CHAPTER 1
Introduction

This book explores the hardware design of graphics processor units (GPUs). GPUs were initially introduced to enable real-time rendering with a focus on video games. Today GPUs are found everywhere from inside smartphones, laptops, datacenters, and all the way to supercomputers. Indeed, an analysis of the Apple A8 application processor shows that it devotes more die area to its integrated GPU than to central processor unit (CPU) cores [A8H]. The demand for ever more realistic graphics rendering was the initial driver of innovation for GPUs [Montrym and Moreton, 2005]. While graphics acceleration continues to be their primary purpose, GPUs increasingly support non-graphics computing. One prominent example of this receiving attention today is the growing use GPUs to develop and deploying machine learning systems [NVIDIA Corp., 2017]. Thus, the emphasis of this book is on features relevant to improving the performance and energy efficiency of non-graphics applications.

This introductory chapter provides a brief overview of GPUs. We start in Section 1.1 by considering the motivation for the broader category of computation accelerators to understand how GPUs compare to other options. Then, in Section 1.2, we provide a quick overview of contemporary GPU hardware. Finally, Section 1.4 provides a roadmap to the rest of this book.

1.1 THE LANDSCAPE OF COMPUTATION ACCELERATORS

For many decades, succeeding generations of computing systems showed exponential increasing performance per dollar. The underlying cause was a combination of reduced transistor sizes, improvements in hardware architecture, improvements in compiler technology, and algorithms. By some estimates half of those performance gains were due to reductions in transistor size that lead to devices that operate faster [Hennessy and Patterson, 2011]. However, since about 2005, the scaling of transistors has failed to follow the classical rules now known as Dennard Scaling [Dennard et al., 1974]. One key consequence is that clock frequencies now improve much more slowly as devices become smaller. To improve performance requires finding more efficient hardware architectures.

By exploiting hardware specialization it is possible to improve energy efficiency by as much as 500× [Hameed et al., 2010]. As shown by Hameed et al., there are several key aspects to attaining such gains in efficiency. Moving to vector hardware, such as that found in GPUs, yields about a 10× gain in efficiency by eliminating overheads of instruction processing. A large part of the remaining gains of hardware specialization are a result of minimizing data movement which can be achieved by introducing complex operations that perform multiple arithmetic operations while avoiding accesses to large memory arrays such as register files.

A key challenge for computer architects today is finding better ways to balance the gains in efficiency that can be obtained by using specialized hardware with the need for flexibility required to support a wide range of programs. In the absence of architectures only algorithms that can be used for a large number of applications will run efficiently. An emerging example is hardware specialized for supporting deep neural networks such as Google's Tensor Processing Unit [Jouppi et al., 2017]. While machine learning appears likely to occupy a very large fraction of computing hardware resources, and these may migrate to specialized hardware, we argue there will remain a need for efficiently supporting computation expressed as software written in traditional programming languages.

One reason for the strong interest in GPU computing outside of the use of GPUs for machine learning is that modern GPUs support a Turing Complete programming model. By Turing Complete, we mean that any computation can be run given enough time and memory. Relative to special-purpose accelerators, modern GPUs are flexible. For software that can make full use of GPU hardware, GPUs can be an order of magnitude more efficient than CPUs [Lee et al., 2010]. This combination of flexibility and efficiency is highly desirable. As a consequence many of the top supercomputers, both in terms of peak performance and energy efficiency now employ GPUs [top]. Over succeeding generations of products, GPU manufacturer's have refined the GPU architecture and programming model to increase flexibility while simultaneously improving energy efficiency.

1.2 GPU HARDWARE BASICS

Often those encountering GPUs for the first time ask whether they might eventually replace CPUs entirely. This seems unlikely. In present systems GPUs are not stand-alone computing devices. Rather, they are combined with a CPU either on a single chip or by inserting an add-in card containing only a GPU into a system containing a CPU. The CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU. One reason for this division of labor between CPU and GPU is that the beginning and end of the computation typically require access to input/output (I/O) devices. While there are ongoing efforts to develop application programming interfaces (APIs) providing I/O services directly on the GPU, so far these all assume the existence of a nearby CPU [Kim et al., 2014, Silberstein et al., 2013]. These APIs function by providing convenient interfaces that hide the complexity of managing communication between the CPU and GPU rather than eliminating the need for a CPU entirely. Why not eliminate the CPU? The software used to access I/O devices and otherwise provide operating system services would appear to lack features, such as massive parallelism, that would make them suitable to run on the GPU. Thus, we start off by considering the interaction of the CPU and GPU.

An abstract diagram showing a typical system containing a CPU and GPU is shown in Figure 1.1. On the left is a typical discrete GPU setup including a bus connecting the CPU and GPU (e.g., PCIe) for architectures such as NVIDIA's Volta GPU, and on the right is a logical diagram of a typical integrated CPU and GPU such as AMD's Bristol Ridge APU or a mobile GPU. Notice that systems including discrete GPUs have separate DRAM memory spaces for the CPU (often called system memory) and the GPU (often called device memory). The DRAM technology used for these memories is often different (DDR for CPU vs. GDDR for GPU). The CPU DRAM is typically optimized for low latency access whereas the GPU DRAM is optimized for high throughput. In contrast, systems with integrated GPUs have a single DRAM memory space and therefore necessarily use the same memory technology. As integrated CPUs and GPUs are often found on low-power mobile devices the shared DRAM memory is often optimized for low power (e.g., LPDDR).

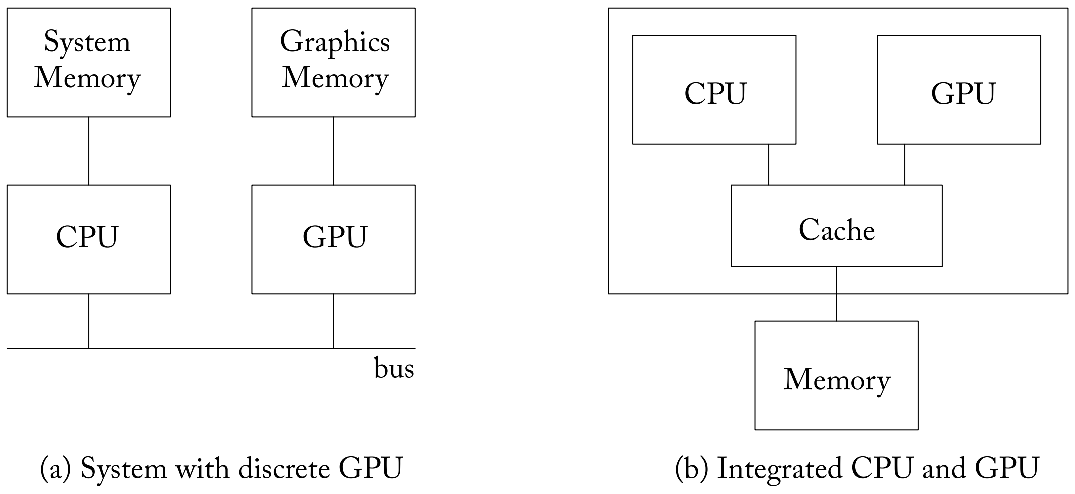

Figure 1.1: GPU computing systems include CPUs.

A GPU computing application starts running on the CPU. Typically, the CPU portion of the application will allocate and initialize some data structures. On older discrete GPUs from both NVIDIA and AMD the CPU portion of the GPU Computing application typically allocates space for data structures in both CPU and GPU memory. For these GPUs, the CPU portion of the application must orchestrate the movement of data from CPU memory to GPU memory. More recent discrete GPUs (e.g., NVIDIA's Pascal architecture) have software and hardware support to automatically transfer data from CPU memory to GPU memory. This can be achieved by leveraging virtual memory support [Gelado et al., 2010], both on the CPU and GPU. NVIDIA calls this "unified memory." On systems in which the CPU and GPU are integrated onto the same chip and share the same memory, no programmer controlled copying from CPU memory to GPU memory is necessary. However, because CPUs and GPUs use caches and some of these caches may be private, there can be a cache-coherence problem, which hardware developers need to address [Power et al., 2013b].

At some point, the CPU must initiate computation on the GPU. In current systems this is done with the help of a driver running on the CPU. Before launching computation on the GPU, a GPU computing application specifies which code should run on the GPU. This code is commonly referred to as a kernel (more details in Chapter 2). At the same time the CPU portion of the GPU computing application also specifies how many threads should run and where these threads should look for input data. The kernel to run, number of threads, and data location are conveyed to the GPU hardware via the driver running on the CPU. The driver will translate the information and place it memory accessible by the GPU at a location where the GPU is configured to look for it. The driver then signals the GPU that it has new computations it should run.

A modern GPU is composed of many cores, as shown in Figure 1.2. NVIDIA calls these cores *streaming multiprocessors* and AMD calls them *compute units*. Each GPU core executes a single-instruction multiple-thread (SIMT) program corresponding to the kernel that has been launched to run on the GPU. Each core on a GPU can typically run on the order of a thousand threads. The threads executing on a single core can communicate through a scratchpad memory and synchronize using fast barrier operations. Each core also typically contains first-level instruction and data caches. These act as bandwidth filters to reduce the amount of traffic sent to lower levels of the memory system. The large number of threads running on a core are used to hide the latency to access memory when data is not found in the first-level caches.

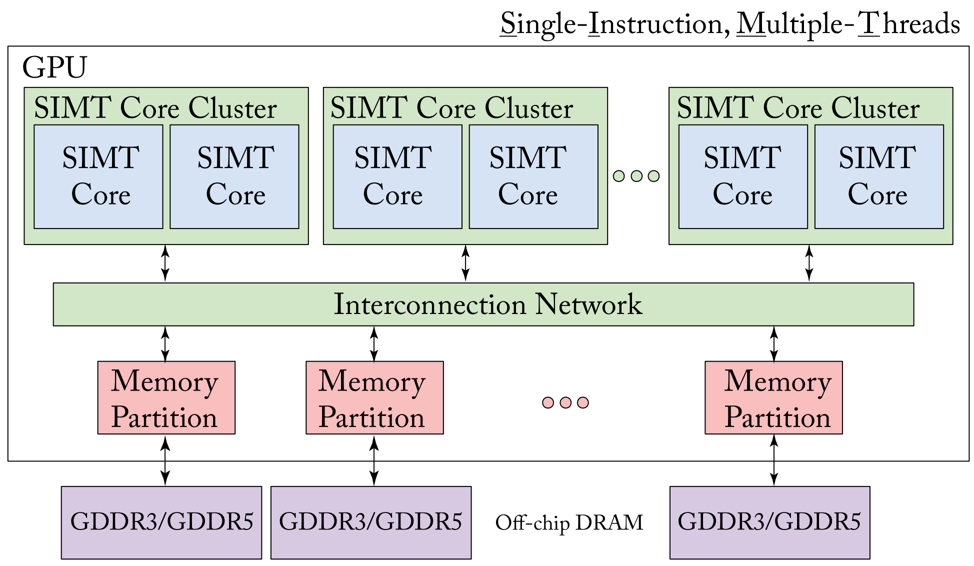

Figure 1.2: A generic modern GPU architecture.

To sustain high computation throughput it is necessary to balance high computational throughput with high memory bandwidth. This in turn requires parallelism in the memory system. In GPUs this parallelism is provided by including multiple memory channels. Often, each memory channel has associated with it a portion of last-level cache in a memory partition. The GPU cores and memory partitions are connected via an on-chip interconnection network such as a crossbar. Alternative organizations are possible. For example, the Intel Xeon Phi, which directly competes with GPUs in the supercomputing market, distributes the last-level cache with the cores.

GPUs can obtain improved performance per unit area vs. superscalar out-of-order CPUs on highly parallel workloads by dedicating a larger fraction of their die area to arithmetic logic units and correspondingly less area to control logic. To develop intuition into the tradeoffs between CPU and GPU architectures, Guz et al. [2009] developed an insightful analytical model showing how performance varies with number of threads. To keep their model simple, they assume a simple cache model in which threads do not share data and infinite off-chip memory bandwidth. Figure 1.3 which reproduces a figure from their paper, illustrates an interesting trade-off they found with their model. When a large cache is shared among a small number of threads (as is the case in multicore CPUs), performance increases with the number of threads. However, if the number of threads increases to the point that the cache cannot hold the entire working set, performance decreases. As the number of threads increases further, performance increases with the ability of multithreading to hide long off-chip latency. GPUs architectures are represented by the right-hand side of this figure. GPUs are designed to tolerate frequent cache misses by employing multithreading.

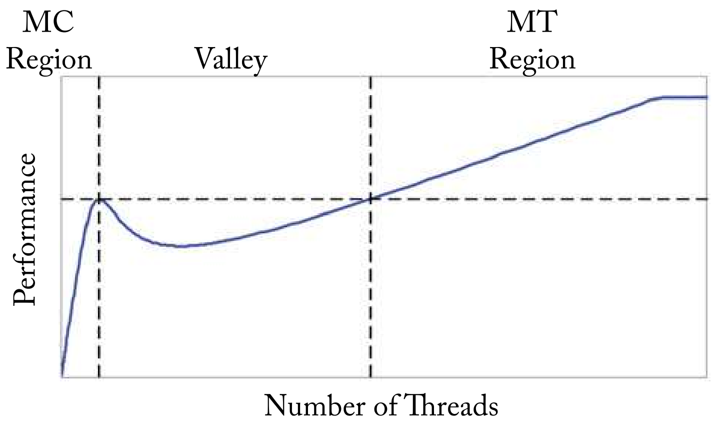

Figure 1.3: An analytical model-based analysis of the performance tradeoff between multicore (MC) CPU architectures and multithreaded (MT) architectures such as GPUs shows a "performance valley" may occur if the number of threads is insufficient to cover off-chip memory access latency (based on Figure 1 from Guz et al. [2009]).

With the end of Dennard Scaling [Horowitz et al., 2005], increasing energy efficiency has become a primary driver of innovation in computer architecture research. A key observation is that accessing large memory structures can consume as much or more energy as computation.

For example, Table 1.1 provides data on the energy for various operations in a 45 nm process technology [Han et al., 2016]. When proposing novel GPU architecture designs it is important to take energy consumption into account. To aid with this, recent GPGPU architecure simulators such as GPGPU-Sim [Bakhoda et al., 2009] incorporate energy models [Leng et al., 2013].

Table 1.1: Energy consumption of various operations for a 45 nm process technology (based on Table 1 in Han et al. [2016])

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit 32KB SRAM	5	50
32 bit DRAM	640	6400

1.3 A BRIEF HISTORY OF GPUS

This section briefly describes the history of graphics processing units. Computer graphics emerged in the 1960s with projects such as Ivan Sutherland's Sketchpad [Sutherland, 1963]. From its earliest days computer graphics have been integral to off-line rendering for animation in films and in parallel the development of real-time rendering for use in video games. Early video cards started with the IBM Monochrome Display Adapter (MDA) in 1981 which only supported text. Later, video cards introduced 2D and then 3D acceleration. In addition to video games 3D accelerators targeted computer-aided design. Early 3D graphics processors such as the NVIDIA GeForce 256 were relatively fixed-function. NVIDIA introduced programmability to the GPU in the form of vertex shaders [Lindholm et al., 2001] and pixel shaders in the GeForce 3 introduced in 2001. Researchers quickly learned how to implement linear algebra using these early GPUs by mapping matrix data into into textures and applying shaders [Krüger and Westermann, 2003] and academic work at mapping general-purpose computing onto GPUs such that the programmer did not need to know graphics soon followed [Buck et al., 2004]. These efforts inspired GPU manufacturers to directly support general-purpose computing in addition to graphics. The first commercial product to do so was the NVIDIA GeForce 8 Series. The GeForce 8 Series introduced several innovations including ability to write to arbitrary memory addresses from a shader and scratchpad memory to limit off-chip bandwidth, which had been lacking in earlier GPUs. The next innovation was enabling caching of read-write data with NVIDIA's Fermi architecture. Subsequent refinements include AMD's Fusion architecture which integrated CPU and GPU on the same die and dynamic parallelism that enables

CHAPTER 3

The SIMT Core: Instruction and Register Data Flow

In this and the following chapter we will examine the architecture and microarchitecture of modern GPUs. We divide our discussion of GPU architecture into two parts: (1) examining the SIMT cores that implement computation in this chapter and then (2) looking at the memory system in the next chapter.

In their traditional graphics-rendering role, GPUs access data sets such as detailed texture maps that are far too large to be fully cached on-chip. To enable high-performance programmability, which is desirable in graphics both to ease verification costs as the number of graphics modes increase and to enable games developers to more easily differentiate their products [Lindholm et al., 2001], it is necessary to employ an architecture that can sustain large off-chip bandwidths. Thus, today's GPUs execute tens of thousands of threads concurrently. While the amount of on-chip memory storage per thread is small, caches can still be effective in reducing a sizable number of off-chip memory accesses. For example, in graphics workloads, there is significant spatial locality between adjacent pixel operations that can be captured by on-chip caches.

Figure 3.1 illustrates the microarchitecture of the GPU pipeline discussed in this chapter. This figure illustrates the internal organization of a single SIMT-core shown in Figure 1.2. The pipeline can be divided into a SIMT front-end and a SIMD back-end. The pipeline consists of three scheduling "loops" acting together in a single pipeline: an instruction fetch loop, an instruction issue loop, and a register access scheduling loop. The instruction fetch loop includes the blocks labeled Fetch, I-Cache, Decode, and I-Buffer. The instruction issue loop includes the blocks labeled I-Buffer, Scoreboard, Issue, and SIMT Stack. The register access scheduling loop includes the blocks labeled Operand Collector, ALU, and Memory. In the rest of this chapter we help you build up a full understanding of the individual blocks in this picture by considering key aspects of the architecture that depend on each of these loops.

As there are many details involved in fully understanding this organization, we divide our discussion up in parts. We order these with the objective of developing an increasingly detailed view of the core microarchitecture. We start with a high-level view of the overall GPU pipeline and then fill in details. We call these increasingly accurate descriptions "approximations" to acknowledge that some details are omitted even in our most detailed descriptions. As the central organizing principle of today's GPUs is multithreading we organize these "approximations"

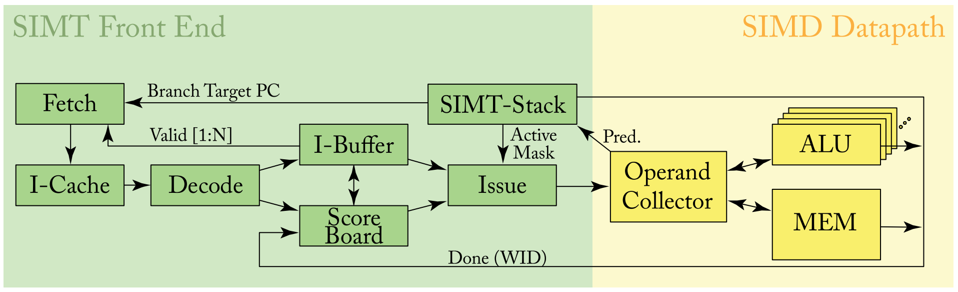

Figure 3.1: Microarchitecture of a generic GPGPU core.

around the three scheduling loops described above. We have found it convenient to organize this chapter by considering three increasingly accurate "approximation loops" that progressively take into account the details of these scheduler loops.

3.1 ONE-LOOP APPROXIMATION

We start by considering a GPU with a single scheduler. This simplified look at the hardware is not unlike what one might expect the hardware to do if they only read the description of the hardware found in the CUDA programming manual.

To increase efficiency, threads are organized into groups called "warps" by NVIDIA and "wavefronts" by AMD. Thus, the unit of scheduling is a warp. In each cycle, the hardware selects a warp for scheduling. In the one loop approximation the warp's program counter is used to access an instruction memory to find the next instruction to execute for the warp. After fetching an instruction, the instruction is decoded and source operand registers are fetched from the register file. In parallel with fetching source operands from the register file, the SIMT execution mask values are determined. The following sub-section describes how the SIMT execution mask values are determined and contrasts them with predication, which is also employed in modern GPUs.

After the execution masks and source registers are available, execution proceeds in a single-instruction, multiple-data manner. Each thread executes on the function unit associated with a lane provided the SIMT execution mask is set. As in modern CPU designs, the function units are typically heterogeneous meaning a given function unit supports only a subset of instructions. For example, NVIDIA GPUs contain a *special function unit* (SFU), *load/store unit*, *floating-point function unit*, *integer function unit*, and, as of Volta, a *Tensor Core*.

All function units nominally contain as many lanes as there are threads within a warp. However, several GPUs have used a different implementation in which a single warp or wavefront is executed over several clock cycles. This is achieved by clocking the function units at a higher frequency, which can achieve higher performance per unit area at the expense of increased energy consumption. One way to achieve higher clock frequencies for the function units is to pipeline their execution or increase their pipeline depth.

3.1.1 SIMT EXECUTION MASKING

A key feature of modern GPUs is the SIMT execution model, which from the standpoint of functionality (although not performance) presents the programmer with the abstraction that individual threads execute completely independently. This programming model can potentially be achieved via predication alone. However, in current GPUs it is achieved via a combination of traditional predication along with a stack of predicate masks that we shall refer to as the *SIMT stack*.

The SIMT stack helps efficiently handle two key issues that occur when all threads can execute independently. The first is nested control flow. In nested control flow one branch is control dependent upon another. The second issue is skipping computation entirely while all threads in a warp avoid a control flow path. For complex control flow this can represent a significant savings. Traditionally, CPUs supporting predication have handled nested control flow by using multiple predicate registers and supporting across lane predicate tests has been proposed in the literature.

The SIMT stack employed by GPUs can handle both nested control flow and skipped computation. There are several implementations described in patents and instruction set manuals. In these descriptions the SIMT stack is at least partly managed by special instructions dedicated to this purpose. Instead, we will describe a slightly simplified version introduced in an academic work that assumes the hardware is responsible for managing the SIMT stack.

To describe the SIMT stack we use an example. Figure 3.2 illustrates CUDA C code that contains two branches nested within a do-while loop and Figure 3.3 illustrates the corresponding PTX assembly. Figure 3.4, which reproduces Figure 5 in Fung et al. [Fung et al., 2007], illustrates how this code interacts with the SIMT stack assuming a GPU that has four threads per warp.

Figure 3.4a illustrates a control flow graph (CFG) corresponding to the code in Figures 3.2 and 3.3. As indicated by the label "`A/1111`" inside the top node of the CFG, initially all four threads in the warp are executing the code in Basic Block A which corresponds to the code on lines 2 through 6 in Figure 3.2 and lines 1 through 6 in Figure 3.3. These four threads follow different (divergent) control flow after executing the branch on line 6 in Figure 3.3, which corresponds to the "if" statement on line 6 in Figure 3.2. Specifically, as indicated by the label "`B/1110`" in Figure 3.4a the first three threads fall through to Basic Block B. These three threads branch to line 7 in Figure 3.3 (line 7 in Figure 3.2). As indicated by the label "`F/0001`" in Figure 3.4a, after executing the branch the fourth thread jumps to Basic Block F, which corresponds to line 14 in Figure 3.3 (line 14 in Figure 3.2).

Similarly, when the three threads executing in Basic Block B reach the branch on line 9 in Figure 3.3 the first thread diverges to Basic Block C while the second and third thread diverges to Basic Block D. Then, all three threads reach Basic Block E and execute together as indicated by the label "`E/1110`" in Figure 3.4a. At Basic Block G all four threads execute together.

How does GPU hardware enable threads within a warp to follow different paths through the code while employing a SIMD datapath that allows only one instruction to execute per cycle?