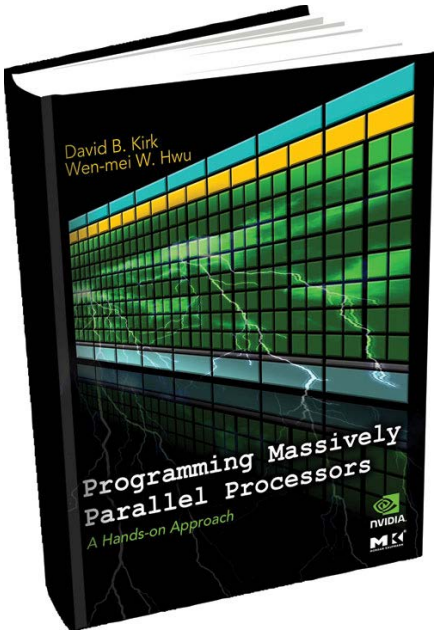
Project Option 3 of 6 – This is a project for a team of 2 students.

Topic: An Overview of GPU

The following content is from the book “Programming Massively Parallel Processors: A Hands-on Approach” written by David Kirk and Wen-mei W. Hwu.



Read Sections 1.1 to 1.3 of the book, and write a report of about 3 pages long. In your report, you should discuss the following content:

The differences between GPU and CPU

The demand of graphical performance on modern computers

The architecture of a modern GPU

The characteristics of CUDA™

You should also prepare for a presentation of 15~20 minutes long. You are responsible to make your own slides.

CHAPTER

Introduction 1

CHAPTER CONTENTS

INTRODUCTION

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 giga (billion) floating-point operations per second (GFLOPS) to the desktop and hundreds of GFLOPS to cluster servers. This relentless drive of 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 cycle for the computer industry.

During the drive, most software developers have relied on the advances in hardware to increase the speed of their applications under the hood; the same software simply runs faster as each new generation of processors is introduced. This drive, however, has slowed since 2003 due to energy-consumption and heat-dissipation issues that have 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. 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 [Sutter 2005].

Traditionally, the vast majority of software applications are written as sequential programs, as described by von Neumann [1945] in his seminal report. The execution of these programs can be understood by a human sequentially stepping through the code. Historically, computer users have become accustomed to the expectation that these programs run faster with each new generation of microprocessors. Such expectation is no longer strictly 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, thus 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* [Sutter 2005]. 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 must 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 GPUs AS PARALLEL COMPUTERS

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessor [Hwu 2008]. The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began as two-core processors, with the number of cores approximately doubling with each semiconductor process generation. A current exemplar is the recent Intel® Core™ i7 microprocessor,

which has four processor cores, each of which is an out-of-order, multiple-instruction issue processor implementing the full x86 instruction set; the microprocessor supports hyperthreading with two hardware threads and is designed to maximize the execution speed of sequential programs.

In contrast, the *many-core* trajectory focuses more on the execution throughput of parallel applications. The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation. A current exemplar is the NVIDIA® GeForce® GTX 280 graphics processing unit (GPU) with 240 cores, each of which is a heavily multithreaded, in-order, single-instruction issue processor that shares its control and instruction cache with seven other cores. Many-core processors, especially the GPUs, have led the race of floating-point performance since 2003. This phenomenon is illustrated in Figure 1.1. While the performance improvement of general-purpose microprocessors has slowed significantly, the GPUs have continued to improve relentlessly. As of 2009, the ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 10 to 1. These are not necessarily achievable application speeds but are merely the raw speed that the execution resources can potentially support in these chips: 1 teraflops (1000 gigaflops) versus 100 gigaflops in 2009.

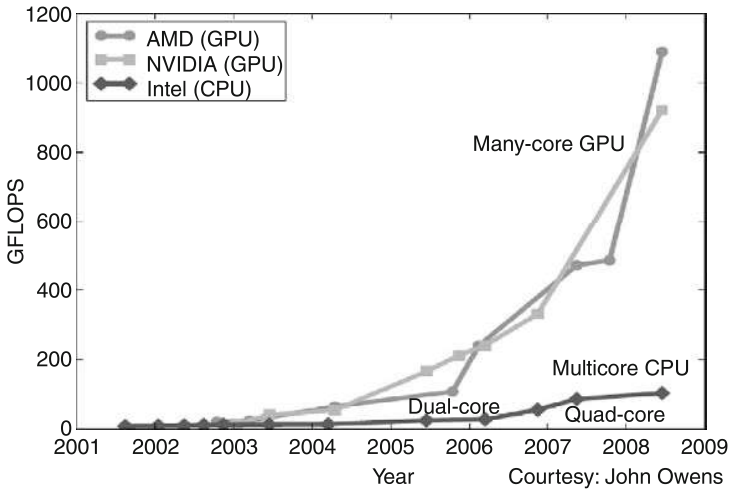


FIGURE 1.1

Enlarging performance gap between GPUs and CPUs.

Such a large performance gap between parallel and sequential execution has amounted to a significant "electrical potential" buildup, 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 applications 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 performance gap between many-core 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 1.2. 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 of execution 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 2009, the new general-purpose, multicore microprocessors typically have four large processor cores designed to deliver strong sequential code performance.

Memory bandwidth is another important issue. Graphics chips have been operating at approximately 10 times the bandwidth of contemporaneously available CPU chips. In late 2006, the GeForce® 8800 GTX, or simply

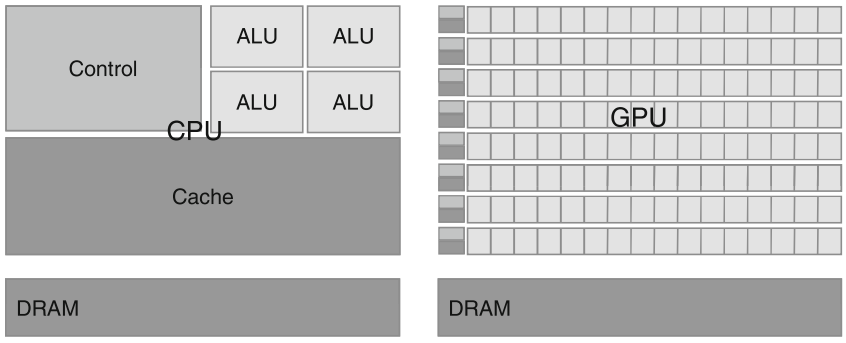


FIGURE 1.2

CPUs and GPUs have fundamentally different design philosophies.

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 frame buffer requirements and the relaxed memory model—the way various system software, applications, and input/output (I/O) devices expect their memory accesses to work—general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. In contrast, with simpler memory models and fewer legacy constraints, the GPU designers can more easily achieve higher memory bandwidth. The more recent NVIDIA® GT200 chip supports about 150 GB/s. Microprocessor system memory bandwidth will probably not grow beyond 50 GB/s for about 3 years, so CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

The design philosophy of the GPUs is shaped by the fast growing video game industry, which 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 the GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The prevailing solution to date is to optimize for the execution throughput of massive numbers of threads. The hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, thus minimizing the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications so multiple threads that access the same memory data do not need to all go to the DRAM. As a result, much more chip area is dedicated to the floating-point calculations.

It should be clear now that GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well; therefore, one should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs. This is why the CUDA™ (Compute Unified Device Architecture) programming model, introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.[1]

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

It is also important to note that performance is not the only decision factor when application developers choose the processors for running their 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 installation 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 the advent of many-core GPUs. Due to their popularity in the PC market, hundreds of millions of GPUs have been sold. Virtually all PCs have GPUs in them. The G80 processors and their successors have shipped more than 200 million units to date. This is the first time that massively parallel computing has been feasible with a mass-market product. Such a large market presence has made these GPUs economically attractive for application developers.

Other important decision factors are 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 are all 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 clusters to clinical settings, but this is common in academic departmental settings. In fact, the 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 support for the Institute of Electrical and Electronics Engineers (IEEE) floating-point standard. The standard makes it possible to have predictable results across processors from different vendors. While 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 to that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable values as the CPUs. Today, a major remaining issue is that the floating-point arithmetic units of the GPUs are primarily single precision. Applications that truly require double-precision floating point were not suitable for GPU execution; however, this has changed with the recent GPUs, whose double-precision execution speed approaches about half 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 graphic application programming interface (API) functions to access the processor cores, meaning that OpenGL® or Direct3D® techniques were needed to program these chips. This technique was called GPGPU, short for general-purpose programming using a graphics processing unit. Even with a higher level programming environment, the underlying code is still limited by the APIs. These APIs limit the kinds of applications that one can actually write for these chips. That's why only a few people could master the skills necessary to use these chips to achieve performance for a limited number of applications; consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent results.

Everything changed in 2007 with the release of CUDA [NVIDIA 2007]. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming, so this did not represent a change in software alone; additional hardware was added to the chip. 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. Moreover, all of the other software layers were redone, as well, so 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 1.3 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, each SM in Figure 1.3 has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 4 gigabytes of graphics 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 three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

The G80 that introduced the CUDA architecture had 86.4 GB/s of memory bandwidth, plus an 8-GB/s communication bandwidth with the CPU. A CUDA application can transfer data from the system 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. The communication bandwidth is much lower than the memory bandwidth and may seem like a limitation; however, the PCI Express® bandwidth is comparable to the CPU front-side bus bandwidth to the system memory, so it's really not the limitation it would seem at first. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

The massively parallel G80 chip has 128 SPs (16 SMs, each with 8 SPs). Each SP has a multiply–add (MAD) unit and an additional multiply unit. With 128 SPs, that's a total of over 500 gigaflops. In addition, special-function units perform floating-point functions such as square root (SQRT), as well as transcendental functions. With 240 SPs, the GT200 exceeds 1 teraflops. Because each SP is massively threaded, it can run thousands of threads per application. A good application typically runs 5000–12,000 threads simultaneously on this chip. For those who are used to simultaneous multithreading, note that Intel CPUs support 2 or 4 threads, depending on the machine model, per core. The G80 chip supports up to 768 threads per SM, which sums up to about 12,000 threads for this chip. The more recent GT200 supports 1024 threads per SM and up to about 30,000 threads

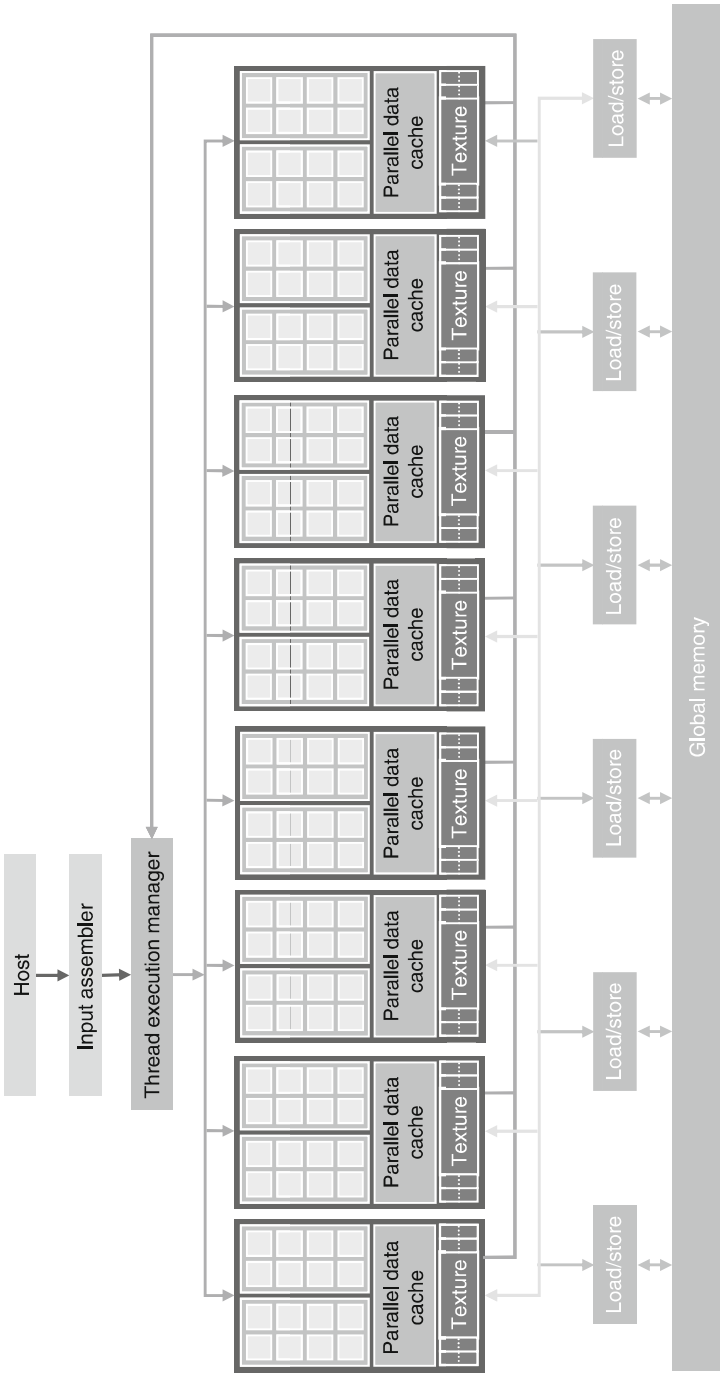


FIGURE 1.3
Architecture of a CUDA-capable GPU.

for the chip. Thus, the level of parallelism supported by GPU hardware is increasing quickly. It is very important to strive for such levels of parallelism when developing GPU parallel computing applications.

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 a continued increase in speed 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. If the application includes what we call *data parallelism*, it is 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 computing applications in today's world, many exciting mass-market applications of the future will be what we currently consider to be *supercomputing applications*, or *superapplications*. 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. From the 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 with regard to science and medicine.

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition television (HDTV) versus older National Television System Committee (NTSC) television. Once we experience the level of details offered by HDTV, it is very hard to go back to older technology. But, consider all the processing that is necessary 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 that televisions have more computing power.

Among the benefits offered by greater computing speed are much better user interfaces. Consider the Apple® iPhone® interfaces; the user enjoys a much more natural interface with the touch screen compared to other cell phone devices, even though the iPhone has a limited-size window. Undoubtedly, future versions of these devices will incorporate higher definition, three-dimensional perspectives, voice and computer vision based interfaces, requiring even more computing speed.

Similar developments are underway in consumer electronic gaming. Imagine driving a car in a game today; the game is, in fact, simply a prearranged set of scenes. If your car bumps into an obstacle, the course of your vehicle does not change; only the game score changes. Your wheels are not bent or damaged, and it is 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 large amounts of computing power.

All of the new applications that we mentioned involve simulating a concurrent world in different ways and at different levels, with tremendous amounts of data being processed. 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. Techniques for doing so are well known to those who work with such applications on a regular basis. Thus, various granularities of parallelism do exist, but the programming model must not hinder parallel implementation, and the data delivery must be properly managed. CUDA includes such a programming model along with hardware support that facilitates parallel implementation. We aim to teach application developers the fundamental techniques for managing parallel execution and delivering data.

How many times speedup can be expected from parallelizing these super-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 29.7%. The speedup for the entire application will be only 1.4×. In fact, even an infinite amount of speedup in the parallel portion can only slash less 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 speedup 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 more than 99.9% of the application execution time is in parallel execution. In general, 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 you to fully understand these optimizations and become skilled in them.

Keep in mind that the level of speedup achieved over CPU execution can also reflect the suitability of the CPU to the application. In some applications, CPUs perform very well, making it more difficult 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 in such a way 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 1.4 illustrates the key parts of a typical application. Much of the code of a real application tends to be sequential. These portions are considered to be 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 difficult 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 superapplications.

Then come the meat portions of the peach. These portions are easy to parallelize, as are some early graphics applications. For example, most of today's medical imaging applications are still running on combinations of

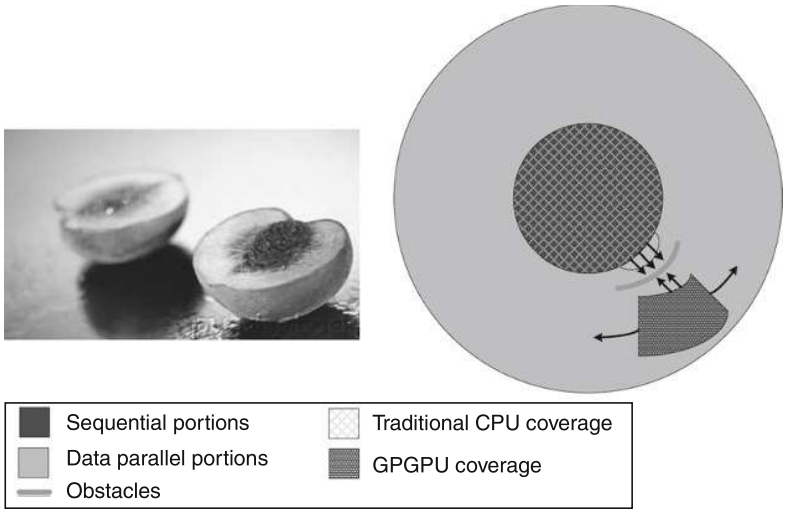


FIGURE 1.4

Coverage of sequential and parallel application portions.

microprocessor clusters and special-purpose hardware. The cost and size benefit of the GPUs can drastically improve the quality of these applications. As illustrated in Figure 1.4, early GPGPUs cover only a small portion of the meat section, which is analogous to a small portion of the most exciting applications coming in the next 10 years. As we will see, the CUDA programming model is designed to cover a much larger section of the peach meat portions of exciting applications.

1.4 PARALLEL PROGRAMMING LANGUAGES AND MODELS

Many parallel programming languages and models have been proposed in the past several decades [Mattson 2004]. The ones that are the most widely used are the Message Passing Interface (MPI) for scalable cluster computing and OpenMP™ for shared-memory multiprocessor systems. MPI is a model where computing nodes in a cluster do not share memory [MPI 2009]; all data sharing and interaction must be done through explicit message passing. MPI has been successful in the high-performance scientific computing domain. Applications written in MPI have been known to run successfully on cluster computing systems with more than 100,000 nodes. The amount of effort required to port an application into MPI,