line in other tiles.

The cache coherency across all L2 caches is enforced by a distributed directory-based MESIF protocol that tracks the cache lines present inside all the tiles in *M*odified, *E*xclusive, *S*hared, *I*nvalid, or *F*orward state. The directory is distributed across all Knights Landing tiles with a portion stored in each tile in the box labeled *CHA*, for *Caching-Home Agent*, as shown in Fig. 4.2. While the CHA box physically resides in the tile, it is logically part of the Knights Landing on-die interconnect *mesh*. The portion of tag directory present in CHA does not have any connection to the lines stored in the colocated L2 cache; the cache line addresses are distributed among all the CHAs using an address hash.

CORE AND VPU

In this section, we describe the core and VPU architecture in more detail. As mentioned before, several major modifications were made to the original low-power Intel Atom processor core design, to create the Knights Landing core. List of some of these changes are below.

- Added four threads per core to provide higher-throughput performance.
- More than doubled the number of inflight instructions, that is, the Reorder Buffer (ROB) size, to 72 slots, to improve the single thread performance.
- Added vector instructions (AVX, AVX2, AVX-512) support.
- Added two VPUs to the main out-of-order pipeline.
- Increased size of the L1 data cache (DL1) to 32 KB to match the DL1 size in Knights Corner.
- Added two 64B loads ports in the DL1 to feed the two VPUs.
- Increased sizes for TLBs and added 1 GB page support.
- Implemented gather/scatter engine to insert the gather loads and scatter stores without consuming fetch and decode slots in the pipeline.
- Added support to complete quickly the accesses that are unaligned or split across multiple cache-lines.
- Doubled the L2 cache bandwidth to 1 cache line read and ½ cache line write per cycle.

Fig. 4.3 shows the block diagram of Knights Landing core and VPU. It is divided into five units: (1) front-end unit (FEU), (2) allocation unit (AU), (3) integer execution unit (IEU), (4) memory execution unit (MEU), and (5) VPU. While the core is 2-wide from the point of view of decoding, allocating, and retiring operations, it is capable of executing up to six operations per cycle, namely, two integer, two FP, and two memory operations. Fig. 4.4 shows a table with sizes for important structures in the core and how these structures are divided among different threads. Later in the chapter, we provide a detailed description of the core's threading support. The division of structures among threads will become clearer after reading that description.

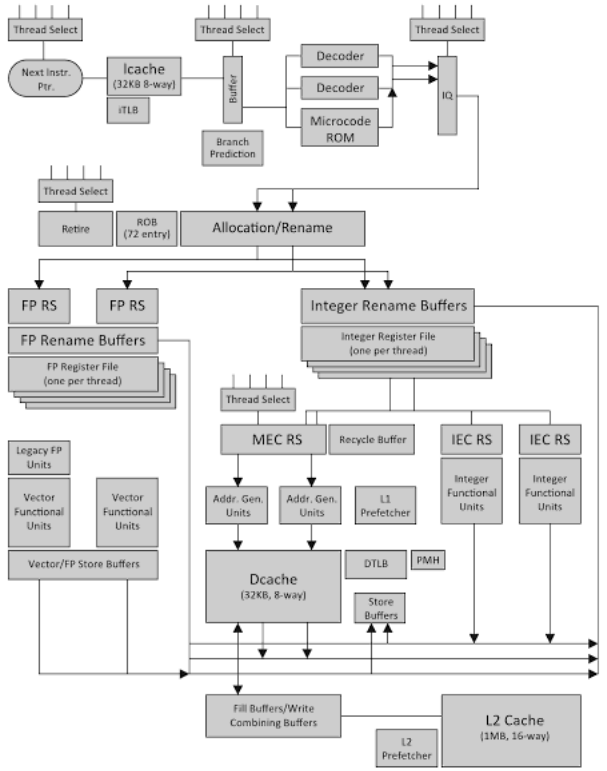


FIG. 4.3

Knights Landing core microarchitecture.

Front-end unit

The FEU is responsible for fetching instructions, decoding them, predicting branches, breaking instructions into simpler *micro-operations* (μops) and delivering them to the AU. Converting instructions into a flow of μops allows handling complex instructions without building complex hardware. Most instructions convert into single μop, however.

The process begins with the next instruction pointer block sending the next instruction to be fetched to the *Instruction Cache* (*Icache*). There is a thread selector at this point which decides which thread to pick for the next instruction fetch. Instruction

Structures	Sizes	Thread Sharing Policy
Instruction L1 Cache	32 KB, 8-way	Shared
iTLB	48	Shared
Instruction Queue	32	Dynamically Partitioned
ReOrder Buffer (ROB) Size (Out of Order Window)	72	Dynamically partitioned
Rename Buffers (RB)	72	Dynamically Partitioned
Register File (RF)	# of x86 architecture registers	Replicated
Store Data Buffers	16 entries	Dynamically Partitioned
Integer Reservation Station	12 x2	Dynamically Partitioned
Memory Reservation Station	12	Dynamically partitioned
Floating-point Reservation Station	20 x 2	Dynamically Partitioned
Data L1 Cache	32KB, 8-way	Shared
uTLB	64	Shared
DTLB	4K – 256 1M - 128 1G - 16	Shared
Inflight Gathers/Scatters Table	4	Dynamically Partitioned
Fill Buffers (outstanding misses per core)	12	Shared
L2 Cache	1MB, 16-way	Shared
Outstanding misses per tile	48 Reads and 32 Writebacks	Shared
L2 Prefetch Streams	48	Shared

FIG. 4.4

Table with sizes and thread-sharing policies for important structures.

bytes are read from the 32 KB, 8-way associative, Icache and written in an intermediate buffer. In parallel, the 48-entry *Instruction TLB* (*ITLB*) is accessed to obtain virtual to physical address translation. The thread is stalled in case of Icache or ITLB miss until the miss is resolved and other threads are selected for instruction fetch.

Instructions are selected from the intermediate buffer for decode. An intelligent thread selector ensures that instructions from all the active threads are selected in a fair manner, while taking into account the ability of that thread to make forward progress. Up to two instructions can be decoded every cycle. If an instruction is a complex instruction (i.e., multi-μop), then the *Microcode ROM* is accessed to obtain the corresponding μop flow for that instruction. Eventually, up to two μops are written per cycle in the *Instruction Queue* (*IQ*).

The *branch prediction unit* provides prediction on direction and target of branches. The unit consists of several types of predictors: Branch target buffer, Return Stack Buffer, Indirect predictor, and an enhanced form of gskew predictor.

Allocation unit

The AU is responsible for preparing μops for out-of-order execution. It assigns the necessary pipeline resources required by μops, such as ROB entries, Rename Buffer (RB) entries, Store Data buffers, gather/scatter table entries, and Reservation Station (RS) entries. It also renames the logical register sources and destinations in the μops to the RB entries. The RB provides storage for the results of in-flight μops until they retire, at which time these results are transferred to the architectural Register File (RF). After assigning the required pipeline resources and renaming the registers, the AU sends the μops to one of the three execution units, Integer, Memory, or Vector Processing, based on their type.

The AU is 2-wide, meaning it operates on two μops every cycle. It reads two μops from the IQ every cycle. A thread selector decides which thread is read from the IQ. The μops that do not have all the resources they need in order to proceed (i.e., a ROB entry, an RS entry, store data buffer entries) are stalled in the allocation until those resources become available. This does not prevent other threads, which may have available resources, from proceeding. The thread selector takes the stalls into account when reading μops from the IQ. This ensures we keep the downstream pipeline busy while some threads may be blocked for resources in AU.

The AU writes up to two μops in Integer, FP, or Memory RS per cycle.

Integer execution unit

The IEU executes integer μops, which are defined as those that operate on general-purpose registers R0–R15 (i.e., RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8…R15). There are two IEUs in the core. Each IEU contains 12-entry RS that issues one μop per cycle. The Integer RSes are fully out-of-order in their scheduling. Most operations have 1-cycle latency and are supported by both IEUs, but a few operations have 3- or 5-cycles latency (e.g., multiplies) and are only supported by one of the IEUs.

Memory execution unit

The MEU executes memory μops and services fetch requests for I-cache misses and ITLB misses. Up to two memory operations, either a load or a store, can be executed in the MEU per cycle. Memory operations are issued in-order from the 12-entry memory RS, but they can execute and complete out-of-order. The μops that were not able to complete successfully are placed into the Recycle Buffer and reissued to the MEU pipe once their conflict conditions are resolved. Completed loads are kept in the memory ordering queue, to maintain consistency, until they are retired. While stores are kept in the store buffer after address translation, they can forward data to any dependent loads. Stores are committed to the memory in the program order, one per cycle.

The 64-entry, 8-way set-associative, L1 μTLB is backed up by the 256-entry, 8-way set-associative, L2 DTLB. The DTLB also contains an 8-way, 128-entry table for 2 MB pages and a fully-associative, 16-entry table for 1 GB pages. The writeback, nonblocking, 32 KB, 8-way set-associative DL1 supports two simultaneous 512-bit reads and one 512-bit write, with a load-to-use latency of four cycles for integer loads and five cycles for FP loads. The L1 hardware prefetcher monitors memory address patterns and generates data prefetch requests to the L2 cache to deliver cache lines in advance.

The MEU supports unaligned memory access without any penalties and supports accesses that are split across two cache lines with a two-cycle penalty. The fast unaligned and cache split accesses help workloads where memory accesses may not always be aligned to natural data type sized boundaries.

The MEU supports 48 virtual address bits and 46 physical address bits (up to 39 physical bits for addressing cacheable memory and the rest for uncacheable memory) to provide the large memory addressing capability commensurate with a standalone, standard Intel® Architecture (IA) processor.

The MEU contains specialized logic to handle gather and scatter instructions efficiently. A single gather/scatter instruction can access multiple memory locations. In Knights Landing, such multiple accesses are generated by a special engine close to the L1 cache pipeline, instead of being a series of μops that flow through the entire core pipeline. This allows for executing the gather and scatter instructions while consuming minimal resources in rest of the core (e.g., FEU, AU, RSs, and Retire), allowing other types of μops to make forward progress in parallel.

Vector processing unit

The VPU is the vector and FP arithmetic execution unit of Knights Landing and is responsible of providing support for x87, MMX, SSE, AVX, and AVX-512 instructions, as well as integer divides. There are two VPUs connected to the core. These are tightly integrated into the pipeline, with AU dispatching instructions directly into the VPUs. The VPUs are mostly symmetrical, and each is able to provide a steady-state throughput of one AVX-512 instruction per cycle, providing a peak of 64 SP or 32 DP FP operations per cycle from the pair of VPUs available per core (i.e., 2 VPUs per core, 1 AVX-512 instruction per cycle per VPU, FMA offers 16 DP, or 32 SP operations per instruction using AVX-512 registers that are 8 DP or 16 SP wide, $2 \times 1 \times 16 = 32$ DP or $2 \times 1 \times 32 = 64$ SP). One of the VPUs is extended to provide support for the legacy FP instructions, such as x87, MMX, and a subset of byte and word SSE instructions.

Each VPU contains a 20-entry FP operation RSs that issues out-of-order one μop per cycle. The FP RS are different from the IEU RS and MEU RS in that they, to help reduce their size, do not hold source data; the FP μops read their source data from FP RB and FP RF after they issue from the FP RS, spending an extra cycle between RS and execution compared to integer and memory μops. Most FP arithmetic operations have a latency of six cycles, while other arithmetic operations have a latency of two or three cycles, depending on the operation type.

The VPU also supports the new AVX-512 transcendental and reciprocal instruction extensions, that is, AVX-512ER, and vector conflict detection instruction extensions, that is, AVX-512CD, introduced in Knights Landing.

THREADING

The core supports four threads of execution. These are implemented as *hyper threads* where the instructions from all threads flow through the pipeline simultaneously. Thread-selection points in the pipeline select from which thread instructions move forward. This decision is made each cycle and takes into account such factors as thread fairness, instruction availability, thread-specific stalls, and other constraints, to help maximize the utilization of hardware.

Fig. 4.4 details how the threads share some of the key core structures. Structures are shared by threads, dynamically partitioned among them, or replicated for each thread. Dynamically partitioned structures readjust their partitions as threads go to sleep or are awoken so that the active threads get to use the full physical structure. For example, when one thread is active, the thread will use the full 72 entries in the ROB. However, when two threads are active, then each thread will use 36 entries, and when three or four threads are active, each thread will use 18 entries. Shared resources do not enforce partitioning; threads get shared resources on first-come-first-served basis. Replicated resources have dedicated copies for each thread, irrespective of whether they are active are not. Very few structures in the core are replicated per thread.

Four threads are not required to be active per core for performance. Threading can be turned off with BIOS settings, in which case there will be only one thread per core. But if threading is on, then it is a software decision about how many threads to start on a core. Software can start 1, 2, 3, or all 4 threads. Threads that are not active stay in HALT state or in monitor mwait state. In many cases, unlike Knights Corner, a single active thread per core is sufficient to utilize all the core resources. (Knights Corner required at least two active threads per core.) Using two active threads is often the best general choice as the out-of-order processing can often hide instruction and cache access latencies for many workloads. Four active threads per core can provide additional significant benefit for memory latency sensitive workloads and for applications with strong thread scaling and datasets that do not increase per thread.

As is explained more in Chapter 6, three active threads per core will generally underperform other options.

L2 ARCHITECTURE

On a Knights Landing tile, the two cores share a 16-way associative, 1 MB unified L2 cache (Fig. 4.1). Intratile coherency is maintained by the BIU, which also acts as the local shared L2 cache management unit (Fig. 4.2). Lines in the L2 cache are maintained in one of the MESIF states. The cores make requests to the BIU via a dedicated request interface to each core. Cacheable requests lookup L2 tags, while other requests are bypassed directly out to the mesh to be serviced by a specific CHA as determined by the built-in address hash.

Knights Landing implements a unique cache topology to minimize coherency maintenance traffic. The L2 cache is inclusive of DL1 but is not inclusive of IL1. The lines brought into IL1 fill into the L2, but when those lines are evicted from L2, the corresponding IL1 lines are not invalidated. This avoids invalidations due to Hot-IL1-Cold-L2 scenarios, where a line in active use in IL1 gets invalidated due to eviction of the corresponding line due to inactivity from L2. The "presence" bits per line are stored in L2 to track which lines are in active use in DL1. This information is used to filter the coherency probes for inclusive DL1 when not applicable. It also factors into L2 victim selection algorithm to minimize eviction of in-use lines.

BIU also contains a L2 Hardware Prefetcher that trains based on requests coming from the cores. The BIU supports up to 48 independent prefetch streams. Once a stream is detected to be stable, in either the forward or backward direction, prefetch requests are issued to successive cache lines in that stream (i.e., unit-cache line stride distance).

CLUSTER MODES

In this section, we describe the cluster modes on Knights Landing, which we introduced in Chapter 2, in further detail. Cluster modes are ways to divide the chip into separate virtual regions with the intention of keeping the on-die communications to be as local as possible. The goal is to lower the latency and increase the bandwidth of on-die communications. We support three primary cluster modes: (1) All-to-all mode, (2) Quadrant mode, and (3) Sub-Numa Cluster (SNC) mode. The only issue is performance; software will run in any cluster mode, and cache-coherency is always maintained across the entirety of a Knights Landing.

These modes can be best understood by observing how the data flow required to service an L2 cache miss would vary between the different modes. A typical L2 miss flow is shown in Fig. 4.5 (which is also the figure that describes all-to-all cluster mode). There are three main agents that participate in an L2 miss flow: (a) the tile that generates the miss, (b) the tag directory (as part of CHA) that "owns" the missing address and tracks whether any other tile on the chip has that address in its caches, and lastly, (c) the memory that supplies the data for servicing the miss. On an L2 miss, the tile sends the request to the CHA to check if any other tile on the chip has that address in its

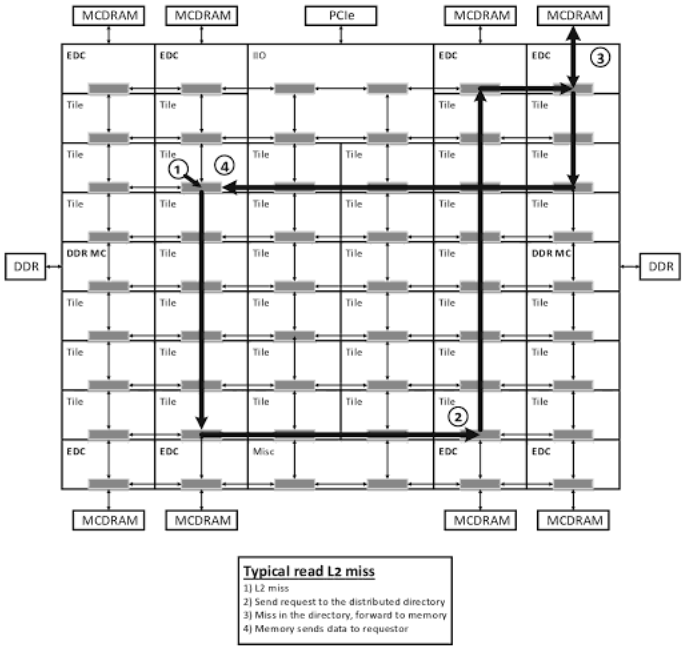


FIG. 4.5

An L2 cache miss flow example in all-to-all cluster mode.

caches. If the CHA finds that no other tile has the address cached, then it sends the request to the memory. The memory reads the data and sends it to the requesting tile. If the CHA finds that some other tile has the data in its caches, then it will send the request to that tile to send the data to the tile that had the miss (that case is not illustrated in the figure). The three cluster modes create different levels of "affinity" between the three agents (i.e., tile, CHA, and memory) to localize the communication between them.

The cluster modes are chosen via BIOS options and can only be set once during boot time. The cluster mode cannot be changed again until after the next system reset.

ALL-TO-ALL CLUSTER MODE

The all-to-all cluster mode (Fig. 4.5) is the most general among all three modes. This mode will be lowest in general performance than the other memory modes, but it can be used with any DDR DIMM configuration. It is the only mode which can be used

when DDR DIMMS are not identical in capacity. In this mode, there is no affinity between the tile, CHA, and the memory. The addresses are uniformly hashed across all CHAs and over all memory, independently. Any tile may request data at an address that is tracked by a CHA in any part of the chip, and the memory location may reside in any part of the chip. A L2 miss transaction will traverse longer distances, on average, across the chip to get the data in all-to-all mode.

The purpose of this mode is to offer flexibility in supporting any combination of allowed DDR DIMM configurations. This mode can route physical addresses from any tile to any CHA, to any memory controller. While not the default, this mode is automatically chosen in the event of any memory asymmetries or other configuration irregularities are detected in the system at boot time.

QUADRANT CLUSTER MODE

The quadrant mode (Fig. 4.6) will be the mode in which most software would want to run. This mode is expected to be the default mode set in the BIOS although all-to-all mode will be entered if DDR DIMMs are not found to be identical in capacity. This mode is transparent to software, in that software does not need to take any specific actions to benefit from this mode. In this mode, we establish affinity between the two of the three agents, that is, the CHA and the memory. The chip is divided into four virtual quadrants. The memory addresses are hashed such that the CHA that has the tag directory for a memory address is in the same quadrant as the memory where data for the address resides. This way a memory access transaction, as shown in Fig. 4.6, travels locally within the same quadrant after checking the tag directory in the CHA to reach the memory. This reduces the latency of memory accesses compared to all-to-all mode. Since reduced latency allows hardware buffers involved in memory access transactions to be freed sooner, the quadrant mode allows more memory transactions to occur in a given time; therefore it offers higher bandwidth than all-to-all mode.

The only requirement for using quadrant mode is that the memory configuration be symmetric, that is all DDR DIMMS be of the same capacity. This is needed for the address hashing to work to enable the quadrant division. Once booted in quadrant mode, software does not need to do anything special to benefit from this mode.

SNC-4 MODE

The SNC mode (Fig. 4.7) provides the shortest latency and most localized communications among all modes. It extends the two-agent affinity in the quadrant mode to three-agent affinity which includes all three agents: the tile, the CHA, and the memory.

In this mode, each quadrant is exposed as a separate cache-coherent NUMA cluster that is visible to operating system, via BIOS ACPI tables. The memory addresses are distributed such that a continuous region of memory is mapped to each cluster and the tag directories for memory addresses mapped in a cluster are in CHAs that are also all within the same cluster.

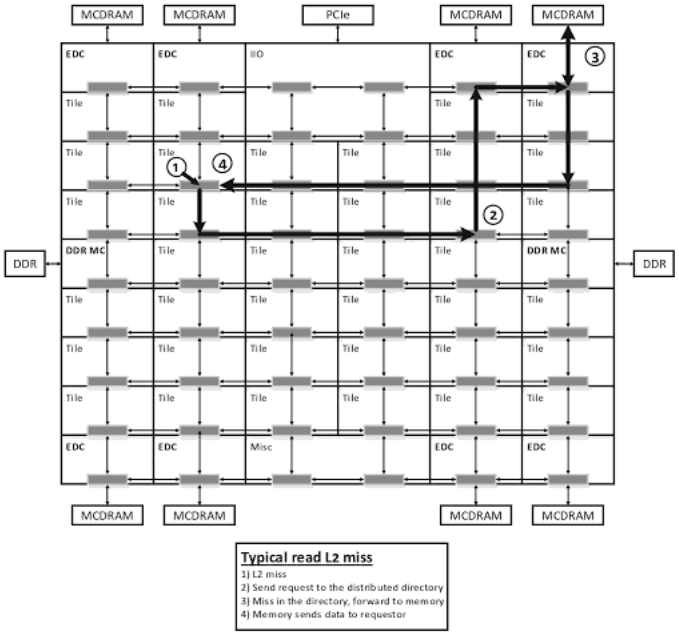


FIG. 4.6

An L2 cache-miss flow example in quadrant cluster mode.

A NUMA-optimized software (i.e., software that allocates memory from the same cluster where it runs) will see all its memory access transactions complete locally so that they will stay entirely within the same cluster as shown in Fig. 4.7. Such memory accesses from a tile will go to a CHA that is in the same cluster and, thereafter, will go to a memory which is also in that cluster. This local communication enables shortest communication latencies in this mode. As with quadrant mode, SNC-4 mode enables higher bandwidths due to better utilization of hardware buffers that support memory transactions resulting in higher bandwidth than all-to-all and quadrant mode.

Unlike quadrant and all-to-all modes, this mode is not entirely software transparent. The software needs to be NUMA optimized to benefit from this mode. The only issue is performance; software will work in any cluster mode including SNC. However, software that is not NUMA optimized is most likely to run better in quadrant mode.

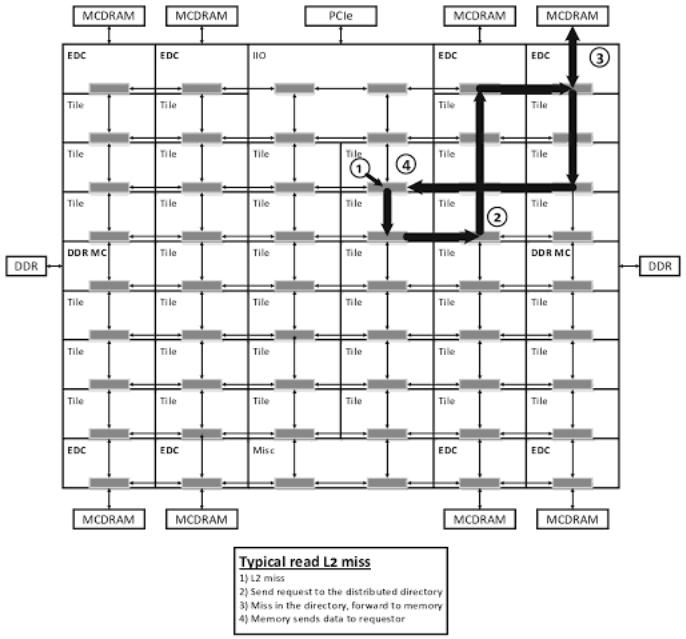


FIG. 4.7

An L2 cache-miss flow example in sub-NUMA cluster mode.

Knights Landing in SNC-4 mode with four clusters will look analogous to a four socket Intel Xeon processor-based machine (Fig. 4.8) in terms of NUMA nodes and memory allocation. In fact, any question on how a particular scenario will work in SNC-4 mode on Knights Landing can be answered by asking how that same scenario will be handled in a multisocket Intel Xeon processor-based machine.

HEMISPHERE CLUSTER AND SNC-2 MODES

For the quadrant and SNC-4 modes, we described the ability to divide Knights Landing into four parts (quadrants). Knights Landing also supports variations that divide into hemispheres (two parts) instead of quadrants (four parts). Everything described about quadrant and SNC-4 modes is identical, excepting that the Knights Landing is logically divided into two regions instead of four. The average latencies will be higher due to the increased average distances for resolving memory requests.

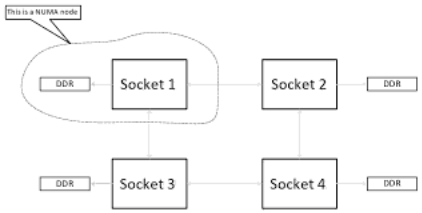


FIG. 4.8

A four socket with four NUMA nodes using Intel Xeon processors. This is analogous to Knights Landing with SNC-4 mode with four clusters.

Since the latencies are higher than quadrant and SNC-4 modes but lower than all-to-all mode, the memory bandwidth is higher than all-to-all mode but lower than quadrant and SNC-2 modes. The most interesting mode would seem to be the SNC-2 mode offering NUMA clusters equal to half the cores mimicking a dual socket processor platform, but the lower bandwidth probably makes this mode less interesting than quadrant or SNC-4 mode.

CLUSTER MODE SUMMARY

Fig. 4.9 summarizes the different characteristics of the three cluster modes in a table. The table in Fig. 4.9 also describes *memory interleaving* in each mode, which we expand upon in the following section.

MEMORY INTERLEAVING

Memory interleaving is a technique to spread out consecutive memory access across multiple memory channels, in order to parallelize the accesses to increase effective bandwidth. The interleaving granularity is always a cache line, which is 64 bytes for Knights Landing; this is the same size cache line used by all other current Intel processors. In this section, we describe how memory interleaving works with the various cluster modes.

In all-to-all, quadrant and hemisphere modes, addresses are uniformly distributed across the memory channels, as shown in Fig. 4.10. The distribution pattern is different for each mode as it depends on the specific hash function used in each mode to assign memory addresses to different CHAs. The net effect is that the addresses are uniformly distributed across the memory channels. In flat memory mode, contiguous ranges of memory are assigned to DDR and MCDRAM, respectively, with the MCDRAM range above the DDR range. The addresses in the DDR memory range are uniformly distributed among the DDR channels, while the addresses in the

Cluster Mode	Transaction Routing	Memory Interleaving	NUMA	DIMM (DDR4) Configuration
All-to-all	Originate from any tile, route to any tag directory, to any MC	Single address space uniformly interleaved across all MCs	None	All configurations are possible regardless of DDR population characteristics
Quadrant/ hemisphere	Originate from any tile, to a tag directory colocated in the same quadrant/hemisphere as the memory	Single address space uniformly interleaved across all memory channels	None	Equally populated equal capacity DDR DIMMs installed
SNC-4/ SNC-2	Tile, tag directory, and target MC are all colocated within the same cluster. SNC-4: 4 clusters SNC-2: 2 clusters	Contiguous address space per cluster, uniform interleaving within cluster	SNC-4: 4 nodes SNC-2: 2 nodes	Equally populated equal capacity DDR DIMMs installed

FIG. 4.9

Cluster mode summary.

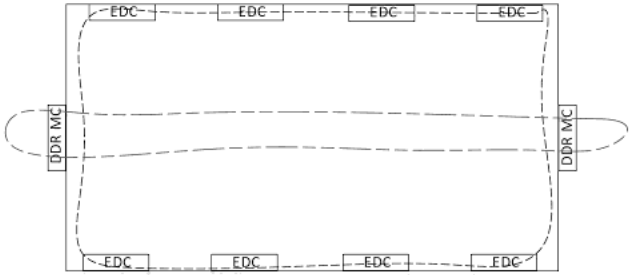


FIG. 4.10

Memory interleaving in all-to-all, quadrant, and hemisphere cluster modes with flat memory mode.

MCDRAM memory range are uniformly distributed among the MCDRAM channels, as shown in Fig. 4.10. In cache memory mode, since only DDR memory is visible to software (as MCDRAM is the cache), the entire memory range is uniformly distributed among the DDR channels. The address distribution in hybrid memory mode is

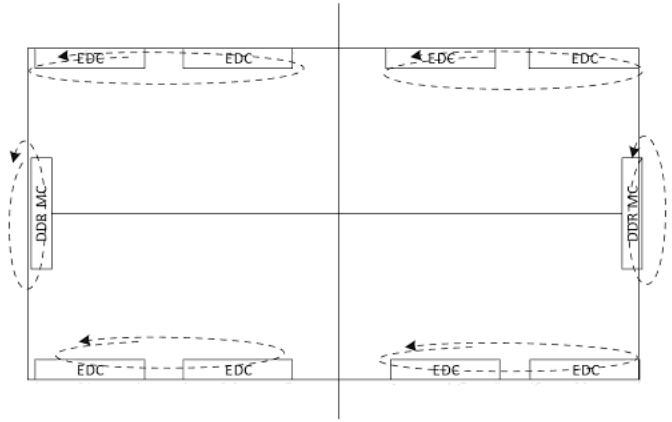


FIG. 4.11

SNC-4 memory interleaving for flat memory mode.

similar to the distribution patterns of flat memory mode and cache memory mode for the address ranges that are mapped as flat and cache, respectively.

In SNC-4 and SNC-2 cluster modes, contiguous regions of memory are assigned to each cluster (also a NUMA node) and are cache line interleaved among the memory channels within that NUMA node, as shown in Figs. 4.11 and 4.12, respectively. In flat memory mode, the memory region mapped to each SNC cluster is divided into two contiguous portions, one for MCDRAM and other for DDR. These portions are interleaved over the MCDRAM and DDR channels that are in that cluster (for SNC-4, since DDR channels are not entirely within a cluster, the interleaving is over all the three channels that are closer to the cluster; this looks similar to SNC-2). In cache memory mode, the addresses are interleaved over the DDR channels, since MCDRAM is a cache and is hidden behind the memory. In hybrid mode, the address interleaving will be similar to flat and cache mode based on whether the memory region is mapped to flat or cache portion of the memory.

MEMORY MODES

As mentioned in Chapter 2, Knights Landing supports three memory modes for configuring the MCDRAM and DDR memory (Fig. 4.13): (1) cache mode, (2) flat mode, and (3) hybrid mode. In cache mode, MCDRAM is configured as a cache for the DDR memory; in flat mode, MCDRAM is configured as memory just like DDR

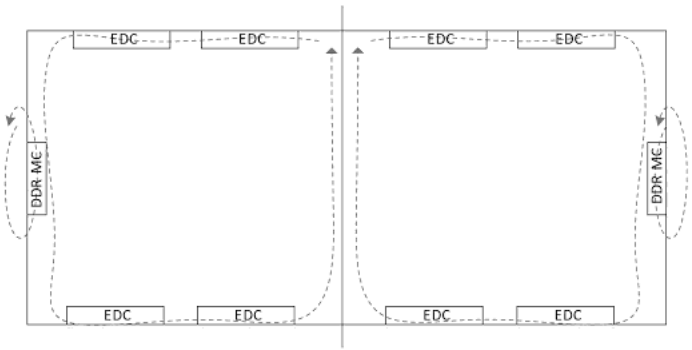


FIG. 4.12

SNC-2 memory interleaving in flat memory mode.

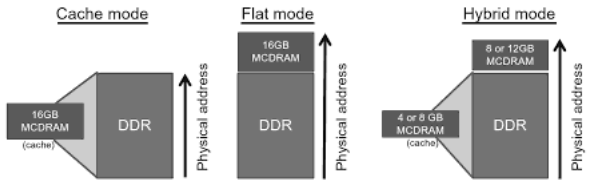


FIG. 4.13

Three modes for MCDRAM usage and how they related to DDR memory.

and is part of the same address space; and in hybrid mode, a portion of MCDRAM is configured as cache and the remainder is configured as flat in the same address space as DDR. Like cluster modes, the memory modes are also selected at boot time via BIOS options and can only be set once during boot time. The memory mode cannot be changed again until the next system reset.

CACHE MODE

In cache mode, MCDRAM is configured as a cache for the DDR memory. The MCDRAM cache is completely hardware managed, requiring no software enablement. Legacy, unmodified software can use this mode and benefit from the high bandwidth provided by MCDRAM. The benefits will, of course, depend on the hit rate observed by the software.

MCDRAM cache is a direct-mapped cache with 64-byte cache lines. The tags are stored in the MCDRAM, as part of extra bits available in each cache line. These extra

bits are read at the same time as the cache line, thereby allowing tag and data to be read on the same access, and hence, enabling the determination of a hit or miss in cache without needing a separate read access. Since MCDRAM size is much larger than traditional caches, in most cases there should be no significant change in conflict misses due to the direct-mapped policy (see Chapter 6 for further discussion on direct-mapped policy).

In this mode, all requests first go to the MCDRAM for a cache look-up and then, in case of a miss, a request is sent to DDR. The memory accesses will never bypass the MCDRAM and go to the DDR. On a miss, the data is read from the DDR and sent to the MCDRAM, to fill the cache, and the requesting tile, simultaneously.

MCDRAM cache is a *memory-side cache*, as opposed to *CPU-side caches* such as an L1, L2, or last level caches, in that a memory-side cache is closer to memory in terms of its properties as compared to CPU-side caches on the cores or tiles. It acts more like a high-bandwidth buffer sitting on the way to memory, exhibiting memory semantics, instead of a cache sitting closer to the cores. Unlike caches on the cores or tiles, MCDRAM cache does not need to be snooped by external transactions since any access to memory first goes through MCDRAM cache. An "uncacheable" memory type that does not allocate in core cache will allocate in MCDRAM cache, since MCDRAM cache is but a part of memory, just higher bandwidth.

MCDRAM cache is made inclusive of all modified lines in the L2 caches. In other words, if a cache line resides in a modified state in an L2 cache, it must also be in MCDRAM. This ensures that when a modified line is written back from an L2 cache, there is no need to query the MCDRAM cache first before writing the line into it. There is no risk of overwriting a different line, since with the modified-inclusive property, the line being written back is guaranteed to be present in the MCDRAM cache. To maintain this modified-inclusive property, before a line is evicted from MCDRAM, a snoop is sent to check if a *modified* copy of that line exists in L2 cache and, if so, downgrade it from *modified* to *shared* by forcing a writeback of the modified line. The line does not get evicted from the cache.

As mentioned before, the MCDRAM cache mode is completely transparent to software and will work out of the box on unmodified code. The benefits derived from the MCDRAM cache will depend on the hit rate seen by software. The cache works well for many applications, but for applications that do not show good hit rates in the MCDRAM, the other two memory modes provide more control for applications to directly manage their use of MCDRAM (also see Chapter 3).

FLAT MODE

In flat mode, both MCDRAM and DDR are presented as regular memory mapped in the same system address space. There is no difference between the MCDRAM region and the DDR region in terms of semantics; both act as regular load/store memory. Unlike cache mode, flat mode is not software transparent. The software needs to explicitly allocate best suited data in the MCDRAM for it to benefit from the provided high bandwidth. However, once data is allocated in MCDRAM, software will

FIG. 4.14

Analogy between Knights Landing (KNL) in flat mode and an Intel Xeon processor-based machine with two sockets. Both have two NUMA nodes.

see stable high bandwidth access to that data, since there is no dependency on "hit" rate like in cache mode.

For software to allocate explicitly into MCDRAM, we enabled a way to "name" the MCDRAM region such that we keep software portable to processors that do not provide MCDRAM. The naming scheme exposes MCDRAM and DDR as two separate NUMA nodes. If a Knights Landing system has only DDR or only MCDRAM, then there would be only one NUMA node. SNC-2 and SNC-4 complicate this slightly by multiplying the number of NUMA nodes by two or four, respectively. This is covered in more detail toward the end of this chapter in the *Interactions of Cluster and Memory Modes* section.

The software can then allocate into MCDRAM referring to the NUMA node assigned to MCDRAM. Knights Landing in flat mode, with two NUMA nodes, looks analogous to a two-socket Intel Xeon processor-based machine with two NUMA nodes (Fig. 4.14). In fact, almost in all cases we can answer a question on how a certain memory allocation scenario would work in flat mode by referring to how a similar scenario would work for the two NUMA nodes in a two-socket Intel Xeon processor-based machine.

As described in Chapter 3, we provide the *memkind* software library that contains functions for managing MCDRAM memory. This library uses the NUMA node information to do the allocation into MCDRAM. The code written with this library remains portable, in that the functions resort to standard memory allocation when code is run on a system that does not have MCDRAM memory or is in MCDRAM cache mode.

There are two modes that are degenerate forms of the flat mode: MCDRAM-only mode and DDR-only mode. In these cases, there is only one type of memory present in the system, that is, MCDRAM or DDR, but not both. Since these modes are similar to standard systems with one type of memory, no software modification is needed; programs will run by allocating all their memory in MCDRAM or DDR, whichever is available.

HYBRID MODE

The hybrid mode, as the name suggests, is a combination of the cache mode and the flat mode. A portion of MCDRAM is configured as a cache (25% or 50%) and the remaining is configured as flat memory (75% and 50%). This is ideal for applications

	Capacity	Bandwidth	Idle Latency
MCDRAM	Up to 16GB	Up to 450GB/sec	Approx. 150ns
DDR	Up to 384GB	Up to 90GB/sec	Approx. 125ns

FIG. 4.15

MCDRAM and DDR characteristics.

that benefit from general caching and can also take advantage of high bandwidth memory by storing critical or frequently accessed data in flat memory. The portion of MCDRAM that is used as cache serves all of DDR memory. Using the cache portion of MCDRAM does not require any application changes. The flat memory portion of MCDRAM will get exposed to software as a separate NUMA node, like in flat mode, and the application will need to use the memkind library or other NUMA-based allocators to use this portion of MCDRAM. As with flat mode, SNC-2 and SNC-4 complicate this slightly by multiplying the number of NUMA nodes by two or four, respectively. This is covered in more detail toward the end of this chapter in the *Interactions of Cluster and Memory Modes* section.

Hybrid is an excellent choice to allow benefiting from cache mode, without software changes, but allowing experimentation with flat mode to get better performance. It will also be useful for systems that are used by variety of users with different types of applications, some that may benefit from flat mode optimization, while others may do fine with cache mode. Since changing modes requires a reboot, it may be convenient to boot the system in hybrid mode, thereby providing both cache mode and flat mode support to the users (albeit with a smaller capacity for each compared to all cache or all flat mode).

CAPACITY, BANDWIDTH, LATENCY

Fig. 4.15 compares the capacity, streams bandwidth, and the idle latencies of MCDRAM and DDR memories.

INTERACTIONS OF CLUSTER AND MEMORY MODES

The cluster modes and the memory modes are selectable completely independently of each other; any of the three cluster modes can be selected with any of the three memory modes. There are however some interactions between SNC cluster modes and the hybrid/flat memory mode that are worth understanding. These interactions arise because both these modes use NUMA enumeration: SNC uses NUMA nodes to partition the chip, while the flat memory mode uses NUMA nodes to distinguish between DDR and MCDRAM memory. Thus, while the SNC-4 mode exposes four

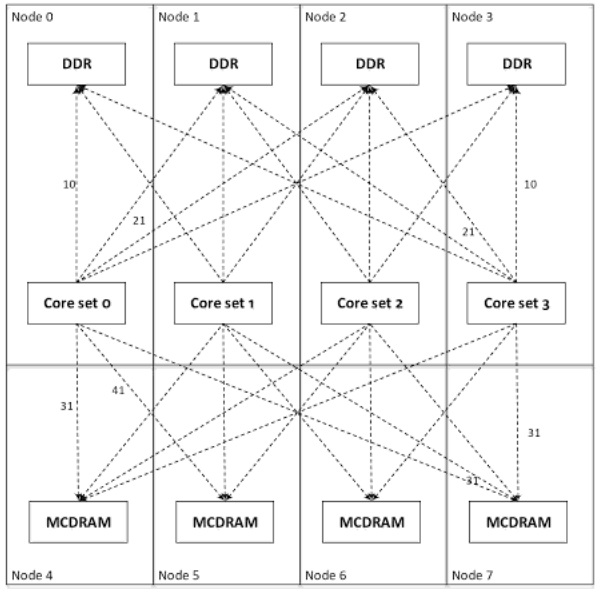


FIG. 4.16

Flat mode with SNC4.

NUMA nodes and the flat memory mode exposes two NUMA nodes (assuming MCDRAM and DDR are both present), when used together they expose eight NUMA nodes. This is illustrated in Fig. 4.16. The arrows with values 10, 21, 31, and 41 show the default *distance* values between the various NUMA nodes as programmed in BIOS. These distance settings determine the affinity between different NUMA nodes.

The hybrid mode and SNC-4 interaction will look the same as in Fig. 4.16 from the NUMA node point of view. The hybrid mode has a portion of MCDRAM as cache, but that does not change the NUMA topology. Similarly, the interactions of flat mode and hybrid mode with SNC-2 will result in four NUMA nodes in all: two for the memory and two for the SNC-2 clusters. Fig. 4.17 enumerates the key cluster and memory mode combinations and shows their various attributes.

BIOS Setup Option			Additional Information	
Cluster Mode	**Memory Mode**	**MCDRAM Cache Size**	**NUMA Nodes**	**DIMMs Configuration Allowed**
All-to-all	**Cache**	MCDRAM 100% cache, 0% as memory	None	All configurations are possible regardless of DDR population characteristics.
	Flat	MCDRAM 0% cache, 100% as memory	Total 2: 1 node for MCDRAM, 1 node for DDR	
	Hybrid	MCDRAM 25% or 50% as cache, and 75% or 50% as memory		
Quadrant or Hemisphere	**Cache**	MCDRAM 100% cache, 0% as memory	None	Both memory controllers have same number of equal capacity DIMMs installed.
	Flat	MCDRAM 0% cache, 100% as memory	Total 2: 1 node for MCDRAM, 1 node for DDR	
	Hybrid	MCDRAM 25% or 50% as cache, and 75% or 50% as memory		
SNC-4 (with notes for SNC-2)	**Cache**	MCDRAM 100% cache, 0% as memory	Total 4: 4 clusters, each with 1 node for DDR (SNC-2: total 2)	Both memory controllers have same number of equal capacity DIMMs installed.
	Flat	MCDRAM 0% cache, 100% as memory	Total 8: 4 clusters, each with 1 node for MCDRAM and 1 node for DDR (SNC-2: total 4)	
	Hybrid	MCDRAM 25% or 50% as cache, and 75% or 50% as memory		

FIG. 4.17

Memory and cluster mode combinations, assuming systems with MCDRAM and DDR both.

SUMMARY

In this chapter, we described the details of the Knights Landing architecture. We dove deeper into the details of the tile architecture, described the various cluster and memory modes, and explained the interactions between them.

FOR MORE INFORMATION

- "Knights Landing: 2nd generation Intel® Xeon Phi™ Product," Hot Chips Special Issue of IEEE Micro Magazine, 36.2 (2016): 34–46. http://doi.ieeecomputersociety.org/10.1109/MM.2016.25.

CHAPTER

Intel Omni-Path Fabric

5

What is new with Knights Landing in this chapter?

Some versions of Knights Landing integrate Intel® Omni-Path Fabric in the same package (a.k.a. "on-package").

The Intel® Omni-Path Architecture (Intel® OPA) interconnect fabric design enables a broad class of multiple node computational applications requiring scalable, tightly coupled processing, memory, and storage resources. Furthermore, options for close "on-package" integration between Intel® OPA family devices, Intel® Xeon® processors, and Intel® Xeon Phi™ (Knights Landing) processors enable significant system-level packaging and network efficiency improvements. When coupled with recent user-focussed open standard Application Program Interfaces (APIs) developed by the Open Fabrics Alliance (OFA) Open Fabrics Interface (OFI) workgroup, Host Fabric Interfaces (HFIs; known as NICs in Ethernet), and switches in the Intel OPA family, systems are optimized to provide low latency, high bandwidth, and high message rate needed by large-scale high-performance computing (HPC) applications. Intel OPA provides important innovations for a multi-generation, scalable fabric, including link layer reliability, extended fabric addressing, and optimizations for many-core processors such as Knights Landing. High-performance datacenter needs are also a core Intel OPA focus, including: link-level traffic flow optimization to minimize datacenter-wide jitter for high-priority packets, robust partitioning support, quality of service (QoS) support, and a centralized fabric management system.

OVERVIEW

Intel OPA delivers a next-generation fabric with heritage from the Intel® TrueScale product line and the Cray Aries interconnect. See the *For More Information* section, at the end of this chapter, for resources on these earlier interconnects. Intel OPA integrates fabric components with processor and memory components enabling the low latency, high bandwidth, dense systems required for next-generation datacenters. Fabric integration takes advantage of processing, cache and memory subsystems, and communication infrastructure locality enabling rapid hardware innovation. Enhancements found in the first generation of Intel OPA include higher overall bandwidth, lower latency, and denser form factor systems.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00005-3

Intel OPA-based first-generation products focus on HPC systems including large supercomputing environments. Nevertheless, Intel OPA is generally applicable to any class of datacenter-level computation requiring scalable, tightly coupled, CPU, memory, and storage resources. Intel OPA defines an Open Systems Interconnect (OSI) layers 1 and 2 architecture that provides connectivity between elements in energy efficient supercomputer systems (e.g., based on Intel Xeon Phi processors), mission critical enterprise computer systems (e.g., based on Intel Xeon processors), and inexpensive datacenter servers (e.g., based on Intel® Atom™ processors).

To enable the largest scale systems in both HPC and the datacenter, fabric reliability is substantially enhanced by combining the link-level retry typically found in HPC fabrics, with the conventional end-to-end retry used in traditional networks. Layer 2 network addressing is extended to account for systems with over 10 million endpoints, thereby enabling use on the largest scale datacenters for years to come. To enable support for a breadth of topologies, Intel OPA provides mechanisms for packets to change Virtual Lanes (VLs) as they progress through the fabric. In addition, higher priority packets are able to preempt lower priority packets to provide more predictable system performance, especially when multiple applications are running simultaneously. Finally, fabric partitioning is provided to isolate traffic between jobs or between users.

The software ecosystem includes three key APIs.

1. The OFA OFI represents a long-term direction for high-performance, user-level and kernel-level network APIs.
2. The Performance-Scaled Messaging (PSM) API provides HPC focussed transports and an evolutionary software path from the Intel TrueScale fabric.
3. OFA Verbs provides support for existing applications and includes extensions to support the Intel OPA fabric manager.

Higher level communication libraries, such as the Message Passing Interface (MPI — see Chapter 15), and Partitioned Global Address Space (PGAS — see Chapter 16) libraries are layered on top of these low-level OFA APIs. See the *For More Information* section for resources on these APIs and more in depth treatments of the software ecosystem supporting Intel® OPA.

HOST FABRIC INTERFACE

Each host is connected to the fabric via an HFI (Fig. 5.1). HFIs bridge between the semantics of the host processor and the semantics of the fabric. The HFI minimally consists of the logic necessary to implement the physical and link layers of the fabric architecture, such that a node can attach to a fabric and send and receive packets to other servers or devices. An HFI may include specialized logic for executing or accelerating upper layer protocols. An HFI must also support whatever logic is necessary to respond to messages from network management components.

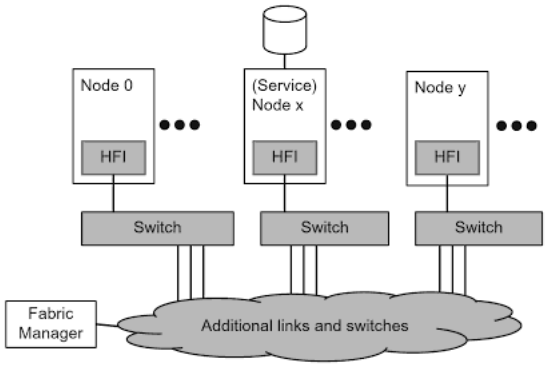


FIG. 5.1

Intel Omni-Path Architecture elements.

INTEL OPA SWITCHES

Intel OPA switches (Fig. 5.1) are OSI Layer 2 (link layer) devices, and act as packet forwarding mechanisms within a single Intel OPA fabric. Intel OPA switches are responsible for implementing QoS features, such as adaptive routing and load balancing. Switches also implement the Intel OPA congestion management functions. Switches are centrally provisioned and managed by the fabric manager (FM) software, and each switch includes a management agent (MA) to handle management transactions. Central provisioning means that switch configurations are programmed by the FM, including managing the forwarding tables to implement specific fabric topologies, configuring the QoS parameters, and providing alternate routes for adaptive routing. As such, all OPA switches must include MAs to communicate with the Intel OPA FM.

INTEL OPA MANAGEMENT

The Intel OPA fabric (Fig. 5.1) is centrally managed, and supports redundant FMs that provision and manage every device (i.e., HFIs, switches) in the fabric through MAs associated with those devices. The primary FM is an Intel OPA fabric software component selected during the fabric initialization process. The primary FM is responsible for: (1) discovering the fabric's topology; (2) setting up Fabric Identifiers and other necessary values needed for operating the fabric; (3) creating and populating the Switch-forwarding tables; (4) maintaining the Fabric Management Database; and (5) monitoring fabric utilization, performance, and error rates.

The fabric is managed by sending management packets over the fabric. These packets are sent *in-band* (i.e., over the same wires as regular network packets) using dedicated buffers on a specific VL (VL15). This specific VL for management may be configured to operate with or without flow control. Without flow control, management packets will be dropped if queue resources are not available at a port. End-to-end reliability protocols are used to detect dropped packets.

PERFORMANCE AND SCALABILITY

Large system cluster sizes continue growing because application performance demands are outpacing even Moore's law. See the *For More Information* section for information on the top 500 supercomputers. As a result, this increasing demand for performance has resulted in the transition from large Symmetric MultiProcessing (SMP) and vector-processing machines popular in the early 1990s into today's tightly connected microprocessor-based clusters interconnected via high-speed fabrics. Cluster sizes are continuing to grow significantly each year.

Given this trend, one of the objectives of Intel OPA fabric product family is to continually improve application performance and scalability. There are three key attributes (i.e., extreme message rates, low latency, and addressing) that drive these improvements which we will explain in more detail now.

EXTREME MESSAGE RATES

As applications are redesigned for increasing numbers of nodes and processing cores in a cluster, algorithm decomposition often results in more frequent communication and smaller message sizes. This trend requires fabrics and APIs that optimize for message rate.

To meet this need, Intel OPA is architected toward small message efficiency and performance. Intel OPA first-generation fabric achieves MPI application message rates up to 160 million packets per second. In addition, Intel OPA switches are capable of 100 Gbits/s (with 64 byte packets) full wire speed yielding packet rates up to 195 million packets per second per switch port.

While two-sided communication via the MPI API (see Chapter 15) is very popular for HPC applications today, another set of APIs and programming models (often referred to as one-sided or PGAS) are also emerging as well suited for certain types of graph analytic algorithms (see Chapter 16). PGAS APIs such as OpenSHMEM, Universal Parallel C (UPC), Coarray Fortran, and MPI-3 permit applications to efficiently access memory on remote nodes and often perform best with very high message rates that support smaller size data accesses.

Scalable message rate and latency is essential to support ever growing cluster sizes.

LOW LATENCY

Latency is very often as important as message rate. It is often difficult to implement applications to accurately prefetch data before it is needed, and in many cases, the data is not yet ready to fetch or send until it is immediately needed. This drives increasing importance for low data request latency.

While latency is easily measured at small scales, such as two-node tests, what really matters for applications is latency behaviors at scale as cluster size and application node count grow. To this end, Intel OPA fabric has been designed with scalable communication algorithms which ensure that the latency at scale is very similar to the observed latency for two nodes communication.

Having scalable latency is more than just building a fast wire speed. All contributions to latency must be considered. For interconnects, this includes wire speed, error detection and correction mechanisms, HFI hardware latency, HFI software latency, and processor cache behaviors in HFI software and applications.

Intel OPA fabric's first generation achieves latency of 100–110 ns in switches with full link layer resiliency enabled. Link layer resiliency is handled locally, per link, using an efficient link replay mechanism which adds no latency for successful packets and only adds a local link round trip (on the order of 100 ns) to recover the portions of a packet with detected errors.

ADDRESSING

The Intel OPA Link Layer (Layer 2) is designed for large-scale systems. OSI Layer 2 fabric packets enable 24-bit fabric addresses, as well as optimized formats for smaller systems. The OFA APIs generally hide these details from the application. Applications which are well behaved and make use of standard OFA APIs for address resolution and connection establishment will continue to function at scale on Intel OPA fabrics. It is important for programmers to avoid bypassing these standard mechanisms, for instance by exchanging LIDs directly. Using such mechanisms will not successfully scale and may not work with Intel OPA fabric extended addressing features in larger fabrics.

It is important to not bypass these standard addressing mechanisms. Doing so will degrade scaling and may not work in larger fabrics.

MULTICAST

Some applications make use of multicast, most notably IP applications doing Address Resolution Protocol (ARP), DHCP, or broadcast. Intel OPA fabric fully supports the OFA APIs for multicast, and Intel OPA switches implement full multicast packet handling using multicast spanning tree algorithms provided by the Intel OPA-centralized FM.

For scalable applications, using multicast mechanisms is generally discouraged. Multicast can have some negative side effects impacting the performance and scalability of the application:

- Multicast group join and leave operations require FM interactions and switch multicast routing table changes. On a large fabric, making frequent multicast membership changes can be inefficient and may outweigh multicast benefits. As such, multicast tends to work best when multicast groups are created and changed infrequently, such as creating IP multicast groups only on server boot and shutdown.
- Multicast traffic may go to uninterested nodes. Generic multicast groups, such as the IP broadcast groups, will include all nodes in the fabric. As such, it is likely that many nodes will not be interested in the message being sent. However, these nodes must interrupt their processor to handle the message. At small scale such interruptions are tolerable. However at large scale, such interruptions can become frequent and may inject processor and OS jitter effects delaying overall application progress.

TRANSPORT LAYER APIs

Intel OPA fabric provides three main transport layer APIs: OFA OFI, PSM, and Open Fabrics verbs. Above these transport layer APIs, numerous additional middleware and higher level APIs have been developed, such as MPI, OpenSHMEM, and sockets. The transport layer APIs are often used directly by I/O applications desiring maximum scalability and/or performance, such as parallel filesystems (e.g., Lustre, NFS).

OFA OPEN FABRIC INTERFACE

The OFI provides a general purpose software framework that is capable of handling a variety of fabric hardware and provides a standardized set of communication operations to higher code layers. The framework provides a library called libfabric that user-level applications may use.

The communication operations of libfabric are layered on top of the APIs mentioned later in this section. The message queue and tagged message queue operations are layered on PSM matched queue (MQ), and collectives can be synthesized in terms of PSM MQ operations. Put, get, atomics, and other operations designed to support PGAS or MPI3 RMA operations are layered on top of PSM active message (AM) implementation. Verbs semantics are layered on top of the standard Verbs access libraries. See the *For More Information* section for resources on this API.

PERFORMANCE-SCALED MESSAGING

PSM is a user-level library that provides a reliable fabric transport API for the Intel OPA HFI. The PSM API provides an MQ semantic that is a building block for MPI tag matching send and receive calls. The PSM API has been extended for the Intel

OPA architecture to provide 96 bits of tag matching supporting up to 32-bit user tags, up to 32-bit source rank information, and up to 32-bit communicator contexts. This allows significant scaling beyond the 64-bit tag matching provided by prior PSM implementations. The message-passing primitives provided by PSM are point-to-point. The Intel OPA PSM implementation is designed to scale to millions of MPI ranks.

Collective MPI operations can be synthesized from the point-to-point send and receive primitives using optimized algorithms that can be selected by parameters such as message size, collective communicator size, and topology. Additionally, PSM provides an active message (AM) API that can be used to implement arbitrary communication protocols using the AMs paradigm. This is used to implement a PGAS programming model using the OpenSHMEM API running over a GasNET conduit. See the *For More Information* section for resources on GASNet.

In the PSM library, short message send and receive is implemented using direct access to the send and receive hardware giving low latency and high message rate. For receive, PSM actively polls for arriving packets to eliminate host interrupt overheads. A registration caching scheme is used to reduce receive side overhead for typical MPI applications that reuse destination memory buffers often.

For each connection between a PSM sender and a PSM receiver, state is required to hold identification, status, sequencing, and control information. This is termed connection state or flow state. The Intel OPA HFI architecture holds no connection state in the HFI. This means that there are no hardware limits for capacity, caching, or scaling as the cluster size increases. Instead, all such connection state is held in the host benefiting from the host memory system capacity, cache capacity, and available bandwidth.

OPEN FABRICS VERBS AND COMPATIBILITY

The Intel OPA HFI supports a fully compliant OFA Verbs implementation. This includes support for reliable connected, unreliable datagram, and unreliable connected queue pairs. Shared receive queues are also supported. The standard user-level and kernel-level Verbs library interfaces are provided. All standard Verbs connection management protocols are supported as well as the 2 and 4 KB packet maximum transfer unit (MTU) sizes supported in InfiniBand implementations, Intel OPA fabric introduces an 8 KB MTU usable by a Verbs implementation to reduce the required packet rate for large messages. In addition, Verbs mechanisms are used for fabric management via the OFA user management datagram (umad) interface, with Intel OPA introducing a 2 KB management packet MTU.

The Intel OPA fabric implementation is partitioned between hardware capability in the HFI and host software. On the receive side, the incoming Verbs protocol packets are spread across receive contexts and processor cores using lower order bits from the queue pair number value and a mapping table. Interrupt coalescing moderates the host interrupt rate without overly delaying incoming latency sensitive packets. These features allow the Verbs performance to scale as more processor core resources are assigned to running the Verbs protocol code.

QUALITY OF SERVICE

Within Intel OPA fabrics, QoS features provide a number of capabilities, among them are job separation/resource allocation; service separation/resource allocation; application traffic separation within a given job; protocol (i.e., request/response) deadlock avoidance; fabric deadlock avoidance; traffic prioritization and bandwidth allocation; and latency jitter optimization by allowing traffic preemption.

SERVICE LEVELS

Within OFA APIs, the Service Level (SL) is the application identifier of a QoS domain. On the wire, a Service Channel (SC) is used as a per packet QoS identifier. Across each link, VLs are used to manage the hardware resources available. The FM handles the association of SLs to SCs and VLs.

TRAFFIC FLOW OPTIMIZATION AND PACKET INTERLEAVING

Traffic flow optimization allows Intel OPA fabric to support both high- and low-priority traffic on the same link while not sacrificing latency of the high-priority traffic. This feature permits storage and bulk I/O to be optimized using large packet sizes for maximum efficiency while latency sensitive compute traffic runs on the same links at higher priority.

Intel OPA fabric accomplishes this by permitting different packets on different VLs to be interleaved when they are sent across the link, allowing both higher link utilization, and lower latency for high-priority packets. A packet using a high-priority VL arriving at the link egress point can preempt and suspend an in-progress packet minimizing the latency of the high-priority packet. Once the high-priority packet is transmitted the suspended packet resumes. A low-priority packet can be preempted multiple times at an individual link egress point. The link egress point monitors the time a low-priority packet is preempted, and completes the packet's transmission if an FM-configured limit is exceeded.

CREDIT-BASED FLOW CONTROL

Intel OPA fabric uses credit-based link flow control. For a fabric port to send a packet, it must have sufficient flow control credits at the receiving port. Credit-based flow control means that data transfers on links are rigidly managed; there are no unauthorized data transfers, and it also means that Intel OPA fabric is so-called "lossless." In this case, lossless means simply that during normal operations packets are not dropped due to congestion.

Applications generally need not be aware of flow control, however scalable applications should be designed to be congestion aware and should avoid intentionally creating congestion trees for bulk data. In general, if an application wants to send large amounts of data to a shared server, it is better to send a small request

and let the server pull the data from the client when it is ready. An approach such as this is used in many scalable filesystem solutions. Under the covers, MPI and PSM use this mechanism for performance and scalability, but applications coded directly to OFI or verbs should also consider this approach.

If coding directly to OFI or Verbs, scalable applications can benefit from being congestion aware by using the same mechanisms MPI, PSM, and scalable filesystems use under the covers.

SECURITY

As cluster sizes grow, the need to share resources among multiple users, departments, and even multiple companies likewise grows. This need brings with it the implicit requirement of security. Security can be a matter of traditional separation and protection from malicious users, but it also can be a matter of protection from mistakes by well-meaning users.

PARTITION-BASED SECURITY

In Intel OPA fabric, Partitions are an isolation mechanism that operates at the Link Layer, with every Fabric packet being associated with a single partition. A partition provides isolation to the group of endpoints that are partition members for different types of traffic (i.e., all Transport Layers); however, this does not prevent a Transport Layer from providing finer grained security mechanisms. A partition in an Intel OPA fabric contains a group of Intel OPA endpoints. Communication is allowed within the partition, and is prevented with endpoints that are not in the partition. This allows partitions to be used to provide isolation between applications running on a fabric or between users on a fabric.

Individual endpoints may be identified as a full or limited member of a given partition. Full members are permitted to communicate with any partition member, but limited members are only permitted to communicate with full members. This mechanism permits the fabric to have shared services, such as management or a common global filesystem, while preserving isolation between nonservice partitions. Such services often require all end points to be members of the partition for that shared service. By making the providers of the shared service full members and the clients limited members, clients may access the service while still preventing clients from communicating directly to each other.

Each endpoint is a member of at least the management partition, and may be a member of multiple partitions. A typical endpoint will be a member of at least one application partition. When a Host node has multiple endpoints, each endpoint may have membership in different partitions. There are no architectural restrictions regarding which endpoints may be members of which partitions. For example, partitions may overlap and partitions need not be related to the physical topology of the fabric.

Partition security is enforced in the edge port of the switch connected to the HFI. The security mechanisms ensure that a source injects packets only into partitions of

which it is a member, and the packets are delivered only to endpoints in the same partition.

Partitions are created and managed by the Fabric Manager (FM). It initializes the registers and tables used to enforce partition security, and modifies them as partitions are redefined. Software running on the Host Nodes does not control the partition security registers and tables.

Within OFA APIs, the P_Key is the identifier for a security domain. There can be up to 32,767 unique partitions in a fabric. However, a given host can be a member of a modest number of P_Keys, in Intel OPA first-generation products, there is a limit of 32 P_Keys per switch port.

Fig. 5.2 shows a sample fabric with two application partitions (A and B). In this example, Nodes which are only in Partition A, such as Node 0, cannot communicate with Nodes in Partition B, such as Node y.

MANAGEMENT SECURITY

Intel OPA fabric has been designed with fabric management security as a first-order consideration. As such, Intel OPA fabrics define partition `0x7fff` as the management partition. In a typical deployment, all fabric management nodes will be full members of this partition and all remaining nodes will be limited members. The security for this partition applies to all management protocols between the fabric manager and MAs as well as name services and multicast membership queries between the clients and the fabric manager.

Intel OPA fabric by default has this partition secured. As such, applications should not hardcode the `0x7fff/0xffff` P_Key for use in communications with

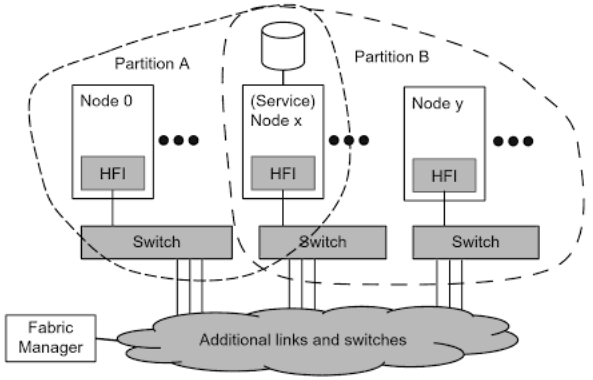


FIG. 5.2

Intel Omni-Path partitioning.

nonfabric management nodes. Instead, the appropriate `PathRecord` or multicast member record queries should be used to obtain the appropriate P_Key from the fabric manager.

Applications should not assume they can access management information from switches nor other servers. An application may access its own local management information (if nonroot umad access is enabled by the sysadmin). In general, use of such management information is discouraged.

Use `PathRecord` or multicast member record queries to obtain the appropriate P_Key from the fabric manager. Applications should not hardcode the `0x7fff/0xffff` P_Key for use in communications with nonfabric management nodes.

VIRTUAL FABRICS

Virtual Fabrics (vFabrics) provides a unifying concept by which Intel OPA Security and QoS features are configured and managed. It brings to the Intel OPA fabric many of the capabilities of Ethernet virtual LANs (VLANs). Using vFabrics, the administrator may slice up the physical fabric into many overlapping vFabrics. The administrator's selections determine how the configuration of the switches, HFIs, and links in the fabric is performed.

The goal of vFabrics is to permit multiple applications to be run on the same fabric at the same time with limited interference. The administrator can control the degree of isolation. As in the Ethernet VLANs, a given node may be in one or more vFabrics. vFabrics may have overlapping or completely independent membership. When IPoFabric (IPoIB) is in a fabric, typically each vFabric using IPoFabric represents a unique IP subnet, in the same way a unique subnet is assigned to different VLANs.

Each vFabric can be assigned QoS and security policies to control how common resources in the fabric are shared among vFabrics.

Additionally, as in the Fibre Channel Zoning, vFabrics can be used in an Intel OPA Fabric with shared I/O and storage. vFabrics help control and reduce the number of I/O devices visible to each host and prevent hosts from communicating with each other using storage vFabrics. This can further secure the fabric and in some cases, may make Plug and Play host software and storage-management tools easier to use.

Some typical usage models for vFabrics include the following:

- Separating a cluster into multiple vFabrics so independent applications run with minimal or no effect on each other.
- Separating classes of traffic. For example, putting a storage controller in one vFabric and a network controller in another enabling all networking to be secure and predictable.

vFabrics provide a unifying concept by which Security and QoS features are configured and managed.

A vFabric consists of a group of applications that run on a group of devices. For each vFabric the operational parameters of the vFabric can be selected.

As shown in Fig. 5.3, a set of applications, device groups, and finally vFabrics are defined. Each application is defined as a set of ServiceIDs and/or Multicast Group IDs (MGIDs). These are defined in terms of pattern match rules (or explicit lists), so that the wide range of 64 and 128 bit inputs can be easily specified. Each DeviceGroup consists of a list of device ports. Those device ports may be explicitly listed or matched via Node Description pattern matching. Finally, a VFabric is defined as a list of applications, a list of device groups, with a set of QoS and security policies, along with the P_Key and SL identifiers for the VirtualFabric.

Within the OFA APIs, packets for a given VFabric are identified by a P_Key (indicating security) and an SL (indicating QoS). During fabric initialization, the FM will have configured the devices to be aware of these two identifiers and assign the appropriate security (e.g., limited/full/none membership) and QoS (e.g., MTU, bandwidth, priority, preemption rank, static rate, flow control, and HoqLife) policies

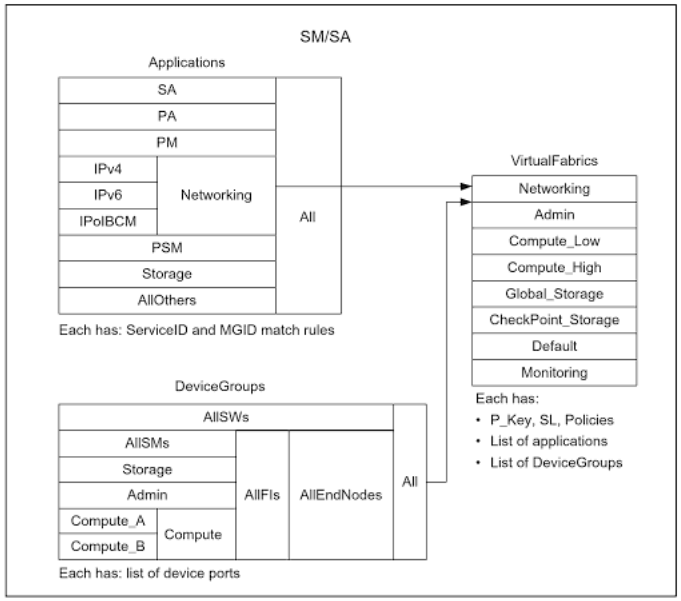


FIG. 5.3

Virtual Fabrics definition within FM.

to these identifiers at each port in the fabric. The settings for these policies may also influence other parameters for the port such as how buffers, VLs, and other resources are allocated to each VFabric.

The vFabrics mechanisms are closely tied to address resolution and multicast membership in the OFA APIs and applications.

When an application makes an address resolution or multicast membership query to the FM, the FM checks the application identifier (ServiceId or MGID) against the FM's list of applications to identify one or more potential matches. It also checks the source and destination nodes against the FM's list of DeviceGroups to identify one or more potential matching device groups. The FM then looks for vFabrics which include both a matching application and a matching device group and which also match any supplied SL, P_Key, and MTU. The one or more resulting vFabrics are used to compose address resolution responses which include the P_Key, SL, and other communications parameters (MTU, StaticRate, PktLifeTime) which the application should use to communicate through the fabric to the identified destinations.

Fig. 5.4 shows an example of this mechanism. In this example, a ServiceID (address resolution) or an MGID (multicast join/create) is supplied which matches the IPv4 Application. The IPv4 Application is part of the Networking and All applications, so they also are matched. Also a source and a destination are supplied explicitly or implicitly and a DeviceGroup is selected which matches both. The Compute_A group includes both the source and destination. The Compute_A group is also included in the Compute, AllFIs, AllEndNodes, and All groups, so they are also matched.

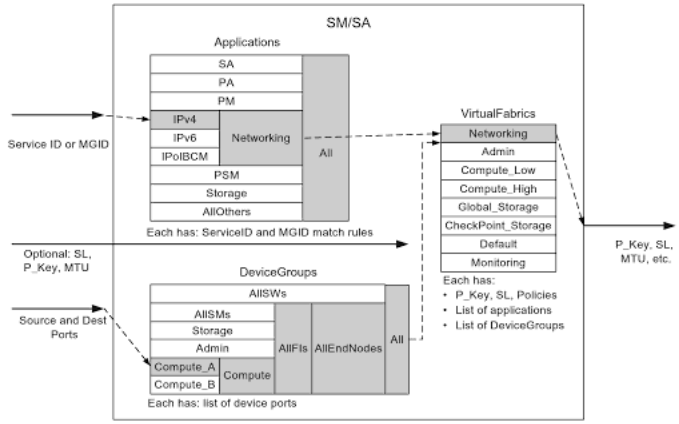


FIG. 5.4

Example of Virtual Fabrics resolution during address or multicast resolution by SM/SA.

Depending on the definition of Applications and DeviceGroups, it is possible for a given input to match two or more disjoint entries.

Given the list of matched Applications and matched DeviceGroups, the SM/SA searches the vFabrics list for an entry which includes both a matched Application and a matched DeviceGroup. In the example, the Networking VirtualFabric entry includes the Networking Application and the All DeviceGroup. If supplied, additional input parameters (SL, P_Key, MTU) will also be compared to the VirtualFabric to confirm a match. Finally, the response will include the P_Key, SL, MTU and other relevant polices from the matched VirtualFabric.

Some of the concepts of vFabrics are best explained via some simple sample configurations.

Fig. 5.5 shows the default FM configuration of vFabrics. In Fig. 5.5, a series of devices are shown, compute nodes A, compute nodes B, storage, switch Port 0s, FMs and other Admin nodes (such as job schedulers). In a given cluster there may be many of each type of device. Horizontally, two vFabrics are shown, Default VF and Admin VF. For each of these vFabrics the P_Key and SL are indicated. Parameters such as the P_Key and SL can be explicitly specified or dynamically assigned by the FM. Lastly on the right are shown the Applications which are associated with each vFabric.

In this example, the Default virtual fabric consists of all devices and "All Other" applications. This vFabric will be used by the majority of applications and is assigned a P_Key `0x0001` and SL 0. This example also shows the Admin virtual fabric. This fabric is used for secure fabric management of the fabric (SA, PA, PM, and implicitly the SM). As such it includes all the FM nodes and all the switch port 0's (which may also manage devices inside their given switch chassis). All other nodes are limited members. This vFabric is assigned P_Key `0x7fff` and SL 0. In this configuration, all traffic shares a single SL, so there is no fabric QoS. The Admin traffic is secured so that SA, PA, PM, and SM traffic is only permitted between the FM (and switch port 0's) and other devices. However other devices, such as Compute A devices, cannot attempt to manage other devices in the fabric.

Fig. 5.6 builds on Fig. 5.5 by providing an additional vFabric named Compute which includes all devices and the Compute applications. This vFabric uses

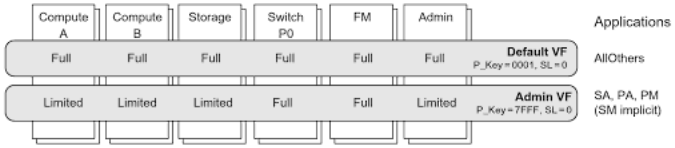


FIG. 5.5

Default Virtual Fabrics configuration.

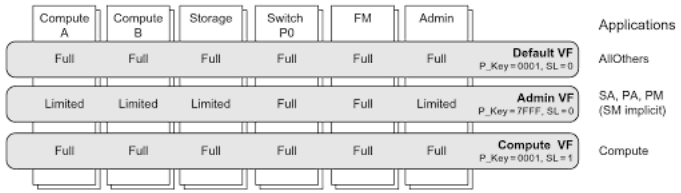


FIG. 5.6

Simple QoS Virtual Fabrics configuration.

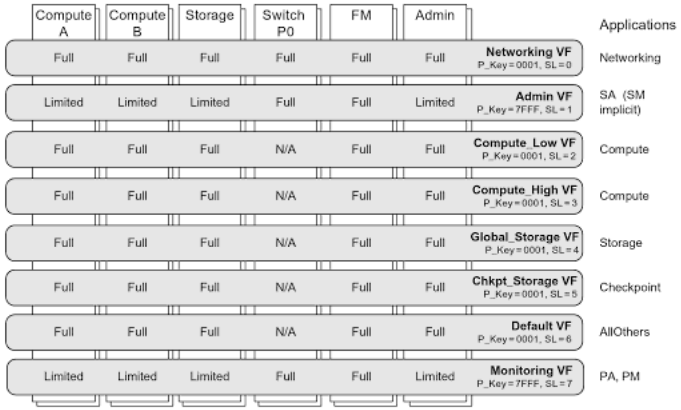


FIG. 5.7

Advanced QoS Virtual Fabrics configuration.

P_Key `0x0001` and SL 1. In this configuration, no new security is enabled, however by assigning Compute to a unique SL (and configuring the Compute virtual fabric's QoS policies such as bandwidth and preemption), the Compute fabric's traffic can be separated from other fabric traffic by using a unique SL and hence unique SC's and VL's throughout the fabric. This example also points out the flexibility and cross matching. The Default and Compute virtual fabrics share a common P_Key and hence have the same security rules. However, the Default and Admin virtual fabrics share a common SL and hence have the same QoS rules.

Fig. 5.7 builds further on Fig. 5.6. In this figure, eight unique vFabrics are defined and each is assigned a unique SL. This configuration has limited security as only management traffic is secured on P_Key `0x7fff`, while all other traffic is

unsecured and uses P_Key `0x0001`. However by assigning each vFabric a unique SL, they each have unique QoS characteristics and the performance impacts between vFabrics will be controlled by the QoS policies configured for each vFabric. In this configuration, due to separate vFabrics, the fabric performance statistics can also be monitored per vFabric. This occurs in the FM's Performance Manager and is based on analysis of the per VL statistics on each port. This example also shows that, if desired, the management traffic can also be split out; the Admin virtual fabric is used for SA traffic while the Monitoring virtual fabric is used for PM and PA traffic. This can permit the PM traffic to proceed at regular intervals with less impact from bursty name resolution traffic which may be occurring to the SA.

Fig. 5.8 is a security-focused example. In this example, each vFabric is assigned a unique P_Key. The Compute A virtual fabric is limited to the Compute A devices. Similarly the Compute B virtual fabric is limited to the Compute B devices. This means a Compute application can only be run within the Compute A or the Compute B devices, but may not use devices from both groups. Both Networking and Compute applications are permitted within those two vFabrics, so Compute A can be its own separate IP subnet from Compute B. This style of configuration may be useful when multiple departments or tenants of the cluster each need separate subsets of the cluster to run jobs and the sysadmin wants to secure the traffic so applications cannot mistakenly communicate with each other. This example also has an Admin virtual fabric, which is similar to Fig. 5.5 and secures the fabric management traffic from the other traffic. Finally there is a Services virtual fabric which includes the storage and admin nodes as full members with Storage and AllOthers applications. This permits the storage and admin nodes to communicate with any node in the fabric, however it prevents the nodes in Compute A from attempting to communicate with Compute B via this P_Key.

The concepts presented in these examples can easily be combined to create more advanced configurations with a mixture and security and QoS polices as required by the given fabric and its operations.

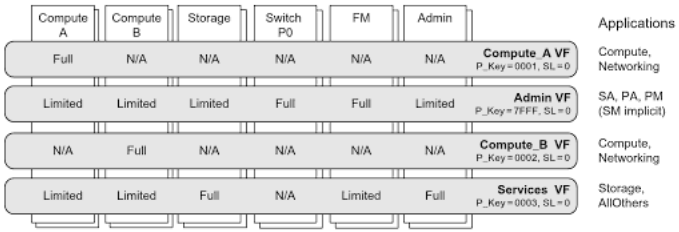


FIG. 5.8

Simple security Virtual Fabric configuration.

The Fabric Manager also has some capabilities to dynamically adjust vFabrics by making changes to the configuration in a live fabric. This can be used to facilitate occasional changes in the needs, such as adding or removing a department or tenant on the fabric. See the *For More Information* section for resources with more details on the OPA FM and vFabrics.

UNICAST ADDRESS RESOLUTION

vFabrics is designed to integrate automatically within the OFA stack. Applications which take advantage of standard connection establishment and address resolution (aka `PathRecord` resolution) mechanisms such as RDMA CM, IB CM, and ibacm will automatically make the necessary SA queries so that the fabric manager can provide `PathRecord`s with the appropriate settings for SL, Partition Key (P_Key), MTU and other parameters used for fabric communications.

TYPICAL FLOW FOR WELL BEHAVED APPLICATIONS

The standard flow for `PathRecord`s and vFabric address resolution is as shown in Fig. 5.9.

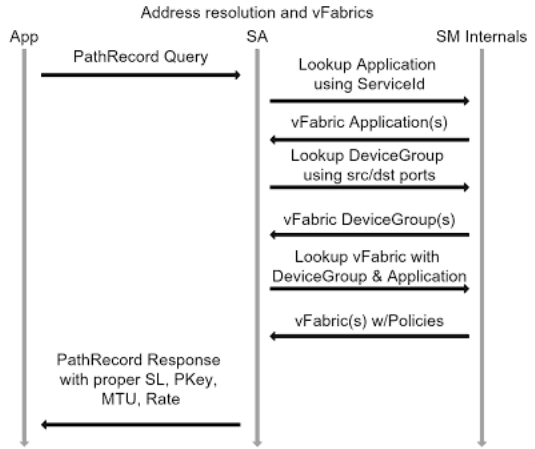


FIG. 5.9

Address resolution flow within SM/SA.

As show in the flow diagram in Fig. 5.9, the SA makes use of the Service Identifier (Service Id) supplied in the `PathRecord` query to identify the application making the request. The source and destination nodes specified in the query will imply the possible set of DeviceGroups which are relevant, and then the SA will find the virtual fabric(s) which represent the intersection of the given application with the possible device groups. To ensure this process works smoothly and correctly, it's important that the FM configuration of vFabrics specifies the appropriate set of Service Ids for the application. Some applications may provide direct configuration of the Service Id as a 64 bit number. This is especially true of those using the IB CM or ibacm directly. Applications using the RDMA CM will specify a protocol and port number and these will be used to compose a 64 bit service ID.

RDMA Service IDs take the form `0x00000000`*`01NNPPPP`* where *`N`* is the Internet Assigned Numbers Authority (IANA) protocol number and *`P`* is the port number the application needs to bind on, both in hexadecimal. For example, Lustre is known to use the TCP protocol (IANA `0x06`) on port 987 (`0x03db`) so its Service ID is expected to be `0x00000000010603db`.

OUT OF BAND MECHANISMS

In some cases, the application may use non-standard, out of band or ad-hoc mechanisms to establish connections. In which case, the key connection parameters of P_Key, SL, and MTU will need to be specified a priori. This approach is shown in Fig. 5.10.

Intel OPA fabric software provides assorted tools which may be used to identify the P_Key, SL, and MTU associated with a given vFabric. In some cases, those tools may be used to automate the discovery of the SL, P_Key, and MTU and then provide them directly to the application, hence reducing the risk of human mistakes and permitting future runs of the application to correctly obtain any changes to the vFabrics configuration.

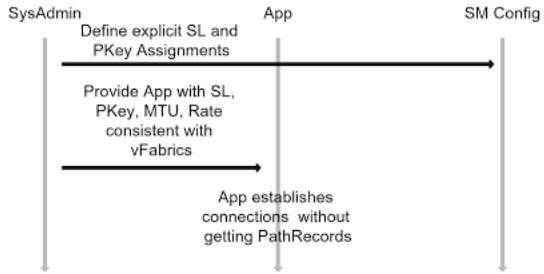


FIG. 5.10

Out of band mechanisms for QoS and security.

MULTICAST ADDRESS RESOLUTION

Applications which take advantage of standard multicast mechanisms (aka Multicast Member Records) either directly, via IPoFabric (aka IPoIB) or via ibacm, will automatically receive the proper SL, P_Key, and MTU as part of the multicast group parameters.

TYPICAL FLOW FOR WELL-BEHAVED APPLICATIONS

The standard flow for Multicast Member Records when creating a new multicast group is as shown in Fig. 5.11.

As shown in the flow diagram in Fig. 5.11, the SA makes use of the MGID supplied in the Multicast Member Record to identify the application making the request. The source nodes specified in the query will imply the possible set of DeviceGroups which are relevant, and then the SA will find the virtual fabric(s) which represent the intersection of the given application with the possible device groups. To ensure this process works smoothly and correctly, it's important that the FM configuration of vFabrics specifies the appropriate set of MGIDs for the application. Some applications may provide direct configuration of the MGID as a 128-bit number. This is especially true of those using the IB SA or ibacm directly. Applications using IPoFabric (aka IPoIB) will specify an IP multicast group and these will be used to compose a 128-bit MGID.

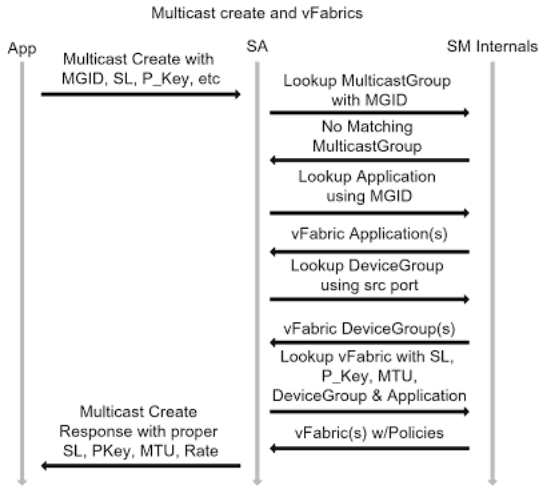


FIG. 5.11

Multicast create flow within SM/SA.

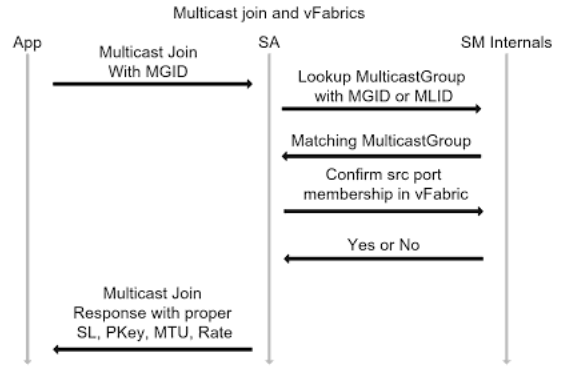


FIG. 5.12

Multicast join flow within SM/SA.

For IPv4, MGIDs are composed by IPoFabric as follows:

```
MGID=0xffFS401bPPPP0000:00000000GGGGGGGG
where F=flags, S=scope, P=PKey, and G=IP Multicast Group
```

For IPv6, MGIDs are composed by IPoFabric as follows:

```
MGID=0xffFS601bPPPPGGGG:GGGGGGGGGGGGGGGG
where F=flags, S=scope, P=PKey, and G=IP Multicast Group
```

When an application joins an existing multicast group via the Multicast Member Record mechanisms, the steps in Fig. 5.12 occur.

In this case the pre-existing multicast group will already have an assigned SL, P_Key, and MTU. The new join request will merely need to confirm the source nodes ability to communicate with that P_Key.

SUMMARY

The Intel Omni-Path Architecture introduces a multi-generation fabric architecture designed to meet the scalability needs of datacenters ranging from the high-end of HPC to the breadth of commercial datacenters. Link-level reliability features provide the reliability needed for systems at large scales. The QoS architecture is coupled with new packet preemption capabilities to enable both bandwidth fairness and low latency jitter for high-priority packets. In the first product generation, each link provides 100 Gb/s of bandwidth, and each HFI can achieve 160 million messages per second. Switch latency has been reduced to under 110 ns. These are substantial improvements relative to prior-generation products while preserving the existing

software ecosystem. In addition, a new community standard network API, i.e., OFA OFI (libfabric), is designed to match user-level semantic requirements to enable hardware innovation beneath the API.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- Further Omni-Path information, documentation and software can be found at www.intel.com/omnipath and http://lotsofcores.com/omnipath.
- Intel, "Intel True Scale Fabric Architecture: Enhanced HPC Architecture and Performance," Intel, Santa Clara, CA, USA, 2012.
- G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins and J. Reinhard, "Cray cascade: a scalable HPC system based on a Dragonfly network," in SC12: International Conference on High Performance Computing, Networking, Storage and Analysis, Los Alimitos, CA, USA, 2012.
- L. A. Barroso, J. Clidara and U. Hozle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition, California, USA: Morgan and Claypool Publishers, 2013.
- P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard and J. M. Squyres, "A Brief Introduction to the OpenFabrics Interfaces: A New Network API for Maximizing High Performance Application Efficiency," in IEEE Hot Interconnect 23, Santa Clara, CA, USA, 2015.
- ISO/IEC, 7498-1: Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, Geneva, Switzerland: ISO, 1996.
- InfiniBand Trade Association and others, Infiniband Architecture Specification: Release 1.0, InfiniBand Trade Association, 2000.
- IEEE, IEEE Std 802.3bj(TM)-2012; Amendment 2: Physical Layer Specifications and Management Parameters for 100 Gb/s Operation Over Backplanes and Copper Cables, New York, NY, USA: IEEE Computer Society, 2014.
- W. J. Dally, "Virtual-channel Flow Control," Parallel and Distributes Systems, IEEE Transactions on, vol. 3, no. 2, pp. 194–205, 1992.
- M. Luo, K. Seager, K. Murthy, C. Archer, S. Sur and S. Hefty, "Early Evaluation of Scalable Fabric Interface for PGAS Programming Models," in PGAS 2014, 8th International Conference on Partitioned Global Address Space Programming Models, Eugene, OR, USA, 2014.
- www.top500.org — a web site tracking the current and historical top 500 performing HPC systems in the world. The Performance Development charts are especially interesting.
- http://gasnet.lbl.gov — a Web site describing GasNET and its use on assorted PGAS compilers and libraries including UPC and SHMEM.

This page intentionally left blank

CHAPTER

µarch optimization advice 6

This chapter offers advice on tuning specific to the Knights Landing design which we know as the microarchitecture, and we like to abbreviate as µarch but pronounce as micro-architecture (or micro-ark). In other words, we focus on tuning advice arising specifically from the Knights Landing µarch design when compared with the Knights Corner µarch (found in the first generation Intel Xeon Phi products) or the µarch of a recent Intel® Xeon® processor. While maximal performance tuning of any machine relies on detailed knowledge of the underlying µarch, code may remain portable and performance portable by parameterizing choices such as how many threads per core to use, which vector instruction set to ask for with compiler switches, and how to best utilize the available memory types. The optimal choices, for these settings, are generally rewarded with higher performance.

What is new with Knights Landing in this chapter?

This chapter is exclusively about Knights Landing, including advice on how to deal with differences from the prior Intel Xeon Phi product (Knights Corner).

This chapter offers a collection of microarchitectural details and discusses their implications on tuning. The section names have been chosen to reflect a very short version of the advice, and the sections themselves offer more detailed explanations. There are also a series of sections on differences between AVX-512 and IMCI (Intel® Initial Many-core Instructions—the 512-bit SIMD ISA on Knights Corner) specifically for those transitioning code.

If you prefer to not dive into such machine specific details, we still recommend flipping through the sections and reading the first paragraph of each. If you crave details, perhaps because you are a compiler writer or a very advanced application tuner, we trust you will find the details within this chapter to be very useful.

BEST PERFORMANCE FROM 1, 2, OR 4 THREADS PER CORE, RARELY 3

The choice of how many threads per core to utilize is highly application dependent, and it is good advice to parameterize this choice and run performance tests to choose the best number for an application on any given machine. One thread per core on the

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00006-5

Knights Corner was generally far from the peak performance for every application (the best was most often two or three per core); Knights Landing µarch, which may reach maximum performance with a single thread per core.

An individual thread has the highest performance when running as the only thread on a core. As thread count per core grows to two or four, most applications will have higher aggregate performance, but lower per thread performance. If an application can scale its performance to an arbitrary number of threads, four threads per core are likely to have the highest instruction throughput. Practical limitations on memory capacity or parallelism may limit the scaling effectiveness when increasing the threads used per core. However, memory *latency* sensitive applications usually benefit from using the maximum number of threads per core available, as it is more likely that one of the thread's data is available for processing at any given time.

One thread per core can be powerful enough for an application to get maximal performance. Running three threads per core may be the least likely to be the best choice, but it can be.

The Knights Landing µarch partitions "per core" resources in fourths when using three or four threads on a core. Because of this, a three thread configuration will have fewer total internal resources than any other configuration of threads per core. Placing three threads on a core is less likely to perform better than two or four threads per core, but we have seen examples of three being the optimal—obviously when the reduced resources on the core were not the key bottleneck.

Most highly tuned applications are likely to use one thread per core, with two thread per core being in a close second. Unlike Knights Corner, a single thread on Knights Landing does have the ability to use the full capabilities of the core and to saturate memory. There will be some applications (e.g., random access with memory latency sensitivity) which find four threads per core useful, so it is worth parameterizing this decision and doing speed comparisons.

If multiple threads per core are active, but only a subset are doing useful work (e.g., the other threads are in a barrier), performance will suffer. The threads doing nonuseful work will take up pipestages and resources from the threads that we want to make progress—limiting their potential performance. If threads are likely to be stalled for a long period, it might be better to suspend the stalled thread, and wake it up later. The overheads for an OS level thread switch are high, so this should be done carefully.

HYPERTHREADING: DO NOT TURN IT OFF

It is possible to turn off hyperthreading on Knights Landing via a BIOS boot time option. However, turning it off is unlikely to ever be useful to any application. Keep hyperthreading on, and if only one thread per core is used, it will give all the resources to that one thread.

MEMORY SUBSYSTEM

CACHES

Knights Landing has multiple levels of caches. Data and instruction sets that fit in caches are likely to perform better. Knights Landing has separate 32 KB caches for instructions and data. These are commonly referred to as the "level 1" (L1) caches. The two cores in a tile share a 1 MB cache that can hold a mix of instructions and data. This is commonly referred to as the "level 2" (L2) cache. When multiple tiles read the same cache line, each tile might have a copy of the cache line. If both cores in the same tile read a cache line, there will only be a single copy in the L2 cache of that tile. Cache lines found in the L1 cache can be accessed with high bandwidth and low latency. Cache lines found in the L2 cache have lower bandwidth and higher latency than the L1 cache, but higher bandwidth and lower latency than an access to memory (including MCDRAM).

If MCDRAM is configured as a cache, then it can hold data or instructions that are accessed by the cores in a single place. If multiple tiles request the same line, only one MCDRAM cache line will be used.

MCDRAM AND DDR

MCDRAM and DDR memory have different latency and throughput profiles. When using the various memory modes (cache/hybrid/flat) and deciding on where to allocate memory, this knowledge is important. In most memory configurations, the DDR capacity will be substantially larger than MCDRAM capacity. Likewise, MCDRAM capacity will be much larger than the combined L2 cache. Working sets that fit in MCDRAM capacity, but not in the L2 caches should be in MCDRAM. Large or rarely accessed structures should migrate to DDR. The processor will try to do this dynamically if MCDRAM is put in cache or hybrid mode. If memory is in "flat" mode, working sets are bound to one memory or the other at allocation time.

Fig. 6.1 conceptually depicts the relationship between memory bandwidth and memory latency for MCDRAM and DDR. In general, memory latency increases with

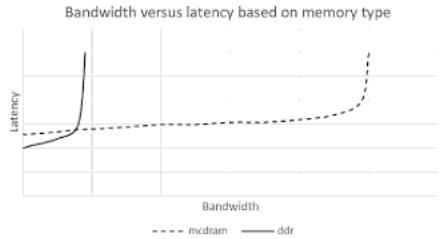


FIG. 6.1

Latency and bandwidth curves for MCDRAM and DDR.

memory bandwidth because as memory gets busier with increasing number of requests, each request waits longer in hardware queues for their turn to access memory. While this increase is gradual for most part, it becomes exponential as demand for memory bandwidth approaches memory's peak bandwidth. Although MCDRAM latency is higher than DDR when bandwidth demand is low, it does not rise as fast as DDR latency when bandwidth demand increases because MCDRAM sustains much higher bandwidth compared to DDR. The actual memory latency experienced by software will depend on the bandwidth demand it places on MCDRAM and DDR.

ADVICE: LARGE PAGES CAN BE GOOD (2M/1G)

Most programs default to using 4KB pages. There are many instances where using 2M or 1G pages can improve performance. For instance, when accessing a large data structure (64 MB) randomly, 2M pages will stay in the DTLB, while 4KB pages will constantly require the hardware to translate linear to physical addresses. For many applications with large datasets and nonsequential access patterns, larger pages can provide significant performance improvements.

µARCH NUANCES (TILE)

In this section, we review several suggestions on how to optimize the performance of the tile for the Knights Landing µarch.

INSTRUCTION CACHE, DECODE, AND BRANCH PREDICTORS

The front end unit (instruction cache, decoders, and branch prediction) handles most HPC code well. There are some restrictions that rarely impact performance of which optimizers and compiler writers should be aware. Code generation should avoid large instructions, or instructions with many prefixes. Branch targets should be within 4GB of the branch.

The instruction cache can fetch up to 16 bytes per cycle. Since the processor can process up to two instructions per cycle, this could limit performance when the average instruction length is greater than 8 bytes. The only common instructions that use more than 8 bytes are vector math instructions with a memory source that have a 4-byte displacement. If executing a long sequence of such operations, constraining the displacement size will provide an improvement in performance. Chapter 21 has an example of using this knowledge to increase FMA issue rate by encoding FMAs as 8-byte instructions, instead of longer instructions. Using multiple base or index registers is one option to reduce the size of the displacement.

When selecting instructions, try to avoid instructions with many prefixes. If an instruction has three or more prefixes, the hardware requires more time to determine instruction boundaries. The penalty will happen each time the instruction

is encountered. This is generally not an issue, but some instructions, like ADCX and ADOX, are affected. These instructions should not be used in performance critical code.

The indirect branch predictor assumes that the branch destination is in the same 4 GB chunk of the virtual address-space as the indirect branch instruction (note for these purposes a "ret" is an indirect branch instruction). Since indirect branches are used to enter dynamic shared libraries, which are normally scattered through the virtual address space, calls into DLLs incur two misprediction penalties (one on the call and one on the return). Similarly, system calls in Linux are made via a pseudodynamic library; therefore, they will also suffer from the misprediction. To mitigate this, try to limit the frequency of system calls and calls to dynamic libraries. Statically linking libraries (instead of using shared libraries) can help. This is especially important for low latency library calls that are accessed frequently—whether MPI, a math function, or a network driver. An updated version of the dynamic linker (ld.so) is shipped with some OS releases. If requested, it will try to place all shared libraries into the low 4 GB of the address space, to avoid the mispredictions on shared library calls. If this linker is installed, setting the environment variable LD_PREFER_MAP_32BIT_EXEC is likely to improve performance. Since it has no impact if the modified dynamic linker is not present, we set the environment variable on all of the machines where we run our program. Therefore, we like to put this in our ~/.bash_profile file:

```
export LD_PREFER_MAP_32BIT_EXEC=1
```

This works for 64-bit applications because branch prediction performance on Knights Landing cores can be negatively impacted when the target of a branch is more than 4 GB away from the branch. On Linux, adding the `LD_PREFER_MAP_32BIT_EXEC` bit will affect `mmap`, so it will map executable pages with `MAP_32BIT` first. `LD_PREFER_MAP_32BIT_EXEC` does reduce bits available for address space layout randomization and can only be enabled by creating the environment variable `LD_PREFER_MAP_32BIT_EXEC` (its value is unimportant, its existence creates the behavior change). Anecdotal feedback indicates this can often give a boost of several percent in performance. This is a feature of glibc 2.23 and later.

Having "`export LD_PREFER_MAP_32BIT_EXEC=1`" in your ~/.bash_profile file may be worth a few percent in performance.

INTEGER

Knights Landing integer instructions perform similar to the Intel® Atom™ processors. There is no optimization for instructions with partial integer writes, or partial flag writes. Instructions like PUSH, POP, and DIV should be avoided if simpler instructions can be used.

If a code sequence needs to PUSH or POP several values sequentially, it is better to issue multiple stores or loads (each using a unique RSP offset) and then update

RSP once. Each PUSH/POP instruction generates its own µop to update RSP. This puts more pressure on allocation and integer execution bandwidth. PUSH and POP instructions are commonly used as part of the calling convention, and eliminating redundant RSP updates can yield some benefit to code with many short functions.

The Knights Landing µarch does not optimize 8- and 16-bit integer partial registers (like AH, AL, and AX). Each reference to an integer register reads and writes the full 64-bit version. For (very old) code that was written when 32-bit CPUs were a new concept, this might be an issue, but is unlikely to be a problem for high performance applications. It is simplest, and not slower, to just refer to the 32- and 64-bit versions of the integer registers (like EAX and RAX).

The Knights Landing µarch does not optimize partial flag writes. Instructions that write EFLAGs status bits will write all the status bits. If a subset of the bits is unchanged by an instruction, then the EFLAGs writer cannot execute until the previous writer of EFLAGs has completed. This can cause unexpected performance drops relative to recent Core microprocessors. INC and DEC are the two most frequently encountered instructions that are affected by this. Unless your code is dependent on not writing all flags, it is usually better to replace DEC with "SUB 0x1," and INC with "ADD 0x1."

Integer division is a common mathematical operation, used whenever two integers are divided, or a modulus operation is specified. Unfortunately for those who care about performance, computing this operation is quite slow. If the integers are known to be relatively small (16 bits or less), there are fast software sequences to emulate the division (that the compiler might try to generate). If the divisor is known to be a power of 2, please use SHR (division) and/or AND (remainder) instead of DIV. If the divisor is a constant, multiplication by another constant will provide the correct answer. If the input values are highly constrained, a precomputed lookup table might provide better performance. Division instructions should be aggressively minimized by the compiler, either by using the techniques mentioned earlier or by hoisting redundant divisions out of inner loops.

VECTOR

Performance optimized code should not use x87 and SSE instructions. Instead, code should use the AVX (packed and scalar) equivalents (128-, 256-, or 512-bit width) as much as possible. SSE and x87 instructions are generally similar performance to AVX but are occasionally much slower. There are potential performance glass jaws when mixing SSE with 256- or 512-bit operations mentioned later on.

Do not use the COMIS* and UCOMIS* set of instructions. Replacing these instructions with the two instruction sequence of VCMPS* and KORTEST is almost always better for performance.

Some instructions, like VCOMPRESS*, are fast when writing a register, but much slower when storing to memory. Where possible, it is faster to do a VCOMPRESS to a register and then store it. However, potential setup (e.g., masking) may limit the benefit.

MASKMOVQ and similar instructions should be replaced by k-masked ZMM load/store instructions.

The Knights Landing µarch does not have the same restrictions on mixing SSE and AVX code that Core processors have. Knights Landing will suffer some performance loss if an AVX instruction that operates on 256- or 512-bits is allocated before all previous SSE instructions for that thread have retired. The performance loss is comparable to a mispredicted branch. When compiling for the Knights Landing µarch, it is best to not generate SSE instructions. If the binary has been freshly compiled, only legacy libraries should have SSE code in them. This limits any performance hiccups to a few instructions after returning from legacy library calls. Extremely bad code generation would alternate SSE instructions with wider AVX instructions. To achieve excellent performance using the Knights Landing µarch, avoid this.

The Knights Landing µarch does not optimize the performance of VZEROUPPER or VZEROALL. The VZEROALL instruction requires about 10 cycles of bandwidth in the allocation, execution, and retirement pipelines. The frontend in the Knights Landing µarch uses the microcode sequences (MSs) for VZEROUPPER and will need about 15 cycles. Core processors generally take a cycle of bandwidth at each point for VZEROUPPER. Code for Knights Landing should avoid both instructions as much as possible.

Experience has shown that intrinsic heavy code from IMCI (Knights Corner µarch) implementations will not necessarily generate optimal code for the Knights Landing µarch. IMCI had free simple permutes and a more limited Instruction Set Architecture (ISA) relative to AVX-512 in the Knights Landing µarch. An expert coder will want to scrub legacy 512-bit intrinsics tuned for IMCI to limit the number of MS flows generated, and to fix limitations imposed by the Knights Corner µarch (unaligned memory accesses). Large speedups are possible from even a quick scrub of the generated code.

Do not use the HADDP* instruction to do a horizontal addition with Knights Landing µarch. There are examples later in the document that list more efficient instruction sequences.

For vector code, it is generally best to keep consistent with the element types. For instance, if a vector compare on 32-bit integer elements writes a YMM register, performance is likely to be better if the consumer of the YMM register has a 32-bit integer data type (like PXOR).

There are a few instructions relevant to HPC that are long latency. The most commonly used are floating-point division, floating-point square root, and integer division. The programmer and/or compiler should carefully consider various tradeoffs for each of these instructions. The DIVPD, DIVPS, SQRTPD, and SQRTPS instructions are implemented as MSs that use Newton-Raphson to iterate to an IEEE correct answer. This gives better bandwidth and slightly worse latency than the same instructions on most recent Intel Xeon processor cores. The scalar versions of these instructions are shorter MSs, that use a hardware unit, with worse bandwidth and better latency than the packed versions of the instructions. The integer divide

performance is slow, similar to Intel Atom processor cores, and should be avoided in performance critical code.

Compiler options like "-no-prec-div" in the Intel compilers take advantage of loosening IEEE compatibility requirements and can generate faster instructions sequences with the AVX-512ER instructions (more on this later).

For floating point divisions and square roots, it is greatly beneficial to be able to vectorize. The vector versions are approximately as fast as the scalar versions, so able to do 8 or 16 operations simultaneously greatly improves performance.

Many complex math operations (with real and imaginary values) benefit from the VFMADDSUB* instructions. Please use them when appropriate.

MEMORY ACCESSES AND PREFETCH OPTIONS

The Knights Landing µarch performs well on accesses that split cache lines, but there is still a performance penalty relative to accesses that do not split cache lines. If an algorithm has an access pattern that streams through memory, it would be beneficial to align as many of the accesses as possible to a 64-byte boundary. Likewise, if we want a 32-byte value (YMM), do not access 64-byte (ZMM) in memory and then mask off the last 32-bytes. This creates cache line splits for no reason.

The Knights Landing µarch must take extra time when a load or store access crosses a 4KB boundary in order to test the permissions for each page involved. This happens regardless of the page size. This introduces a larger performance penalty for the access that crosses the boundary. If an algorithm is streaming through memory with unaligned 64-byte loads, every 64th load will cross a 4KB boundary. It is best to align the access pattern as much as possible. If it is not possible, consider inserting a few software prefetches to the L2 cache (PREFETCHT1 and similar instructions) several iterations ahead of the stream. This will start the page translation early and permit the L2 hardware prefetcher to start fetching the stream on the next page.

Some access patterns for gather and scatter will always have pairs of consecutive addresses. One common example are complex numbers, where the real and imaginary parts are laid out contiguously. It is also common when w, x, y, and z information is contiguous. If the values are 32-bit, it is faster to gather and scatter the elements as half as many 64-bit elements. If the numbers are 64-bit, then it is usually faster to load and insert 128-bit values instead of doing a gather.

There are multiple hardware prefetchers on each tile in the Knights Landing µarch. One analyzes all the accesses in the data cache and the instructions that generated them—the Instruction Pointer Prefetcher (IPP). This prefetcher attempts to insert hardware prefetches to the L1 if a strided access pattern is detected on a cacheable page. The IPP will not generate prefetches that cross a 4KB boundary.

The L2 hardware prefetcher tries to identify streaming access patterns (where consecutive cachelines are accessed) and can track up to 48 streams. The L2 hardware prefetcher only looks at the address of requests that make it to the L2 cache. It has no knowledge of the instruction, or any accesses that were satisfied inside the individual cores. If there are four threads enabled per core, this means that the L2

hardware prefetcher can track up to six streams per thread (two cores share an L2 cache). If a stream is detected, and there are few memory accesses pending, hardware prefetches for later elements of the stream will be send to the L2, and if they miss, those accesses go to memory. The hardware prefetcher will not stream across a 4KB address boundary.

The hardware prefetchers can be disabled within the BIOS menus at boot time or using BIOS tools, specific to your machine, similar to those used in Chapter 3 for memory and cluster modes.

Tuning tools can measure memory activity and can help understand the effectiveness of both hardware and software prefetches. The counters available to tuning tools, such as the Intel® VTune™ Amplifier, include only "demand" loads in the "miss" counters. This was not possible on Knights Corner. Effective prefetching should cause "miss" counts to be reduced. Performing comparisons by turning on or off different types of prefetching (software prefetching comes in different types and usages, there are two hardware prefetchers) and observing the "miss" rates can be very instructive. More information on the specific event counters are available in Chapter 14. We also have more pointers to additional prefetching advice on our website as mentioned in *For More Information* at the end of this chapter.

Experiments with prefetching will see "miss" rates go down when prefetching is effective because the "miss" counters include only "demand" loads, not prefetches. Chapter 14 discusses these counters.

AVX-512PF supports many flavors of software prefetch instructions. The Knights Landing µarch is more resilient to cache misses than Knights Corner, so programmers with experience from Knights Corner should not feel compelled to aggressively insert software prefetches. With the two hardware prefetchers described in previous paragraphs, most streaming and short strided access patterns should be detected by hardware prefetchers. If the access pattern is streaming, then a programmer will likely benefit from software prefetches beyond a 4KB boundary, since the hardware prefetchers will miss the first several accesses to a new 4KB region. If the access pattern is known, but nonstreaming, then software prefetches can be beneficial. This is especially true if the access pattern is a relatively large stride (>256 bytes), since the hardware prefetches will not fetch across a 4KB boundary. The software prefetch will do the PMH walk to fill the TLB and start the memory reference early. When accessing multidimensional arrays (like A[i][j][k]) sequentially in *i*, *j*, or *k*, remember that one access order has sequential addresses, and the others involve large strides in the address. The former is captured by hardware prefetchers, and the latter is generally best captured by software prefetches.

Generally, software prefetching into the L2 will show more benefit than L1 prefetches. L1 prefetches hold critical hardware resources (e.g., fill buffer) until the cache line fill completes. L2 prefetches do not hold those resources, so it is less likely that inserting an L2 software prefetch will negatively impact performance. When

using L1 software prefetches, it is strongly advised that the software prefetch hits in the L2 cache so that the length of time that the hardware resources are held is minimized. Common practice is for L1 software prefetches to be used well after an L2 prefetch has previously been executed for the same address.

Avoid inserting software prefetches too aggressively. It is always a good idea to try turning software prefetching on and off and compare results. Code tuned for Knights Corner generally needed prefetching more than Knights Landing. Chapter 26 describes an application which benefited strongly from software prefetching with Knights Corner, but runs best with no software prefetching on Knights Landing.

Additionally, if software prefetches reference addresses that are not valid, or are not mapped by the OS, an application may experience a significant slowdown from such software prefetches. The performance monitoring event NUKE.ALL provides an indication of when this might be affecting performance. Tools such as Intel® VTune™ Amplifier (see Chapter 14) can help.

The MEU is partially out of order. Memory accesses are dispatched from the scheduler in-order but can complete in any order. This impacts performance negatively when the second oldest memory µop is ready to dispatch (address is good), but the oldest memory µop has not yet dispatched (memory address is not good). This can be easily observed in algorithms that do pointer chasing. If code loads the pointer from memory (cycle 0), and then dereferences it in the next µop, the earliest point that µop can dispatch is in cycle 4. This means that no other MEC µops from that thread can dispatch in the second slot of cycle 0, or in any slot of cycles 1, 2, or 3. A more common example of pointer chasing relevant to supercomputing is when the base address of many arrays (&a[0]) is kept on the stack. The compiler and/or ninja programmer should try to maximize the number of memory operations between the load of the base address, and the instruction that dereferences it. A simple option is to load base pointers as consecutive memory references and then dereference them in pairs. For instance, if we want to access a[i] and b[i], we can write the code shown in Fig. 6.2. The naïve sequence, shown in Fig. 6.3, interchanges the second and third

```
movq    r15, [rsp+0x40]     ;;; cycle N (load &a[0])
movq    r14, [rsp+0x48]     ;;; cycle N+1 (load &b[0])
vmovups zmm1, [r15+rax*8]   ;;; executes in cycle N+4
vmovups zmm2, [r14+rax*8]   ;;; cycle N+5
```

FIG. 6.2

Interleaved pointer chasing.

```
movq    r15, [rsp+0x40]     ;;; cycle N (load &a[0])
vmovups zmm1, [r15+rax*8]   ;;; executes in cycle N+4
movq    r14, [rsp+0x48]     ;;; cycle N+4 (load &b[0])
vmovups zmm2, [r14+rax*8]   ;;; cycle N+8
```

FIG. 6.3

Naïve pointer chasing.

instructions and is three cycles slower. The advantage of the naïve sequence is that it only requires a single integer register to hold the base address. The faster sequence requires two integer registers for the base addresses, which can put more pressure on the register allocator. The benefit of the first sequence increases if the first pointer load misses the data cache.

If there are many loads in the machine, it might be possible to hoist up the pointer loads so that there are several memory references between the pointer load and dereference, without requiring more integer registers to be reserved.

Store to load forwarding is simplistic in the Knights Landing µarch. Integer loads and stores (RBX, EAX) can forward if the store and load have the same beginning memory address and the load is not larger than the store. Vector, x87, and MMX loads and stores can forward (ZMM0, YMM1, XMM2, MM3, and ST4) with the same conditions. Vector to Integer and vice-versa do not forward between themselves, and the load must wait until the store is postretirement. Vector stores that use a mask (a k-register other than k0) cannot be immediately forwarded from. If an algorithm requires such behavior, it may benefit from merging the value in a register and then storing to memory using k0. Later loads can then forward from the merged value.

The memory hierarchy that caches lines and determines forwarding uses the address of the access. The L1 data cache uses bits 11:6 to identify which cache set to use. Forwarding logic uses bits 11:0 and the size of the access to identify potential forwarding or conflicts between loads and stores. If there are many conflicts, performance could be degraded. Unfortunately, many dynamic memory allocation routines (dependent on OS and compiler) will start large memory regions with the same bottom 12 bits. If a program accesses many arrays with identical shapes (element size and dimensions) and similar indices, performance could be significantly degraded. It is beneficial for bits [11..6] of memory accesses to be different, as illustrated in Fig. 6.4.

There are multiple ways to offset dynamic arrays. If `free()` is not relevant, we can just allocate a few hundred bytes and manually offset the arrays: `b=(double*)((char*)b+192)`. We can also code our own versions of memory allocation routines to achieve this. Alternatively, we can use `memalign()` with different alignment

```
a = malloc(sizeof(double) * 10000);
b = malloc(sizeof(double) * 10000);
// very likely in most OSes that (a & 0xfff) == (b & 0xfff)
for (i=0; i < 10000; i++) {
      // a[i] and b[i] of iteration N collide
      // a[i] of iteration N-1 and
      //     b[i-1] of iteration N collide
      a[i] = b[i] + 0.5 * b[i-1]);
}
```

FIG. 6.4

Array access pattern that collides in the same L1 set.

directives for each dynamic allocation to induce the OS to provide different memory alignments.

Many scientific applications that use large arrays are vulnerable to this. The SPEC06 application leslie3d can be affected by this quite easily.

When using VGATHER and VSCATTER, we often need to set a mask to all ones. An efficient instruction to do this is KXNOR of a mask register with itself. Since VSCATTER and VGATHER clear their mask as the last thing they do, a loop carried dependence from the VGATHER to KXNOR can be generated. Because of this, it is wise to avoid using the same mask for source and destination in KXNOR. Since it is rare for the k0 mask to be used as a destination, it is likely that "`KXNORW k1, k0, k0`" will be faster than "`KXNOR k1, k1, k1`."

CODE EXAMPLES

In this section, we look at some common kernels and access patterns and how to best code for the Knights Landing µarch.

Fig. 6.5 shows macrocode for a horizontal reduction of 32-bit elements. This is for single-precision floating point but can be easily altered for a 32-bit integer reduction. Fig. 6.6 shows code for a 64-bit floating-point horizontal reduction. Fig. 6.7 has a simplified code fragment for the inner loop of DGEMM, trying to compute

```
;; vector to reduce is in zmm6
vextractf64X4     zmm1, zmm6, 0x1
vaddps       ymm1, ymm6, ymm1
vpermpd      ymm4, ymm1, 0xFF
vpermpd      ymm5, ymm1, 0xAA
vpermpd      ymm3, ymm1, 0x44
vaddps       xmm1, xmm1, xmm4
vaddps       xmm3, xmm5, xmm3
vaddps       xmm3, xmm1, xmm3
vpsrlq       xmm1, xmm3, 32
vaddss       xmm3, xmm3, xmm1
```

FIG. 6.5

Single-precision horizontal reduction from ZMM vector to scalar.

```
;; vector to reduce is in zmm6
vextractf64X4 zmm1, zmm6, 0x01
vaddpd        ymm1, ymm6, ymm1
valignq       ymm4, ymm1, 0x3
valignq       ymm5, ymm1, 0x2
valignq       ymm3, ymm1, 0x1
vaddsd        ymm1, ymm1, ymm4
vaddsd        ymm3, ymm5, ymm3
vaddsd        ymm3, ymm1, ymm3
```

FIG. 6.6

Double-precision horizontal reduction from ZMM vector to scalar.

```
;; matrix - matrix dense multiplication
prefetcht0  [rdi+0x400]   ;; get A matrix element into L1$
vmovapd     zmm30, [%rdi]
prefetcht0  [rsi+0x400]   ;; get B matrix element into L1$
vfmadd231pd zmm1, [rsi+r12]{b}, zmm30 ;; b-cast B element
vfmadd231pd zmm2, [rsi+r12+0x08]{b}, zmm30
vfmadd231pd zmm3, [rsi+r12+0x10]{b}, zmm30
vfmadd231pd zmm4, [rsi+r12+0x18]{b}, zmm30
vfmadd231pd zmm5, [rsi+r12+0x20]{b}, zmm30
vfmadd231pd zmm6, [rsi+r12+0x28]{b}, zmm30
vfmadd231pd zmm7, [rsi+r12+0x30]{b}, zmm30
vfmadd231pd zmm8, [rsi+r12+0x38]{b}, zmm30
prefetcht0  [rsi+0x440]         ;; pull line into the L1 $
vfmadd231pd zmm9, [rsi+r12+0x40]{b}, zmm30
vfmadd231pd zmm10, [rsi+r12+0x48]{b}, zmm30
vfmadd231pd zmm11, [rsi+r12+0x50]{b}, zmm30
vfmadd231pd zmm12, [rsi+r12+0x58]{b}, zmm30
vfmadd231pd zmm13, [rsi+r12+0x60]{b}, zmm30
vfmadd231pd zmm14, [rsi+r12+0x68]{b}, zmm30
vfmadd231pd zmm15, [rsi+r12+0x70]{b}, zmm30
vfmadd231pd zmm16, [rsi+r12+0x78]{b}, zmm30
```

FIG. 6.7

Fragment of DGEMM inner loop.

C = A * B. The code in Fig. 6.7 has 16 partial sums. There should always be FMA instructions ready to execute in the VFU (6 cycles per FMA, up to 2 FMAs per cycle). It is important to keep the average instruction length at 8 bytes or less, to enable maximum throughput. This is why the index register in these examples `(r12)` is used, so the displacement used in the FMAs can be kept small. At the end of the inner loop, the partial sums will need to be added to produce a single value to be stored out (to the C matrix).

DIRECT MAPPED MCDRAM CACHE

The MCDRAM cache is a convenient way to increase memory bandwidth. As a memory side cache, it can automatically cache recently used data and provide much higher bandwidth than what DDR memory can achieve. When MCDRAM is placed in cache mode, it is a direct mapped cache. This means that multiple memory locations map to a single place in the cache. Because of this, a simple first optimization for a program is to turn on the MCDRAM cache. Some applications that heavily utilize a few GB of memory could see performance improvements of up to 4×. Because of the simplicity of this—no source code changes, and the large possible performance benefits, moving from DDR only to MCDRAM cache mode should be one of the first performance optimizations to try.

There are a few scenarios where enabling the cache could reduce performance. One case is when the MCDRAM cache is not able to hold the accessed working set. If an application streams through 64 GB of memory without reuse, then checking the MCDRAM cache (and missing) will only increase latency.

Another thing to note is that the direct mapped cache uses the physical address, not the linear address. Even if an address is contiguous in the linear/virtual address, the physical addresses that the OS gives to the application memory are not required to be. This can cause cache contention when using a significant portion of the MCDRAM cache. This is likely to reduce the peak memory bandwidth achievable. This can vary from run to run, as how the OS allocates pages can change from run to run. Monitoring the cache hit rate events from `perfmon` can be instructive in diagnosing this.

If MCDRAM cache is enabled, every modified line in the tile caches (L1 or L2 cache) must have an entry in the MCDRAM cache. If it is not in the MCDRAM cache, then the line will be downgraded to "shared" from "modified" in the tile cache. There is a very small probability that a pair of lines that are frequently read and written will map to the same MCDRAM set. This could cause a pair of reads that would normally hit in the L1 caches to become reads that need to go to DDR. This would cause a pair of threads to become substantially slower than the other threads in the chip. Due to linear to physical mapping variation, this behavior could vary from run to run, making it difficult to diagnose.

This case is very hard to create, but the way we employed to forcibly create this case was with two threads read and write their local stacks. Conceptually, any data location that is commonly read and written would work, but register spills to the stack are the most frequent case. If the stacks are offset by a multiple of 16 GB in physical memory, they would collide into the same MCDRAM cache set. A run-time that forced all thread stacks to allocate into a contiguous hardware memory region would avoid this from occurring. There is hardware to reduce the frequency of set conflicts from occurring. The probability of hitting this scenario on a given node is extremely small. The best clue to detecting this is that a pair of threads on the same chip are significantly slower than all other threads during a program phase—the threads that collide should vary from run to run, happen rarely, and only when MCDRAM cache mode is enabled.

ADVICE: USE AVX-512

ADVICE: UPGRADE TO AVX-512 FROM AVX/AVX2 AND IMCI

The Knights Landing microprocessor fully supports the full ISA from Intel processors, including the recent Intel Xeon processor (codenamed Haswell and Broadwell) µarchs, with the exception of Transactional Synchronization Extensions. Knights Landing supports the new 512-bit Advanced Vector Extensions (AVX-512). Because of this, vector instruction support exists for MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and AVX2. For floating point instructions, SSE* operates on 128-bits at a time, AVX* on 128- or 256-bits and AVX-512 on 512-bits at a time. Roughly speaking, AVX-512 offers 4× performance over SSE* and 2× over AVX* for similar mathematical operations, purely because of the number

of mathematical operations done in parallel. Using the instructions with the widest support will offer the highest performance on any processor, and that holds true for Knights Landing as well.

Specifically, the additional AVX-512 support includes AVX-512F (foundation), AVX-512CD (conflict detection), AVX-512PF (prefetching), and AVX-512ER (exponential and reciprocal) ISA extensions. With these new instructions, AVX-512 introduces the ZMM register set (32 512b vector registers—ZMM0-31) and mask registers (eight 64-bit mask registers—k0–7). All devices supporting AVX-512 must support the foundational instructions (AVX-512F), while the AVX-512 extensions are dependent on the μarch. All Intel Xeon Phi processors from Knights Landing forward will support at least the AVX-512CD, AVX-512PF, and AVX-512ER extensions. All Intel Xeon processors that include AVX-512F will also include at least the AVX-512CD extension. In general, this level of detail is left to be dealt with by a compiler or library. It is a detail with which programmers using intrinsics will also need to deal with (see Chapter 12).

The Knights Corner coprocessor did not support the AVX-512 ISA but did support a similar 512-bit vector ISA referred to as IMCI. Many of the instructions in IMCI and AVX-512 are similar, with different bit encodings. However, some differences exist. AVX-512 offers superior scatter and gather support and is significantly different than IMCI. In addition, IMCI supported "free" swizzles inside 128-bit lanes, and several up-conversions and down-conversions to and from memory, which are not supported by AVX-512 on Knights Landing. There are no other announced devices from Intel that support the IMCI ISA extensions. IMCI will likely live up to its name as being only "initial" instruction set to support 512-bit vectors. Intel has announced future support for AVX-512 in its Intel Xeon processor line as well.

The bottom-line is clear—use the AVX-512 ISA to achieve the best performance.

SCALAR VERSUS VECTOR CODE

The compiler or programmer can determine that a loop can be vectorized, but there is a threshold of loop count below which vectorization may not be better than scalar code. A very simple guideline is that loops that iterate more than 16 times will be faster with vector code, and scalar code is faster for loops that iterate less than four times. In between those two values, vector and scalar code has about the same performance.

For the rest of this section, a slightly more sophisticated algorithm to choose between scalar and vector code is discussed. It assumes that the operations in the loop and the number of iterations the loop will execute are known.

A horizontal reduction operation transforms a vector into a scalar value. These generally occur at the end of a loop when a scalar value must be produced from the vector registers. The most common type of horizontal reduction sums the vector elements. The constants used in the sample cost model are listed in Fig. 6.8.

Operation	Cost	Example
Simple math	1	A*B+C
Load (with cache-line split)	1 (2)	A[i]
Store (with cache-line split)	1 (2)	A[i] = 2;
Gather (scatter) 8 element	15 (20)	A[column[i]]
Gather (scatter) 16 element	20 (25)	A[column[i]]
Horizontal reduction	30	sum += A[i]
Division or square root	15	A/B

FIG. 6.8

Vectorization cost model.

```
for (i=0; i<N; i++) { sum += a[i]*K + b[i]; }
```

FIG. 6.9

Horizontal reduction.

It is easiest to explain the algorithm via examples. Due to differences in various compilers' internal formats and programmers' knowledge, the algorithm has to be a bit flexible. Some of the corner cases are glossed over.

Fig. 6.9 shows a simple horizontal reduction loop.

There are two loads (a[i] & b[i]), an FMA, and a horizontal reduction in the vector version (to get sum). If compiled to scalar code, the horizontal reduction becomes a scalar add per loop iteration.

The heuristic costs are $4*N$ for scalar code, and 3*ceiling(N/8) +30 for double precision AVX-512 vector code. The vector version of the code assumes that the main loop and remainder loop are vectorized. The break-even point between scalar and vector code would be around $N=9$. If N is smaller, then scalar is better. If N is larger, then vectorized code will be faster.

The code in Fig. 6.10 utilizes gathers. There are two loads (indir[i] & b[i]), an FMA, a store (c[i] =), and a gather/load, depending on whether vector or scalar code is produced. The costs are $5*N$ for the scalar code and 19*ceiling(N/8) for the vector code. Scalar is better if $N<4$.

The code in Fig. 6.11 uses even more gathers. There is one load (ind[i]), an FMA, a store (c[i] =) and 2 gathers/loads. The scalar cost is $5*N$, and the vector cost is 33*ceiling(N/8). Scalar is better for small values of N. For very large values of N, vector code is probably better, but the margin is likely small.

```
for (i=0; i<N; i++) {c[i] = a[indir[i]]*K + b[i]; }
```

FIG. 6.10

Gather computation.

```
for (i=0; i<N; i++) {c[i] = a[ind[i]]*K + b[ind[i]]; }
```

FIG. 6.11

Double gather.

The code in Fig. 6.12 has a gather and a horizontal reduction. There are two loads (ind[i] & b[i]), an FMA, a gather/load, and a horizontal reduction/sum. The scalar cost is 5*N, and the vector cost is 19*ceiling(N/8)+30. Scalar code is better for $N \leq 13$, and vector code is probably better for larger trip counts.

The code in Fig. 6.13 uses a scatter and a division. There are three loads (a[i], b[i], & ind[i]), a scatter/store, and a division (a[]/b[]). The scalar cost is 19*N, and the vector cost is 38*ceiling(N/8). Scalar code is better if $N \leq 1$. In other words, vectors are best if you have more than one iteration to do!

The code in Fig. 6.14 uses a gather and scatter operation. There is one load (ind [i]), a scatter/store (b[ind[i]]), and a gather/load (a[ind[i]]). The scalar cost is 3*N, and the vector cost is 36*ceiling(N/8). According to the heuristic, scalar code is always better. With the in-order MEC, it is likely that the scalar version of the loop would benefit from being unrolled a bit.

A code fragment from miniMD is shown in Fig. 6.15. Outside the IF clause, there is a load, three gathers, and six math operations. Inside the IF clause, there is a

```
for (i=0; i<N; i++) {sum += a[ind[i]]*K + b[i]; }
```

FIG. 6.12

Gather/horizontal reduction.

```
for (i=0; i<N; i++) {c[ind[i]] = a[i] / b[i]; }
```

FIG. 6.13

Division/scatter loop.

```
for (i=0; i<N; i++) {b[ind[i]] = a[ind[i]]; }
```

FIG. 6.14

Gather/scatter copy loop.

```
for (int k = 0; k < numneigh; k++) {
  int j = neighs[k];
  double rsq = (xtmp - x[3*j])^2 +
               (ytmp - x[3*j+1])^2 +
               (ztmp - x[3*j+2])^2;
  if (rsq < cutforcesq) {
     double sr2 = 1.0/rsq;
     double sr6 = sr2*sr2*sr2;
     double force = sr6*(sr6-0.5)*sr2;
     res1 += delx*force;
     res2 += dely*force;
     res3 += delz*force;
  }
}
```

FIG. 6.15

Code fragment from miniMD.

division, eight math operations, and three horizontal reductions. The scalar cost is 10*numneigh+23 * numneigh * percent_rsq_less_than_cutforcesq. The vector cost is (52+23) * ceiling(numneigh/8)+3 * 30. Scalar code makes sense if (numneigh < 6) or if the compiler is highly confident that the if clause is almost never taken.

For many compilers, a vectorized loop is generated, and a remainder loop is used to take care of the rest of the operations. In other words, the vectorized loop is executed floor(N/8) times, and the remainder loop is executed (N mod 8) times. In that case, modify the equations to use floor instead of ceiling to determine whether the primary loop should be vectorized. For the remainder loop, the maximum value of the loop trip count is known: one less than the vector width. If N is unknown, it is simplest to set N to half the vector width (four for a ZMM vector of doubles).

More sophisticated analysis can be done in this area. This is a starting point for optimizers.

INSTRUCTION LATENCY TABLES

Knowledge of the latency and bandwidths of instructions is important to achieve good performance. Care should be taken to select instructions that can be executed in the least amount of time. Vector math instructions are generally 2 or 6 cycle latency and have their information listed in Fig. 6.16. Scalar integer instructions are mostly 1 cycle latency and have their information listed in Fig. 6.17.

Some of the instructions have throughput lower than 1. They might be bottlenecked by the number of operations being done (gather/scatter and division) or by the time it takes to generate the μops in the frontend. The converts that are listed

Vector Instructions	Latency	Instructions/cycle
Simple integer	2	2
Most vector math (FMA)	6	2
Mask operations	2	2
X87 / MMX math	6	1
EMU (AVX-512ER)	7	0.5
Shuffle / permutes (1 src)	2	1
Shuffle / permutes (2 src)	3	0.5
Convert – same width	2	1
Convert – different width	6	0.2
Vector Loads	5	2
Store to load forwarding	2	2
Gather (8 elements)	15	0.2
Gather (16 elements)	19	0.1
Float to integer move	2	1
Integer to float move	4	1
DIVSS or SQRTSS	25	0.05
DIVSD or SQRTSD	40	0.03
Packed DIV or SQRT	38	0.1

FIG. 6.16

Vector latency and bandwidth.

Scalar Integer instructions	Latency	Instructions/cycle
Most math	1	2
Integer multiply	3 or 5	1
Store to load forwarding	2	1
Integer Loads	4	1
Integer division	Variable	0.05

FIG. 6.17

Scalar integer latency and bandwidth.

as "different width" are conversions where the vector width of the sources and destination are different.

ADVICE: USE AVX-512 EXTENSIONS FOR KNIGHTS LANDING

These relatively small sets of instructions (AVX-512ER, AVX-512PF, and AVX-512CD) are important for efficient vectorization of HPC programs.

ADVICE: USE AVX-512ER

The AVX-512ER instructions provide high precision approximations of exponential, reciprocal, and reciprocal square root functions. Approximations from earlier ISA extensions, like rcp11ps, are far less accurate. An accurate approximation can reduce execution time for iterative methods like Newton-Raphson. Fig. 6.18 shows code using the Newton-Raphson method to compute a single 32-bit float division with vrcp28ss. Both values are read off the stack. Note the use of rounding mode overrides on some of the math operations.

IMCI TO AVX-512: RECIPROCAL AND EXPONENTIALS

The 23-bit single-precision reciprocals and exponentials found in Knights Corner are replaced by AVX-512ER in Knights Landing as listed in Fig. 6.19. AVX-512ER brings Knights Landing a high accuracy implementation of base 2 exponential,

```
vgetmantss    xmm18, xmm18, [rsp+0x10], 0
vgetmantss    xmm20, xmm20, [rsp+0x8], 0
vrcp28ss      xmm19, xmm18, xmm18
vgetexpss     xmm16, xmm16, [rsp+0x8]
vgetexpss     xmm17, xmm17, [rsp+0x10]
vsubss        xmm22, xmm16, xmm17
vmulss        xmm21{rne-sae}, xmm19, xmm20
vfnmadd231ss  xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss   xmm21, xmm19, xmm20
vscalefss     xmm0, xmm21, xmm22
```

FIG. 6.18

Division via Newton-Raphson algorithm using vrcp28ss.

Instruction	Description
VEXP2PD, VEXP2PS	Compute approximate exponential 2^x with a relative error of at most 2^{-23}.
VRCP28PD, VRCP28PS, VRCP28SD, VRCP28SS	Compute approximate reciprocals with a relative error is at most 2^{-28}.
VRSQRT28PD, VRSQRT28PS, VRSQRT28SD, VRSQRT28SS	Compute approximate reciprocals of square roots with a relative error is at most 2^{-28}.

FIG. 6.19

AVX-512ER instructions.

reciprocal, and reciprocal of square root with 28-bit mantissa bits. The two reciprocal instructions are much more accurate than the 14-bit mantissa version in AVX-512F. The single-precision version is also more accurate than Knights Corner's 23-bit version.

The reason for increased accuracy and double-precision support is to avoid expensive divide instruction and runtime calls to exponential function when the application accuracy requirement is relaxed.

ADVICE: USE AVX-512CD

The AVX-512CD instructions (see Fig. 6.20) allow for efficient vectorization of several access patterns. A common scheme can be characterized as the "histogram update." This is when a memory location is read, operated on, and then stored to. A sample piece of C code with this access pattern is shown in Fig. 6.21. This code can be incorrectly vectorized with a gather, vpadd, and a scatter. To get the correct answer with vectorization, we must worry about cases where key[n] and key[m] have the same value, and n and m are in the same vector chunk. The AVX-512CD

Instruction	Description
VPCONFLICTD, VPCONFLICTQ	Detect duplicate values within a vector and create conflict-free subsets
VPLZCNTD, VPLZCNTQ	Count the number of leading zero bits in each element
VPBROADCASTMB2Q, VPBROADCASTMW2D	Broadcast vector mask into vector elements

FIG. 6.20

AVX-512CD instructions.

```
for (i=0; i < 512; i++)
     histo[key[i]] += 1;
```

FIG. 6.21

Histogram update in C.

instructions detect these cases and permit correct vectorization of the loop. Disassembly from a compiler that vectorizes the histogram update code is shown in Fig. 6.20.

Another example of using AVX-512CD is found in Chapter 20 to create a function called `conflict_safe_accumulate` using a sequence of intrinsics for a conflict safe gather-modify-scatter force update. The use of AVX-512CD allows accumulation of forces from multiple lanes with the same neighbor index into a single-data lane so that the last value written to memory will contain the correct result.

ADVICE: USE AVX-512PF

The AVX-512PF instructions (see Fig. 6.22) are used to aid memory performance when the programmer or compiler knows with a high degree of certainty the set of cache lines that will be accessed in the near future. Their relationship to gather and scatter is similar to the relationship of prefetch(w)* to loads and stores. They should be used in equivalent scenarios, where the compiler or software writer is comfortable in providing hints to hardware on which lines should be in cache. Prefetch instructions do not impact architectural state.

Instruction	Description
VGATHERPF0DPS, VGATHERPF0QPS, VGATHERPF0DPD, VGATHERPF0QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using opmask k1 and T0 hint.
VGATHERPF1DPS, VGATHERPF1QPS, VGATHERPF1DPD, VGATHERPF1QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using opmask k1 and T1 hint.
VSCATTERPF0DPS, VSCATTERPF0QPS, VSCATTERPF0DPD, VSCATTERPF0QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using writemask k1 and T0 hint with intent to write.
VSCATTERPF1DPS, VSCATTERPF1QPS, VSCATTERPF1DPD, VSCATTERPF1QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using writemask k1 and T1 hint with intent to write.

FIG. 6.22

AVX-512PF instructions.

IMCI TO AVX-512: SOFTWARE PREFETCHING

Software Prefetching is mainly used to hide the memory latency for an application. For Knights Corner, software prefetching is essential. In addition to latency sensitive applications, software prefetching proved useful even on many streaming and/or memory bandwidth bound workloads. This was largely because of the hardware prefetcher Knights Corner was limited to only a streaming L2 Hardware prefetcher per core.

Knights Landing has multiple hardware prefetchers per core which greatly diminishes the need for aggressive software prefetching for many applications. We recommend reading about a methodology for tuning prefetching in Chapter 21, *Prefetch Tuning Optimizations*, of the Pearls Volume Two book. See reference in *For More Information* at the end of this chapter.

Since it may still be worthwhile to experiment with software prefetching in applications, we explain the different types of prefetch hints available (with some "intrinsics" example) and the differences that exist between Knights Corner and Knights Landing.

The table in Fig. 6.23 shows the different prefetch instructions available on Knights Corner with details on the level of the cache the instruction brings the cache line into and also the state of the cache line (which is controlled using "prefetch hints").

The table in Fig. 6.24 shows the different prefetch instructions available on Knights Landing with details on the level of the cache the instruction brings the cache

Instruction	Cache Level	Non-Temporal	Bring as exclusive	Prefetch Hint (for intrinsics)
vprefetch0	L1	No	No	_MM_HINT_T0
vprefetchnta	L1	Yes	No	_MM_HINT_NTA
vprefetch1	L2	No	No	_MM_HINT_T1
vprefetch2	L2	Yes	No	_MM_HINT_T2
vprefetche0	L1	No	Yes	_MM_HINT_ET0
vprefetchenta	L1	Yes	Yes	_MM_HINT_ENTA
vprefetche1	L2	No	Yes	_MM_HINT_ET1
vprefetche2	L2	Yes	Yes	_MM_HINT_ET2

FIG. 6.23

Knights Corner (IMCI) prefetch instructions.

Instruction	Cache Level	Non-Temporal	Bring as exclusive	Prefetch Hint (for intrinsics)
prefetcht0/	L1	No	No	_MM_HINT_T0/
prefetchnta	L2	No	No	_MM_HINT_NTA
prefetcht1/				_MM_HINT_T1/
prefetcht2				_MM_HINT_T2
prefetchw	L1	No	Yes	_MM_HINT_ET0
prefetchwt1	L2	No	Yes	_MM_HINT_ET1

FIG. 6.24

Knights Landing prefetch instructions.

```
int L2dist=64*8;
int L1dist=4*8;
#pragma omp parallel for
for (j=0; j<ARRAY_SIZE; j++) {
// Prefetch "a" from memory to L2 and bring line as "exclusive"
  _mm_prefetch ((const void*)&a[j+L2dist],_MM_HINT_ET1);
//Prefetch "a" from L2 to L1 and bring line as "exclusive"
  _mm_prefetch ((const void*)&a[j+L1dist],_MM_HINT_ET0);
//Prefetch "b" from memory to L2 and bring line as "exclusive"
  _mm_prefetch ((const void*)&b[j+L2dist],_MM_HINT_ET1);
//Prefetch "b" from L2 to L1 and bring line as "exclusive"
  _mm_prefetch ((const void*)&b[j+L1dist],_MM_HINT_ET0);
  c[j] = a[j] + b[j];
}
```

FIG. 6.25

Prefetch intrinsics suitable for both Knights Landing and Knights Corner.

line into and also the state of the cache line (which is controlled using "hints"). Note that Knights Landing does not have nontemporal software prefetching. If a cacheline is brought into a cache level, it is also allocated in the lower level caches (L2 and/or MCDRAM cache).

Fig. 6.25 is a simple example showing how to use the **prefetch intrinsics**.

ADVICE: GATHER AND SCATTER INSTRUCTIONS ONLY WHEN IRREGULAR

In order to increase the performance of algorithms with irregular data access patterns, Knights Landing supports scatter and gather instructions as part of AVX-512. Gather can be thought of as multiple element size loads, where the elements are merged into a single-vector register. A second vector register is used to specify multiple addresses. A scatter is comprised of multiple element-sized stores, where one vector register holds the data to be stored, and another vector register specifies the multiple addresses for the stores. AVX2 supports gathers for XMM and YMM vectors but does not support scatter.

AVX-512 gather and scatter operation should only be used when the data needed is truly scattered in memory (not contiguous). If we have data that is contiguous in memory, but needs to be in a different order in the registers before computations are done, we should not use gather/scatter instructions to achieve the rearranging. It is better to load data using regular AVX-512 instructions and permute the data once it is in the SIMD (ZMM) registers. Chapter 12 includes an excellent "load plus permute" sequence because such sequences are more efficient than using gather instructions to combine the loading and permuting. The example in Chapter 12 involves complex numbers which are layed out sequentially in memory but need to be operated on in a different pattern than they are stored. Our example shows how to load, and permute, the incoming data to maximize performance. Similarly, the example shows how to organize results so that a masked store can be used instead of resorting to a scatter.

To maximize performance, use gather/scatter instructions only when the data is truly sparse in memory. If some data is contiguous in memory, but needs rearranging before computation, we should do regular loads and then use permute instructions. Such "load plus permute" sequences are more efficient than using gather instructions to combine the loading and permuting. Similarly, permute plus store will be more efficient than a scatter.

AVX-512F versions of gather and scatter are quite different from the IMCI implementation, which would only process the elements in a single cache line for each instruction. The AVX-512F version of the instructions completes the entire gather or scatter in a single instruction which makes them much faster than the IMCI instructions. There are several microarchitectural improvements that allow Knights Landing µarch to be faster than the Broadwell µarch for AVX2 gathers.

Gather and scatter instructions support various index, element, and vector widths. The AVX-512 flavors of gather and scatter use the mask registers to identify the lanes that should be loaded to, or stored from, the vector registers. AVX2 does not support scatter instructions or mask registers. In addition, AVX2 is limited to at most a 256-bit vector width. Knights Corner supports scatter and gather instructions with mask registers and 512-bit vector registers. For correct completion of a Knights Corner gather or scatter, the programmer would need to wrap the instruction in a test of the mask for 0, and a conditional branch.

For instance, consider the C code fragment shown in Fig. 6.26. The optimal AVX2 code would be similar to what is shown in Fig. 6.27; the optimal IMCI coding would be similar to what is shown in Fig. 6.28; the optimal AVX-512F coding would be similar to what is shown in Fig. 6.29. The AVX-512F code is more compact that the other instruction formats. In addition to using fewer instructions, the code can be executed faster than the alternate ISA sequences (IMCI or AVX2).

```
for (uint32 i=0; i < 16; i++)
     b[i] = a[indirect[i]];
```

FIG. 6.26

Looping on `b[i] = a[indirect[i]]`.

```
vmovdqu     ymm0, [rsp+0x1000]  ;; load indirect[]
vmovdqu     ymm3, [rsp+0x1020]  ;;   part 2
vpcmpeqd    ymm4, ymm4, ymm4    ;; setup mask
vmovdqa     ymm1, ymm4          ;;   part 2
vpgatherdd  ymm2, [rax+ymm0*4], ymm1
vpgatherdd  ymm5, [rax+ymm3*4], ymm4
vmovdqu     [rsp], ymm2         ;; store b[]
vmovdqu     [rsp+0x20], ymm5    ;;   part 2
```

FIG. 6.27

AVX2 code for `b[i] = a[indirect[i]]`.

```
    vmovaps    zmm0,[rsp+0x1000]   ;; load indirect[]
    kxnor      k1,k1               ;; setup mask for gather
  gather_loop:
    vgatherdps zmm2{k1},[rax+zmm0*4]
    vgatherdps zmm2{k1},[rax+zmm0*4]
    jknzd      k1,<gather_loop>    ;; see if gather is done
    vmovaps    [rsp],zmm2          ;; store b[]
```

FIG. 6.28

IMCI code for `b[i] = a[indirect[i]]`.

```
    vmovups     zmm0, [rsp+0x1040]     ;; load indirect[]
    kxnor       k1, k0, k0             ;; mask for gather
    vpgatherdd  zmm1{k1}, [rsi+zmm0*4] ;; gather a[]
    vmovdqu32   [rsp], zmm1            ;; store b[]
```

FIG. 6.29

AVX-512 code for `b[i] = a[indirect[i]]`.

IMCI TO AVX-512: GATHERS/SCATTERS

The hardware implementation and the corresponding software (compiler generated code/intrinsics) for gather-scatter are different between Knights Corner and Knights Landing. Consider a simple *Indirect Access* Kernel Loop as shown in Fig. 6.30. This code snippet will explain the gather-scatter differences between Knights Corner and Knights Landing in detail.

In Fig. 6.30, the load of "a [b[j]]" is a "gather" operation and store of "a [b[j]]" is a scatter operation. The corresponding compiler generated code for Knights Corner and Knights Landing is shown in Figs. 6.31 and 6.32.

On Knights Corner, the gather/scatter implementation is "load per cacheline" which means a single gather and scatter instruction can load or store multiple elements from the same cacheline, due to which we see a "gather (scatter) loop" with branch on mask test in the compiler-generated code.

For example, if we look at the Knights Corner gather code in Fig. 6.31, there are two gather instructions wrapped in a loop with jkz/jknz. If the first gather instruction loads all elements from the same cacheline, then the "jkz" branch will be true and will exit the "gather loop" (since internally the "k2" mask is unset (set to zero) for every loaded element). If the elements are in multiple cache-lines, then it will loop until all the elements are loaded (mask (here "k2") controls the gather completion).

On Knights Landing, the gather/scatter implementation is "load (store) per element" which means every element is treated as an individual load (store) whether in the same cache line or not. But from the instruction point of view, gather/scatter is a single instruction as shown in Knights Landing code in Fig. 6.32 and need not be


```
int main() {
/* Memory Allocation*/
  a = (double *)_mm_malloc(sizeof(double)*(ARRAY_SIZE),64);
  b = (int *)_mm_malloc(sizeof(int)*(ARRAY_SIZE),64);
  c = (double *)_mm_malloc(sizeof(double)*(ARRAY_SIZE),64);

/* Initializing  the Arrays*/
for (k = 0; k < ARRAY_SIZE; k++) {
  a[k] = rand() % 20;
  b[k] = ARRAY_SIZE;
  c[k] = rand() % 20;
}
/* Calling the Sparse Function (which does Gathers/Scatters)*/
for (k = 0; k < NTIMES; k++)
  Sparse();
/* Memory Free*/
  _mm_free(a);
  _mm_free(b);
  _mm_free(c);
}

void Sparse() {
  int j;
  #pragma simd
  #pragma vector aligned
  #pragma omp parallel for
  for (j = 0; j < ARRAY_SIZE; j++) {
    //Gather (Load) "a" with multiple index loaded from b[j]
    c[j] = a[b[j]]; -- > Gather
    //Scatter (Store)  "a" with multiple index loaded from b[j]
    a[b[j]] = c[j]; -- > Scatter
  }
}
```

FIG. 6.30

Indirect Access Kernel Loop.

wrapped in any loop. Gathers can process two "elements" per clock, and scatter can process one "element" per clock.

The implementation for Knights Corner provides better performance than the AVX-512 implementation in Knights Landing if the number of cache lines touched in the gather is small (0, 1, or 2). The implementation in Knights Landing is likely to perform better than Knights Corner if many cache lines are accessed per gather. Please refer to the Knights Landing Optimization Manual for some more details.

Figs. 6.33 and 6.34 are simple examples (for the Sparse loop) showing how to use the "gather/scatter intrinsics" on Knights Corner and Knights Landing. There is a slight variation in the syntax for gather/scatter between Knights Corner and Knights Landing. But for both Knights Corner and Knights Landing, the arguments of the gather/scatter intrinsic are the same.

```
Gather: (datatype vindex, void const* base_addr, int scale)
```

Gathers starting at *base_addr* and offsets by elements in *vindex* (index is scaled by factor specified in *scale*). Scale depends on the data type of the element loaded.

```
Scatter: (void* base_addr, datatype vindex, datatype a, const int scale)
```

```
..B1.17:
        vloadunpackld (%r11,%r13,4), %zmm0{%k1}
```
Loading the multiple index values "b[i]". Note that the index for gather has to "integral" type
```
        kxnor       %k2, %k2
```
Setting k2 mask of all "1's" for Gather Operation
```
..L32:
        vgatherdpd (%r10,%zmm0,8), %zmm2{%k2}
        jkzd       ..L31, %k2    # Prob 50%
        vgatherdpd (%r10,%zmm0,8), %zmm2{%k2}
        jknzd      ..L32, %k2    # Prob 50%
```
Load (gather) the multiple values for the different loaded index to zmm2 [a[b[i]] . Internally mask "k2" is cleared (unset) for every loaded value. Mask "k2" of all 0's indicates gather completion
```
..L31:
        vmovaps     %zmm2, (%rbx,%r13,8)
        kxnor       %k3, %k3
```
Setting k3 mask of all "1's" for Gather Operation
```
      vloadunpackld (%r11,%r13,4), %zmm1{%k1}
```
Loading the multiple index values "b[i]". Note that the index for gather has to "integral" type
```
        addq        $8, %r13
        nop
..L34:
        vscatterdpd %zmm2, (%r10,%zmm1,8){%k3}
        jkzd        ..L33, %k3
        vscatterdpd %zmm2, (%r10,%zmm1,8){%k3}
        jknzd       ..L34, %k3
```
Store (gather) the multiple values from the different loaded index (zmm1) to mem addr Internally mask "k3" is cleared (unset) for every stored value. Mask "k3" of all 0's indicates scatter completion
```
..L33:
        cmpq        %rax, %r13
        jb          ..B1.17
```

FIG. 6.31

IMCI (Knights Corner) code for Fig. 6.30 Indirect Access Kernel Loop.

```
        vpxord      %zmm1, %zmm1, %zmm1
```
Clearing the contents of the zmm1 registers for Gather/Scatter Operation
```
        kxnorw      %k1, %k0, %k0
```
Setting k1 mask of all "1's" for Gather Operation
```
        kxnorw      %k2, %k0, %k0
```
Setting k2 mask of all "1's" for Scatter Operation
```
        vmovdqu     (%r11,%r13,4), %ymm0
```
Loading the multiple index values "b[i]". Note that the index for gather has to "integral" type
```
        vgatherdpd (%r8,%ymm0,8), %zmm1{%k1}
```
Load (gather) the multiple values for the different loaded index to zmm1 [a[b[i]] . Internally mask "k1" is cleared (unset) for every loaded value. Mask "k1" of all 0's indicates gather completion
```
        vmovups     %zmm1, (%r10,%r13,8)
        addq        $8, %r13
        vscatterdpd %zmm1, (%r8,%ymm0,8){%k2}
```
Store (scatter) the multiple values from the different loaded index (zmm1) to mem addr Internally mask "k2" is cleared (unset) for every stored value. Mask "k2" of all 0's indicates scatter completion
```
        cmpq        %r9, %r13
        jb            ..B1.17
```

FIG. 6.32

AVX-512 (Knights Landing) code for Fig. 6.30 Indirect Access Kernel Loop.


```
void Sparse() {
  int j;
  #pragma nounroll
  #pragma omp parallel for
  for (j=0; j<ARRAY_SIZE; j+=8) {
    //LOAD "b"
    __m512i Vb = _mm512_loadunpacklo_epi32 (
                        _mm512_undefined(), &b[j]);
    //GATHER "a"
    __m512d Vc = _mm512_i32logather_pd (Vb, &a[0], 8);
    //Store "c"
    _mm512_store_pd (&c[j], Vc);

    //Scatter "c" to "a"
    _mm512_i32loscatter_epi64 (&a[0], Vb, Vc, 8);
  }
}
```

FIG. 6.33

IMCI (Knights Corner) intrinsics code for Fig. 6.30 Indirect Access Kernel Loop.

```
void Sparse() {
  int j;
  #pragma nounroll
  #pragma vector aligned
  #pragma omp parallel for
  for (j=0; j<ARRAY_SIZE; j+=8){
    //LOAD "a"
    __m256i Vb = _mm256_loadu_si256 ((void*)&b[j]);
    //GATHER "a"
    __m512d Vc = _mm512_i32gather_pd (Vb, &a[0], 8);
    //STORE "c"
    _mm512_store_pd (&c[j], Vc);
    //Scatter "c" to "a"
    _mm512_i32scatter_pd (&a[0], Vb, Vc, 8);
  }
}
```

FIG. 6.34

AVX-512 (Knights Landing) intrinsics code for Fig. 6.30 Indirect Access Kernel Loop.

Scatters elements from "a" into memory starting at *base_addr* and offsets using index values in "vindex" (index is scaled by factor specified in *scale*). Scale depends on the data type of the element stored.

There are many intrinsic flavors of the gather/scatter instruction supporting the different data types and required conversions for the same. Chapter 12 has an excellent introduction to intrinsics, and information about the online Intrinsics Guide.

IMCI TO AVX-512: SWIZZLE INSTRUCTIONS

Swizzle is an operation to perform data element rearrangement/permutations of the source operands before execution. A temporary copy of the source operand with the swizzle is created which is fed to the ALU as the source of the operation. The temporary copy is destroyed after the operation.