Chapter 1

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 GFLOPS to the desktop and hundreds of GFLOPS to cluster servers. This relentless drive of performance improvement has allowed application software to perform more functionality, have better user interfaces, and generate more useful results with more. The users, in turn, demand even more improvements once they become accustomed to these improvements, creating a positive cycle for the computer industry.

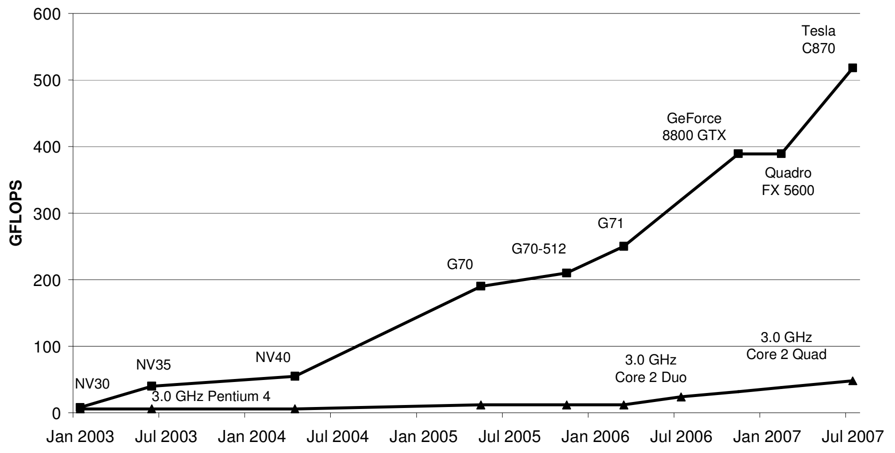
During the drive, most software developers have relied on the advances in hardware to increase the speed of their applications under the hood; the same software simply runs faster as each new generation of processors is introduced. This drive, however, has slowed since 2003 due to power consumption issues 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 multi-core and many-core 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.

Traditionally, the vast majority of software applications are written as sequential programs, as described by von Neumann in his seminal paper in 1947(?). For these sequential programs, their execution can be understood by 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 Such expectation is no longer valid from this day onward. A sequential program will only run on one of the processor cores, which will not become any 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 achieve the functionality faster. This new, dramatically escalated incentive for parallel program development has been referred to as the parallelism revolution [Larus ACM Queue article]. 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. GPUs as Parallel Computers

Since 2003, a class of many-core processors called Graphics Processing Units (GPUs), have led the race for floating-point performance. 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 2008, the ratio of peak floating-point calculation throughput between many-core GPUs and multi-core CPUs is about 10. These are not necessarily achievable speeds, merely the raw speed that the execution resources can potentially support in these chips: 367 gigaflops vs. 32 gigaflops. NVIDIA has subsequently delivered software driver and clock improvements that allow a G80 Ultra to reach 518 gigaflops, only about seven months after the original chip. In June 2008, NVIDIA introduced the GT200 chip, which delivers almost 1 teraflop (1,000 gigaflops) of single precision and almost 100 gigaflops of double precision performance.



[1]Based on slide 7 of S. Green, "GPU Physics," SIGGRAPH 2007 GPGPU Course. http://www.gpgpu.org/s2007/slides/15-GPGPU-physics.pdf

Figure 1.1. Enlarging Performance Gap between GPUs and CPUs.

Such a large performance gap between parallel and sequential processors has amounted to a significant "electrical potential build-up," and at some point, something will have to give. We may be nearing 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 GPU 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 up the work amount cooperating threads of execution.

One might ask why there is such a large performance gap between many-core GPUs and general-purpose multi-core 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 2008, the new general-purpose multi-core 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 10x the bandwidth of contemporaneously available CPU chips. In late 2006, G80 was capable of about 80 gigabytes per second (GB/S) into the main DRAM. Because of frame buffer requirements and the relaxed memory model (and without relying too heavily on architecture details of other coherence models and memory models), general-purpose processors have to satisfy requirements from legacy operating systems and applications that make memory bandwidth more difficult to increase. And with simpler memory models, the GPUs must be able to get about 80 GB/S into the memory. The most recent chip supports more than 100 GB/S. Microprocessor system memory bandwidths will probably not grow beyong 20 GB/S for about three 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 forced 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 the GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The general philosophy for GPU design is to optimize for the execution of massive number 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, minimizing the control logic required for each execution thread. 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. As a result, much more chip area is dedicated to the floating-point calculations.

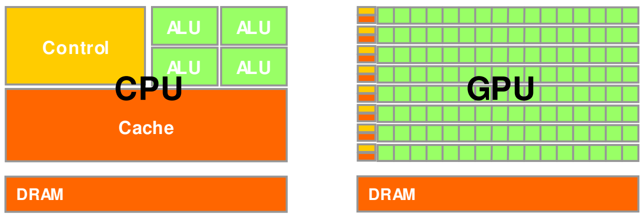


Figure 2.2. CPUs and GPUs have fundamentally different design philosophies.

It should be clear now that GPU is designed as a numeric computing engine and it will not perform well on some tasks that 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 numeric int5ensive parts on the GPUs. This is why the CUDA programming model is designed to support joint CPU-GPU execution of an application.

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 market place, 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 can be 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 processing 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 processing 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. The G80 family of CUDA-capable processors and its successors have shipped almost 100 million units to date. This is the first time that massively parallel computing is part of a mass-market product. Such a large market presence has made these GPUs economically attractive for application developers.

Another important decision factor is practical form factors and easy accessibility. Until 2006, with the advent of parallel programming and newly developed parallel software, production work usually ran on servers or datacenters on departmental clusters. But the use of these applications tends to be limited. 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 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 and racks of clusters into clinical settings, while this is common in academic departmental settings. In fact, 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 in the foreseeable future.

Another important consideration in selecting a processor for executing numeric computing applications is the support for IEEE Floating-Point Standard. The standard makes it possible to have predictable results across processors from different vendors. While the support for IEEE Floating-Point Standard was not strong in early GPUs, this has also changed for the new generations of GPUs such as the GeForce 8 series. As we will discuss in Chapter 5, GPU support for 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 values as the CPUs. A major remaining issue is that the GPUs floating-point arithmetic units are primarily single precision today. Applications that truly require double precision floating-point will not be suitable for GPU execution in the immediate future. Nevertheless, we have already seen many applications where single-precision floating is sufficient.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphic API to access the processor cores, meaning that open GL or direct 3D techniques were needed to program these chips. This technique was called GPGPU, 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. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent results.

But everything changed in 2007 with the release of CUDA. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming, so this does not represent software changes alone; additional hardware was added to the chip. Therefore, if you use the G80 and follow-up chips for GPU Computing, the programming interface will not go through the graphics interface at all. Instead, a new general-purpose interface on the silicon will enable this. Moreover, all the other software layers were redone as well, so that you can do essential reprogramming. This makes a huge difference. Some of our students tried to do their machine problems (MPs) using the old programming interface and MPIs after finishing the first few MPs, and they tremendously appreciated the difference.

1.2. Architecture of a modern GPU

Figure 1.3 shows the architecture of a typical GPU today? It is organized into 16 highly threaded Streaming Multiprocessors (SMs). A pair of SMs form a building block in Figure 1.3. Each SM has 8 streaming processors (SPs), for a total of 128 (16*8). Each SP has a multiply-add (MAD) unit, and and an additional multiply (MUL) unit, all running at 1.35

gigahertz (GHz). If you do the math, that's almost 367 gigaflops for the MADs and a total of over 500 gigaflops if you include the MULs as well. In addition, special function units perform floating point functions such as SQRT and RCP SQRT as well as transcendental functions. Each GPU currently comes with 1.5 megabytes of DRAM. These DRAMs differ from the system memory DIMM DRAMs on the motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold high-definition video images, and texture information for 3D rendering as in games. But for computing, they function like very high bandwidth off-chip cache, though with somewhat more latency regular cache or system memory. If the chip is programmed properly, the high bandwidth makes up for the large latency.

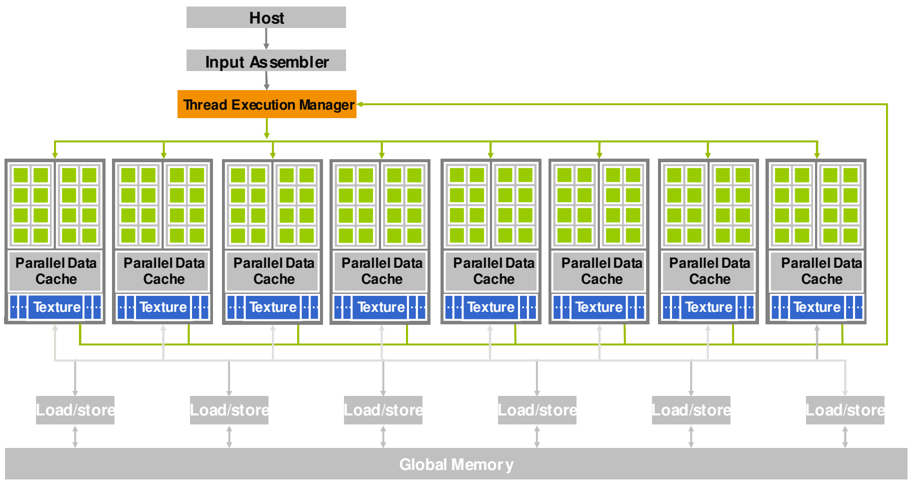


Figure 1.3. Architecture of a CUDA-capable GPU

The processor has 86.4 GB/S of memory bandwidth, plus 4 gigabytes of bandwidth each way across the PCI-express bus – a total of 8 GB/s for communication with the CPU. You can transfer data from the system memory at 4 GB/S, and you can upload data back to the system memory at 4 GB/S. Altogether, there is a combined total of 8 gigabytes/second, but for practical purposes, you'll likely work one way or the other at a time unless you are overlapping data transfers before and after a sequence of computations. This may seem like a limitation, but the PCI-E bandwidth is comparable to the system memory and CPU front-side bus bandwidth, so it's really not the limitation it would seem at first.

Some important characteristics: peak performance is about 10 times better than the current highest end microprocessors. One of my students, John Stone, forwarded an ion placement application, a forced calculation application, onto this machine, and got 265 gigaflops sustained performance for his application. This is about 100 times the speed that he had achieved on the CPU before. His impressive results are part of the benchmark suite developed in the courses.

The G80 chip is massively parallel, with 128 processor cores. Because it is massively threaded, it sustains thousands of threads per application. A good application will run 5,000 to 12,000 threads simultaneously on this chip. For those who are used to simultaneous multithreading, note that Intel supports 2 or 4 threads, depending on the machine model, per core. The G80 chip supports up to 768 threads per core, and 128 altogether, which adds up to about 12,000 threads from this chip. It is very important to understand this particular view so that you can write effective programs.

1.3. Why more speed or parallelism?

One might ask why applications will continue to demand increased speed of computing systems. Many applications that we have today seem to be running quite fast enough. As we will discuss in case studies, when an application is suitable for GPU execution, a good implementation on a GPU can achieve more than 100 times (100x) of speedup over a CPU. If the application includes what we call "data parallelism," it's a simple task to achieve a 10x speedup with just a few hours of work. For anything beyond that, we invite you to keep reading!

The answer to why more speed is in new, innovative applications. Despite the myriad of computing applications in today's world, the exciting applications of the future will be what we currently consider "supercomputing applications." For example, the biology world 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 what we can do with these instruments. They can be greatly improved by using a model to simulate the underlying system with boundary conditions set to enable the simulation. From the simulation we can measure even more details, more principles, and more hypothesis verification than can ever be imagined with direct instrumentation alone. These simulations will continue to benefit from the increasing computing power in the foreseeable future in terms of the size of the biological system and the amount of reaction time that can be simulated within tolerable response time. The enhancements will have tremendous implications to science and medicine.

For applications such as video and audio coding and manipulation, try to compare your satisfaction with digital high-definition (HD) TV vs. older technology. Once you 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 HD TV. It is a very parallel process, as are 3D imaging and visualization. Massively parallel processors will continue to enhance the size and fidelity of the pictures of HDTVs in the coming years.

Among the benefits offered by more computing speed are much better user interfaces. Consider Apple's I-Phone interfaces, compared to other cell phones, even though the I-

Phone still has a limited window. Doubtlessly, future versions of these devices will incorporate higher definition and three-dimensional perspectives, requiring even more computing speed.

We are just at the beginning of these developments, consistent with the new but increasing demands of consumer gaming physics. Imagine driving a car in a game today: the game is in fact simply a prearranged set of scenes. If you bump into an obstacle, the course of your driving does not change; only the game score changes. Your wheels do are bent or damaged, and it's 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 pre-arranged scenes. You can expect to see more of these realistic effects in the future: accidents will damage your wheels and your online driving experience will be affected.

All the new applications that we mentioned actually simulate 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. But techniques for doing that are well known to those who work with such applications regularly. Thus, various granularities of parallelism do exist, but the programming model must not hinder parallel implementation, and the data delivery must be properly managed.

How many times speedup can be expected from this type of 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 100X speedup of the parallel portion will reduce the execution time by 29.7%. The speed up for the entire application will be only 1.4X. In fact, even infinite amount of speedup in the parallel portion can only slash less 30% off execution time. On the other hand, if 99% of the execution time is in the parallel portion, a 100X speedup will reduce the application execution to 1.99% of the original time. This gives the entire application a 50X speedup. Conversely, it is very important that an application had the vast majority of its execution in parallel portion for a massively parallel processor to effectively speedup its execution.

Researchers at Illinois have achieved speedups of more than 100x for some applications. However, this is typically achieved only after extensive optimization and tuning even after the algorithms have been enhanced so that more than 99.9% of the application execution time is in parallel execution. In general, applications often saturate the memory (DRAM) bandwidth, resulting in about 10X speedup rather. The trick is to figure out how to get around memory bandwidth limitations, which involved 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, optimize the code to get around limitations such as limited on-chip memory capacity. Those developers who successfully got around these limitations got 25x - 400x speedups. Our goal is to help you to achieve the same.

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 harder to speed up performance using 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. This is precisely what the CUDA programming model promotes, as we will further explain in the book.

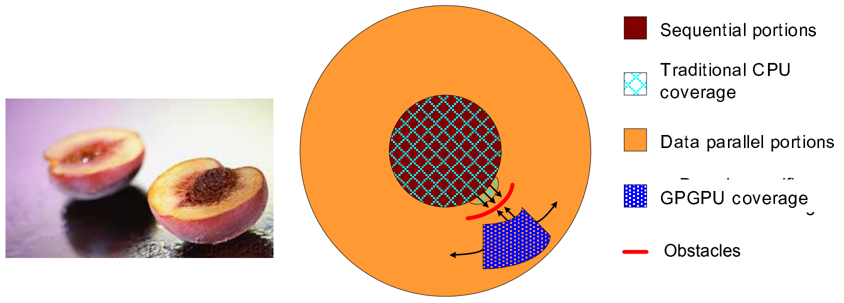


Figure 1.4. Coverage of sequential and parallel application portions

Figure 1.4 illustrates the key parts of a typical application. Much of a real application's code tends to sequential. These portions 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 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. For example, most of today's medical imaging applications are still running on combinations of microprocessor clusters and special-purpose hardware. The cost and size benefit of the GPUs can drastically improve the quality of these applications. Early GPGPUs cover a small variety of such meat portion, which accounts for only a small portion of the most exciting applications coming in the next ten years. As we will see, the CUDA programming model is designed to cover a much larger variety of the peach meat portions of exciting application.

1.3. Overarching Goals

Our primary goal is to teach you, the reader, 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 develop a parallel program in an hour. But we're going to dedicate many pages to showing you how to do *high-performance* parallel programming. And, we believe that it is still very easy if you have 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 parallel computing.

Note that hardware architecture features have constraints. For high-performance parallel programming on most of the chips that will come out in the next five to ten years, you will need some knowledge of how the architecture actually works. It will probably take ten more years before we can build tools and machines so that most programmers can work without this knowledge. But for our purposes, that won't be necessary. We will aim to complete a suite of API programming tools and techniques at least once, so that you will be able to apply the experience to other APIs and other tools in the future.

Our second goal is teach parallel programming for correct functionality and dependability, 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 later debug the code, reproduce the bugs when they reappear, and support the code. We will show that with the CUDA programming model that focus 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 ways to design architecture and do parallel programming so that future machines, which will be more and more parallel, can take advantage of your code. We want to help you to master parallel programming so that that you can achieve high performance regardless of the particular hardware you're working on. We want you to be able to write code that will be able to scale up to the level of performance of new generations of machines.

Much technical knowledge will be required to achieve these goals, so we will cover the principles and patterns of parallel programming in this book. We cannot guarantee that we will cover all of them, however, so we have selected several of 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.

1.4. Organization of the Book

Chapter 2 introduces CUDA programming. This chapter relies on the fact that students have had previous experience with C programming. It first introduces CUDA as a simple, small extension to C and an instance of widely used Single Program Multiple Data (SPMD) parallel programming models. 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 individual threads in the parallelized part, (5) launching the execution of kernel function by parallel threads, and (6) eventually transferring the data back to the host processor with an API function call.

Chapter 2 also covers the use of the CUDA software development kit (SDK) to compile, link, run, and debug the programs. While the objective of Chapter 2 is to prepare the students for writing and debugging a simple parallel CUDA 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 matrix multiplication to make this lecture concrete.

Chapters 3 through 6 give students more in-depth understanding of the CUDA programming model. Chapter 3 covers the thread organization and execution model required for students to fully understand the execution behavior of threads and prepare them for the performance concepts. Chapter 4 is dedicated to the special memories that can be used to hold CUDA variables for improved program execution speed. Chapter 5 introduces the major factors that contribute to the performance of a CUDA kernel function. Chapter 6 introduces the basic concept of floating-point representation and computation so that students can understand concepts such as precision and accuracy.

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

Chapter 7 introduces the fundamentals of parallel programming and computational thinking. It does so by covering the concept of organizing the computation tasks of a program so that one can more easily identify high-level parallel execution opportunities. We start by discussing the translational process of organizing abstract scientific concepts into computational tasks, which is a very important first step in producing quality scientific application software, serial or parallel. It is also an important step towards understanding computational experimentation; students develop better intuition with the capabilities and limitations of computational models.

Computational *thinking* is arguably the most important skill for computational scientists. Like any other thought process or problem-solving skill, computational thinking is an art that varies across scientific disciplines. We believe that the skill can be best taught through a combination of basic principles, systematic case studies, and hands-on exercise. In terms of basic principles, we discuss the major steps in decomposing a large computational problem into smaller, coordinated tasks that can each be realized with numerical methods and well-known algorithms. When writing parallel programs, it is particularly important for the programmer to analyze and understand the structure of the domain problem: which parts are inherently serial tasks, and which parts are amenable to parallel execution. As part of computational thinking, we hope to teach the students to analyze the problem structure and plan the strategy for organizing their serial host and parallel device computation. This is not just a necessary step before parallel programming, and it is also a valuable skill that will serve them well throughout their computing careers.

Chapter 8 discusses parallel algorithm structures and their effects on application performance. It is grounded in students' performance tuning experience with CUDA. Starting with concrete needs and observations, students are motivated to understand more deeply the connection between algorithmic steps and execution performance. Through the course of the lecture, students develop the ability to reason about the performance effects of their own algorithms and implementation decisions.

Chapter 9 covers parallel programming styles, enabling students to place their knowledge in a wider context. With this lecture, students begin to broaden their knowledge from the SPMD programming style to other styles of parallel programming, such as loop parallelism in OpenMP and fork-join in p-thread programming [6]. The main objective of the lecture is to make an initial connection, so students can understand how these models relate to each other. Although we do not go deeply into these alternative parallel programming styles, we expect that the students will be able to learn to program in any of them with the foundation they gain in this course.

Chapters 9 and 10 are case studies of two real applications [7, 8, 9], which take students 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 reason about the advantages and disadvantages of each alternative. We then go through the steps of code transformation needed to achieve high performance. These two lectures help students put all the materials from the previous lectures together and prepare for their own application development projects.