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 giga floating-point operations per second (GFLOPS, or Giga (10^9) Floating-Point Operations per Second), to the desktop and tera floating-point operations per second (TFLOPS, or Tera (10^{12}) Floating-Point Operations per Second) to datacenters. 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 [Sutter 2005].

Traditionally, the vast majority of software applications are written as sequential programs that are executed by processors whose design was envisioned by von Neumann in his seminal report in 1945 [vonNeumann 1945]. The execution of these

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00001-7

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 processor generation 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 Such expectation is no longer valid from this day onward.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 from generation to generation. 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 significant 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 typically ran on large scale, expensive computers. Only a few elite applications could 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 [Hwu 2008]. 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 a recent *Intel* multicore microprocessor with up to 12 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. For more discussion of CPUs, see https://en.wikipedia.org/wiki/Central_processing_unit.

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 Tesla P100 graphics processing unit (GPU) with 10s of 1000s of threads, executing in a large number of simple, in order pipelines. Many-thread processors, especially the GPUs, have led the race of floating-point performance

since 2003. As of 2016, the ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs is about 10, and this ratio has been roughly constant for the past several years. These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips. For more discussion of GPUs, see https://en.wikipedia.org/wiki/Graphics_processing_unit.

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. 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 the work among cooperating parallel workers.

One might ask why there is such a large peak throughput gap between many-threaded 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 Fig. 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 throughput. As of 2016, the high-end general-purpose multicore microprocessors typically have eight or more large processor cores and many 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 memory system into the processors. Graphics chips have been operating at approximately 10x the memory bandwidth of contemporaneously available CPU chips. A GPU must be capable of moving extremely large amounts of data in and out of its main Dynamic Random

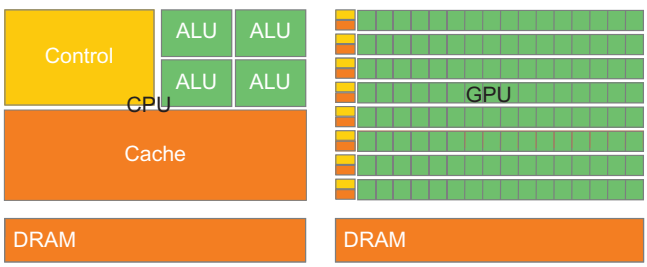


FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies.

Access Memory (DRAM) because of graphics frame buffer requirements. 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, we expect that CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

The design philosophy of the GPUs has been 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. An important observation is that reducing latency is much more expensive than increasing throughput in terms of power and chip area. Therefore, 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 for these GPUs 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 as 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 (Appendix A), OpenACC (see chapter: Parallel programming with OpenACC), and C++AMP (Appendix D).

It is also important to note that speed 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 *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-thread 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 nearly 1 billion CUDA enabled GPUs in use to date. 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. However, real-world clinical applications on MRI machines utilize 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 computer server boxes 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. Today, many companies ship 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 IEEE Floating-Point Standard. The standard enables 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 2006. As we will discuss in Chapter 6, Numerical considerations, GPU support for the IEEE Floating-Point Standard has become comparable with that of the CPUs. As a result, one can expect

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

that more numerical applications will be ported to GPUs and yield comparable result values as the CPUs. Up to 2009, a major barrier was that the GPU floating-point arithmetic units were 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 only high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications. In addition, GPUs support Fused Multiply-Add, which reduces errors due to multiple rounding operations.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics application programming interface (API) functions to access the processing units, 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 in order to execute on these early GPUs. This technique was called GPGPU, for General-Purpose Programming using a GPU. 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 GPUs. 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 [NVIDIA 2007]. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming, so this did not represent software changes 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. 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 general-purpose computing applications.

1.2 ARCHITECTURE OF A MODERN GPU

Fig. 1.2 shows a high level view of the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Fig. 1.2, two SMs form a building block. However, the number of SMs in a building block can vary from one generation to another. Also, in Fig. 1.2, each SM has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with gigabytes of Graphics Double Data Rate (GDDR), Synchronous DRAM (SDRAM), referred to as Global Memory in Fig. 1.2. These

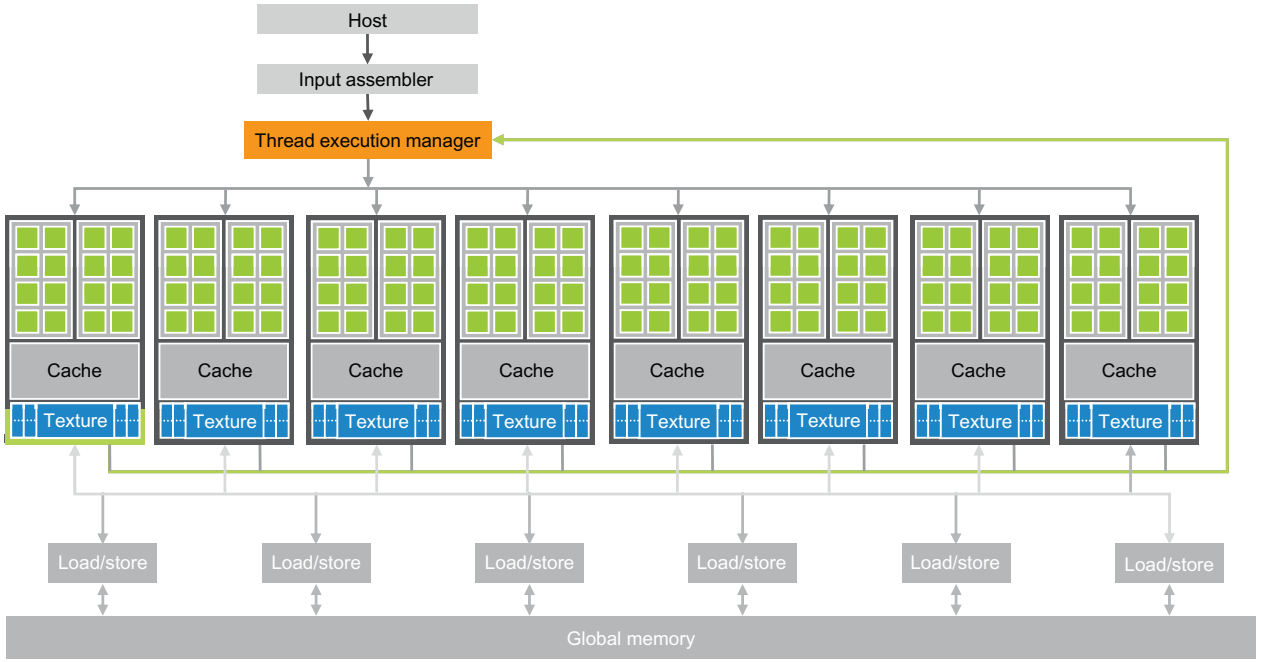


FIGURE 1.2

Architecture of a CUDA-capable GPU.

GDDR SDRAMs 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. 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. More recent products, such as NVIDIA's Pascal architecture, may use High-Bandwidth Memory (HBM) or HBM2 architecture. For brevity, we will simply refer to all of these types of memory as DRAM for the rest of the book.

The G80 introduced the CUDA architecture and had 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 or Gen4, which supports 8–16 GB/s in each direction. The Pascal family of GPUs also supports NVLINK, a CPU–GPU and GPU–GPU interconnect that allows transfers of up to 40 GB/s per channel. As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E or NVLINK 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.

A good application typically runs 5000 to 12,000 threads simultaneously on this chip. For those who are used to multithreading in CPUs, note that Intel CPUs support 2 or 4 threads, depending on the machine model, per core. CPUs, however, are increasingly using Single Instruction Multiple Data (SIMD) 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.

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 question if applications will continue to demand increased speed. Many applications that we have today seem to be running fast enough. As we will discuss in the case study chapters (see chapters: Application case study—non-Cartesian MRI, Application case study—molecular visualization and analysis, and Application case study—machine learning), when an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100 times (100x) speedup over sequential execution on a single CPU core. If the application contains what we call "data parallelism," it is often possible to achieve a 10x 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 previously 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 for science and medicine.

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition (HD) TV vs. 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 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.

User interfaces can also be improved by improved computing speeds. Modern smart phone users enjoy a more natural interface with high-resolution touch screens that rival that of large-screen televisions. Undoubtedly future versions of these devices will incorporate sensors and displays with three-dimensional perspectives, applications that combine virtual and physical space information for enhanced usability, and voice 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 the player's car collided with obstacles, the behavior of the car did not change to reflect the damage. Only the game score changes—and the score determines the winner. The car would drive the same—despite the fact that the wheels should be bent or damaged. With increased computing speed, the races can actually proceed according to simulation instead of approximate scores and scripted sequences. We can expect to see more of these realistic effects in the future: collisions will damage your wheels and the player's 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 mentioned 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 phrase. 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 the reader to 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

What kind of speedup can we expect 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 100X 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 1.4X. In fact, even infinite amount of speedup in the parallel portion can only slash 30% off execution time, achieving no more than 1.43X speedup. The fact that the level of speedup one can achieve through parallel execution can be severely limited by the parallelizable portion of the application is referred to as Amdahl's Law. On the other hand, if 99% of the execution time is in the parallel portion, a 100X speedup of the parallel portion will reduce the application execution to 1.99% of the original time. This gives the entire application a 50X 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 100X 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 10X 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 the reader to 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.

Fig. 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 are pretty good with 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.

The rest is 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 speed of these applications. As illustrated in Fig. 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 model is designed to cover a much larger section of the peach meat portions of exciting applications. In fact, as we will discuss in Chapter 20, More on CUDA and GPU computing, these programming models and their underlying hardware are still evolving at a fast pace in order to enable efficient parallelization of even larger sections of applications.

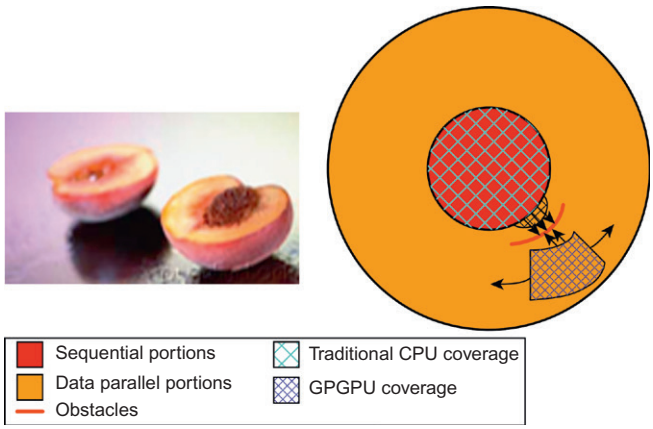


FIGURE 1.3

Coverage of sequential and parallel application portions.

1.5 CHALLENGES IN PARALLEL PROGRAMMING

What makes parallel programming hard? 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 then why bother to write a parallel program if you do not care about performance?

This book addresses several challenges in achieving high-performance in parallel programming. First and foremost, it can be challenging to design parallel algorithms with the same level of algorithmic (computational) complexity as sequential algorithms. Some parallel algorithms can add large overheads over their sequential counter parts so much that they can even end up running slower for larger input data sets.

Second, the execution speed of many applications is limited by memory access speed. We refer to these applications as memory-bound, as opposed to compute bound, which are limited by the number of instructions performed per byte of data. Achieving high-performance parallel execution in memory-bound applications often requires novel methods for improving memory access speed.

Third, the execution speed of parallel programs is often more sensitive to the input data characteristics than their sequential counter parts. Many real world applications need to deal with inputs with widely varying characteristics, such as erratic or unpredictable data rates, and very high data rates. The performance of parallel programs can sometimes vary dramatically with these characteristics.

Fourth, many real world problems are most naturally described with mathematical recurrences. Parallelizing these problems often requires nonintuitive ways of thinking about the problem and may require redundant work during execution.

Fortunately, most of these challenges have been addressed by researchers in the past. There are also common patterns across application domains that allow us to apply solutions derived from one domain to others. This is the primary reason why we will be presenting key techniques for addressing these challenges in the context of important parallel computation patterns.

1.6 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 message passing interface (MPI) [MPI 2009] for scalable cluster computing, and OpenMP [Open 2005] 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: Parallel programming with OpenACC)

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. We can refer to this property as "performance portability." This is why we teach OpenACC programming in Chapter 19, Parallel programming with OpenACC. 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 CUDA style interfaces for parts where OpenACC compilers fall short.

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 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. It is therefore important that a parallel programmer in HPC understands how to do joint MPI/CUDA programming, which is presented in Chapter 18, Programming a Heterogeneous Computing Cluster.

The amount of effort needed to port an application into MPI, however, can be quite high due to 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 CPU and 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 [GCN 2010], are now available. With such support, a CUDA programmer can declare variables and data structures as shared between CPU and GPU. The runtime hardware and software transparently maintains coherence by automatically performing optimized data transfer operations on behalf of the programmer as needed. Such support significantly reduces the programming complexity involved in overlapping data transfer with computation and I/O activities. As will be discussed later in Chapter 20, More on CUDA and GPU Computing, the Pascal architecture supports both a unified global address space and memory.

In 2009, several major industry players, including Apple, Intel, AMD/ATI, NVIDIA jointly developed a standardized programming model called Open Computing Language (OpenCL) [Khronos 2009]. 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. 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 in order 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 Appendix A and explain how one can apply the key concepts in this book to OpenCL programming.

1.7 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. Therefore, we are 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* [Wing 2006] techniques that will enable you to think about problems in ways that are amenable to high-performance parallel computing.

Note that hardware architecture features still have constraints and limitations. High-performance parallel programming on most processors will require some knowledge of how the hardware works. It will probably take ten 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 of our discussions on high-performance parallel programming techniques.

Our second goal is to teach parallel programming for correct functionality and reliability, which constitutes 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 simple forms of barrier synchronization, memory consistency, and atomicity for managing parallelism. In addition, it provides an array of powerful tools that allow one to debug not only the functional aspects but also the performance bottlenecks. We will show that by focusing on data parallelism, one can achieve high performance without sacrificing the reliability of 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 to 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.

Still, much technical knowledge will be required to achieve these goals, so we will cover quite a few principles and patterns [Mattson 2004] of parallel programming in this book. We will not be teaching these principles and patterns in a vacuum. We will teach them in the context of parallelizing useful applications. We cannot cover all of them, however, we have selected what we found to be 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.

1.8 ORGANIZATION OF THE BOOK

Chapter 2, Data parallel computing, introduces data parallelism and CUDA C programming. This chapter expects the reader to 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 single program multiple data (SPMD) 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 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 2, Data parallel computing, 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 illustrate these concepts. In the later part of the book, we also compare CUDA with other parallel programming models including OpenMP, OpenACC, and OpenCL.

Chapter 3, Scalable parallel execution, 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 the reader to implement sophisticated computation using CUDA C and reason about the performance behavior of their CUDA code.

Chapter 4, Memory and data locality, is dedicated to the special memories that can be used to hold CUDA variables for managing data delivery and improving program execution speed. We introduce the CUDA language features that allocate and use these memories. Appropriate use of these memories can drastically improve the data access throughput and help to alleviate the traffic congestion in the memory system.

Chapter 5, Performance considerations, presents several important performance considerations in current CUDA hardware. In particular, it gives more details in desirable patterns of 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 6, Numerical considerations, introduces the concepts of IEEE-754 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 7, Parallel patterns: convolution, Chapter 8, Parallel patterns: prefix sum, Chapter 9, Parallel patterns—parallel histogram computation, Chapter 10, Parallel patterns: sparse matrix computation, Chapter 11, Parallel patterns: merge sort, Chapter 12, Parallel patterns: graph search, present six important parallel computation patterns that give the readers more insight into parallel programming techniques and parallel execution mechanisms. Chapter 7, Parallel patterns: convolution, presents convolution and stencil, frequently used parallel computing patterns that require careful management of data access locality. We also use this pattern to introduce constant memory and caching in modern GPUs. Chapter 8, Parallel patterns: prefix sum, presents reduction tree and prefix sum, or scan, an important parallel computing pattern that converts sequential computation into parallel computation. We also use this pattern to introduce the concept of work-efficiency in parallel algorithms. Chapter 9, Parallel patterns—parallel histogram computation, covers histogram, a pattern widely used in pattern recognition in large data sets. We also cover merge operation, a widely used pattern in divide-and-concur work partitioning strategies. Chapter 10, Parallel patterns: sparse matrix computation, presents sparse matrix computation, a pattern used for processing very large data sets. This chapter introduces the reader to the concepts of rearranging data for more efficient parallel access: data compression, padding, sorting, transposition, and regularization. Chapter 11, Parallel patterns: merge sort, introduces merge sort, and dynamic input data identification and organization. Chapter 12, Parallel patterns: graph search, introduces graph algorithms and how graph search can be efficiently implemented in GPU programming.

While these chapters are based on CUDA, they help the readers build-up the foundation for parallel programming in general. We believe that humans understand best when they learn from concrete examples. That is, we must first learn the concepts in the context of a particular programming model, which provides us with solid footing to allow applying 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.

Chapter 13, CUDA dynamic parallelism, covers dynamic parallelism. This is the ability of the GPU to dynamically create work for itself based on the data or program structure, rather than waiting for the CPU to launch kernels exclusively.

Chapters 14, Application case study—non-Cartesian MRI, Chapter 15, Application case study—molecular visualization and analysis, Chapter 16, Application case study—machine learning, are case studies of three real applications, which take the 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 three chapters help the readers put all the materials from the previous chapters together and prepare for their own application development projects. Chapter 14, Application case study—non-Cartesian MRI, covers non-Cartesian MRI reconstruction, and how the irregular data affects the program. Chapter 15, Application case study—molecular visualization and analysis, covers molecular visualization and analysis. Chapter 16, Application case study—machine learning, covers Deep Learning, which is becoming an extremely important area for GPU computing. We provide an introduction, and leave more in-depth discussion to other sources.

Chapter 17, Parallel programming and computational thinking, introduces 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. Although we do not go into these alternative parallel programming styles, we expect that the readers will be able to learn to program in any of them with the foundation they gain in this book. We also present a high level case study to show the opportunities that can be seen through creative computational thinking.

Chapter 18, Programming a heterogeneous computing cluster, covers CUDA programming on heterogeneous clusters where each compute node consists of both CPU and GPU. We discuss the use of MPI alongside CUDA to integrate both inter-node computing and intra-node computing, and the resulting communication issues and practices.

Chapter 19, Parallel programming with OpenACC, covers Parallel Programming with OpenACC. OpenACC is a directive-based high level programming approach

which allows the programmer to identify and specify areas of code that can be subsequently parallelized by the compiler and/or other tools. OpenACC is an easy way for a parallel programmer to get started.

Chapter 20, More on CUDA and GPU computing and Chapter 21, Conclusion and outlook, offer 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. (2010). An asynchronous distributed shared memory model for heterogeneous parallel systems. *International conference on architectural support for programming languages and operating systems*.

Hwu, W. W., Keutzer, K., & Mattson, T. (2008). The concurrency challenge. *IEEE Design and Test of Computers*, *25*, 312–320.

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns of parallel programming*. Boston, MA: Addison-Wesley Professional.

Message Passing Interface Forum. MPI – A Message Passing Interface Standard Version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, September 4, 2009.

NVIDIA Corporation. CUDA Programming Guide. February 2007.

OpenMP Architecture Review Board, "OpenMP application program interface," May 2005.

Sutter, H., & Larus, J. (September 2005). Software and the concurrency revolution. *ACM Queue*, *3*(7), 54–62.

The Khronos Group. The OpenCL Specification version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

von Neumann, J. (1972). 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. ISBN 0-691-02367-0.

Wing, J. (March 2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.

GPU Teaching Kit

Accelerated Computing

Lecture 1.1 – Course Introduction

Course Introduction and Overview

Course Goals

- Learn how to program heterogeneous parallel computing systems and achieve
  - High performance and energy-efficiency
  - Functionality and maintainability
  - Scalability across future generations
  - Portability across vendor devices
- Technical subjects
  - Parallel programming API, tools and techniques
  - Principles and patterns of parallel algorithms
  - Processor architecture features and constraints

People

- Wen-mei Hwu (University of Illinois)
- David Kirk (NVIDIA)
- Joe Bungo (NVIDIA)
- Mark Ebersole (NVIDIA)
- Abdul Dakkak (University of Illinois)
- Izzat El Hajj (University of Illinois)
- Andy Schuh (University of Illinois)
- John Stratton (Colgate College)
- Isaac Gelado (NVIDIA)
- John Stone (University of Illinois)
- Javier Cabezas (NVIDIA)
- Michael Garland (NVIDIA)

Course Content

Module 1 Course Introduction	• Course Introduction and Overview • Introduction to Heterogeneous Parallel Computing • Portability and Scalability in Heterogeneous Parallel Computing
Module 2 Introduction to CUDA C	• CUDA C vs. CUDA Libs vs. OpenACC • Memory Allocation and Data Movement API Functions • Data Parallelism and Threads • Introduction to CUDA Toolkit
Module 3 CUDA Parallelism Model	• Kernel-Based SPMD Parallel Programming • Multidimensional Kernel Configuration • Color-to-Greyscale Image Processing Example • Blur Image Processing Example
Module 4 Memory Model and Locality	• CUDA Memories • Tiled Matrix Multiplication • Tiled Matrix Multiplication Kernel • Handling Boundary Conditions in Tiling • Tiled Kernel for Arbitrary Matrix Dimensions
Module 5 Kernel-based Parallel Programming	• Histogram (Sort) Example • Basic Matrix-Matrix Multiplication Example • Thread Scheduling • Control Divergence

Course Content

Module 6 Performance Considerations: Memory	• DRAM Bandwidth • Memory Coalescing in CUDA
Module 7 Atomic Operations	• Atomic Operations
Module 8 Parallel Computation Patterns (Part 1)	• Convolution • Tiled Convolution • 2D Tiled Convolution Kernel
Module 9 Parallel Computation Patterns (Part 2)	• Tiled Convolution Analysis • Data Reuse in Tiled Convolution
Module 10 Performance Considerations: Parallel Computation Patterns	• Reduction • Basic Reduction Kernel • Improved Reduction Kernel
Module 11 Parallel Computation Patterns (Part 3)	• Scan (Parallel Prefix Sum) • Work-Inefficient Parallel Scan Kernel • Work-Efficient Parallel Scan Kernel • More on Parallel Scan

Course Content

Module 12 Performance Considerations: Scan Applications	• Scan Applications: Per-thread Output Variable Allocation • Scan Applications: Radix Sort • Performance Considerations (Histogram (Atomics) Example) • Performance Considerations (Histogram (Scan) Example)
Module 13 Advanced CUDA Memory Model	• Advanced CUDA Memory Model • Constant Memory • Texture Memory
Module 14 Floating Point Considerations	• Floating Point Precision Considerations • Numerical Stability
Module 15 GPU as part of the PC Architecture	• GPU as part of the PC Architecture
Module 16 Efficient Host-Device Data Transfer	• Data Movement API vs. Unified Memory • Pinned Host Memory • Task Parallelism/CUDA Streams • Overlapping Transfer with Computation
Module 17 Application Case Study: Advanced MRI Reconstruction	• Advanced MRI Reconstruction
Module 18 Application Case Study: Electrostatic Potential Calculation	• Electrostatic Potential Calculation (Part 1) • Electrostatic Potential Calculation (part 2)

Course Content

Module 19 Computational Thinking For Parallel Programming	• Computational Thinking for Parallel Programming
Module 20 Related Programming Models: MPI	• Joint MPI-CUDA Programming • Joint MPI-CUDA Programming (Vector Addition - Main Function) • Joint MPI-CUDA Programming (Message Passing and Barrier) (Data Server and Compute Processes) • Joint MPI-CUDA Programming (Adding CUDA) • Joint MPI-CUDA Programming (Halo Data Exchange)
Module 21 CUDA Python Using Numba	• CUDA Python using Numba
Module 22 Related Programming Models: OpenCL	• OpenCL Data Parallelism Model • OpenCL Device Architecture • OpenCL Host Code (Part 1) • OpenCL Host Code (Part 2)
Module 23 Related Programming Models: OpenACC	• Introduction to OpenACC • OpenACC Subtleties
Module 24 Related Programming Models: OpenGL	• OpenGL and CUDA Interoperability

Course Content

Module 25 Dynamic Parallelism	• Effective use of Dynamic Parallelism • Advanced Architectural Features: Hyper-Q
Module 26 Multi-GPU	• Multi-GPU
Module 27 Using CUDA Libraries	• Example Applications Using Libraries: CUBLAS • Example Applications Using Libraries: CUFFT • Example Applications Using Libraries: CUSOLVER
Module 28 Advanced Thrust	• Advanced Thrust
Module 29 Other GPU Development Platforms: QwickLABS	• Other GPU Development Platforms: QwickLABS
Where to Find Support	

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Lecture 1.2 – Course Introduction

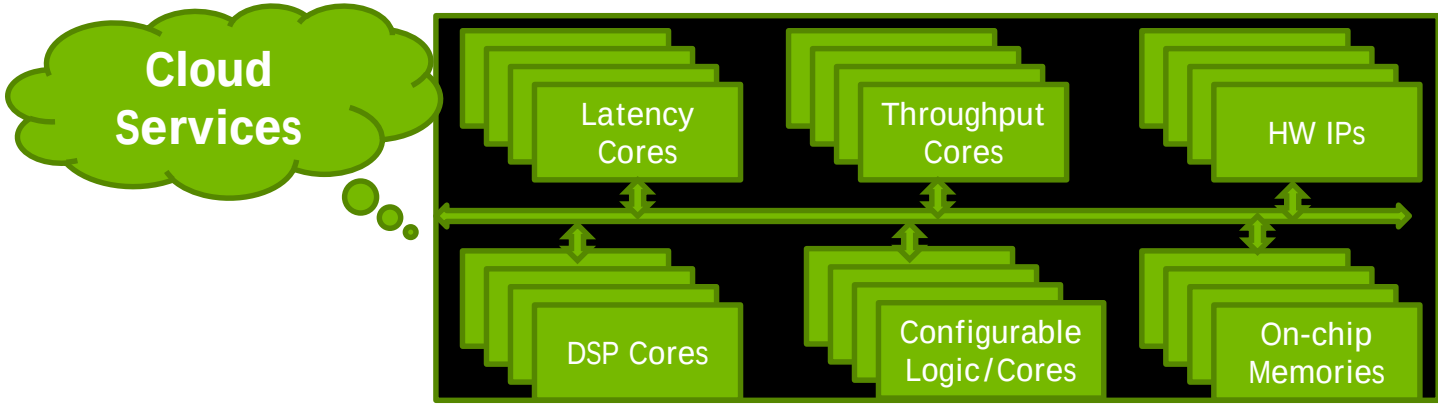
Introduction to Heterogeneous Parallel Computing

Objectives

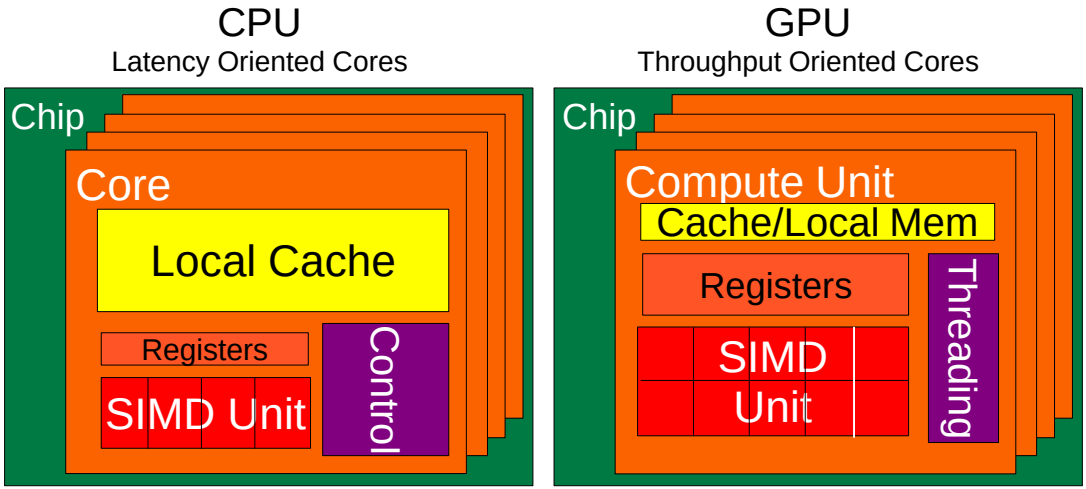
- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Heterogeneous Parallel Computing

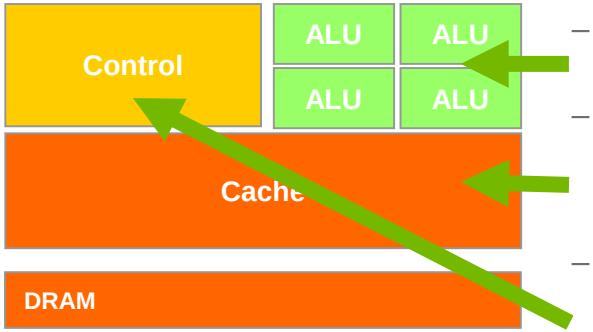
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

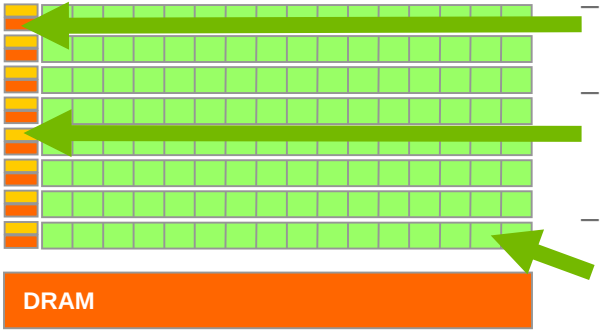


CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design

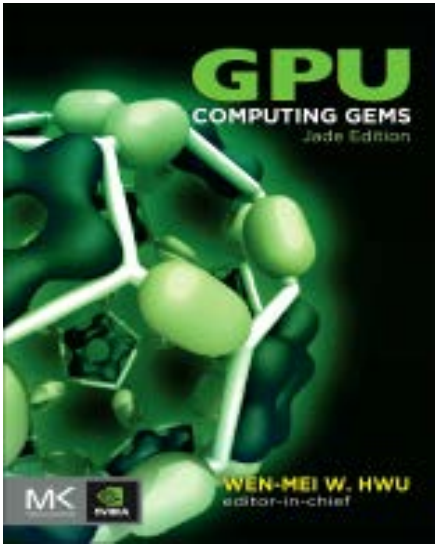


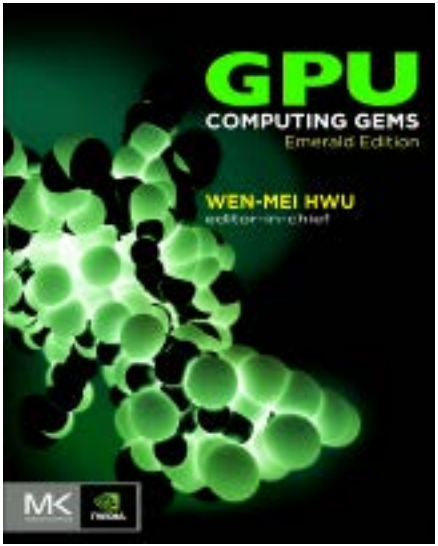
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

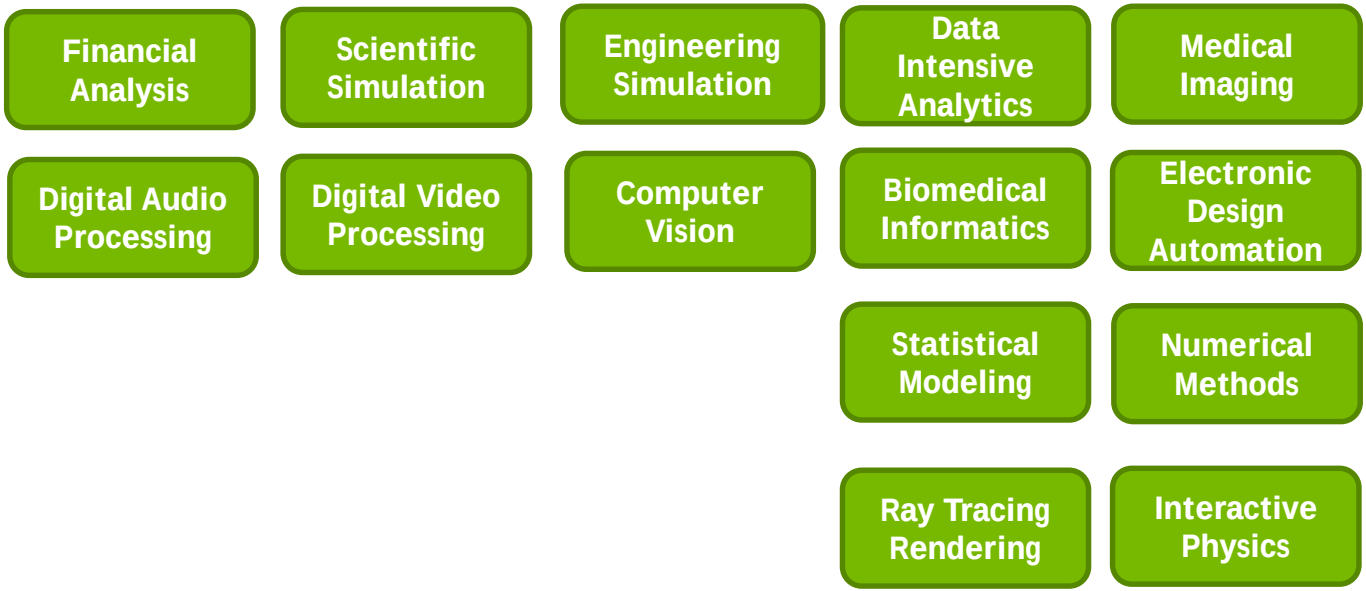
GPU computing reading resources





90 articles in two volumes

Heterogeneous Parallel Computing in Many Disciplines



GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Lecture 1.3 – Course Introduction

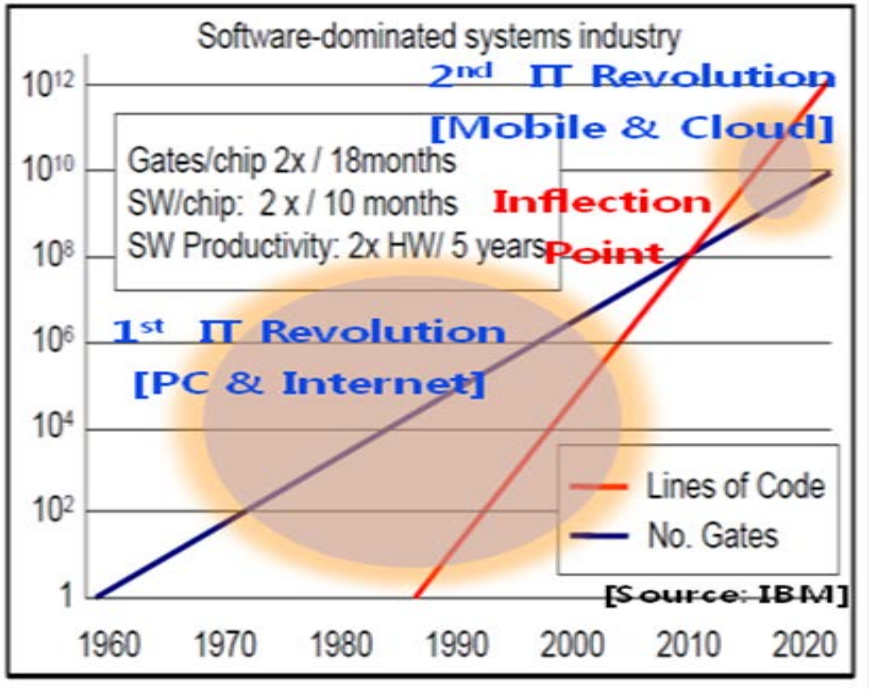
Portability and Scalability in Heterogeneous Parallel Computing

Objectives

– To understand the importance and nature of scalability and portability in parallel programming

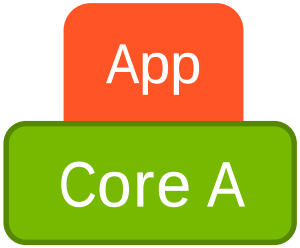
Software Dominates System Cost

- SW lines per chip increases at 2x/10 months
- HW gates per chip increases at 2x/18 months
- Future systems must minimize software redevelopment



Keys to Software Cost Control

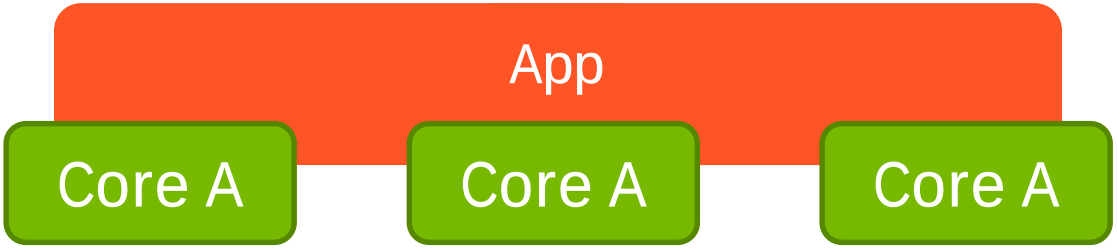
- Scalability



Keys to Software Cost Control

- Scalability
  - **The same application runs efficiently on new generations of cores**

Keys to Software Cost Control

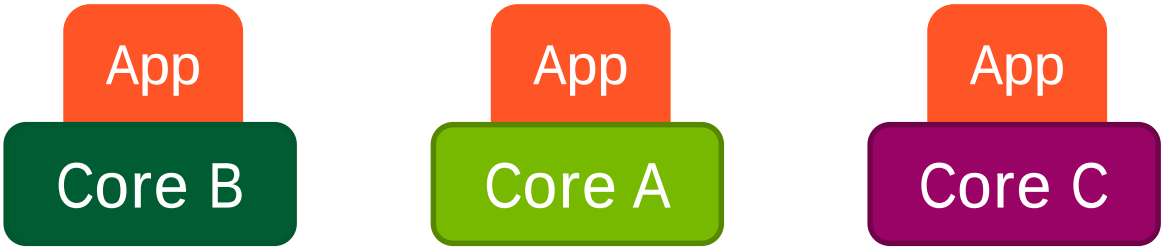


- Scalability
  - The same application runs efficiently on new generations of cores
  - **The same application runs efficiently on more of the same cores**

More on Scalability

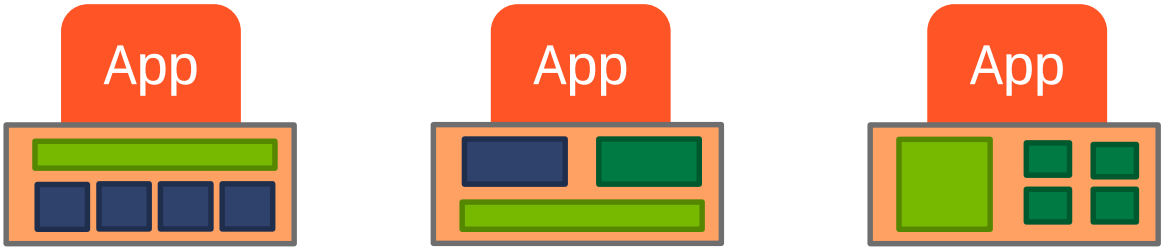
- Performance growth with HW generations
  - Increasing number of compute units (cores)
  - Increasing number of threads
  - Increasing vector length
  - Increasing pipeline depth
  - Increasing DRAM burst size
  - Increasing number of DRAM channels
  - Increasing data movement latency

Keys to Software Cost Control



- Scalability
- **Portability**
  - The same application runs efficiently on different types of cores

Keys to Software Cost Control



- Scalability
- Portability
  - The same application runs efficiently on different types of cores
  - The same application runs efficiently on systems with different organizations and interfaces

More on Portability

- Portability across many different HW types
  - Across ISAs (Instruction Set Architectures) - X86 vs. ARM, etc.
  - Latency oriented CPUs vs. throughput oriented GPUs
  - Across parallelism models - VLIW vs. SIMD vs. threading
  - Across memory models - Shared memory vs. distributed memory

GPU Teaching Kit

Accelerated Computing