CHAPTER

Introduction

1

CHAPTER OUTLINE

Microprocessors based on a single central processing unit (CPU), such as those in the Intel Pentium family and the AMD Opteron family, drove rapid performance increases and cost reductions in computer applications for more than two decades. These microprocessors brought GFLOPS, or giga (10^{12}) floating-point operations per second, to the desktop and TFLOPS, or tera (10^{15}) floating-point operations per second, to cluster servers. This relentless drive for performance improvement has allowed application software to provide more functionality, have better user interfaces, and generate more useful results. The users, in turn, demand even more improvements once they become accustomed to these improvements, creating a positive (virtuous) cycle for the computer industry.

This drive, however, has slowed since 2003 due to energy consumption and heat dissipation issues that limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Since then, virtually all microprocessor vendors have switched to models where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power. This switch has exerted a tremendous impact on the software developer community [Sutter2005].

Traditionally, the vast majority of software applications are written as sequential programs, as described by von Neumann in his seminal report in 1945 [vonNeumann1945]. The execution of these programs can be understood by a human sequentially stepping through the code. Historically, most software developers have relied on the advances in hardware to increase the speed of their sequential applications under the hood; the same software simply runs faster as each new generation of processors is introduced. Computer users have also become accustomed to the expectation that these programs run faster with each new generation of microprocessors. Such expectation is no longer valid from this day onward. A sequential program will only run on one of the processor cores, which will not become significantly faster than those in use today. Without performance improvement, application developers will no longer be able to introduce new features and capabilities into their software as new microprocessors are introduced, reducing the growth opportunities of the entire computer industry.

Rather, the applications software that will continue to enjoy performance improvement with each new generation of microprocessors will be parallel programs, in which multiple threads of execution cooperate to complete the work faster. This new, dramatically escalated incentive for parallel program development has been referred to as the concurrency revolution [Sutter2005]. The practice of parallel programming is by no means new. The high-performance computing community has been developing parallel programs for decades. These programs run on large-scale, expensive computers. Only a few elite applications can justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers. Now that all new microprocessors are parallel computers, the number of applications that need to be developed as parallel programs has increased dramatically. There is now a great need for software developers to learn about parallel programming, which is the focus of this book.

1.1 HETEROGENEOUS PARALLEL COMPUTING

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessors [Hwu2008]. The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began with two core processors with the number of cores increasing with each semiconductor process generation. A current exemplar is the recent Intel Core i7™ microprocessor with four processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full X86 instruction set, supporting hyper-threading with two hardware threads, designed to maximize the execution speed of sequential programs. In contrast, the *many-thread* trajectory focuses more on the execution throughput of parallel applications. The many-threads began with a large number of threads, and once again, the number of threads increases with each generation. A current exemplar is the NVIDIA GTX680 graphics processing unit (GPU) with 16,384 threads, executing in a large number of simple, in-order pipelines.

Many-threads processors, especially the GPUs, have led the race of floating-point performance since 2003. As of 2012, the ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs is about 10. These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips: 1.5 teraflops versus 150 gigaflops double precision in 2012.

Such a large performance gap between parallel and sequential execution has amounted to a significant "electrical potential" build-up, and at some point, something will have to give. We have reached that point now. To date, this large performance gap has already motivated many application developers to move the computationally intensive parts of their software to GPUs for execution. Not surprisingly, these computationally intensive parts are also the prime target of parallel programming—when there is more work to do, there is more opportunity to divide the work among cooperating parallel workers.

One might ask why there is such a large peak-performance gap between many-threads GPUs and general-purpose multicore CPUs. The answer lies in the differences in the fundamental design philosophies between the two types of processors, as illustrated in Figure 1.1. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed. As of 2012, the high-end general-purpose multicore microprocessors typically have six to eight large processor cores and multiple megabytes of on-chip cache memories designed to deliver strong sequential code performance.

Memory bandwidth is another important issue. The speed of many applications is limited by the rate at which data can be delivered from the

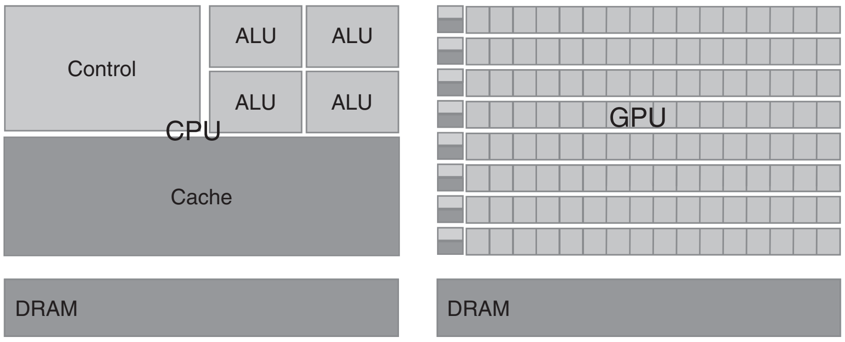

FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies.

memory system into the processors. Graphics chips have been operating at approximately six times the memory bandwidth of contemporaneously available CPU chips. In late 2006, GeForce 8800 GTX, or simply G80, was capable of moving data at about 85 gigabytes per second (GB/s) in and out of its main dynamic random-access memory (DRAM) because of graphics frame buffer requirements and the relaxed memory model (the way various system software, applications, and input/output (I/O) devices expect how their memory accesses work). The more recent GTX680 chip supports about 200 GB/s. In contrast, general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. As a result, CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

The design philosophy of GPUs is shaped by the fast-growing video game industry that exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced games. This demand motivates GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The prevailing solution is to optimize for the execution throughput of massive numbers of threads. The design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency. The reduced area and power of the memory access hardware and arithmetic units allows the designers to have more of them on a chip and thus increase the total execution throughput.

The application software is expected to be written with a large number of parallel threads. The hardware takes advantage of the large number of threads to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations. Small cache memories are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM. This design style is commonly referred to as throughput-oriented design since it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute.

The CPUs, on the other hand, are designed to minimize the execution latency of a single thread. Large last-level on-chip caches are designed to capture frequently accessed data and convert some of the long-latency memory accesses into short-latency cache accesses. The arithmetic units and operand data delivery logic are also designed to minimize the effective latency of operation at the cost of increased use of chip area and power. By reducing the latency of operations within the same thread, the CPU hardware reduces the execution latency of each individual thread. However, the large cache memory, low-latency arithmetic units, and sophisticated operand delivery logic consume chip area and power that could be otherwise used to provide more arithmetic execution units and memory access channels. This design style is commonly referred to as latency-oriented design.

It should be clear now that GPUs are designed as parallel, throughput-oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well. For programs that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs. When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs. Therefore, one should expect that many applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs. This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPU−GPU execution of an application.[1] The demand for supporting joint CPU−GPU execution is further reflected in more recent programming models such as OpenCL (see Chapter 14), OpenACC (see Chapter 15), and C++AMP (see Chapter 18).

It is also important to note that performance is not the only decision factor when application developers choose the processors for running their

[1]See Chapter 2 for more background on the evolution of GPU computing and the creation of CUDA.

applications. Several other factors can be even more important. First and foremost, the processors of choice must have a very large presence in the marketplace, referred to as the *installed base* of the processor. The reason is very simple. The cost of software development is best justified by a very large customer population. Applications that run on a processor with a small market presence will not have a large customer base. This has been a major problem with traditional parallel computing systems that have negligible market presence compared to general-purpose microprocessors. Only a few elite applications funded by government and large corporations have been successfully developed on these traditional parallel computing systems. This has changed with many-core GPUs. Due to their popularity in the PC market, GPUs have been sold by the hundreds of millions. Virtually all PCs have GPUs in them. There are more than 400 million CUDA-enabled GPUs in use to date. This is the first time that massively parallel computing is feasible with a mass-market product. Such a large market presence has made these GPUs economically attractive targets for application developers.

Another important decision factor is practical form factors and easy accessibility. Until 2006, parallel software applications usually ran on data center servers or departmental clusters. But such execution environments tend to limit the use of these applications. For example, in an application such as medical imaging, it is fine to publish a paper based on a 64-node cluster machine. But actual clinical applications on magnetic resonance imaging (MRI) machines have been based on some combination of a PC and special hardware accelerators. The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks of compute server boxes into clinical settings, while this is common in academic departmental settings. In fact, National Institutes of Health (NIH) refused to fund parallel programming projects for some time: they felt that the impact of parallel software would be limited because huge cluster-based machines would not work in the clinical setting. Today, GE ships MRI products with GPUs and NIH funds research using GPU computing.

Yet another important consideration in selecting a processor for executing numeric computing applications is the level of support for the Institute of Electrical and Electronic Engineers' (IEEE) floating-point standard. The standard makes it possible to have predictable results across processors from different vendors. While the support for the IEEE floating-point standard was not strong in early GPUs, this has also changed for new generations of GPUs since the introduction of the G80. As we will discuss in Chapter 7, GPU support for the IEEE floating-point standard has become comparable with that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable result values as the CPUs. Up to 2009, a major remaining issue was that the GPUs' floating-point arithmetic units were primarily single precision. Applications that truly require double-precision floating-point arithmetic units were not suitable for GPU execution. However, this has changed with the recent GPUs of which the double-precision execution speed approaches about half of that of single precision, a level that high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics API (application programming interface) functions to access the processor cores, meaning that OpenGL or Direct3D techniques were needed to program these chips. Stated more simply, a computation must be expressed as a function that paints a pixel in some way to execute on these early GPUs. This technique was called GPGPU (general-purpose programming using a graphics processing unit). Even with a higher-level programming environment, the underlying code still needs to fit into the APIs that are designed to paint pixels. These APIs limit the kinds of applications that one can actually write for early GPGPUs. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent research results.

But everything changed in 2007 with the release of CUDA [NVIDIA2007]. NVIDIA stared to devote silicon areas on their GPU chips to facilitate the ease of parallel programming.This did not represent software changes alone; additional hardware was added to the chips. In the G80 and its successor chips for parallel computing, CUDA programs no longer go through the graphics interface at all. Instead, a new general-purpose parallel programming interface on the silicon chip serves the requests of CUDA programs. The general-purpose programming interface greatly expands the types of applications that one can easily develop for GPUs. Moreover, all the other software layers were redone as well, so that the programmers can use the familiar C/C++ programming tools. Some of our students tried to do their lab assignments using the old OpenGL-based programming interface, and their experience helped them to greatly appreciate the improvements that eliminated the need for using the graphics APIs for computing applications.

1.2 ARCHITECTURE OF A MODERN GPU

Figure 1.2 shows the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Figure 1.3, two SMs form a building block. However, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Also, in Figure 1.3, each SM has a number of streaming processors (SPs) that share control logic and an instruction cache. Each GPU currently comes with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory in Figure 1.3. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images and texture information for 3D rendering. But for computing, they function as very high bandwidth off-chip memory, though with somewhat longer latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

The G80 introduced the CUDA architecture and had 86.4 GB/s of memory bandwidth, plus a communication link to the CPU core logic over a PCI-Express Generation 2 (Gen2) interface. Over PCI-E Gen2, a CUDA application can transfer data from the system memory to the global memory at 4 GB/s, and at the same time upload data back to the system memory at 4 GB/s. Altogether, there is a combined total of 8 GB/s. More recent GPUs use PCI-E Gen3, which supports 8 GB/s in each direction. As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E to communicate with the CPU system memory if there is need for using a library that is only available on the CPUs. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

With 16,384 threads, the GTX680 exceeds 1.5 teraflops in double precision. A good application typically runs 5,000–12,000 threads simultaneously on this chip. For those who are used to multithreading in CPUs, note that Intel CPUs support two or four threads, depending on the machine model, per core. CPUs, however, are increasingly used with SIMD (single instruction, multiple data) instructions for high numerical performance. The level of parallelism supported by both GPU hardware and CPU hardware is increasing quickly. It is therefore very important to strive for high levels of parallelism when developing computing applications.

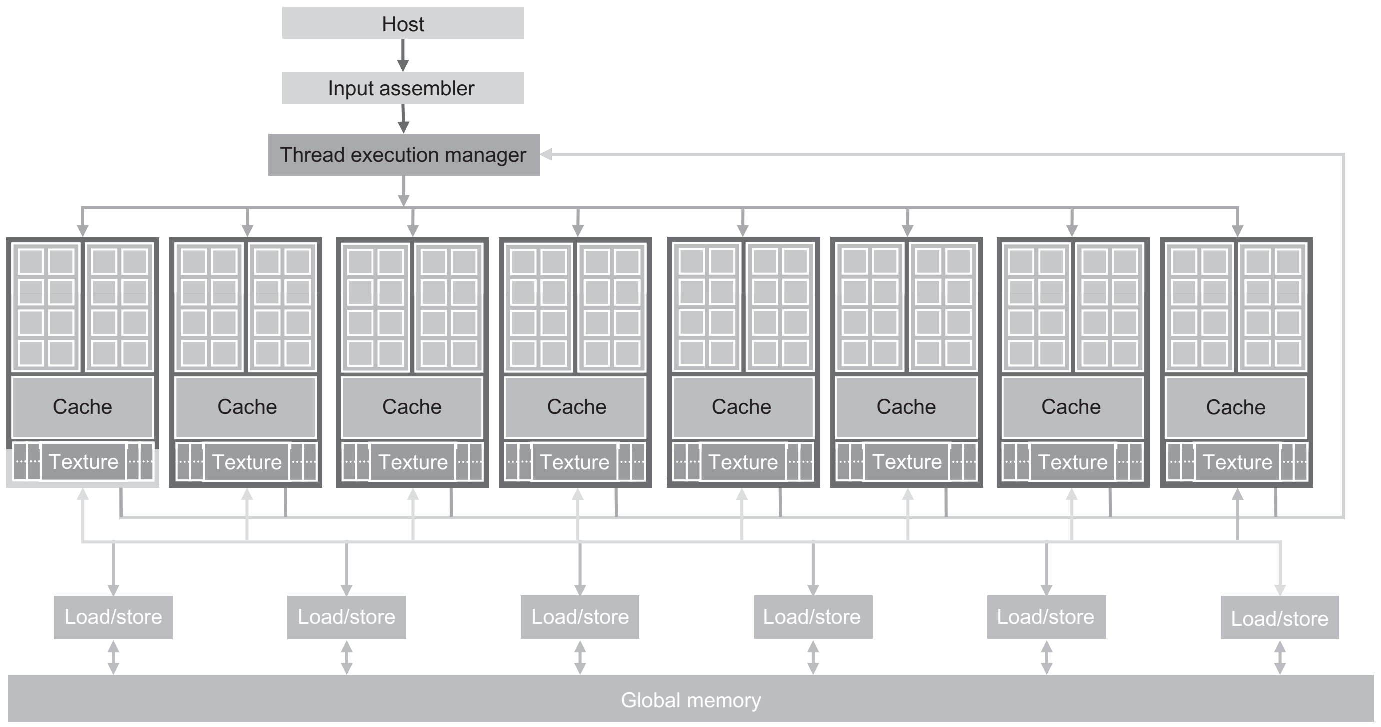

FIGURE 1.2

Architecture of a CUDA-capable GPU.

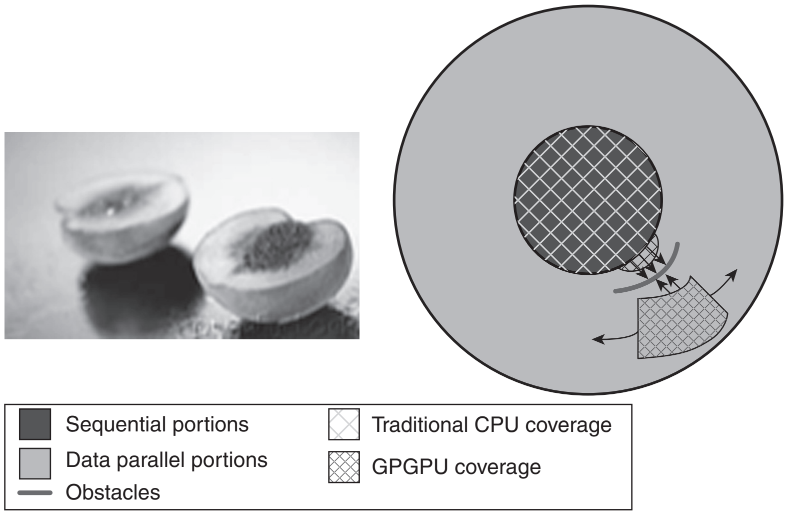

FIGURE 1.3

Coverage of sequential and parallel application portions.

1.3 WHY MORE SPEED OR PARALLELISM?

As we stated in Section 1.1, the main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations. One might ask why applications will continue to demand increased speed. Many applications that we have today seem to be running quite fast enough. As we will discuss in the case study chapters, when an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100 times (100×) speedup over sequential execution on a single CPU core. If the application includes what we call *data parallelism*, it's often a simple task to achieve a 10× speedup with just a few hours of work. For anything beyond that, we invite you to keep reading!

Despite the myriad of computing applications in today's world, many exciting mass-market applications of the future are what we currently consider "supercomputing applications," or super-applications. For example, the biology research community is moving more and more into the molecular level. Microscopes, arguably the most important instrument in molecular biology, used to rely on optics or electronic instrumentation. But there are limitations to the molecular-level observations that we can make with these instruments. These limitations can be effectively addressed by incorporating a computational model to simulate the underlying molecular activities with boundary conditions set by traditional instrumentation. With simulation we can measure even more details and test more hypotheses than can ever be imagined with traditional instrumentation alone. These simulations will continue to benefit from the increasing computing speed in the foreseeable future in terms of the size of the biological system that can be modeled and the length of reaction time that can be simulated within a tolerable response time. These enhancements will have tremendous implications to science and medicine.

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition (HD) TV verses older NTSC TV. Once we experience the level of details in an HDTV, it is very hard to go back to older technology. But consider all the processing that's needed for that HDTV. It is a very parallel process, as are 3D imaging and visualization. In the future, new functionalities such as view synthesis and high-resolution display of low-resolution videos will demand more computing power in the TV. At the consumer level, we will begin to have an increasing number of video and image processing applications that improve the focus, lighting, and other key aspects of the pictures and videos.

Among the benefits offered by more computing speed are much better user interfaces. Consider Apple's iPhone™ interfaces: the user enjoys a much more natural interface with the touchscreen than other cell phone devices even though the iPhone still has a limited-size window. Undoubtedly, future versions of these devices will incorporate higher-definition, 3D perspectives, applications that combine virtual and physical space information for enhanced usability, and voice-based and computer vision—based interfaces, requiring even more computing speed.

Similar developments are underway in consumer electronic gaming. In the past, driving a car in a game was in fact simply a prearranged set of scenes. If your car bumped into an obstacle, the course of your vehicle did not change, only the game score changed. Your wheels were not bent or damaged, and it was no more difficult to drive, regardless of whether you bumped your wheels or even lost a wheel. With increased computing speed, the games can be based on dynamic simulation rather than prearranged scenes. We can expect to see more of these realistic effects in the future: accidents will damage your wheels and your online driving experience will be much more realistic. Realistic modeling and simulation of physics effects are known to demand very large amounts of computing power.

All the new applications that we mention here involve simulating a physical, concurrent world in different ways and at different levels, with tremendous amounts of data being processed. In fact, the problem of handling massive amounts of data is so prevalent that the term *big data* has become a household word. And with this huge quantity of data, much of the computation can be done on different parts of the data in parallel, although they will have to be reconciled at some point. In most cases, effective management of data delivery can have a major impact on the achievable speed of a parallel application. While techniques for doing so are often well known to a few experts who work with such applications on a daily basis, the vast majority of application developers can benefit from more intuitive understanding and practical working knowledge of these techniques.

We aim to present the data management techniques in an intuitive way to application developers whose formal education may not be in computer science or computer engineering. We also aim to provide many practical code examples and hands-on exercises that help readers acquire working knowledge, which requires a practical programming model that facilitates parallel implementation and supports proper management of data delivery. CUDA offers such a programming model and has been well tested by a large developer community.

1.4 SPEEDING UP REAL APPLICATIONS

How many times speedup can be expected from parallelizing an application? It depends on the portion of the application that can be parallelized. If the percentage of time spent in the part that can be parallelized is 30%, a 100× speedup of the parallel portion will reduce the execution time by no more than 29.7%. The speedup for the entire application will be only about 1.4×. In fact, even an infinite amount of speedup in the parallel portion can only slash 30% off execution time, achieving no more than 1.43× speedup. On the other hand, if 99% of the execution time is in the parallel portion, a 100× speedup will reduce the application execution to 1.99% of the original time. This gives the entire application a 50× speedup. Therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speed up its execution.

Researchers have achieved speedups of more than 100× for some applications. However, this is typically achieved only after extensive optimization and tuning after the algorithms have been enhanced, so that more than 99.9% of the application execution time is in parallel execution. In practice, straightforward parallelization of applications often saturates the memory (DRAM) bandwidth, resulting in only about a 10× speedup. The trick is to figure out how to get around memory bandwidth limitations, which involves doing one of many transformations to utilize specialized GPU on-chip memories to drastically reduce the number of accesses to the DRAM. One must, however, further optimize the code to get around limitations such as limited on-chip memory capacity. An important goal of this book is to help readers fully understand these optimizations and become skilled in them.

Keep in mind that the level of speedup achieved over single-core CPU execution can also reflect the suitability of the CPU to the application: in some applications, CPUs perform very well, making it harder to speed up performance using a GPU. Most applications have portions that can be much better executed by the CPU. Thus, one must give the CPU a fair chance to perform and make sure that code is written so that GPUs *complement* CPU execution, thus properly exploiting the heterogeneous parallel computing capabilities of the combined CPU–GPU system. This is precisely what the CUDA programming model promotes, as we will further explain in the book.

Figure 1.3 illustrates the main parts of a typical application. Much of a real application's code tends to be sequential. These sequential parts are illustrated as the "pit" area of the peach: trying to apply parallel computing techniques to these portions is like biting into the peach pit—not a good feeling! These portions are very hard to parallelize. CPUs tend to do a very good job on these portions. The good news is that these portions, although they can take up a large portion of the code, tend to account for only a small portion of the execution time of super-applications.

Then come what we call the "peach meat" portions. These portions are easy to parallelize, as are some early graphics applications. Parallel programming in heterogeneous computing systems can drastically improve the quality of these applications. As illustrated in Figure 1.3, early GPGPUs cover only a small portion of the meat section, which is analogous to a small portion of the most exciting applications. As we will see, the CUDA programming models are designed to cover a much larger section of the peach meat portions of exciting applications. In fact, as we will discuss in Chapter 20, these programming models and their underlying hardware are still evolving at a fast pace to enable efficient parallelization of even larger sections of applications.

1.5 PARALLEL PROGRAMMING LANGUAGES AND MODELS

Many parallel programming languages and models have been proposed in the past several decades Mattson2004]. The ones that are the most widely used are Message Passing Interface (MPI) [MPI2009] for scalable cluster computing, and OpenMP [Open2005] for shared-memory multiprocessor systems. Both have become standardized programming interfaces supported by major computer vendors. An OpenMP implementation consists of a compiler and a runtime. A programmer specifies directives (commands) and pragmas (hints) about a loop to the OpenMP compiler. With these directives and pragmas, OpenMP compilers generate parallel code. The runtime system supports the execution of the parallel code by managing parallel threads and resources. OpenMP was originally designed for CPU execution. More recently, a variation called OpenACC (see Chapter 15) has been proposed and supported by multiple computer vendors for programming heterogeneous computing systems.

The major advantage of OpenACC is that it provides compiler automation and runtime support for abstracting away many parallel programming details from programmers. Such automation and abstraction can help make the application code more portable across systems produced by different vendors, as well as different generations of systems from the same vendor. This is why we teach OpenACC programming in Chapter 15. However, effective programming in OpenACC still requires the programmers to understand all the detailed parallel programming concepts involved. Because CUDA gives programmers explicit control of these parallel programming details, it is an excellent learning vehicle even for someone who would like to use OpenMP and OpenACC as their primary programming interface. Furthermore, from our experience, OpenACC compilers are still evolving and improving. Many programmers will likely need to use CDUA-style interfaces for parts where OpenACC compilers fall short.

MPI is a model where computing nodes in a cluster do not share memory [MPI2009]. All data sharing and interaction must be done through explicit message passing. MPI has been successful in high-performance computing (HPC). Applications written in MPI have run successfully on cluster computing systems with more than 100,000 nodes. Today, many HPC clusters employ heterogeneous CPU−GPU nodes. While CUDA is an effective interface with each node, most application developers need to use MPI to program at the cluster level. Therefore, it is important that a parallel programmer in HPC understands how to do joint MPI/CUDA programming, which is presented in Chapter 19.

The amount of effort needed to port an application into MPI, however, can be quite high due to the lack of shared memory across computing nodes. The programmer needs to perform domain decomposition to partition the input and output data into cluster nodes. Based on the domain decomposition, the programmer also needs to call message sending and receiving functions to manage the data exchange between nodes. CUDA, on the other hand, provides shared memory for parallel execution in the GPU to address this difficulty. As for CPU and GPU communication, CUDA previously provided very limited shared memory capability between the CPU and the GPU. The programmers needed to manage the data transfer between the CPU and the GPU in a manner similar to the "one-sided" message passing. New runtime support for global address space and automated data transfer in heterogeneous computing systems, such as GMAC [GCN2010] and CUDA 4.0, are now available. With GMAC, a CUDA or OpenCL programmer can declare C variables and data structures as shared between CPU and GPU. The GMAC runtime maintains coherence and automatically performs optimized data transfer operations on behalf of the programmer on an as-needed basis. Such support significantly reduces the CUDA and OpenCL programming complexity involved in overlapping data transfer with computation and I/O activities.

In 2009, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, jointly developed a standardized programming model called Open Compute Language (OpenCL) [Khronos2009]. Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors. In comparison to CUDA, OpenCL relies more on APIs and less on language extensions than CUDA. This allows vendors to quickly adapt their existing compilers and tools to handle OpenCL programs. OpenCL is a standardized programming model in that applications developed in OpenCL can run correctly without modification on all processors that support the OpenCL language extensions and API. However, one will likely need to modify the applications to achieve high performance for a new processor.

Those who are familiar with both OpenCL and CUDA know that there is a remarkable similarity between the key concepts and features of OpenCL and those of CUDA. That is, a CUDA programmer can learn OpenCL programming with minimal effort. More importantly, virtually all techniques learned using CUDA can be easily applied to OpenCL programming. Therefore, we introduce OpenCL in Chapter 14 and explain how one can apply the key concepts in this book to OpenCL programming.

1.6 OVERARCHING GOALS

Our primary goal is to teach the readers how to program massively parallel processors to achieve high performance, and our approach will not require a great deal of hardware expertise. Someone once said that if you don't care about performance, parallel programming is very easy. You can literally write a parallel program in an hour. But we're going to dedicate many pages to techniques for developing *high-performance* parallel programs. And, we believe that it will become easy once you develop the right insight and go about it the right way. In particular, we will focus on *computational thinking* techniques that will enable you to think about problems in ways that are amenable to high-performance parallel computing.

Note that hardware architecture features have constraints. High-performance parallel programming on most processors will require some knowledge of how the hardware works. It will probably take 10 or more years before we can build tools and machines so that most programmers can work without this knowledge. Even if we have such tools, we suspect that programmers with more knowledge of the hardware will be able to use the tools in a much more effective way than those who do not. However, we will not be teaching computer architecture as a separate topic. Instead, we will teach the essential computer architecture knowledge as part our discussions on high-performance parallel programming techniques.

Our second goal is to teach parallel programming for correct functionality and reliability, which constitute a subtle issue in parallel computing. Those who have worked on parallel systems in the past know that achieving initial performance is not enough. The challenge is to achieve it in such a way that you can debug the code and support users. The CUDA programming model encourages the use of a simple form of barrier synchronization and memory consistency for managing parallelism. We will show that by focusing on data parallelism, one can achieve both high performance and high reliability in their applications.

Our third goal is scalability across future hardware generations by exploring approaches to parallel programming such that future machines, which will be more and more parallel, can run your code faster than today's machines. We want to help you master parallel programming so that your programs can scale up to the level of performance of new generations of machines. The key to such scalability is to regularize and localize memory data accesses to minimize consumption of critical resources and conflicts in accessing and updating data structures.

Much technical knowledge will be required to achieve these goals, so we will cover quite a few principles and patterns of parallel programming in this book. We cannot guarantee that we will cover all of them, however, so we have selected the most useful and well-proven techniques to cover in detail. To complement your knowledge and expertise, we include a list of recommended literature. We are now ready to give you a quick overview of the rest of the book.

1.7 ORGANIZATION OF THE BOOK

Chapter 2 reviews the history of GPU computing. It starts with a brief summary of the evolution of graphics hardware toward more programmability and then discusses the historical GPGPU movement. Many of the current features and limitations of the CUDA programming model find their root in these historic developments. A good understanding of these historic developments will help readers better understand the current state and the future trends of hardware evolution that will continue to impact the types of applications that will benefit from CUDA.

Chapter 3 introduces data parallelism and the CUDA C programming. This chapter relies on the fact that students have had previous experience with C programming. It first introduces CUDA C as a simple, small extension to C that supports heterogeneous CPU−GPU joint computing and the widely used SPMD (single program, multiple data) parallel programming model. It then covers the thought process involved in (1) identifying the part of application programs to be parallelized; (2) isolating the data to be used by the parallelized code, using an API (Application Programming Interface) function to allocate memory on the parallel computing device; (3) using an API function to transfer data to the parallel computing device; (4) developing a kernel function that will be executed by threads in the parallelized part; (5) launching a kernel function for execution by parallel threads; and (6) eventually transferring the data back to the host processor with an API function call.

While the objective of Chapter 3 is to teach enough concepts of the CUDA C programming model so that the students can write a simple parallel CUDA C program, it actually covers several basic skills needed to develop a parallel application based on any parallel programming model. We use a running example of vector addition to make this chapter concrete. We also compare CUDA with other parallel programming models including OpenMP and OpenCL.

Chapter 4 presents more details of the parallel execution model of CUDA. It gives enough insight into the creation, organization, resource binding, data binding, and scheduling of threads to enable readers to implement sophisticated computation using CUDA C and reason about the performance behavior of their CUDA code. Chapter 5 is dedicated to the special memories that can be used to hold CUDA variables for managing data delivery and improving program execution speed.

Chapter 6 presents several important performance considerations in current CUDA hardware. In particular, it gives more details in thread execution, memory data accesses, and resource allocation. These details form the conceptual basis for programmers to reason about the consequence of their decisions on organizing their computation and data.

Chapter 7 introduces the concepts of floating-point number format, precision, and accuracy. It shows why different parallel execution arrangements can result in different output values. It also teaches the concept of numerical stability and practical techniques for maintaining numerical stability in parallel algorithms.

Chapters 8-10 present three important parallel computation patterns that give readers more insight into parallel programming techniques and parallel execution mechanisms. Chapter 8 presents convolution, a frequently used parallel computing pattern that requires careful management of data access locality. We also use this pattern to introduce constant memory and caching in modern GPUs. Chapter 9 presents prefix sum, or scan, an important parallel computing pattern that coverts sequential computation into parallel computation. We also use this pattern to introduce the concept of work efficiency in parallel algorithms. Chapter 10 presents sparse matrix computation, a pattern used for processing very large data sets. This chapter introduces readers to the concepts of rearranging data for more efficient parallel access: padding, sorting, transposition, and regularization.

While these chapters are based on CUDA, they help readers build up the foundation for parallel programming in general. We believe that humans understand best when we learn from the bottom up. That is, we must first learn the concepts in the context of a particular programming model, which provides us with solid footing when we generalize our knowledge to other programming models. As we do so, we can draw on our concrete experience from the CUDA model. An in-depth experience with the CUDA model also enables us to gain maturity, which will help us learn concepts that may not even be pertinent to the CUDA model.

Chapters 11 and 12 are case studies of two real applications, which take readers through the thought process of parallelizing and optimizing their applications for significant speedups. For each application, we start by identifying alternative ways of formulating the basic structure of the parallel execution and follow up with reasoning about the advantages and disadvantages of each alternative. We then go through the steps of code transformation needed to achieve high performance. These two chapters help readers put all the materials from the previous chapters together and prepare for their own application development projects.

Chapter 13 generalizes the parallel programming techniques into problem decomposition principles, algorithm strategies, and computational thinking. It does so by covering the concept of organizing the computation tasks of a program so that they can be done in parallel. We start by discussing the translational process of organizing abstract scientific concepts into computational tasks, which is an important first step in producing quality application software, serial or parallel. It then discusses parallel algorithm structures and their effects on application performance, which is grounded in the performance tuning experience with CUDA. The chapter concludes with a treatment of parallel programming styles and models, enabling readers to place their knowledge in a wider context. With this chapter, readers can begin to generalize from the SPMD programming style to other styles of parallel programming, such as loop parallelism in OpenMP and fork-join in p-thread programming. Although we do not go into these alternative parallel programming styles, we expect that readers will be able to learn to program in any of them with the foundation they gain in this book.

Chapter 14 introduces the OpenCL programming model from a CUDA programmer's perspective. Readers will find OpenCL to be extremely similar to CUDA. The most important difference arises from OpenCL's use of API functions to implement functionalities such as kernel launching and thread identification. The use of API functions makes OpenCL more tedious to use. Nevertheless, a CUDA programmer has all the knowledge and skills needed to understand and write OpenCL programs. In fact, we believe that the best way to teach OpenCL programming is to teach CUDA first. We demonstrate this with a chapter that relates all major OpenCL features to their corresponding CUDA features. We also illustrate the use of these features by adapting our simple CUDA examples into OpenCL.

Chapter 15 presents the OpenACC programming interface. It shows how to use directives and pragmas to tell the compiler that a loop can be parallelized, and if desirable, instruct the compiler how to parallelize the loop. It also uses concrete examples to illustrate how one can take advantage of the interface and make their code more portable across vendor systems. With the foundational concepts in this book, readers will find the OpenACC programming directives and pragmas easy to learn and master.

Chapter 16 covers Thrust, a productivity-oriented C++ library for building CUDA applications. This is a chapter that shows how modern object-oriented programming interfaces and techniques can be used to increase productivity in a parallel programming environment. In particular, it shows how generic programming and abstractions can significantly reduce the efforts and code complexity of applications.

Chapter 17 presents CUDA FORTRAN, an interface that supports FORTRAN-style programming based on the CUDA model. All concepts and techniques learned using CUDA C can be applied when programming in CUDA. In addition, the CUDA FORTRAN interface has strong support for multidimensional arrays that make programming of 3D models much more readable. It also assumes the FORTRAN array data layout convention and works better with an existing application written in FORTRAN.

Chapter 18 is an overview of the C++AMP programming interface from Microsoft. This programming interface uses a combination language extension and API support to support data-parallel computation patterns. It allows programmers to use C++ features to increase their productivity. Like OpenACC, C++AMP abstracts away some of the parallel programming details that are specific to the hardware so the code is potentially more portable across vendor systems.

Chapter 19 presents an introduction to joint MPI/CUDA programming. We cover the key MPI concepts that a programmer needs to understand to scale their heterogeneous applications to multiple nodes in a cluster environment. In particular, we will focus on domain partitioning, point-to-point communication, collective communication in the context of scaling a CUDA kernel into multiple nodes.

Chapter 20 introduces the dynamic parallelism capability available in the Kepler GPUs and their successors. Dynamic parallelism can potentially help the implementations of sophisticated algorithms to reduce CPU-GPU interaction overhead, free up CPU for other tasks, and improve the utilization of GPU execution resources. We describe the basic concepts of dynamic parallelism and why some algorithms can benefit from dynamic parallelism. We then illustrate the usage of dynamic parallelism with a small contrived code example as well as a more complex realistic code example.

Chapter 21 offers some concluding remarks and an outlook for the future of massively parallel programming. We first revisit our goals and summarize how the chapters fit together to help achieve the goals. We then present a brief survey of the major trends in the architecture of massively parallel processors and how these trends will likely impact parallel programming in the future. We conclude with a prediction that these fast advances in massively parallel computing will make it one of the most exciting areas in the coming decade.

References

Gelado, I., Cabezas, J., Navarro, N., Stone, J. E., Patel, S. J., & Hwu, W. W. An Asynchronous Distributed Shared Memory Model for Heterogeneous Parallel Systems, International Conference on Architectural Support for Programming Languages and Operating Systems, March 2010. Technical Report, IMPACT Group, University of Illinois, Urbana-Champaign.

Hwu, W. W., Keutzer, K., & Mattson, T. (2008). The Concurrency Challenge. *IEEE Design and Test of Computers,* July/August 312−320.

The Khronos Group, The OpenCL Specification Version 1.0, Available at: (*http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf*).

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). Patterns of Parallel Programming Boston: Addison-Wesley Professional.

Message Passing Interface Forum, "MPI—A Message Passing Interface Standard Version 2.2," Available at: <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>., Sept. 4, 2009.

NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.0, June 2007, Available at: <http://www.cs.berkeley.edu/~yelick/cs194f07/handouts/NVIDIA_CUDA_Programming_Guide.pdf>

OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1." July 2011, Available at: <http://www.openmp.org/mp-documents/OpenMP3.1.pdf>

Sutter, H., & Larus,, J. (2005). Software and the Concurrency Revolution. *ACM Queue*, *3*(7), 54−62.

von Neumann, J. (1945). First Draft of a Report on the EDVAC. In H. H. Goldstine (Ed.), *The Computer: From Pascal to von Neumann*. Princeton, NJ: Princeton University Press.

Wing, J. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33−35.