CHAPTER 1

Introduction to Xeon Phi Architecture

Technical computing can be defined as the application of mathematical and computational principles to solve engineering and scientific problems. It has become an integral part of the research and development of new technologies in modern civilization. It is universally relied upon in all sectors of industry and all disciplines of academia for such disparate tasks as prototyping new products, forecasting weather, enhancing geosciences exploration, performing financial modeling, and simulating car crashes and the propagation of electromagnetic field from mobile phones.

Computer technology has made substantial progress over the past couple of decades by introducing superscalar processors with pipelined vector architecture. We have also seen the rise of parallel processing in the lowest computational segment, such as handheld devices. Today one can buy as much computational power as earlier supercomputers for less than a thousand dollars.

Current computational power still is not enough, however, for the type of research needed to push the edge of understanding of the physical and analytical processes addressed by technical computing applications. Massively parallel processors such as the Intel Xeon Phi product family have been developed to increase the computational power to remove these research barriers. Careful design of algorithm and data structures is needed to exploit the Intel *Many Integrated Core* (MIC) architecture of coprocessors capable of providing teraflops (trillions of mathematical operations per second) of double-precision floating-point performance. This book provides an in-depth look at the Intel Xeon Phi coprocessor architecture and the corresponding parallel data structure and algorithms used in the various technical computing applications for which it is suitable. It also examines the source code-level optimizations that can be performed to exploit features of the processor.

Processor microarchitecture describes the arrangements and relationship between different components to perform the computation. With the advent of semiconductor technologies, hardware companies were able to put many processing cores on a die and interconnect them intelligently to allow massive computing power in the modern range of teraflops of double-precision arithmetic. This type of computing power was achieved first by the supercomputer *Accelerated Strategic Computing Initiative* (ASCI) Red in the not-so-distant past in 1996.

This chapter will help you develop an understanding of the design decisions behind the Intel Xeon Phi coprocessor microarchitecture and how it complements the Intel Xeon product line. To that end, it provides a brief refresher of modern computer architecture and describes various aspects of the Intel Xeon Phi architecture at a high level. You will develop an understanding of Intel MIC architecture and how it addresses the massively parallel one-chip computational challenge. This chapter summarizes the capabilities and limitations of the Intel Xeon Phi coprocessor, as well as key impact points for software and hardware evaluators who are considering this platform for technical computing, and sets the stage for the deeper discussions in following chapters.

History of Intel Xeon Phi Development

Intel Xeon Phi started its gestation in 2004 when Intel processor architecture teams began looking for a solution to reduce the power consumption of the Intel Xeon family of processors developed around 2001. We ultimately determined in 2010 that the simple low-frequency Intel MIC architecture with appropriate software support would be able to produce better performance and watt efficiency. This solution required a new microarchitectural design. The question was: Could we use the x86 cores for it? The answer was yes, because the *instruction set architecture* (ISA) needed for x86 compatibility dictates a small percentage of power consumption, whereas the hardware implementation and circuit complexity drive most of the power dissipation in a general-purpose processor.

The architecture team experimented on a simulator with various architecture features—removing out-of-order execution, hardware multithreading, long vectors, and so forth—to develop a new architecture that could be applied to throughput-oriented workloads. A graphics workload fits throughput-oriented work nicely, as many threads can work in parallel to compute the final solution.

The design team focused on the in-order core, x86 ISA, a smaller pipeline, and wider *single instruction multiple data* (SIMD) and *symmetric multithreading* (SMT) units. So they started with Pentium 5 cores connected through a ring interface and added fixed-function units such as a texture sampler to help with graphics. The design goal was to create architecture with the proper balance between chip-level multiprocessing with thread and data-level parallelism. A simulator was used to anticipate various performance bottlenecks and tune the core and uncore designs (discussed in the next section).

In addition to understanding the use of such technology in graphics, Intel also recognized that scientific and engineering applications that are highly compute-intensive and thread- and process-scalable can benefit from manycore architecture. During this time period the high-performance computing (HPC) industry also started playing around with using graphics cards for general-purpose computation. It was obvious that there was promise to such technology.

Working with some folks at Intel Labs in 2009, I was able to demonstrate theoretically to our management and executive team that one could make some key computational kernels that would speed up quite a bit with such a low-frequency, highly-parallel architecture, such that overall application performance would improve even in a coprocessor model. This demonstration resulted in the funding of the project that led to Intel Xeon Phi development. The first work had started in 2005 on Larrabee 1 (Figure 1-1) as a graphics processor. The work proceeded in 2010 as a proof-of-concept prototype HPC coprocessor project code-named *Knights Ferry*. The visual computing product team within Intel started developing software for technical computing applications. Although the hardware did not change, their early drivers were based on graphics software needs and catered to graphics application programming interface (API) needs, which were mainly Windows-based at that point.

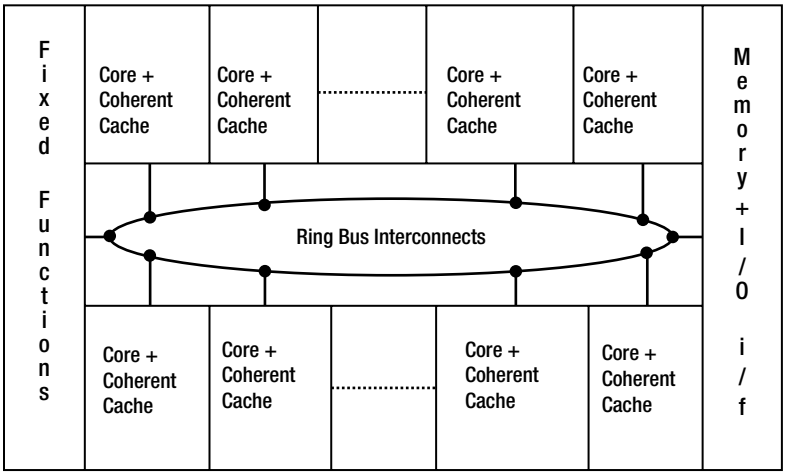

Figure 1-1. *Larrabee 1 silicon block diagram*

The first thing the software architects recognized was that a lot of technical and scientific computing is done on the Linux platform. So the first step was to create software support for Linux. We also needed to develop a programming language that could leverage the existing skills of the software developers to create multithreaded applications using Message Passing Interface (MPI) and OpenMP with the C, C++, and Fortran languages. The Intel compiler team went to the drawing board to define language extensions that would allow users to write applications that could run on coprocessors and host at the same time, leveraging the compute power of both. Other Intel teams went back to the design board to make tools and libraries—such as cluster tools (MPI), Debugger, Amplifier XE, Math Kernel Library, and Numeric—to support the new coprocessor architecture.

As the hardware consisted of x86 cores, the device driver team ported a modular microkernel that was based on standard Linux kernel source. The goal of the first phase of development was to prove and hash out the usability of the tools and language extensions that Intel was making. The goal was to come out with a hardware and software solution that could fill the needs of technical computing applications. The hardware roadmap included a new hardware architecture code-named *Knights Corner* (KNC) which could provide 1 teraflop of double-precision performance with the reliability and power management features required by such computations. This hardware was later marketed as *Intel® Xeon Phi™*—the subject of this book.

Evolution from Von Neumann Architecture to Cache Subsystem Architecture

There are various functional units in modern-day computer architecture that need to be carefully designed and developed to achieve target power and performance. The center of these functional units is a generic programmable processor that works in combination with other components such as memory, peripherals, and other coprocessors to perform its tasks. It is important to understand the basic computer architecture to get the grasp of Intel Xeon Phi architecture, since in essence the latter is a specialized architecture with many of the components used in designing a modern parallel computer.

Basic computer architecture is known as Von Neumann architecture. In this fundamental design, the processor is responsible for arithmetic and logic operations and gets its data and instructions from the memory (Figure 1-2). It fetches instructions from memory pointed to by an instruction pointer and executes the instruction. If the instruction needs data, it collects the data from the memory location pointed to by instruction and executes on them.

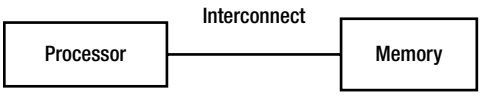

Figure 1-2. *Von Neumann architecture*

Over the past few decades, computer architecture has evolved from this basic Von Neumann architecture to accommodate physical necessities such as the need for faster data access to implement cache subsystems. Depending on the computational tasks at hand, demands are increasingly made upon various other elements of computer architecture. This book's focus is on Xeon Phi architecture in the context of scientific computing.

Modern scientific computing often depends on fast access to the data it needs. High-level processors are now designed with two distinct but important components known as the *core* and *uncore*. The core components consist of engines that do the computations. These include vector units in many of the modern processors. The uncore components includes cache, memory, and peripheral components. A couple of decades ago, the core was assumed to be the most important component of computer architecture and was subject to a lot of research and development. But in modern computers the uncore components play a more fundamental role in scientific application performance and often consume more power and silicon chip area than the core components.

General computer architecture with a cache subsystem is designed to reduce the memory bandwidth/latency bottleneck encountered in the Von Neumann architecture. A cache memory is a high-speed memory with low latency and a high-bandwidth connection to the core to supply data to instructions executing in the core. A subset of data currently being worked on by a computer program is saved in the cache to speed up instruction execution based on generally observed temporal and spatial locality of data accessed by computer programs. The general architecture of such a computer (Figure 1-3) entails the addition of a cache to the processor core and its communication through a memory controller (MC) with the main memory. The MC on modern chips is often fabricated on a die to reduce the memory access latency.

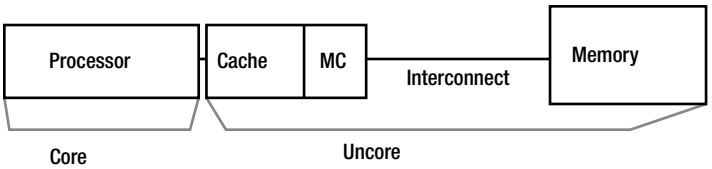

Figure 1-3. *Computer architecture with cache memory. The memory controller is responsible for managing data movement to and from the processor*

One common cache architecture design progression is to introduce and vary multiple levels of caches between the core and the main memory to reduce the access latency and interconnect bandwidth. Cache design continues to evolve in tandem with processor technology to mitigate memory bottlenecks. New memory technologies and semiconductor processes are allowing processor designers to play with various cache configurations as the architecture evolves.

The cache subsystem plays an extremely important role in application performance on a given computer architecture. In addition, the introduction of cache to speed-up applications causes a cache coherency problem in a manycore system. This problem results from the fact that the data updated in the cache may not reflect the data in the memory for the same variable. The coherency problem gets even more complex when the processor implements a multilevel cache.

There are various protocols designed to ensure that the data in the cache of each core of a multicore processor remain consistent when they are modified to maintain application correctness. One such protocol implemented in Intel Xeon Phi is described in Chapter 5.

During the development of the cache subsystem, the computer architecture remained inherently single-threaded from the hardware perspective, although clever time-sharing processes developed and supported in the computer operating systems gave the users the illusion of multiple processes being run by the computer simultaneously. I will explain in subsequent sections in this chapter how each of the components of the basic computer architecture shown in Figure 1-3—memory, interconnect, cache, and processor cores—has evolved in functionality to achieve the current version of Xeon Phi coprocessor architecture.

Improvements in the Core and Memory

To improve the single-threaded performance of programs, computer architects started looking at various mechanisms to reduce the amount of time it takes to execute each instruction, increase instruction throughput, and perform more work per instruction. These developments are described in this section.

Instruction-Level Parallelism

With the development of better semiconductor process technologies, computer architects were able to execute more and more instructions in a parallel and pipelined fashion, implementing what is known as *instruction-level parallelism*—the process of executing more than one instruction in parallel.

The instructions executed in a processor core go through several stages as they flow through logic circuits in sync with core clock pulses. At each clock pulse a part of the instruction is executed. It is possible, however, to stagger multiple instructions so that the various stages of multiple instructions can be executed in the same cycle. This is the principle behind pipelined executions.

All computer instructions based on Von Neumann architecture go through certain high-level basic stages. The first stage performs *instruction fetches* (IF), by which the next instruction to be executed by the core is accessed. The instructions usually reside in the instruction cache or are fetched from the main memory and cache hierarchy at this stage. Note that each stage will take a minimum of one cycle but may extend to further cycles if it gets blocked on some resource issue. For example, if the instructions to be executed are not in the cache, they have to be fetched from memory and, in the worst case, from a nonvolatile storage area such as a hard disk, solid state disk, or even flash memory.

Once the instructions have been fetched, they have to be decoded to understand how to execute the instructions. Now the instructions usually work on some sort of data, which might be in a processor register (the fastest memory nearest to the core), in a cache, or in a memory location. The semantics of the instructions are well defined by a set of rules and a behavioral model—namely, the instruction set architecture.[1]

A decoded instruction next moves to the execution (E) stage, where all necessary memory or cache access happens. The execution completes when all the necessary data are available. Otherwise, a pipeline stall might happen while waiting for data to come from the memory. Once the E stage completes, the data are written back (WB) to the memory/register or flags are updated to change the processor state.

Instruction Pipelining

The execution stage itself may take multiple cycles to accommodate the complexity of the semantics of that instruction. The fundamental pipelining described in the preceding section is shown in Figure 1-4. Note that this is a very simplified representation compared with the complex execution stages for the Xeon Phi processor that will be described in this book. Nonetheless, today's complex execution stages recapitulate the high-level classical instruction stages shown in Figure 1-4.

Figure 1-4. *Pipeline stages for an instruction execution. IF = instruction fetch; ID = instruction decode; EX = instruction execution; M = memory fetch; WB = write back, whereby the output of the instruction execution is written back to main memory*

Figure 1-5 shows how the pipelining process helps the respective stages of two different instructions to overlap, thus providing instruction-level parallelism. In this figure, the first instruction (inst1), after being fetched from memory, enters the instruction-decodes stage. Since these stages are executed in different hardware components, the second instruction fetch can happen while the first instruction is in the decode stage. So in clock (clk) tick 2, the first instruction is decoded and the second instruction is fetched, thus overlapping the execution of two instructions.

[1]How ISAs affect the overall performance and productivity of software systems developed for particular lines of computer hardware is an important research area but beyond the scope of this book.

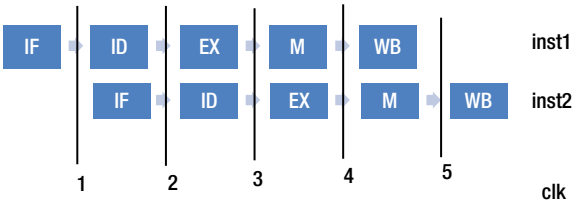

Figure 1-5. *Instruction pipeline showing two instructions executing at the same clock cycle but at different stages*

Processor engineers were, however, looking for more parallelism to satisfy the demand of computer users wanting to execute faster and more complex applications on these pieces of hardware. To further improve processor architecture, the architects designed the cores such that they could execute multiple instructions in parallel in the same cycle. In this case, some of the hardware functions were replicated so that, in addition to the pipelining shown in Figure 1-5, two independent instructions could be executed in two different pipelines. Thus they could both be in the execution stage at the same clock cycle. This architecture is known as *superscalar architecture*. One such architecture which was in wide use in early 1990s was Intel P5 architecture. The Intel Xeon Phi core is based on such architecture and contains two independent pipelines arbitrarily known as the *U* and *V* pipelines. Chapter 4 details how instructions are dispatched to these two pipelines, as well as some of the limitations of superscalar architecture.

Engineers kept increasing processor execution speed by increasing core clock frequencies. Increased clock rate required, however, that each of the stages described above be broken into several substages to be able to execute with each clock tick. Eventually the number of stages increased from the five basic stages shown in Figure 1-5 to over 30 stages to accommodate faster processor clock rate.[2] This increase resulted in faster and faster processors that could execute a single thread of instructions at a speed that was improving with clock rate improvement in each subsequent processor generation. The Intel Pentium 4 processors could run at 3.7GHz at 90nm technology during the 2004 launch date.[3] Given the technology limitations of that time, this was a great achievement.

But progress hit the "power wall": the increased clock rate was resulting in too much wasted energy in the form of heat. So engineers went back to the design board. Intel Xeon Phi instructions go through fewer stages (5 stages for best case execution) than the Pentium 4 families of processors (20 stages for the best case).

Another way to improve instruction-level parallelism was through the introduction of out-of-order instruction processing. In general the processor executes the code in the order generated by the compiler based on the source code provided by the programmer. If the instructions are independent in the instruction stream that is fed to the processor, however, it is possible to execute the instructions out of order—that is, the instruction that comes later in the compiler-generated code may be executed earlier than the instruction before it in the same code stream.

In out-of-order instruction execution, the hardware can detect independent instructions and execute them in any order that speeds up the instruction execution. This meant that the order of the instructions given in the source code by the programmer was not maintained by the execution unit. This was all right from the program-correctness perspective, since the instructions were executed in parallel or even earlier than the following instruction in the program order, independently of each other. This feat was achieved by increasing resources in the various stages of the processor—primarily in the dispatch and execution units of the pipeline. The processor was able to execute them out of order and, in many cases, in speculative fashion. For example, if there were a branch in the code stream, the processor could go ahead and execute both sides of the branch even though one of the branches were later thrown out, as that branch did not meet the actual branch criterion when it reached that point. To maintain the consistency semantics of program execution, which requires that the processor state should be in the order that the programmer desired in the original code, the WB stage was nevertheless maintained in the program order.

---

[2]Intel Pentium 4 Processors with Netburst Architecture (Codenamed Prescott) had a 31-stage pipeline (http://people.apache.org/~xli/presentations/history_Intel_CPU.pdf).

[3]http://ark.intel.com/products/27492/Pentium-4-Processor-Extreme-Edition-supporting-HT-Technology-3_73-GHz-2M-Cache-1066-MHz-FSB

Single Instruction Multiple Data

In order to increase parallelization within the hardware, the architects implemented a new hardware that allows you to work on multiple data items in parallel with a single instruction. Suppose, for example, that you have image-processing software in which you want to increase the brightness of every pixel by a certain amount. The computation involves working on consecutive bytes of data to be incremented by a certain value. Before the introduction of the *single instruction multiple data* (SIMD) feature, the hardware had to read one byte at a time and add the constant and write the data back. With the introduction of the *SIMD unit*, also commonly dubbed the *vector unit*, the hardware can now work on many bytes in the same cycle by one instruction.

As you will learn in Chapter 3, the vector unit in Intel Xeon Phi can work on 16 single-precision floating point values at the same time. This provided a big performance gain in applications that are *data-parallel*—meaning that the dataset being processed by the application has no dependencies among the data and can be processed at the same time.

Multithreading

As processor frequency was coming down to reduce the power dissipation resulting from high-speed switching, the engineers turned to hardware multithreading to increase parallelism. In this strategy, many processor resources are replicated in hardware, so that applications can indicate to the operating system that it can execute multiple instruction streams in parallel through high-level parallelism constructs such as OpenMP and Posix thread.

To the operating system, this looked like multiple processors working together to achieve the performance it wants. In the Intel Xeon Phi processor, there are four hardware threads sharing the same core as though there were four processors connected to a shared cache subsystem. Figure 1-6 shows the multithreading support in the core is displayed as logical processors, as they still share some resources among themselves.

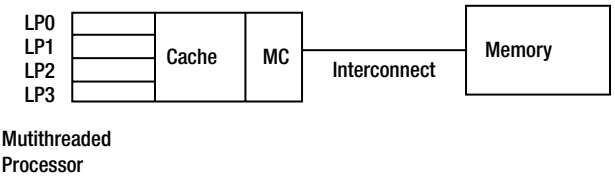

Figure 1-6. *Multithreaded processor cores with superscalar execution units. LP0–3 in the diagram indicate logical processors. MC indicates the memory controller controlling data flow to or from the logical processors*

Multicore and Manycore Architecture

A logical design evolution from multithreading's sharing of some of the resources needed for instruction execution was the cloning of the whole core multiple times to allow multiple threads of execution to happen in parallel. As a first step, architects cloned the big cores used in single-core processors multiple times to create *multicore processors*. These cores started life with a lower frequency than an equivalent single-core processor to limit the total power consumption of the chip. If the applications are properly parallelized, however, parallel processing provided a much bigger gain than the loss due to core frequency reduction. But big-core cloning is limited to a certain number of cores owing to the power envelope imposed by the physical process technology. In order to gain more parallelism, the architects needed to create simpler core running at even lower frequencies but numbered in the hundreds. This architecture is known as *manycore architecture* and the cores are often designated *small cores* as contrasted with the *big cores* used in manycore architecture. This massive level of parallelism of manycore architecture can be exploited only by the codes designed to run on such architecture. This type of manycore architecture, in which all cores are similar, is known as *homogeneous manycore architecture.* (There are other possibilities, such as *heterogeneous manycore architecture*, where all the cores in the processor may not be identical.)

The evolution to manycore architecture allowed applications to improve performance without increasing the clock frequencies. But this advance shifted the burden of achieving application performance improvement from hardware engineers toward software engineers. Software engineers and computer scientists leveraged years of experience in developing parallel applications used in technical computing and HPC applications to start exploiting the manycore architecture. Although the parallel constructs to exploit such machines are still in their infancy, there are sufficient tools to start developing for these machines. Figure 1-7 shows the initial thinking on architectures in which more than one core is made part of a processor, such that the processor cores P0–Pn are connected to a common interconnect known as a *bus* through the cache subsystem (C) and share the bus bandwidth with each other.

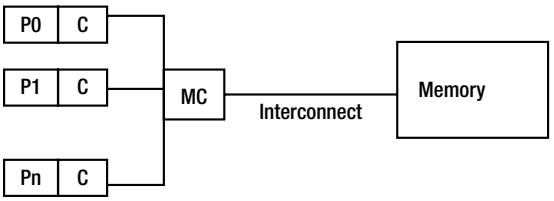

Figure 1-7. *Architecture evolving toward a manycore processor-based computer. C = cache; MC = memory controller; Px = processor cores*

Interconnect and Cache Improvements

In using multiple cores to create a processor, it was soon discovered that single shared-interconnect architectures, such as the bus used in some early processor designs, were a bottleneck for extracting parallel application performance.

The interconnect topology selected for a manycore processor is determined by the latency, bandwidth, and cost of implementing such technology. The interconnect technology chosen for Intel Xeon Phi is a bidirectional ring topology. Here all the cores talk to one another through a bidirectional interconnect. The cores also access the data and code residing in the main memory through the interconnect ring connecting the cores to memory controller. Chapter 5 examines the interconnect technology implemented in Xeon Phi. As new designs come out, the interconnect technology will also evolve to provide a low-latency/high-bandwidth network.

Figure 1-8 shows the evolution of the manycore processor depicted in Figure 1-7 so that the cores are connected in a ring network, which allows memory to be connected to the network through a memory controller responsible for getting data to the cores as requested. There may be one or more memory controllers to improve the memory bandwidth.

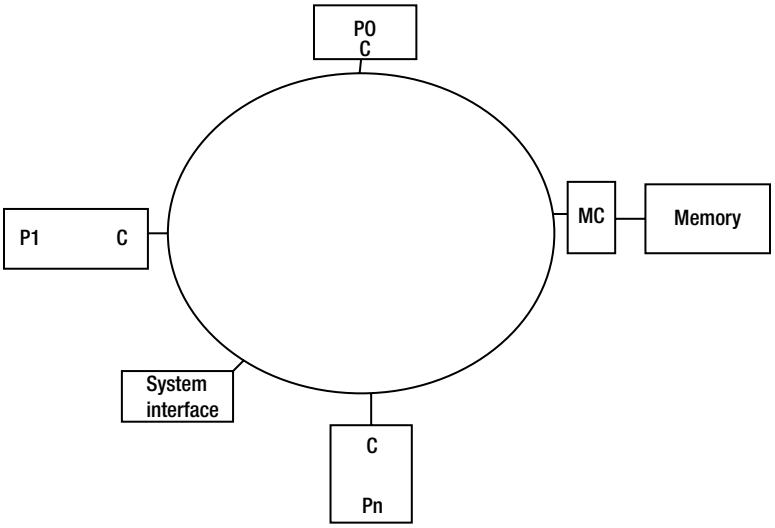

Figure 1-8. *Manycore processor architecture with cores connected through a ring bus. P0–Pn = cores; C = cache; MC = memory controller*

System Interconnect

In addition to talking to memory through memory interconnects, coprocessors such as Intel Xeon Phi are also often placed on *Peripheral Component Interconnect Express* (PCIe) slots to work with the host processors, such as Intel Xeon processors. This is done by incorporating a system interface logic that can support a standard *input/output* (I/O) protocol such as PCIe to communicate with the host. In Figure 1-8, the system interface controller is shown as another box connected to the ring.

Figure 1-9 shows a system-level view of Xeon Phi coprocessor working with a host processor over a PCIe interface. Note that the data movement between the host memory and Xeon Phi memory can happen through *direct memory access* (DMA) without host processor intervention in certain cases, which will be covered in Chapter 6. It is possible to connect multiple Intel Xeon Phi cards to the host system to increase computational power.

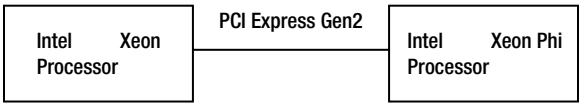

Figure 1-9. *System with the Intel Xeon Phi coprocessor*

Figure 1-10 shows the Intel Xeon Phi coprocessor packaged as a PCIe gen2-based card. The card with the fan is known as the *active-cooled* version, whereas the other is *passively cooled*. The passively cooled version of the card needs to be placed in a special server where the host system needs to provide sufficient cooling for the card. These cards can be placed on a validated server or workstation platforms in various configurations to complement the parallel processing power of host processors.

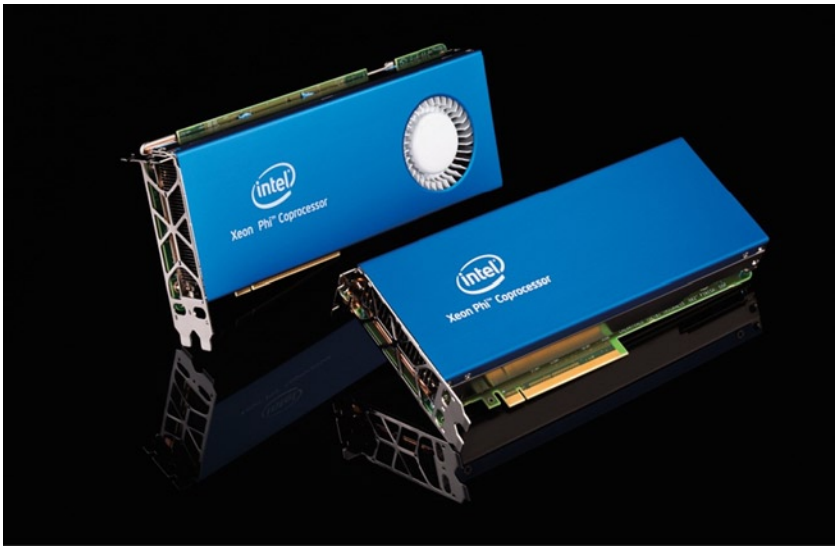

Figure 1-10. *Intel Xeon Phi coprocessors in actively-cooled and passively-cooled versions. (Source: http://newsroom.intel.com/docs/DOC-3126#multimedia)*

Intel Xeon Phi Coprocessor Chip Architecture

This section describes the various functional components of the Intel Xeon Phi coprocessor and explains why they are designed the way they are.

Figure 1-11 is a simple diagram of the logical layout of some of the critical chip components of the Intel Xeon Phi coprocessor architecture, which include the following:

- *coprocessor cores*: These are based on P54c (Intel Pentium from 1995) cores with major modifications including Intel 64 ISA, 4-way SMT, new vector instructions, and increased cache sizes.[4]
- *VPU*: The *vector processing units* are part of the core and capable of performing 512-bit vector operations on 16 single-precision or 8 double-precision floating-point arithmetic operations as well as integer operations.
- *L2 Cache*: The L2 cache and uncore interface.
- *tag directories* (TD): Components used to look up cache data distributed among the cores.
- *ring interconnect*: The interconnect between the cores and the rest of the coprocessor's components—memory controllers, PCI interface chip, and so on.
- *memory controller (MC)*: Interface between the ring and the *graphics double data rate* (GDDR) memory.
- *PCIe interface*: To connect with PCIe bus.

[4]Various other features in the coprocessor such as the debug features required to validate and debug the hardware will not be covered in this book.

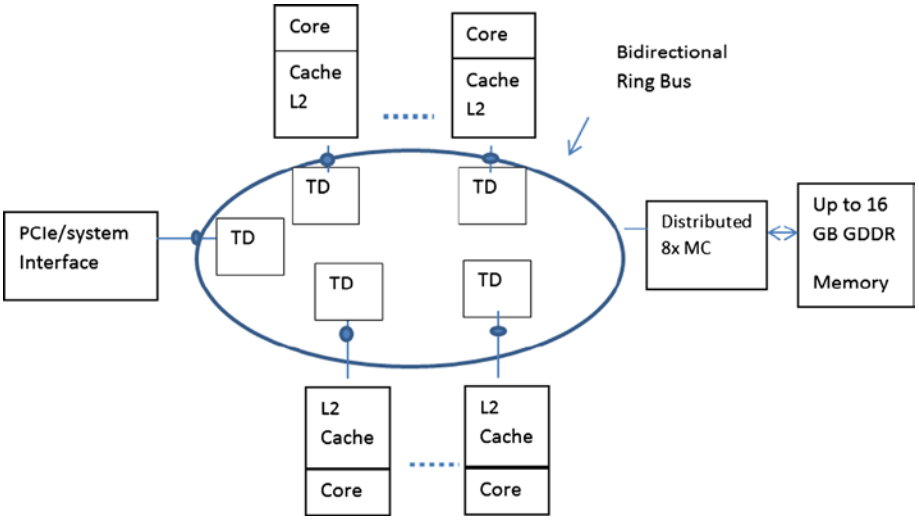

Figure 1-11. *Logical layouts of functional components*

Intel Xeon Phi consists of up to 61 Intel-architecture cores. For so many cores and functional units to access and communicate with each other, carefully designed interconnects are needed to hold the memory/data/control traffic between the cores and various parts of the chip. Figure 1-11 shows the logical layout of the Intel Xeon Phi coprocessor; the actual physical layout of the individual functional units may be vastly different from the depiction in the figure. For example, the eight memory controllers represented as "distributed 8x MC" in the figure are physically distributed on the ring for optimal memory access latency. The L2 caches are fully coherent with each other. Coherency is maintained by the globally owned and locally shared (GOALS) coherency protocols, described in Chapter 5. The functional units communicate with one another by on-die bidirectional interconnects.

For eight memory controllers with two GDDR5 channels running at 5.5 GT/s, one can compute the theoretical memory bandwidth as follows:

```
aggregate memory bandwidth = 8 memory controllers × 2 channels × 5.5 (GT/s) × 4
(bytes/transfer) = 352 GB/s
```

The system interface of the chip supports PCIe2 × 16 protocols with 256-byte packets.

The chip also provides reliability features useful in a technical computing environment. These include parity support in the L1 cache, *error correction code* (ECC) on L2 and memory transactions, *cyclic redundancy code* (CRC), and *command-address parity* on the memory I/O. Chapter 6 will provide further details on these.

Applicability of the Intel Xeon Phi Coprocessor

As seen in the preceding section, Xeon Phi is a manycore processor with up to 61 cores, with each core capable of performing 512-bit vector operations per cycle. The coprocessor card can also host up to 16GB of high-bandwidth memory. The card is, however, in a PCIe card form factor and must incur some overhead transferring data from the host processor or other nodes in a cluster. Another issue is that the cores of Xeon Phi run at about a third the speed of Intel Xeon processors, causing computations to be single-threaded. Hence the question customers often face: In which situations does it make sense to employ the Xeon Phi coprocessor-based model?

The card is not a replacement for the host processors. It has instead to be thought of a coprocessing element providing optimal power performance efficiency to the overall system. In order to achieve that power efficiency, the code must have certain characteristics that fit the hardware architecture, as follow:

- The fragment of the code that is offloaded or running on the coprocessor must be highly parallel and must scale with the number of cores. In short, the code fragment must be able to make use of all the available cores without keeping them idle. As many of the serial functions of the computation should be performed on the host as possible.
- The code must be efficiently vectorizable. It is not sufficient that the code simply be scalable to all the threads and cores in the Xeon Phi manycore coprocessor. The threads and cores must also be able to make use of the vector units efficiently. By efficient usage, I mean that the vector unit should not stall on the data, so that it can throughput at the rate Xeon Phi has been designed for.
- The code should be able to hide the I/O communication latency with the host by overlapping the I/O with the computation whenever possible. This is necessary because the host will be responsible for most of the data input and output to the permanent data storage and for managing network traffic to other nodes in a cluster or grid.

These three power-efficiency optimization characteristics will be explored in detail in Chapter 9. In my opinion, Xeon Phi's prime value is in providing the best power performance per node in a cluster. The chief benefit to be gained from adding Xeon Phi to your Xeon cluster of nodes is the increase in overall cluster performance resulting from improved performance per watt if you exploit Xeon Phi's architecture and tools in accordance with the programming guidance presented in this book.

Intel Xeon Phi is an architecture designed to enable scientific and technical computing applications in areas as disparate as weather forecasting, medical sciences, energy exploration, manufacturing, financial services, and academic research. All of these technical computing domains rely on applications that are highly parallel. If such applications are deemed a proper fit for the Xeon Phi architecture and if they are programmed properly, Xeon Phi enables a much higher performance and power efficiency than is attainable by host nodes only.

Summary

This chapter reviewed the development of the Intel Xeon Phi coprocessor and examined the thinking behind the evolution of Von Neumann basic architecture into a complex manycore design embedded in the current Xeon Phi architecture. The motive force driving this evolution has been the quest for the ever higher levels of processor performance necessary for executing the computational tasks that today underlie scientific discoveries and new technical applications.

The next chapter will delve into the programming for Xeon Phi.