Chapman & Hall/CRC
Computational Science Series

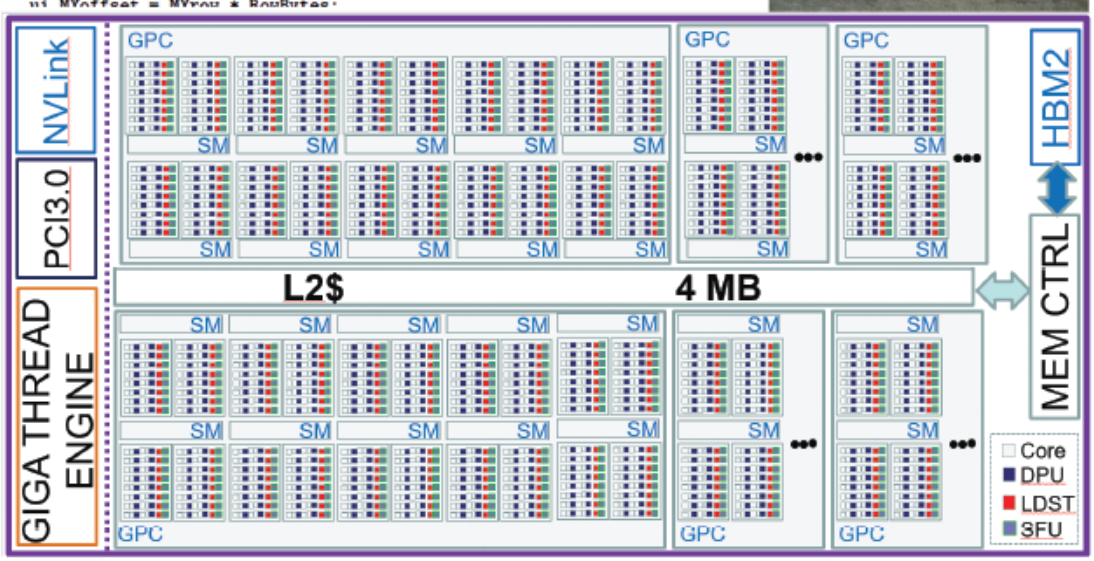
GPU Parallel Program Development Using CUDA

Tolga Soyata

CODE 9.2: imflipGCM.cu Hflip2() {...}
Hflip2() avoids computing `BlkPerRow` and `RowBytes` repeatedly.

```
// Improved Hflip() kernel that flips the given image horizontally
// BlkPerRow, RowBytes variables are passed, rather than calculated
__global__
void Hflip2(uch *ImgDst, uch *ImgSrc, ui Hpixels, ui BlkPerRow, ui RowBytes)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;

   //ui BlkPerRow = CEIL(Hpixels,ThrPerBlk);
   //ui RowBytes = (Hpixels * 3 + 3) & (~3);
   ui MYrow = MYbid / BlkPerRow;
   ui MYcol = MYgtid - MYrow*BlkPerRow*ThrPerBlk;
   if (MYcol >= Hpixels) return;    // col out of range
   ui MYmirrorcol = Hpixels - 1 - MYcol;
   ui MYoffset = MYrow * RowBytes;
```



CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Tolga Soyata

GPU Parallel Program Development Using CUDA

About the Author

Tolga Soyata received his BS degree from Istanbul Technical University, Department of Electronics and Communications Engineering in 1988. He came to the USA to pursue his graduate studies in 1990; he received his MS degree from Johns Hopkins University, Department of Electrical and Computer Engineering (ECE), Baltimore, MD in 1992 and PhD degree from University of Rochester, Department of ECE in 2000. Between 2000 and 2015, he owned his IT outsourcing and copier sales/service company. While operating his company, he came back to academia, joining University of Rochester (UR) ECE as a research scientist. Later he became an Assistant Processor - Research and continued serving as a research faculty member at UR ECE until 2016. During his tenure at UR ECE, he supervised three PhD students. Two of them received their PhD degree under his supervision and one stayed at UR ECE when he joined State University of New York - Albany (SUNY Albany) as an Associate Professor of ECE in 2016. Soyata's teaching portfolio includes VLSI, Circuits, and Parallel Programming using FPGA and GPUs. His research interests are in the field of Cyber Physical Systems, Digital Health, and high-performance medical mobile-cloud computing systems.

His entry into GPU teaching dates back to 2009, when he registered University of Rochester as a CUDA Teaching Center and CUDA Research Center. He served as the PI for these programs at UR until he left to join SUNY Albany in 2016. These programs were later named GPU Education Center and GPU Research Center by Nvidia. While at UR, he taught GPU Programming and Advanced GPU Project Development for five years, which were cross-listed between the ECE and CS departments. He has been teaching similar courses at SUNY Albany since he joined the department in 2016. This book is a product of the experiences he gained in the past seven years, while teaching GPU courses at two different institutions.

To purchase this book, use the following weblink: `https://www.amazon.com/Parallel-Program-Development-Chapman-Computational/dp/1498750753`

To cite this book, use the following BiBTeX entry:

```
@BOOK{soyata.crc18,
    AUTHOR = "T. Soyata",
    TITLE = "GPU Parallel Program Development Using CUDA",
    PUBLISHER = "Taylor and Francis",
    ISBN = "978-1498750752",
    MONTH = "Feb",
    YEAR = "2018",
}
```

To my wife Eileen
and my step children Katherine, Andrew, and Eric.

Contents

List of Figures

List of Tables

Preface

I am from the days when computer engineers and scientists had to write assembly language on IBM mainframes to develop high-performance programs. Programs were written on punch cards and compilation was a one day process; you dropped off your punch-code written program and picked up the results next day. If there was an error, you did it again. In those days, a good programmer had to understand the underlying machine hardware to produce good code. I get a little nervous when I see Computer Science students being taught only at a high abstraction level and languages like Ruby. Although abstraction is a beautiful thing to develop things without getting bogged down with unnecessary details, it is a bad thing when you are trying to develop super high performance code.

Since the introduction of the first CPU, computer architects added incredible features into CPU hardware to "forgive" bad programming skills; while you had to order the sequence of machine code instructions by hand two decades ago, CPUs do that in hardware for you today (e.g., out of order processing). A similar trend is clearly visible in the GPU world. Most of techniques that were taught as *performance improvement techniques* in GPU programming five years ago (e.g., thread divergence, shared memory bank conflicts, and reduced usage of atomics) are becoming less relevant with the improved GPU architectures, because GPU architects are adding hardware features that are improving these previous inefficiencies so much that it won't even matter if a programmer is sloppy about it within another 5–10 years. However, this is just a guess. What GPU architects can do depends on their (i) *transistor budget*, as well as (ii) their customers' demands. When I say *transistor budget*, I am referring to how many transistors the GPU manufacturers can cram into an Integrated Circuit (IC), aka a "chip." When I say *customer demands*, I mean that even if they can implement a feature, the applications that their customers are using might not benefit from it, which will mean a wasted transistor budget.

From the standpoint of writing a book, I took all of these facts to heart and decided that the best way to teach GPU programming is to show the differences among different families of GPUs (e.g., Fermi, Kepler, Maxwell, and Pascal) and point out the *trend*, which lets the reader be prepared about the upcoming advances in the next generation GPUs, and the next, and the next ... I put a lot of emphasis on concepts that will stay relevant for long period of time, rather than concepts that are platform-specific. That being said, GPU programming is all about performance and you can get a lot higher performance if you know the exact architecture you are running on, i.e., if you write platform-dependent code. So, providing platform-dependent explanations are as valuable as generalized GPU concepts. I engineered this book in such a way that the higher chapters you are reading, the more platform-specific it gets.

I believe that the most unique feature of this book is the fact that it starts explaining parallelism by using CPU multi-threading in Part I. GPU massive parallelism (which differs from CPU parallelism) is introduced in Part II. Due to the way the CPU parallelism is explained in Part I, there is a smooth transition into understanding GPU parallelism in Part II. I devised this methodology within the past six years of teaching GPU programming; I realized that the concept of massive parallelism was not clear with students who have never

taken a parallel programming class. Understanding the concept of "parallelizing a task" is a lot easier to understand in a CPU architecture, as compared to a GPU.

The book is organized as follows. In Part I (Chapters 1 through 5), a few simple programs are used to demonstrate the concept of dividing a large task into multiple parallel sub-tasks and map them to CPU threads. Multiple ways of parallelizing the same task are analyzed and their pros/cons are studied in terms of both core and memory operation. In Part II (Chapters 6 through 11) of the book, the same programs are parallelized on multiple Nvidia GPU platforms (Fermi, Kepler, Maxwell, and Pascal) and the same performance analysis is repeated. Because the core and memory structures of CPUs and GPUs are different, the results differ in interesting —and sometimes counter-intuitive— ways; these differences are pointed out and detailed discussions are provided as to what would make a GPU program work faster. The end goal is to make the programmer aware of all of the good ideas —as well as the bad ideas— so (s)he can apply the good ideas and avoid the bad ideas to his/her programs.

Although Part I and Part II totally cover what you need to write successful CUDA programs, there is always more to know. In Part III of the book, pointers are provided for readers who want to expand their horizons. Part III is *not* meant to be a complete reference to the topics that are covered in it; rather, it is meant to provide the initial introduction, from which the reader can build a momentum towards understanding the entire topic. Included topics in this part are an introduction to some of the popular CUDA libraries, such as cuBLAS, cuFFT, Nvidia Performance Primitives, and Thrust (Chapter 12), an introduction to the Open CL programming language (Chapter 13), and an overview of GPU programming using other programming languages and API libraries such as Python, Metal, Swift, OpenGL, OpenGL ES, Open CV, and Microsoft HLSL (Chapter 14), and finally the deep learning library cuDNN (Chapter 15).

Tolga Soyata

PART I

Understanding CPU Parallelism

CHAPTER 1

Introduction to CPU Parallel Programming

This book is a self-sufficient GPU and CUDA programming textbook. I can imagine the surprise of somebody who purchased a GPU Programming book and the first chapter is named "Introduction to CPU Parallel Programming." The idea is that, this book expects the readers to be sufficiently good at a low-level programming language, like C, but not in CPU parallel programming. To make this book a self-sufficient GPU programming resource for somebody that meets this criteria, any prior CPU parallel programming experience cannot be expected from the readers, yet, it is not difficult to gain sufficient CPU parallel programming skills within a few weeks with an introduction such as the Part I of this book.

No worries, in these few weeks of learning CPU parallel programming, no time will be wasted towards our eventual goal of learning GPU programming, since, almost every concept that I introduce here in the CPU world will be applicable to the GPU world. If you are skeptical, here is one example for you: The *thread ID*, or, as we will call it *tid*, is the identifier of an executing thread in a multi-threaded program, whether it is a CPU or GPU thread. All of the CPU parallel programs we write will use the *tid* concept, which will make the programs directly transportable to the GPU environment. Don't worry if the term *thread* is not familiar to you. Half the book is about threads, as it is the backbone of how CPUs or GPUs execute multiple tasks simultaneously.

1.1 EVOLUTION OF PARALLEL PROGRAMMING

A natural question that comes to one's mind is : why even bother with parallel programming ? In the 1970s, 80s, even part of 90s, we were perfectly happy with *single-threaded* programming, or, as one might call it serial programming. You wrote a program to accomplish one task. When done, it gave you an answer. Task is done ... Everybody was happy ... Although the task was done, if you were, say, doing a particle simulation that required millions, or billions of computations per second, or any other Image Processing computation that works on thousands of pixels, you wanted your program to work much faster, which meant that, you needed a faster CPU.

Up until year 2004, the CPU makers IBM, Intel, AMD gave you a faster processor by making it work at a higher speed, 16 MHz, 20 MHz, 66, 100 MHz, and eventually 200, 333, 466 MHz ... It looked like, they could keep increasing the CPU speeds and provide higher performance every year. But, in 2004, it was obvious that, continuously increasing the CPU speeds couldn't go on forever due to technological limitations. Something else was needed to continuously deliver higher performances. The answer of the CPU makers was to put two CPUs inside one CPU, even if each CPU worked at a lower speed than a single one would. For example, two CPUs (*cores*, as they called them) working at 200 MHz could do more

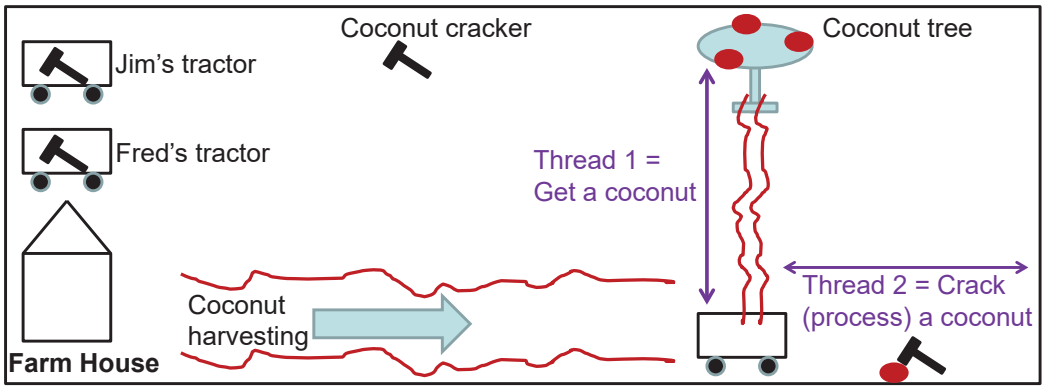


FIGURE 1.1 Harvesting each coconut requires two consecutive 30-second tasks (threads). Thread 1: get a coconut. Thread 2: crack (process) that coconut using the hammer.

computations per second cumulatively, as compared to a single core working at 300 MHz (i.e., $2 \times 200 > 300$, intuitively).

Even if the story of "multiple cores within a single CPU" sounded like a dream come true, it meant that, the programmers would now have to learn the *parallel programming* methods to take advantage of both of these cores. If a CPU could execute two programs at the same time, this automatically implied that, a programmer had to write those two programs. But, could this translate to twice the program speed ? If not, then our $2 \times 200 > 300$ thinking is flawed. What if there wasn't enough work for one core ? So, only, truly a single core was busy, while the other one was doing nothing ? Then, we are better off with s single core at 300 MHz. Numerous similar questions highlighted the biggest problem with introducing multiple cores, which is the *programming* that can allow utilizing those cores efficiently.

1.2 MORE CORES, MORE PARALLELISM

Programmers couldn't simply ignore the additional cores that the CPU makers introduced every year. By 2015, INTEL had an 8-core desktop processor, i7-5960X [11], and 10-core workstation processors such as Xeon E7-8870 [14] in the market. Obviously, this multiple-core frenzy continued and will continue in the foreseeable future. Parallel programming turned from an exotic programming model in early 2000 to the only acceptable programming model as of 2015. The story doesn't stop at desktop computers either. On the mobile processor side, iPhones and Android phones all have two or four cores. Expect to see an ever-increasing number of cores in the mobile arena in the coming years.

So, what is a *thread* ? To answer this, let's take a look at the eight-core INTEL CPU i7-5960X [11] one more time. The INTEL archive says that, this is indeed an 8C/16T CPU. In other words, it has 8 cores, but, can execute 16 threads. You also hear parallel programming being *incorrectly* referred to as multi-core programming. The correct terminology is *multi-threaded* programming. This is because, when the CPU makers started designing multi-core architectures, they quickly realized that, it wasn't difficult to add the capability to execute two tasks within one core by sharing some of the core resources, such as cache memory.

ANALOGY 1.1: *Cores vs. Threads.*

Figure 1.1 shows two brothers, Fred and Jim, who are farmers that own two tractors. They drive from their farmhouse to where the coconut trees are every day. They harvest the coconuts and bring them back to their farm house. To harvest (process) the coconuts, they use the hammer inside their tractor. The *harvesting* process requires two separate consecutive tasks, each taking 30 seconds: Task 1) go from the tractor to the tree, bringing one coconut at a time, and Task 2) crack (process) them by using the hammer, and store them in the tractor. Fred alone can process one coconut per minute, and Jim can also process one coconut per minute. Combined, they can process two coconuts per minute.

One day, Fred's tractor breaks down. He leaves the tractor with the repair shop, forgetting that the coconut cracker is inside his tractor. It is too late by the time he gets to the farmhouse. But, they still have work to do. With only Jim's tractor, and a single coconut cracker inside it, can they still process two coconuts per minute ?

1.3 CORES VS. THREADS

Let's look at our Analogy 1.1, which is depicted in Fig. 1.1. If harvesting a coconut requires the completion of two consecutive tasks (we will call them *threads*): Thread 1) picking a coconut from the tree and bringing it back to the tractor in 30 seconds, and Thread 2) cracking (i.e., *processing*) that coconut using the hammer inside the tractor within 30 seconds, then, each coconut can be harvested in 60 seconds (one coconut per minute). If Jim and Fred each have their own tractors, they can simply harvest twice as many coconuts (two coconuts per minute), since during the harvesting of each coconut, they can share the road from the tractor to the tree, and they have their own hammer.

In this analogy, each tractor is a **core**, and harvesting one coconut is the **program execution** using one data element. Coconuts are **data elements**, and each person (Jim, Fred) is an **executing thread**, using the coconut cracker. Coconut cracker is the **execution unit**, like **ALU** within the core. This program consists of two dependent threads: You cannot execute Thread 2 before you execute Thread 1. The number of coconuts harvested is equivalent to the **program performance**. The higher the performance, the more money Jim and Fred make selling coconuts. Coconut tree is the **memory**, from which you get data elements (coconuts), so, the process of getting a coconut during Thread 1 is analogous to **reading data elements from the memory**.

1.3.1 More threads or more cores to parallelize ?

Now, let's see what happens if Fred's tractor breaks down. They used to be able to harvest two coconuts per minute, but, now, they only have one tractor and only one coconut cracker. They drive to the trees and park the tractor. To harvest each coconut, they have to execute Thread 1 (Th1) and Th2 consecutively. They both get out of the tractor and walk to the trees in 30 seconds, thereby completing Th1. They bring back the coconut they picked, and now, they have to crack their coconut. However, they cannot execute Th2 simultaneously, since there is only one coconut cracker. Fred has to wait for Jim to crack the coconut, and he cracks his after Fred is done using the cracker. This takes 30+30 more seconds, and they

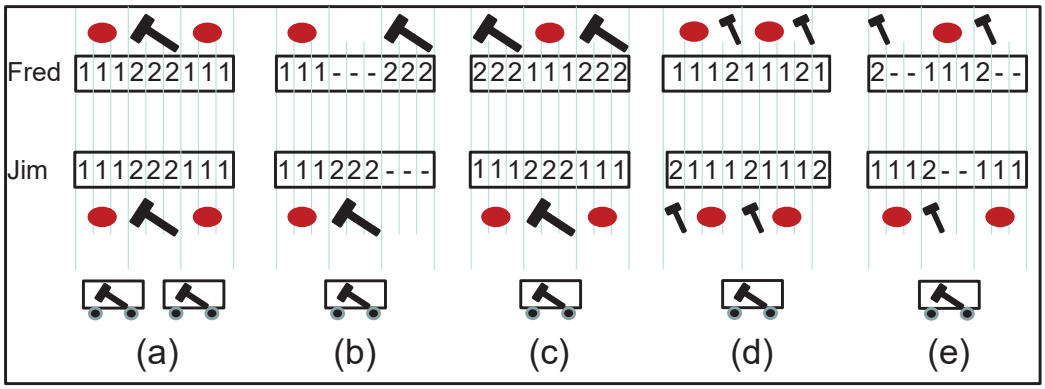


FIGURE 1.2 Simultaneously executing Thread 1 ("1") and Thread 2 ("2"). Accessing shared resources will cause a thread to wait ("-").

finish harvesting two coconuts in 90 seconds total. Although not as good as two coconuts per minute, they still have a performance improvement from 1 to 1.5 coconuts per minute.

After harvesting a few coconuts, Jim asks himself the question : "*Why do I have to wait for Fred to crack the coconut? When he is cracking the coconut, I can immediately walk to the tree, and get the next coconut. Since Th1 and Th2 take exactly the same amount of time, we never have to be in a situation where they are waiting for the cracker to be free. Exactly the time Fred is back from picking the next coconut, I will be done cracking my coconut and we will both be 100% busy.*" This genius idea brings them back to the 2 coconuts/minute speed without even needing an extra tractor. The big deal was that, Jim re-engineered the **program**, which is the sequence of the threads to execute, so, the threads are never caught in a situation where they are waiting for the *shared resources* inside the core, like the cracker inside the tractor. As we will see very shortly, a shared resource inside a core is an ALU, FPU, cache memory, and more ... For now, don't worry about these.

The two scenarios I described in this analogy are having two cores (2C), each executing a single thread (1T) vs. having a single core (1C) that is capable of executing two threads (2T). In the CPU world, they are called 2C/2T, vs. 1C/2T. In other words, there are two ways to give a program the capability to execute two simultaneous threads: 2C/2T (2 cores, which are capable of executing a single thread each - just like two separate tractors for Jim and Fred) or 1C/2T (a single core, capable of executing two threads - just like a single tractor shared by Jim and Fred). Although, from the programmers standpoint, both of them mean the ability to execute two threads, they are very different options from the hardware standpoint, and they require the programmer to be highly aware of the implications of the threads that share resources. Otherwise, the performance advantages of the extra threads could vanish. Just to remind again: our almighty INTEL i7-5960X [11] CPU is an 8C/16T, which has eight cores, each capable of executing two threads.

Three options are shown in Figure 1.2: (a) is the 2C/2T option with two separate cores. (b) is the 1C/2T option with bad programming, yielding only 1.5 coconuts per minute, and (c) is the sequence-corrected version, where the access to the cracker is never simultaneous, yielding 2 coconuts per minute.

1.3.2 Influence of core resource sharing

Being so proud of his discovery which brought their speed back to 2 coconuts per minute, Jim wants to continue inventing ways to use a single tractor to do more work. One day, he goes to Fred and says "*I bought this new automatic coconut cracker which cracks a coconut in 10 seconds.*" Extremely happy with this discovery, they hit the road and park the tractor next to the trees. This time they know that, they have to do some **planning**, before harvesting ...

Fred asks : "*if our Th1 takes 30 seconds, and Th2 takes 10 seconds, and the only task for which we are sharing resources is Th2 (cracker), how should we harvest the coconuts ?*" The answer is very clear to them: The only thing that matters is the sequence of execution of the threads (i.e., the design of the program), so that, they are never caught in a situation where they are executing Th2 together and needing the only cracker they have (i.e., shared core resources). To re-phrase, their program consists of two dependent threads: Th1 is 30 seconds, and does not require shared (memory) resources, since two people can walk to the trees simultaneously, and Th2 is 10 seconds and cannot be executed simultaneously, since it requires the shared (core) resource: the cracker. Since each coconut requires a 30+10=40 seconds of total execution time, the best they can hope for is : 40 seconds to harvest two coconuts, shown in Fig. 1.2d. This would happen if everybody executed Th1 and Th2 sequentially, without waiting for any shared resource. So, their speed will be an average of 3 coconuts per minute (i.e., average 20 seconds per coconut).

1.3.3 Influence of memory resource sharing

After harvesting 3 coconuts per minute using the new cracker, Jim and Fred come back the next day and see something terrible. The road from the tractor to the tree could only be used by one person today, since a heavy rain last night blocked half of the road. So, they plan again ... Now, they have two threads that each require resources that cannot be shared. Th1 (30 seconds — denoted as 30 s) can only be executed by one person, and Th2 (10 s) can only be executed by one person. Now what?

After contemplating multiple options, they realize that the *limiting factor* in their speed is Th1; the best they can hope for is harvesting each coconut in 30 s. When the Th1 could be executed together (shared memory access), each person could execute 10+30 s sequentially and both of them could continue without ever needing to access shared resources. But now, there is no way to sequence the threads to do so. The best they can hope for is to execute 10+30 s and wait for 20 s during which both need to access the memory. Their speed is back to 2 coconuts per minute on average, as depicted in Fig. 1.2e.

This heavy rain reduced their speed back to 2 coconuts per minute. Th2 no longer matters, since somebody could easily crack a coconut while the other is on the road to pick a coconut. Fred comes up with the idea that, they should bring the second (slower) hammer from the farmhouse to help. However, this would absolutely not help anything in this case, since the limiting factor to harvesting is Th1. This concept of a limiting factor by a resource is called *resource contention*. This example shows what happens when our access to memory is the limiting factor for our program speed. It simply doesn't matter how fast we can process the data (i.e., core execution speed). We will be limited by the speed at which we can *get* the data. Even if Fred had a cracker that could crack a coconut in 1 second, they would still be limited to 2 coconuts per second if there is a resource contention during memory access. In this book, we will start making a distinction between two different programs: ones that are *core intensive*, which do not depend so much on the memory access

FIGURE 1.3 Serial (single-threaded) program imflip.c flips a 640 × 480 dog picture (left) horizontally (middle) or vertically (right).

speed, and ones that are *memory intensive*, which are highly sensitive to the memory access speed, as I have just shown.

1.4 OUR FIRST SERIAL PROGRAM

Now that we understand parallel programming in the coconut world, it is time to apply this knowledge to real computer programming. I will start by introducing our first serial (i.e., single-threaded) program, and we will parallelize it next. Our first serial program imflip.c takes the dog picture in Fig. 1.3 (left) and flip it horizontally (middle) or vertically (right). For simplicity in explaining the program, we will use Bitmap (BMP) images and will write the result in BMP format too. This is an extremely easy to understand image format and will allow us to focus on the program itself. Do not worry about the details in this chapter. They will become clear soon. For now, just focus on high-level functionality.

The imflip.c source file can be compiled and executed from a Unix prompt as follows:

```
gcc imflip.c -o imflip

./imflip dogL.bmp dogh.bmp V
```

"H" is specified at the command line to flip the image *horizontally* (Fig. 1.3 middle), while "V" specifies a *vertical* flip (Fig. 1.3 right). You will get an output that looks like this (numbers might be different, based on the speed of your computer):

```
Input BMP File name : dogL.bmp (3200×2400)

Output BMP File name : dogh.bmp (3200×2400)

Total Execution time : 81.0233 ms (10.550 ns per pixel)
```

The CPU that this program is running on is so fast that I had to artificially expand the original 640×480 image dog.bmp to 3200×2400 dogL.bmp, so, it could run in an amount of time that can be measured; dogL.bmp is 5× bigger in each dimension, thereby making it 25× bigger than dog.bmp. To time the program, we have to record the clock at the beginning of image flipping and at the end.

1.4.1 Understanding data transfer speeds

It is very important to understand that the process of reading the image from the disk (whether it is an SSD or hard drive) should be excluded from the execution time. In other words, we read the image from the disk, and make sure that it is in memory (in our array),

and then measure only the amount of time that we spend flipping the image. Due to the drastic differences in the data transfer speeds of different hardware components, we need to analyze the amount of time spent in the disk, memory, and CPU separately.

In many of the parallel programs we will develop in this book, our focus is CPU time and memory access time, because we can influence them; disk access time (which we will call I/O time) typically saturates even with a single thread, thereby seeing almost no benefit from multi-threaded programming. Also make a mental note that this slow I/O speed will haunt us when we start GPU programming; since I/O is the slowest part of a computer and the data from the CPU to the GPU is transferred through the PCI express bus, which is a part of the I/O subsystem, we will have a challenge in feeding data to the GPU fast enough. But, nobody said that GPU programming was easy ! To give you an idea about the magnitudes of transfer speeds of different hardware components, let me now itemize them:

- A typical Network Interface Card (NIC) has a transfer speed of 1 Gbps (Giga-bits-per-second or billion-bits-per-second). These cards are called "Gigabit network cards" or "Gig NICs" colloquially. Note that 1 Gbps is only the amount of "raw data," which includes a significant amount of error correction coding and other synchronization signals. The amount of *actual* data that is transferred is less than half of that. Since my goal is to give the reader a *rough idea* for comparison, this details is not that important for us.

- A typical hard disk drive (HDD) can barely reach transfer speeds of 1–2 Gbps, even if connected to a SATA3 cable that has a peak 6 Gbps transfer speed. The mechanical read-write nature of the HDDs simply do not allow them to access the data that fast. The transfer speed isn't even the worse problem with an HDD, but the *seek time* is; it takes the mechanical head of an HDD some time to locate the data on the spinning metal cylinder, therefore forcing it to wait until the rotating head reaches the position, where the data resides. This could take milli-seconds (ms) if the data is distributed in an irregular fashion (i.e., fragmented). Therefore, HDDs could have transfer speeds that are far less than the peak speed of the SATA3 cable that they are connected to.

- A flash disk that is hooked up to a USB 2.0 port has a peak transfer speed of 480 Mbps (Mega-bits-per-second or million-bits-per-second). However, the USB 3.0 standard has a faster 5 Gbps transfer speed. The newer USB 3.1 can reach around 10 Gbps transfer rates. Since flash disks are built using flash memory, there is no seek time, as they are directly accessible by simply providing the data address.

- A typical Solid state disk (SSD) can be read from a SATA3 cable at speeds close to 4–5 Gbps. Therefore, an SSD is really the only device that can saturate a SATA3 cable, i.e., deliver data at its intended peak rate of 6 Gbps.

- Once the data is transferred from I/O (SDD, HDD, or flash disk) into the memory of the CPU, transfer speeds are drastically higher. Core i7 family all the way up to the sixth generation (i7-6xxx) and the higher-end Xeon CPUs use DDR2, DDR3, and DDR4 memory technologies and have memory-to-CPU transfer speeds of 20–60 GBps (Giga-Bytes-per-second). Notice that this speed is **Giga-Bytes**; a Byte is 8 bits, thereby translating to memory transfer speeds of 160–480 Gbps (Giga-bits-per-second) just to compare readily to the other slower devices.

- As we will see in Part II and beyond, transfer speeds of GPU internal memory subsystems can reach 100–1000 GBps. The new Pascal series GPUs, for example, have an internal memory transfer rate, which is close to the latter number. This translates

to 8000 Gbps, which is an order-of-magnitude faster than the CPU internal memory and three orders-of-magnitude faster than a flash disk, and almost four orders-of-magnitude faster than an HDD.

CODE 1.1: imflip.c main() {...}

The main() function of the imflip.c reads three command line parameters to determine the input and output BMP images and also the flip direction (horizontal or vertical). Operation is repeated multiple times (REPS) to improve the accuracy of the timing.

```
#define REPS  129
...
int main(int argc, char** argv)
{
    double timer;      unsigned int a;    clock_t start,stop;
    if(argc != 4){ printf("\n\nUsage: imflip [input][output][h/v]");
                   printf("\n\nExample: imflip square.bmp square_h.bmp h\n\n");
                   return 0;   }
    unsigned char** data = ReadBMP(argv[1]);
    start = clock();         // Start timing the code without the I/O part
    switch (argv[3][0]){
      case 'v' :
      case 'V' : for(a=0; a<REPS; a++) data = FlipImageV(data); break;
      case 'h' :
      case 'H' : for(a=0; a<REPS; a++) data = FlipImageH(data); break;
      default : printf("\nINVALID OPTION\n"); return 0;
    }
    stop = clock();
    timer = 1000*((double)(stop-start))/(double)CLOCKS_PER_SEC/(double)REPS;
    //merge with header and write to file
    WriteBMP(data, argv[2]);
    // free() the allocated memory for the image
    for(int i = 0; i < ip.Vpixels; i++) { free(data[i]); }
    free(data);
    printf("\n\nTotal execution time: %9.4f ms",timer);
    printf(" (%7.3f ns/pixel)\n", 1000000*timer/(double)(ip.Hpixels*ip.Vpixels));
}
```

1.4.2 The main() function in imflip.c

Our program, shown in Code 1.1, reads a few command line parameters and flips an input image either vertically or horizontally, as specified by the command line. The command line arguments are placed by C into the `argv` array.

The clock() function reports time in ms. intervals; this crude reporting resolution is improved by repeating the same operations an odd number of times (e.g., 129), specified in the "`#define REPS 129`" line. This number can be changed based on your system.

The ReadBMP() function reads the source image from disk and WriteBMP() writes the processed (i.e., flipped) image back to disk. The amount of time spent reading and writing the image from/to the disk is defined as *I/O time* and we will exclude it from the *processing*

time. This is why I placed the "`start = clock()`" and "`stop = clock()`" lines between the actual code that flips the image that is in memory and intentionally excluded the I/O time.

Before reporting the elapsed time, the imflip.c program de-allocates all of the memory that was allocated by ReadBMP() using a bunch of free() functions to avoid memory leaks.

CODE 1.2: imflip.c ... FlipImageV() {...}

To flip the rows vertically, each pixel is read and replaced with the corresponding pixel of the mirroring row.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "ImageStuff.h"
#define REPS  129
struct ImgProp ip;

unsigned char** FlipImageV(unsigned char** img)
{
    struct Pixel pix; //temp swap pixel
    int row, col;
    for(col=0; col<ip.Hbytes; col+=3){ //go through the columns
        row = 0;
        while(row<ip.Vpixels/2){    // go through the rows
            pix.B = img[row][col];          pix.G = img[row][col+1];
            pix.R = img[row][col+2];

            img[row][col]  = img[ip.Vpixels-(row+1)][col];
            img[row][col+1] = img[ip.Vpixels-(row+1)][col+1];
            img[row][col+2] = img[ip.Vpixels-(row+1)][col+2];

            img[ip.Vpixels-(row+1)][col] = pix.B;
            img[ip.Vpixels-(row+1)][col+1] = pix.G;
            img[ip.Vpixels-(row+1)][col+2] = pix.R;

            row++;
        }
    }
    return img;
}
```

1.4.3 Flipping rows vertically: FlipImageV()

The FlipImageV() in Code 1.2 goes through every column and swaps each vertical pixel with its mirroring vertical pixel for that column. The Bitmap (BMP) image functions are placed in another file named ImageStuff.c and ImageStuff.h is the associated header file. They will be explained in detail in the next chapter. Each pixel of the image is stored as type "`struct Pixel`," which contains the R, G, and B color components of that pixel in `unsigned char` type; since `unsigned char` takes up one byte, each image pixel requires 3 bytes to store.

The ReadBMP() places the image width and height in two variables `ip.Hpixels` and `ip.Vpixels`, respectively. The number of bytes that we need to store each row of the image is placed in `ip.Hbytes`. FlipImageV() function has two loops: The outer loop goes through all `ip.Hbytes` of the image and for every column, it swaps the corresponding vertical mirror pixels one at a time in the inner loop.

CODE 1.3: imflip.c FlipImageH() {...}

To flip the columns horizontally, each pixel is read and replaced with the corresponding pixel of the mirroring column.

```
unsigned char** FlipImageH(unsigned char** img)
{
    struct Pixel pix; //temp swap pixel
    int row, col;

    //horizontal flip
    for(row=0; row<ip.Vpixels; row++){  // go through the rows
        col = 0;
        while(col<(ip.Hpixels*3)/2){     // go through the columns
            pix.B = img[row][col];
            pix.G = img[row][col+1];
            pix.R = img[row][col+2];

            img[row][col]  = img[row][ip.Hpixels*3-(col+3)];
            img[row][col+1] = img[row][ip.Hpixels*3-(col+2)];
            img[row][col+2] = img[row][ip.Hpixels*3-(col+1)];

            img[row][ip.Hpixels*3-(col+3)] = pix.B;
            img[row][ip.Hpixels*3-(col+2)] = pix.G;
            img[row][ip.Hpixels*3-(col+1)] = pix.R;

            col+=3;
        }
    }
    return img;
}
```

1.4.4 Flipping columns horizontally: FlipImageH()

The FlipImageH() function of imflip.c flips the image horizontally, as shown in Code 1.3. This function is identical to the vertical counterpart, except the inner loops are opposite. Each swap uses the temporary pixel variable `pix`, which is "`struct` Pixel" type.

Since the pixels are stored as 3 bytes in a row using the RGB, RGB, RGB, ... format, accessing consecutive pixels requires reading 3 bytes at a time. This will be detailed in the next section. For now, all we need to know is that the following lines

```
        pix.B = img[row][col];
        pix.G = img[row][col+1];
        pix.R = img[row][col+2];
```

CHAPTER 2

Developing Our First Parallel CPU Program

This chapter is dedicated to understanding our very first CPU parallel program, imflipP.c. Notice the 'P' at the end of the file name that indicates *parallel*. For the CPU parallel programs, the development platform makes no difference. In this chapter, I will slowly start introducing the concepts that are the backbone of a parallel program, and these concepts will be readily applicable to GPU programming when we start developing GPU programs in Part II. As you might have noticed, I never say *GPU Parallel Programming*, but, rather, *GPU Programming*. This is much like there is no reason to say *a car with wheels*; it suffices to say *a car*. In other words, there is no GPU serial programming, which would mean using one GPU thread out of the available 100,000s ! So, GPU programming by definition implies GPU parallel programming.

2.1 OUR FIRST PARALLEL PROGRAM

It is time to write our first parallel program imflipP.c, which is the parallel version of our serial imflip.c program, introduced in Chapter 1.4. To parallelize imflip.c, we will simply have the main() function create multiple threads and let them do a portion of the work and exit. If, for example, in the simplest case, we are trying to run the two-threaded version of our program, the main() will create two threads, let them each do half the work, join the threads, and exit. In this scenario, main() is nothing but the *organizer* of events. It is not doing actual heavy-lifting.

To do what we just described, main() needs to be able to create, terminate, and organize threads and assign tasks to threads. The functions that allow it to perform such tasks are a part of the *Pthreads* library. *Pthreads* only work in a POSIX-compliant Operating System. Ironically, Windows is not POSIX compliant! However, the Cygwin64 allows Pthreads code to run in Windows by performing some sort of API-by-API translation between POSIX and Windows. This is why everything we describe here will work in Windows, and hence the reason for me to use Cygwin64 in case you have a Windows PC. Here are a few functions that we will use from the Pthreads library:

1. pthread_create() allows you to create a thread,
2. pthread_join() allows you to join any given thread into the thread that originally created it. Think of the "join" process as "uncreating" threads, or like the top thread "swallowing" the thread if just created,
3. pthread_attr() allows you to initialize attributes for threads,

4. pthread_attr_setdetachstate() allows you to set attributes for the threads you just initialized.

2.1.1 The main() function in imflipP.c

Our serial program imflip.c, shown in Code 1.1, read a few command line parameters and flipped an input image either vertically or horizontally, as specified by the user's command line entry. The same flip operation was repeated an odd number of times (e.g., 129) to improve the accuracy of the system time read by clock().

Code 2.1 and Code 2.2 show the same main() function in imflipP.c with one exception: Code 2.1 shows main() {..., which means the "first part" of main(), which is further emphasized by the ... at the end of this listing. This part is for command line parsing and other routine work. In Code 2.2, an opposite main() ...} notation is used along with the ... in the beginning of the listing, indicating the "second part" of the main() function, which is for launching threads and assigning tasks to threads.

To improve readability, I might repeat some of the code in both parts, such as the time-stamping with gettimeofday() and the image reading with our own function ReadBMP() that will be detailed very soon. This will allow the readers to clearly follow the beginning and connection points within the two separate parts. As you might have noticed already, whenever a function is entirely listed, the "func() {...}" notation is used. When a function and some surrounding code is listed, "... func() {...}" notation will be used, denoting "some common code ... followed by a complete listing of func()."

Here is the part of main() which parses command arguments, given in the `argv[]` array (a total of `argc` of them). It issues errors if the user enters an unaccepted number of arguments. It saves the flip direction that the user requested in a variable called `Flip` for further use. The global variable `NumThreads` is also determined based on user input and is used later in the functions that actually perform the flip.

```
int main(int argc, char** argv)
{
   ...
   switch (argc){
      case 3: NumThreads=1;         Flip='V';               break;
      case 4: NumThreads=1;         Flip=toupper(argv[3][0]); break;
      case 5: NumThreads=atoi(argv[4]); Flip=toupper(argv[3][0]); break;
      default:printf("Usage: imflipP input output [v/h] [threads]");
            printf("Example: imflipP infile.bmp out.bmp h 8\n\n");
            return 0;
   }
   if((Flip != 'V') && (Flip != 'H')) {
      printf("Invalid option '%c' ... Exiting...\n",Flip);
      exit(EXIT_FAILURE);
   }
   if((NumThreads<1) || (NumThreads>MAXTHREADS)){
      printf("Threads must be in [1..%u]... Exiting...\n",MAXTHREADS);
      exit(EXIT_FAILURE);
   }else{
   ...
```

2.1.2 Timing the Execution

When we have more than one thread executing, we want to be able to quantify the speed-up. We used the clock() function in our serial code, which was included in the time.h header file and was only millisecond-accurate.

The gettimeofday() function we will use in imflipP.c will get us down to μs-accuracy. gettimeofday() requires us to `#include` the sys/time.h header file and provides the time in two parts of a `struct`: one for seconds through the `.tv_sec` member and one for the micro-seconds through the `.tv_usec` member. Both of these members are `int` types and are combined to produce a `double` time value before being displayed.

An important note here is that, the accuracy of the timing does not depend on the C function itself, but, rather, the *hardware*. If your computer's OS or hardware cannot provide a μs-accurate timestamp, gettimeofday() will provide only as accurate of a result as it can obtain from the OS (which itself gets its value from a hardware clock unit). For example, Cygwin64 does not achieve μs-accuracy even with the gettimeofday() function due to its reliance on the underlying Windows APIs.

```
#include <sys/time.h>
...
struct timeval     t;
double             StartTime, EndTime;
double             TimeElapsed;
   ...
   gettimeofday(&t, NULL);
   StartTime = (double)t.tv_sec*1000000.0 + ((double)t.tv_usec);
   // work is done here : thread creation, task/data assignment, join
   ...
   gettimeofday(&t, NULL);
   EndTime = (double)t.tv_sec*1000000.0 + ((double)t.tv_usec);
   TimeElapsed=(EndTime-StartTime)/1000.00;
   TimeElapsed/=(double)REPS;
   ...
   printf("\n\nTotal execution time: %9.4f ms ...",TimeElapsed,...
```

2.1.3 Split Code Listing for main() in imflipP.c

I am intentionally staying away from providing one long listing for the main() function in a single Code fragment. This is because, as you can see from this first example, Code 2.1 and Code 2.2 provide listings for entirely different functionality: Code 2.1 provides a listing for flat-out "boring" functionality for getting command line arguments, parsing them, and warning the user. On the other hand, Code 2.2 is where the "cool action" of creating and joining threads happens. Most of the time, I will arrange my code to allow such partitioning and try to put a lot more emphasis on the important part of the code.

TABLE 2.1 Serial and multi-threaded execution time of imflipP.c, both for vertical flip and horizontal flip, on an i7-960 (4C/8T) CPU.

#Threads	Command line	Run time (ms)
Serial	imflipP dogL.bmp dogV.bmp v	131
2	imflipP dogL.bmp dogV2.bmp v 2	70
3	imflipP dogL.bmp dogV3.bmp v 3	46
4	imflipP dogL.bmp dogV4.bmp v 4	67
5	imflipP dogL.bmp dogV5.bmp v 5	55
6	imflipP dogL.bmp dogV6.bmp v 6	51
8	imflipP dogL.bmp dogV8.bmp v 8	52
9	imflipP dogL.bmp dogV9.bmp v 9	47
10	imflipP dogL.bmp dogV10.bmp v 10	51
12	imflipP dogL.bmp dogV10.bmp v 12	44
Serial	imflipP dogL.bmp dogH.bmp h	81
2	imflipP dogL.bmp dogH2.bmp h 2	41
3	imflipP dogL.bmp dogH3.bmp h 3	28
4	imflipP dogL.bmp dogH4.bmp h 4	41
5	imflipP dogL.bmp dogH5.bmp h 5	33
6	imflipP dogL.bmp dogH6.bmp h 6	28
8	imflipP dogL.bmp dogH8.bmp h 8	32
9	imflipP dogL.bmp dogH9.bmp h 9	30
10	imflipP dogL.bmp dogH10.bmp h 10	33
12	imflipP dogL.bmp dogH7.bmp h 12	29

For each row that a thread is responsible for, each pixel's 3-byte RGB values are swapped with its horizontal mirror. This swap starts at `col`=[0...2] which holds the RGB values of pixel 0, and continues until the last RGB (3 byte) value has been swapped. For a 640×480 image, since `Hbytes`=1920, and there is no wasted byte, the last pixel (i.e., pixel 639) is at `col`=[1917...1919].

2.4 TESTING/TIMING THE MULTI-THREADED CODE

Now that we know how the imflipP program works in detail, it is time to test it. The program command line syntax is determined via the parsing portion of main() as shown in Code 2.1. To run imflipP, the general command line syntax is:

imflipP InputfileName OutputfileName [v/h/V/H] [1-128]

where InputFileName and OutputFileName are the BMP files to read and write, respectively. The optional command line argument **[v/h/V/H]** is to specify the flip direction ('V' is the default). The next optional argument is the number of threads and can be specified between 1 and `MAXTHREADS` (128), with a default value of 1 (serial).

Table 2.1 shows the run times of the same program by using 1 through 10 threads on an Intel i7-960 CPU that has 4C/8T (4 cores, 8 threads). Results all the way to 10 threads is reported, not that we expect an improvement beyond 8 threads, but, as a *sanity check*. These kind of checks are useful to quickly discover a potentially hidden bug. The functionality is also confirmed by looking at the picture, and checking the file size, and running a comparison program that checks two binary files, the the Unix diff command.

So, what do these results tell us ? First of all, in both the vertical and horizontal flip case, it is clear that, using more than a single thread helps. So, our efforts to parallelize the program wasn't for nothing. However, the troubling news is that, beyond 3 threads, there seems to be no performance improvement at all, in both the vertical and horizontal case. For ≥ 4 threads, you can simply regard the data as *noise* !

What Table 2.1 clearly shows is that, *multi-threading helps up to 3 threads*. Of course, this is not a generalized statement. This statement strictly applies to my i7-960 test CPU (4C/8T) and the code I have shown in Code 2.7 and Code 2.8, which are the heart of the imflipP.c code. By this time, you should have a thousand questions in your mind. Here are some of them:

- Would the results be different with a less powerful CPU like a 2C/2T ?
- How about a more powerful CPU, such as a 6C/12T ?
- In Table 2.1, considering the fact that we tested it on a 4C/8T CPU, shouldn't we have gotten better results up to 8 threads ? Or, at least 6 or something ? Why does the performance collapse beyond 3 threads ?
- What if we processed a smaller 640×480 image, such as dog.bmp, instead of the giant 3200×2400 image dog.bmp? Would the *inflection point* for performance be at a different thread count ?
- Or, for a smaller image, would there even be an inflection point ?
- Why is the horizontal flip faster, considering that, vertical is also processing the same number of pixels; 3200×2400 ?
- ...

The list goes on ... Don't lose sleep over Table 2.1. For this chapter, we have achieved our goal. We know how to write multi-threaded programs now, and we are getting *some* speed-up. This is more than enough so early in our parallel programming journey. I can guarantee that, you will ask another 1000 questions to yourself about why this program isn't as fast as we hoped. I can also guarantee that, you will **NOT** ask some of the key questions that actually contribute to this lackluster performance. The answer to these questions deserves an entire chapter, and this is what I will do. In Chapter 3, I will answer all of the above questions and more of the ones you are not asking. For now, I invite you to think about what you might **NOT** be asking ...

CHAPTER 3

Improving Our First Parallel Program

We parallelized our first serial program imflip.c and developed its parallel version imflipP.c in Chapter 2. The parallel version achieved a reasonable speed-up using pthreads, as shown in Table 2.1. Using multiple threads reduced the execution time from 131 ms (serial version) down to 70 ms, and 46 ms when we launched two and three threads, respectively, on an i7-960 CPU with 4C/8T. Introducing more threads (i.e., ≥ 4) didn't help. In this chapter, we want to understand the factors that contributed to the performance numbers that were reported in Table 2.1. We might not be able to improve them, but, we have to be able to explain why we cannot improve them. We do not want to achieve good performance by *luck* !

3.1 EFFECT OF THE "PROGRAMMER" ON PERFORMANCE

Understanding the *hardware* and the *compiler* helps a programmer write good code. Over many years, CPU architects and compiler designers kept improving their CPU architecture and the optimization capabilities of compilers. A lot of these efforts helped ease the burden on the software developer, so, (s)he can write code without worrying too much about low-level hardware details. However, as we will see in this chapter, understanding the underlying hardware and utilizing the hardware efficiently might allow a programmer to develop 10x higher performance code in some cases.

Not only this statement is true for CPUs, the potential GPU performance improvement is even more emphasized when the hardware is utilized efficiently, since a lot of the impressive GPU performance comes from shifting the performance-improvement responsibility from the *hardware* to the *programmer*. This chapter explains the interaction among all parties that contribute to the performance of a program. They are the *programmer*, *compiler*, *OS*, and *hardware* (and, to a certain degree, the *user*).

- **Programmer** is the ultimate intelligence and should understand the capabilities of the other pieces. No software or hardware can match what the programmer can do, since the programmer brings in the most valuable asset to the game : *the logic*. Good programming logic requires a complete understanding of all pieces of the puzzle.

- **Compiler** is a large piece of code that is packed with *routine* functionality for two things : 1) compilation, and 2) optimization. (1) is the compiler's *job* and (2) is the additional work the compiler has to do at *compile time* to potentially optimize the inefficient code that the programmer wrote. So, the compiler is the "organizer" at *compile time*. At compile time, time is *frozen*, meaning that, the compiler could contemplate many alternative scenarios that can happen at *run time* and produce

the best code for run time. When we run the program, the clock starts ticking. One thing the compiler cannot know is the *data*, which could completely change the flow of the program. The data can only be known at runtime, when the OS and CPU are in action.

- The Operating System (**OS**) is the software that is the "boss" or the "manager" of the hardware at *run time*. Its job is to allocate and map the hardware resources efficiently at *run time*. Hardware resources include the virtual CPUs (i.e., threads), memory, hard disk, flash drives (via Universal Serial Bus (USB) ports), network cards, keyboard, monitor, GPU (to a certain degree) and more. A good OS knows its resources and how to map them very well. Why is this important ? Because, the resources themselves (e.g., CPU) have no idea about what to do. They simply follow orders. The OS is the general, and the threads are the soldiers.

- **Hardware** is the CPU+memory+peripherals. OS takes the binary code that the compiler produced and assigns them to virtual cores at run time. Virtual cores execute them as fast as possible at *run time*. OS also facilitates the data movement between the CPU and the memory, disk, keyboard, network card, etc.

- **User** is the final piece of the puzzle: Understanding the user is also important in writing good code. The user of a program isn't a programmer, yet, the programmer has to appeal to the user and has to *communicate* with him/her. This is not an easy task !

In this book, the major focus will be on *hardware*, particularly the CPU and memory (and, later in Part II, GPU and memory). Understanding the hardware holds the key to developing high-performance code, whether for CPU or GPU. In this chapter, we will discover the truth about whether it is possible to speed-up our first parallel program, imflipP.c. If we can, *how ?* The only problem is: we don't know which part of the hardware we can use more efficiently for performance improvement. So, we will look at everything.

3.2 EFFECT OF THE "CPU" ON PERFORMANCE

In Section 2.3.3, I explained the sequence of events that takes place when we launch multi-threaded code. In Section 2.4, I also listed numerous questions you might be asking yourself to explain Table 2.1. Let us answer the very first and the most obvious family of questions:

- how would the results differ on different CPUs ?

- is it the *speed* of the CPU, or the *number of cores*, *number of threads* ?

- anything else related to the CPU ? like *cache memory ?*

Perhaps, the most *fun* way to answer this question is the go ahead and run the same program on many different CPUs. Once the results are in place, we can try to make sense out of them. The more variety of CPUs we test this code on, the better idea we might get. I will go ahead and run this program on 6 different CPUs shown in Table 3.1.

Table 3.1 lists important CPU parameters, such as the number of cores and threads (C/T), per-core L1$ and L2$ cache memory sizes, denoted as L1$/C and L2$/C, and the shared L3$ cache memory size (shared by all 4, 6, or 8 cores). Table 3.1 also lists the amount of memory and *memory bandwidth* (BW) in Giga-Bytes-per-second (GB/s) for each computer that a given column indicates. In this section, we focus on CPU's role on

TABLE 3.1 Different CPUs used in testing the imflipP.c program.

Feature	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6
Name	i5-4200M	i7-960	i7-4770K	i7-3820	i7-5930K	E5-2650
C/T	2C/4T	4C/8T	4C/8T	4C/8T	6C/12T	8C/16T
Speed:GHz	2.5-3.1	3.2-3.46	3.5-3.9	3.6-3.8	3.5-3.7	2.0-2.8
L1$/C	64KB	64KB	64KB	64KB	64KB	64KB
L2$/C	256KB	256KB	256KB	256KB	256KB	256KB
shared L3$	3MB	8MB	8MB	10MB	15MB	20MB
Memory	8GB	12GB	32GB	32GB	64GB	16GB
BW:GB/s	25.6	25.6	25.6	51.2	68	51.2

performance, but the role of the memory in determining the performance will be explained thoroughly in this book. We will also look at the operation of memory in this chapter. For now, just in case the performance numbers had anything to do with the memory instead of the CPU, they are also listed in Table 3.1.

In this section, we have no intention to make an intelligent assessment on how the performance results could differ among these 6 CPUs. We simply want to watch the CPU horse race and have fun ! As we see the numbers, we will develop theories about what could be the most responsible party in determining the overall performance of our program. In other words, we are looking at things from long distance at this point. We will dive deep into details later, but, the experimental data we gather first will help us develop methods to improve program performance later.

3.2.1 In-Order vs. Out-Of-Order Cores

Aside from *how many* cores a CPU has, there is another consideration that relates to the cores; almost every CPU manufacturer started manufacturing **in order** cores and upgraded their design to **out of order** in their more advanced family of offerings. Abbreviations inO and OoO will be used going forward. For example, MIPS R2000 was an inO CPU, while the more advanced R10000 was OoO. Similarly, Intel 8086, 80286, 80386 and the newer Atom CPUs are inO, whereas Core i3, i5, and i7, as well as every Xeon is OoO.

The difference between inO andOoO is the way the CPU is capable of executing a given set of instructions; while an inO CPU can only execute the instructions in precisely the order that is listed in the binary code, an OoO CPU executes them in the order of *operand availability*. In other words, an OoO CPU can find a lot more work to do in the later list of instructions, whereas an inO CPU simply sits idle if the next instruction in the order of all given instructions do not have data available, perhaps because the memory controller did not yet read the necessary data from the memory.

This allows an OoO CPU to execute instructions a lot faster due to the ability to avoid getting stuck when the next instruction cannot be immediately executed until the operands are available. However, this luxury comes at a price: an inO CPU takes up a lot less chip area, therefore allowing the manufacturer to fit a lot more inO cores in the same Integrated circuit chip. Because of this very reason, each inO core might actually be clocked a little faster, since they are *simpler*. Time for an analogy ...

ANALOGY 3.1: *In order vs. Out of Order Execution.*

Cocotown had a competition in which two teams of farmer families had to make coconut pudding. These were the instructions provided to them: 1) crack the coconut with the automated cracker machine, 2) grind the coconuts that come out of the cracker machine using the grinder machine, 3) boil milk, and 4) put cocoa in milk and boil more, 5) put the ground coconuts in the cocoa-mixed-milk and boil more.

Each step took 10 minutes. Team 1 finished their pudding in 50 minutes, while Team 2 shocked everyone by finishing in 30 minutes. Their secret became obvious after the competition: they started cracking the coconut (step 1) and boiling the milk (step 3) at the same time. These two tasks did not depend on each other and could be started at the same time. Within 10 minutes, both of them were done and the coconuts could be placed in the grinder (step 2), while, in parallel, cocoa is mixed with the milk and boiled (step 4). So, in 20 minutes, they were done with steps 1–4.

Unfortunately, step 5 had to wait for steps 1–4 to be done, making their total execution time 30 minutes. So, the secret of Team 2 was to execute the tasks *out of order*, rather than *in* the *order* they were specified. In other words, they could start executing any step if it did not depend on the results of a previous step.

Analogy 3.1 emphasizes the performance advantage of OoO execution; an OoO core can execute independent *dependence-chains* (i.e., chains of CPU instructions that do not have dependent results) in parallel, without having to wait for the next instruction to finish, achieving a healthy speed-up. But, there are other trade-offs when a CPU is designed using one of the two paradigms. One wonders which one is a better design idea: 1) more cores that are inO, or 2) fewer cores that are OoO ? What if we took the idea to the extreme and placed, say, 60 cores in a CPU that are all inO ? Would this work faster than a CPU that has 8 OoO cores ? The answer is not as easy as just picking one of them.

Here are the facts related to inO vs OoO CPUs:

- Since both design ideas are valid, there is a real inO CPU design like this, called *Xeon Phi*, manufactured by Intel. One model, Xeon Phi 5110P, has 60 inO cores and 4 threads in each core, making it capable of executing 240 threads. It is considered a Many Integrated Core (MIC) rather than a CPU; each core works at a very low speed like 1 GHz, but it gets its computational advantage from the sheer number of cores and threads. Since inO cores consume much less power, a 60C/240T Xeon Phi power consumption is only slightly higher than a comparable 6C/12T Core i7 CPU. I will be providing execution times on Xeon 5110P shortly.

- An inO CPU would only benefit a restricted set of applications; not every application can take advantage of so many cores or threads. In most applications, we get diminishing returns beyond a certain number of cores or threads. Generally, image and signal processing applications are perfect for inO CPUs or MICs. Scientific high performance processing applications are also typically a good candidate for inO CPUs.

- Another advantage of inO cores is their low power consumption. Since each core is much simpler, it does not consume as much power as a comparable OoO core. This is why most of today's Netbooks incorporate Intel Atom CPUs, which have inO cores. An Atom CPU consumes only 2–10 Watts. The Xeon Phi MIC is basically 60 Atom cores, with 4 threads/core, stuffed into a chip.

the sequence to make every thread do useful work. However, in the GPU world, you will be dealing with 1000s of threads. Teaching how to think in such an absurdly parallel world should start by learning how to sequence two threads first ! This is the reason why the CPU environment was perfect to warm up to parallelism, and is the philosophy of this book. By the time you finish Part I of this book, you will not only learn CPU parallelism, but will be totally ready to take on the massive parallelism that the GPUs bring you in Part II.

If you are still not convinced, let me mention this to you: GPUs actually support 100000s of threads, not just 1000s ! Convinced yet ? A corporation like IBM with 100000s of employees can run as well as a corporation with 1 or 2 employees, and, yet, IBM is able to harvest the manpower of all of its employees. But, it takes extreme discipline and a systematic approach. This is what the GPU programming is all about. If you cannot wait until we reach Part II to learn GPU programming, you can peak at it now; but, unless you understand the concepts introduced in Part I, something will always be missing.

GPUs are here to give us shear computational power. A GPU program that works 10× faster than a comparable CPU program is better than one that works only 5× faster. If somebody could come in and re-write the same GPU program to be 20× faster, that person is the king (or queen). There is no point in writing a GPU program unless you are targeting *speed*. There are three things that matter in a GPU program : *speed*, *speed*, and *speed* ! So, the goal of this book is to get you to be a GPU programmer that writes super-fast GPU code. This doesn't happen unless we systematically learn every important concept, so, in the middle of a program, when we encounter some weird bottleneck, we can explain it and remove the bottleneck. Otherwise, if you are going to write slow GPU code, you might as well spend your time learning much better CPU multi-threading techniques, since there is no point in using a GPU unless your code gets every little bit of extra speed out of it. This is the reason why we will time our code throughout the entire book and will find ways to make our GPU code faster.

3.11 CHAPTER SUMMARY

We now have an idea about how to write efficient multi-threaded code. Where do we go from here ? First, we need to be able to quantify everything we discussed in this chapter: what is memory bandwidth ? how do the cores really operate ? how do the cores get data from the memory ? how do the threads share the data ? Without answering these, we will only be *guessing* why we got any speedup.

It is time to quantify everything and get a 100% precise understanding of the architecture. This is what we will do in Chapter 4. Again, everything we learn will be readily applicable to the GPU world, although there will be distinct differences, which I will point out as they come up.

CHAPTER 4

Understanding the Cores and Memory

When we say *hardware architecture*, what are we talking about ?
The answer is : *everything physical*. What does "everything physical" include ? CPU, memory, I/O controller, PCI express bus, DMA controller, hard disk controller, hard disk(s), SSDs, CPU chipsets, USB ports, network cards, CD-ROM controller, DVDRW, I can go on for a while ... The big question is : which one of these do we *care about* as a programmer ?

The answer is : CPU and memory, and more specifically, cores inside the CPU and memory. Especially if you are writing high performance programs (like the ones in this book), $> 99\%$ of your program performance will be determined by these two things. Since the purpose of this book is high-performance programming, let's dedicate this chapter to understanding the cores and memory.

4.1 ONCE UPON A TIME ... INTEL ...

Once upon a time, in the early 1970's, a tiny Silicon Valley company named INTEL, employing about 150 people at the time, designed a programmable chip in the beginning days of the concept of a "microprocessor," or "CPU," a digital device that can execute a *program* stored in *memory*. Every CPU had the capability to *address* a certain amount of memory, primarily determined at *design time* by the CPU designers, based on technological and business-related constraints.

The *program* (aka, *CPU instructions*), and the *data* (that was fed into the program) had to be stored somewhere. INTEL designed the 4004 processor to have 12 address bits, capable of *addressing* 4096 Bytes (i.e., 2^{12}). Each piece of information was stored in 8-bit units (a Byte). So, all together, 4004 could execute a program that was, say, 1 KB and could work on data that was in the remaining 3 KB. Or, may be, some customers only needed to store 1 KB program and 1 KB data: so, they would have to buy the memory chips to attach to the 4004 somewhere else.

Although INTEL designed support chips, such as an I/O controller and memory controller to allow their customers to interface the 4004 CPU to the memory chips they bought somewhere else, they did not particularly focus on making memory chips themselves. This might look a little counter-intuitive at first, but, from a business standpoint, it makes a lot of sense. At the time of the release of the 4004, one needed at least 6, 7 other types of interface chips to properly interface other important devices to the 4004 CPU. Out of those 6, 7 different chips, one was very special : the *memory* chips.

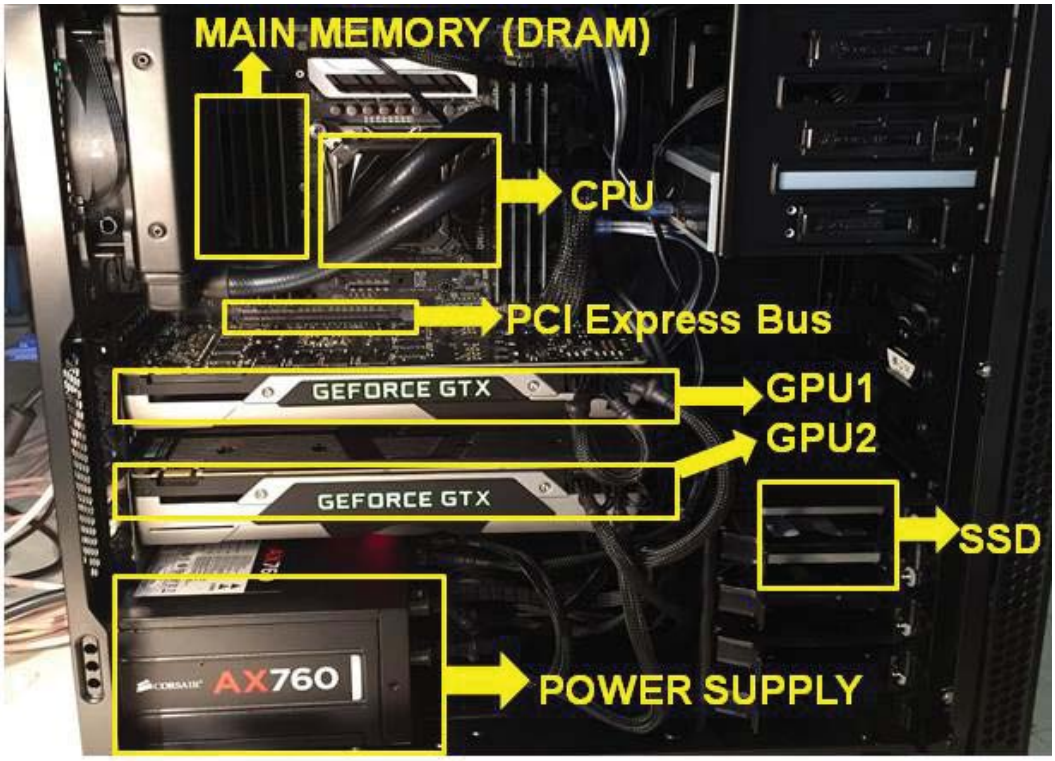


FIGURE 4.1 Inside a computer containing an i7-5930K CPU [10] (CPU5 in Table 3.1), and 64GB of DDR4 memory. This PC has a GTX Titan Z GPU that will be used to test a lot of the programs in Part II.

4.2 CPU AND MEMORY MANUFACTURERS

Much like 30–40 years ago, INTEL still doesn't make the memory chips as of year 2017, at the time of the projected publication of this book. The players in the memory manufacturing world are Kingston Technology, Samsung, Crucial Technology, to name a few. So, the trend in the 4004 days four decades ago never changed. Memory manufacturers are different from CPU manufacturers, although, a lot of the CPU manufacturers make their own support chips (chipsets). Figure 4.1 shows the inside of the PC containing the CPU5 in our Table 3.1. This CPU is an i7-5930K [10], made by INTEL. However, the memory chips in this computer (top left of Fig. 4.1) are made by a completely different manufacturer: Kingston Technology Corp. No big surprise for the manufacturer of the GPU: Nvidia Corp ! The SSD (solid state disk) is manufactured by Crucial Technology. Power supply is Corsair. Ironically, the CPU liquid cooler in Fig. 4.1 is not made by INTEL. It is made by the same company that the chassis and power supply are made by (Corsair).

Support chips (which were later called *chipsets*) are made out of logic gates, AND, OR, XOR gates etc. Their primary building blocks are Metal Oxide Semiconductor (MOS) transistors. For example, the X99 chipset in Fig. 4.1 was made by INTEL (not marked in the figure). This chip also controls the PCI Express bus that is connecting to the GPU to the CPU (marked in Fig. 4.1), as well as the SATA3 bus that is connecting to the SSD.

4.3 DYNAMIC (DRAM) VS. STATIC (SRAM) MEMORY

While CPU manufacturers have all the interest in the world to design and manufacture their chipsets, why are the memory chips so special ? Why do the CPU manufacturers have no interest in making memory chips ? To answer this, first, let's look at different types of memory:

4.3.1 Static Random Access Memory (SRAM)

This type of memory is still made with MOS transistors, something that is already the building block of the CPUs and their chipsets. It is easy for the CPU manufacturers to incorporate this type of memory into their CPU design, since it is made out of the same material. About 10 years after the introduction of the first CPUs, CPU designers introduced a type of SRAM that could be built right into the CPU. They called it *cache memory* that was able to buffer a very small portion of the main memory. Since most computer programs require access to very small portions of data in a repeated fashion, the effective speedup by the introduction of the cache memory was significant, although only very small amounts of this type of memory was possible to incorporate into the CPU.

4.3.2 Dynamic Random Access Memory (DRAM)

This type of memory was the only option to manufacture huge amounts of memory. For example, as of today, only 8–30 MB of cache memory can be built into the CPU, while a computer with a 32 GB of main memory (DRAM) is mainstream (a stunning 1000× difference). To be able to build DRAMs, a completely different technology has to be employed: While the building block of chipsets and CPUs is primarily MOS transistors, the building block of DRAM is extremely small capacitors that store charge. This charge, with proper interface circuitry, is interpreted as *data*. There are a few things that are very different with DRAM:

- Since the charge is stored in extremely small capacitors, it drains (i.e., leaks) after a certain amount of time (something like 50 ms).
- Due to this leakage, data has to be read and put back on the DRAM (i.e., refreshing).
- It makes no sense to allow byte-at-a-time access to data, considering the disadvantages of refreshing. So, the data is accessed in big rows at a time (e.g., 4 KB at a time).
- There is a long delay in getting a row of data, although, once read, access to that row is extremely fast
- In addition to being *row accessible*, DRAMs have all sorts of other delays, such as row-to-row delay, etc. Each one of these parameters is specified by the memory interface standards that are defined by a consortium of companies.

4.3.3 DRAM Interface Standards

How can CPU manufacturers control the compatibility of the memory chips that some other companies are making ? The answer is : *memory interface standards*. From the first days of the introduction of the memory chips decades ago, there was a need to define standards for memory chips. SDRAM, DDR (double data rate), DDR2, DDR3, and finally, in 2015 the

DDR4 standard (contained inside the PC in Fig. 4.1). These standards are determined by a large consortium of chip manufacturers that define such standards and precise timing of the memory chips. If the memory manufacturers design their memory to be 100% compliant with these standards, and INTEL designs their CPU to be 100% compliant, there is no need for both of these chips to be manufactured by the same company.

In the past four decades, the main memory was always made out of DRAM. A new DRAM standard was released every 2, 3 years to take advantage of the exciting developments in the DRAM manufacturing technology. As the CPUs improved, so did the DRAM. However, not only the improvements in CPU and DRAM technology followed different patterns, but also *improvement* meant something different for CPU and DRAM:

- For CPU designs, *improvement* meant more work done per second.
- For DRAM designs, *improvement* meant i) more data read per second (bandwidth) as well as more storage (capacity).

CPU manufacturers improved their CPUs using better architectural designs and by taking advantage of more MOS transistors that became available at each shrinking technology node (130 nm, 90 nm, ... and 14 nm as of 2016). On the other hand, DRAM manufacturers improved their memories by the ability to continuously pack more capacitors into the same area, thereby resulting in more storage. Additionally, they were able to continuously improve their bandwidths by the newer standards that became available (e.g., DDR3, DDR4).

4.3.4 Influence of DRAM on our Program Performance

The most important question for us as a programmer is this: how does this separation of the CPU vs. DRAM manufacturing technology influence the performance of our programs ? The characteristics of the SRAM vs. DRAM, listed in Sec. 4.3, will stay relatively the same in the foreseeable future. While the CPU manufacturers will always try to increase the amount of cache memory inside their CPU by using more SRAM-based cache memory, DRAM manufacturers will always try to increase the bandwidth and storage area of the DRAMs. However, they will be less concerned about access speeds to small amounts of memory, since, this should be remedied by the increasing amounts of cache memory inside the CPUs. The access speeds of DRAM are as follows:

- DRAM is accessed one row at a time, each row being the smallest accessible amount of DRAM memory. A row is approximately 2–8 KB in modern DRAMs.
- To access a row, a certain amount of time (latency) is required. However, once the row is accessed (brought into the row buffer internally by the DRAM), this row is practically free to access going forward.
- The latency in accessing DRAM is about 200–400 cycles for a CPU, while accessing subsequent elements in the same row is just a few cycles.

Considering all of these DRAM facts, the clear message to a programmer is this:

➢ *Write your programs in such a way that:*
i) accessing data from distant memory locations should be in large chunks, since, we know that, this data will be in DRAM,
ii) accessing small amounts should be highly localized and repetitive, since, we know that, this data will be in SRAM (cache),
iii) Pay extreme attention to multiple threads, since, there is only one memory, but multiple threads: simultaneous threads might cause bad access DRAM patterns although individually they look Ok.

4.3.5 Influence of SRAM (Cache) on our Program Performance

Since there is only one main memory (DRAM), as long as we obey the rules set forth in Table 3.3 and pay attention to the comments in Section 4.3.4, we should have fairly good DRAM performance. But, cache memory, made out of SRAM, is a little different. First of all, there are multiple types of cache memory, making their design *hierarchical*. For a modern CPU like the one shown in the PC in Fig. 4.1, there are three different types of cache memory, built into the CPU:

- L1$ is 32 KB for data (L1 data cache, or L1D$), and 32 KB for instructions (L1 instruction cache, or L1I$). Total L1$ is 64 KB. Access to L1$ is extremely fast (4 cycle load-to-use latency). Each core has its own L1$.
- L2$ is 256 KB. There is no separation between data or instructions. Access to L2$ is fairly fast (11–12 cycles). Each core has its own L2$.
- L3$ is 15 MB. Access to L3$ is faster than DRAM, but much slower than L2$ ($\approx$ 22 cycles). L3$ is shared by all of the cores (6 in this CPU).
- The data that is brought into and evicted out of L1$, L2$, and L3$ is purely controlled by the CPU and is completely out of the programmer's control. However, by keeping the data operations confined into small loops, the programmer can have a significant impact on the effectiveness of the cache memory operation.
- Alternatively, by disobeying the cache efficiency rules, the programmer could nearly render the cache memory useless.
- The best way for the programmer to take advantage of caching is to know the exact sizes of each cache hierarchy and design the programs to stay within these ranges.

Considering all of these SRAM facts, the clear message to a programmer is this:

➢ *To take advantage of caching, write your programs, so that:*

i) each thread accesses 32 KB data regions repetitively

ii) try to confine broader accesses to 256 KB if possible,

iii) when considering all of the launched threads:
try to confine their cumulative data access within L3$ (e.g., 15 MB)

iv) if you must exceed this, make sure that, there is
heavy usage of L3$ before exceeding this region

4.4 IMAGE ROTATION PROGRAM : IMROTATE.C

We covered a lot of information regarding the cache memory (SRAM) and main memory (DRAM). It is time to put this information to good use. In this section, I will introduce a program, imrotate.c, that rotates an image by a specified amount of degrees. This program will put a lot of pressure to the core and memory components of the CPU, thereby allowing us to understand their influence on the overall program performance. We will comment on the performance bottlenecks caused by over-stressing either the cores or the memory and will improve our program. The improved program, imrotateMC.c, the memory-friendly ("M") and core-friendly ("C") version of imrotate.c, will implement the improvements in an incremental fashion via multiple different steps, and each step will be explained in detail.

FIGURE 4.2 The imrotate.c program rotates a picture by a specified angle. Original dog (top left), rotated +10° (top right), +45° (bottom left) and -75° (bottom right) clockwise. Scaling is done to avoid cropping of the original image area.

4.4.1 Description of the imrotate.c

The purpose of imrotate.c is to create a program that is both memory-heavy and core-heavy. imrotate.c will take an image as shown in Fig. 4.2 (top left) and will rotate it clockwise by a specified amount (in degrees). Example outputs of the program are shown in Fig. 4.2 for a +10° rotation (top right), a +45° rotation (bottom left), and a -75° rotation (bottom right). These angels are all specified clockwise, thereby making the last -75° rotation effectively a counter-clockwise +75° rotation. To run the program, the following command line is used:

```
imrotate InputfileName OutputfileName [degrees] [1-128]
```

where degrees specifies the clockwise rotation amount and the next parameter [1–128] is the number of threads to launch, similar to the previous programs.

4.4.2 imrotate.c : Parametric Restrictions and Simplifications

Some simplifications had to be made in the program to avoid unnecessary complications and diversion of our attention to unrelated issues. These are:

- For rectangular images where the width and height are not equal, part of the rotated image will end up outside the original image area.

4.8 CHAPTER SUMMARY

In this chapter, we looked at what happens inside the core and during the memory transfers from the CPU to the main memory. We used this information to make one example program faster. The rules we outlined are simple:

- *Stay away from sophisticated core instructions, such as* sin(), sqrt(). *If you must use them, make sure to have a limited number of them,*
- *The ALU executes integer instructions, FPU executes floating point, Try to have a mix of these instructions in an inner loop to use both units, If you can use only one type, use integer.*
- *Avoid choppy memory accesses. Use big bulk transfers if possible. Try to take heavy computations outside the inner loops.*

Aside from these simple rules, Table 4.3 shows that if the threads we design are *thick*, we will not be able to take advantage of the multiple threads that a core can execute. In our code so far, even the improved Rotate7() function has thick threads. So, the performance falls off sharply when either the launched threads gets close to the number of the physical cores, or we saturate the memory. This brings up the concept of "whichever constraint comes first." In other words, when we change our program to improve it, we might eliminate one bottleneck (say, FPU inside the core), but create a totally different bottleneck (say, main memory bandwidth saturation).

The "whichever constraint comes first" concept will be the key ingredient in designing GPU code, since inside a GPU, there are multiple constraints that can saturate and the programmer has to be fully aware of every single one of them. For example, the programmer could saturate the *total number of launched threads* before the memory is saturated. The *constraint ecosystem* inside a GPU will be very much like the one we will learn in the CPU. More on that coming up very shortly ...

Before we jump right into the GPU world, we have one more thing to learn: *thread synchronization* in Chapter 5. Then, we are ready to take on the GPU challenge starting Chapter 6.

CHAPTER 5

Thread Management and Synchronization

When we say *hardware architecture*, what are we talking about ?
The answer is *everything physical*: CPU, memory, I/O controller, PCI express bus, DMA controller, hard disk controller, hard disk(s), SSDs, CPU chipsets, USB ports, network cards, CD-ROM controller, DVDRW, I can go on for a while ...

How about, when we say *software architecture*, what are we talking about ?
The answer is *all of the code* that runs on the hardware. Code is everywhere: your hard disk, SSDs, your USB controller, your CD-ROM, even your cheap keyboard, your Operating System, and, finally, the programs you write.

The big question is : which one of these do we *care about* as a high-performance programmer ? The answer is : CPU cores and threads, as well as memory in the hardware architecture and the Operating System (OS) and your Application code in the software architecture. Especially if you are writing high performance programs (like the ones in this book), a vast percentage of your program performance will be determined by these things. The disk performance generally does not play an important role in the overall performance because most modern computers have plenty of RAM to cache large portions of the disk. The OS tries to efficiently allocate the CPU cores and memory to maximize performance while your application code requests these two resources from the OS and uses them - hopefully efficiently. Because the purpose of this book is high-performance programming, let's dedicate this chapter to understanding the interplay between your our own code and the OS when it comes to allocating/using the cores and memory.

5.1 EDGE DETECTION PROGRAM: IMEDGE.C

In this section, I will introduce an image edge detection program, imedge.c, that detects the edges in an image as shown in Fig. 5.1. In addition to providing a good template example that we will continuously improve, edge detection is actually one of the most common image processing tasks and is an essential operation that the human retina performs. Just like the programs I introduced in the preceding chapters, this program is also resource-intensive; therefore, I will introduce its variant imedgeMC.c that is memory-friendly ("M") and core-friendly ("C"). One interesting variant I will add is imedgeMCT.c, which adds the thread-friendliness ("T") concept by utilizing a *MUTEX* structure to facilitate communication among multiple threads.

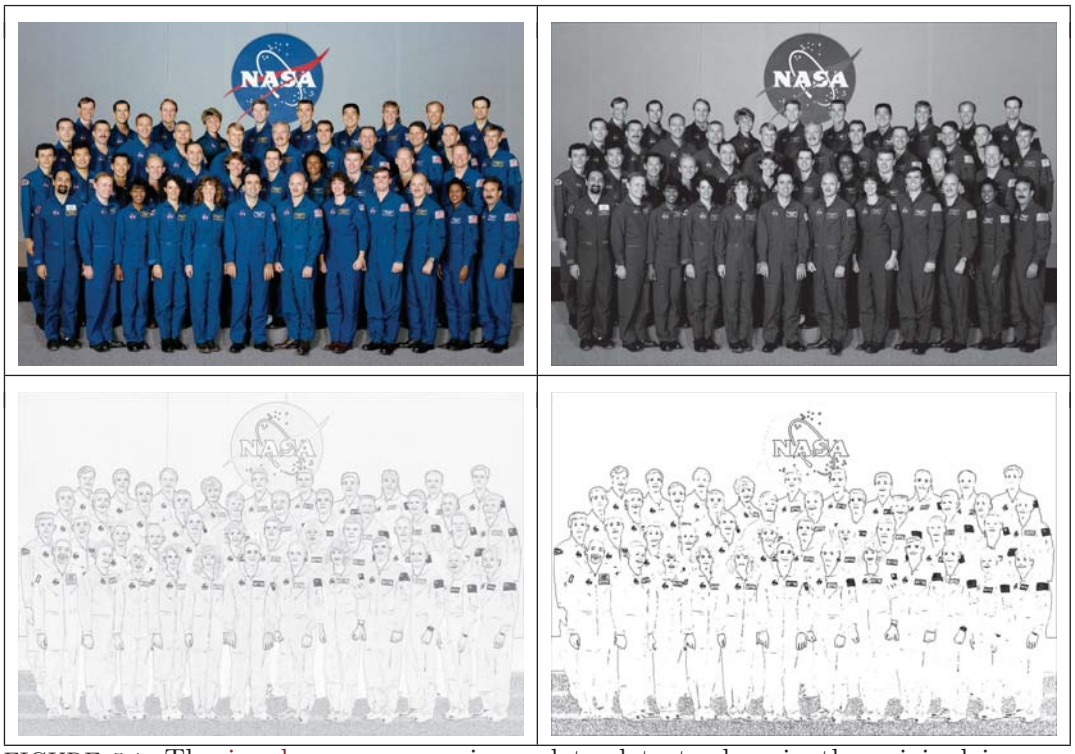


FIGURE 5.1 The imedge.c program is used to detect edges in the original image astronaut.bmp (top left). Intermediate processing steps are: GaussianFilter() (top right), Sobel() (bottom left), and finally Threshold() (bottom right).

5.1.1 Description of the imedge.c

The purpose of imedge.c is to create a program that is core-heavy, memory-heavy, and is composed of multiple independent operations (functions):

- GaussianFilter(): The initial smoothing filter to reduce noise in the original image.
- Sobel(): The edge detection kernel we apply to amplify the edges.
- Threshold(): The operation that turns a gray scale image to a binary (black & white) image (denoted B&W going forward), thereby "detecting" the edges.

imedge.c takes an image such as the one shown in Fig. 5.1 (top left) and applies these three distinct operations one after the other to finally yield an image that is composed of only the edges (bottom right). To run the program, the following command line is used:

```
imedge InputfileName OutputfileName [1-128] [ThreshLo] [ThreshHi]
```

where [1–128] is the number of threads to launch as usual and the [ThreshLo], [ThreshHi] pair determines the thresholds used during the Threshold() function.

5.1.2 imedge.c : Parametric Restrictions and Simplifications

Some simplifications had to be made in the program to avoid unnecessary complications and diversion of our attention to unrelated issues. Numerous improvements are possible

`NextRowToProcess` is being updated by all N threads, necessitating the usage of a MUTEX to avoid updating problems.

- Each instance of the AMTPreCalcRow() function makes no assumption on how many rows it is supposed to process, since it could differ among different instances of this function; therefore, the only terminating condition for each instance of AMTPreCalcRow() is when the `NextRowToProcess` reaches the end of the image and there is nothing more to process.
- AMTPreCalcRow() idles in the case that the `LastRowRead` variable indicates a slow hard disk read by PrAMTReadBMP(). Although, remember that the initially-buffered 20 rows should avoid this in Code 5.14.

5.8 PERFORMANCE OF IMEDGEMCT

Table 5.4 shows the run time results for the imedgeMCT.c program. The top part of the code compares the runtimes of imedgeMCT.c and imedgeMC.c. Because we only redesigned the PrReadBMP() function, its redesigned asymmetric multi-threaded version PrAMTReadBMP() benefits from multi-threading and gets progressively faster as the number of threads increases. The impact of this on the overall performance is clearly visible.

I also ran imedgeMCT.c on a Xeon 5110P, which has 60 cores and 240 threads; because each thread is a thick thread in almost every function we are using, the additional threads did not benefit the Xeon, saturating the performance when we got closer to 60 threads. Remember from Section 3.9 that benefitting from the vast amount of threads that exist in a Xeon required meticulous engineering; unless thick threads are intermixed with thin threads as described in Section 3.2.2, no additional benefit from Xeon will be gained when the number of threads increases beyond the core count.

TABLE 5.4 imedgeMCT.c execution times (in ms) for the W3690 CPU (6C/12T), using the Astronaut.bmp image file (top) and Xeon Phi 5110P (60C/240T) using the dogL.bmp file (bottom).

Function #threads ⟹	**1**	**2**	**4**	**8**	**10**	**12**		
PrAMTReadBMP()	2267	1264	920	1014	1020	1078		
Create arrays	33	31	31	33	32	33		
PrGaussianFilter()	2223	1157	567	556	582	611		
PrSobel()	7415	3727	1910	1124	948	842		
PrThreshold()	341	195	119	107	99	104		
WriteBMP()	61	62	60	63	61	63		
imedgeMCT.c w/o IO	12640	6436	3607	2897	2742	2731		
PrReadBMP()	2836	2846	2833	2881	2823	2898		
Create arrays	31	32	31	36	31	31		
PrGaussianFilter()	2179	1143	570	526	539	606		
PrSobel()	7475	3833	1879	1141	945	864		
PrThreshold()	358	193	121	107	113	107		
WriteBMP()	61	60	61	61	60	61		
imedgeMC.c w/o IO	12940	8107	5495	4752	4511	4567		
Speedup (W3690)	1.02×	1.26×	1.52×	1.64×	1.64×	1.67×		
Xeon #threads ⟹	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Xeon Phi 5110P no IO	3994	2178	1274	822	604	507	486	532

PART II

GPU Programming Using CUDA

CHAPTER 6

Introduction to GPU Parallelism and CUDA

We have spent a considerable amount of time in understanding the CPU parallelism and how to write CPU parallel programs. During the process, we have learned a great deal about how, simply bringing a bunch of cores together will not result in a magical performance improvement of a program that was designed as a serial program to begin with. This is the first chapter where we will start understanding the inner-workings of a GPU; good news is that we have such a deep understanding of the CPU that we can make comparisons between a CPU and GPU along the way. While so many of the concepts will be dead similar, some of the concepts will only have a place in the GPU world. It all starts with the monsters ...

6.1 ONCE UPON A TIME ... NVIDIA ...

Yes, it all starts with the monsters. As many game players know, many games have monsters or planes or tanks moving from here to there and interacting heavily during this movement, whether it is crashing into each other or being shot by the game player to kill the monsters. What do these actions have in common ? i) a plane moving in the sky, ii) a tank shooting, iii) a monster trying to grab you by moving his arms and his body. The answer —from a mathematical standpoint— is that all of these *objects* are high resolution graphics elements, composed of many pixels, and moving them (such as rotating them) requires heavy floating point computations, as exemplified in Eq. 4.1 during the implementation of the imrotate.c program.

6.1.1 The birth of the GPU

Computer games have existed as long as computers did. I was playing computer games in the late 1990's and I had an Intel CPU in my computer; something like a 486. Intel offered two flavors of the 486 CPU: 486SX and 486DX. 486SX CPUs did not have a built-in floating point unit (FPU), whereas 486DX CPUs did. So, 486SX was really designed for more general purpose computations, whereas 486DX made games work much faster. So, if you were a gamer like me in the late 1990's and trying to play a game that had a lot of these tanks, planes etc. in it, hopefully you had a 486DX, because otherwise your games would play so slow that you would not be able to enjoy them. Why ? Because games require heavy floating point operations and your 486SX CPU does not incorporate an FPU that is capable of performing floating point operations fast enough. You would resort to using your ALU to emulate an FPU, which is a very slow process.

This story gets worse. Even if you had a 486DX CPU, the FPU inside your 486DX was

still not fast enough for most of the games. Any exciting game demanded a 20× (or even 50×) higher-than-achievable floating point computational power from its host CPU. Surely, in every generation the CPU manufacturers kept improving their FPU performance, just to witness a demand for FPU power that grew much faster than what the improvements they could provide. Eventually, starting with the Pentium generation, the FPU was an integral part of a CPU, rather than an option, but this didn't change the fact that significantly higher FPU performance was needed for games. In an attempt to provide much higher scale FPU performance, Intel went on a frenzy to introduce vector processing units inside their CPUs: the first ones were called MMX, then SSE, then SSE2, and the ones in 2016 is SSE4.2. These vector processing units were capable of processing many FPU operations in parallel and their improvement has never stopped.

Although these vector processing units helped certain applications a lot —and they still do, the demand for an ever-increasing amount of FPU power was insane ! When Intel could deliver a 2× performance improvement, game players demanded 10× more. When they could eventually manage to deliver 10× more, they demanded 100× more. Game player were just monsters that ate lots of FLOPS ! And, they were always hungry ! Now what ? This was the time when a paradigm shift had to happen. Late 1990's is when the manufacturers of many plug-in boards for PCs —such as sound cards or ethernet controller— came up with the idea of a card that could be used to accelerate the floating point operations. Furthermore, by providing dedicated hardware to perform monotonous conversions, such as a 3D-to-2D projections of images, and the computation of triangles from pixels. Note that the actual unit element of a monster in a game is a *triangle*, not a pixel. Using triangles allows the games to associate a *texture* for the surface of any object, like the skin of a monster or the surface of a tank, something that you cannot do with simple pixels.

These efforts of the PC card manufacturers to introduce products for the game market gave birth to a type of card that would soon be called a *Graphics Processing Unit*. Of course, we love acronyms: it is a GPU ... A GPU was designed to be a "plug-in card" that required a connector such as PCI, AGP, PCI Express, etc. Early GPUs in the late 1990's strictly focused on delivering as high of a floating point performance as possible. This freed the CPU resources and allowed a PC perform 5× or 20× better in games (or even more if you were willing to spend a lot of money on a fancy GPU). Someone could purchase a $100 GPU for a PC that was worth $500; for this 20% extra investment, the computer performed 5× faster in games. Not a bad deal. Alternatively, by purchasing a $200 card (i.e., a 40% extra investment), your computer could perform 20× faster in games. Late 1990's was the point of no return, after which the GPU was an indispensable part of every computer, not just for games, but for a multitude of other applications explained below. Apple computers used a different strategy to build a GPU-like processing power into their computers, but sooner or later (e.g., in year 2017, the release year of this book) the PC and Mac lines have converged and they started using GPUs from the same manufacturers.

6.1.2 Early GPU architectures

If you were playing Pacman, an astonishingly popular game in the 1980's, you really didn't need a GPU. First of all, all of the objects in Pacman were 2D —including the monster that is chasing you and trying to eat you— and the movement of all of the objects were in the restricted to 2D —x and y— dimensions. Additionally, there were no sophisticated computations in the game that required the use of transcendental functions, such as sin() or cos(), or even floating point computations of any sort. The entire game could run by performing integer operations, thereby requiring only an ALU. Even a low-powered CPU

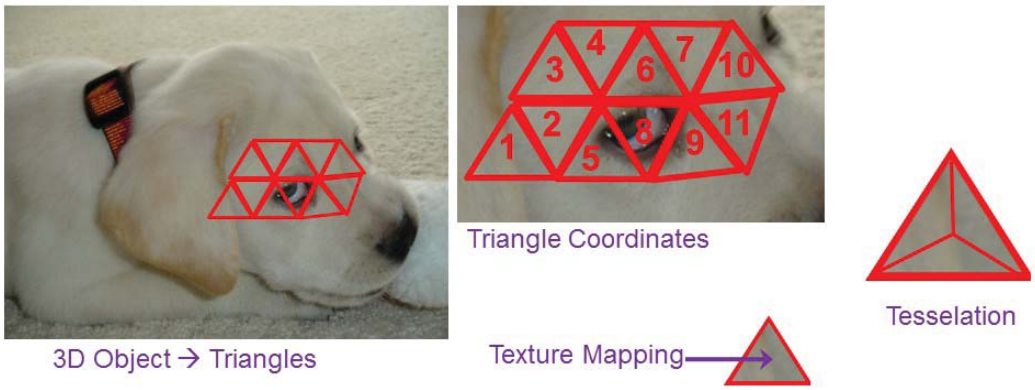


FIGURE 6.1 Turning the dog picture into a 3D wire frame. Triangles are used to represent the object, rather than pixels. This representation allows us to map a texture to each triangle. When the object moves, so does each triangle, along with their associated textures. To increase the resolution of this kind of an object representation, we can divide triangles into smaller triangles in a process called *tesselation*.

was perfectly sufficient to compute all of the required movements in real time. However, having watched the Terminator 2 movie a few years ago, the Pacman game was far from exciting for gamers of the 90's. First of all, objects had to be 3D in any good computer game and the movements were substantially more sophisticated than Pacman —and in 3D, requiring every transcendental operation you can think of. Furthermore, because the result of any transcendental function due to a sophisticated object move —such as the rotation operation in Eq. 4.1 or the scaling operation in Eq. 4.3— required the use of floating point variables to maintain image coordinates, GPUs, by definition, had to be computational units that incorporated significant FPU power. Another observation that the GPU manufacturers made was that the GPUs could have a significant edge in performance if they also included dedicated processing units that performed routine conversions from pixel-based image coordinates to triangle-based object coordinates, followed by texture mapping.

To appreciate what a GPU has to do, consider Fig. 6.1, in which our dog is represented by a bunch of triangles. Such a representation is called a *wire-frame*. In this representation, a 3D *object* is represented using triangles, rather than an *image* using 2D pixels. The unit of element for this representation is a triangle with an associated texture. Constructing a 3D wire-frame of the dog will allow us to design a game in which the dog jumps up and down; as he makes these moves, we have to apply some transformation —such as rotation, using the 3D equivalent of Eq. 4.1— to each triangle to determine the new location of that triangle and map the associated texture to each triangle's new location. Much like a 2D image, this 3D representation has the same "resolution" concept; to increase the resolution of a triangulated object, we can use *tesselation*, in which a triangle is further sub-divided into smaller triangles as shown in Fig.6.1. Note: Only 11 triangles are shown in Fig. 6.1 to avoid cluttering the image and make our point on a simple figure; in a real game, there could be millions of triangles to achieve sufficient resolution to please the game players.

Now that we appreciate what it takes to create scenes in games where 3D objects are moving freely in the 3-dimensional space, let's turn our attention the underlying computations to create such a game. Figure 6.2 depicts a simplified diagram of the steps involved in moving a 3D object. The designer of a game is responsible for creating a wire frame of

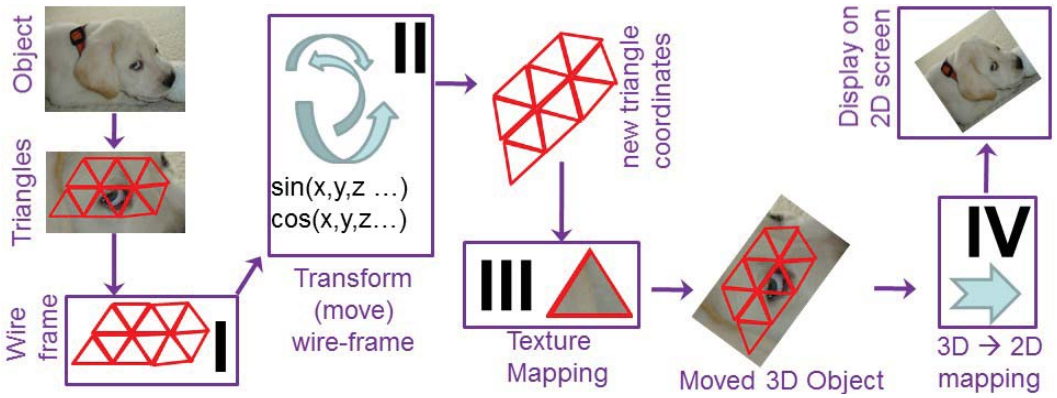


FIGURE 6.2 Steps to move triangulated 3D objects. Triangles contain two attributes: their *location* and their *texture*. Objects are moved by performing mathematical operations only on their coordinates. A final texture mapping places the texture back on the moved object coordinates, while a 3D-to-2D transformation allows the resulting image to be displayed on a regular 2D computer monitor.

each object that will take part in the game. This wire frame includes not only the locations of the triangles —composed of 3 points for each triangle, having an x, y, and z coordinate each— but also a texture for each triangle. This operation *decouples* the two components of each triangle: i) the *location* of the triangle, and ii) the *texture* that is associated with that triangle. After this coupling, triangles can be moved freely, requiring only mathematical operations on the coordinates of the triangles. The texture information —stored in a separate memory area called *texture memory*— doesn't need to be taken into account until all of the moves have been computed and it is time to display the resulting object in its new location. Texture memory does not need to be changed at all, unless, of course, the object is changing its texture, as in the *Hulk* movie, where the main character turns green when stressed out ! In this case, the texture memory also needs to be updated in addition to the coordinates, however, this is a fairly infrequent update when compared to the updates on the triangle coordinates. Before displaying the moved object, a *texture mapping* step fills the triangles with their associated texture, turning the wire-frame back into an object. Next, the re-computed object has to be displayed on a computer screen; because every computer screen is composed of 2D pixels, a 3D-to-2D transformation has to be performed to display the *object* as an *image* on the computer screen.

6.1.3 The birth of the GPGPU

Early GPU manufacturers noticed that the GPUs could excel in being specialized game cards if they incorporated dedicated hardware into the GPUs that performed the following operations. Each operation is denoted with a Roman numeral in Fig. 6.2:

- ability to deal with the natural data type *triangle*, Box I in Fig. 6.2,
- ability to perform heavy floating point operations on triangles (Box II),
- ability to associate texture with each triangle and store this texture in a separate memory area called *texture memory* (Box III),

- ability to convert from triangle coordinates back to image coordinates for display in a computer screen (Box IV).

Based on this observation, right from the first day, every GPU was manufactured with the ability to implement some sort of functionality that matched all of these boxes. GPUs kept evolving by incorporating faster Box II's, although the concept of Box I, III, and IV never changed too much. Now, imagine that you are a graduate student in the late 1990s —in a Physics department— and trying to write a particle simulation program that requires an extensive amount of floating point computations. Before the introduction of the GPUs, all you could use was a CPU that had an FPU in it and, potentially a vector unit. However, when you bought one of these GPUs at an affordable price and realize that they can perform a much higher volume of FPU operations, you would naturally start thinking: "Hmmm... I wonder if I could use one of these GPU things in my particle simulations ?" This investigation would be worth every minute you put into it, because you know that these GPUs are capable of 5× or 10× faster FPU computations. The only problem at that time was that the functionality of Box III abd Box IV couldn't be "shut off." In other words, GPUs were not designed for non-gamers who are trying to do particle simulations!

Nothing can stop a determined mind ! It didn't take too long for our graduate student to realize that if (s)he mapped the location of the particles as the triangle locations of the monsters and somehow performed particle movement operations by emulating them as monster movements, it could be possible to "trick" the GPU into thinking that you are actually playing a game, in which particles (monsters) are moving here and there and smashing into each other (particle collisions). You can only imagine the massive challenges our student had to endure: First, the native language of the games was Open GL, in which objects were graphics objects and computer graphics APIs had to be used to "fake" particle movements. Second, there were major inefficiencies in the conversions from monster-to-particle and particle-back-to-monster. Third, accuracy was not that great because the initial cards could only support single precision FPU operations, not double precision. It is not like our student could make a suggestion to the GPU manufacturers to incorporate double precision to improve the particle simulation accuracy; GPUs were game cards and they were game card manufacturers, period ! None of these challenges stopped our student ! Whoever that student was, the unsung hero, created a multi-billion dollar industry of GPUs that are in almost every top supercomputer today.

Extremely proud of the success in tricking the GPU, the student published the results ... The cat was out of the bag ... This started an avalanche of interest; if this trick can be applied to particle simulations, why not circuit simulations ? So, another student applied it to circuit simulations. Another one to astro-physics, another one to computational biology, another ... These students invented a way to do General Purpose computations using GPUs, hence the birth of the term GPGPU.

6.1.4 Nvidia, ATI Technologies, and Intel

While the avalanche was coming down the mountain and universities were purchasing GPUs in a frenzy to perform scientific computations —despite the challenges associated with trying to use a game card for "serious" purposes by using sophisticated mappings— GPU manufacturers were watching on the sidelines and trying to gauge the market potential of their GPUs in the GPGPU market. To significantly expand this market, they had to make modifications to their GPUs to eliminate the tedious transformations the academics had to go through to use GPUs as scientific computation cards. What they realized fairly quickly was that the market for GPGPUs was a lot wider than just the academic arena; for

example, oil explorers could analyze the underwater SONAR data to find oil under water, an application that requires a substantial volume of floating point operations. Alternatively, the academic and research market, including many universities and research institutions such as NASA or Sandia National Labs, could use the GPGPUs for extensive scientific simulations. For these simulations, they would actually purchase 100's of the most expensive versions of GPGPUs and GPU manufacturers could make a significant amount of money in this market and create an alternative product to the already-healthy game products.

In the late 1990s, GPU manufacturers were small companies that saw GPUs as ordinary add-on cards that were no different than hard disk controllers, sound cards, ethernet cards, or modems. They had no vision of the month of September 2017, when Nvidia would become a company that is worth $112 B (112 billion US dollars) in the Nasdaq stock market (Nasdaq stock ticker NVDA), a pretty impressive 20 year accomplishment considering that Intel, the biggest semiconductor manufacturer on the planet with its five decade history, was worth $174 B the same month (Nasdaq stock ticket INTC). The vision of the card manufacturers changed fairly quickly when the market realized that GPUs were not in the same category as other add-on cards; it didn't take a genius to figure out that the GPU market was ready for an explosion. So the gold rush started. GPU cards needed two main ingredients: 1) the GPU chips, responsible for all of the computation, 2) GPU memory, something that could be manufactured by the CPU DRAM manufacturers that were already making memory chips for the CPU market, 3) interface chips to interface to the PCI bus, 4) power supply chips that provide the required voltages to all of these chips, and 5) other semiconductors to make all of these to work together, sometimes called "glue logic."

The market already had manufacturers for (2), (3), and (4). Many small companies were formed to manufacture (1), the GPU "chips," so the functionality shown in Fig. 6.2 could be achieved. The idea was that GPU chip designers —such as Nvidia— would design their chips and have them manufactured by third parties —such as TSMC— and sell the GPU chips to contractor manufacturers such as FoxConn. FoxConn would purchase the other components (2,3,4, and 5) and manufacture GPU add-on cards. Many GPU chip designers entered the market just to see a massive consolidation towards the end of 1990's. Some of them bankrupted and some of them sold out to bigger manufacturers. As of 2016, only three key players remain in the market (Intel, AMD, and Nvidia), two of them being actual CPU manufacturers. Nvidia became the biggest GPU manufacturer in the world as of 2016 and made multiples pushes to enter into the GPU/CPU market by incorporating ARM cores into their Tegra line GPUs. Intel and AMD kept incorporating GPUs into their CPUs to provide an alternative to consumers that didn't want to buy a discrete GPU. Intel has gone through many generations of designs eventually incorporating Intel HD Graphics and Intel Iris GPUs into their CPUs. Intel's GPU performance improved to the point when in 2016, Apple deemed them the built-in Intel GPU performance sufficient to be included in their Mac Books as the only GPU, instead of discrete GPUs. Additionally, Intel introduced the Xeon Phi cards to compete with Nvidia in the high-end supercomputing market. While this major competition was taking place in the desltop market, the mobile market saw a completely different set of players emerge. QualComm and Broadcom built GPU cores into their mobile processors by licensing them from other GPU designers. Apple purchased a processor designers to design their "A" family processors that had built-in CPUs and GPUs with extreme low power consumption. By about 2011 or 2012, CPUs couldn't be thought as hte only processing unit of any computer or mobile device. CPU+GPU was the new norm.

6.2 COMPUTE-UNIFIED DEVICE ARCHITECTURE (CUDA)

Nvidia, convinced that the market for the GPGPUs was large, decided to turn all of their GPUs into GPGPUs by designing the "Box II" in Fig. 6.2 as a general purpose computation unit and exposing it to the GPGPU programmers. For efficient GPGPU programming, bypassing the graphics functionality (Box I, Box III, and Box IV, in which *graphics-specific* operations take place) was necessary, while Box II is the only necessary block for scientific computations. To phrase alternatively, games needed access to Boxes I, III, and IV (the "G" part) nad while scientific computation needs only Box II (the "PU" part). They also had to allow the GPGPU programmers to input data directly into Box II without having to go through Box I. Furthermore, *triangle* wasn't a friendly data type for the scientific computations, suggesting that the natural data types in Box II had to be the usual integers, float, and double.

6.2.1 CUDA, Open CL, and other GPU languages

In addition to these hardware implications, a software architecture was necessary to allow GPGPU programmers to develop GPU code without having to learn anything about computer graphics, which uses Open GL. Considering all of these facts, Nvidia introduced their language Computer-Unified Device Architecture (CUDA) in 2007, which was —and still is— designed strictly for Nvidia platforms. Two years later the Open CL language emerged that allowed GPU code to be developed for Intel, AMD, and other GPUs. While AMD initially introduced their own language CTM (Close to Metal), they eventually abandoned these efforts and went strictly with Open CL. As of year 2016, there are two predominant desktop GPU languages in the world: Open CL and CUDA. I must note here that the landscape is different in the mobile market and this is not the focus of this book.

Right from the introduction, GPUs were never viewed as *processors*; they were always conceptualized as being "*co-processors*" that worked under the supervision of a host CPU. All of the data went through the CPU first before reaching the GPU. Therefore, a connection of some sort, for example a PCI Express bus, was always necessary to interface the GPU to its host CPU. This fact completely dictated the hardware design of the GPU, as well as the programming language required for GPU coding. Both Nvidia's CUDA and its competition Open CL developed their programming languages to include a *host side code* and a *device side code*. Instead of calling it *GPU code*, it is more general and appropriate to call it "device side code," because, for example, a device doesn't have to be a GPU in Open CL; it can be an FPGA, DSP, or any other device that has a similar parallel architecture to the GPU. The current implementation of Open CL 2.3 allows the same code to be used for a multitude of aforementioned devices with very minor modifications. While this generalization is great in some applications, our focus is strictly the GPU code in this book. So, my apologies if I slip and call it GPU code in some parts of the book.

6.2.2 Device side vs. Host side code

The initial CUDA language developers had the following dilemma: CUDA had to be a programming language that allowed the programmers to write code for both the CPU and the GPU. Knowing that two completely different processing elements (CPU and GPU) had to be programmed, how would CUDA work ? Because the CPU and GPU both had their separate memory, how would the data transfers work ? Programmers simply didn't want to learn a brand new language, so, CUDA has to have a similar syntax on both the CPU and

GPU code side ... Furthermore, a single compiler would be great to compiler both sides' code, without requiring two separate compilations.

- ➢ *There is no such thing as GPU Programming ...*
- ➢ *GPU always interfaces to the CPU through certain APIs ...*
- ➢ *So, there is always CPU+GPU programming ...*

Given these facts, CUDA had to be based on the C programming language (for the CPU side) to provide high performance. The GPU side also had to be almost exactly like th CPU side with some specific keywords to distinguish between host vs. device code. The burden to determine how the execution would take place at runtime —regarding CPU vs. GPU execution sequences— had to be determined by the CUDA compiler. GPU parallelism had to be exposed on the GPU side with a mechanism similar to the Pthreads we saw in the Part I of this book. By taking into account all of these facts, Nvidia designed their nvcc compiler that is capable of compiling CPU and GPU code simultaneously. CUDA, since its inception, has gone through many version updates, incorporating an increasing set of sophisticated features. The version I use in this book is CUDA 8.0, released in September 2016. Parallel to the progress of CUDA, Nvidia GPU architectures have gone through massive updates as I will document shortly.

6.3 UNDERSTANDING GPU PARALLELISM

The reason you are reading a GPU programming book is the fact that you want to program a device (GPU) that can deliver a superior computational performance as compared to a CPU. The big questions is: why can A GPU deliver such a high performance ? To understand this, let us turn our attention to an analogy.

ANALOGY 6.1: *CPU vs. GPU*

Cocotown had an annual competition for harvesting 2048 coconuts. The strongest farmer in town, Arnold, had a big reputation for owning the fastest tractor and being the strongest guy that can pick and harvest coconuts twice faster than any other farmer in town. This year, a group of ambitious farmer brothers, Fred and Jim, challenged Arnold; they claimed that although their tractor was half the size of Arnold's, they could still beat Arnold in the competition. Arnold gladly accepted the challenge. This was going to be the most fun competition to watch for the residents who stayed in the sidelines and cheered for their team.

Just before the competition started, another farmer —Tolga, who has never competed before— claimed that he could win this. His setup was completely different: he would drive a bus, which could seat 32 people and the driver. Inside the bus, he would actually seat 32 boy and girl scouts that would help him harvest the coconuts. He wouldn't do any work, but, actually give instructions to the scouts, so, they could know what to do next. Additionally, he would report the results and return the harvested coconuts. The scouts had no experience, so, their individual performance was a quarter of the other farmers. Additionally, Tolga faced major challenges in coordinating the instructions among the scouts. He actually had to give them a piece of rope to hold onto, so, they could coordinate their movements in lock step.

Who do you think won the competition?

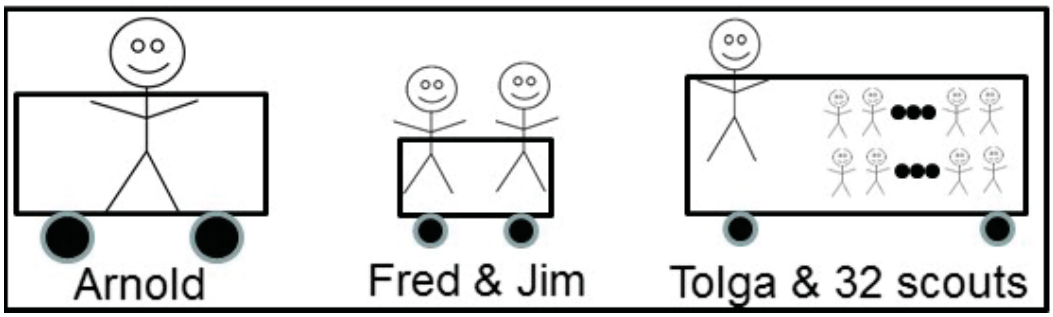


FIGURE 6.3 Three farmer teams compete in Analogy 6.1: i) Arnold competes alone with his 2× bigger tractor and " the strongest farmer" reputation, ii) Fred and Jim compete together in a much smaller tractor than Arnold. iii) Tolga, along with 32 boy and girl scouts, compete together using a bus. Who wins ?

Analogy 6.1 is depicted in Fig. 6.3 with three alternatives: Arnold represents a single-threaded CPU that can work at 4 GHz, while Fred and Jim together in their smaller tractor represent a dual-core CPU in which each core works at something like 2.5 GHz. We have done major evaluations on the performance differences between these two alternatives in the Part I of the book. The interesting third alternative in Fig. 6.3 is Tolga with the 32 boy/girl scouts. This represents a single CPU core —probably working at 2.5 GHz— and a GPU co-processor composed of 32 small cores that each work at something like 1 GHz. How could we compare this alternative to the first two ?

6.3.1 How does the GPU achieve high performance ?

First of all, looking at Fig. 6.3, if Tolga was alone, his performance would be half that of the second team and probably half that of Arnold's. So, Tolga wouldn't be able to win the competition alone. But, as long as Tolga can coordinate efficiently with the 32 scouts, we can expect a significant performance from the third team. But, how much ? Let's do the math. If the *parallelization overhead* was negligible —ie., the threading efficient was 100%, translating to $\eta = 1.0$ in Eq. 4.4— the theoretical maximum we expect would be $2.5+32*1 = 34.5$, which is close to 8× of Arnold's performance. This is a very rough estimate to provide a simple back-of-the-envelope number —by simply adding the GHz values of the different processing elements together, i.e., Tolga is at 2.5 GHz, and the scouts are at 1 GHz each— and ignores architectural differences as well as many other phenomena that contribute to reduced performance in parallel architectures. However, it definitely provides a *theoretical maximum*, because any of these negative factors will *reduce* the performance further beyond this number.

In this example, even if Tolga was burdened with shuttling data from/to the scouts and didn't have time to do any real work, we are still at 32, instead of 34.5. The power of this third alternative comes from the sheer quantity of the small GPU cores; multiplying any number by 32 creates a large number. So, even if we work the GPU cores at 1 GHz, as long as we can pack 32 cores into the GPU, we can still surpass a single core CPU working at 4 GHz. However, in reality, things are different: CPUs have a lot more than two cores and so do GPUs. In 2016, an eight core, 16-thread (8C/16T) desktop processor was common and a high-end GPU incorporated 1000–3000 cores. As the CPU manufacturers kept building an increasing number of cores into their CPUs, so did GPU manufacturers. What makes

6.4.16 Hflip(): The GPU kernel for horizontal flipping

Code 6.8 shows the GPU kernel function that is responsible for flipping a pixel's 3 bytes in the horizontal direction. Despite some differences in the way the indexes are calculated, Code 6.8 is almost identical to Code 6.7, in which a pixel was vertically-flipped.

6.4.17 Hardware parameters: threadIDx.x, blockIdx.x, blockDim.x

Both Code 6.7 and Code 6.8 one thing in common: There are no explicit looping in either one, because the Nvidia hardware is responsible for providing the `threadIDx.x`, `blockIdx.x`, and `blockDim.x` variables to every thread it launches. Every launched thread knows exactly what its thread and block ID is, as well as how many threads are launched in each block. So, with clever indexing, it is possible for every thread to get the `for` loops for free, as we see in both Code 6.7 and Code 6.8. Considering that each thread does such a small amount of work in GPU kernels, the savings from the looping overhead can be drastic.

We will spend a lot of time in the next chapter analyzing every single line of each function and will improve them as well as understand how they map to the GPU architecture.

CODE 6.9: imflipG.cu PixCopy() {...}

The GPU kernel PixCopy() that copies an image.

```
// Kernel that copies an image from one part of the
// GPU memory (ImgSrc) to another (ImgDst)
__global__
void PixCopy(uch *ImgDst, uch *ImgSrc, ui FS)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;

   if (MYgtid > FS) return;         // outside the allocated memory
      ImgDst[MYgtid] = ImgSrc[MYgtid];
}
```

6.4.18 PixCopy(): The GPU kernel for copying an image

Code 6.9 is how we copy an image entirely into another one. Comparing the PixCopy() function in Code 6.9 to the other two functions in Code 6.7 and Code 6.8, we see that the major difference is the usage of one-dimensional pixel indexing in PixCopy(). In other words, PixCopy() does not try to calculate which row and column it is responsible for. Each thread in PixCopy() copies only a single byte. This also eliminates the wasted threads at the edge of the rows, however, there are still wasted threads at the very end of the image. If we were to launch our program with the ('C' — copy) option, we would still have to launch it with, say, 256 threads per block. However, because the copying is linear, rather than based on a 2D indexing, we would have a different number of blocks to process it. Let's do the math: The image is a total of 127712256 Bytes. Each each block has 256 threads, we need a total of $\left\lceil \frac{127712256}{256} \right\rceil = 498876$ blocks launched, each block containing 256 threads. In this

CHAPTER 7

CUDA Host/Device Programming Model

GPU is a co-processor, with almost no say in how the CPU does its work. So, although we will study the GPU architecture in great detail in the following chapters, we will focus on the CPU-GPU interaction in this chapter. The programming model of a GPU is a host/device model, in which the host (CPU) issues co-processor like commands to the device (GPU) and has no idea what the GPU is doing during their execution, although it can use a rich set of query commands to determine its status during the execution. To the CPU, all that matters is that the GPU finishes its execution and the results shows up somewhere that it can access. GPU, on the other hand, receives commands from the CPU through a set of API functions —provided by Nvidia— and executes them. So, although CPU programming can be learned independently, GPU programming should be learned in conjunction with CPU programming, hence the reason for the organization of this book.

- *There is no such thing as GPU programming; there is only CPU+GPU programming ... You can't learn "just GPU programming."*
- *When you were learning how to ride a bicycle, what did you do ?*
  Did you learn "just how to pedal" and ignore steering ?
  No, you either learned both or you didn't know how to ride a bike.
- *The CPU code dictates what the GPU does.*
  So, you need to learn their programming together; you can't just learn one.

In this chapter, we will focus on the parts of the GPU programming that involves both the CPU and GPU, which are the launch dimensions of a GPU kernel, PCI Express bandwidth and its impact on the overall performance, and the memory bandwidth of the CPU and the GPU.

7.1 DESIGNING YOUR PROGRAM'S PARALLELISM

In this section, before we worry about what the GPU parameters —be it launch dimensions or GPU core structure— we will worry about the parallelization of the task. Remember from Section 6.4.8 that we could launch millions and millions of threads to run the GPU code. This amazing luxury means that we will do a lot less *looping* during our kernel execution, however, it introduces an interesting dilemma: how can we take advantage of this massive parallelism in the best way possible ? So, the first step in GPU programming is to determine how the task should be parallelized to map to the GPU hardware perfectly. In the case of GPU parallelism, our options were fairly limited; with the potential of being able to run only a few threads (say, 8), we simply chopped up the image into 8 pieces and had each

thread process a portion of $\frac{1}{8}$ of the image. However, when there are millions of threads we can launch, a slew of very different considerations come up. Let's first conceptualize our parallelism without writing a single line of GPU code.

7.1.1 Conceptually parallelizing a task

I am writing this section with the assumption that we are vertically flipping the 121 MB Astronaut.bmp image —shown in Fig. 5.1— using the imflipG.cu program. The first step in designing this program is to determine where the source and destination images will be stored in both the CPU and GPU side memory. We saw in Section 6.4.14 that the Vflip() kernel (Code 6.7) is the GPU kernel that actually performs this function. The main() function allocated the CPU and GPU memory areas using malloc and cudaMalloc(), respectively. The pointers that came from this allocation were as follows (shown in Code 6.1):

```
uch *TheImg, *CopyImg;          // Where images are stored in CPU
uch *GPUImg, *GPUCopyImg;       // Where images are stored in GPU
```

The pointer `TheImg` points to the original CPU image array, which is then copied onto the GPU image array, pointed to by the `GPUImg` pointer. When the GPU kernel executes, it takes the image that is stored in the array that is pointed to by `GPUImg`, flips it, and stores the flipped image in the GPU memory, pointed to by `GPUCopyImg`. This image, then, is transferred back to the CPU's second image array, pointed to by the `CopyImg` pointer.

If this code was being executed using 8 CPU threads each thread would be responsible for copying an eight of the image, but, GPU parallelism differs despite major similarities. Here are the basic rules for GPU parallelism:

- The GPU code is written using *threads*, exactly like the CPU code, so, a GPU kernel is the code for each *thread*. In that sense, a thread can be thought of as being a *building block of the task*. So, the programmer designs the GPU program based on threads.
- Because no less than 32 threads (a warp) executes at any given point in time, a *warp* can be thought of as being the *building block of code execution*. So, the GPU executes the program based on warps.
- In many cases, a warp is too small of an execution unit for the GPU. So, we launch our kernels in terms of *blocks*, which are a bunch of warps, clumped up together. In that sense, a *block* can be thought of as being *building block of code launch*. So, the programmer launches his/her kernels in terms of blocks. The notion of the warp is more like a good trivia; a programmer conceptualizes everything in terms of threads/block. In all of our GPU kernels, we will always ask ourselves the following question: "how many threads should our blocks have ?" Rarely we will worry about warps (if ever). The size of a warp for Nvidia GPUs never changed in the past two decades of Nvidia GPU designs, but, we have to keep the possibility in mind that Nvidia might decide to change the warp size to something other than 32 in future GPU generations. But, the notion of *how many threads are in a block ?* will never change. The programmer doesn't really think of the block size in terms of warps.
- Common block sizes are 32, 64, 128, 256, 512, or 1024 threads/block.

7.1.2 What is a good block size for Vflip() ?

Based on the information in Section 7.1.1, let's try to determine what the best block size is for vertically flipping astronaut.bmp, which is a 7918×5376 image. There are many options to parallelize this, but, the most important aspect of the Vflip() GPU kernel is that it is memory intensive. Although —at this point— we don't know exactly what a *memory intensive GPU program* is supposed to look like, we know that Vflip() consists of nothing more than a tornado of memory transfers within the GPU itself. That being the case, we can rely on one thing: the GPU memory likes very similar access patterns to the CPU memory, as we saw in Section 3.5.3. So, it is better to access the memory in a sequential horizontal pattern, which will map to big chunks of consecutive data being pulled out of GPU memory, thereby obeying the DRAM rules that we saw in Section 3.5.3. As described in Chapter 6, we will have multiple blocks to copy one row. If you chose 256 threads/block as our *block size* —which is the number of threads per each launched block— let's see how many block each row will require:

$$\text{Hpixels} = 7918 \quad , \quad \text{NumThreads} = 256 \quad \Longrightarrow \quad \text{BlkPerRow} = \left\lceil \frac{7918}{256} \right\rceil = 31 \qquad (7.1)$$

For coherency, variables names in Eq. 7.1 are the same as Code 6.3. So, each GPU thread will copy one pixel (3 bytes) and a total of $31 \times 256 = 7936$ threads will be launched to copy one row, thereby leading to a waste of $7936 - 7918 = 18$ threads per row. Based on this computation, we will need to launch $31 \times 5376 = 166656$ blocks to flip the entire image, leading to a wasted thread count of $18 \times 5376 \approx 95$ K threads out of the total $7918 \times 5376 \approx 41$ M threads launched. This is an extremely small percentage loss and can be completely ignored.

Now, let's see what happens if we use the same idea —multiple blocks per row— on a 1920×1200 image. Each row will have 8 blocks launched (each block with 256 threads) and the waste will be 128 threads/row, totaling 150 K total wasted threads out of the total 2.2 M launched. Although still a small percentage, you can see that it is dependent on the dimensions of the image. If you, for example, change the block size to 128 threads/block instead of 256, the waste will be less, however, you will be launching twice as many blocks to execute the same code. Which one is better? Instead of guessing, it is better to run the same code with a few different block sizes to see how the block size affects the performance.

7.1.3 imflipG.cu: Interpreting the program output

An example output of the imflipG.cu, when run with the 'V' option —for vertical flip— and a block size of 256, using the astronaut.bmp image, is shown in Fig. 6.10. Here is the information displayed:

- The GPU was GeForce GTX Titan Z, which supports 2048 M blocks ($\approx$ 2 billion).
- We choose `ThePerBlk`=256, which means that we need 166656 blocks to flip the image (`NumBlocks`=166656). This is well under the 2048 M blocks supported by this GPU.
- We display `BlkPerRow`=31 to confirm what we computed in Eq. 7.1.
- The Compute Capability of the TITAN Z is 3.5. So, because we compiled our code with the 3.0 option (as detailed Section 6.5.3), this code will run fine, because 3.5 means that a GPU supports 3.5 and anything below.
- The program reports the completion time of three different events, as described in Section 6.4.3: a) The CPU→GPU transfer time is 30.89 ms, b) the kernel execution time of Vflip() is 4.21 ms, and c) the GPU→CPU transfer time is 28.80 ms.

TABLE 7.1 Vflip() kernel execution times (ms) for different size images on a GTX TITAN Z GPU. Different block sizes (**ThrPerBlk**) correspond to different total number of blocks launched (**NumBlocks**) and the total number of blocks needed to process each row (**BlkPerRow**), as tabulated. The CPU⟷GPU data transfer times are not included in this table.

ThrPerBlk	**astronaut.bmp (121 MB)** 7918 × 5376			**mars.bmp (241 MB)** 12140 × 6940
	NumBlocks	BlkPerRow	time (ms)	time (ms)
32	1333248 ≈ 1.27 M	248	12.04	22.93
64	666624 = 651 K	124	6.58	12.63
128	333312 ≈ 326 K	62	4.24	7.90
256	166656 ≈ 163 K	31	4.48	8.15
512	86016 = 84 K	16	4.64	8.53
1024	43008 = 42 K	8	5.33	9.22

7.1.4 imflipG.cu: Performance impact of block and image size

After this example run, we naturally ask ourselves the question: how would different values of `ThrPerBlk` affect the performance of the program? Furthermore, would it make a difference whether the image was large or small? To partially answer both of these questions, let's run the same program on two different images —the 121 MB astronaut.bmp (7918 × 5376) and the 241 MB mars.bmp (12140 × 6940)— with a varying number of `ThrPerBlk` values. Table 7.1 tabulates the execution time of the Vflip() kernel for different `ThrPerBlk` values, entered from the command line argument. The CPU→GPU and GPU→CPU data transfer times are not shown in Table 7.1, because we know that the `ThePerBlk` parameter wouldn't affect them. Only the Vflip() kernel execution times are shown (`case` 'V' in Code 6.3).

From Table 7.1, we observe the following:

- There is an *optimum* `ThrPerBlk` value that gives us the shortest execution time, which is 128 in this specific example.
- For both images, the performance degrades significantly for `ThrPerBlk`<128.
- For both images, the performance degrades slightly for `ThrPerBlk`>128.
- This behavior is identical, regardless of the size of the image.

Although a very detailed analysis of this concave performance curve that we see in Table 7.1 will be provided later in this chapter, for now, let's think *simple* and see what the reasons could be. Here is our logic:

- This program is memory intensive; each thread copies 3 bytes and the `ThrPerBlk` parameter we choose has a direct impact on *how many bytes each block copies.*
- Knowing this, let's calculate how different `ThrPerBlk` values translate to *number of bytes copied per block.* (i) For `ThrPerBlk`=32, each one of the ≈1.27 M blocks copies 96 bytes, while (ii) for `ThrPerBlk`=64, each one of the 651 K blocks copies 192 bytes. We know that these two are the two *bad* options.
- Alternatively, (iii) for `ThrPerBlk`=128, each one of the ≈326 K blocks copies 384 bytes, making it our *optimum* option, followed by the remaining *not so terrible* `ThrPerBlk` options of 256, 512, and 1024, yielding results that are slightly worse then the optimum.

I described in Section 6.5.3. In Mac or any other Unix computer, this Compute Capability (compute_30, sm_30) is a command line parameter for nvcc. I will use "CC" to denote "Compute Capability" going forward to shorten the text.

When you compile your code with CC 3.0, for example, you are guaranteeing that the executable application (imflipG.exe in Windows and typically imflipG in Mac and Unix) can only run with GPUs that support CC 3.0 or higher. A built-in feature of imflipG.cu queries the GPU and outputs the supported highest CC. Looking at Table 7.3, we observe that every GPU in this table supports CC 3.0 or higher (specifically, 3.0, 3.5, 3.7, and 6.1).

7.7.1 Fermi, Kepler, Maxwell, Pascal, and Volta families

Using CC 3.0 allows the code to take advantage of the CC 3.0, however it will not work on GPUs that support CC 2.0 or 2.1. Fermi family Nvidia GPUs (engine names starting with **GF**) supports CC 2.x, while the Kepler family GPUs (engine names starting with **GK**) supports CC 3.x and higher. Maxwell family (engine names starting with **GM**) supports CC 5.x. Pascal family (engine names starting with **GP**) supports CC 6.x. The upcoming Volta family (I assume that engine names starting with **GV**) will support CC 7.x. I have no clue what happened to CC 4.x ? I assume that the engine design got abandoned by Nvidia, however, this is just my humble opinion.

In Table 7.3, five out of the six GPUs belong to the Kepler family (with GK engine names). The only Pascal family GPU (with a GP engine name) supports CC 6.1; unfortunately, our compiled executable will not be able to take advantage of the additional instructions in the Pascal GPU, because we chose a command line that restricts the code execution to CC 3.0. Any higher CC is backwards compatible with CC 3.0, so, the Pascal GPU will happily execute this application, however, had I chosen "compute_61, sm_61" to compile this code, the executable wouldn't even run on the other five GPUs, however, chances are that it would execute faster in the Pascal GPU. So, when you choose a specific CC, you are instructing the compiler to use only the instructions within that CC to compile the GPU code. In a sense, if you choose CC 3.0, you are choosing something like *Kepler family or later generation families*, which would means that the code would run on Kepler, Maxwell, Pascal, and Volta, which support CC 3.x, 5.x, 6.x, and 7.x, respectively.

In many cases, if you know that your code will only execute on Pascal GPUs (or higher), it would actually be a good idea to choose CC 6.0 to compile it (even 6.1 if you want to get that aggressive), to take advantage of the additional Pascal-and-beyond instructions. One important point to understand is that when Nvidia designs a new engine family, they design it to perform better in the new CC and *anything below that CC*. In other words, even if you choose CC 3.0 to compile a code and run it on a Pascal GPU, it will potentially perform better than its older friends. Because, the two improvements Nvidia makes in every generation are: i) the introduction of a *new set of instructions* that perform operations that coudln't have been performed in the previous CC, and ii) performance improvements for all of the instructions that were introduced in the previous CC generations.

Just to provide a few examples, CC 3.x started supporting what is known as *Unified Memory*, something that didn't exist in CC 2.x. This was a major enhancement going from the Fermi family to the Kepler family. Furthermore, CC 5.3 and above started supporting half precision floating point numbers, something I will elaborate on in Section 9.3.10.

7.7.2 Relative bandwidth achieved in different families

Looking at Table 7.3, we see a good proof of what I just described. Box IV is the only Pascal GPU in the table and shows the following characteristics:

7.10.1 How is our brain involved in writing/debugging code?

There are two major categories of bugs, which are the *sneaky* bugs and *obnoxious* bugs.

Sneaky bugs hide themselves and it takes a lot of "conscious" effort to find them. They are usually a product of bad programming logic. They are very annoying, because you might think that you got them one day just to have them show their ugly face a few weeks later when different data is input into the program. They are the ones that might be difficult to catch with old school debugging. They might require you to run the code step by step, sometimes tens or even hundreds of steps in a loop until you hit a value that gives you the *gotcha* output. I found that the best way to eliminate them is to stop working on the program and start the next day with a fresh mind. As they say: there is no point in *pushing it*. Clearly you won't be able to solve this problem today.

Obnoxious bugs are good bugs. They immediately crash your program, so, it is easy to identify them. Memory pointer bugs are in this category, when you malloc() a memory area and try to step even a single byte out of that area, the OS will shut you down. Preventing one program from writing into another program's memory area is about one of the most important functions of the OS, so, your buggy program will give you a Segmentation fault or another type of memory error and will exit abruptly. The bad news about this type of a bug is that the program stops working, so, it never reaches the many printf()'s you put into the program.

When you cannot debug your code, your brain might get stuck in a *local minimum* and cannot converge to a global value. *Attention* is a limited brain resource and you will run out of it if you keep working on the same problem [30]. Everything I just described actually has a neuro-scientific reason behind it. Here is how the human brain works:

- *The human brain consists of two parts: i) conscious and ii) motor [29].*
- *Conscious part is responsible for deliberate action and requires continuous attention. Clearly, this part wasn't able to debug your code and pushing it won't help today. All you will do is drain your blood glucose, which is demonstrated to be the food of your brain and your entire nervous system*
  Motor part does more automated work, not requiring constant attention.
  Go to sleep … Try again tomorrow …
  Motor part of the human brain is known to process a difficult problem continuously, even in your sleep —actually especially *in your sleep— because it is not bothered with other daily tasks.*
- *It is common for an experienced programmer to wake up and start working on a program that had one of these sneaky bugs and fix it in 5 minutes.*
- *So, I have two Tolga's in my brain; Motor-Tolga and Conscious-Tolga.*
  When I am in front of my computer, C-Tolga is debugging, in my sleep, M-Tolga is. I don't care who debugs the code first, because they are both "me."

7.10.2 Do we write buggy code when we are tired?

A good programmer should understand how his/her brain works. Our brain is the CPU that we use during programming and debugging; it is different than a Xeon CPU or a GTX Titan GPU, because its functionality is affected by many factors that involve the entire body. Brain gets *tired*, a CPU or a GPU doesn't. Every bug will likely come back to have a cause related to one of the tiredness states listed below. Let's go a little deeper into the factors that make a programmer tired (more specifically, *a programmer's brain* tired) and increase his/her program's chances of having more bugs. There are different types of *tiredness* states that you must be aware of. Although all of them will reduce your performance, their sources are completely different; so are their remedies.

7.10.2.1 Attention

Attention is a limited resource in your brain. Unfortunately, after millions of years of development, our brain still works as a single-threaded CPU, being able to process only a single thing heavily. We cannot focus on anything more than one —and only one— thing heavily at any point in time. The conscious part of our brain, described in Section 7.10.1, is solely responsible for our attention; it performs heavy processing during code development and especially debugging. If your attention is diverted to somewhere else during code development, your programming and debugging performance will fall of a cliff.

7.10.2.2 Physical tiredness

Physical tiredness is related to your muscles, which are spread throughout your entire body. During your daily physical activity, your muscles turn ATP (Adenosine Tri Phospate) into ADP (Adenosine Di Phosphate) and AMP (Mono Phosphate), and eventually Adenosine. The ATP→ADP→AMP→Adenosine degradation is through breaking a phosphate bond, which releases the bonding energy of that phospate bond, thereby powering up your muscles. The eventual product, *Adenosine* is basically *burnt fuel*. So, if there was a detector in your body to detect the increasing Adenosine levels, it could be used to warn you about the decreasing ATP levels, much like the fuel gauge in your car and the yellow warning light telling you that you are getting close to running out of fuel. Guess what? the brain is that detector [30]. The brain detects the increasing Adenosine levels and *warns you* to avoid fully-depleting your ATP resources. This warning could be by telling you to sleep or eat. This "*eat because you are hungry*" message is a function of the *leptin* and *ghrelin* hormones, whereas the "*sleep because you are tired*" is controlled by the *melatonin* hormone. The remedy against ATP depletion is to eat proteins, fats, and carbohydrates, which will produce ATP during their break-down [39].

7.10.2.3 Tiredness due to heavy physical activity

A nice work-out at the gym is another reason for physical tiredness. You have two types of muscles in your body: i) fast twitch muscles that use *anaerobic* metabolism (*anaerobic* means "without requiring oxygen"), and your ii) slow twitch muscles that use *aerobic* processes (*aerobic* means "requiring oxygen"). When you need to perform heavy physical activity, your heart cannot supply oxygen fast enough to the slow twitch muscles, thereby requiring you to resort to using your fast twitch muscles, which do not need oxygen to work. Both of these muscles use ATP as their energy source, but they produce a different output; although fast twitch muscles produce burst energy, they produce lactate and your brain has a detection mechanism for lactate build-up in these muscles; it will try to slow you down —by communicating with you through the feeling of *pain*— to avoid damage to your body, because too much lactate build up will eventually hurt your muscles. The pain that your brain produces as a result of heavy workout will have a negative impact on your programming performance. The only remedy for this is to rest until the pain goes away, because your *attention* will be diverted away from your code towards the pain. When the lactate is out of the muscles, pain is gone and you are back to work.

7.10.2.4 Tiredness due to needing sleep

You get tired when you don't get enough sleep, because there is a small part of your brain that is responsible for detecting the 24-hour circadian cycle. It is your brain's clock, which

is responsible for releasing the *melatonin* hormone to prepare you for sleep and increasing *cortisol* levels when it is time to wake you up. During sleep, your ATP levels are replenished and you are ready to burn them back to Adenosine again. The only remedy for sleep deprivation is to sleep! There is no way to fight the brain! Just get a good night sleep and you will be the best debugger again tomorrow.

7.10.2.5 Mental tiredness

MEntal tiredness is related to your neurons, which are spread throughout your entire Central Nervous System (CNS). Neurons burn glucose when they work. To state simplistically, the *brain food* is *sugar*. So, when your blood glucose levels go lower, your neurons do not have immediate energy. This is no different than your computer consuming so many Watts per FLOP. For example, GTX 1070 consumes 110 W of power when it works at its peak, although it consumes way less than that during normal operation. Human brain consumes an equivalent of 20 W on average. However, during a heavy-duty debugging session, when you are trying super hard to find a bug in your code, I am sure that this number goes way up, causing a higher rate of depletion in glucose.

More debugging → more neural activity → more glucose consumption

There is the other side of the story. Your neurons use *neurotransmitters* to carry neural signals. The leftover neurotransmitters are mopped up and placed back into you system when you *rest*. So, if you work very hard, your neurons will keep using sugar and the leftover neuro-garbage must be collected, which takes time. So, a nice walk along a river can have a mentally-replenishing effect, although not necessarily comparable to consuming sugary foods ! This is more like your neurons malloc() energy with sugar and the rest of the system has to free() the garbage out of your system, otherwise your brain will issue a Segmentation fault. This is when you stare at the screen and nothing happens! You need a reboot. Go get a nice walk...

CHAPTER 8

Understanding GPU Hardware Architecture

In the previous two chapters we looked at the structure of a CUDA program, learned how to edit, compile, and run a CUDA program, and analyzed the performance of the compiled executable on different generation of GPUs, which have different Compute Capabilities (CCs). We noted that a CPU is good for *parallel computing*, which has the building block named *thread*; it is usual to expect the CPU-based parallel programs to execute 10, 20, 100, even 1000 threads at any point in time. However, a CUDA program (more generally, a GPU program) is suitable for problems that can take advantage of *massively parallel computing*, which implies the execution of hundreds of thousands or even millions of threads at a time. To allow the execution of such an enormous number of threads, GPUs had to add two additional hierarchical organization of threads:

1. A **Warp** is a clump of 32 threads, which is really the minimum number of threads you can break your tasks down to; in other words, nothing less than 32 threads executes in a GPU. If you need to execute 20 threads, too bad, you will be wasting 12 threads, because a GPU is not designed to execute such a "small" number of threads.

2. A **Block** is a clump of 1 to 32 warps. In other words, you launch your program as 1, 2, 3, ..., or 32 warps, corresponding to anywhere from 32 to 1024 threads. A block must be designed as an isolated set of threads that can execute independently from the other blocks. If you design your program to have such smooth *separation* (or, *independence*) from other blocks, you will achieve blazing parallelism and will take advantage of the GPU parallelism.

Clearly not every problem is amenable to massive-parallelism. Image processing, or more generally, Digital Signal Processing (DSP) problems are natural candidates for GPU massive parallelism, because i) you apply the same exact computation to different pixels (or pixel groups), where one pixel can be computed independently from another, and ii) there are typically hundreds of thousands or millions of pixels in an image we encounter in today's digital world. Following this understanding from the previous CUDA chapters, in this chapter, we now want to understand how the GPU achieves this parallelism in hardware.

In this chapter, we will introduce the GPU edge detection program (imedgeG.cu) and run it to observe its performance. We will relate this performance to the building blocks of the GPU, such as the GPU cores and Streaming Multiprocessors (SM), which are the execution units that house a bunch of these GPU cores. We will also study the relationship among SM, GPU cores, and the things we have just learned, thread, warp, and block over the course of a few chapters, during which we will learn the *CUDA Occupancy Calculator*, which is a simple tool that tells us how "occupied" our GPU is, i.e., how busy we are keeping

it with our program. This will be our primary tool for crafting efficient GPU programs. The good news is that although writing efficient GPU programs is an art as much as a science, we are not alone! Throughout the following few chapters, we will learn how to use a few such useful tools that will give us a good view of *what is going on inside the GPU*, as well as during the transfers between the CPU and the GPU. It is only with this understanding of the hardware that a programmer can write efficient GPU code.

8.1 GPU HARDWARE ARCHITECTURE

So far, we wrote GPU code and tweaked the *threads per block* parameter to observe the changes in performance. In this chapter, we will look at the hardware at a high level to get an idea about how the cores and memory are organized and how the data flows inside the GPU. Remember from Analogy 6.1 (Fig. 6.3) that we viewed the GPU as a school bus with 32 scouts in it; the scouts represented the slower GPU cores who got more work done than any single- or dual-core (or even quad-core) CPU architecture due to their shear quantity, as long as the application was massively-parallel computing friendly.

One very important note about Fig. 6.3 is that we had somebody, **Tolga**, to organize things inside this bus, which was such an intense task that Tolga couldn't do any useful work; he was dedicated to organizing things. Let's remember: *what did Tolga organize ?* The answer is: *data flow and task distribution to scouts*. Because the scouts are inexperienced executers, somebody had to tell them exactly what to do and put the data right in front of them to do it. The big question is, we do not see anything too impressive being done with only 32 scouts; we need 1000s of them to beat the CPU performance. In real-life GPUs, as exemplified in our Table 8.3, GTX 1070 has 1920 GPU cores and the GTX Titan Z has 5760 cores. Is it even possible to expand our scout analogy to cover such a huge number of cores? The answer is Yes, but, major additional organizational structures will be needed.

8.2 GPU HARDWARE COMPONENTS

Analogy 8.1 (Fig. 8.1) is exactly what the is required to be able to process Gigabytes of data using thousands of cores. None of these components can be missing, because the execution will be a mess otherwise! Let us now see what each person (and object) corresponds to inside the GPU. Note: I gave a little bit of a hint in the caption of Fig. 8.1.

8.2.1 SM: Streaming Multiprocessor

In Analogy 8.1, each school bus, including the small barrel and the box (to receive the instructions) is equivalent to an SM (Streaming Multiprocessor) inside the GPU. This was the term that Nvidia used in their older GPUs; for example, in the Fermi family, the GTX550Ti GPU (with 192 cores, as shown in Table 7.6) had 32 GPU cores in each SM. Later, Nvidia changed this term to SMX in the Kepler family, SMM in the Maxwell family, and back to SM in the Pascal family, however, Pascal introduced another hierarchical structure named GTC, as we will see very shortly.

it, it wouldn't even consider taking another block, (ii) if it can still take more, it compares the parameters of the kernel you are advertising to its own parameters and see if it has "resources" to take this new block. Resources include a lot of things, such as cache memory, register file, among many others as we will see in Chapter 9.

Back to our example: By this time, each SM absorbed the following blocks:

- *SM0* $\Longrightarrow$ [*Block0*, *Block6*, *Block12*, *Block18*, *Block24*, *Block30*, *Block36*, *Block42*]
- *SM1* $\Longrightarrow$ [*Block1*, *Block7*, *Block13*, *Block19*, *Block25*, *Block31*, *Block37*, *Block43*]

  ...
- *SM5* $\Longrightarrow$ [*Block5*, *Block11*, *Block17*, *Block23*, *Block29*, *Block35*, *Block41*, *Block47*]

Now that you have scheduled 48 blocks (*Block0...Block47*), you have to wait for somebody to be free again to continue with the other 166608 blocks (*Block48 ... Block166655*). It is not necessarily true that the SMs will finish the execution of the blocks assigned to them in exactly the same order of assignment. At this moment, you have each SM executing 8 blocks, but they can only execute one at a time, and put 7 to sleep temporarily. When a block accesses resources that will take a while to get (say, some data from Global Memory), it has the option to switch to another block to avoid staying idle. This is why you stuff 8 blocks to the SM and give it 8 options to choose from to keep itself busy. This concept is identical to why assigning two threads to the CPU helped it *do more work* on average, although it could not execute more than one of those threads at a time. In the case of the SM, it grabs a bunch of blocks, so, it can switch to another one when one comes to a standstill.

Now, let's fast forward the time a little bit. Say, *SM1* got finished with *Block7* before anybody else. It would immediately raise its hand and volunteer to take in another block. Having gotten rid of 0...47, your next block to schedule is 48. So, you would make the following scheduling decision: *Block48*→*SM1*. After this assignment, *SM1* would clean up all of the resources it needed for *Block7* and replace it with *Block48*. So, *SM1*'s queue of 8 blocks is looking like this now:

- *SM1* $\Longrightarrow$ [*Block1*, ***Block48***, *Block13*, *Block19*, *Block25*, *Block31*, *Block37*, *Block43*]

Let's say that *SM5* finished *Block23* next and raise its hand; you would assign your next block (*Block49*) to it, which will change its queue to the following:

- *SM5* $\Longrightarrow$ [*Block5*, *Block11*, *Block17*, ***Block49***, *Block29*, *Block35*, *Block41*, *Block47*]

This would continue until you finally assigned *Block166655*. When you assign this very last block, GTS's responsibility is over. It might take a while to finish what is in the queue of each SM after this very last assignment, but, as far as the GTS is concerned, job is done !

8.9.6 Executing blocks

Now that we understand how the blocks are scheduled to SMs for execution, let's understand how they are executed. Assume that you are *SM5* and *Block49* is scheduled to you for execution. Here is what you receive from the GTS:

- `gridDim.x`=166656, `gridDim.y`=1, `gridDim.z`=1,
- `blockDim.x`=256, `blockDim.y`=1, `blockDim.z`=1,

- `blockIdx.x`=49, `blockIdx.y`=0, `blockIdx.z`=0.

This is enough for you to understand that, out of the 166656 blocks, you are #49. Because each block consists of 256 threads, you must execute this block using 256 threads. So, the SMs responsibility is to execute Block49 using 256 —single-dimensional— threads, numbered `threadIdx.x`=0...255. This means that it will facilitate the execution of 256 threads, where each threads gets the exact same parameters above; additionally, they also get their `threadIdx.x` computed and passed onto them. So, if you are thread #75 out of the 256 threads, this is what is passed on to you when you are executing:

- `gridDim.x`=166656, `blockDim.x`=256, `blockIdx.x`=49, `threadIdx.x`=75.

I omitted the parameters with values "1" because the programmer will not even look at them and they will not be used during the execution anyway. To summarize, the responsibility of the GTS ends when the block is scheduled and the responsibility of the SM itself starts, which is further numbering the threads before starting the execution of the block. Of course, in addition to assigning the thread ID, the shared resources in an SM, such as cache memory, register file, and more have to be allocated. More on that later. In this chapter, all we care about is the scheduling functionality at the block level.

8.9.7 Transparent Scalability

As you see, the only responsibility of the programmer is to write the CUDA code in such a way that each block is a highly independent code with no dependence on the other blocks. If the code is written this way, the hardware details are transparent to the programmer and the more SMs a GPU has, the faster your code can execute. This is called *transparent scalability*. After all, the last thing a GPU user wants is to purchase a GPU with more cores (which really corresponds to more SMs) and have his/her program barely benefit from the additional SMs. Ideally, the speed-up should be linear with the increasing number of cores.

CHAPTER 9

Understanding GPU Cores

In the previous chapters, we looked at the GPU architecture at a high level; we tweaked the *threads per block* parameter and observed its impact on the performance. The readers should have a clear understanding by now that the unit of kernel launch is a *block*. In this chapter, we will go much deeper into how the GPU actually executes the blocks. As I mentioned at the very beginning of Part II, the unit of execution is not a block; rather, it is a *warp*. You can think of the block as the big task to do, which can be chopped down into much smaller sub-tasks named warps. The significance of the warp is that a smaller unit of execution than a warp makes no sense, because it is too small considering the vast parallelism the GPU is designed for.

In this chapter, we will understand how this concept of warp ties to the design of the GPU cores and their placement inside a Streaming Multiprocessor (SM). With this understanding, we will design many different versions of the kernels inside the imflipG.cu and imedgeG.cu programs, run them, and observe their performance. We will run these experiments in four different GPU architecture families: Fermi, Kepler, Maxwell, and Pascal. With each new fmaily, a new instruction set and compute capability have been introduced; to accommodate these new instructions sets, the cores and other processing units had to be designed differently. Because of this fact, some of the techniques we will learn in this chapter will be broadly applicable to every family, while some of them will only work faster in the new generations, due to the utilization of the more advanced instructions, available only in the newer generations such as Pascal.

While we "guessed" the *threads per block* parameter in the previous chapters, we will learn how to use a tool named *CUDA Occupancy Calculator* in the next chapter, which will allow us to establish a formal methodology for determining this important parameter, along with many other critical parameters that ensure optimum utilization of the SM resources during kernel execution.

9.1 GPU ARCHITECTURE FAMILIES

In Table 8.1, we briefly introduced the architectural components of each family. In this chapter, we are interested in the details of each family's internal architecture and how we can utilize this knowledge to write higher performance GPU code. First, we want to know the organization of the SMs inside each family.

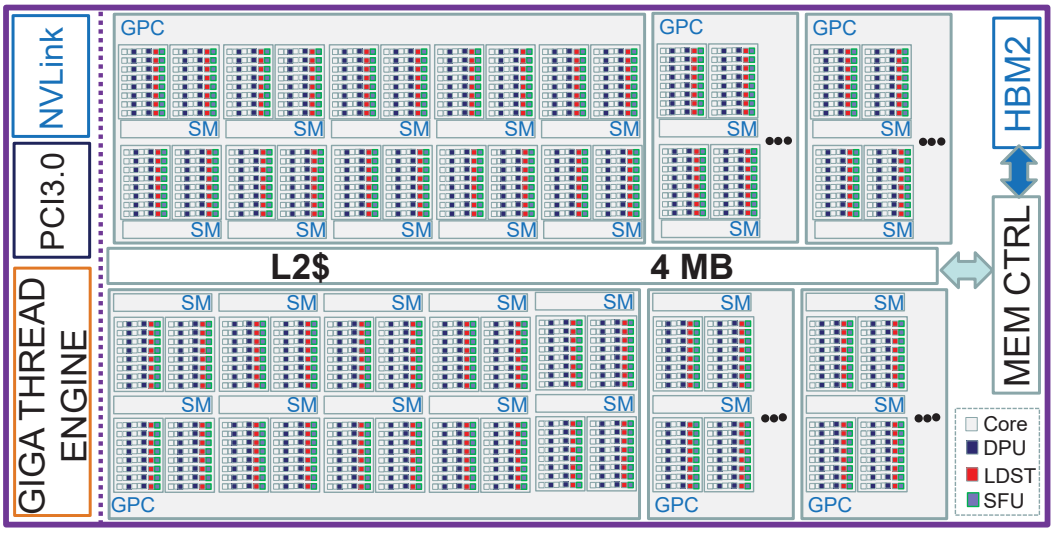


FIGURE 9.7 GP100 Pascal architecture with 60 SMs, housed inside 6 larger GPC units, each containing 10 SMs. The highest end Pascal GPU contains 3840 cores (e.g., P100 compute accelerator). NVLink and High Bandwidth Memory (HBM2) allow significantly faster memory bandwidths as compared to previous generations.

9.1.7 Pascal GP100 architecture

Pascal architecture is shown in Fig. 9.7. Noticeable changes are:

- The L2$ is 4 MB, although it is only 2 MB in GTX 1070.
- The total amount of Global Memory is 12 GB for Titan X, 8 GB for GTX 1070, and 16 GB for the compute accelerator P100. This is slightly larger than the Kepler and Maxwell generations.
- K80 was a two-in-one GPU, containing two K40's inside it. Although a double-P100 is not available as of mid-2017, you can expect it to be available in late 2017, which will possibly incorporate a 12 GB or 16 GB per GPU, with a total of 24 GB or 32 GB. Having such a large amount of global memory helps in heavy computations that require the storage of a significant amount of data inside the GPU. For example, in the emerging *Deep Learning* application a large GPU memory is necessary to store the "neural" nodes.
- The most drastic change in the Pascal microarchitecture is the support for the emerging High Bandwidth Memory (HBM2). Using this memory technology, Pascal is able to deliver a whopping 720 GBps by using its ultra-wide 4096-bit memory bus. In comparison, K80 GM bandwidth is 240 GBps for each GPU using a 384-bit bus.
- The next major change is the support for a new type of bus that is introduced by Nvidia: the NVLink bus, which boasts a 80 GBps transfer rate. This is much faster than the 15.75 GBps the PCIe 3.0 bus and alleviates the CPU$\longleftrightarrow$GPU data transfer bottleneck. However, NVLink is only available on high-end servers.

be assumed to execute independently. As we will see in Chapter 10, this will force us to use explicit synchronization for memory reads that occur at intra-warp boundaries. For now, the reader does not need to worry about this.

9.3 PARALLEL THREAD EXECUTION (PTX) DATA TYPES

In this section, we will look at the data types as they are defined in the Parallel Thread Execution (PTX) instruction set of the Nvidia GPUs. Understanding the assembly language of Nvidia (PTX) is important in getting a sense for how the GPU cores execute the instructions in a kernel. Remember that the first PTX was PTX 1.0, which was introduced in 2009. The newest PTX 5.0 was introduced in January 2017, which is only supported by Pascal GPUs. Most of the data types and arithmetic, logic, and floating point operations stayed the same while others were added to provide support for emerging applications such as deep learning. When I introduce each data type, I will provide a set of PTX instructions that use that data type. We will use this insight to improve our kernels in the following few chapters.

9.3.1 INT8: 8-bit Integer

These are the 8-bit integers as defined in PTX:

.u8 PTX type is the unsigned 8-bit integer (range $0 \cdots 255$)

.s8 PTX type is the signed 8-bit integer (range $-128 \cdots 127$)

.b8 PTX type is the untyped 8-bit integer

Example PTX instructions for the INT8 data type are as follows:

```
add.u8 d, a, b;    // add unsigned 8-bit integers a to b, save result in d
```

Here are two new instructions that work on "vector" 8-bit data:

```
dp4a.u32.u32 d,b,a,c;    // d=c+four-way dot product of bytes of a,b
dp2a.lo.u32.u32 d,b,a,c; // d=c+two-way dot product of words of a, Low bytes of b
```

Note that these two instructions are only available in PTX ISA 5.0 (i.e., Pascal family), which allow the processing of four bytes in one clock cycle, or two 16-bit words. Because of this, the Pascal family can achieve 4× the performance in processing byte-size data, as long as the code is compiled to take advantage of these instructions. As you can see here, Nvidia's architecture design trend is to turn their integer cores into more like the MMX and SSE, and AVX units that the i7 CPUs have. With, for example, the Intel AVX instruction extensions, it is possible to process a 512 bit vector as either 8 64-bit numbers, 16 32-bit integers, 32 16-bit, or 64 8-bit integers. The `dp4a` and `dp2a` instructions resemble this a little bit. My guess is that in the Volta family (the next generation after Pascal), there will be a much wider set of these instructions, potentially applicable to other data types.

9.3.2 INT16: 16-bit Integer

These are the 16-bit integers as defined in PTX:

.u16 PTX type is the unsigned 16-bit integer (range $0 \cdots 65536$)

.s16 PTX type is the signed 16-bit integer (range -32768 ··· 32767)

.b16 PTX type is the untyped 16-bit integer

Example PTX instructions for the INT16 data type are as follows:

```
min.u16 d, a, b;          // store the minimum of unsigned int16 numbers a,b in d
min.s16 d, a, b;          // same as above, but assumes that a and b are signed int16
mul.u16.lo d, a, b;       // unsigned multiply 16-bit a,b and store low 16-bit in d
mul.u16.wide d, a, b;     // unsigned mult. 16-bit a,b and store 32-bit result in d
mad.hi.sat.s32 d,a,b,c;   // signed mul a,b. add c, saturate; store hi-32 bits in d
```

9.3.3 24-bit Integer

There was a 24-bit integer data type in the early days of PTX 1.0. This was necessary because supporting 32-bit native operations would mean that every 32-bit multiplication would be forced to save its results in a 64-bit destination. However, if you multiply two 24-bit numbers, you get a 48-bit result and if the numbers are small enough, the result may actually fit in 32-bits; based on this, 24-bit multiplication instructions allowed one to save either the upper or lower 32 bits of the result. That way, you can use the lower 32 bits if you know that your numbers are small to start with, Alternatively, you can use the upper 32 bits if you were storing fixed point numbers and the lower bits only mean more resolution and can be ignored. If you cannot live without all 48 bits of the result, you can always perform both of the multiplications and save both of the results for future use.

Example PTX instructions for the 24-bit data type are as follows:

```
mul24.hi.u32 d,a,b;         // unsigned multiply 24b a, b; save higher 32b in d
mul24.lo.s32 d,a,b;         // signed multiply 24b a, b; save lower 32b in d
mad24.hi.u32 d,a,b,c;       // multiply a, b and add c. Save the result in d
mad24.hi.sat.s32 d,a,b,c;   // saturating signed multiply a, b, add c, save in d
```

9.3.4 INT32: 32-bit Integer

These are the 32-bit integers as defined in PTX:

.u32 PTX type is the unsigned 32-bit integer (range 0 ··· 2^{32}-1)

.s32 PTX type is the signed 32-bit integer (range -2^{31} ··· 2^{31}-1)

.b32 PTX type is the untyped 32-bit integer

Example PTX instructions for the INT32 data type are as follows:

```
rem.u32 d,a,b;              // d=remainder of integer division of u32-type a/b
mad.lo.u32 d,a,b,c;         // d=a*b+c (lower 32 bits)
abs.s32 d,a;                // d=absolute value of a. Only applies to signed types
popc.b32 d,a;               // d=population count (number of 1s) in a
add.sat.s32 d, a, b;        // saturating add s32 a, b and save result into d
ld.global.b32 f, [addr];    // load a 32-bit value from memory into f register
```

Note that a "saturating" addition avoids overflow by limiting the result to the MININT···MAXINT range; it only applies to the `s32` type. For example, adding 1 the highest number (2^{31}-1) would cause an overflow, because the result (2^{31}) is an out-of-range `s32` value. However, `add.sat` limits the result to MAXINT ($2^{31}-1$) and voids overflow. This is perfect for Digital Signal Processing applications, where a lot of filter coefficients and sampled voice or image data are being multiplied. The inaccuracy caused by the saturation is inaudible to the human ear, but avoiding overflow prevents the results from being completely wrong and meaningless and outputting white noise like garbage at the output of the filter.

9.3.5 Predicate Registers (32-bit)

This is the predicate type as defined in PTX:

.pred PTX type is a 32-bit predicate register

Example PTX instructions for the predicate registers are as follows:

```
     .reg .pred p,q,r;       // declare p, q, r as predicate registers
     setp.lt.s32 p, a,b;     // p= (a<b);
@p   add.s32 c,c,2;          // c+=2 if p is True (i.e., if a<b);
```

Here, the `@p` is the guard predicate, which executes the conditional add instruction based on the Boolean value of the `p` predicate register. The reverse of the predicate can also be used for conditional instructions, as follows:

```
     setp.lt.s32 p, a,b;   // p= (a<b);
@!p  bra OUT;              // branch if predicate p is false (i.e., a>=b)
     mul...
     ...
OUT:
```

9.3.6 INT64: 64-bit Integer

These are the 64-bit integers as defined in PTX:

.u64 PTX type is the unsigned 64-bit integer (range 0 ··· 2^{64}-1)

.s64 PTX type is the signed 64-bit integer (range -2^{63} ··· 2^{63}-1)

.b64 PTX type is the untyped 64-bit integer

Example PTX instructions for the INT64 data type are as follows:

```
rem.s64 d.a.b;     // d=remainder of integer division s64-type a/b
abs.s64 d,a;       // d=absolute value of a. Only applies to signed types
clz.b64 d,a;       // d=leading zeros in the 64 bits of a
bfind.s64 d,a;     // d=position of the most significant non-sign bit
bfe.u64 d,a,b,c;   // d=bit field extract(a) at position b, length c
bfi.b64 f,a,b,c,d; // f=aligned,inserted bitfield from a into b, start@d,length=d
```

9.3.10 FP16: Half Precision Floating Point (`half`)

Half precision floating point numbers were introduced with the IEEE754-2008 standard. PTX ISA 4.2 (Compute Capability 5.3) introduced the following instructions to support half-precision floating point in CUDA:

```
fma.ftz.f16 d,a,b,c;     // d=a*b+c (all half precision). ftz=flush to zero.
fma.ftz.f16x2 d,a,b,c;   // d=a*b+c (a, b, c are an array of two half-precision)
mov.b32 f, (h0,h1);      // pack h0,h1 (half's) into a f (float)
ld.global.b32 f, [addr];  // load a packed f16x2 into f (32-bit) from memory
cvt.rn.f16.f32 h,f;      // down-convert from float to half
```

Here, we see the ability of the GPU to do "packed" computations (i.e., two additions in one instruction). This is somehow similar to the `dp4a` instruction that allows the addition of 4 bytes in one instruction.

9.3.11 What is a FLOP?

Recall from Table 9.1 that we quantified the theoretical computational peak of multiple GPUs using the metric **G**iga **FL**oating point **OP**erations per **S**econd (GFLOPS). This begs the question: *What is a FLOP ?* Considering the fact that a GPU's core (or the FPU inside the core of a CPU) is capable of a Fused Multiply Accumulate (FMA) in a single instruction, what should we call a floating point "operation" ? The answer is FMA.
In other words, if you executed 1 billion floating point additions, 1 billion multiplications, and 1 billion FMA's in a second, you computed 3 GFLOPS. With the FMA, you buy one (multiplication) and get one (addition) free! So, you don't get additional bonus points for executing these two operations in one instruction.

9.3.12 Fused Multiply-Accumulate (FMA) vs. Multiply-Add (MAD)

The multiple-add instruction (`mad`) we saw before was introduced in PTX 1.0 and the newer fused multiply-accumulate (`fma`) instruction was only available with the later PTX versions; the FP64 version for `double` data types (`fma.f64`) was available in PTX1.4, the FP32 version for `float` data types (`fma.f32`) became available in PTX 2.0, while the FP16 version for the `half` data types (`fma.f16` and `fma.f16x2`) became available in PTX 4.2, as mentioned in Section 9.3.10. Also some `mad` instructions have exceptions, the difference is in the way the rounding is done. There are two possible ways of rounding for the general multiply-accumulate operations:

- **Double rounding**: the multiply-accumulate operation is computed as follows:

$$d \;=\; a \times b + c \;=\; \text{Round}(\text{MultiplyAndRound}(a, b) + c) \tag{9.4}$$

- **Single rounding**: the "fused" multiply-accumulate operation is computed as follows:

$$d \;=\; a \times b + c \;=\; \text{Round}(\text{Multiply}(a, b) + c) \tag{9.5}$$

The difference is that while the MultiplyAndRound operation rounds the resulting number to the precision of the operands, which reduces the intermediate resolution, the Multiply operation produces a result that has infinite resolution. Thus, the `fma` family operations prevent the accuracy loss twice. In modern CPUs and GPUs, `fma` is the only type of operation that makes sense to use, while the double rounding is somehow deprecated.

CODE 9.2: imflipGCM.cu Hflip2() {...}

Hflip2() avoids computing `BlkPerRow` and `RowBytes` repeatedly.

```
// Improved Hflip() kernel that flips the given image horizontally
// BlkPerRow, RowBytes variables are passed, rather than calculated
__global__
void Hflip2(uch *ImgDst, uch *ImgSrc, ui Hpixels, ui BlkPerRow, ui RowBytes)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;

   //ui BlkPerRow = CEIL(Hpixels,ThrPerBlk);
   //ui RowBytes = (Hpixels * 3 + 3) & (~3);
   ui MYrow = MYbid / BlkPerRow;
   ui MYcol = MYgtid - MYrow*BlkPerRow*ThrPerBlk;
   if (MYcol >= Hpixels) return;   // col out of range
   ui MYmirrorcol = Hpixels - 1 - MYcol;
   ui MYoffset = MYrow * RowBytes;
   ui MYsrcIndex = MYoffset + 3 * MYcol;
   ui MYdstIndex = MYoffset + 3 * MYmirrorcol;

   // swap pixels RGB @MYcol , @MYmirrorcol
   ImgDst[MYdstIndex] = ImgSrc[MYsrcIndex];
   ImgDst[MYdstIndex + 1] = ImgSrc[MYsrcIndex + 1];
   ImgDst[MYdstIndex + 2] = ImgSrc[MYsrcIndex + 2];
}
```

9.4.1 Hflip2(): Pre-computing kernel parameters

Our first idea is very straightforward. It really has nothing to do with the cores. It has something to do with the efficiency in *passing function arguments* into a GPU kernel. Let us look at Code 6.8, where we introduced the Hflip() kernel), to see what the kernel does as its first few steps:

```
   ui BlkPerRow = (Hpixels + ThrPerBlk - 1) / ThrPerBlk; // ceil
   ui RowBytes = (Hpixels * 3 + 3) & (~3);
```

Note that the first line above is equivalent to `ui BlkPerRow = CEIL(Hpixels,ThrPerBlk);` This is a simple pre-computation of the `BlkPerRow` and `RowBytes` values, which can be computed from what is already passed into the kernel (`Hpixels`), as well as what can be obtained from the special registers (in this case, `ThrPerBlock` can be obtained from the special register `blockDim.x`). Although `ThrPerBlock` is needed later when `MYcol` is being computed, do we really need to compute `BlkPerRow` and `RowBytes` inside the kernel? Remember that anything we calculate inside the kernel is computed for ***every single thread***. What happens if we simply pass them as function arguments? This is possible, because their values never change during the execution of any of the threads. So, instead of computing it millions of times, why not compute it once inside main() and pass it as a function argument? If we look at

CHAPTER 10

Understanding GPU Memory

Remember from the previous chapters that we introduced terms such as *memory friendly* and *core friendly* and tried to make our programs one or the other. The reality is that they are not independent concepts. Consider our GaussKernel(), which is a *core-intensive* kernel. It is so core-intensive that it is pointless to try to make it memory friendly. As a quantitative example, assume that this kernel is spending 10% of its time in memory accesses and 90% of its time in core computations. Let us assume that you made the kernel much more memory-friendly by making memory accesses 2× faster. Now, instead of memory and core taking 10+90 units of time, respectively, they will take 5+90 units of time; you just made your program 5% faster! Instead, if you tried to make the core accesses 2× faster, your program would take 10+45=55 units of time, which would make it 45% faster. So, does this mean that we should pick one or the other and not bother with the other one? Not really. Let us continue the same example. Assume that your memory+core time was 10+90 units and you applied tricks that could make core accesses 6× faster, which would drop your execution time to 10+15=25 units and make your kernel 4× faster overall. Now, assume that you can still apply the same memory-friendly techniques to this kernel and make memory accesses 2× faster, which would drop your execution time to 5+15=20 units. Now, instead of a puny 5% improvement, the same memory-friendly technique can make your program 20% faster. The moral of the story is that the reason for the initial improvement to look weak was because your core accesses were very inefficient and were masking the potential improvements due to memory-friendliness. This is why memory and core optimizations should be viewed as a "co-optimization" problem, rather than individual —and unrelated— problems.

In this chapter, we will study the memory architecture of different Nvidia GPU families and improve our kernels so their access to the data in different memory regions is efficient, i.e., we will make them memory-friendly. As I mentioned in the previous chapter, we will learn how to use a very important tool named *CUDA Occupancy Calculator* in this chapter, which will allow us to establish a formal methodology for determining kernel launch parameters to ensure optimum resource utilization inside the SMs of the GPU.

10.1 GLOBAL MEMORY

When you are purchasing a GPU for gaming or scientific computing, what do you look at? Possibly the model name of the GPU (e.g., GTX 1080), how much memory it has (e.g., 8 GB for the GTX 1080), and possibly the number of cores or the advertised GFLOPS/TFLOPS. More savvy buyers will also look at the *type* of memory the GPU has and may be even specific L1$, L2$ parameters, etc. For example, while the GTX Titan X Pascal GPU has a 480 GBps memory bandwidth due to its GDDR5X memory type, the GTX 1070 only has a 256 GBps bandwidth, due to its lower-bandwidth GDDR5 memory type. In either case,

which memory is being advertised? The answer is the *Global Memory* (abbreviated GM), which is the main memory of the GPU.

Starting with Section 3.5, we spent quite a bit of time talking about CPU memory and how *accessing consecutive large chunks* was the best way to read data from the CPU's main memory. The GPU memory is surprisingly similar and almost every bit of intuition you gained about how to efficiently access CPU memory will apply to GPU memory. The only big difference is that because a significantly higher number of cores —as compared to a CPU— need to be fed data simultaneously from the GPU memory, GDDR5 (and the newer GDDR5X) are designed to provide data to multiple sources a lot more efficiently.

10.2 L2 CACHE

L2$ is where all of the data read from GM is cached. L2$ is *coherent*, meaning that an address in L2$ means exactly the same address to every core in the GPU. So far, every Nvidia GPU architecture we looked at had an L2$ as its Last Level Cache (LLC): Fermi (Fig. 9.2) had a 768 KB L2$, while Kepler (Fig. 9.4) had 1.5 MB. Maxwell (Fig. 9.6) increased its L2$ to 2 MB, while Pascal (Fig. 9.8) enjoys a large 4 MB L2$. While GK110, GM200, and GP100 represent the biggest architectures in their respective family, smaller (scaled-down) versions were released also; for example, although GTX 1070 is a Pascal family GPU, it only has a 2 MB L2$ because this is what the GP104-200 engine includes.

Table 10.1 lists some example GPUs and their Global Memory and L2$ sizes. Additionally, a new metric, *bandwidth per core*, is shown to demonstrate how many much memory a GPU has, relative to its number of cores. Bandwidth per core was ≈0.1–0.17 GBps per core in the Kepler family and went down to about 0.08 in Maxwell and came back up to 0.13 GBps per core in Pascal. A comparison to two CPUs is shown below the thick line in Table 10.1. A CPU is designed to have nearly 50× more bandwidth allocated per core (e.g., 0.134 vs. 6.4 GBps per core). Similarly, a CPU-based PC enjoys nearly 1000× more main memory per core (e.g., 4.27 vs. 4096). Comparing the LLCs (which is L3$ in the CPU), a CPU is, again, equipped with 2000× more (e.g., 1.14 vs. 2560). This shouldn't come as a surprise, because the CPU architecture is significantly more sophisticated, allowing the CPU cores to do a lot more work. However, to deliver this performance the CPU cores need a lot more resources.

10.3 TEXTURE / L1 CACHE

In some generations, texture cache is separate from L1$, but in Maxwell (Fig. 9.6) and Pascal (Fig. 9.8) they share the same cache area. In this book, we will not do any game design, so, we are not interested in texture memory. However, the L1$ in each SM works in exactly the same way a CPU L1$ works. The data that the cores need immediately are cached in L1$ by the hardware. The user has no say in which data gets thrown out or kept in cache. In this regard, L1$ is called the *hardware cache*, or, more clearly *hardware controlled cache*. L1$is *not coherent*, meaning that once the cores read some data from L2$ and place it in their local L1$, the address they use to refer to that data in their local L1$ has no relationship to the other L1$ memories in other SMs. In other words, L1$ is strictly to improve access to specific data, rather than to share it with other L1$ memories. In the Fermi and Kepler families, L1$ is co-located with the Shared Memory, whereas in Maxwell and Pascal families, they are separate entities.

TABLE 10.1 Nvidia microarchitecture families and the size of Global Memory, L1$, L2$ and shared memory in each one of them. Below the thick line, the same parameters are shown for two different CPUs. Note: /C means *per core*.

Model (Engine)	**#cores**	**Sh. Mem**	**L1$**	**L2$**	**Global Memory**	
		(KB/core)			**(MB/C)**	**(GBps/C)**
GTX550Ti (GF110)	192	64/32 (2.00 combined)		256/192 (1.33)	1024/192 (5.33)	98.5/192 (0.513)
GTX 760 (GK104)	1152	64/192 (0.33 combined)		768/1152 (0.67)	2048/1152 (1.78)	192/1152 (0.167)
Titan Z (2xGK110)	2x2880	64/192 (0.33 combined)		1536/2880 (0.53)	6144/2880 (2.13)	336/2880 (0.117)
Tesla K80 (2xGK210)	2x2496	112/192 (0.58 combined)		1536/2496 (0.62)	12288/2496 (4.92)	240/2496 (0.096)
GTX 980Ti (GM200)	2816	96/192 (0.50)		2048/2816 (0.73)	6144/2816 (2.19)	224/2816 (0.080)
GTX 1070 (GP104-200)	1920	96/128 (0.75)	48/128 (0.38)	2048/1920 (1.07)	8192/1920 (4.27)	256/1920 (0.134)
Titan X (GP102)	3584	64/64 (1.00)		4096/3584 (1.14)	12288/3584 (3.43)	480/3584 (0.134)
Xeon E5-2690 (Sandy Br EP)	8	**L3$** (2560)	64/1 (64)	256/1 (256)	32768/8 (4096)	51.2/8 (6.400)
E5-2680v4 (Broadwell)	14	**L3$** (2560)	64/1 (64)	256/1 (256)	262144/14 (18724)	76.8/14 (5.486)

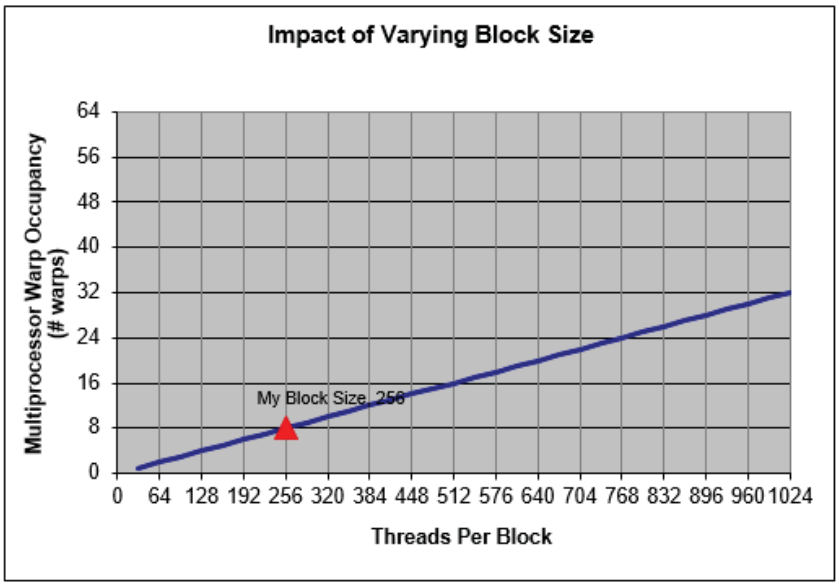
10.4 SHARED MEMORY

The most important type of memory in a GPU is the Shared Memory. Although this memory works similarly to L1$, it is purely controlled by the programmer. The GPU hardware has no say in which data goes into Shared Memory. For this reason, Shared Memory is called *software cache* or *Scratch pad Memory*. The idea behind Shared Memory is that nobody knows which data elements a program will need —throughout its execution— better than the program developer. So, if the programmer is given the opportunity to cache certain data elements whenever necessary and evict them whenever they are not needed anymore, the efficiency of data caching can go up to 100%, as compared to even the best cache replacement algorithms, which achieve 80–90% efficiency.

10.4.1 Split vs. Dedicated Shared Memory

As seen from Table 10.1, Fermi and Kepler families placed the L1$ and Shared Memory in the same memory area, which is "split" between the two based on kernel demand as follows:

- Fermi : (Shared Memory, L1$) : (16 KB, 48 KB) , (48 KB, 16 KB)
- Kepler: (Shared Memory, L1$) : (16 KB, 48 KB) , (32 KB, 32 KB) , (48 KB, 16 KB)



					= Allocatable
40	**Allocated Resources**		Per Block	Limit Per SM	Blocks Per SM
41	Warps	(Threads Per Block / Threads Per Warp)	8	64	8
42	Registers	(Warp limit per SM due to per-warp reg count)	8	128	16
43	Shared Memory (Bytes)		40960	49152	1
44	Note: SM is an abbreviation for (Streaming) Multiprocessor				
45					
46	**Maximum Thread Blocks Per Multiprocessor**		Blocks/SM	* Warps/Block	= Warps/SM
47	Limited by Max Warps or Max Blocks per Multiprocessor		8		
48	Limited by Registers per Multiprocessor		16		
49	**Limited by Shared Memory per Multiprocessor**		**1**	**8**	**8**
50	Note: Occupancy limiter is shown in orange			**Physical Max Warps/SM = 64**	
51				**Occupancy = 8 / 64 = 13%**	

FIGURE 10.5 Analyzing the GaussKernel7() with **(i)** Registers/thread=16, **(ii)** Shared Memory/kernel=40960, and **(iii)** Threads/block=256.

For the GaussKernel7() case study, threads/block and limiting conditions are shown in Fig. 10.5. The top plot shows that you could have gone up to 32 Warps if you launched 1024 threads/block, although this would have made you hit the Shared Memory limitation, shown in Fig. 10.4. The implication of the occupancy concept is far-reaching: may be even a technically-worse kernel can perform better, if it is not resource-hungry, because it can "occupy" the SM better. To phrase differently: writing kernels with a high resource profile has the danger of putting shackles on the kernel's ankles. May be it is better to design kernels with much smaller profiles, although they are not *seemingly* high performance. When a lot of them are running on the SM, at a much higher occupancy, may be they will translate to an overall higher performance. This is the beauty of GPU programming. It is art as much as a science. You can be sure that the resource limitations will not go away even 10 generations from today, although you might have a little more Shared Memory in each, a little more registers etc. So, this "resource-constrained thinking" will always be a part of GPU programming.

To provide an analogy: if you have two runners —one is a world class marathon runner

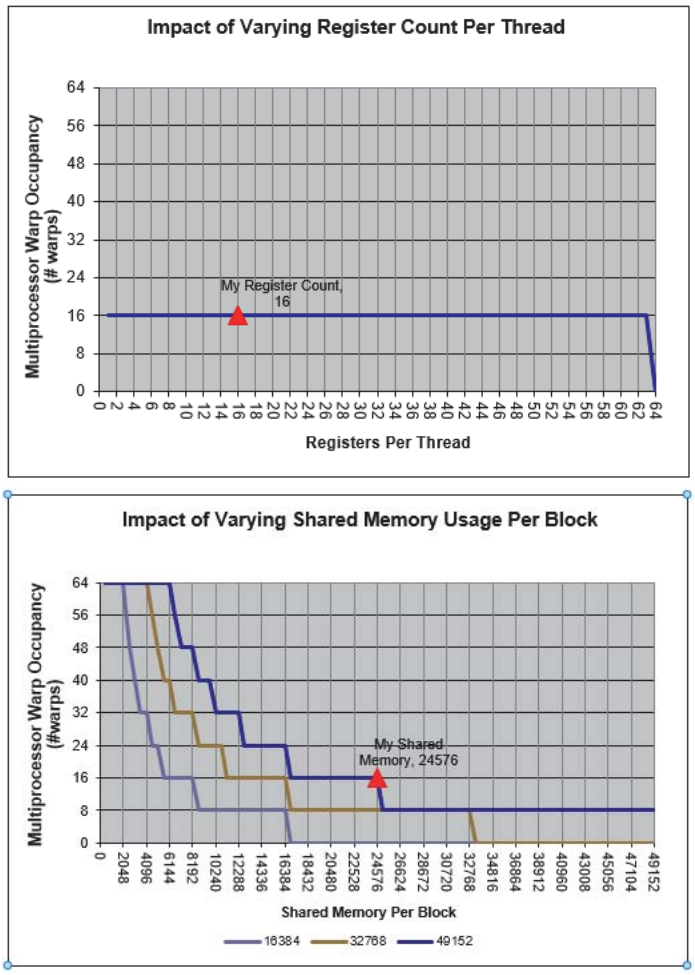


FIGURE 10.6 Analyzing the GaussKernel8() with **(i)** Registers/thread=16, **(ii)** Shared Memory/kernel=24576, and **(iii)** Threads/block=256.

athlete and the other one is an ordinary guy like me— who will win the race? This is an obvious answer. But, here is the second question: who will win the race, *if you put shackles on the marathon runner's feet?* This is what a technically-excellent, but resource-heavy kernel is like. It is a marathon runner with shackles in his feet. You are destroying his performance due to resource limitations.

10.9.6 Case Study: GaussKernel8().

This is the case study for GaussKernel8(): 24576 KB Shared Memory per block, 256 threads per block, 16 registers per kernel. Register usage and Shared Memory usage are shown in Fig. 10.6. Just because it uses 24 KB Shared Memory, it can squeeze two blocks into each SM; together, they will max out the Shared Memory at 48 KB, but it works! It is launched with 256 threads/block (8 Warps/block) and two blocks occupy 16 Warps in the SM. Note that even a 24.1 KB Shared Memory requirement would have halved the occupancy.

CHAPTER 11

CUDA Streams

In the entire rest of this book, we focused on improving the kernel execution time. In a CUDA program, first the data must be transferred from the CPU memory into the GPU memory; it is only when the data is in GPU memory that the GPU cores can access it and process it. When the kernel execution is done, the processed data must be transferred back into the CPU memory. One clear exception to this sequence of events is when we are using the GPU as a graphics machine; if the GPU is being used to render the graphics that are required for a computer game, the output of the processed data is the monitor (or multiple monitors), which is directly connected to the GPU card. So, there is no loss of time to transfer data back and forth between the CPU and the GPU.

However, the type of programs we are focusing on in this book are GPGPU applications, which use the GPU as a general purpose processor; these type of a computations are, in the end, totally under the CPU control, which are originated by the CPU code. Therefore, the data must be shuttled back and forth between the CPU and GPU. What is worse, the data transfer rate between these two dear friends (CPU and GPU) is bottlenecked by the PCIe bus, which has a much lower rate than either the CPU-Main memory bus or GPU-Global Memory bus (see Sections 4.6 and 10.1 for details).

As an example, what if we are performing edge detection on the astronaut.bmp file? Table 11.1 (top half) shows the run time results of this operation on four different GPUs (two Kepler and two Pascal), which is broken down into three different components for the execution time, as shown below:

- CPU→GPU transfers take, on average, 31% of the total execution time.
- Kernel execution inside the GPU takes, on average, 39% of the execution time.
- GPU→CPU transfers take, on average, 30% of the total execution time.

For the purposes of this discussion, the details for the Kepler vs. Pascal are not important. It suffices to say that the CPU→GPU, kernel execution, and GPU→CPU components of the runtime are approximately a third of the total time. Taking the Pascal GTX 1070 as an example; we wait to transfer the entire 121 MB astronaut.bmp image into the GPU (which takes 25 ms); once this transfer is complete, we run the kernel and finish performing edge detection (which takes another 41 ms); once the kernel execution is complete, we transfer it back to the CPU (which is another 24 ms).

When we look at the results for the horizontal flip in Table 11.1 (bottom half), we see a different ratio for the execution time of operations; the CPU→GPU and GPU→CPU transfers are almost half of the total execution time (53% and 43%, respectively), while the kernel execution time is negligible (only 4% of the total).

The striking observation we make from these two cases is that most of the execution time goes to shuttling the data around, rather than doing the actual (and useful) work! Although

TABLE 11.1 Runtime for edge detection and horizontal flip for astronaut.bmp (in ms). Kernel execution times are lumped into a single number for clarity. GTX Titan Z and K80 use the Kepler architecture, while the GTX 1070 and the Titan X (Pascal) use the Pascal architecture.

Operation	Task	Titan Z	K80	GTX 1070	Titan X	Avg %
Edge Detection	CPU→GPU tfer	37	46	25	32	31%
	Kernel execution	64	45	41	26	39%
	GPU→CPU tfer	42	18	24	51	30%
	Total	143	109	90	109	100%
Horizontal Flip	CPU→GPU tfer	39	48	24	32	53%
	Kernel execution	4	4	2	1	4%
	GPU→CPU tfer	43	17	24	34	43%
	Total	86	69	50	67	100%

our efforts to try to decrease the kernel execution time was well justified so far, Table 11.1 makes it clear that we cannot just worry about the kernel execution time when trying to improve our program's overall performance. The data transfer times must be considered as an integral part of the total runtime. So, here is our motivating question for this chapter: could we have started processing *a part of this image* once it was in GPU memory, so, we wouldn't have to wait for the *entire* image to be transferred? The answer is most definitely YES; the idea is that the CPU⟷GPU data transfers can be overlapped with the kernel execution, because they use two different pieces of hardware (PCI controller and the GPU cores) that can work independently and, more importantly, *concurrently*. Here is an analogy that helps us understand this concept:

ANALOGY 11.1: *Pipelining*

Cocotown hosted an annual competition among two teams; the goal was to harvest 100 coconuts in the shortest amount of time. First team did this in three separate steps: **(i)** Charlie went to the jungle, picked 100 coconuts, brought them to the harvesting area, **(ii)** Katherine met Charlie at the harvesting area precisely when he was done, as they coordinated; Kenny started harvesting the coconuts and Charlie went home, and when Katherine finished, she went home, **(iii)** Greg brought the harvested coconuts back to the competition area when Katherine was done, at which time the clock stopped. Each step took approximately 100 minutes and the first team was done in 300 minutes.

Second team had a completely different approach: **(i)** Cindy went and picked only 20 coconuts and brought them to Keith, **(ii)** Keith immediately started harvesting the coconuts immediately when he received them, and **(iii)** Gina immediately brought the finished coconuts to the competition area. The team continued this cycle 5 times.

Although both Team 1 (Charlie, Katherine, Greg) and Team 2 (Cindy, Keith, Gina) were equally experienced farmers, the second team shocked everyone because they completed their harvesting in 140 minutes.

TABLE 11.2 Execution timeline for the second team in Analogy 11.1. Cindy brings 20 coconuts at a time from the jungle to Keith. Keith harvests the coconuts immediately when he receives them. When 20 of them are harvested, Gina delivers them from Keith's harvesting area to the competition desk. When the last 20 coconuts are delivered to the competition desk, the competition ends.

Time	Cindy	Keith	Gina
0–20	bring 20 coconuts to Keith	—	—
20–40	bring 20 more coc. to Keith	harvest 20 coconuts	—
40–60	bring 20 more coc. to Keith	harvest 20 coconuts	deliver 20 coconuts
60–80	bring 20 more coc. to Keith	harvest 20 coconuts	deliver 20 coconuts
80–100	bring 20 more coc. to Keith	harvest 20 coconuts	deliver 20 coconuts
100–120	—	harvest 20 coconuts	deliver 20 coconuts
120–140	—	—	deliver 20 coconuts

11.1 WHAT IS PIPELINING ?

According to Analogy 11.1, why was the second team able to finish their harvesting in 140 minutes? Let us think. It is obvious when three different tasks are *serialized*; because each task takes 100 minutes, they take 300 minutes cumulatively when executed serially. However, if the three tasks can be executed concurrently, the formula changes; the second team was able to overlap the execution time of the two different transfers with the amount of time it takes to actually do the work. What Charlie and Cindy did in Analogy 11.1 corresponds to the CPU→GPU transfers, while Greg and Gina's job is analogous to GPU→CPU transfers. Clearly, Katherine and Keith had tasks that compares to kernel execution.

11.1.1 Execution Overlapping

Table 11.2 sheds light into how the second team finished their job in 140 minutes. In the ideal theoretical case, if the three tasks could be perfectly overlapped, it would only take 100 minutes to finish all of them, because all three of them are being executed concurrently. Let us use a notation to explain the run time of a serial operation (the case for Team 1):

- Task1 execution time = $T1 = 100$,
- Task2 execution time = $T2 = 100$,
- Task3 execution time = $T3 = 100$,

$$\begin{aligned} \text{Serialized runtime} &= T1 \ + T2 \ + T3, \\ &= 100 + 100 + 100 = 300. \end{aligned} \tag{11.1}$$

If any of the executions can be overlapped, as shown in Table 11.1, we can use the following notation to explain the total runtime, by breaking the individual tasks (say, Task1, denoted as $T1$) into subtasks that can be partially overlapped with other subtasks (e.g., $T1a$, $T1b$, and $T1c$). Some of these subtasks might be serially executed (e.g., $T1a$) and some can be overlapped with other subtasks (e.g., $T1b$ can be overlapped with $T2a$). So, the total runtime is (the case for Team 2):

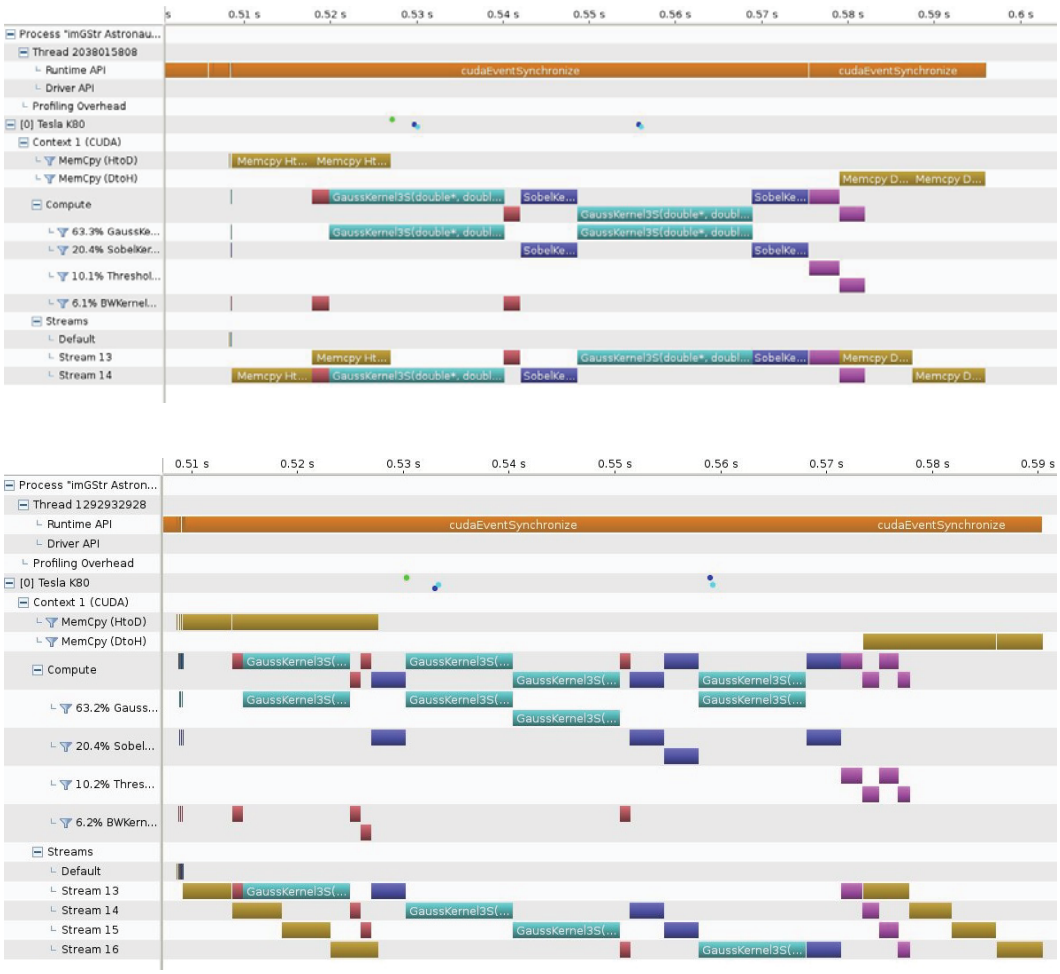


FIGURE 11.4 Nvidia NVVP Results with 2 and 4 streams, on the K80 GPU.

11.9.5 imGStr 2- and 4-stream results

Figure 11.4 depicts the 2-stream and 4-stream results of imGStr, on a K80 GPU. When more than one stream is involved, the Copy Engine and Kernel Engine have some flexibility in choosing from the available operations. Unfortunately, what we witness in Fig. 11.4 is that something close to what we described as "the worst case scenario," described in Section 11.8.5 happens. While the CPU→GPU transfers are 100% coalesced, excepting the first one, the GPU→CPU transfers are only partially coalesced.

It looks like the final ThresholdKernel() batch is preventing the GPU→CPU transfers to be initiated. Instead of providing what has to be done to remedy the situation, I will leave it up to the reader to try a few different things. Here is a list of ideas:

- Look at the order in which we launched the kernels. Would it make a difference had we placed the ThresholdKernel() launches right after the SobelKernel() launches?
- Would it be a better idea to merge all four kernels into a single piece and launch just a single kernel?

- Can we use cudaStreamSynchronize() to somehow manually control when a stream starts and when it ends ?
- Can we launch different versions of the same kernel to achieve better occupancy in the SMs ?

There are so many things to try that some of these ideas will eventually lead to the most efficient answer. Thank goodness for the Visual Profiler; with it, you can see what kind of a runtime execution pattern is being induced for these different ideas.

PART III

More To Know

CHAPTER 12

CUDA Libraries

Mohamadhadi Habibzadeh
University at Albany, SUNY

Omid Rajabi Shishvan
University at Albany, SUNY

Tolga Soyata
University at Albany, SUNY

In the previous chapters of this book, we learned how to create a CUDA program without the help of any "pre-packaged" library, like the ones we will see in this chapter. This was intentional; a deeper understanding of the inner-workings of the GPU can only be gained — and appreciated— when you create a program with the primitives that allow you to get close to the metal (i.e., the GPU cores). Surely, assembly language is a little too low-level, but the libraries we will see in this chapter barely require you to know how the GPU works; so, they are too high-level. The clear choice for the right "level" was plain and simple CUDA, which we based our GPU programming on up to this point in the book. In real life, however, when you are developing GPU programs, it is tedious to have to build everything from scratch. For example, there is no way you can build an optimize a matrix multiplication code the way Nvidia engineers can; because they spend weeks, months optimizing it. Because of this, CUDA programmers typically use high level libraries, such as cuBLAS, cuFFT, etc. and use CUDA itself as a *glue* language, to make everything work together. Every now and then, you will find something that there is no library for; well, this is when you go back to good-and-old CUDA. Aside from that, there is nothing wrong to use the libraries, especially because they are provided free of charge.

12.1 CUBLAS

The roots of Basic Linear Algebra Subprograms (BLAS) go back to the late 1970s and was initially written in Fortran. Note: The exciting programming language of the late 1950's, Formula Translation (Fortran), provided a way for programmers to do scientific computation without requiring assembly language. Having BLAS on top of that was a God sent.

12.1.1 BLAS Levels

cuBLAS is an implementation of BLAS on the CUDA architecture. It can be downloaded from Nvidia's website royalty free. cuBLAS APIs provide support for vector and matrix algebraic operations such as addition, multiplication, etc, allowing developers to accelerate

CHAPTER 13

Introduction to Open CL

Chase Conklin

University of Rochester

Tolga Soyata

University at Albany, SUNY

In this chapter, we will be familiarized with Open CL, which is the most popular GPU programming language, excluding CUDA. This chapter is designed to show how OpenCL simplifies writing multi-platform parallel programs. From previous chapters, we have become familiar with programs such as imflip and imedge. Though OpenCL and CUDA both exist to write highly-parallel code, you will quickly see how the approach for writing programs in each differ.

13.1 WHAT IS OPEN CL?

OpenCL was released in 2009 by the Khronos Group as a framework for writing parallel programs on many different platforms. Unlike CUDA, which only runs on NVidia GPUs, OpenCL code is capable of running on CPUs, GPUs, and other devices such as Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs), as long as the device supports OpenCL. Also unlike CUDA, OpenCL kernels are compiled at runtime, which is how it is able to easily work on many different platforms.

13.1.1 Multi-Platform

OpenCL supports many different devices, but it is up to the device manufacturer to implement the drivers that allow OpenCL to work on their devices. These different implementations are known as *platforms*. Depending on your hardware, your computer may have multiple OpenCL platforms available; for example one from Intel to run on the integrated graphics, and one from NVidia to run on their discrete graphics.

OpenCL considers devices to be in one of three categories: **(i)** CPU, **(ii)** GPU, and **(iii)** accelerator. Of the three, only the accelerator should be unfamiliar. Hardware accelerators include FPGAs and DSPs, or devices such as Intel's Xeon Phi (See Section 3.9 for a detailed introduction to Xeon Phi).

13.1.2 Queue Based

Unlike CUDA, which operates on either synchronous blocking calls, or can operate asynchronously using streams, OpenCL's execution is queue-based, where all commands are dispatched to a command queue, and execute once they reach the head of the queue.

CHAPTER 14

Other GPU Programming Languages

Sam Miller
University of Rochester

Andrew Boggio-Dandry
University at Albany, SUNY

Tolga Soyata
University at Albany, SUNY

In this chapter, we will briefly look at GPU programming languages other than OpenCL and CUDA. Additionally, we will investigate some of the common APIs, such as OpenGL, Open GL ES, OpenCV, and Apple's Metal API. Although these APIs are not programming languages, they transform an existing language into a much more practical one.

14.1 GPU PROGRAMMING WITH PYTHON

Python has the ability to write GPU code through two powerful libraries, `PyOpenCL` and `PyCUDA`. Both of these libraries are written by the same author, and have similar functionality [31]. They closely follow the OpenCL and CUDA APIs respectively and abstract away much of the boilerplate code often needed to write GPU code. The base library itself is written in C++ so that Python can stay out of the way for maximum performance. Another extremely popular Python library, `Numpy`, is used throughout the community for numeric and scientific specific Python operations, and provides a basic, yet flexible numeric array type, that prioritizes speed when compared to the standard Python list. Both `PyOpenCL` and `PyCUDA` create a `Numpy`-like array, that make operations with these device arrays very similar to working with standard `Numpy` arrays, but handled on a CUDA or OpenCL device. Transferring data, error checking, profiling, and more are all wrapped in convenient Python methods to dovetail nicely with your existing Python code.

Both GPU libraries extensively use template code generation methods that can allow for simple creation of element-wise kernels as well as using parallel primitives like scan, reduce, and stream compaction. PyOpenCl also has the functionality to be combined with OpenGL and clBLAS libraries. All of this together makes it extremely easy to prototype code ideas or even speed up existing projects. Further information for working with PyOpenCL or PyCUDA can be found at https://documen.tician.de/pyopencl/ [26] and https://documen.tician.de/pycuda/ [25].

CHAPTER 15

Deep Learning Using CUDA

Omid Rajabi Shishvan
University at Albany, SUNY

Tolga Soyata
University at Albany, SUNY

In this chapter, we will study how GPUs can be used in deep learning. Deep learning is an emerging machine intelligence algorithm based on Artificial Neural Networks (ANNs). ANNs were proposed to be computational models of neurological systems; they were designed to "learn" performing a certain task by mimicking the way a brain learns.

15.1 ARTIFICIAL NEURAL NETWORKS (ANNS)

ANNs are formed by multiple layers of "neurons" connected to each other, which create a network that takes input data, processes the data in each layer and creates an output in the final layer. A common structure of a neural network is shown in Fig. 15.1.

15.1.1 Neurons

Neurons are the building blocks of ANNs. The structure of a neuron is shown in Fig. 15.2, in which the neuron is shown to take multiple inputs and create an output by passing a weighted sum of the inputs through an activation function. These input values may come from other neurons in the previous layer or from the primary inputs of a system.

15.1.2 Activation Functions

Activation functions in a neuron are implemented to introduce non-linearity to the network. If each neuron just output a linear combination of it's inputs, the overall output of the network would be a linear combination of the inputs, which is not a desirable outcome for ANNs. To capture the nonlinear relation in input data, activation functions are used. Common activation functions in neural networks are shown in Table 15.1.

15.2 FULLY CONNECTED NEURAL NETWORKS

The ANNs, in which the connections between their neurons do not make a cycle are called *Feed-Forward Neural Networks*. A common architecture for this type of ANN is when all of the neurons in one layer are connected to all of the neurons in the previous layer and the next layer; this makes a *Fully Connected* neural network. Figure 15.1 depicts a *Fully-Connected Feed-Forward Neural Network*.

and output data, the size that the training data need to be chopped in the training process, input and output validation data if there are any available, etc.

The final stage is evaluating the network by using the `evaluate` method that takes input and output test data and checks the performance of the network on these data.

Other useful methods include `predict` that processes a given input and generates an output, `get_layer` that returns a layer in the network, and `train_on_batch` and `test_on_batch` that train and test the network on only one batch of input data.

Note that if Keras is running on the TensorFlow or CNTK backends, it automatically runs on the GPU if any GPU is detected. If the backend is Theano, there are multiple methods to use the GPU. One way is manually setting the device of the Theano configuration, as follows:

```
import theano
theano.config.device = 'gpu'
theano.config.floatX = 'float32'
```

Bibliography

[1] Apple: Metal Programming Language. https://developer.apple.com/metal/.

[2] Apple: Swift Programming Language. https://developer.apple.com/swift/.

[3] cuBLAS: CUDA Implementation of BLAS. http://docs.nvidia.com/cuda/cublas/index.html#axzz4reNqlFkR.

[4] cuDNN: Nvidia Deep Learning Library. https://developer.nvidia.com/cudnn.

[5] Cygwin Project. http://www.cygwin.com/.

[6] Free GLUT. http://freeglut.sourceforge.net/.

[7] INTEL DX79SR Motherboard. http://ark.intel.com/products/65143/Intel-Desktop-Board-DX79SR.

[8] INTEL i7-3820 4 Core Processor. https://ark.intel.com/products/63698/Intel-Core-i7-3820-Processor-10M-Cache-up-to-3_80-GHz.

[9] INTEL i7-4770K Quad Core Processor. http://ark.intel.com/products/75123/Intel-Core-i7-4770K-Processor-8M-Cache-up-to-3_90-GHz.

[10] INTEL i7-5930K Six Core Processor. https://ark.intel.com/products/82931/Intel-Core-i7-5930K-Processor-15M-Cache-up-to-3_70-GHz.

[11] INTEL i7-5960X 8 Core Extreme Processor. http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz.

[12] INTEL Xeon E5-2680v4 Processor. https://ark.intel.com/products/91754/Intel-Xeon-Processor-E5-2680-v4-35M-Cache-2_40-GHz.

[13] INTEL Xeon E5-2690 8-Core Processor. https://ark.intel.com/products/64596/Intel-Xeon-Processor-E5-2690-20M-Cache-2_90-GHz-8_00-GTs-Intel-QPI.

[14] INTEL Xeon E7-8870 Processor. http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI.

[15] INTEL Xeon W-3690 Six Core Processor. https://ark.intel.com/products/52586/Intel-Xeon-Processor-W3690-12M-Cache-3_46-GHz-6_40-GTs-Intel-QPI.

[16] Mobility Meets Performance: NvidiaOptimus Technology. http://www.nvidia.com/object/optimus_technology.html.

[17] Notepad++ Editor. https://notepad-plus-plus.org/.

[18] Nvidia CUDA Installation Guide for Mac OSX. http://docs.nvidia.com/cuda/pdf/CUDA_Installation_Guide_Mac.pdf.

[19] Nvidia Profiler Metrics Reference. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference.

[20] Nvidia Visual Profiler User's Guide: Settings Options. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#settings-view.

[21] Nvidia Visual Profiler User's Guide: Timeline Options. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#timeline-view.

[22] OpenCV Library. http://opencv.org/.

[23] OpenGL 4.0 Specificaton. https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf.

[24] OpenGL ES. https://www.khronos.org/opengles/.

[25] PyCUDA Rererence. https://documen.tician.de/pycuda/.

[26] PyOpenCL Rererence. https://documen.tician.de/pyopencl/.

[27] Vulkan 1.0.59 Specificaton. https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html.

[28] A. Alling, N. Powers, and T. Soyata. Face Recognition: A Tutorial on Computational Aspects. In *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing*, chapter 20, pages 405–425. IGI Global, 2016.

[29] Susan Blackmore. *Consciousness: an introduction*. Routledge, 2013.

[30] Daniel Kahneman. *Thinking, fast and slow*. Macmillan, 2011.

[31] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[32] Nvidia. GTX 1080 White Paper. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[33] N. Powers, A. Alling, K. Osolinsky, T. Soyata, M. Zhu, H. Wang, H. Ba, W. Heinzelman, J. Shi, and M. Kwon. The Cloudlet Accelerator: Bringing Mobile-Cloud Face Recognition into Real-Time. In *Globecom Workshops (GC Wkshps)*, pages 1–7, San Diego, CA, Dec 2015.

[34] N. Powers and T. Soyata. AXaaS (Acceleration as a Service): Can the Telecom Service Provider Rent a Cloudlet ? In *Proceedings of the 4th IEEE International Conference on Cloud Networking (CNET)*, pages 232–238, Niagara Falls, Canada, Oct 2015.

[35] N. Powers and T. Soyata. Selling FLOPs: Telecom Service Providers Can Rent a Cloudlet via Acceleration as a Service (AXaaS). In T. Soyata, editor, *Enabling Real-Time Mobile Cloud Computing through Emerging Technologies*, chapter 6, pages 182–212. IGI Global, 2015.

[36] T. Soyata, H. Ba, W. Heinzelman, M. Kwon, and J. Shi. Accelerating Mobile Cloud Computing: A Survey. In H. T. Mouftah and B. Kantarci, editors, *Communication Infrastructures for Cloud Computing*, chapter 8, pages 175–197. IGI Global, Sep 2013.

[37] T. Soyata, R. Muraleedharan, S. Ames, J. H. Langdon, C. Funai, M. Kwon, and W. B. Heinzelman. COMBAT: mobile Cloud-based cOmpute/coMmunications infrastructure for BATtlefield applications. In *Proceedings of SPIE*, volume 8403, pages 84030K–84030K, May 2012.

[38] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman. Cloud-Vision: Real-Time Face Recognition Using a Mobile-Cloudlet-Cloud Acceleration Architecture. In *Proceedings of the 17th IEEE Symposium on Computers and Communications (ISCC)*, pages 59–66, Cappadocia, Turkey, Jul 2012.

[39] Jane Vanderkooi. *Your Inner Engine: An Introductory Course on Human Metabolism.* 2017.

Index