CHAPTER

Introduction 1

CHAPTER CONTENTS

INTRODUCTION

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

During the drive, most software developers have relied on the advances in hardware to increase the speed of their applications under the hood; the same software simply runs faster as each new generation of processors is introduced. This drive, however, has slowed since 2003 due to energy-consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Virtually all microprocessor vendors have switched to models where multiple processing units, referred to as *processor cores*, are used in each chip to increase the

cost reduction 成本削减. vendors 供应商.
relentless drive. 不懈的追求 virtually 事实上的
accustomed to. 习惯于.
heat-dissipation issues. 散热问题.

exerted. 施加，运用，发挥.

seminal report. 开创性的报告

escalated. 升级，加剧.

incentive. 动机，激励力

trajectories 轨迹，发展路径，运行轨道.

processing power. This switch has exerted a tremendous impact on the software developer community [Sutter 2005].

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

Rather, the applications software that will continue to enjoy performance improvement with each new generation of microprocessors will be parallel programs, in which multiple threads of execution cooperate to complete the work faster. This new, dramatically escalated incentive for parallel program development has been referred to as the *concurrency revolution* [Sutter 2005]. The practice of parallel programming is by no means new. The high-performance computing community has been developing parallel programs for decades. These programs run on large-scale, expensive computers. Only a few elite applications can justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers. Now that all new microprocessors are parallel computers, the number of applications that must be developed as parallel programs has increased dramatically. There is now a great need for software developers to learn about parallel programming, which is the focus of this book.

1.1 GPUs AS PARALLEL COMPUTERS

CPU 层面

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

hyperthreading 超线程

out-of-order processor 乱序处理器

in-order, single-instruction processor

顺序执行 单指令发射处理器

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

GPU 层面

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

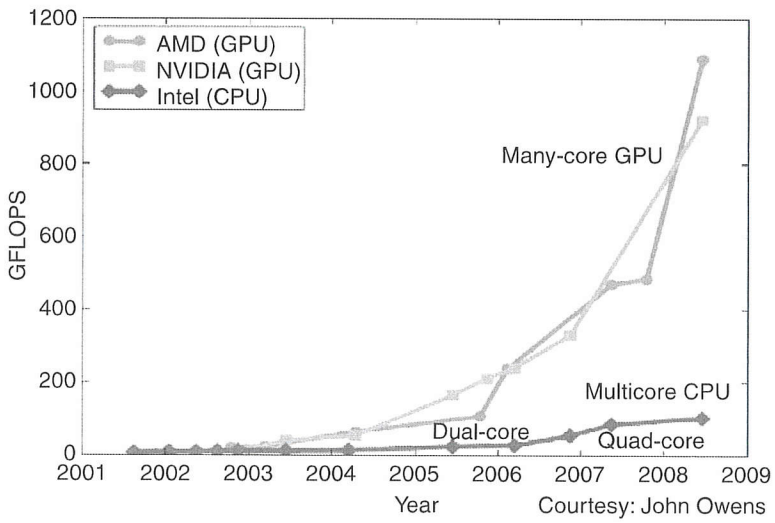


FIGURE 1.1

Enlarging performance gap between GPUs and CPUs.

amounted to 相当于. 达到

electrical potential 电势

to date. 到今天

sophisticated 精密的. 高级. 复杂

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

One might ask why there is such a large performance gap between many-core GPUs and general-purpose multicore CPUs. The answer lies in the differences in the fundamental design philosophies between the two types of processors, as illustrated in Figure 1.2. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed. As of 2009, the new general-purpose, multicore microprocessors typically have four large processor cores designed to deliver strong sequential code performance.

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

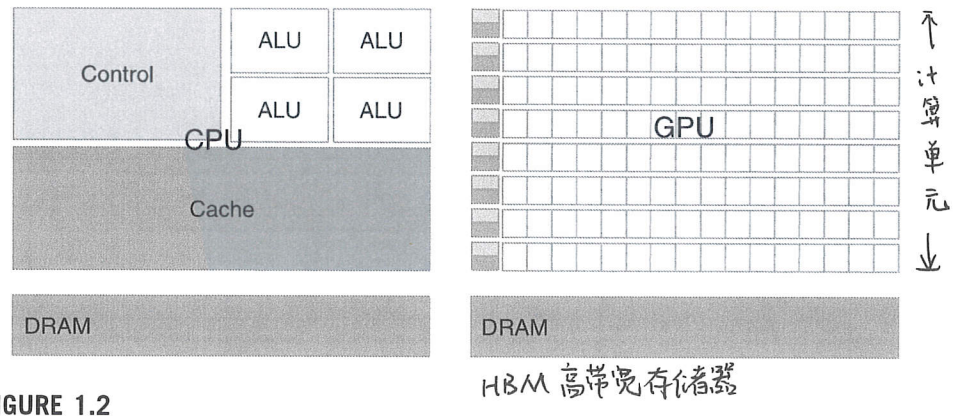


FIGURE 1.2

CPUs and GPUs have fundamentally different design philosophies.

Q: 这段总结一下.

通用处理器因为要考虑到各种系统软件. APP. 硬件相关的要求, 很难提升内存带宽

exerts 施加, 发挥. 运用

prevailing solution 主流, 普遍的解决方式

Q: 目前主流的解决方案是优化大量线程的执行吞吐量

大致说 GPU 通过大量并行线程, 减少每个线程需要的控制逻辑.

不再管理复杂的上下文切换. 应对长延迟内存访问的问题.

GPU 相比 CPU 一大优势在于内存 IO 的带宽

the relaxed memory mode

松散内存模型

: 在并行计算/多核处理器中, 内存访问的顺序和同步性不会严格按照程序指定顺序执行

在 GPU 编程中, 松散模型是默认内存模型

frame buffer requirements.

帧缓冲区要求.

和图像处理相关的硬件要求

需要大量的内存带宽支持图形渲染. 显示

Q: GPU 用大量并行线程, 反而减少控制逻辑

GPU 的线程调度和同步是批量化的.

核心在批量化 ☆

CUDA 实际上是支持 CPU 和 GPU 同时使用的模型.

G80, was capable of moving data at about 85 gigabytes per second (GB/s) in and out of its main dynamic random access memory (DRAM). Because of frame buffer requirements and the relaxed memory model—the way various system software, applications, and input/output (I/O) devices expect their memory accesses to work—general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. In contrast, with simpler memory models and fewer legacy constraints, the GPU designers can more easily achieve higher memory bandwidth. The more recent NVIDIA® GT200 chip supports about 150 GB/s. Microprocessor system memory bandwidth will probably not grow beyond 50 GB/s for about 3 years, so CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

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

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

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

negligible 可以忽略的

the advent of ...的出现

clinical applications. 临床应用.

eg: 临床影像不能依赖服务器集群处理图像.

It is also important to note that performance is not the only decision factor when application developers choose the processors for running their applications. Several other factors can be even more important. First and foremost, the processors of choice must have a very large presence in the marketplace, referred to as the installation base of the processor. The reason is very simple. The cost of software development is best justified by a very large customer population. Applications that run on a processor with a small market presence will not have a large customer base. This has been a major problem with traditional parallel computing systems that have negligible market presence compared to general-purpose microprocessors. Only a few elite applications funded by government and large corporations have been successfully developed on these traditional parallel computing systems. This has changed with the advent of many-core GPUs. Due to their popularity in the PC market, hundreds of millions of GPUs have been sold. Virtually all PCs have GPUs in them. The G80 processors and their successors have shipped more than 200 million units to date. This is the first time that massively parallel computing has been feasible with a mass-market product. Such a large market presence has made these GPUs economically attractive for application developers.

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

Yet another important consideration in selecting a processor for executing numeric computing applications is the support for the Institute of Electrical and Electronics Engineers (IEEE) floating-point standard. The standard makes it possible to have predictable results across processors from different vendors. While support for the IEEE floating-point standard

be ported to 被移植到

primarily single precision 主要单精度

heroic 英勇，非凡

Q: OpenCL与 OpenGL

OpenGL: Open Graphics Lib. 图形渲染.

OpenCL: Open Computing Lang.
并行计算.

was not strong in early GPUs, this has also changed for new generations of GPUs since the introduction of the G80. As we will discuss in Chapter 7, GPU support for the IEEE floating-point standard has become comparable to that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable values as the CPUs. Today, a major remaining issue is that the floating-point arithmetic units of the GPUs are primarily single precision. Applications that truly require double-precision floating point were not suitable for GPU execution; however, this has changed with the recent GPUs, whose double-precision execution speed approaches about half that of single precision, a level that high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphic application programming interface (API) functions to access the processor cores, meaning that OpenGL® or Direct3D® techniques were needed to program these chips. This technique was called GPGPU, short for general-purpose programming using a graphics processing unit. Even with a higher level programming environment, the underlying code is still limited by the APIs. These APIs limit the kinds of applications that one can actually write for these chips. That's why only a few people could master the skills necessary to use these chips to achieve performance for a limited number of applications; consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent results.

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

rendering. 渲染,表现

the PCI Express. 外设组件快速通道

transcendental functions. 超越函数

SMs: Streaming Multiprocessors.

Nvidia. A100. GPU

多个GPC组成.

TPC: Texture Processor Cluster

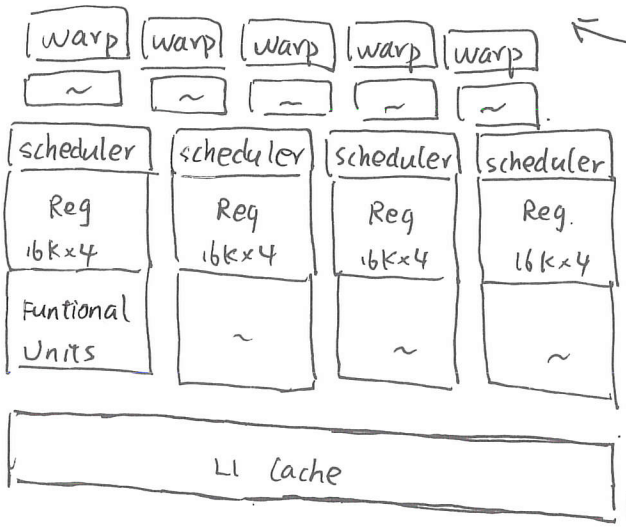
SMs 流处理器(核心)

CORE.

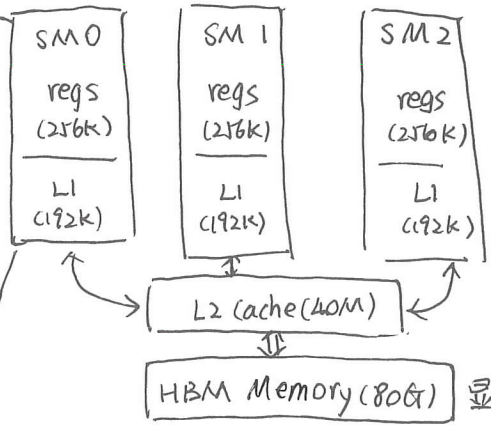
SM: 可并发执行数百个线程

一个Block上线程在同一个SM上.

而一个SM的 cache 限制了每个Block 线程数量



GPU的缓存机制(同样多级设计)



regs: register files.

~ SM108

显存(GPU的独立内存)

L2 cache: Shared across all SMs.

L1 cache: Shared Per SMs.

GPU run threads in groups of 32.

each group is known as a warp.

在一个SM内部有64个warp, Warp: GPU单个时钟周期并行执行的线程组

其中每4个 warp 可以进行一个并发的执行

一个warp包含32个线程

Schedular: 线程调度器

GPU如何加速AI计算 Img2Col

eg. 卷积会变成两个大的矩阵相乘.

1.2 ARCHITECTURE OF A MODERN GPU

Figure 1.3 shows the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Figure 1.3, two SMs form a building block; however, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Also, each SM in Figure 1.3 has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 4 gigabytes of graphics double data rate (GDDR) DRAM, referred to as *global memory* in Figure 1.3. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images, and texture information for three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

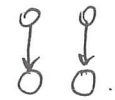
The G80 that introduced the CUDA architecture had 86.4 GB/s of memory bandwidth, plus an 8-GB/s communication bandwidth with the CPU. A CUDA application can transfer data from the system memory at 4 GB/s and at the same time upload data back to the system memory at 4 GB/s. Altogether, there is a combined total of 8 GB/s. The communication bandwidth is much lower than the memory bandwidth and may seem like a limitation; however, the PCI Express® bandwidth is comparable to the CPU front-side bus bandwidth to the system memory, so it's really not the limitation it would seem at first. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

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

AI计算与GPU的关系

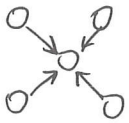
1). 不是所有AI计算都是线程独立

① Element-wise.



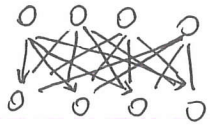
eg. AX+Y.

② Local



Convolution.

③ All to All.



MatMul.

2). AI计算层级.

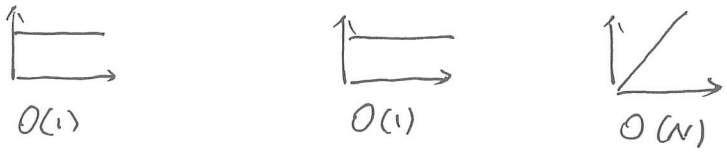
① 网格 Grid.: 所有要执行的任务.

② 网格Grid中包含了很多相同 thread.数量的Block.

③ Block中的线程数独立执行. 可以通过本地数据共享.

(BlockA和BlockB是独立的) (一个BlockA内部的线程数据共享)

3). 通过并行提升计算强度 (GPU角度)



O(1) O(1) O(N)

$$\text{Arithmetic Intensity} = \frac{\text{算术操作次数}}{\text{内存访问次数}}$$

越高说明内存的利用率越高.

适合并行计算.

N: 输入数据规模.

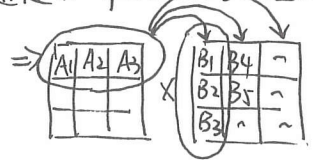
X- increasing N

Y- Arithmetic Intensity.

Arithmetic Complexity: 某任务所需的算术操作[总数]

4). 计算强度与矩阵乘的关系.

通过Img2Col.多维的输入已经被拉成一个大矩阵.



⇒ line_A x [ColB1, ColB2, ColB3]

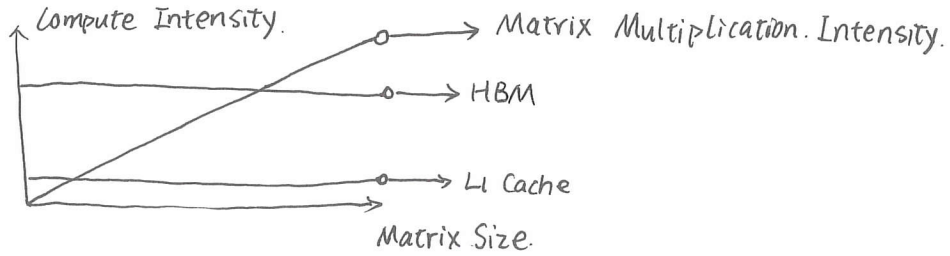
矩阵A的一行与B的每一列运算

矩阵A只加载一行数据,但是B要加载多列

Q: 让硬件恰好把缓存起来的维度设置 ⑦

维度设置过高的话, GPU就会空闲等待内存搬运 ⇒ 内存读写延时.

数据设置哪个层级的缓存中非常重要 ☆



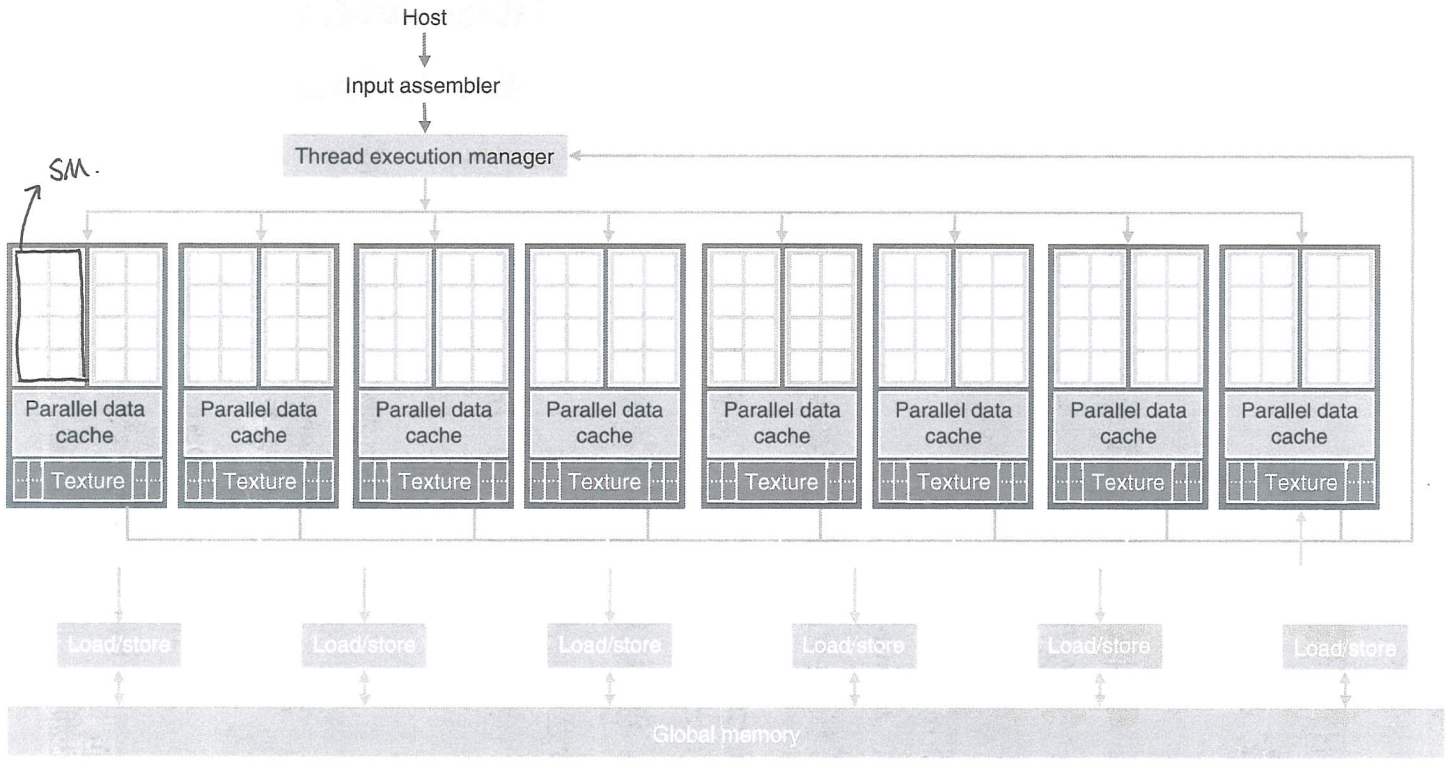


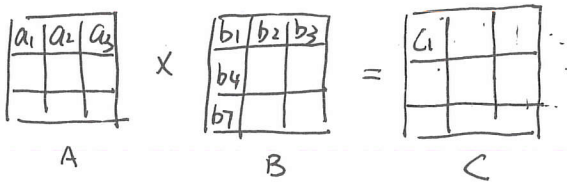
FIGURE 1.3

Architecture of a CUDA-capable GPU.

Q: 关于 Arithmetic Complexity 与 Arithmetic Intensity.

1). Arithmetic Complexity: 执行某个计算任务所需要的算术操作总数.

eg1: 矩阵乘法.



在真实计算中 FMA: Fused Multiply-Add.

$(a_1, a_2, a_3) \times (b_1, b_2, b_3)$ 有特殊的实现方式, 计 $O(N)$

∴ 最后的结果并不是 $O(N^4)$. 注意.

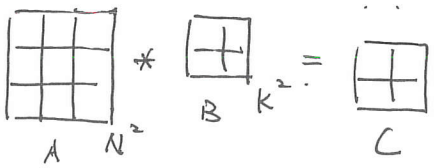
<硬件电路层面的实现>

针对 A 的一行与 B 的一列的计算总量是 $O(N)$

A 的一行与 B 的每一列计算总量 $O(N^2)$

⇒ 整个矩阵乘法的计算总量 $O(N^3)$

eg2. 卷积运算



针对每个元素 C_i 都是 K^2 的计算总量.

⇒ 推广到 C 的所有元素, 计算总量为 $O(N^2K^2)$

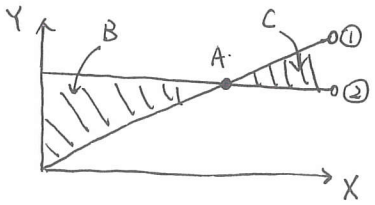
2). $\text{Arithmetic Intensity} = \frac{\text{Arithmetic Complexity}}{\text{Total Memory Access}}$

衡量计算密度与内存带宽需求的比率, 越高越适合并行计算

3). $\text{Compute Intensity} = \frac{\text{Arithmetic Complexity}}{\text{Total Data Access.}}$ 计算强度

PS. 这里的 Total Data Access 数据访问通常是内存访问

Compute Intensity 越高越适合在 GPU 执行.



该图反应的是内存带宽对计算强度的影响.

Y: Compute Intensity

X: Matrix Size.

①: Matrix Multiplication Arithmetic Intensity. $O(N)$

②: GPU Compute Intensity.

GPU 最大并行能提供的最大计算强度

A点: GPU IO 能力刚刚好完美满足计算需求的点.

B区: 计算单元空闲

C区: 内存空闲 / 读写带宽不足

Tip: 这个图要结合当前缓存的空间大小一起判断☆

比如 L1 缓存 192K, 空间小, 只对应一个 SM, 所以提供的计算强度低.

如果 SM 计算完当前数据, 则需要 L1 去 L2 拿到新的计算数据 ⇒ 产生对内存带宽的要求

tile

! 所以在计算不同阶段, 根据矩阵不同维度, 预判断矩阵数据应该存在哪个缓存层级就很重要

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

1.3 WHY MORE SPEED OR PARALLELISM?

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

myriad : 无数, 大量
mass-market 面向大众的.
molecular 分子.
optics 光学.

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

manipulation 操作, 控制
versus 对比 (vs.)

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition television (HDTV) versus older National Television System Committee (NTSC) television. Once we experience the level of details offered by HDTV, it is very hard to go back to older technology. But, consider all the processing that is necessary for that HDTV. It is a very parallel process, as are 3D imaging and

view synthesis 视觉合成.

high-resolution display 高分辨率显示屏

be reconciled at 被协调.

on a regular basis. 定期. 经常性的.

various granularities. 不同粒度.

hinder 阻碍.

Q: 编程模型不应阻碍并行实现
模型应该足够灵活来支持不同粒度并行.

visualization. In the future, new functionalities such as view synthesis and high-resolution display of low-resolution videos will demand that televisions have more computing power.

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

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

#通过提高算力支持物理模拟仿真动态道路生成之类.

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

注[1]

How many times speedup can be expected from parallelizing these super-application? It depends on the portion of the application that can be parallelized. If the percentage of time spent in the part that can be parallelized is 30%, a 100× speedup of the parallel portion will reduce the execution time by 29.7%. The speedup for the entire application will be only 1.4×. In fact, even an infinite amount of speedup in the parallel portion can only slash less 30% off execution time, achieving no more than 1.43× speedup.

extensive optimization 广泛优化/粗糙优化

tuning after 在...后调整.

saturates. 饱合,达到饱合值

utilize. 利用,使用.

complement 补充

exploiting. 利用,开发.

heterogeneous. 异构

pit area 坑区.

简单粗暴的程序并行会导致.
带宽饱合,所以不推荐.

On the other hand, if 99% of the execution time is in the parallel portion, a 100× speedup will reduce the application execution to 1.99% of the original time. This gives the entire application a 50× speedup; therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speedup its execution.

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

合理利用GPU显存.

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

Figure 1.4 illustrates the key parts of a typical application. Much of the code of a real application tends to be sequential. These portions are considered to be the pit area of the peach; trying to apply parallel computing techniques to these portions is like biting into the peach pit—not a good feeling! These portions are very difficult to parallelize. CPUs tend to do a very good job on these portions. The good news is that these portions, although they can take up a large portion of the code, tend to account for only a small portion of the execution time of superapplications.

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

analogous. 类似的.

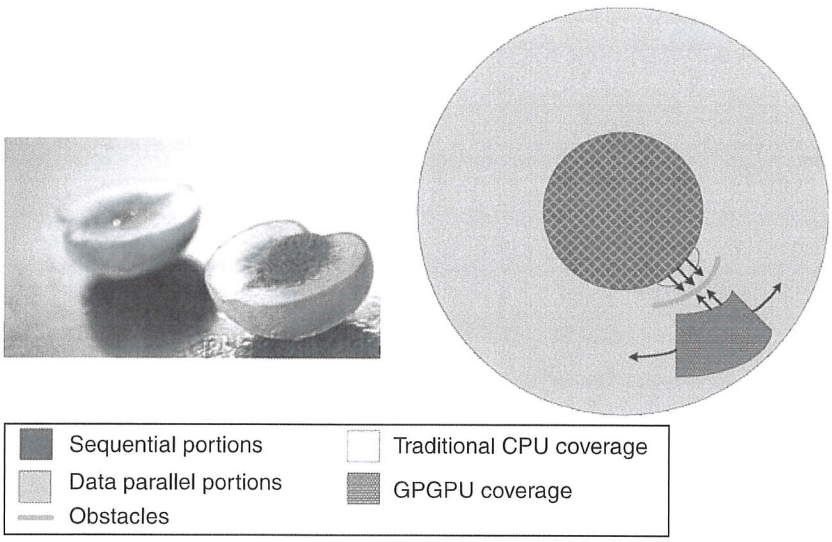


FIGURE 1.4

Coverage of sequential and parallel application portions.

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

1.4 PARALLEL PROGRAMMING LANGUAGES AND MODELS

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

scale beyond. 扩展超越 / 超越…规模.

however, can be extremely high due to lack of shared memory across computing 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 currently provides very limited shared memory capability between the CPU and the GPU. Programmers need to manage the data transfer between the CPU and GPU in a manner similar to "one-sided" message passing, a capability whose absence in MPI has been historically considered as a major weakness of MPI.

OpenMP supports shared memory, so it offers the same advantage as CUDA in programming efforts; however, it has not been able to scale beyond a couple hundred computing nodes due to thread management overheads and cache coherence hardware requirements. CUDA achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements. As we will see, however, CUDA does not support as wide a range of applications as OpenMP due to these scalability tradeoffs. On the other hand, many superapplications fit well into the simple thread management model of CUDA and thus enjoy the scalability and performance.

Aspects of CUDA are similar to both MPI and OpenMP in that the programmer manages the parallel code constructs, although OpenMP compilers do more of the automation in managing parallel execution. Several ongoing research efforts aim at adding more automation of parallelism management and performance optimization to the CUDA tool chain. Developers who are experienced with MPI and OpenMP will find CUDA easy to learn. Especially, many of the performance optimization techniques are common among these models.

More recently, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a standardized programming model called 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. OpenCL is a standardized programming model in that applications developed in OpenCL can run without modification on all processors that support the OpenCL language extensions and API.

The reader might ask why the book is not based on OpenCL. The main reason is that OpenCL was still in its infancy when this book was written. The level of programming constructs in OpenCL is still at a lower level than CUDA and much more tedious to use. Also, the speed achieved in an application expressed in OpenCL is still much lower than in CUDA on

amenable. 适合的，顺从的.

initial 初始.

①. 从并行编程的角度思考.

②. 如何确保并行编程的正确性，可靠性

③. 针对未来的硬件迭代 做到可扩展性

the platforms that support both. Because programming massively parallel processors is motivated by speed, we expect that most who program massively parallel processors will continue to use CUDA for the foreseeable future. Finally, those who are familiar with both OpenCL and CUDA know that there is a remarkable similarity between the key features of OpenCL and CUDA; that is, a CUDA programmer should be able to learn OpenCL programming with minimal effort. We will give a more detailed analysis of these similarities later in the book.

1.5 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 write a parallel program in an hour. But, we're going to dedicate many pages to materials on how to do *high-performance* parallel programming, and we believe that it will become easy once you develop the right insight and go about it the right way. In particular, we will focus on *computational thinking* techniques that will enable you to think about problems in ways that are amenable to high-performance parallel computing.

Note that hardware architecture features have constraints. High-performance parallel programming on most of the chips will require some knowledge of how the hardware actually works. It will probably take 10 more years before we can build tools and machines so most programmers can work without this knowledge. 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 constitute a subtle issue in parallel computing. Those who have worked on parallel systems in the past know that achieving initial performance is not enough. The challenge is to achieve it in such a way that you can debug the code and support the users. We will show that with the CUDA programming model that focuses on data parallelism, one can achieve both high performance and high reliability in their applications.

Our third goal is achieving 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

SPMD.

Single Program Multi Data.

machines. We want to help you to master parallel programming so your programs can 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 quite a few principles and patterns of parallel programming in this book. We cannot guarantee that we will cover all of them, however, so we have selected 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.

1.6 ORGANIZATION OF THE BOOK

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

Chapter 3 introduces 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 that supports heterogeneous CPU/GPU joint computing and the widely used single-program, multiple-data (SPMD) parallel programming model. It then covers the thought processes involved in: (1) identifying the part of application programs to be parallelized, (2) isolating the data to be used by the parallelized code by 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 individual 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. Although the objective of Chapter 3 is to teach enough concepts of the CUDA programming model so the readers can write 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–matrix multiplication to make this chapter concrete.

thought processes. 思维过程.

grounded in 根基于

concludes. 推断，总结.

concludes with 以…结束.

a treatment of parallel programming styles and models.
对并行编程风格和模型的讨论.

pertinent 切题的，恰当的

decomposition. 分解.

draw on A from B. 借鉴A来自B. 使用B的经验来学习A.

CUDA kernel function.
在GPU上执行一个并行计算任务.
其实就是在GPU上启动的一个函数，会被上万线程并行执行.

Chapters 4 through 7 are designed to give the readers more in-depth understanding of the CUDA programming model. Chapter 4 covers the thread organization and execution model required to fully understand the execution behavior of threads and basic performance concepts. Chapter 5 is dedicated to the special memories that can be used to hold CUDA variables for improved program execution speed. Chapter 6 introduces the major factors that contribute to the performance of a CUDA kernel function. Chapter 7 introduces the floating-point representation and concepts such as precision and accuracy. Although these chapters are based on CUDA, they help the readers build a foundation for parallel programming in general. We believe that humans understand best when we learn from the bottom up; that is, we must first learn the concepts in the context of a particular programming model, which provides us with a solid footing to generalize our knowledge to other programming models. As we do so, we can draw on our concrete experience from the CUDA model. An in-depth experience with the CUDA model also enables us to gain maturity, which will help us learn concepts that may not even be pertinent to the CUDA model.

Chapters 8 and 9 are case studies of two real applications, which take the readers through the thought processes of parallelizing and optimizing their applications for significant speedups. For each application, we begin 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 necessary to achieve high performance. These two chapters help the readers put all the materials from the previous chapters together and prepare for their own application development projects.

Chapter 10 generalizes the parallel programming techniques into problem decomposition principles, algorithm strategies, and computational thinking. It does so by covering the concept of organizing the computation tasks of a program so they can be done in parallel. We begin by discussing the translational process of organizing abstract scientific concepts into computational tasks, an important first step in producing quality application software, serial or parallel. The chapter then addresses parallel algorithm structures and their effects on application performance, which is grounded in the performance tuning experience with CUDA. The chapter concludes with a treatment of parallel programming styles and models, allowing the readers to place their knowledge in a wider context. With this chapter, the readers can begin to generalize from the SPMD programming style to other styles of parallel programming, such as loop parallelism in OpenMP

fork-join in P-thread programming.

fork-join 模式：由两阶段组成的并行编程模式

fork：主线程分割为多个子任务，下发到不同的线程处理。

join：主线程等待子线们执行结束合并。

P-thread：POSIX 多线程编程

and fork–join in p-thread programming. 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 gained in this book.

Chapter 11 introduces the OpenCL programming model from a CUDA programmer's perspective. The reader will find OpenCL to be extremely similar to CUDA. The most important difference arises from OpenCL's use of API functions to implement functionalities such as kernel launching and thread identification. The use of API functions makes OpenCL more tedious to use; nevertheless, a CUDA programmer has all the knowledge and skills necessary to understand and write OpenCL programs. In fact, we believe that the best way to teach OpenCL programming is to teach CUDA first. We demonstrate this with a chapter that relates all major OpenCL features to their corresponding CUDA features. We also illustrate the use of these features by adapting our simple CUDA examples into OpenCL.

Chapter 12 offers some concluding remarks and an outlook for the future of massively parallel programming. We 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 and Further Reading

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

Khronos Group. (2009). *The OpenCL Specification Version 1.0*. Beaverton, OR: Khronos Group. (http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf).

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns of parallel programming*. Upper Saddle River, NJ: Addison-Wesley.

Message Passing Interface Forum. (2009). *MPI: A Message-Passing Interface Standard, Version 2.2*. Knoxville: University of Tennessee. (http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf).

NVIDIA. (2007). *CUDA programming guide*. Santa Clara, CA: NVIDIA Corp.

OpenMP Architecture Review Board. (2005). *OpenMP Application Program Interface*. (http://www.openmp.org/mp-documents/spec25.pdf).

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

④. 从高层抽象到底层硬件的执行分析

核心角度：内存，带宽，时延. ⇒ 如何根据这三个指标分配已资源

eg. 输入图片　　全部计算任务的合集，作为输入. ⇒ Input

网格切分　　全局范围的计算任务. 把任务划分到多个 Block. ⇒ GPU

块切分.　　局部范围的计算. 使用 L1 Cache / Shared Memory ⇒ SM.

单像素.　　计算单元 Register Files. ⇒ CUDA core.

计算任务分级 ⇒ 优化内存访问. ⇒ 不同缓存层级对应效率实现 100% 的计算强度不同

⇒ 目的是匹配计算强度.　eg. All-To-All 的计算模式局部性差，而 Local, Element-wise 相反

∴ 后面两种计算模式更适合 GPU.

计算强度比 Arithmetic Intensity Scales.

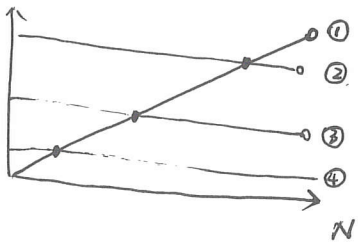
eg. 矩阵乘的计算强度 $O(N^3)$. 而数据搬运的强度是 $O(N^2)$ ⇒ 比值 $= \frac{O(N^3)}{O(N^2)} = O(N)$.

随着比值 $O(N)$ 的升高，不同的缓存层级对应不同范围的 $O(N)$

当 $O(N)$ 较小时，数据读写量低，计算强度低. 为了掩盖时延的问题，考虑 Register Files / L1 Cache.

当 $O(N)$ 较大时，对应的计算强度高 ⇒ L2 Cache / HBM.

Data Location.	Bandwidth.	Compute Intensity.	Tensor Core.
L1 Cache / Shared.	1 (max).	3	3
L2.	2.	2	2
HBM.	3.	1 (max)	1 (max).



① Matrix Multiplication Arithmatic Intensity.

② HBM

③. L2 Cache.

④. L1 Cache / Shared Memory.

☆ 数据的位置对实现 GPU 满功率执行很重要

总结：GPU 为了掩盖时延 ⇒ 设计多级缓存，超配已线程.

多级缓存的带宽限制 ⇒ 决定了内存空间，硬件资源的分配.

von Neumann, J. (1945). *First draft of a report on the EDVAC*. Contract No. W-670-ORD-4926, U.S. Army Ordnance Department and University of Pennsylvania (reproduced in Goldstine H. H. (Ed.), (1972). *The computer: From Pascal to von Neumann*. Princeton, NJ: Princeton University Press).

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

① Tip：关于CPU与GPU的核心区别

首先，我们的目的是程序运行的越快越好，但是以循环为例，

单次循环的计算速度 >>> 内存中的数据读写速度。

key ⇒ 如何提高内存的利用率，降低内存低利用率带来的影响。

CPU	GPU
增加单个线程的复杂度	增加更多线程，隐藏时延的影响
Single Instr Multi Data.	Single Instr Multi Threads.

② GPU缓存：

Register Files.	单个线程使用的独立Reg空间	Driver和Runtime动态分配，激活线程数	256K
L1 Cache / Shared Memory	线程块内使用/共享的空间		192K
L2 Cache.	跨线程块共享	自动.	40M
HBM	全局	自动.	80G

Tip：Register File 与 L1 Cache 是单个SM的空间大小，

而L2和HBM本身就是共享，所以它们就这么大。

因为越靠下的缓存层级离SM越远，传输数据的时延越高，

那么要实现"隐时延"的效果，需要数据在SM中的计算时长增加，对应的计算强度越高。

③ thread required & thread available & 线程超配

假如一块板子满功率执行任务，使内存利用率达到100%？

此时计算任务需要要求的线程数是 thread required.

而所有板子的物理结构都是超配：thread available > thread required.

线程超配 Thread Oversubscription.

任务要求的线程数 > GPU能提供的最大并行线程数。

从硬件角度来看：计算单元的硬件数量 >> 内存带宽能支持的最大并行度。