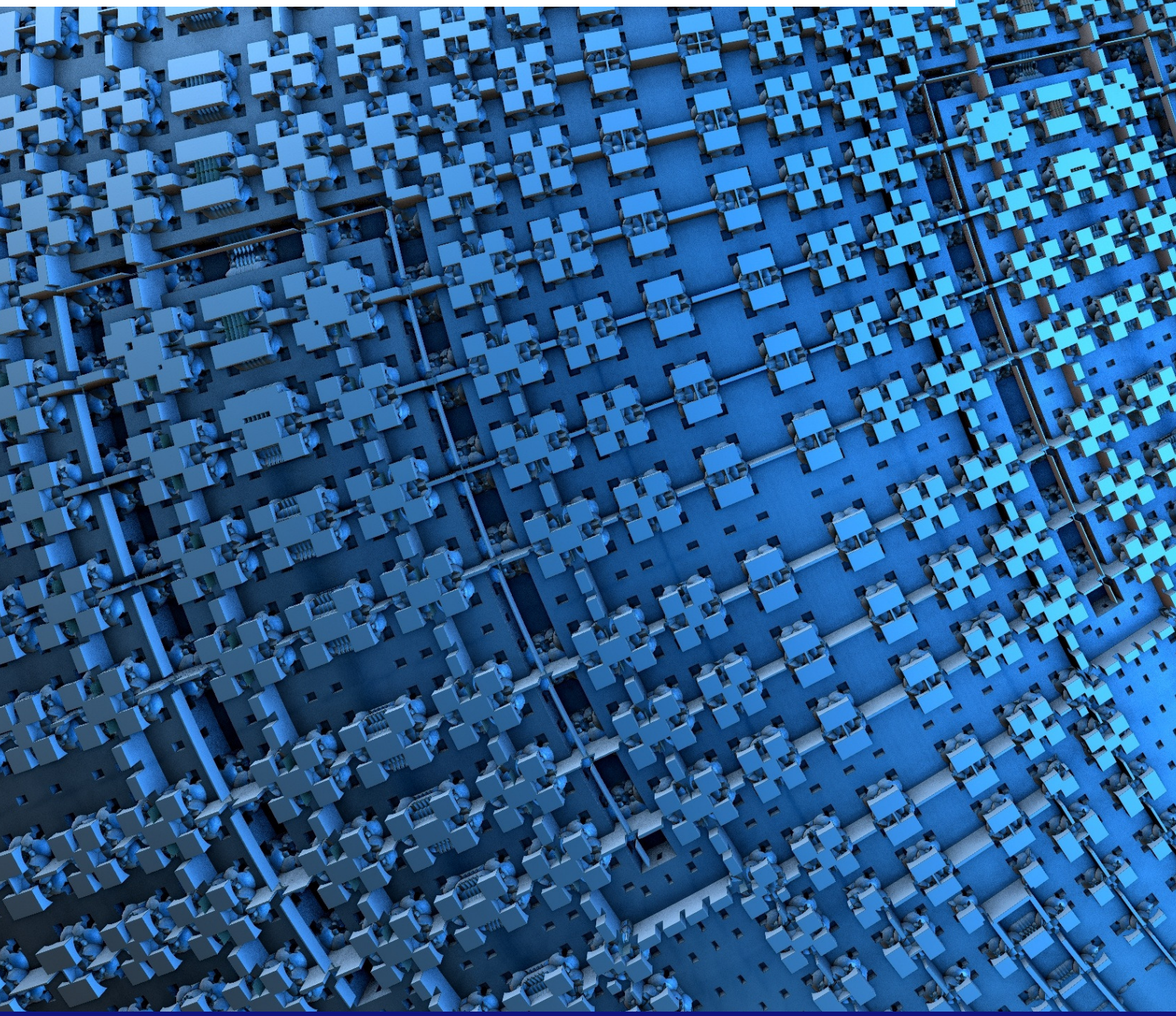
CRC PRESS ▪ TAYLOR & FRANCIS

High Performance Computing

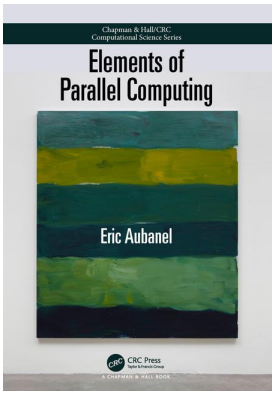
A Chapter Sampler

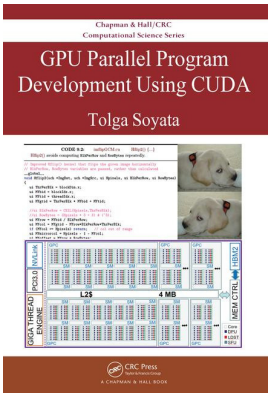


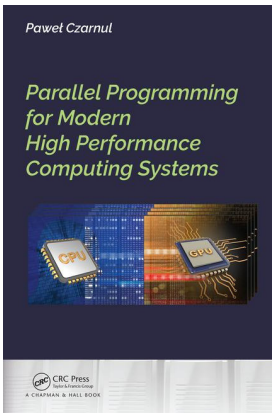
CRC Press
Taylor & Francis Group

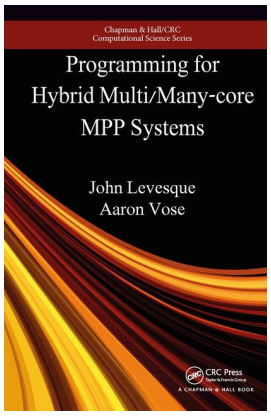
www.crcpress.com

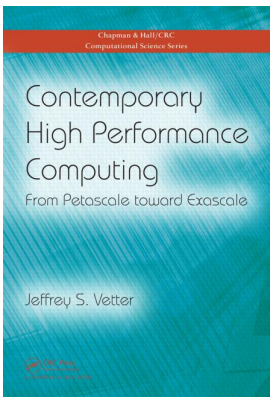
Contents

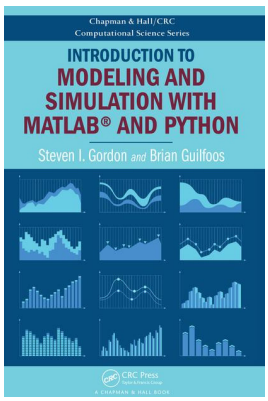












CHAPTER 1

Overview of Parallel Computing

1.1 INTRODUCTION

In the first 60 years of the electronic computer, beginning in 1940, computing performance per dollar increased on average by 55% per year [52]. This staggering 100 billion-fold increase hit a wall in the middle of the first decade of this century. The so-called *power wall* arose when processors couldn't work any faster because they couldn't dissipate the heat they produced. Performance has kept increasing since then, but only by placing multiple processors on the same chip, and limiting clock rates to a few GHz. These *multicore* processors are found in devices ranging from smartphones to servers.

Before the multicore revolution, programmers could rely on a free performance increase with each processor generation. However, the disparity between theoretical and achievable performance kept increasing, because processing speed grew much faster than memory bandwidth. Attaining peak performance required careful attention to memory access patterns in order to maximize re-use of data in cache memory. The multicore revolution made things much worse for programmers. Now increasing the performance of an application required parallel execution on multiple cores.

Enabling parallel execution on a few cores isn't too challenging, with support available from language extensions, compilers and runtime systems. The number of cores keeps increasing, and *manycore* processors, such as Graphics Processing Units (GPUs), can have thousands of cores. This makes achieving good performance more challenging, and parallel programming is required to exploit the potential of these parallel processors.

Why Learn Parallel Computing?

Compilers already exploit instruction level parallelism to speed up sequential code, so couldn't they also automatically generate multithreaded code to take advantage of multiple cores? Why learn distributed parallel programming, when frameworks such as MapReduce can meet the application programmer's needs? Unfortunately it's not so easy. Compilers can only try to optimize the code that's given to them, but they can't rewrite the underlying algorithms to be parallel. Frameworks, on the other hand, do offer significant benefits. However, they tend to be restricted to particular domains and they don't always produce the desired performance.

High level tools are very important, but we will always need to go deeper and apply parallel programming expertise to understand the performance of frameworks in order to make

better use of them. Specialized parallel code is often essential for applications requiring high performance. The challenge posed by the rapidly growing number of cores has meant that more programmers than ever need to understand something about parallel programming. Fortunately parallel processing is natural for humans, as our brains have been described as parallel processors, even though we have been taught to program in a sequential manner.

Why is a New Approach Needed?

Rapid advances in hardware make parallel computing exciting for the devotee. Unfortunately, advances in the field tend to be driven by the hardware. This has resulted in solutions tied to particular architectures that quickly go out of date, as do textbooks. On the software front it's easy to become lost amid the large number of parallel programming languages and environments.

Good parallel algorithm design requires postponing consideration of the hardware, but at the same time good algorithms and implementations must take the hardware into account. The way out of this parallel software/hardware thicket is to find a suitable level of abstraction and to recognize that a limited number of solutions keep being re-used.

This book will guide you toward being able to think in parallel, using a task graph model for parallel computation. It makes use of recent work on parallel programming patterns to identify commonly used algorithmic and implementation strategies. It takes a language-neutral approach using pseudocode that reflects commonly used language models. The pseudocode can quite easily be adapted for implementation in relevant languages, and there are many good resources online and in print for parallel languages.

1.2 TERMINOLOGY

It's important to be clear about terminology, since *parallel computing*, *distributed computing*, and *concurrency* are all overlapping concepts that have been defined in different ways. Parallel computers can also be placed in several categories.

Definition 1.1 (Parallel Computing)**.** *Parallel Computing means solving a computing problem in less time by breaking it down into parts and computing those parts simultaneously.*

Parallel computers provide more computing resources and memory in order to tackle problems that cannot be solved in a reasonable time by a single processor core. They differ from sequential computers in that there are multiple processing elements that can execute instructions in parallel, as directed by the parallel program. We can think of a sequential

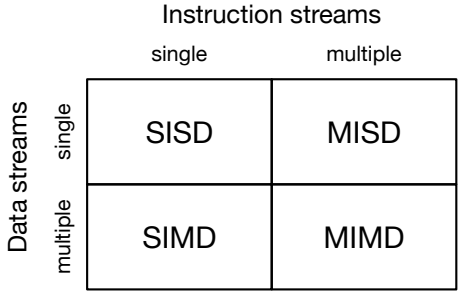


Figure 1.1: Flynn's Taxonomy.

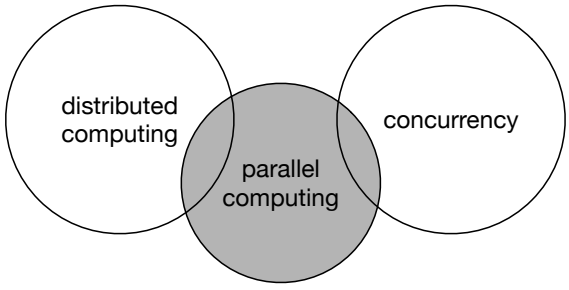


Figure 1.2: Three overlapping disciplines.

computer, as described by the von Neumann architecture, as executing a stream of instructions that accesses a stream of data. Parallel computers work with multiple streams of data and/or instructions, which is the basis of Flynn's taxonomy, given in Figure 1.1. A sequential computer is in the *Single Instruction Single Data* (SISD) category and parallel computers are in the other categories. *Single Instruction Multiple Data* (SIMD) computers have a single stream of instructions that operate on multiple streams of data in parallel. *Multiple Instruction Multiple Data* (MIMD) is the most general category, where each instruction stream can operate on different data. There aren't currently any *Multiple Instruction Single Data* (MISD) computers in production. While most parallel computers are in the MIMD category, most also incorporate SIMD processing elements.

MIMD computers are further classified into *shared memory* and *distributed memory* computers. The processing elements of a shared memory computer all have access to a single memory address space. Distributed memory computers have memory that is distributed among compute nodes. If a processing element on one node wants to access data on another node, it can't directly refer to it by its address, but must obtain it from the other node by exchanging messages.

Distributed memory parallel computing can be done on a cluster of computers connected by a network. The larger field of distributed computing has some of the same concerns, such as speed, but is not mainly concerned with the solution of a single problem. Distributed systems are typically loosely coupled, such as peer-to-peer systems, and the main concerns include reliability as well as performance.

Parallel computing can also be done on a shared memory multiprocessor. Simultaneous access to the same data can lead to incorrect results. Techniques from the field of concurrency, such as mutual exclusion, can be used to ensure correctness. The discipline of concurrency is about much more than parallel computing. Shared access to data is also important for databases and operating systems. For example, concurrency addresses the problem of ensuring that two simultaneous operations on a bank account don't conflict.

We can summarize the three overlapping disciplines with the diagram in Figure 1.2.

1.3 EVOLUTION OF PARALLEL COMPUTERS

Parallel computers used to be solely large expensive machines with specialized hardware. They were housed at educational and governmental institutions and were mainly dedicated to scientific computing. An important development came in the 1990s with the so-called Beowulf revolution, where the best performance to cost ratio could be obtained with clusters of commodity PCs connected by network switches rather than with expensive purpose-built supercomputers. While this made parallel computing more accessible, it still remained

the domain of enthusiasts and those whose needs required high performance computing resources.

Not only is expert knowledge needed to write parallel programs for a cluster, the significant computing needs of large scale applications require the use of shared supercomputing facilities. Users had to master the complexities of coordinating the execution of applications and file management, sometimes across different administrative and geographic domains. This led to the idea of Grid computing in the late 1990s, with the dream of computing as a utility, which would be as simple to use as the power grid. While the dream hasn't been realized, Grid computing has provided significant benefit to collaborative scientific data analysis and simulation. This type of distributed computing wasn't adopted by the wider community until the development of cloud computing in the early 2000s.

Cloud computing has grown rapidly, thanks to improved network access and virtualization techniques. It allows users to rent computing resources on demand. While using the cloud doesn't require parallel programming, it does remove financial barriers to the use of compute clusters, as these can be assembled and configured on demand. The introduction of frameworks based on the MapReduce programming model eliminated the difficulty of parallel programming for a large class of data processing applications, particularly those associated with the mining of large volumes of data.

With the emergence of multicore and manycore processors all computers are parallel computers. A desktop computer with an attached manycore co-processor features thousands of cores and offers performance in the trillions of operations per second. Put a number of these computers on a network and even more performance is available, with the main limiting factor being power consumption and heat dissipation. Parallel computing is now relevant to all application areas. Scientific computing isn't the only player any more in large scale high performance computing. The need to make sense of the vast quantity of data that cheap computing and networks have produced, so-called Big Data, has created another important use for parallel computing.

1.4 EXAMPLE: WORD COUNT

Let's consider a simple problem that can be solved using parallel computing. We wish to list all the words in a collection of documents, together with their frequency. We can use a list containing key-value pairs to store words and their frequency. The sequential Algorithm 1.1 is straightforward.

Algorithm 1.1: Sequential word count

Input: collection of text documents
Output: list of ⟨word, count⟩ pairs

```
foreach document in collection do
    foreach word in document do
        if first occurrence of word then
            add ⟨word, 1⟩ to ordered list
        else
            increment count in ⟨word, count⟩
        end
    end
end
```

If the input consists of two documents, one containing "The quick brown fox jumps over a lazy dog" and the other containing "The brown dog chases the tabby cat," then the output would be the list: $\langle a, 1\rangle, \langle brown, 2\rangle, \ldots, \langle tabby, 1\rangle, \langle the, 3\rangle$.

Think about how you would break this algorithm in parts that could be computed in parallel, before we discuss below how this could be done.

1.5 PARALLEL PROGRAMMING MODELS

Many programming models have been proposed over the more than 40 year history of parallel computing. Parallel computing involves identifying those tasks that can be performed concurrently and coordinating their execution. Fortunately, abstraction is our friend, as in other areas of computer science. It can mask some of the complexities involved in implementing algorithms that make use of parallel execution. Programming models that enable parallel execution operate at different levels of abstraction. While only a few will be mentioned here, parallel programming models can be divided into three categories, where the parallelism is implicit, partly explicit, and completely explicit [66].

1.5.1 Implicit Models

There are some languages where parallelism is implicit. This is the case for functional programming languages, such as Haskell, where the runtime works with the program as a graph and can identify those functions that can be executed concurrently. There are also algorithmic skeletons, which may be separate languages or frameworks built on existing languages. Applications are composed of parallel skeletons, such as in the case of MapReduce. The programmer is concerned with the functionality of the components of the skeletons, not with how they will be executed in parallel.

The word count problem solved in Algorithm 1.1 is a canonical MapReduce application. The programmer writes a `map` function that emits $\langle$word, count$\rangle$ pairs and a `reduce` function to sum the values of all pairs with the same word. We'll examine MapReduce in more detail below and in Chapter 4.

1.5.2 Semi-Implicit Models

There are other languages where programmers identify regions of code that can be executed concurrently, but where they do not have to determine how many resources (threads/processes) to use and how to assign tasks to them. The Cilk language, based on C/C++, allows the programmer to identify recursive calls that can be done in parallel and to annotate loops whose iterations can be computed independently. OpenMP is another approach, which augments existing imperative languages with an API to enable loop and task level specification of parallelism. Java supports parallelism in several ways. The Fork/Join framework of Java 7 provides an Executor service that supports recursive parallelism much in the same way as Cilk. The introduction of streams in Java 8 makes it possible to identify streams that can be decomposed into parallel sub-streams. For these languages the compiler and runtime systems take care of the assignment of tasks to threads. Semi-implicit models are becoming more relevant with the increasing size and complexity of parallel computers.

For example, parallel loops can be identified by the programmer, such as:

parallel for $i \leftarrow 0$ *to* $n-1$ **do**
    $c[i] \leftarrow a[i] + b[i]$
end

Once loops that have independent iterations have been identified, the parallelism is very simple to express. Ensuring correctness is another matter, as we'll see in Chapters 2 and 4.

1.5.3 Explicit Models

Finally, there are lower level programming models where parallelism is completely explicit. The most popular has been the Message Passing Interface (MPI), a library that enables parallel programming for C/C++ and Fortran. Here the programmer is responsible for identifying tasks, mapping them to processors, and sending messages between processors. MPI has been very successful due to its proven performance and portability, and the ease with which parallel algorithms can be expressed. However, MPI programs can require significant development time, and achieving the portability of performance across platforms can be very challenging. OpenMP, mentioned above, can be used in a similar way, where all parallelism and mapping to threads is expressed explicitly. It has the advantage over MPI of a shared address space, but can suffer from poorer performance, and is restricted to platforms that offer shared memory. So-called partitioned global address space (PGAS) languages combine some of the advantages of MPI and OpenMP. These languages support data parallelism by allowing the programmer to specify data structures that can be distributed across processes, while providing the familiar view of a single address space. Examples include Unified Parallel C and Chapel.

In the following message passing example for the sum of two arrays a and b on p processors, where the arrays are initially on processor 0, the programmer has to explicitly scatter the operands and gather the results:

```
scatter(0, a, n/p, aLoc)
scatter(0, b, n/p, bLoc)
for i ← 0 to n/p − 1 do
    cLoc[i] ← aLoc[i] + bLoc[i]
end
gather(0, c, n/p, cLoc)
```

Here arrays a and b of length n are scattered in contiguous chunks of n/p elements to p processors, and stored in arrays $aLoc$ and $bLoc$ on each processor. The resulting $cLoc$ arrays are gathered into the c array on processor 0. We'll examine message passing in detail in Chapter 4.

1.5.4 Thinking in Parallel

Parallel programming models where the parallelism is implicit don't require programmers to "think in parallel." As long as the underlying model (e.g., functional language or skeleton framework) is familiar, then the benefit of harnessing multiple cores can be achieved without the additional development time required by explicit parallel programming. However, not only can this approach limit the attainable performance, more importantly it limits the exploration space in the design of algorithms. Learning to think in parallel exposes a broader landscape of algorithm design. The difficulty is that it requires a shift in the mental model of the algorithm designer or programmer. Skilled programmers can look at code and run it in their mind, executing it on the *notional machine* associated with the programming language. A notional machine explains how a programming language's statements are executed. Being able to view a static program as something that is dynamically executed on a notional machine is a *threshold concept*. Threshold concepts are transformative and lead to new

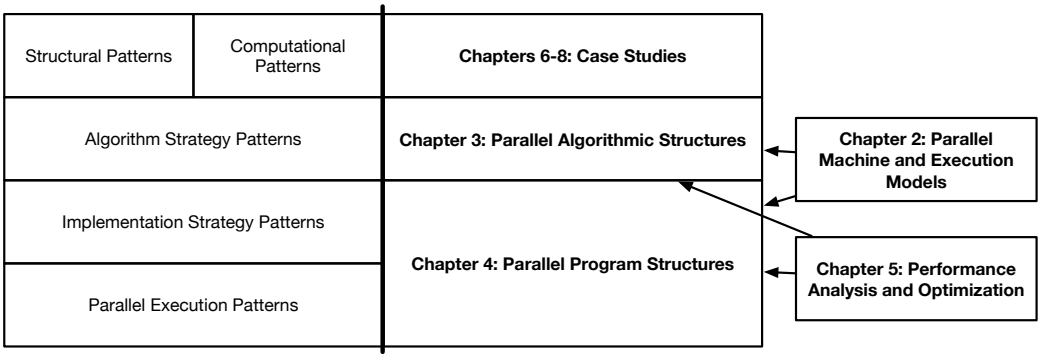


Figure 1.3: OPL hierarchy (left) and corresponding chapters (right).

ways of thinking [69]. It could be argued that making the transition from sequential to parallel notional machines is another threshold concept.

1.6 PARALLEL DESIGN PATTERNS

While the prospect of an enlarged algorithmic design space may be thrilling to some, it can be daunting to many. Fortunately, we can build on the work of the parallel programming community by reusing existing algorithmic techniques. This has been successful particularly in object-oriented software design, where design patterns offer guidance to common software design problems at different layers of abstraction. This idea has been extended to parallel programming, notably with Berkeley's Our Pattern Language (OPL) [44]. Programmers can identify patterns in program structure at a high level of abstraction, such as pipe-and-filter, or those that are found in particular application domains, such as in graph algorithms.

Although these high level patterns do not mention parallelism, they can naturally suggest parallel implementation, as in the case of pipe-and-filter, which can benefit from pipelined parallel execution. These patterns also document which lower level patterns are appropriate. At lower levels there are patterns that are commonly used in algorithm and software design, such as divide-and-conquer, geometric decomposition, and the master-worker pattern. There is a seemingly never ending number of algorithmic and design patterns, which is why mastering the discipline can take a long time. The true benefit of the work of classifying patterns is that it can provide a map of parallel computing techniques.

Figure 1.3 illustrates the OPL hierarchy and indicates how the chapters of this book refer to different layers. The top two layers are discussed in this chapter. We will not be considering the formal design patterns in the lower layers, but will examine in Chapters 3 and 4 the algorithmic and implementation structures that they cover.

1.6.1 Structural Patterns

Structural patterns consist of interacting components that describe the high level structure of a software application. These include well known structures that have been studied by the design pattern community. Examples include the model-view-controller pattern that is used in graphical user interface frameworks, the pipe-and-filter pattern for applying a series of filters to a stream of data, and the agent-and-repository and MapReduce patterns used

Figure 1.4: Part of a text processing pipeline.

in data analysis. These patterns can be represented graphically as a group of interacting tasks.

Consider the pipe-and-filter pattern, which is useful when a series of transformations needs to be applied to a collection of data. The transformations, also called filters, can be organized in a pipeline. Each filter is independent and does not produce side effects. It reads its input, applies its transformation, then outputs the result. As in a factory assembly line, once the stream of data fills the pipeline the filters can be executed in parallel on different processors. If the filters take roughly the same amount of time, the execution time can be reduced proportionately to the number of processors used. It can happen that some filters will take much longer to execute than others, which can cause a bottleneck and slow down execution. The slower filters can be sped up by using parallel processing techniques to harness multiple processors. An example is shown in Figure 1.4, which shows the first few stages of a text processing pipeline. A stream of social media messages is first filtered to only keep messages in English, then any metadata (such as URLs) is removed in the next filter. The third filter tokenizes the messages into words. The pipeline could continue with other filters to perform operations such as tagging and classification.

Another pattern, MapReduce, became popular when Google introduced the framework of the same name in 2004, but it has been used for much longer. It consists of a map phase, where the same operation is performed on objects in a collection, followed by a reduce phase where a summary of the results of the map phase is collected. Many applications that need to process and summarize large quantities of data fit this pattern. Continuing with the text processing example, we might want to produce a list of words and their frequency from a stream of social media messages, as in the example of Section 1.4.

Both map and reduce phases of this pattern can be executed in parallel. Since the map operations are independent, they can easily be done in parallel across multiple processors. This type of parallel execution is sometimes called embarrassingly parallel, since there are no dependencies between the tasks and they can be trivially executed in parallel. The reduce phase can be executed in parallel using the reduction operation, which is a frequently used lower level parallel execution pattern. The attraction of MapReduce, implemented in a framework, is that the developer can build the mapper and reducer without any knowledge of how they are executed.

1.6.2 Computational Patterns

Whereas in cases like pipe-and-filter and MapReduce the structure reveals the potential for parallel execution, usually parallelism is found in the functions that make up the components of the software architecture. In practice these functions are usually constructed from a limited set of computational patterns. Dense and sparse linear algebra computations are probably the two most common patterns. They are used by applications such as games, machine learning, image processing and high performance scientific computing. There are well established parallel algorithms for these patterns, which have been implemented in many

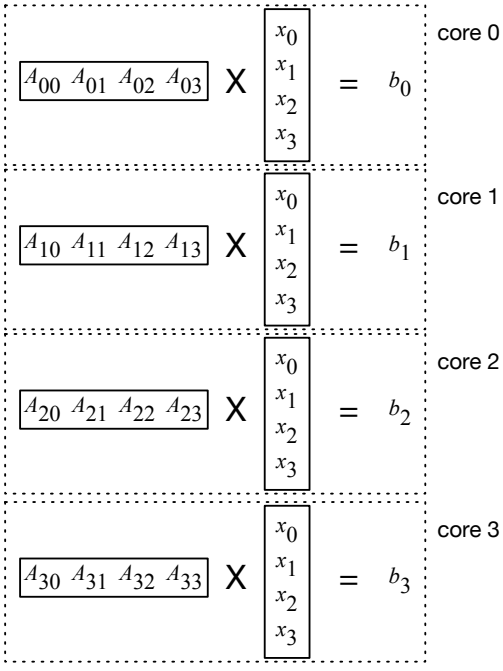


Figure 1.5: Parallel matrix-vector multiplication $b = Ax$.

libraries. The High Performance Linpack (HPL) benchmark used to classify the top 500 computers involves solving a dense system of linear equations. Parallelism arises naturally in these applications. In matrix-vector multiplication, for example, the inner products that compute each element of the result vector can be computed independently, and hence in parallel, as seen in Figure 1.5. In practice it is more difficult to develop solutions that scale well with matrix size and the number of processors, but the plentiful literature provides guidance.

Another important pattern is one where operations are performed on a grid of data. It occurs in scientific simulations that numerically solve partial differential equations, and also in image processing that executes operations on pixels. The solutions for each data point can be computed independently but they require data from neighboring points. Other patterns include those found in graph algorithms, optimization (backtrack/branch and bound, dynamic programming), and sorting. It is reassuring that the lists that have been drawn up of these patterns include less than twenty patterns. Even though the landscape of parallel computing is vast, most applications can be composed of a small number of well studied computational patterns.

1.6.3 Patterns in the Lower Layers

Structural and computational patterns allow developers to identify opportunities for parallelism and exploit ready-made solutions by using frameworks and libraries, without the need to acquire expertise in parallel programming. Exploration of the patterns in the lower layers is necessary for those who wish to develop new parallel frameworks and libraries, or need a customized solution that will obtain higher performance. This requires the use of

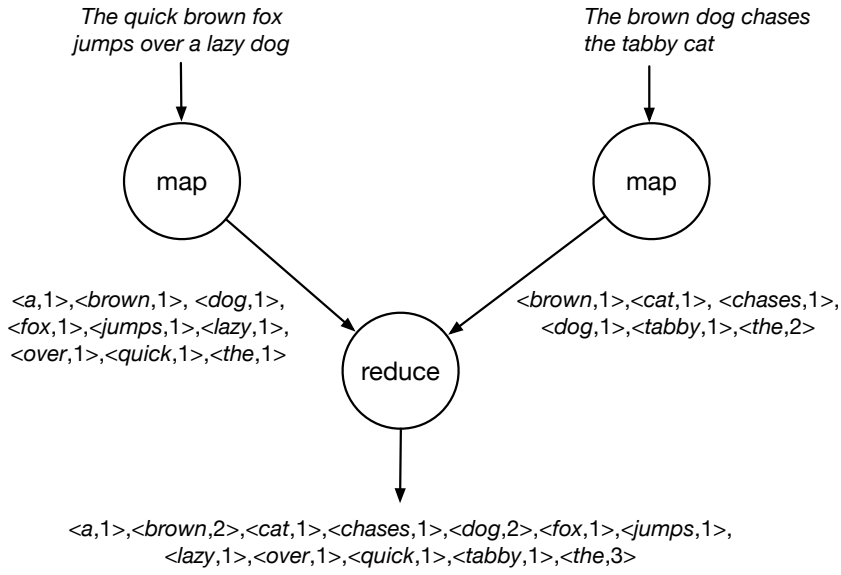


Figure 1.6: Word count as a MapReduce pattern.

algorithmic and implementation structures to create parallel software. These structures will be explored in detail in Chapters 3 and 4.

1.7 WORD COUNT IN PARALLEL

Let's return to the word count problem of Section 1.4. Figure 1.6 illustrates how this algorithm can be viewed as an instance of the MapReduce pattern. In the map phase, $\langle \mathsf{word}, \mathsf{count} \rangle$ pairs are produced for each document. The reduce phase aggregates the pairs produced from all documents. What's not shown in Figure 1.6 is that there can be multiple reducers. If we're using a MapReduce framework then we just need to implement a mapper function to generate $\langle \mathsf{word}, \mathsf{count} \rangle$ pairs given a document and a reducer function to sum the values of a given word, as we'll see in Chapter 4.

To produce an explicitly parallel solution we need to start by finding the parallelism in the problem. The task of creating $\langle \mathsf{word}, \mathsf{count} \rangle$ pairs can be done independently on each document, and therefore can be done trivially in parallel. The reduction phase is not as simple, since some coordination among tasks is needed, as described in the reduction algorithm pattern. Next, we need to consider what type of computational platform is to be used, and whether the documents are stored locally or are distributed over multiple computers. A distributed approach is required if the documents are not stored in one place or if the number of documents is large enough to justify using a cluster of computers, which might be located in the cloud.

Alternatively, a local computer offers the choice of using a shared data structure. A concurrent hash map would allow multiple threads to update $\langle \mathsf{word}, \mathsf{count} \rangle$ pairs, while providing the necessary synchronization to avoid conflicts should multiple updates overlap. A distributed implementation could use the master-worker pattern, where the master performs the distribution and collection of work among the workers. Note that this approach could also be used on multiple cores of a single computer.

Both distributed and shared implementations could be combined. Taking a look at the sequential algorithm we can see that word counts could be done independently for each line of text. The processing of the words of the sentences of each document could be accomplished in parallel using a shared data structure, while the documents could be processed by multiple computers.

This is a glimpse of how rich the possibilities can be in parallel programming. It can get even richer as new computational platforms emerge, as happened in the mid 2000s with general purpose programming on graphics processing units (GPGPU). The patterns of parallel computing provided a guide in our example problem, as they allowed the recognition that it fit the MapReduce structural pattern, and that implementation could be accomplished using well established algorithmic and implementation patterns. While this book will not follow the formalism of design patterns, it does adopt the similar view that there is a set of parallel computing elements that can be composed in many ways to produce clear and effective solutions to computationally demanding problems.

1.8 OUTLINE OF THE BOOK

Parallel programming requires some knowledge of parallel computer organization. Chapter 2 begins with a discussion of three abstract machine models: SIMD, shared memory, and distributed memory. It also discusses the hazards of access to shared variables, which threaten program correctness. Chapter 2 then presents the task graph as an execution model for parallel computing. This model is used for algorithm design, analysis, and implementation in the following chapters. Chapter 3 presents an overview of algorithmic structures that are often used as building blocks. It focuses on the fundamental task of finding parallelism via task and data decomposition. Chapter 4 explores the three machine models more deeply through examination of the implementation structures that are relevant to each model.

Performance is a central concern for parallel programmers. Chapter 5 first shows how work-depth analysis allows evaluation of the task graph of a parallel algorithm, without considering the target machine model. The barriers to achieving good performance that are encountered when implementing parallel algorithms are discussed next. This chapter closes with advice on how to effectively and honestly present performance results.

The final chapters contain three detailed case studies, drawing on what was learned in the previous chapters. They consider different types of parallel solutions for different machine models, and take a step-by-step voyage through the algorithm design and analysis process.

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; the 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? (1) A plane moving in the sky, (2) a tank shooting, or (3) 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 graphic elements, composed of many pixels, and moving them (such as rotating them) requires heavy floating point computations, as exemplified in Equation 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 have I was playing computer games in the late 1990s 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 1990s 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 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 are 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 players 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 1990s 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, routine image coordinate conversions during the course of a game, such as 3D-to-2D conversions and handling of triangles, could be performed significantly faster by dedicated hardware rather than wasting precious CPU time. 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 1990s strictly focused on delivering as high of a floating point performance as possible. This freed the CPU resources and allowed a PC to 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 1990s 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 the 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 1980s, 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 was 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()

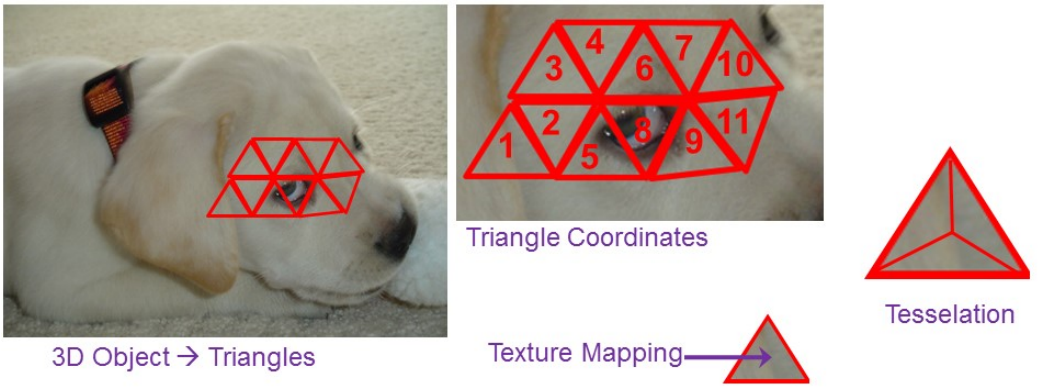


FIGURE 6.1 Turning the dog picture into a 3D wire frame. Triangles are used to represent the object, rather than pixels. This representation allows us to map a texture to each triangle. When the object moves, so does each triangle, along with their associated textures. To increase the resolution of this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

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 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 1990s. 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 Equation 4.1 or the scaling operation in Equation 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 Figure 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 Equation 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 subdivided into smaller triangles as shown in Figure 6.1. Note: Only 11 triangles are shown in Figure 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 to the underlying

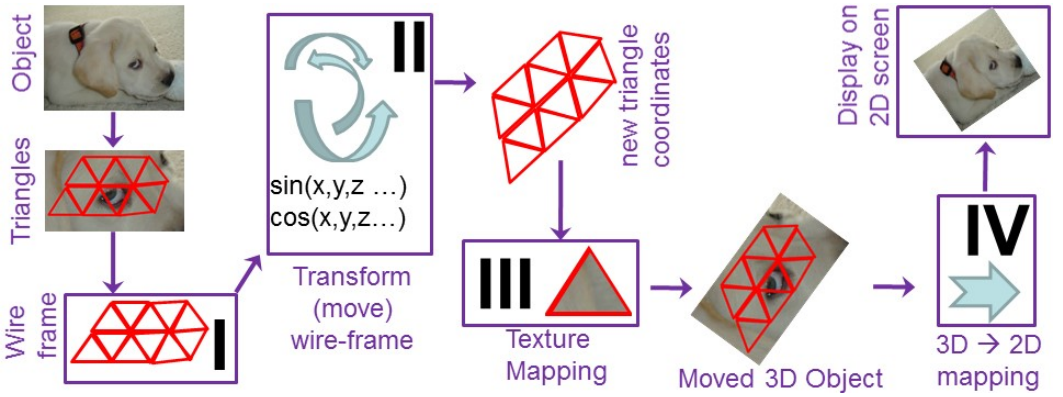


FIGURE 6.2 Steps to move triangulated 3D objects. Triangles contain two attributes: their *location* and their *texture*. Objects are moved by performing mathematical operations only on their coordinates. A final texture mapping places the texture back on the moved object coordinates, while a 3D-to-2D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

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 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: (1) the *location* of the triangle, and (2) 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 recomputed 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 Figure 6.2:

- Ability to deal with the natural data type *triangle*, Box I in Figure 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 IIs, 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 realized that they could 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 and 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 he or she 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 OpenGL, 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 multibillion 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 hundreds 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 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 Figure 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 toward the end of 1990s. 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 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 desktop 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 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 of as the 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 GPGPUs was large, decided to turn all of their GPUs into GPGPUs by designing the "Box II" in Figure 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) 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, OpenCL, 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 OpenGL. 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 OpenCL language emerged that allowed GPU code to be developed for Intel, AMD, and other GPUs. While AMD initially introduced its own language CTM (Close to Metal), it eventually abandoned these efforts and went strictly with OpenCL. As of the year 2016, there are two predominant desktop GPU languages in the world: OpenCL 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 OpenCL 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 OpenCL; it can be an FPGA, DSP, or any other device that has a similar parallel architecture to the GPU. The current implementation of OpenCL 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 versus 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 had 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 the CPU side with some specific keywords to distinguish between host versus device code. The burden to determine how the execution would take place at runtime — regarding CPU versus 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 its 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 versus 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 could pick and harvest coconuts twice as fast as 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 on the sidelines and cheered for their team.

Just before the competition started, another farmer — Tolga, who had 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?

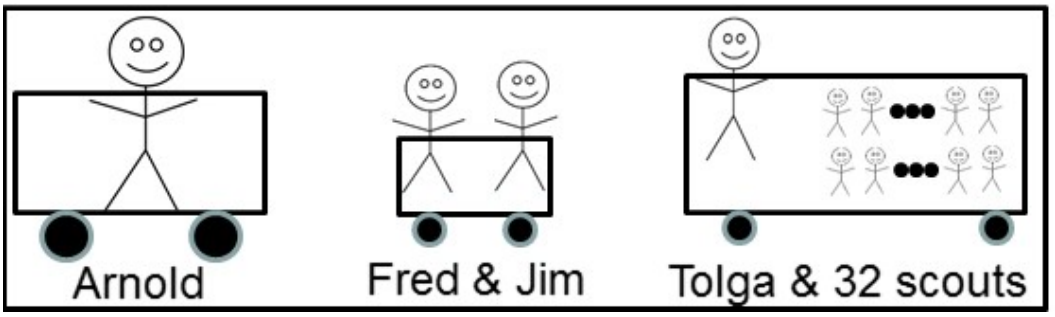


FIGURE 6.3 Three farmer teams compete in Analogy 6.1: (1) Arnold competes alone with his 2× bigger tractor and "the strongest farmer" reputation, (2) Fred and Jim compete together in a much smaller tractor than Arnold. (3) Tolga, along with 32 boy and girl scouts, compete together using a bus. Who wins?

Analogy 6.1 is depicted in Figure 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 Part I of the book. The interesting third alternative in Figure 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 Figure 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, i.e., the threading efficient was 100%, translating to $\eta = 1.0$ in Equation 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 8 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

the GPUs win is the fact that the GPU cores are much simpler and they work at lower speed. This allows the GPU chip designers to build a significantly higher number of cores into their GPU chips and the lower speed keeps the power consumptions below the magic 200–250 W, which is about the peak power you can consume from any semiconductor device (i.e., "chip").

Note that the power that is consumed by the GPU is **not** proportional to the frequency of each core; instead, the dependence is something like quadratic. In other words, a 4 GHz CPU core is expected to consume 16× more power than the same core working at 1 GHz. This very fact allows GPU manufacturers to pack hundreds or even thousands of cores into their GPUs without reaching the practical power consumption limits. This is actually exactly the same design philosophy behind multicore CPUs too. A single core CPU working at 4 GHz versus a dual-core CPU in which both cores work at 3 GHz could consume similar amounts of power. So, as long as the parallelization overhead is low (i.e., η is close to 1), a dual-core 3 GHz CPU is a better alternative than a single core 4GHz CPU. GPUs are nothing more than this philosophy taken to the ultimate extreme with one big exception: while the CPU multicore strategy calls for using multiple **sophisticated** (out-of-order) cores that work at lower frequencies, this design strategy only works if you are trying to put 2, 4, 8, or 16 cores inside the CPU. It simply won't work for 1000! So, the GPUs had to go through an additional step of making each core **simpler**. Simpler means that each core is *in-order* (see Section 3.2.1), work at lower frequencies, and their L1$ memories are not coherent. Many of these details are going to become clear as we go through the following few chapters. For now, the take-away from this section should be that GPUs incorporate a lot of architectural changes — as compared to CPUs — to provide a manageable execution environment for such a high core count.

6.3.2 CPU versus GPU Architectural Differences

Although we will learn the GPU architecture in great depth in the following chapters, for now, let's analyze the most important distinctions between CPUs and GPUs conceptually by just looking at the symbolic Figure 6.3. Here are our observations:

1. In our conceptualization of the "bus" that Tolga drives with the scouts in the back, Tolga does all of the driving. Scouts never get involved in it. In a GPU, the GPU cores do all of the execution, but the work orders always come from the CPU.

2. Tolga will be the one that is getting the coconuts and distributing them to the scouts. Scouts never directly get the coconuts by themselves. They simply wait in the bus until Tolga brings them the coconuts. In the GPU case, the GPU cores never actually get the data themselves. It always comes from the CPU side and the results always go back to the CPU side. So, the GPU simply acts as a computation accelerator in the background, working for the CPU for outsourcing certain tasks.

3. This type of an architecture only works well when there is a significant amount of parallel processing units, not just two or four. Indeed, in a GPU, nothing less than 32 things get executed at any point in time. This number — 32 — is almost the replacement of the "thread" concept we saw in the CPU world, something like a "super thread." There is even a name for it in the GPU world: 32 threads glued together are called a "warp." Although GPUs call the tasks executed by a single GPU core a *thread*, there is a necessity to define this other term *warp* to indicate the fact that no less than a warp's worth of threads get executed at any point in time.

4. The existence of the *warp* concept has dramatic implications on the GPU architecture. In Figure 6.3, we never talked about how the coconuts arrive in the bus. If you brought only 5 coconuts into the bus, 27 of the scouts would sit there doing nothing; so, the data elements must be brought in to the GPU in the same bulk amounts, although the unit of these data chunks is *half warp* or 16 elements.

5. The fact that the data arrives into the GPU cores in half warp *chunks* means that the memory sub-system that is bringing the data into the GPU cores should be bringing in the data 16-at-a-time. This implies a parallel memory subsystem that is capable of shuttling around data elements 16 at a time, either 16 floats or 16 integers, etc. This is why the GPU DRAM memory is made from GDDR5, which is *parallel memory*.

6. Because the CPU cores and GPU cores are completely different processing units, it is expected that they have different ISAs (instruction set architectures). In other words, they speak a different language. So, two different sets of instructions must be written: one for Tolga, one for the scouts. In the GPU world, a single compiler — nvcc — compiles both the CPU instructions and GPU instructions, although there are two separate *programs* that the developer must write. Thank goodness, the CUDA language combines them together and makes them so similar that the programmer can write both of these programs without having to learn two totally different languages.

6.4 CUDA VERSION OF THE IMAGE FLIPPER: IMFLIPG.CU

It is time to write and analyze our first CUDA program. Our first program will have the main() and everything else we saw in the CPU programs in Part I. If I didn't show you the additional CUDA code that is stuck in the few tens of lines inside the program, you would have thought that it is CPU code. This is good news in that although the ISAs of the CPU and GPU cores are vastly different, we write code for both of them in the same C language and add a few keywords to indicate whether a specific part of the code belongs to the CPU (*host side code*) or the GPU (*device side code*).

If this is the case, let's start writing the CPU code as if we had nothing to do with the GPU, with one exception. Because we know that we will eventually incorporate the GPU code into this program, we will call it imflipG.cu — the .cu extension denoting CUDA, which is the GPU version of the imflipP.c code we developed in Section 2.1. Remember that imflipP.c flipped an image either vertically or horizontally, based on the command line option that the user specified. It had two functions, MTFlipH() and MTFlipV(), that did the actual work of flipping the pixels in the image memory. The rest of the code in main() was responsible for reading and parsing the command lines and shuttling the data from/to these functions. What the CPU does in the imflipG.cu will be surprisingly similar to imflipP.c, because, remember from Analogy 6.1 that Tolga (the CPU core) will do a lot of the *onesy-twosy work* while the GPU cores will do the *massively parallel work*. In other words, why waste the scouts' time in Figure 6.3 to harvest just one coconut; you should have Tolga do it! If you assigned that task to the scouts, 31 of them will be idle. Worse yet, one scout works at 1 GHz, while Tolga can work at 2.5 GHz. So, the general principle is that the "serial" part of the code is more suitable for the CPU, even the "parallel" part is to an extent. However, the "massively parallel" part that requires at least the execution of 32-things or 64-things at a time is a perfect match for the GPU cores. The *programmer* decides how to split the tasks among the CPU and GPU cores.

Before I start explaining the details of imflipG.cu, let's go through the conceptual steps one by one. How is this program going to work?

1. First, the CPU will read the command line arguments and will parse them and place the parsed values in the appropriate CPU-side variables. Exactly the same story as the plain-simple CPU version of the code, the imflipP.c.

2. One of the command line variables will be the file name of the image file we have to flip, like the file that contains the dog picture, dogL.bmp. The CPU will read that file by using a CPU function that is called ReadBMP(). The resulting image will be placed inside a CPU-side array named `TheImg[]`. Notice that the GPU does absolutely **nothing** so far.

3. Once we have the image in memory and are ready to flip it, now it is time for the GPU's sun to shine! Horizontal or vertical flipping are both massively parallel tasks, so the GPU should do it. At this point in time, because the image is in a CPU-side array (more generally speaking, in CPU memory), it has to be transferred to the device side. What is obvious from this discussion is that the GPU has *its own memory*, in addition to the CPU's own memory — DRAM — that we have been studying since the first time we saw it in Section 3.5.

4. The fact that the CPU memory versus GPU memory are completely different memory areas (or "chips") should be pretty clear because the GPU is a different plug-in device that shares none of the electronic components with the CPU. The CPU memory is soldered on the motherboard and the GPU memory is soldered on the GPU plug-in card; the only way a data transfer can happen between these two memory areas is an explicit data transfer — using the available APIs we will see shortly in the following pages — through the PCI Express bus that is connecting them. I would like the reader to refresh his or her memory with Figure 4.3, where I showed how the CPU connected to the GPU through the X99 chipset and the PCI Express bus. The X99 chip facilitates the transfers, while the I/O portion of the CPU "chip" employs hardware to interface to the X99 chip and shuttle the data back and forth between the GPU memory and the DRAM of the CPU (by passing through the L3$ of the CPU along the way).

5. So, this *transfer* must take place from the CPU's memory into the GPU's memory before the GPU cores can do anything with the image data. This transfer occurs by using an API function that looks like an ordinary CPU function.

6. After this transfer is complete, now somebody has to tell the GPU cores what to do with that data. It is the *GPU side code* that will accomplish this. Well, the reality is that you should have transferred the code before the data, so by the time the image data arrives at the GPU cores they are aware of what to do with it. This implies that we are really transferring two things to the GPU side: (1) data to process, (2) code to process the data with (i.e., compiled GPU instructions).

7. After the GPU cores are done processing the data, another GPU→CPU transfer must transfer the results back to the CPU.

Using our Figure 6.3 analogy, it is as if Tolga is first giving a piece of paper with the instructions to the scouts so they know what to do with the coconuts (GPU side code), grabbing 32 coconuts at a time (read from CPU memory), dumping 32 coconuts at a time in front of the scouts (CPU→GPU data transfer), telling the scouts to execute their given instructions, which calls for harvesting the coconuts that just got dumped in front of them (GPU-side execution), and grabbing what is in front of them when they are done (GPU→CPU data transfer in the reverse direction) and putting the harvested coconuts back in the area where he got them (write the results back to the CPU memory).

CODE 6.1: imflipG.cu ... main() {...

First part of main() in imflipG.cu. `TheImg` and `CopyImg` are CPU-side image pointers, while `GPUImg`, `GPUCopyImg`, and `GPUResult` are GPU-side pointers.

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <iostream>
#include <ctype.h>
#include <cuda.h>

typedef unsigned char uch;
typedef unsigned long ul;
typedef unsigned int ui;

uch *TheImg, *CopyImg;                    // Where images are stored in CPU
uch *GPUImg, *GPUCopyImg, *GPUResult;  // Where images are stored in GPU
...
int main(int argc, char **argv)
{
   char InputFileName[255], OutputFileName[255], ProgName[255];
   ...
   strcpy(ProgName, "imflipG");
   switch (argc){
      case 5:  ThrPerBlk=atoi(argv[4]);
      case 4:  Flip = toupper(argv[3][0]);
      case 3:  strcpy(InputFileName, argv[1]);
               strcpy(OutputFileName, argv[2]);
               break;
      default: printf("\n\nUsage: %s InputFilename Outp ...");
            ...
   }
   ...
   TheImg = ReadBMPlin(InputFileName); // Read the input image
   CopyImg = (uch *)malloc(IMAGESIZE); // allocate space for the work copy
   ...
```

This process sounds a little inefficient due to the continuous back-and-forth data transfer, but don't worry. There are multiple mechanisms that Nvidia built into their GPUs to make the process efficient and the sheer processing power of the GPU eventually partially hides the underlying inefficiencies, resulting in a huge performance improvement.

6.4.1 imflipG.cu: Read the Image into a CPU-Side Array

OK. We are done with analogies. Time for real CUDA code. The main() function within imflipG.cu is a little long, so I will chop it up into small pieces. These pieces will look very much like the steps I just listed in the last page. First, let's see the variables involved in holding the original and processed CPU side images. Code 6.1 lists the first part of the

main() function and the following five pointers that facilitate image storage in CPU and GPU memory:

- `TheImg` variable is the pointer to the memory that will be malloc()'d by the ReadBMPLin() function to hold the image that is specified in the command line (e.g., dogL.bmp) in the CPU's memory. Notice that this variable, `TheImg`, is a pointer to the CPU DRAM memory.
- `CopyImg` variable is another pointer to the CPU memory and is obtained from a separate malloc() to allocate space for a copy of the original image (the one that will be flipped while the original is not touched). Note that we have done nothing with the GPU memory so far.
- As we will see very shortly, there are APIs that we will use to allocate memory in the *GPU memory*. When we do this, using an API called cudaMalloc(), we are asking the GPU memory manager to allocate memory for us **inside the GPU DRAM**. So, what the cudaMalloc() returns back to us is *a pointer to the GPU DRAM memory*. Yet, we will take that pointer and will store it in a *CPU-side variable*, `GPUImg`. This might look confusing at first because we are saving a pointer to the GPU side inside a CPU-side variable. It actually isn't confusing. Pointers are nothing more than "values" or more specifically 64-bit integers. So, they can be stored, copied, added, and subtracted in exactly the same way 64-bit integers can be. When do we store GPU-side pointers on the CPU side? The rule is simple: *Any* pointer that you will ever use in an API that is called by the CPU must be saved on the CPU side. Now, let's ask ourselves the question: will the variable `GPUImg` ever be used by the CPU side? The answer is definitely yes, because we will need to transfer data from the CPU to the GPU using cudaMalloc(). We know that cudaMalloc() is a CPU-side function, although its responsibility has a lot to do with the GPU. So, we need to store the pointers to *both* sides in CPU-side variables. We will most definitely use the same GPU-side pointer on the GPU side itself as well! However, we are now making a copy of it at the host (CPU), so the CPU has the chance of accessing it when it needs it. If we didn't do this, the CPU would never have access to it in the future and wouldn't be able to initiate memory transfers to the GPU that involved that specific pointer.
- The other GPU-side pointers, `GPUCopyImg` and `GPUResult`, have the same story. They are pointers to the GPU memory, where the resulting "flipped" image will be stored (`GPUResult`) and another temporary variable that the GPU code needs for its operation (`GPUCopyImg`). These two variables are CPU-side variables that store pointers that we will obtain from cudaMalloc(); storing GPU pointers in CPU variables shouldn't be confusing.

There are multiple `#include` directives you will see in every CUDA program, which are <cuda_runtime.h>, <cuda.h>, and <device_launch_parameters.h> to allow us to use Nvidia APIs. These APIs, such as cudaMalloc(), are the bridge between the CPU and the GPU side. Nvidia engineers wrote them and they allow you to transfer data between the CPU and the GPU side magically without worrying about the details.

Note the types that are defined here, ul, uch, and ui, to denote the `unsigned long`, `unsigned char`, and `unsigned int`, respectively. They are used so often that it makes the code cleaner define them as user-defined types. It serves, in this case, no purpose other than to reduce the clutter in the code. The variables to hold the file names are `InputFileName` and `OutputFileName`, which both come from the command line. The `ProgName` variable is hard-coded into the program for use in reporting as we will see later in this chapter.

CODE 6.2: imflipG.cu main() {...

Initializing and querying the GPU(s) using Nvidia APIs.

```
int main(int argc, char** argv)
{
   cudaError_t cudaStatus, cudaStatus2;
   cudaDeviceProp GPUprop;
   ul SupportedKBlocks, SupportedMBlocks, MaxThrPerBlk;  char SupportedBlocks[100];
   ...
   int NumGPUs = 0;            cudaGetDeviceCount(&NumGPUs);
   if (NumGPUs == 0){
      printf("\nNo CUDA Device is available\n"); exit(EXIT_FAILURE);
   }
   cudaStatus = cudaSetDevice(0);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaSetDevice failed! No CUDA-capable GPU installed?");
      exit(EXIT_FAILURE);
   }
   cudaGetDeviceProperties(&GPUprop, 0);
   SupportedKBlocks = (ui)GPUprop.maxGridSize[0] * (ui)GPUprop.maxGridSize[1] *
                      (ui)GPUprop.maxGridSize[2] / 1024;
   SupportedMBlocks = SupportedKBlocks / 1024;
   sprintf(SupportedBlocks, "%u %c", (SupportedMBlocks >= 5) ? SupportedMBlocks :
           SupportedKBlocks, (SupportedMBlocks >= 5) ? 'M' : 'K');
   MaxThrPerBlk = (ui)GPUprop.maxThreadsPerBlock;
   ...
```

6.4.2 Initialize and Query the GPUs

Code 6.2 shows the part of the main() that is responsible for *querying* the GPU(s), that is, gathering information about all of the existing GPUs in the system. There could be one (the most typical case), or more than one GPUs with different characteristics. The cudaGetDeviceCount() function, within the vast list of APIs that Nvidia provides us, writes the number of available GPUs in an `int` type variable that we defined, `NumGPUs`. If this variable is zero, clearly, we have no GPUs available and we can quit with a nasty message! Alternatively, if we have at least one GPU, we can choose the one that we want to execute our CUDA code on by using the API cudaSetDevice(). In this case we choose 0, which means *the first GPU*. Much like C variable indexes, GPU indexes range in 0, 1, 2... Once we choose a GPU, we can query that GPU's parameters using the cudaGetDeviceProperties() API function and place the results in a variable `GPUProp` (which is of type **cudaDeviceProp**). Let's look at a few features that we received from this API:

`GPUProp.maxGridSize[0]`, `GPUProp.maxGridSize[1]`, and `GPUProp.maxGridSize[2]` variables denote the maximum number of blocks that can be launched in x, y, and z dimensions, thereby allowing us to calculate the total number of blocks when you take the product of the three. We write the result into a variable named `SupportedKBlocks` after dividing by 1024 (to extract out the result in "kilo" terms). Similarly, divide it by 1024 again to get the result in "Mega." This is one simple example of how we can get the query answer to: "*what is the maximum number of blocks I can launch with this GPU?*" The upper limit of the number of threads you can launch in a block is given in the `MaxThrPerBlk` variable.

CODE 6.3: imflipG.cu main() {...

The part of main() in imflipG.cu that launches the GPU kernels.

```
uch *TheImg, *CopyImg;                  // Where images are stored in CPU
uch *GPUImg, *GPUCopyImg, *GPUResult;  // Where images are stored in GPU
#define IPHB        ip.Hbytes
#define IPH         ip.Hpixels
#define IPV         ip.Vpixels
#define IMAGESIZE   (IPHB*IPV)
...
int main(int argc, char** argv)
{
   cudaError_t cudaStatus, cudaStatus2;
   cudaEvent_t time1, time2, time3, time4;
   ui BlkPerRow, ThrPerBlk=256, NumBlocks, GPUDataTransfer;
   ...
   cudaEventCreate(&time1);      cudaEventCreate(&time2);
   cudaEventCreate(&time3);      cudaEventCreate(&time4);

   cudaEventRecord(time1, 0);  // Time stamp at the start of the GPU transfer
   // Allocate GPU buffer for the input and output images
   cudaStatus = cudaMalloc((void**)&GPUImg, IMAGESIZE);
   cudaStatus2 = cudaMalloc((void**)&GPUCopyImg, IMAGESIZE);
   if ((cudaStatus != cudaSuccess) || (cudaStatus2 != cudaSuccess)){
      fprintf(stderr, "cudaMalloc failed! Can't allocate GPU memory");
      exit(EXIT_FAILURE);
   }
   // Copy input vectors from host memory to GPU buffers.
   cudaStatus = cudaMemcpy(GPUImg, TheImg, IMAGESIZE, cudaMemcpyHostToDevice);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMemcpy CPU to GPU failed!");  exit(EXIT_FAILURE);
   }
   cudaEventRecord(time2, 0);  // Time stamp after the CPU --> GPU tfr is done
   BlkPerRow = (IPH + ThrPerBlk -1 ) / ThrPerBlk;
   NumBlocks = IPV*BlkPerRow;
   switch (Flip){
      case 'H': Hflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH);
           GPUResult = GPUCopyImg;     GPUDataTransfer = 2*IMAGESIZE;  break;
      case 'V': Vflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH, IPV);
          GPUResult = GPUCopyImg;      GPUDataTransfer = 2*IMAGESIZE;  break;
      case 'T': Hflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH);
           Vflip <<< NumBlocks, ThrPerBlk >>> (GPUImg, GPUCopyImg, IPH, IPV);
           GPUResult = GPUImg;         GPUDataTransfer = 4*IMAGESIZE;  break;
      case 'C': NumBlocks = (IMAGESIZE+ThrPerBlk-1) / ThrPerBlk;
           PixCopy <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IMAGESIZE);
           GPUResult = GPUCopyImg;     GPUDataTransfer = 2*IMAGESIZE;  break;
   }
   ...
```

6.4.3 GPU-Side Time-Stamping

You will know what a *block* means in detail very shortly when we get into the CUDA code execution details. For now, let's focus our attention on the CPU-GPU interaction; you can query the GPU right at the beginning of your CUDA code and based on the query results you can potentially execute your CUDA code using different parameters for optimum efficiency. Here is an analogy about how the CPU-GPU interaction works:

ANALOGY 6.2: *CPU-side versus GPU-side.*

CocoTown had their best year with more than 40 million coconuts ready to be harvested. Because they weren't used to harvesting so many coconuts, they decided to get help from their buddies at the moon; they knew that the moon city of cudaTown had a state-of-the-art technology to harvest the coconuts 10–20 times faster. This required them to send a spaceship to the moon with 40+ million coconuts in it. The folks at cudaTown were very organized; they were used to harvesting the coconuts in chunks they called "blocks" and required cocoTown to package each block in a single box, in sizes of 32, 64, 128, 256, or 1024. They requested cocoTown to inform them of the block size and how many blocks were coming in the spaceship.

CocoTown-cudaTown city officials had to come up with some ideas to communicate the block sizes and number of blocks. Additionally, cudaTown workers needed to *allocate* some space in their warehouse *before* the coconuts arrived. Luckily an engineer in cudaTown designed a satellite phone for the cocoTown engineers to call ahead and reserve space in cudaTown's warehouse, as well as to let them know about the block size and the number of blocks coming. One more thing: cudaTown was so big that the person responsible for allocating warehouse space for the coconuts had to give the "warehouse number" to the cocoTown people, so when the spaceship arrived cudaTown people would know where to store the coconuts.

This relationship was very exciting, however, cocoTown people didn't know what the right block size was and how to measure the time it took for space travel and harvesting in cudaTown. Because all of the work was going to be performed by the cudaTown folks, it was a good idea to make somebody at cudaTown responsible for timing all of these events; they hired an event manager for this. After long deliberations, cocoTown decided to ship the coconuts in a block size of 256, which required 166,656 blocks to be shipped. They put a notebook in the spaceship that includes warehouse addresses, and the other parameters they received from the satellite phone, so cudaTown people had a clear direction in harvesting the coconuts.

Analogy 6.2 actually has quite a bit of detail. Let's understand it.

- The city of cocoTown is the CPU and cudaTown is the GPU. Launching the spaceship between the two cities is equivalent to executing GPU code. The notebook they left in the spaceship contains the function parameters for the GPU-side function (for example, the Vflip()); without these parameters cudaTown couldn't execute any function.
- It is clear that the data transfer from the earth (cocoTown) to the moon (cudaTown) is a big deal; it takes a lot of time and might even marginalize the amazing execution speed at cudaTown. The spaceship is representing the data transfer engine, while the *space* itself is the PCI Express bus that is connecting the CPU and GPU.

- The speed of the spaceship is the PCI Express bus speed.
- The satellite phone represents the CUDA runtime API library for cocoTown and cudaTown to communicate. One important detail is that just because the satellite phone operator is in cudaTown, it doesn't guarantee that a copy is also saved in cudaTown; so, these parameters (e.g., warehouse number) must still be put inside the spaceship (written inside the notebook).
- Allocating the space in cudaTown's warehouse is equivalent to cudaMalloc(), which returns a pointer (the warehouse number in cudaTown). This pointer is given to cocoTown people, although it must be shipped back in the spaceship as a function parameter, as per my previous comment.
- Because the cocoTown people have no idea how things unfold as the spaceship leaves earth, they shouldn't be timing the events. cudaTown people should. I will provide a lot more detail on this below.

A GPU-side function — such as Vflip() — is termed a *kernel*. Code 6.3 shows how the CPU launches GPU kernels, which use the GPU as a co-processor; this is the most important part of the code to understand, so I will explain every piece of it in detail. Once you understand this part of code, it will be a smooth ride in the following chapters. First, much like the timing of the CPU code, we want to time how long it takes to do the CPU→GPU transfers as well as the GPU→CPU transfers. Additionally, we want to time the GPU code execution in between these two events. When we look at Code 6.3, we see that the following lines facilitate the timing of the GPU code:

```
cudaEvent_t time1, time2, time3, time4;
...
cudaEventCreate(&time1);   cudaEventCreate(&time2);
cudaEventCreate(&time3);   cudaEventCreate(&time4);
... // This is where we copy data from CPU to GPU
cudaEventRecord(time1, 0);  // Time stamp at the start of the GPU transfer
... // This is where we transfer data from the CPU to the GPU
cudaEventRecord(time2, 0);  // Time stamp after the CPU --> GPU tfr is done
...   // This is where we execute GPU code
cudaEventRecord(time3, 0);   // Time stamp after the GPU code execution
```

The variables `time1`, `time2`, `time3`, and `time4` are all CPU-side variables that store time-stamps during the transfers between the CPU and GPU, as well as the execution of the GPU code on the device side. A curious observation from the code above is that we only use Nvidia APIs to time-stamp the GPU-related events. Anything that touches the GPU must be time-stamped with the Nvidia APIs, specifically cudaEventRecord() in this case. But, why? Why can't we simply use the good-and-old gettimeofday() function we saw in the CPU code listings?

The answer is in Analogy 6.2: We totally rely on Nvidia APIs (the people from the moon) to time anything that relates to the GPU side. If we are doing that, we might as well let them time all of the space travel time, both forward and back. We are recording the beginning and end of these data transfers and GPU kernel execution as *events*, which allows us to use *Nvidia event timing APIs* to time them, such as cudaEventRecord(). To be used in this API an event must be first *created* using the cudaEventCreate() API. Because the event recording mechanism is built into Nvidia APIs, we can readily use them to time our GPU kernels and the CPU⟷GPU transfers, much like we did with our CPU code.

In Code 6.3, we use `time1` to time-stamp the very beginning of the code and `time2` to time-stamp the point when the CPU→GPU transfer is complete. Similarly, `time3` is when the GPU code execution is done and `time4` is when the arrival of the results to the CPU side is complete. The difference between any of these two time-stamps will tell us how long each one of these *events* took to complete. Not surprisingly, the *difference* must also be calculated by using the cudaEventElapsedTime() API — shown in Code 6.4 — in the CUDA API library, because the stored time-stamps are in a format that is also a part of the Nvidia APIs rather than ordinary variables.

6.4.4 GPU-Side Memory Allocation

Now, let's turn our attention to *GPU-side memory allocation* in Code 6.3. The following lines facilitate the creation of GPU-side memory by using the API cudaMalloc():

```
// Allocate GPU buffer for the input and output images
cudaStatus = cudaMalloc((void**)&GPUImg, IMAGESIZE);
cudaStatus2 = cudaMalloc((void**)&GPUCopyImg, IMAGESIZE);
if ((cudaStatus != cudaSuccess) || (cudaStatus2 != cudaSuccess)){
   fprintf(stderr, "cudaMalloc failed! Can't allocate GPU memory");
   exit(EXIT_FAILURE);
}
```

Nvidia Runtime Engine contains a mechanism — through the cudaMalloc() API — for the CPU to "ask" Nvidia to see if it can allocate a given amount of GPU memory. The answer is returned in a variable of type cudaError_t. If the answer is cudaSuccess, we know that the Nvidia runtime Engine was able to create the GPU memory we asked for and placed the starting point of this memory area in a pointer that is named `GPUImg`. Remember from Code 6.1 that the `GPUImg` is a CPU-side variable, pointing to a GPU-side memory address.

6.4.5 GPU Drivers and Nvidia Runtime Engine

As you see from the way the Nvidia APIs work, the *Nvidia Runtime Engine* is pretty much the *Nvidia Operating System* that manages your GPU resources, much like the regular OS managing the CPU resources. In Analogy 6.2, this is the cudaTown government offices. This Nvidia OS is placed inside your *GPU drivers*, the drivers that you install when you plug in your GPU card and get your CPU OS to recognize it. After you install the drivers, an icon on your SysTray will show up in Windows 10 Pro, as you see in Figure 6.4. The GPU drivers make it seamless for you to access GPU resources through a set of easy-to-use APIs to copy data back and forth between the CPU and GPU and execute GPU code. Figure 6.4 is a screen shot from a Windows 10 Pro PC that has an Nvidia GTX Titan Z GPU installed, which registers itself as two separate GPUs, containing 2880 cores each. The version of the Nvidia driver is 369.30 in this case. Some of the driver versions may be buggy and may not work well with certain APIs and these issues will be the continuous discussion topics in Nvidia forums. In other words, some driver versions will "drive" you nuts until you downgrade them back to a stable version or Nvidia fixes the bugs. If you create an Nvidia Developer account by going to developer.nvidia.com, you can be a part of these discussions and can provide answers when you build sufficient experience. These issues are nothing different than the bugs that Windows (or Mac) OS's have. It is just part of the OS development process.

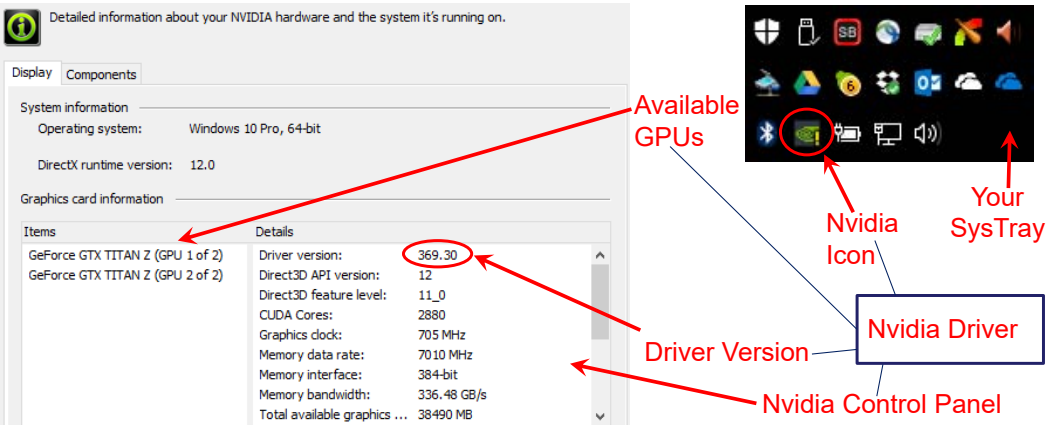


FIGURE 6.4 Nvidia Runtime Engine is built into your GPU drivers, shown in your Windows 10 Pro SysTray. When you click the Nvidia symbol, you can open the Nvidia control panel to see the driver version as well as the parameters of your GPU(s).

6.4.6 CPU→GPU Data Transfer

Once the GPU-side memory is allocated by the Nvidia runtime, we use another Nvidia API — named cudaMemcpy() — to perform the CPU→GPU data transfer. This API transfers data from a CPU-side memory area (pointed to by the `TheImg` pointer) to a GPU-side memory area (pointed to by the `GPUImg` pointer). Both of these pointers are declared in Code 6.1. The following lines perform the CPU→GPU data transfer:

```
// Copy input vectors from host memory to GPU buffers.
cudaStatus = cudaMemcpy(GPUImg, TheImg, IMAGESIZE, cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy CPU to GPU failed!");
   exit(EXIT_FAILURE);
}
cudaEventRecord(time2, 0);  // Time stamp after the CPU --> GPU tfr is done
```

Much like the memory allocation API cudaMalloc(), the memory transfer API cudaMemcpy() also uses the same status type cudaError_t, which returns cudaSuccess if the transfer completes without an error. If it doesn't, then we know that something went wrong during the transfer.

Going back to our Analogy 6.2, the cudaMemcpy() API is a specialized function that the spaceship has; a way to transfer 166,656 coconuts super fast in the spaceship, instead of worrying about each coconut one by one. Fairly soon, we will see that this memory transfer functionality will become a lot more sophisticated and the transfer time will end up being a big problem that will face us. We will see a set of more advanced memory transfer functions from Nvidia to ease the pain! In the end, just because the transfers take a lot of time, cudaTown people do not want to lose business. So, they will invent ways to make the coconut transfer a lot more efficient to avoid discouraging cocoTown people from sending business their way.

6.4.7 Error Reporting Using Wrapper Functions

We can further query the *reason* for the error by using the cudaGetErrorString() API function (which returns a pointer to a string that contains a simple explanation for the error such as "*invalid device pointer*"), but this is something that is not shown in Code 6.3. A simple modification of Code 6.3 would print the reason for the failure as follows:

```
if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy failed! %s",cudaGetErrorString(cudaStatus));
   exit(EXIT_FAILURE);
}
```

It is fairly common for programmers to write a *wrapper* function that wraps every single CUDA API call around some sort of error checking as shown below:

```
chkCUDAErr(cudaMemcpy(GPUImg, TheImg, IMAGESIZE, cudaMemcpyHostToDevice));
```

where the wrapper function — chkCUDAErr() — is one that we write within our C code, which directly uses the error code coming out of a CUDA API. An example wrapper function is shown below, which exits the program when a GPU runtime code is returned by any CUDA API:

```
// helper function that wraps CUDA API calls, reports any error and exits
void chkCUDAErr(cudaError_t ErrorID)
{
   if (ErrorID != CUDA_SUCCESS){
      printf("CUDA ERROR :::%\n", cudaGetErrorString(ErrorID));
      exit(EXIT_FAILURE);
   }
}
```

6.4.8 GPU Kernel Execution

After the completion of the CPU→GPU data transfer, we are now ready to *execute* the GPU kernel on the GPU side. The lines that relate to the GPU side code execution in Code 6.3 are shown below:

```
int IPH=ip.Hpixels;   int IPV=ip.Vpixels;
...
BlkPerRow = (IPH + ThrPerBlk -1 ) / ThrPerBlk;
NumBlocks = IPV*BlkPerRow;
switch (Flip){
   case 'H': Hflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH);
             GPUResult = GPUCopyImg;       GPUDataTransfer = 2*IMAGESIZE;
             break;
   case 'V': Vflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH, IPV);
             GPUResult = GPUCopyImg;       GPUDataTransfer = 2*IMAGESIZE;
             break;
   ...
```

The `Flip` parameter is set based on the command line argument the user enters. When the option 'H' is chosen by the user, the Hflip() GPU-side function is called and the three

specified arguments (`GPUCopyImg`, `GPUImg`, and `IPH`) are passed onto Hflip() from the CPU side. The 'V' option launches the Vflip() kernel with four arguments, as opposed to the three arguments in the Hflip() kernel; `GPUCopyImg`, `GPUImg`, `IPH`, and `IPV`. Once we look at the details of both kernels, it will be clear why we need the additional argument inside Vflip().

The following lines show what happens when the user chooses the 'T' (transpose) or 'C' (copy) options in the command line. I could have implemented *transpose* in a more efficient way by writing a specific kernel for it; however, my goal was to show how two kernels can be launched, one after the other. So, to implement 'T', I launched Hflip followed by Vflip, which effectively transposes the image. For the implementation of the 'C' option, though, I designed a totally different kernel PixCopy().

```
switch (Flip){
   ...
   case 'T': Hflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH);
             Vflip <<< NumBlocks, ThrPerBlk >>> (GPUImg, GPUCopyImg, IPH, IPV);
             GPUResult = GPUImg;           GPUDataTransfer = 4*IMAGESIZE;
             break;
   case 'C': NumBlocks = (IMAGESIZE+ThrPerBlk-1) / ThrPerBlk;
             PixCopy <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IMAGESIZE);
             GPUResult = GPUCopyImg;       GPUDataTransfer = 2*IMAGESIZE;
             break;
}
```

When the option 'H' is chosen by the user, the execution of the following line is handled by the Nvidia Runtime Engine, which involves launching the Hflip() kernel and passing the three aforementioned arguments to it from the CPU side.

```
Hflip <<< NumBlocks, ThrPerBlk >>> (GPUCopyImg, GPUImg, IPH);
```

Going forward, I will use the terminology *launching GPU kernels*. This contrasts with the terminology of *calling CPU functions*; while the CPU *calls* a function within its own *planet*, say *earth* according to Analogy 6.2, this is possibly not a good terminology for GPU kernels. Because the GPU really acts as a co-processor, plugged into the CPU using a far slower connection than the CPU's own internal buses, calling a function in a far location such as *moon* deserves a more dramatic term like *launching*. In the GPU kernel launch line above, Hflip() is the *GPU kernel name*, and the two parameters that are inside the ⋘ and ⋙ symbols (`NumBlocks` and `ThrPerBlk`) tell the Nvidia Runtime Engine what *dimensions* to run this kernel with; the first argument (`NumBlocks`) indicates *how many blocks* to launch, and the second argument (`ThrPerBlk`) indicates *how many threads* are launched in each block. Remember from Analogy 6.2 that these two numbers are what the cudaTown people wanted to know; the number of boxes (`NumBlocks`) and the number of coconuts in each box (`ThrPerBlk`). The generalized kernel launch line is as follows:

```
GPU Kernel Name <<< dimension, dimension >>> (arg1, arg2, ...);
```

where `arg1`, `arg2`, ... are the parameters passed from the CPU side onto the GPU kernel. In Code 6.3, the arguments are the two pointers (`GPUCopyImg` and `GPUImg`) that were given to us by cudaMalloc() when we created memory areas to store images in the GPU memory and `IPH` is a variable that holds the number of pixels in the horizontal dimension of the image (`ip.Hpixels`). GPU kernel Hflip() will need these three parameters during its execution and

would have no way of getting them had we not passed them during the kernel launch. Remember that the two launch dimensions in Analogy 6.2 were 166,656 and 256, effectively corresponding to the following launch line:

```
Hflip <<< 166,656, 256 >>> (GPUCopyImg, GPUImg, IPH);
```

This tells the Nvidia Runtime Engine to launch 166,656 blocks of the Hflip() kernel and pass the three parameters onto every single one of these blocks. So, the following blocks will be launched: Block 0, Block 1, Block 2, ... Block 166,655. Every single one of these blocks will execute 256 threads ($tid = 0$, $tid = 1$, ... , $tid = 255$), identical to the *pthreads* examples we saw in Part I of the book. What we are really saying is that we are launching a total of $166{,}656 \times 256 \approx 41$ M threads with this single launch line.

It is worth noting the difference between *Million* and *Mega*: Million threads means 1,000,000 threads, while Mega threads means $1024 \times 1024 = 1{,}048{,}576$ threads. Similarly *Thousand* is 1000 and *Kilo* is 1024. I will notate 41 Mega threads as 41 M threads. Same for 41,664 Kilo threads, being notated as 41,664 K threads. To summarize:

$$166{,}656 \times 256 = 42{,}663{,}936 \;=\; 41{,}664\,\text{K} \;=\; 40.6875\,\text{M threads} \;\approx 41\,\text{M threads}. \qquad (6.1)$$

One important note to take here is that the GPU kernel is a bunch of GPU machine code instructions, generated by the nvcc compiler, on the CPU side. These are the instructions for the cudaTown people to execute in Analogy 6.2. Let's say you wanted them to flip the order in which the coconuts are stored in the boxes and send them right back to earth. You then need to send them instructions about how to flip them (Hflip()). Because cudaTown people do not know what to do with the coconuts once they receive them. They need the coconuts (data), as well as the sequence of commands to execute (instructions). So, the compiled instructions also travel to cudaTown in the spaceship, written on a big piece of paper. At runtime, these instructions are executed on each block independently. Clearly, the performance of your GPU program depends on the efficiency of the kernel instructions, i.e., the programmer.

Let's refresh our memory with Code 2.8, which was the MTFlipH() CPU function that accepted a single parameter named `tid`. By looking at the `tid` parameter that is passed onto it, this CPU function knew "who it was." Based on who it was it processed a different part of the image, indexed by `tid` in some fashion. The GPU kernel Hflip() has stark similarities to it: This kernel acts almost exactly like its CPU sister MTFlipH() and the entire functionality of the Hflip() kernel will be dictated by a thread ID. Let's now compare them:

- MTFlipH() function is launched with 4–8 threads, while the Hflip() kernel is launched with almost 40 million threads. I talked about the *overhead* in launching CPU threads in Part I, which was really high. This overhead is almost negligent in the GPU world, allowing us to launch a million times more of them.
- MTFlipH() expects the Pthread API call to pass the `tid` to it, while the Hflip() kernel will receive its thread ID (0...255) directly from Nvidia Runtime Engine, at runtime. As the GPU programmer, all we have to worry about is to tell the kernel *how many* threads to launch and they will be numbered automatically.
- Due to the million times higher number of the threads we launch, some sort of hierarchy is necessary. This is why the thread numbering is broken down into two values: the *blocks* are little chunks that execute, with 256 threads in each. Each block executes completely independent from each other.

CODE 6.4: imflipG.cu main() {...

Finishing the GPU kernel execution and transferring the results back to the CPU.

```
int main(int argc, char** argv)
{
   ...
   // cudaDeviceSynchronize waits for the kernel to finish, and returns
   // any errors encountered during the launch.
   cudaStatus = cudaDeviceSynchronize();
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "\n\ncudaDeviceSynchronize error code %d ...\n", cudaStatus);
      exit(EXIT_FAILURE);
   }
   cudaEventRecord(time3, 0);
   // Copy output (results) from GPU buffer to host (CPU) memory.
   cudaStatus = cudaMemcpy(CopyImg, GPUResult, IMAGESIZE, cudaMemcpyDeviceToHost);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaMemcpy GPU to CPU failed!");
      exit(EXIT_FAILURE);
   }
   cudaEventRecord(time4, 0);

   cudaEventSynchronize(time1);     cudaEventSynchronize(time2);
   cudaEventSynchronize(time3);     cudaEventSynchronize(time4);
   cudaEventElapsedTime(&totalTime, time1, time4);
   cudaEventElapsedTime(&tfrCPUtoGPU, time1, time2);
   cudaEventElapsedTime(&kernelExecutionTime, time2, time3);
   cudaEventElapsedTime(&tfrGPUtoCPU, time3, time4);

   cudaStatus = cudaDeviceSynchronize();
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "\n Program failed after cudaDeviceSynchronize()!");
      free(TheImg);      free(CopyImg);     exit(EXIT_FAILURE);
   }
   WriteBMPlin(CopyImg, OutputFileName);  // Write the flipped image back to disk
   ...
```

6.4.9 Finish Executing the GPU Kernel

Although I explained the GPU kernel execution strictly in terms of the Hflip() kernel, almost everything is the same for the vertical flip (option 'V'), transpose (option 'T'), and copy (option 'C'). The only thing that changes is the kernel that is launched in these other cases. Vertical flip option causes the launch of the Vflip() kernel, which requires four function arguments, whereas the image transpose option simply launches both the horizontal and vertical flips, one after the other. Copy option launches another kernel PixCopy() that requires three arguments, the last one being different than the other kernels.

Code 6.4 shows the part of imflipG.cu() where we wait for the GPU to finish executing its kernel(s). When we launch one or more kernels, they continue executing until they are done. We must wait for the execution to be done using the lines below:

```
// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "\n\ncudaDeviceSynchronize error code %d...", cudaStatus);
   exit(EXIT_FAILURE);
}
```

The cudaDeviceSynchronize() function waits for every single launched kernel to complete its execution. The result could be an error, in which case cudaDeviceSynchronize() will return an error code. Otherwise, everything is good and we move onto reporting the results.

6.4.10 Transfer GPU Results Back to the CPU

Once the execution of the kernels that we have launched is complete, the result (the flipped image) is sitting in the GPU memory area, pointed to by the pointer `GPUResult`. We want to transfer it to the CPU memory area, pointed to by the pointer `CopyImg`. We can use the cudaMemcpy() function to do this with one exception: the very last argument of cudaMemcpy() specifies the direction of the transfer. Remember from Code 6.3 that we used the same API with the cudaMemcpyHostToDevice, which meant a CPU→GPU transfer. Now we use the cudaMemcpyDeviceToHost option, which means a GPU→CPU transfer. Aside from that everything else is the same as shown below:

```
cudaEventRecord(time3, 0);
// Copy output (results) from GPU buffer to host (CPU) memory.
cudaStatus = cudaMemcpy(CopyImg, GPUResult, IMAGESIZE, cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
   fprintf(stderr, "cudaMemcpy GPU to CPU failed!");
   exit(EXIT_FAILURE);
}
cudaEventRecord(time4, 0);
```

6.4.11 Complete Time-Stamping

We time-stamp the end of the GPU→CPU transfer with `time4` and we are ready to calculate the amount of time each event took as follows:

```
cudaEventSynchronize(time1);     cudaEventSynchronize(time2);
cudaEventSynchronize(time3);     cudaEventSynchronize(time4);
cudaEventElapsedTime(&totalTime, time1, time4);
cudaEventElapsedTime(&tfrCPUtoGPU, time1, time2);
cudaEventElapsedTime(&kernelExecutionTime, time2, time3);
cudaEventElapsedTime(&tfrGPUtoCPU, time3, time4);
```

The API cudaEventSynchronize() tells Nvidia runtime to synchronize a given event to ensure that the variables `time1` ... `time4` have correct timing values. The cudaEventElapsedTime() API is used to calculate the difference between a pair of them, because they are not simple types. For example, the difference between `time` and `time4` indicates how long it took for the data to depart from the CPU and arrive at the GPU, get processed by the GPU, and return back to CPU memory. The rest are easy to follow.

CODE 6.5: imflipG.cu main() ...}

Last part of main() reports the results and cleans up GPU and CPU memory areas.

```
int main(int argc, char** argv)
{
   ...
   printf("--...--\n"); printf("%s ComputeCapab=%d.%d [supports max %s blocks]\n",
        GPUprop.name,GPUprop.major,GPUprop.minor,SupportedBlocks); printf("...\n");
   printf("%s %s %s %c %u [%u BLOCKS, %u BLOCKS/ROW]\n", ProgName, InputFileName,
       OutputFileName,Flip, ThrPerBlk, NumBlocks, BlkPerRow);
   printf("-------------------- ... ----------------------------\n");
    printf("CPU->GPU Transfer = %5.2f ms ... %4d MB ... %6.2f GB/s\n",
   tfrCPUtoGPU, IMAGESIZE / 1024 / 1024, (float)IMAGESIZE / (tfrCPUtoGPU *
       1024.0*1024.0));
   printf("Kernel Execution = %5.2f ms ... %4d MB ... %6.2f GB/s\n",
   kernelExecutionTime, GPUDataTransfer / 1024 / 1024, (float)GPUDataTransfer /
       (kernelExecutionTime * 1024.0*1024.0));
   printf("GPU->CPU Transfer = %5.2f ms ... %4d MB ... %6.2f GB/s\n",
   tfrGPUtoCPU, IMAGESIZE / 1024 / 1024, (float)IMAGESIZE / (tfrGPUtoCPU *
       1024.0*1024.0));
   printf("Total time elapsed = %5.2f ms\n", totalTime);
   printf("-------------------- ... ----------------------------\n");
   // Deallocate CPU, GPU memory and destroy events.
   cudaFree(GPUImg);          cudaFree(GPUCopyImg);
   cudaEventDestroy(time1);   cudaEventDestroy(time2);
   cudaEventDestroy(time3);   cudaEventDestroy(time4);
   // cudaDeviceReset must be called before exiting, so profiling and tracing tools
   // like Parallel Nsight and Visual Profiler shows complete traces.
   cudaStatus = cudaDeviceReset();
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaDeviceReset failed!");
      free(TheImg);   free(CopyImg);  exit(EXIT_FAILURE);
   }
   free(TheImg);   free(CopyImg);
   return(EXIT_SUCCESS);
}
```

6.4.12 Report the Results and Cleanup

Code 6.5 shows how we report the results of the program. Some of these reported results are coming from our query of the GPU. For example, `GPUprop.name` is the name of the GPU such as "GeForce GTX Titan Z." I will go over the detailed output in Section 6.5.4.

Much like every malloc() must be cleaned up with a corresponding free(), every cudaMalloc() must be cleaned up with a corresponding cudaFree() by telling the Nvidia runtime that you no longer need that GPU memory area. Similarly, every event you created using cudaEventCreate() must now be destroyed by cudaEventDestroy(). After all of this is done, we call cudaDeviceReset(), which tells Nvidia runtime that we are done using the GPU. Then, we go ahead and free() our CPU memory areas and imflipG.cu is done!

CODE 6.6: imflipG.cu ReadBMPlin() {...}, WriteBMPlin() {...}
Functions that read/write a BMP file in linear indexing, rather than the x, y indexing.

```
// Read a 24-bit/pixel BMP file into a 1D linear array.
// Allocate memory to store the 1D image and return its pointer.
uch *ReadBMPlin(char* fn)
{
   static uch *Img;
   FILE* f = fopen(fn, "rb");
   if (f == NULL){ printf("\n\n%s NOT FOUND\n\n", fn); exit(EXIT_FAILURE); }
   uch HeaderInfo[54];
   fread(HeaderInfo, sizeof(uch), 54, f); // read the 54-byte header
   // extract image height and width from header
   int width = *(int*)&HeaderInfo[18];   ip.Hpixels = width;
   int height = *(int*)&HeaderInfo[22];  ip.Vpixels = height;
   int RowBytes = (width * 3 + 3) & (~3);  ip.Hbytes = RowBytes;
   memcpy(ip.HeaderInfo, HeaderInfo,54);  //save header for re-use
   printf("\n Input File name: %17s (%u x %u) File Size=%u", fn,
   ip.Hpixels, ip.Vpixels, IMAGESIZE);
   // allocate memory to store the main image (1 Dimensional array)
   Img  = (uch *)malloc(IMAGESIZE);
   if (Img == NULL) return Img;   // Cannot allocate memory
   // read the image from disk
   fread(Img, sizeof(uch), IMAGESIZE, f);   fclose(f);      return Img;
}

// Write the 1D linear-memory stored image into file.
void WriteBMPlin(uch *Img, char* fn)
{
   FILE* f = fopen(fn, "wb");
   if (f == NULL){ printf("\n\nFILE CREATION ERROR: %s\n\n", fn); exit(1); }
   fwrite(ip.HeaderInfo, sizeof(uch), 54, f); //write header
   fwrite(Img, sizeof(uch), IMAGESIZE, f);  //write data
   printf("\nOutput File name: %17s (%u x %u) File Size=%u", fn, ip.Hpixels,
        ip.Vpixels, IMAGESIZE);
   fclose(f);
}
```

6.4.13 Reading and Writing the BMP File

Code 6.6 shows the two functions that read and write a BMP image to/from CPU memory. The difference between these functions and the ones we saw in Code 2.4 and Code 2.5 is that they read the image in a *linear* memory area, hence the suffix "lin" at the end of the function names. Linear memory area means a single index, rather than an x, y index to store the pixels. This makes the image reading from the disk extremely simple because the image is, indeed, stored in a linear fashion on the disk. However, when we are processing it, we will need to convert it back to the x, y format using a very simple transformation:

$$\text{Pixel Coordinate Index} = (x, y) \quad \longrightarrow \quad \text{Linear Index} = x + (y \times ip.Hpixels) \tag{6.2}$$

CODE 6.7: imflipG.cu Vflip() {...}

The GPU kernel Vflip() that flips an image vertically.

```
// Kernel that flips the given image vertically
// each thread only flips a single pixel (R,G,B)
__global__
void Vflip(uch *ImgDst, uch *ImgSrc, ui Hpixels, ui Vpixels)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;

   ui BlkPerRow = (Hpixels + ThrPerBlk - 1) / ThrPerBlk; // ceil
   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 MYmirrorrow = Vpixels - 1 - MYrow;
   ui MYsrcOffset = MYrow    * RowBytes;
   ui MYdstOffset = MYmirrorrow * RowBytes;
   ui MYsrcIndex = MYsrcOffset + 3 * MYcol;
   ui MYdstIndex = MYdstOffset + 3 * MYcol;

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

6.4.14 Vflip(): The GPU Kernel for Vertical Flipping

Code 6.7 shows the GPU kernel named Vflip(). This kernel shows what each *thread* does. As we computed a few pages ago, there are 40 million plus threads that will run this kernel, assuming that we launch it with the dimensions Vflip $\lll 166656, 256 \ggg$ (...). If we launch it with higher dimensions, there will be even more threads launched. Every single line of code we saw up to this point in this chapter (for that matter, in this book) was CPU code. Code 6.7 is the first GPU code we are seeing. Although Code 6.7 is written in C and everything else so far was also written in C, what we care about is the final set of CPU instructions that Code 6.7 will be compiled into. While everything we saw so far was going to be compiled into x64 — Intel 64-bit Instruction Set Architecture (ISA) — instructions, Code 6.7 will be compiled into the Nvidia GPU ISA named Parallel Thread Execution (PTX). Much like Intel and AMD extend their ISAs in every new generation of CPUs, so does Nvidia in every generation of GPUs. For example, as of late 2015, the dominant ISA was PTX 4.3 and PTX 5.0 was in development.

One big difference between a CPU ISA and GPU ISA is that while an x64 compiled CPU ISA output contains x86 instructions that will be executed one-by-one at runtime (i.e., fully compiled), PTX is actually an intermediate representation (IR). This means that it is "half-compiled," much like a Java Byte code. The Nvidia Runtime Engine takes the

PTX instructions and further *half-compiles* them at runtime and feeds the *full-compiled* instructions into the GPU cores. In Windows, all of the "Nvidia magic code" that facilitates this "further-half-compiling" is built into a Dynamic Link Library (DLL) named cudart (CUDA Run Time). There are two flavors: in modern x64 OSs, it is cudart64 and in old 32-bit OSs, it is cudart32, although the latter should never be used because all modern Nvidia GPUs require a 64-bit OS for efficient use. In my Windows 10 Pro PC, for example, I was using cudart64_80.dll (Runtime Dynamic Link Library for CUDA 8.0). This file is not something you explicitly have to worry about; the nvcc compiler will put it in the executable directory for you. I am just mentioning it so you are aware of it.

Let's compare Code 6.7 to its CPU sister Code 2.7. Let's assume that both of them are trying to flip the astronaut.bmp image in Figure 5.1 vertically. astronaut.bmp is a 7918×5376 image that takes $\approx$121 MB on disk. How would their functionality be different?

- For starters, assume that Code 2.7 uses 8 threads; it will assign the flipping task of 672 lines to each thread (i.e., $672 \times 8 = 5376$). Each thread will, then, be responsible for *processing* $\approx$15 MB of *information* out of the entire image, which contains $\approx$121 MB of information in its entirety. Because the launch of more than 10–12 threads will not help on an 8C/16T CPU, as we witnessed over and over again in Part I, we cannot really do better than this when it comes to the CPU.

- The GPU is different though. In the GPU world, we can launch a gazillion threads without incurring any overhead. What if we went all the way to the bitter extreme and had *each thread swap a single pixel*? Let's say that each GPU thread takes a single pixel's RGB value (3 bytes) from the source image GPU memory area (pointed to by `*ImgSrc`) and writes it into the intended *vertically flipped* destination GPU memory area (pointed to by `*ImgDst`).

- Remember, in the GPU world, our unit of launch is *blocks*, which are clumps of threads, each clump being 32, 64, 128, 256, 512, or 1024 threads. Also remember that it cannot be less than 32, because "32" is the smallest amount of parallelism we can have and 32 threads are called a *warp*, as I explained earlier in this chapter. Let's say that each one of our blocks will have 256 threads to flip the astronaut image. Also, assume that we are processing one row of the image at a time using multiple blocks. This means that we need $\lceil 7918/256 \rceil = 31$ blocks to process each row.

- Because we have 5376 rows in the image, we will need to launch $5376 \times 31 = 166{,}656$ blocks to vertically flip the astronaut.bmp image.

- We observe that 31 blocks-per-row will yield some minor loss, because $31 \times 256 = 7936$ and we will have 18 threads $(7936 - 7918 = 18)$ doing nothing to process each row of the image. Oh well, nobody said that massive parallelism doesn't have its own disadvantages.

- This problem of "useless threads" is actually exacerbated by the fact that not only are these threads useless, but also they have to check to see if they are the useless threads as shown in the line below:

```
if (MYcol >= Hpixels) return;   // col out of range
```

  This line simply says "if my *tid* is between 7918 and 7935 I shouldn't do anything, because I am a useless thread." Here is the math: We know that the image has

7918 pixels in each row. So, the threads $tid = 0...7917$ are useful, and because we launched 7936 threads ($tid = 0...7935$), this designates threads ($tid = 7918...7935$) as useless.

- Don't worry about the fact that we do not see *tid* in the comparison; rather, we see `MYcol`. When you calculate everything, the underlying math ends up being exactly what I just described. The reason for using a variable named `MYcol` is because the code has to be parametric, so it works for any size image, not just the astronaut.bmp.
- Why is it so bad if only 18 threads check this? After all, 18 is only a very small percentage of the 7936 total threads. Well, this is not what happens. Like I said before, what you are seeing in Code 6.7 is what *every* thread executes. In other words, *all 7936 threads* must execute the same code and must check to see if they are useless, just to find that they aren't useless (most of the time) or they are (only a fraction of the time). So, with this line of code, we have introduced overhead to *every thread*. How do we deal with this? We will get to it, I promise. But, not in this chapter ... For now, just know that even with these inefficiencies — which are an artifact of massively parallel programming — our performance is still acceptable.
- And, finally, the `__global__` is the third CUDA symbol that I am introducing here, after ⋘ and ⋙. If you precede any ordinary C function with `__global__` the nvcc compiler will know that it is a *GPU-side* function and it compiles it into PTX, rather than the x64 machine code output. There will be a few more of these CUDA designators, but, aside from that, CUDA looks exactly like C.

6.4.15 What Is My Thread ID, Block ID, and Block Dimension?

We will spend a lot of time explaining the deep details of Vflip() in the next chapter, but for now all we want to know is how this function calculates who it is and which portion of the processing it is responsible for. Let's refresh our memory with the CPU Code 2.7:

```
void *MTFlipV(void* tid)
{
   long ts = *((int *) tid);           // My thread ID is stored here
   ts *= ip.Hbytes/NumThreads;         // start index
   long te = ts+ip.Hbytes/NumThreads-1; // end index
   ...
   for(col=ts; col<=te; col+=3){
      row=0;
      while(row<ip.Vpixels/2){
         pix.B=TheImage[row][col];  pix.G=TheImage[row][col+1];   pix.R=...
         ...
```

Here, the `ts` and `te` variables computed the starting and ending row numbers in the image, respectively. Vertical flipping was achieved by two nested loops, one scanning the columns and the other scanning the rows. Now, let's compare this to the Vflip() function in Code 6.7:

```
__global__
void Vflip(uch *ImgDst, uch *ImgSrc, ui Hpixels, ui Vpixels)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;
   ui BlkPerRow = (Hpixels + ThrPerBlk - 1) / ThrPerBlk; // ceil
   ui RowBytes = (Hpixels * 3 + 3) & (~3);
```

We see that there are major similarities and differences between CPU and GPU functions. The task distribution in the GPU function is completely different because of the blocks and the number of threads in each block. So, although the GPU still calculates a bunch of indexes, they are completely different than the CPU function. GPU first wants to know how many threads were launched with each block. The answer is in a special GPU value named `blockDim.x`. We know that this answer will be 256 in our specific case because we specified 256 threads to be launched in each block (Vflip$\lll \ldots, 256 \ggg$). So, each block contains 256 threads, with thread IDs 0...255. The specific thread ID of this thread is in `threadIDx.x`. It also wants to know, out of the 166,656 blocks, what is its own block ID. This answer is in another GPU value named `blockIdx.x`. Surprisingly, it doesn't care about the total number of blocks (166,656) in this case. There will be other programs that do.

It saves its block ID and thread ID in two variables named `bid` and `tid`. It then computes a global thread ID (`gtid`) using a combination of these two. This `gtid` gives a unique ID to each one of the launched GPU threads (out of the total $166{,}656 \times 256 \approx 41\,\text{M}$ threads), thereby *linearizing them*. This concept is very similar to how we linearized the pixel memory locations on the disk according to Equation 6.2. However, an immediate correlation between linear GPU thread addresses and linear pixel memory addresses is not readily available in this case due to the existence of the useless threads in each row. Next, it computes the *blocks per row* (`BlkPerRow`), which was 31 in our specific case. Finally, because the value of the number of horizontal pixels (7918) was passed onto this function as the third parameter, it can compute the total number of bytes in a row of the image ($3 \times 7918 = 23{,}754\,\text{Bytes}$) to determine the *byte index* of each pixel.

After these computations, the kernel then moves onto computing the row and column index of the single pixel that it is responsible for copying as follows:

```
   ui MYrow = MYbid / BlkPerRow;
   ui MYcol = MYgtid - MYrow*BlkPerRow*ThrPerBlk;
   if (MYcol >= Hpixels) return;    // col out of range
   ui MYmirrorrow = Vpixels - 1 - MYrow;
   ui MYsrcOffset = MYrow     * RowBytes;
   ui MYdstOffset = MYmirrorrow * RowBytes;
   ui MYsrcIndex = MYsrcOffset + 3 * MYcol;
   ui MYdstIndex = MYdstOffset + 3 * MYcol;
```

After these lines, the source pixel memory address is in `MYsrcIndex` and the destination memory address is in `MYdstIndex`. Because each pixel contains three bytes (RGB) starting at that address, the kernel copies three consecutive bytes starting at that address as follows:

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

Let's now compare this to CPU Code 2.7. Because we could only launch 4–8 threads, instead of the massive 41 M threads we just witnessed, one striking observation from the GPU kernel is that the `for` loops are gone! In other words, instead of explicitly scanning over the columns and rows, like the CPU function has to, we don't have to loop over anything. After all, the entire purpose of the loops in the CPU function was to scan the pixels with some sort of two-dimensional indexing, facilitated by the `row` and `column` variables. However, in the GPU kernel, we can achieve this functionality by using the `tid` and `bid`, because we know the precise relationship of the coordinates and the `tid` and `bid` variables.

CODE 6.8: imflipG.cu Hflip() {...}

The GPU kernel Hflip() that flips an image horizontally.

```
// Kernel that flips the given image horizontally
// each thread only flips a single pixel (R,G,B)
__global__
void Hflip(uch *ImgDst, uch *ImgSrc, ui Hpixels)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;

   ui BlkPerRow = (Hpixels + ThrPerBlk -1 ) / ThrPerBlk; // ceil
   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];
}
```

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 have 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 127,712,256 Bytes. Each each block has 256 threads, we need a total of $\left\lceil \frac{127,712,256}{256} \right\rceil = 498,876$ blocks launched, each block containing 256 threads. In this specific

TABLE 6.1 CUDA keyword and symbols that we learned in this chapter.

CUDA Keyword	Description	Examples

__global__	precedes a device-side function (i.e., a kernel)	`__global__`
		`void PixCopy(uch *ImgDst, uch *ImgSrc, ui FS)`
		`{`
		`  ...`
		`}`

⋘, ⋙	Launch a device-side kernel from the host-side	`Hflip<<<NumBlocks, ThrPerBlk>>>(..., ..., ...);`
		
		`Vflip<<<NumBlocks, ThrPerBlk>>>(..., ..., ...);`
		
		`PixCopy<<<NumBlocks, ThrPerBlk>>>(..., ..., ...);`

case we got lucky. However, if we had 20 more bytes in the file, we would have 236 threads wasted out of the 256 in the very last block. This is why we still have to put the following `if` statement in the kernel to check for this condition as shown below:

```
if (MYgtid > FS) return;          // outside the allocated memory
```

The `if` statement in Code 6.9, much like the ones in Code 6.7 and Code 6.8, checks if "it is a useless thread" and does nothing if it is. The performance impact of this line is similar to the previous two kernels: although this condition will be only true for a negligible number of threads, *every* thread still has to execute it for every single byte they are copying. We can improve this, but we will save all of these improvement ideas to the upcoming chapters. For now, it is worth noting that the performance impact of this `if` statement in the PixCopy() kernel is far worse than the one we saw in the other two kernels. The PixCopy() kernel has a much finer granularity as it copies only a single byte. Because of this, there are only 6 lines of C code in PixCopy(), one of which being the `if` statement. In contrast, Code 6.7 and Code 6.8 contain 16–17 lines of code, thereby making the impact of one added line much less. Although "lines of code" clearly does not translate to "the number of cycles that it takes for the GPU core to execute the corresponding instructions" one-on-one, we can still get an idea about the magnitude of the problem.

6.4.19 CUDA Keywords

At this point, it is a good idea to summarize what makes the compiler distinguish between a C program and a CUDA program. There are a few CUDA-specific identifiers like the triple bracket pair (⋘, ⋙), which let the compiler know that these are CUDA function calls from the host-side. Furthermore, there are a few identifiers like `__global__` to let the compiler know that the upcoming function must be compiled into PTX (GPU instruction set), not x64 (CPU instruction set). Table 6.1 lists these two most common CUDA keywords, which we learned in this chapter. There will be more we will learn in the following chapters; as we learn more, we will expand our table.

6.5 CUDA PROGRAM DEVELOPMENT IN WINDOWS

In this section, my goal was to show a working CUDA example: The imflipG.cu program, based on its command line parameters, performs one of four operations: flips an image in

the (1) horizontal or (2) vertical direction, (3) copies it to another image, or (4) transposes it. The command line to run imflipG.cu is as follows:

```
imflipG astronaut.bmp a.bmp V 256
```

This vertically flips an image named astronaut.bmp and writes the flipped image into another file named a.bmp. The 'V' option is the flip direction (vertical) and 256 is the *number of threads in each block*, which is what we will plug into the second argument of our kernel dimensions with the launch parameters Vflip $\lll \ldots, 256 \ggg (\ldots)$. We could choose 'H', 'C', or 'T' for horizontal flip, copy, or transpose operations.

6.5.1 Installing MS Visual Studio 2015 and CUDA Toolkit 8.0

To be able to develop CUDA code in a Windows PC, the best editor/compiler (Integrated Development Environment or IDE) to use is Microsoft Visual Studio 2015. I will call it VS 2015 going forward. With the release of CUDA 8.0, VS 2015 started working. Earlier versions of CUDA, such as even as late as CUDA 7.5, did not work with VS 2015. They only worked with VS 2013. In mid-2016, CUDA 8.0 was released, which allowed Pascal architecture-based GPUs to be used. In this book, VS 2015 and CUDA 8.0 will be used and notes will be made when certain features only apply to CUDA 8.0 and not the previous versions. To be able to compile CUDA 8.0 using VS 2015, you first have to install VS 2015 in your machine, followed by the Nvidia CUDA 8.0 toolkit. This toolkit will install the plug-in for VS 2015, allowing it to edit and compile CUDA code. Both of these are easy steps.

To install VS 2015, follow the instructions below:

- If you have a license for VS 2015 Professional, you can start the installation from the DVD. Otherwise, in your Internet Explorer (or Edge) browser, go to:

  https://www.visualstudio.com/downloads/

  Click "Free Download" under Visual Studio Community

- Once the download is complete, launch the installer and choose "everything" (instead of the limited set of default modules) to avoid some unforeseen issues, especially if you are installing VS 2015 for the first time.

- CUDA 8.0 will need quite a bit of the VS 2015 features. So, choosing *every* option takes up close to 40–50 GB of hard disk space. If you do not choose every option, you will get an error when you are trying to compile even the simplest CUDA code.

- I have not tried to pick and choose specific options to see which ones are absolutely needed to prevent this error. You can surely try to do it and spend a lot of time in the forums, but I can only guarantee that what is described here will work when you install every option of VS 2015.

After VS 2015 is installed, you need to install the CUDA Toolkit 8.0 as follows:

- In your Internet Explorer (or Edge) browser, go to:

  https://developer.nvidia.com/cuda-toolkit

- Click "Download" and download the toolkit if you prefer Local install (it will be about a Gigabyte). Local install option allows you to keep the installer in a temporary directory and double-click on the installer.

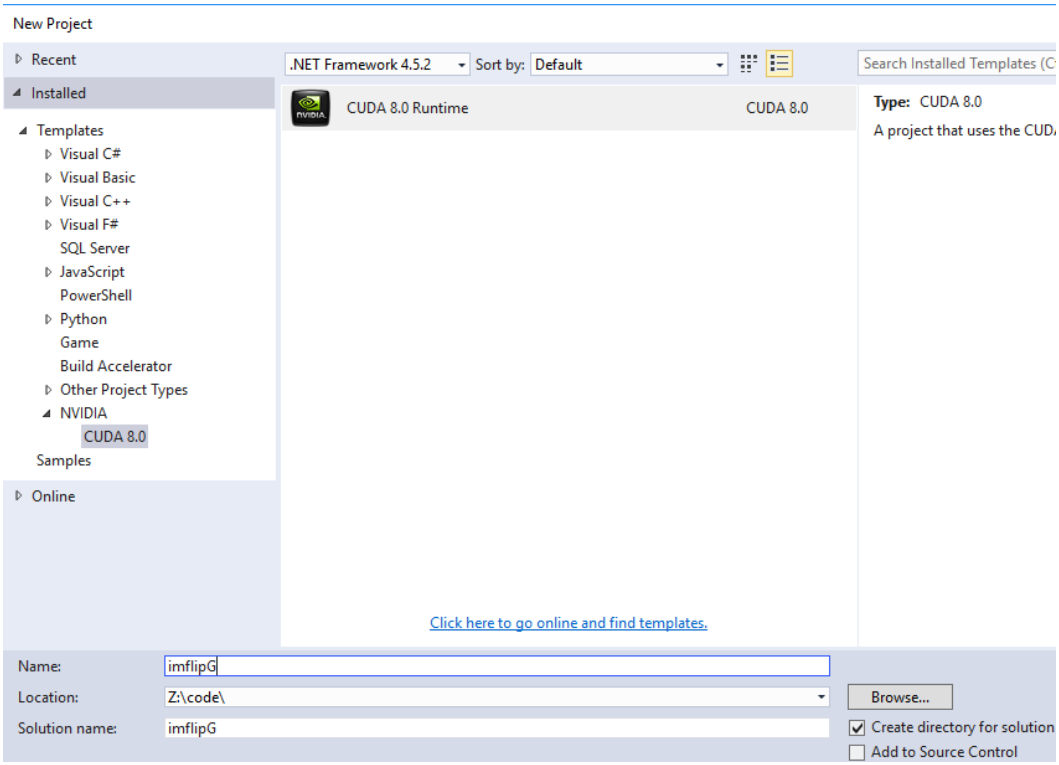


FIGURE 6.5 Creating a Visual Studio 2015 CUDA project named imflipG.cu. Assume that the code will be in a directory named Z:\code\imflipG in this example.

- You can select the Network Installer, which will install straight from the Internet. After spending 50 GB of your hard disk space on VS 2015, you will not be terribly worried about another GB. Either option is fine. I always choose the network installer, so I don't have to worry about deleting the local installer code after the installation is done.
- Click OK for the default extraction paths. The screen may go blank for a few second while the GPU drivers are being configured. After the installation is complete, you will see a new option in your Visual Studio, named "NSIGHT."

6.5.2 Creating Project imflipG.cu in Visual Studio 2015

To create a GPU program, we first clock "Create New Project" in VS 2015 and the dialog box, shown in Figure 6.5 opens up. Visual Studio wants a solution name and a name for the project. By default, it populates the same name into their respective boxes. We will not change this. To create your project, choose the name of the directory in "Location" and choose the project name as "imflipG." In the example shown in Figure 6.5, I am using Z:\code as the directory name and imflipG as the project name. Once you make these selections and click OK, VS 2015 will create a solution directory under Z:\code\imflipG.

A screen shot of the solution directory Z:\code\imflipG is shown in Figure 6.6. If you go into this directory, you will see another directory named imflipG, which is where your

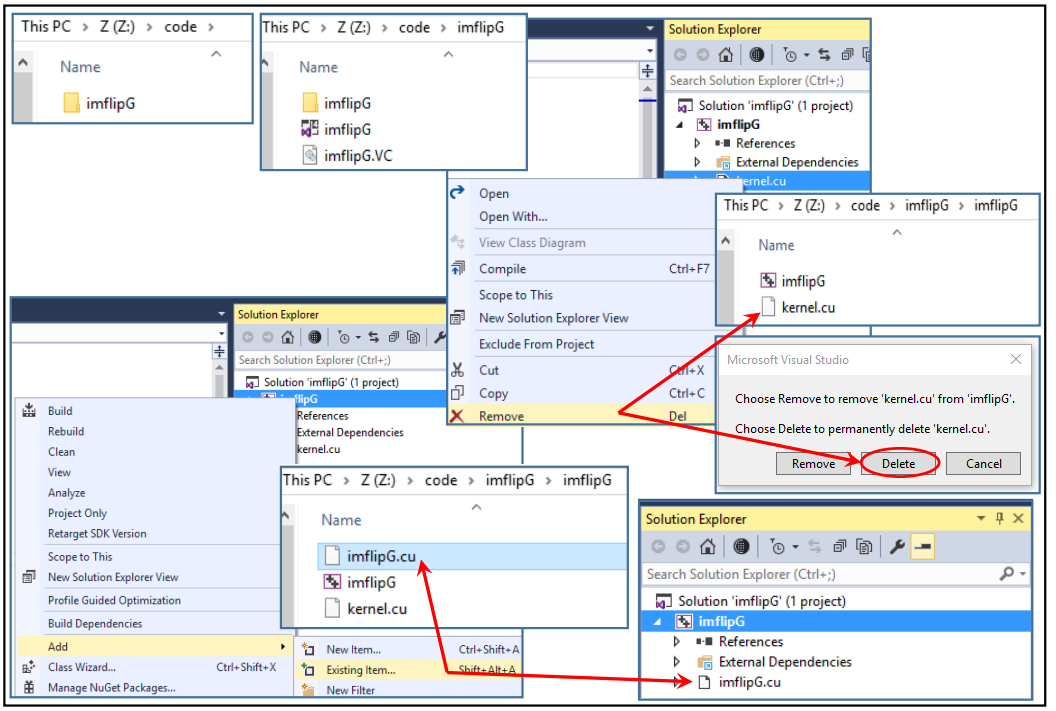


FIGURE 6.6 Visual Studio 2015 source files are in the Z:\code\imflipG\imflipG directory. In this specific example, we will remove the default file, kernel.cu, that VS 2015 creates. After this, we will add an existing file, imflipG.cu, to the project.

project source files are going to be placed by VS 2015; so, the source files will be under the directory Z:\code\imflipG\imflipG. Go into Z:\code\imflipG\imflipG; you will see a file named kernel.cu and another file we don't care about. The kernel.cu file is created in the source file directory automatically by VS 2015 by default.

At this point, there are three ways you can develop your CUDA project:

1. You can enter your code inside kernel.cu by using it as a template and delete the parts you don't want from it and compile it and run it as your only kernel code.

2. You can rename kernel.cu as something else (say, imflipG.cu) by right clicking on it inside VS 2015. You can clean what is inside the renamed imflipG.cu and put your own CUDA code in there. Compile it and run it.

3. You can remove the kernel.cu file from the project and add another file, imflipG.cu, to the project. This assumes that you already had this file; either by acquiring from someone or editing it in a different editor.

I will choose the last option. One important thing to remember is that you should never rename/copy/delete the files from Windows. You should perform any one of these operations *inside Visual Studio 2015*. Otherwise, you will confuse VS 2015 and it will try to use a file that doesn't exist. Because I intend to use the last option, the best thing to do is to actually plop the file imflipG.cu inside the Z:\code\imflipG\imflipG directory first. The screen shot after doing this is shown at the bottom of Figure 6.6. This is, for example, what you would

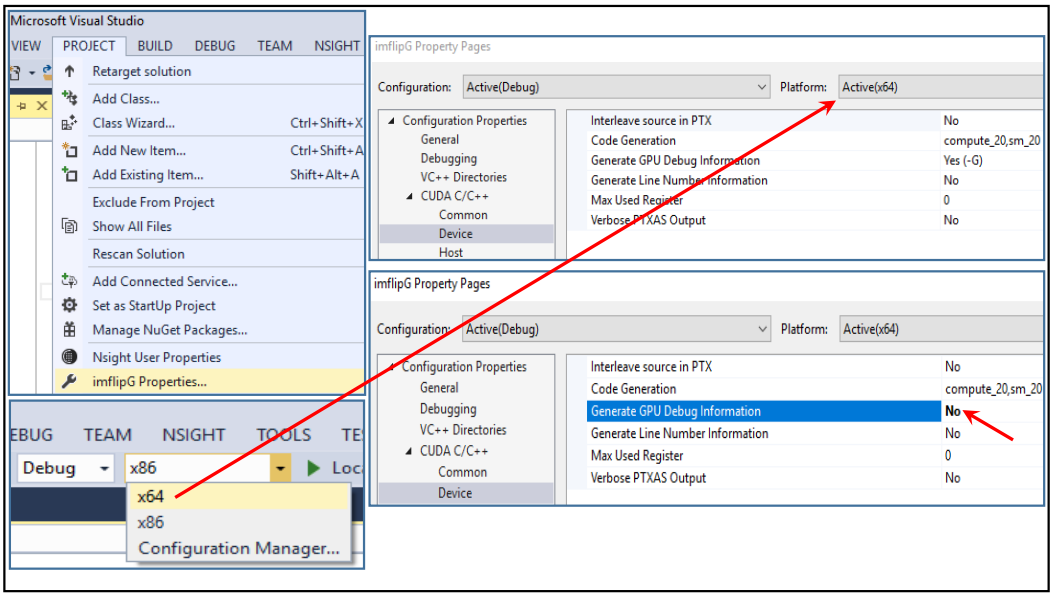


FIGURE 6.7 The default CPU platform is x86. We will change it to x64. We will also remove the GPU debugging option.

do if you are testing the programs I am supplying as part of this book. Although you will get only a single file, imflipG.cu, as part of this book, it must be properly added to a VS 2015 project, so you can compile it and execute it. Once the compilation is done, there will be a lot of miscellaneous files in the project directory, however, the source file is only a single file: imflipG.cu.

Figure 6.6 also shows the steps in deleting the kernel.cu() file. You right click and choose "Remove" first (top left). A dialog box will appear asking you whether you want to just remove it from the project, but keep the actual file (the "Remove" option) or remove it from the project and delete the actual file too (the "Delete" option). If you choose the "Delete" option, the file will be gone and it will no longer be a part of the project. This is the *graceful* way to get this file permanently out of your life, while also letting VS 2015 know about it along the way. After kernel.cu() is gone, you right click the project and this time Add a file to it. You can either add the file that we just dropped into the source directory (which is what we want to do by choosing the "Add Existing Item" option), or add a new file that doesn't exist and you will start editing (the "Add New Item" option). After we choose "Add Existing," we see the new imflipG.cu file added to the project in Figure 6.6. We are now ready to compile it and run it.

6.5.3 Compiling Project imflipG.cu in Visual Studio 2015

Before you can compile your code, you have to make sure that you choose the correct CPU and GPU platforms. As shown in Figure 6.7, the two CPU platform options are x86 (for 32-bit Windows OSs) and x64 (for 64-bit Windows OSs). I am using Windows 10 Pro, which is an x64 OS. So, I drop-down the CPU platform option and choose x64. For the GPU platform, you have to go to the project's properties by choosing PROJECT in the menu bar and selecting *imflipG Properties*. Another "imflipG Property Pages" dialog box

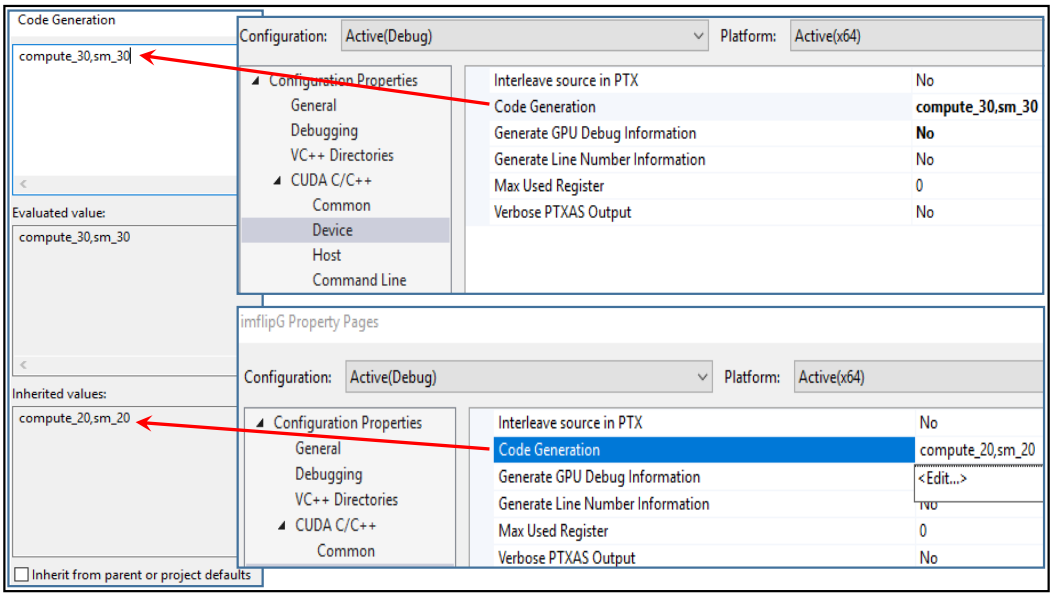


FIGURE 6.8 The default Compute Capability is 2.0. This is too old. We will change it to Compute Capability 3.0, which is done by editing *Code Generation* under *Device* and changing it to compute_30, sm_30.

will open, as shown in Figure 6.7. For the GPU, the first option you choose is *Generate GPU Debug Information*. If you choose "Yes" here, you will be able to run the GPU debugger, however your code will run at half the speed because the compiler has to add all sorts of break points inside your code. Typically, the best thing to do is to keep this at "Yes" while you are developing your code. After your code is fully debugged, you switch it to "No" as shown in Figure 6.7.

After you choose the GPU Debug option, you have to edit the *Code Generation* under *CUDA C/C++* → *Device* and select the *Code Generation* next, as shown in Figure 6.8. The default Compute Capability is 2.0, which will not allow you to run a lot of the new features of the modern Nvidia GPUs. You have to change this to Compute Capability 3.0. Once the "Code Generation" dialog box opens, you have to first uncheck *Inherit from parent of project defaults*. The default Compute Capability is 2.0, which the "compute_20, sm_20" string represents; you have to change it to "compute_30, sm_30" by typing this new string into the textbox at the top of the Code Generation dialog box, as shown in Figure 6.8. Click "OK" and the compiler knows now to generate code that will work for Compute Capability 3.0 and above. When you do this, your compiled code will no longer work with any GPU that *only supports 2.0 and below*. There have been major changes starting with Compute Capability 3.0, so it is better to compile for at least 3.0. Compute Capability of the Nvidia GPUs is exactly like the x86 versus x64 Intel ISA, except there are quite a few more options from Compute Capability 1.0 all the way up to 6.x (for the Pascal Family) and 7.x for the upcoming Volta family.

The best option to choose when you are compiling your code is to set your Compute Capability to the lowest that will allow you to run your code at an acceptable speed. If you set it too high, like 6.0, then your code will only run on Pascal GPUs, however you will have the advantage of using some of the high-performance instructions that are only available

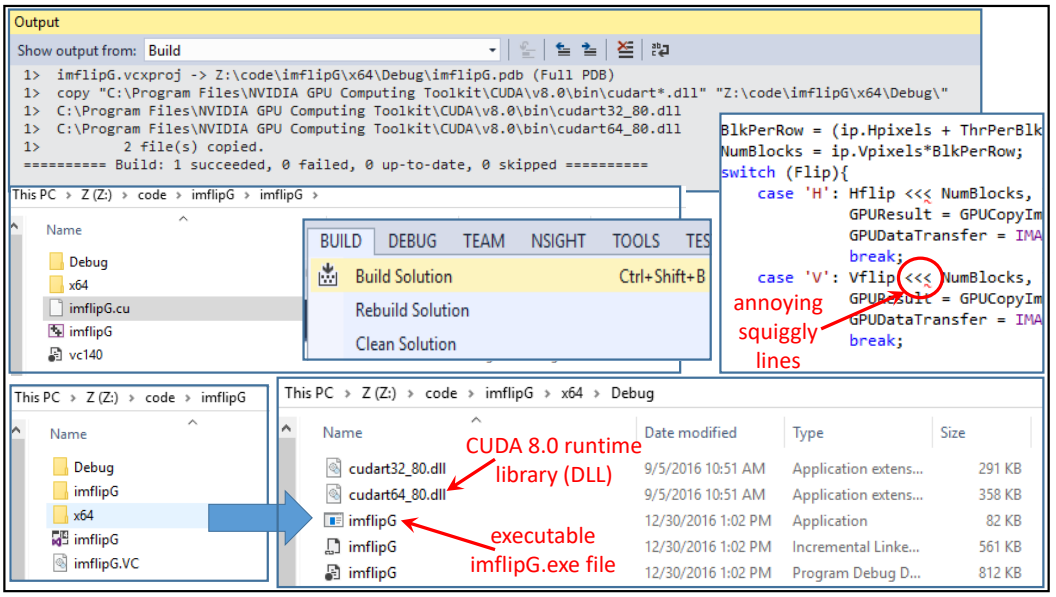


FIGURE 6.9 Compiling imflipG.cu to get the executable file imflipG.exe in the Z:\code\imflipG\x64\Debug directory.

in Pascal GPUs. Alternatively, if you use a low number, like 2.0, then your code might be exposed to the severe limitations of the early year-2000 days, when — just as a quick example — the block size limitations were so restrictive that you had to launch the kernels in a loop because each kernel launch could only have a maximum of ≈60,000 blocks, rather than the multibillion, starting with 3.0. This would be a huge problem even in our very first CUDA program imflipG.cu; as we analyzed in Section 6.4.15, imflipG.cu required us to launch 166,656 blocks. Using Compute Capability 2.0 would require that we somehow chop up our code into three separate kernel launches, which would make the code messy. However, using Compute Capability 3.0 and above, we no longer have to worry about this because we can launch billions of blocks with each kernel. We will study this in great detail in the next chapter. This is why the 3.0 is a good default for your projects and I will choose 3.0 as my default assumption for all of the code I am presenting in this book, unless otherwise stated explicitly. If 3.0 is continuously what you will use, it might be better to change the *Project defaults*, rather than having to change this every time you create a new CUDA program template.

Once you choose the Compute Capability, you can compile and run your code; go to *BUILD* → *Build Solution* as shown in Figure 6.9. If there are no problems, your screen will look like what I am showing in Figure 6.9 (1 succeeded and 0 failed) and your executable file will be in the Z:\code\imflipG\x64\Debug directory. If there are errors, you can click it to go to the source line of the error.

Although Visual Studio 2015 is a very nice IDE, it has a super annoying feature when it comes to developing CUDA code. As you see in Figure 6.9 (see ebook for color version), your kernel launch lines in main() — consisting of CUDA's signature ⋘ and ⋙ brackets — will have squiggly red lines as if they are a syntax error. It gets worse; because VS 2015 sees them as dangerous aliens trying to invade this planet, any chance it gets, it will try to separate them into double and single brackets: "⋘" will become "≪<". It will drive you

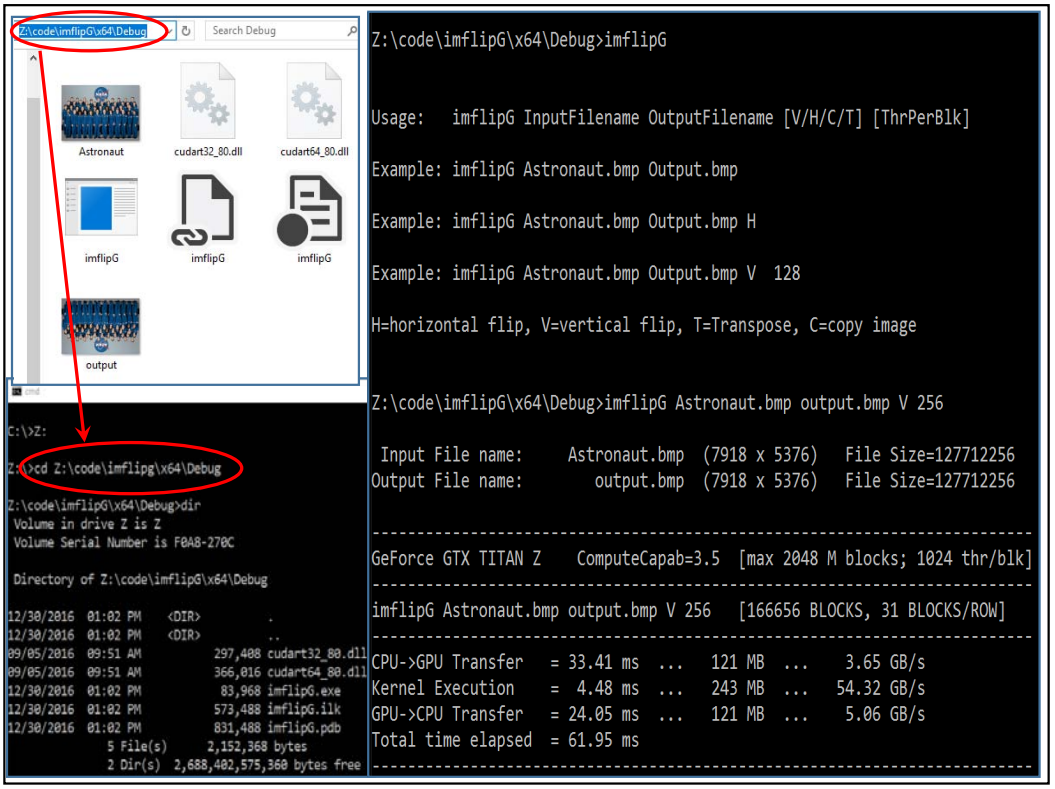


FIGURE 6.10 Running imflipG.exe from a CMD command line window.

nuts when it separates them and you connect them back together and, in a minute, they are separated again. Don't worry. You will figure out how to handle them in time and you will get over it. I no longer have any issues with it. Ironically, even after being separated, nvcc will actually compile it correctly. So, the squiggly lines are nothing more than a nuisance.

6.5.4 Running Our First CUDA Application: imflipG.exe

After a successful compilation, your executable file imflipG.exe will be in the Z:\code\imflipG\x64\Debug directory, as shown in Figure 6.10.

The best way to run this file is to open a CMD (command line interpreter) in Windows and type the following commands to run the application:

C:\> Z:

Z:\> CD Z:\code\imflipG\x64\Debug

Z:\code\imflipG\x64\Debug> imflipG Astronaut.bmp Output.bmp V 256

As seen in Figure 6.10, if you have File Explorer open, you can browse this executable code directory and when you click the location dropdown box, the directory name will be highlighted (Z:\code\imflipG\x64\Debug), allowing you to copy it using Ctrl-C. You can then type "CD" inside your CMD window and paste that directory name after "CD", which eliminates the need to remember that long directory name. The program will require the source file Astronaut.bmp that we are specifying in the command line. If you try to run it

without the Astronaut.bmp in the code executable directory, you will get an error message, otherwise the program will run and place the expected resulting output file, Output.bmp in this case, in the same directory. To visually inspect this file, all you have to do is to open a browse — such as Internet Explorer or Mozilla — and drop the file into the browser window. Even simpler, you can double click on the image and Windows will open the associated application to view it. If you want to change that default application, Windows will typically give you an option to do so.

6.5.5 Ensuring Your Program's Correctness

In Figure 6.10, we see that the Output.bmp file is the vertically flipped version of Astronaut.bmp, however, there is no guarantee that it is exactly what it is supposed to be, pixel by pixel. This is why it is a good idea to have an output file that you know is perfect such as one that you wrote — and 100% confirmed — using the CPU version of the program. You can then use some file comparison tools to see if they are exactly the same. Such a file is named *golden truth* or *ground truth*. However, from my experience, your chances of having a correctly functioning program are pretty high if you go through a list of *common sense* checks:

➢ *Use these as the "minimum common-sense check" rules:*

➢ *Your program is highly likely to be functioning correctly, if*

i) You didn't get an error, causing the program to terminate,
ii) The program didn't take an unusual amount of time to complete,
iii) Your computer didn't start acting weird or sluggish after the program finished running and everything appeared to be running fine,
iv) A file exists in the directory with the expected name Output.bmp*,*
v) The file size of Output.bmp *is identical to what you expect,*
vi) A visual inspection of Output.bmp *shows no signs of problems.*

If everything checks out OK in this list after the execution of the program is complete, then your program may be fine. After these checks the only remaining issues are subtle ones. These issues do not manifest themselves as errors or crashes; they may have subtle effects that are hard to tell through the checklist above, such as the image being one pixel shifted to the right and the one row on the left being blank (e.g., white). You wouldn't be able to tell this problem with the simple visual check, not even when you drag and drop the file into a browser. The one horizontal column of blank pixels would be white, much like the background color of the browser, thereby making the two difficult for you to distinguish between the browser background versus image column. However, a trained eye knows to be suspicious of everything and can spot the most subtle differences like this. In any event, a simple file checker will clear up any doubt that you have in mind for these kinds of problems.

Just as computer programmer trivia, I can't stop myself from mentioning a third kind of a problem: everything checks out fine, and the golden and output files compare fine. However, the program gradually makes computer performance degrade. So, in a sense, although your program is producing the expected output, it is *not running properly*. This is the kind of problem that will really challenge an intermediate programmer, even an experienced one. But, more than likely, an experienced programmer will not have these types of bugs in his or her code; yeah right! Examples of these bugs include ones that allocate memory and do not free it or ones that write a file with the wrong attributes, preventing another program from modifying it, assuming that the intent of the program

is to produce an output that can be further modified by another program, etc. If you are a beginner, you will develop your own experience database as time goes on and will be proficient in spotting these bugs. I can make a suggestion for you though: be suspicious of everything! You should be able to detect any anomalies in performance, output speed, the difference between two different runs of the same code, and more. When it comes to computer software bugs — and, for that matter even hardware design bugs — it might be a good time to repeat Intel's former CEO and legend, late Andy Grove's words: *only the paranoid survive*.

6.6 CUDA PROGRAM DEVELOPMENT ON A MAC PLATFORM

Because a Mac OS has almost an identical structure to Unix, the instructions for Mac and general Unix will be extremely similar. Mac owners are encouraged to read everything in the Unix section too after reading this section.

6.6.1 Installing XCode on Your Mac

CUDA Toolkit, regardless of its version, requires a command line tool — such as gcc — to work. To be able to get gcc into your Mac, you have to install Xcode.

Installation instructions for Xcode are as follows:

- You must have an Apple developer account. Create one if you don't. It is free.
- In your Safari browser, go to:

  https://developer.apple.com/xcode/

- Click "Download" and download the Xcode IDE. It includes an environment for building any Mac, iPhone, iPad app, even Apple Watch or Apple TV apps. Xcode 8 is the current version at the time of the writing of this book.
- When Xcode is initially installed, a command line tool will not be a part of it.
- Go to Xcode → Preferences → Downloads → Components
- Select and install the *Command Line Tools* package. This will bring gcc into your Apple; you can now launch gcc from your terminals.
- Instead of this GUI method, you have another option to install gcc directly from your terminal with the following commands:

  ```
  xcode-select –install
  ```

  ```
  /usr/bin/cc/help
  ```

- The last line is to confirm that the command line tool chain is installed.
- The biggest difference between a Windows and a Mac — or in general, all of Unix — environments is that Windows has a strictly IDE-dictated structure for storing executable and source files, whereas Unix platforms do not create a total mess on your hard drive. One example of this mess is the database file that MS Visual Studio 2015 creates, which takes up 20 MB, etc. So, for a 20 KB imflipG.cu source file, your Mac project directory could be 100 KB including all of the executables, whereas the Windows project directory could be 20 MB!

■ GPU Parallel Program Development Using CUDA

6.6.2 Installing the CUDA Driver and CUDA Toolkit

Once you have gcc installed, you will need to install the CUDA driver and the CUDA toolkit [18], which will bring the nvcc compiler into your Mac; with this compiler, you will be able to compile the imflipG.cu file and the executable will be in the same directory. To install the CUDA driver and the CUDA toolkit, follow these instructions:

- You must have an Intel-CPU based Mac, and a supported Mac OS version (Mac OS X 10.8 or higher), as well as a CUDA-supported Nvidia GPU.
- You must have an Nvidia developer account. Create one if you don't. It is free.
- The CUDA toolkit will install a driver for you unless you have installed the standalone driver before the CUDA toolkit.
- After the CUDA toolkit installation is complete, you will have all of your CUDA stuff in /usr/local/cuda and all of your Apple Developer account in the /Developer/NVIDIA/CUDA-8.0 directory. These names might change slightly depending on the versions you are installing. There might actually be multiple CUDA directories with different versions. If you choose to, the CUDA toolkit will install all sorts of useful samples for you in the /Developer/NVIDIA/CUDA-8.0/samples directory. Once you develop a sufficient understanding of the operation/programming of the GPUs, you can look at the samples to get advanced CUDA programming ideas.
- In order to be able to run the nvcc compiler along with many other tools, set up your environment variables:

  ```
  export PATH=/Developer/NVIDIA/CUDA-8.0/bin:$PATH

  export DYLD_LIBRARY_PATH=/Dev...8.0/lib:$DYLD_LIBRARY_PATH
  ```

- If you have a Mac Book Pro that has an Nvidia, as well as an Intel-CPU-integrated GPU, the laptop will try to use the Intel GPU as much as it can to conserve energy. The Nvidia GPU is termed a *discrete GPU* — and, it indeed is a separate *card* that plugs into some slot inside your laptop, allowing you to change it in the future — and the Intel GPU is termed the *integrated GPU*, which is built-into your CPU's VLSI Integrated Circuit and cannot be upgraded unless you upgrade the CPU itself. Nvidia's Optimus technology [16] allows your laptop to switch between the integrated and discrete GPUs, however, you have to tell OS to do so by following the steps below (instructions are taken from [18], which is an online document for Getting Stared with CUDA on your Mac OS X):

  Uncheck System Preferences → Energy Saver → Automatic Graphic Switch

  Choose *Never* in the Computer Sleep bar.

6.6.3 Compiling and Running CUDA Applications on a Mac

Once you have Xcode installed, you can either let Xcode compile it for you — much like what we saw in the case of Visual Studio — or go to a terminal and type the following command line to compile it. Once this compilation succeeds, you can type the name of the executable to run it:

```
nvcc -o imflipG imflipG.cu

imflipG
```

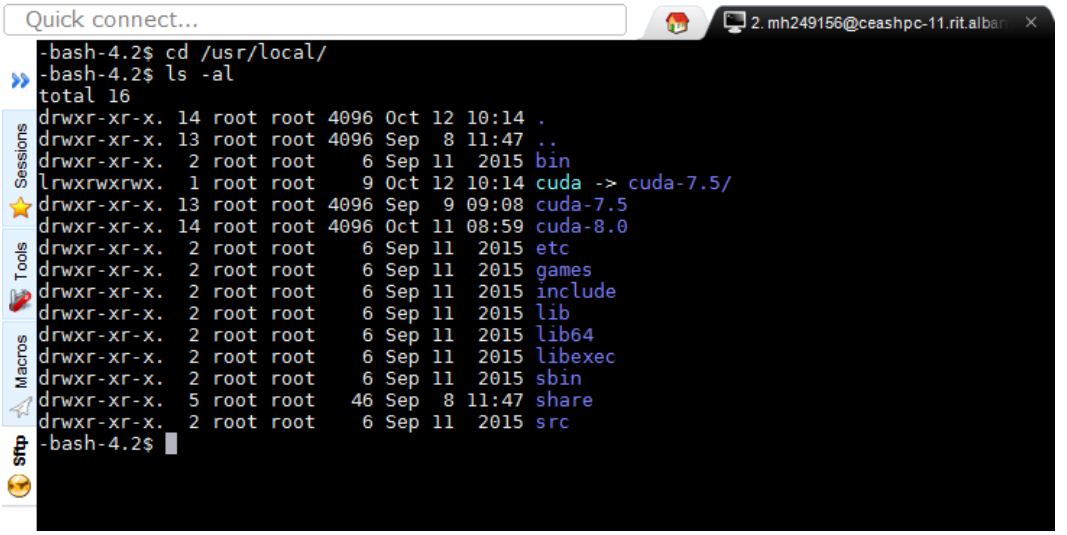


FIGURE 6.11 The /usr/local directory in Unix contains your CUDA directories.

Actually, in a Windows platform, this is precisely what Visual Studio does when you click the "Build" option. You can view and edit the command line options that VS 2015 will use when compiling your CUDA code by going to *PROJECT* → *imflipG Properties* on the menu bar. Xcode IDE is no different. Indeed, the Eclipse IDE that I will describe when showing the Unix CUDA development environment is identical. Every IDE will have an area to specify the command line arguments to the underlying nvcc compiler. In Xcode, Eclipse, or VS 2015, you can completely skip the IDE and compile your code using the command line terminal. The CMD tool of Windows also works for that.

6.7 CUDA PROGRAM DEVELOPMENT IN A UNIX PLATFORM

In this section, I will give you the guidelines for editing, compiling, and running your CUDA programs in a Unix environment. In Windows, we used the VS 2015 IDE. On a Mac, we used the Xcode IDE. In Unix, the best IDE to use is Eclipse. So, this is what we will do.

6.7.1 Installing Eclipse and CUDA Toolkit

The first step — much like Windows and Mac — is to install the Eclipse IDE, followed by the CUDA 8.0 toolkit. Before you can compile and test your CUDA code, you have to add the environment variables into your path, just like we saw in Mac. Edit your .bashrc by using an editor like gedit, vim (that's real old school), or emacs. Add this line into your .bashrc:

```
$ export PATH=$PATH:/usr/local/cuda-8.0/bin/
```

If you do not want to close and open your terminal again, just source your .bashrc by typing

```
$ source .bashrc
```

This will make sure that the new path made it into your PATH environment variable. If you browse your /usr/local directory, you should see a screen like the one in Figure 6.11.

Here, there are two different CUDA directories shown. This is because CUDA 7.5 was installed first, followed by CUDA 8.0. So, both of the directories for 7.5 and 8.0 are there. The /usr/local/cuda symbolic link points to the one that we are currently using. This is why it might be a better idea to actually put this symbolic link in your PATH variable, instead of a specific one like cuda-8.0, which I showed above.

6.7.2 ssh into a Cluster

You can either develop your GPU code on your own Unix-based computer or laptop, or, as a second option, you can login to a GPU cluster using an X terminal program. You cannot login by using a simple text-based terminal. You have to either have an xterm, or another program that allows you to do "X11 Forwarding." This forwarding means that you can run graphics applications remotely and display the results on your local machine. A good program is MobaXterm, although if you installed Cygwin to run the code in Part I, you might have also installed Cygwin-X, which will have an xterm for you to use as an X Windows-based terminal and run CUDA programs and display the graphics output locally.

To run a program remotely and display the result locally, follow these instructions:

- ssh into the remote machine with the X11 forwarding flag, "-X".

```
$ ssh -X username@clustername
```

- Potentially, the best thing to do is to open a second terminal strictly for file transfers.
- On the second terminal, transfer the source BMP files into the cluster

```
$ sftp username@clustername
```

- Use put and get commands, found in sftp, to transfer files back and forth. Alternatively, you can use scp for secure copy.
- Compile your code on the first terminal and make sure that the resulting output file is there.
- Transfer the output file back to your local machine.
- Alternatively, you can display the file remotely and let X11 forwarding show it on your local machine, without having to worry about transferring the file back.

6.7.3 Compiling and Executing Your CUDA Code

Instead of running nvcc on a command line, you can use the Eclipse IDE to develop and compile your code, much like the case in Visual Studio 2015. Assuming that you configured your environment correctly by following the instructions in Section 6.7.1, type this to run the Eclipse IDE:

```
$ nsight &
```

A dialogue box opens asking you for the workspace location. Use the default or set it to your preferred location and press OK. You can create a new CUDA project by choosing *File* → *New* → *CUDA C/C++ Project*, as shown in Figure 6.12.

Build your code by clicking the hammer icon and run it. To execute a compiled program on your local machine, run it as you would any other program. However, because we are

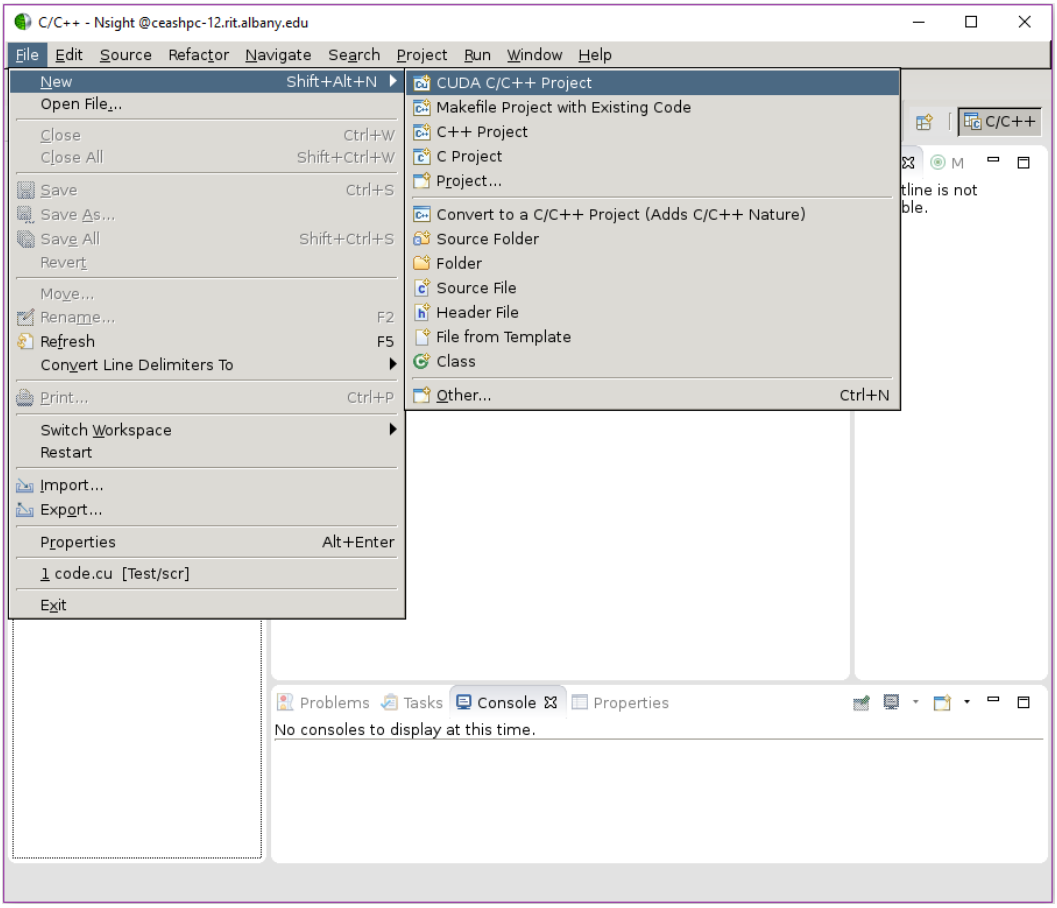


FIGURE 6.12 Creating a new CUDA project using the Eclipse IDE in Unix.

normally going to be passing files and command line arguments to the program, you will probably want to put the cd into the directory with the binary and run it from there. You could specify the command line arguments from within your IDE, but this is somewhat tedious if you are changing them frequently. The binaries generated by IDEs generally appear in some subfolder of your project (Eclipse puts them in the Debug and Release folders). As an example, to run the "Release" version of an application in Linux that we developed in Eclipse/Nsight, we may type the following commands:

```
cd ~/cuda-workspace/imflipG/Release
./imflipG
```

This will run your CUDA code and will display the results exactly like in Windows.

CHAPTER 6

Optimization techniques and best practices for parallel codes

CONTENTS

- Optimization techniques and best practices for parallel codes

6.1 DATA PREFETCHING, COMMUNICATION AND COMPUTATIONS OVERLAPPING AND INCREASING COMPUTATION EFFICIENCY

Efficient parallelization of application execution should lead to minimization of application execution time. This generally requires the following:

1. Engaging all compute units for computations that can be achieved through load balancing in order to avoid idle times on the compute units.
2. Minimization of communication and synchronization that result from parallelization.

In some instances, engaging a compute unit may result in unwanted idle times, specifically (shown in Figure 3.6a):

1. Since computation of a data packet requires prior sending or provision of the data packet, idle time may occur before computations.
2. Similarly, after a data packet has been computed, then a process or a thread will usually request another data packet for processing.

Such idle times may show up at various levels in a parallel system, including:

1. Communication between nodes in a cluster when a process requests a data chunk from another process, especially running on a different node.
2. Communication between a host and a GPU within a single node since a thread running on a GPU will need input data initially stored in the host RAM.
3. Fetching input data from global memory on a GPU. Specifically, using shared memory and registers within a GPU is much faster than fetching data from global memory. As a consequence, fetching input data from global memory may become a bottleneck.

A solution to this problem is to prefetch data before it is actually processed. Specifically, at the same time when a certain data packet is processed, another can be fetched in the background such that it is already available when processing of the former data packet has finished. This universal approach can be implemented using various APIs. Specific pseudocodes with proper API calls are provided next.

Data prefetching, overlapping communication and computations ... ■

6.1.1 MPI

There are at least two programming approaches in MPI that allow implementation of overlapping communication and computations and data prefetching. Listing 6.1 presents an approach with non-blocking API calls described in detail in Section 4.1.11. The solution uses `MPI_I*` calls for starting fetching data. Since these are non-blocking calls, a calling process can perform computations immediately after the call. The latter only issues a request for starting communication. After computations, i.e. processing of a data packet, have completed, non-blocking communication needs to be finalized using `MPI_Wait` and processing of the just received data packet can follow. If there are to be more data packets processed then a new data packet can be fetched before computations start.

Listing 6.1 Receiving data using MPI and overlapping communication and computations with non-blocking calls

```
// the first data packet can be received using MPI_Recv

MPI_Recv(inputbuffer,...);
packet=unpack(inputbuffer);

while (shallprocess(packet)) {
  // first start receiving a data packet
  MPI_Irecv(inputbuffer,...,&mpirequest);

  // process the already available data packet
  process(&packet);

  // now finish waiting for the next data packet
  MPI_Wait(&mpirequest);

  // unpack data so that the buffer can be reused
  packet=unpack(inputbuffer);

}
...
```

Alternatively, the code without unpacking of data from a buffer and using two buffers instead is shown in Listing 6.2.

Listing 6.2 Receiving data using MPI and overlapping communication and computations with non-blocking calls and using two buffers

- Optimization techniques and best practices for parallel codes

```
// the first data packet can be received using MPI_Recv
buffer=inputbuffer0;
MPI_Recv(buffer,...);

while (shallprocess(buffer)) {

  if (buffer==inputbuffer1) {
    buffer=inputbuffer0;
    prevbuffer=inputbuffer1;
  } else {
    buffer=inputbuffer1;
    prevbuffer=inputbuffer0;
  }

  // first start receiving a data packet
  MPI_Irecv(buffer,...,&mpirequest);

  // process the already available data packet
  process(prevbuffer);

  // now finish waiting for the next data packet
  MPI_Wait(&mpirequest);

}
...
```

In fact, a slave process would normally send back its results to the parent process. Overlapping sends and processing of subsequent data, or packets can also be arranged. Such a solution is shown in Listing 6.3.

Listing 6.3 Receiving data and sending results using MPI and overlapping communication and computations with non-blocking calls and using two buffers

```
MPI_Request requests[2]; // two requests:
// requests[0] used for receive
// requests[1] used for send

requests[1]=MPI_REQUEST_NULL; // do not consider
// the request for send in the first iteration

// the first data packet can be received using MPI_Recv
buffer=inputbuffer0;
MPI_Recv(buffer,...);
```

Data prefetching, overlapping communication and computations ... ■

```
while (shallprocess(buffer)) {

  if (buffer==inputbuffer1) {
    buffer=inputbuffer0;
    prevbuffer=inputbuffer1;
    prevresultbuffer=outputbuffer1;
  } else {
    buffer=inputbuffer1;
    prevbuffer=inputbuffer0;
    prevresultbuffer=outputbuffer0;
  }

  // first start receiving a data packet
  MPI_Irecv(buffer,...,&(requests[0]));

  // process the already available data packet
  process(prevbuffer,prevresultbuffer);

  // now finish waiting for the next data packet
  MPI_Waitall(2,requests,statuses); // note that in the first
    iteration
  // only requests[0] will be active

  // now start sending back the result
  MPI_Isend(prevresultbuffer,...,&(requests[1]));

}

...
```

Another programming approach in MPI can involve two threads in a process, each for performing a distinct task. This, however, requires an MPI implementation that supports multithreading in the required mode (see Section 4.1.14). Then the approach could be as follows – threads would perform their own tasks:

1. Communication i.e. fetching input data and possibly sending results; there can also be a separate thread for sending out results.
2. Processing the already received data packet(s).

This approach is natural and straightforward but requires proper synchronization among the threads. Such synchronization can be implemented e.g. using Pthreads [129, Chapter 4]. The most natural way of implementing such a solution using three threads (receiving, processing and sending) would be to use two queues for incoming data packets and outgoing packets with results, as shown in Section 5.1.4.

- Optimization techniques and best practices for parallel codes

6.1.2 CUDA

Implementation of overlapping communication between the host and a GPU and processing on the GPU can be done using streams described in Section 4.4.5. Specifically, using two streams potentially allows overlapping communication between page-locked host memory and a device (in one stream), computations on the device (launched in another stream) as well as processing on the host:

```
cudaStreamCreate(&streamS1);
cudaStreamCreate(&streamS2);

cudaMemcpyAsync(devicebuffer1,sourcebuffer1,copysize1,
cudaMemcpyHostToDevice,streamS1);
kernelA<<<gridsizeA,blocksizeA,0,streamS1>>>(...);
cudaMemcpyAsync(hostresultbuffer1,
deviceresultbuffer1,copyresultsize1,
cudaMemcpyDeviceToHost,streamS1);

cudaMemcpyAsync(devicebuffer2,sourcebuffer2,copysize2,
cudaMemcpyHostToDevice,streamS2);
kernelB<<<gridsizeB,blocksizeB,0,streamS2>>>(...);
cudaMemcpyAsync(hostresultbuffer2,
deviceresultbuffer2,copyresultsize2,
cudaMemcpyDeviceToHost,streamS2);

processdataonCPU();

cudaDeviceSynchronize();
```

As indicated in [137], the issue order of commands may have an impact on execution time. This is due to the fact that host to device, device to host and kernel launch commands are added to respective queues on the device based on the issue order. Commands in a queue in a given stream may need to wait for a sequence of commands issued to another stream if they are preceded in a queue on the device.

Additionally, when using CUDA, the Multi-Process Service (MPS) can be used to increase performance in some scenarios for GPUs with Hyper-Q. Specifically, MPS provides an implementation of the CUDA API that allows multiple processes to use a GPU or many GPUs with better performance than without this mechanism. In case a single process is not able to saturate the whole computing capability of a GPU(s) then many processes can be used with an MPS server as a proxy to the GPU. This allows overlapping computations and data copying from many processes [119] and consequently exploiting a GPU(s) to a higher degree. Additionally, this approach is transparent to the application which is a considerable benefit. MPS requires compute capability

3.5+ and a 64-bit application running under Linux [119]. An MPS daemon can be started as follows:

```
nvidia-cuda-mps-control -d
```

The following experiment tests a scenario without MPS and with MPS for the geometric SPMD application implemented with MPI and CUDA and shown in Section 5.2.4. In every scenario the code was run with various numbers of MPI processes. For every configuration execution time for the best out of three runs is presented in Table 6.1. Application parameters used were `384 384 960 10 2`. Tests were performed on a workstation with 2 x Intel Xeon E5-2620v4, 2 x NVIDIA GTX 1070 GPUs and 128 GB RAM Two GPUs were used.

TABLE 6.1 Execution times [s] for a geometric SPMD MPI+CUDA code, without and with MPS

number of processes	execution time without MPS [s]	execution time with MPS [s]
2	4.668	3.971
4	4.770	3.042
8	7.150	3.339

6.2 DATA GRANULARITY

Data granularity and partitioning may have an an impact on execution time of a parallel application. Specifically, assuming a master-slave application in which input data is divided into data packets which are then distributed among slave processes for computations and then results are gathered, the following would apply:

1. Small data packets would allow good balancing of data among computing nodes/processors. This might be especially useful if there are processors of various computing speed. On the other hand, too many small data packets would result in considerable overhead for communication.

2. Large data packets might result in poor load balancing among computing nodes/processors. Also, in case of really large data packets, computations might start with a delay.

As a result, there is a trade-off and an optimum size of a data packet can be found. This is illustrated in Figure 6.1 for a parallel master-slave numerical integration application for function $f(x) = 10.0/(1.0 + x)$ implemented with MPI and for various numbers of data packets used. Each packet is then partitioned into rectangles of $\frac{1}{40 \cdot 1024}$ width. The test was run on a workstation

- Optimization techniques and best practices for parallel codes

with 2 x Intel Xeon E5-2620v4 and 128 GB RAM. For each configuration, the best out of 3 runs are shown. Testbed results are shown for 4 and 16 processes of an application.

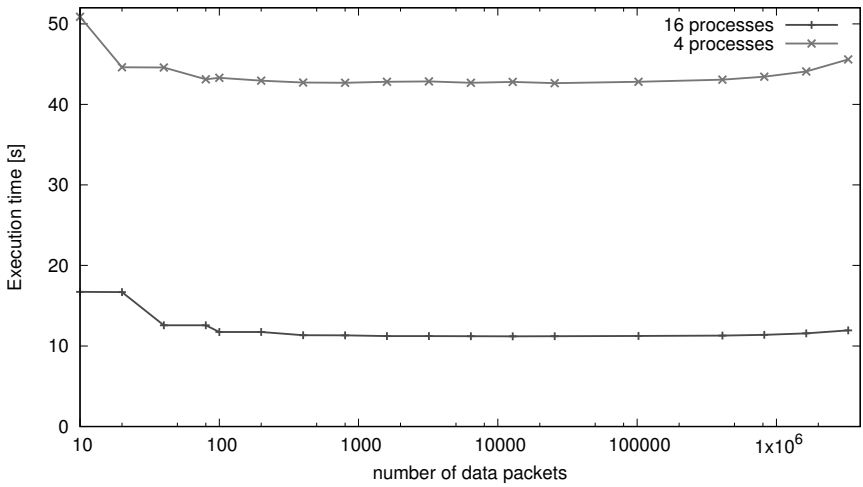


FIGURE 6.1 Execution time of an MPI master-slave application vs number of data packets for fixed input data size

It should be noted that proper distribution of such data among e.g. nodes in a cluster requires efficient load balancing, as discussed in Section 3.1.4.

6.3 MINIMIZATION OF OVERHEADS

6.3.1 Initialization and synchronization overheads

In parallel programming attention should be paid to the costs related to:

1. spawning processes or threads,
2. synchronization.

Specifically, in iterative algorithms such as SPMD implemented with OpenMP, overheads related to thread creation/initialization will be present when entering a `#pragma omp parallel` region. On the other hand, various synchronization constructs discussed in Section 4.2.4 will bring their own delays. In the aforementioned SPMD applications, there are, in particular, two ways of implementation of a loop in which iterations typically correspond to time steps in FDTD simulations [56, 50]. In each loop iteration, the domain is partitioned into subdomains each of which needs to be updated by a separate thread. After updates, synchronization before a following iteration is needed.

Such approaches are also discussed in [23] in the context of OpenMP applications run on an Intel Xeon Phi. In the case of a hybrid MPI+OpenMP application discussed here, a master thread of a process would be involved in communication with other processes. Furthermore, processing of subdomain data assigned to a process must be partitioned into several threads within the process, in this case using OpenMP.

Two functionally identical implementations are as follows:

1. The main loop is executed by the master thread with some parts parallelized using OpenMP constructs. Specifically:
   (a) Exchange of data by the master thread (might be needed first if each process initializes its own domain independently) without any special constructs.
   (b) Parallel update of subdomain cells – parallelization performed using `#pragma omp parallel for`.
   (c) Substitution of pointers for source domain and target domain performed by the master thread without any special constructs.
2. Entering a parallel region outside of the loop. Then each thread would execute loop iterations independently which results in the need for synchronization. Specifically, steps within a loop iteration include:
   (a) Exchange of data by the master thread (might be needed first if each process initializes its own domain independently) in code within `#pragma omp master`.
   (b) Synchronization using `#pragma omp barrier`.
   (c) Parallel update of subdomain cells – parallelization performed using `#pragma omp for`.
   (d) Substitution of pointers for source domain and target domain performed by the master thread in code within `#pragma omp master`.

The two versions of the code can be compiled as follows:

```
mpicc -fopenmp <flags> SPMD-MPI+OpenMP-1.c
mpicc -fopenmp <flags> SPMD-MPI+OpenMP-2.c
```

and executed on a workstation with 2 x Intel Xeon E5-2620v4 and 128 GB RAM with parameters `200 200 200 10000`.

Out of 10 runs for each version, Table 6.2 presents best results for a single process and domain size 200x200x200 and 10000 iterations.

- Optimization techniques and best practices for parallel codes

TABLE 6.2 Execution times [s] for two versions of MPI+OpenMP SPMD code

version	minimum execution time [s]	average execution time out of 10 runs [s]
`#pragma omp parallel for` inside main loop	161.362	163.750
`#pragma omp parallel` outside of the main loop	160.349	163.010

6.3.2 Load balancing vs cost of synchronization

In some cases, it might be possible to reduce the time of synchronization at the cost of ability to balance load. For instance, if threads are to fetch input data packets or pointers to input data packets they are supposed to process, the threads would do so in a critical section. In case there are many threads executing, such as for manycore devices such as Intel Xeon Phi, a global critical section might potentially become a bottleneck. A solution to this problem could be to split threads into several groups with the following assumptions and steps:

1. Input data packets are divided into the number of groups equal to the number of thread groups.
2. Instead of one critical section, the number of critical sections equal to the number of thread groups is used, one per group. Then there are fewer threads per critical section which potentially reduces time a thread needs to wait for fetching a new data packet.

In OpenMP, this can be implemented using named critical sections, with a different name for each critical section. A regular implementation would use a regular unnamed critical section.

In terms of results, for runs performed on an Intel Xeon Phi x100 3000 series coprocessor, for a numerical integration application in which each data packet corresponds to a subrange which is further divided into a number of smaller subranges and areas of corresponding rectangles are added, no differences for a version with one critical section as compared to 4 critical sections were observed for up to 32 threads. Then, for 64 threads, the speed-up was better for the latter version by 0.14 up to 58.74 for 128 threads up by 0.8 up to 88.56 and for 228 threads by 1.2 up to 108.78 [45].

6.4 PROCESS/THREAD AFFINITY

In multicore and manycore systems, from performance point of view it might be important to consider how processes or threads of a parallel application

are mapped to particular CPUs or CPU cores. It is especially important when threads may share data during computations and proper mapping of threads to cores might efficiently use caches.

In the Linux operating system, it is possible to indicate which CPUs/logical processors in a system will be used for an application. Specifically:

```
taskset -c <cpunumberlist> application
```

will start `application` on the given CPUs in the system. `<cpunumberlist>` contains a comma separated list of CPU numbers starting with `0` e.g. `0,2`, `0-3` etc. In Linux file `/proc/cpuinfo` allows to find out physical ids and core ids of particular logical processors.

In some cases, e.g. when OpenMPI is used, binding can be specified via options to `mpirun`, as shown in Section 4.7.1.

Thread affinity might have a considerable impact on performance results when used on a computing device with a high number of computing cores, such as Intel Xeon Phi [23, 75].

6.5 DATA TYPES AND ACCURACY

Optimization of code also includes the decision on what data types should be used for particular variables in a program. There might be specific functions operating on various types. For instance, the following versions are available for a sine function:

```
float sinf(float a)
double sin(double a)
long double sinl(long double a)
```

Resulting code may offer better performance and, in case of smaller data types, open more potential for vectorization e.g. on Intel Xeon Phi.

Similarly, precision for floating point operations can be controlled with various compiler switches as discussed in [34]. As a result, various trade-offs between execution times and accuracy can be obtained.

In certain applications, smaller data types can be used effectively such as 16-bit fixed point instead of 32-bit floating for training deep networks [79].

6.6 DATA ORGANIZATION AND ARRANGEMENT

It should be noted that proper data arrangement can have an impact on application execution time. If data was prefetched to a cache then it can be accessed much faster.

Let us consider the geometric SPMD example discussed in Section 5.2.1. Specifically, the loops for updates of cells within a subdomain are arranged by indices `z`, `y` and `x`, in that direction. It can be seen from function `getcell`

that data cells that are fetched are located next to each other in memory. Let us call this configuration A.

If, for the sake of a test, the order of loops is reversed i.e. indices are browsed in the `x`, `y` and `z` dimension this is no longer the case. Let us call this configuration B.

Table 6.3 presents comparison of execution times of the two configurations for selected numbers of processes of an MPI application, run on a workstation with 2 x Intel Xeon E5-2620v4 and 128 GB RAM. For each configuration, the best out of 3 runs are shown. The codes can be compiled as follows:

```
mpicc <flags> SPMD-MPI-1.c
mpicc <flags> SPMD-MPI-1-xyz-loop.c
```

Tests were run using 16 processes as follows, for a domain of size 600x600x600:

```
mpirun -np 16 ./a.out 600 600 600 100
```

TABLE 6.3 Execution time [s] depending on data layout – 100 iterations of the SPMD code

domain size X×Y×Z	data layout A	data layout B
600×600×600	23.81	95.081
800×800×800	55.236	236.032
1000×1000×1000	106.808	460.754

Often, tiling or blocking data [110] allows reusing of data that has been loaded into cache or registers. For instance, if a loop stores data in a large array (that does not fit into cache) such that in each iteration a successive element is stored and then this data is used for subsequent updates of other data, such a large loop can be tiled into a few passes each of which reuses data from the cache.

6.7 CHECKPOINTING

Checkpointing is a mechanism that allows saving of the state of an application and resume processing at a later time. Saving the state of a parallel application is not easy because it requires saving a global consistent state of many processes and/or many threads, possibly running on various computing devices, either within a node or on a cluster, with consideration of communication and synchronization.

Checkpointing might be useful for maintenance of the hardware. In some cases, it may also allow moving the state of an application or individual application processes to other locations. This effectively implements migration. The latter might be useful for minimization of application execution times

when processes perform computations for which execution times cannot be predicted in advance such as in alpha beta search.

In general, checkpointing can be implemented at various levels [167], in particular provided by:

1. Hardware.
2. Operating system kernel.
3. User-level [49] in which an existing library is linked with an executable to provide checkpointing.
4. Application level checkpointing [167] which requires a programmer's effort but can result in higher performance as only data really necessary to carry on execution need to be stored in an application state [49].

Many works concerning checkpointing on modern high performance computing systems have appeared, suitable for applications using particular, previously discussed parallel programming APIs:

- CPPC (ComPiler for Portable Checkpointing) [140] allows checkpointing of parallel message passing applications with independence from the operating system and the message passing protocol. Code transformation is done transparently by a provided compiler. The solution works at the level which requires only saving variables necessary for restart. The compiler inserts checkpoints at identified safe points.
- Paper [107] presents extension of CPPC for checkpointing of hybrid MPI+OpenMP applications with a protocol that applies coordinated checkpointing among threads of a team and proper analysis of communication.
- CheCUDA [154] allows checkpointing of CUDA applications using an approach with transferring data from device to host memory, checkpointing, reinitializing and transferring data back to the device. Berkeley Lab Checkpoint/Restart (BLCR) [81] is used for checkpoint and restart on the host side.
- NVCR [117] – a solution for checkpointing CUDA programs that is more transparent than CheCUDA as the former replaces CUDA libraries with proper wrappers. NVCR allows storing of data associated with CUDA and restoring it after restart. It can work with MPI+CUDA applications.
- CheCL [153] – a solution for checkpointing OpenCL applications that replaces an original OpenCL library which allows intercepting original calls. CheCL uses an API proxy process to which original calls are forwarded and which makes actual OpenCL calls. This allows storing needed data in the original process and checkpointing it. BLCR is used. The approach can be used with MPI applications. It was shown that the approach allows migration of a process from one node to another.

- Optimization techniques and best practices for parallel codes

– Application level checkpointing for OpenMP applications [26]. A programmer is required to denote checkpoint points in the code. The C^3 precompiler is used to for code instrumentation and inserting code for checkpointing. According to the authors this solution can be combined with their approach to checkpointing MPI applications [25]. The latter is also based on a instrumenting the source code with the precompiler for subsequent checkpointing at the application level. A coordination layer is used to intercept MPI invocations from the application and implement non-blocking, coordinated protocol to achieve global checkpointing.

– Hybrid Kernel Checkpoint that can save and restore a GPU kernel state [151].

– Checkpointing using NVRAM for MPI applications – paper [62] proposes how to use new technologies such as NVRAMs located in cluster nodes in order to optimize application execution with checkpointing.

6.8 SIMULATION OF PARALLEL APPLICATION EXECUTION

As the sizes of modern HPC systems have increased considerably and there is a great variety of configurations that can be built with many components such as CPUs, GPUs, coprocessors and network interconnects, it is not straightforward to choose the best hardware platform for a given application or a set of applications. Furthermore, in some cases it would be beneficial to test an application for a larger size (such as in terms of the number of nodes, CPUs, cores) compared to an available configuration or potential configurations considered for purchase – such as in case of a considered upgrade. In such scenarios, it is possible to use one of the simulation environments that allow simulation of the execution of an application (can be modeled with a dedicated language or derived from existing applications) on a system that consists of a number of computing devices connected with a particular network. In general, there can be several use cases in which such an environment can be helpful, in particular [52]:

1. Prediction of application performance on a larger system e.g. to assess potential benefits.

2. Identification of bottlenecks and tuning of an application by testing various configurations including sizes of buffers, algorithm improvements etc.

3. Assessment of the best system to run the actual application.

Examples of such systems and use cases applicable in the context of this book include:

- MERPSYS [52] – for modelling and simulation of execution time, energy consumption and reliability of parallel applications run on cluster and volunteer based systems.
- MARS [58] – for performance prediction and tuning of MPI applications running on systems with various network topologies.
- SST/macro [8] – for performance analysis of parallel applications run on a parallel system with computing and interconnect components.

6.9 BEST PRACTICES AND TYPICAL OPTIMIZATIONS

6.9.1 GPUs/CUDA

Typically, several techniques can be used to ensure that high performance of computations can be achieved on a GPU, some of which include:

1. Dynamic scheduling of warps on available multiprocessors of a GPU – handled by the runtime layer.
2. Minimization of thread divergence – making sure that possibly all threads in a warp take the same execution path.
3. Caching that can be done through using shared memory – data is copied from global to shared memory first, then computations are applied and results copied back to global memory and RAM.
4. Prefetching data/hiding communication latency – this technique tries to avoid idle times between the moment previous computations completed and following computations for which new input data needs to be copied. In essence, new input data is prefetched while previous computations are still running. In the context of GPU processing, this technique applies to the following:
   (a) Fetching input data from global memory which is reasonably slow compared to shared memory and registers. If an algorithm works on data chunks in a loop, data can be first loaded from global memory to registers, and then in a loop the following actions can be performed: a data chunk is loaded from registers to shared memory, synchronization among threads within a block is performed, a new data chunk is fetched from global memory to registers, computations are performed on the current data chunk, synchronization among threads within a block is performed and a new loop iteration is executed. This approach allows overlapping computations on a current data chunk with prefetching of a new one.

- Optimization techniques and best practices for parallel codes

(b) communication between the host and the GPU(s). If multiple streams are used and capable GPUs are used, it is possible to overlap communication between the host and the device (and possibly between the device and the host) with kernel execution on the device. This is discussed in Section 6.1.2.

5. Memory coalescing – from the performance point of view, it is best if threads executing in parallel read/write data from successive memory location of the global memory; in some cases proper reorganization of data and/or code can consequently result in shorter execution time.

6. Accessing various shared memory banks from warp threads. Shared memory is divided into banks, each of which can be accessed in parallel. In general, if requests from threads are directed to one bank, accesses will be serialized which affects performance. An n-way bank conflict occurs if n threads access the same bank at the same time. Documentation [122] describes how shared memory organization for particular compute capabilities which will have an impact on performance. For instance, for compute capability 5.x devices, there are 32 banks with mapping of successive 32-bit words to various banks. If two threads of a warp access data within one word, no conflict occurs either for read or write operations [122].

7. Loop unrolling.

8. Minimization of thread synchronization in the algorithm.

6.9.2 Intel Xeon Phi

The architecture of the Intel Xeon Phi manycore solution requires from a programmer to pay attention to several issues that have considerable impact on potential speed-up and performance [135, 110, 31, 5, 23, 75, 166] of an application running on such a system. General guidelines would be to investigate the following checklist – make sure that:

1. Code is highly parallel i.e. it exposes enough parallelism to make use of the several tens+ physical cores of the processor and the possibility to run a few hardware threads per core. In fact, at least two threads (sometimes 3 or 4) per core should be engaged in order to make use of the full potential of the coprocessor [135]. In practice, this requires that for a considerable percentage of the running time of the application, a large number of threads should be running in parallel.

2. Make sure that vectorization is used within the code as much as possible. This can be relied on thanks to automatic parallelization by a compiler. Intel's `icc` compiler allows compiling applications targeted for Xeon Phi

with, in particular, support for vectorization. The `icc` compiler will attempt vectorization and comments on its output can be checked when providing compilation flag `-vec-report<n>` where `n` is equal or greater than 1, the higher the `n`, the more information is displayed. Section 4.2.7 presents OpenMP directives that can be used for declaration of execution using SIMD instructions. There are several pragmas accepted by the Intel `icc` compiler that allow the indication to ignore dependencies and enforcing vectorization [110].

3. Use alignment of data structures in memory.

4. Cache is utilized efficiently among threads running within a core and between cores.

5. Proper thread affinity is used. Article [76] discusses how threads can be mapped to cores with either OpenMP affinity environment variables (discussed in Section 4.2.5) or Intel environment variables. The latter includes `KMP_AFFINITY` that specifies assignment of threads in OpenMP to hardware threads with granularity: fine – in which case each thread is bound to a single hardware thread, core – in which case four threads form a group which is bound to a core as well as affinity which can be compact, scatter and also balanced (the latter specific to Intel Xeon Phi):

   - compact – the next thread is assigned beside a previous one considering hardware threads within cores of a Xeon Phi,
   - scatter – the next thread will be assigned to the next physical core,
   - balanced – all threads are divided into possibly equal sized groups (with threads in a group with successive ids) and assigned to physical cores of the Intel Xeon Phi.

   `KMP_PLACE_THREADS` allows setting affinity more precisely using the following fields: `C` – cores, `T` – threads, `O` – denotes an offset in cores starting from core 0. For example, `export KMP_PLACE_THREADS=32C,3T,16O` will schedule 3 threads per core on a total of 32 cores starting from core 16. The default value for the offset is `0O`.

6. False sharing is minimized. Each core of the system has a cache memory. As cache needs to be coherent among the cores, it may result in an overhead. For instance, if threads executing on various cores update array cells that are in one cache line, this will result in overhead due to cache lines that need to be coherent among nodes. Consequently, if various threads modify data in memory locations not even overlapping but in close proximity, falling into one cache line, this may result in considerable overhead across the system. So a code, while formally correct, may not execute as fast as it could if it was organized differently. As an example, in case of array elements of which are updated by several

threads, there is a risk of false sharing. There are a few ways of dealing with false sharing, including:

(a) Instead of updates in locations that may result in false sharing, each thread could use its own local copies with private variables and only update final memory space at the end of computations. This may potentially increase memory usage but may result in shorter execution time.

(b) Padding a type of an element updated by a thread so that the whole type is a multiple of the size of a cache line. Similarly to the first solution, this leads to shorter execution time at the cost of higher memory requirements.

7. Memory per thread is used efficiently.

8. Synchronization among threads is kept to minimum and optimum synchronization methods are used. For instance, refer to Section 6.3.2.

In case of Intel Xeon Phi x200, efficient execution of parallel codes will also involve decisions on the following (article [74] discusses these modes in more detail along with potential use cases):

1. memory mode that defines which type of memory (DRAM, MCDRAM) and how they are visible to an application:

   - Flat – both DRAM and MCDRAM are available for memory allocation (as NUMA nodes), the latter preferably for bandwidth critical data.
   - Cache – in this case MCDRAM acts as an L3 cache.
   - Hybrid – a part of MCDRAM is configured as cache and a part as memory that can be allocated by an application.
   - MCDRAM – only MCDRAM is available.

2. cluster mode that defines how requests to memory are served through memory controllers:

   - All2All (default in case of irregular DIMM configuration) – a core, a tag directory (to which a memory request is routed) and a memory channel (to which a request is sent in case data is not in cache) can be in various parts.
   - Quadrant (in case of symmetric DIMM configuration) – a tag directory and a memory channel are located in the same region.
   - Sub-NUMA clustering – in this mode regions (quarter – SNC4, half – SNC2) will be visible as separate NUMA nodes. A core, a tag directory and a memory channel are located in the same region

which potentially result in optimization if handled properly through affinity settings such as assignment of various processes to various nodes.

Apart from OpenMP constructs discussed in Section 4.2.8, offloading computations to an Intel Xeon Phi coprocessor is possible with the `offload` directive recognized by the Intel compiler, with the possibility of specifying space allocation and release on the device, data copying between the host and the device, specifying a particular device (in case several Xeon Phi cards are used) and also asynchronous launching of computations on a device to overlap the latter with computations on the host [57].

Regarding fabrics for communication used by MPI, settings for both intranode and internode can be selected using the `I_MPI_FABRICS` environment variable [94, Selecting Fabrics] such as `export I_MPI_FABRICS=<intranode fabric>:<internode fabric>`.

Work [24] discusses performance of various data representations, namely Structure of Arrays (SoA) and AoS (Array of Structures) when running parallel vectorized code on Intel Xeon and Intel Xeon Phi. As a conclusion, in terms of performance generally SoA should be preferred. Generally, it produces fewer instructions.

6.9.3 Clusters

For clusters, that consist of many nodes, the following techniques should be employed during development of parallel programs:

1. Expose enough parallelism in the application considering the number of computing devices: CPUs, streaming multiprocessors within GPUs or coprocessors.

2. Design parallelization and data granularity in a data partitioning algorithm in a way that takes into account performance differences between computing devices. Specifically, small data packets distributed among computing devices allow for fine grained load balancing but such a strategy involves more overheads for handling packets and results. On the other hand, large data packets do not allow for good load balancing in a heterogeneous environment and result in larger execution times [39, 141].

3. Optimize communication taking into account:

   - overlapping computations and communication,
   - minimization of communication operations and piggybacking i.e. in some cases information related to dynamic load balancing can be added to the data of the algorithm itself.

- Optimization techniques and best practices for parallel codes

4. Optimize cache usage such that as much data as possible in consecutive computations is used from a nearby cache.

5. Assess whether idle times can be hidden in the algorithm. Specifically, for some so-called irregular problems, some processes or threads may run out of data or computations unexpectedly which leads to inefficient use of computing devices. It might be possible to hide those idle times e.g. by spawning two or more processes per core to minimize this risk at the cost of some overhead.

6. Design and implement a load balancing strategy. Note that there can be a trade-off between the cost of load balancing and potential gains. Specifically, a load balancing algorithm may try to maintain approximately the same load (within certain ranges) on all nodes with high frequency which would minimize the risk of idle times showing up on the nodes. However, this process also consumes CPU cycles and entails communication costs for exchanging load information. On the other hand, waiting too long with load balancing does not involve these overheads but may result in idle times until computing devices become active again.

7. Implement checkpointing for long running applications, as described in Section 6.7.

6.9.4 Hybrid systems

Hybrid systems can be thought of as systems with different processors that either require programming using various APIs and/or are located at various levels in terms of system architecture. Some processors might be better suited for specific types of codes and consequently might require specific optimizations and differ in performance and often power requirements. An example would be a multicore CPU and a GPU, a multicore CPU and a Xeon Phi hybrid system. It should also be noted that integration of several nodes of such types into a cluster creates a multilevel, hybrid and heterogeneous system in which parallelization must be implemented among nodes and within nodes.

Apart from specific optimizations valid for particular types of systems, a hybrid, heterogeneous system requires paying attention and optimization of the following:

1. Data granularity, data partitioning and load balancing, minimization of synchronization overheads. Often there are optimal batch sizes (Section 6.2) that can even be adjusted at runtime for lowest execution time [46].

2. Management of computations at lower levels. Specifically, within a node it is preferable to dedicate CPU cores to threads that will manage computations on computing devices such as GPUs or Xeon Phi cards [46].

3. Overlapping communication and computations, such as:

- overlapping computations on CPUs and internode communication using non-blocking MPI calls or threads,
- overlapping computations on accelerators such as GPU and host-device communication e.g. using streams with CUDA (Section 6.1.2).

4. Potentially various affinities can be used for various devices. Specifically `MIC_ENV_PREFIX` allows setting a prefix and then an affinity for a Xeon Phi in such a case, as follows:

```
export MIC_ENV_PREFIX=PHI
export PHI_KMP_AFFINITY=balanced
export PHI_KMP_PLACE_THREADS=60c,3t
export PHI_OMP_NUM_THREADS=180
```

5. Multilevel parallel programming can be naturally achieved with a combination of APIs discussed in this book e.g. MPI for internode communication and e.g. OpenMP for management of computations within a node with spawning work on various devices such as GPUs with CUDA, and nested parallelism for parallelization among CPU cores.

Performance wise, best results in a hybrid environment are achieved for various devices that offer reasonably similar performance. Otherwise, a parallel implementation might struggle to get a considerable performance gain from adding slower devices to a system.

CHAPTER 2

Determining an Exaflop Strategy

CONTENTS

2.1 FOREWORD BY JOHN LEVESQUE

In 1965 I delivered bread for Mead's Bakery and was working on finishing my degree at the University of New Mexico. A year later I was hired at Sandia National Laboratory in Albuquerque, New Mexico where I worked for researchers analyzing data from nuclear weapon tests. In 1968, I went to work for Air Force Weapons Laboratory and started optimizing a finite difference code called "Toody". The project looked at ground motions from nuclear explosions. All I did was step through Toody and eliminate all the code that was not needed. Toody was a large application with multi-materials, slip lines, and lots of other options. Well, the result was a factor of 20 speedup on the problems of interest to the project. That is how I got the bug to optimize applications, and I still have not lost the excitement of making important applications run as fast as possible on the target hardware. The target hardware at that time was a Control Data 6600. I worked closely with Air Force Captain John Thompson and we had one of two 6600s from midnight to 6:00 AM most nights. John and I even used interactive graphics to examine the applications progress. We could set a sense switch on the CDC 6600 and the program would dump the velocity vector field to a one-inch tape at the end of the next timestep. We then took the tape to a Calcomp plotter to examine

the computation. If things looked good, we would go over to the Officers club and have a couple beers and shoot pool. The operator would give us a call if anything crashed the system.

There was another Captain "X" (who will remain anonymous) who was looking at the material properties we would use in our runs. There are two funny stories about Captain X. First, this was in the days of card readers, and Captain X would take two trays of cards to the input room when submitting his job. My operator friend, who was rewarded yearly with a half gallon of Seagram's VO at Christmas time, asked if I could talk to Captain X. I told the operator to call me the next time Captain X submitted his job, and I would come down and copy the data on the cards to a tape that could be accessed, instead of the two trays of cards. The next time he submitted the job, X and I would be watching from the window looking into the computer room. I told the operator to drop the cards while we were looking so X would understand the hazards of using trays of cards. Well, it worked – X got the shock of his life and wouldn't speak to me for some time, but he did start using tapes.

The other story is that X made a little mistake. The sites were numbered with Roman numerals, and X confused IV with VI. He ended up giving us the wrong data for a large number of runs.

Some Captains, like John Thompson, were brilliant, while others, like X, were less so. John and I had an opportunity to demonstrate the accuracy of our computations. A high explosive test was scheduled for observing the ground motions from the test. We had to postmark our results prior to the test. We worked long hours for several weeks, trying our best to predict the outcome of the test. The day before the test we mailed in our results, and I finally was able to go home for dinner with the family. The next day the test was cancelled and the test was never conducted. So much for experimental justification of your results.

2.2 INTRODUCTION

It is very important to understand what architectures you expect to see moving forward. There are several trends in the high performance computing industry that point to a very different system architecture than we have seen in the past 30 years. Due to the limitations placed on power utilization and the new memory designs, we will be seeing nodes that look a lot like IBM's Blue Gene on a chip and several of these chips on a node sharing memory. Like IBM's Blue Gene, the amount of memory will be much less than what is desirable (primarily due to cost and energy consumption).

Looking at the system as a collection of nodes communicating across an interconnect, there will remain the need to have the application communicate effectively – the same as the last 20 years. The real challenges will be on the node. How will the application be able to take advantage of thousands of degrees of parallelism on the node? Some of the parallelism will be in the form of a MIMD collection of processors, and the rest will be more powerful SIMD

instructions that the processor can employ to generate more flops per clock cycle. On systems like the NVIDIA GPU, many more active threads will be required to amortize the latency to memory. While some threads are waiting for operands to reach the registers, other threads can utilize the functional units. Given the lack of registers and cache on the Nvidia GPU and Intel KNL, latency to memory is more critical since the reuse of operands within the cache is less likely. Xeon systems do not have as much of an issue as they have more cache. The recent Xeon and KNL systems also have hyper-threads or hardware threads – also called simultaneous multithreading (SMT). These threads share the processing unit and the hardware can context switch between the hyper-threads in a single clock cycle. Hyper-threads are very useful for hiding latency associated with the fetching of operands.

While the NVIDIA GPU uses less than 20 MIMD processors, one wants thousands of threads to be scheduled to utilize those processors. More than ever before, the application must take advantage of the MIMD parallel units, not only with MPI tasks on the node, but also with shared memory threads. On the NVIDIA GPU there should be thousands of shared memory threads, and on the Xeon Phi there should be hundreds of threads.

The SIMD length also becomes an issue since an order of magnitude of performance can be lost if the application cannot utilize the SIMD (or vector if you wish) instructions. On the NVIDIA GPU, the SIMD length is 32 eight-byte words, and on the Xeon Phi it is 8 eight-byte words. Even on the new generations of Xeons (starting with Skylake), it is 8 eight-byte words.

Thus, there are three important dimensions of the parallelism: (1) message passing between the nodes, (2) message passing and threading within the node, and (3) vectorization to utilize the SIMD instructions.

Going forward the cost of moving data is much more expensive than the cost of doing computation. Application developers should strive to avoid data motion as much as possible. A portion of minimizing the data movement is to attempt to utilize the caches associated with the processor as much as possible. Designing the application to minimize data motion is the most important issue when moving to the next generation of supercomputers. For this reason we have dedicated a large portion of the book to this topic. Chapters 3 and 6, as well as Appendix A will discuss the cache architectures of the leading supercomputers in detail and how best to utilize them.

2.3 LOOKING AT THE APPLICATION

So here we have this three-headed beast which we need to ride, and the application is our saddle. How are we going to redesign our application to deliver a good ride? Well, we need a lot of parallelism to get to an exaflop; we are going to need to utilize close to a billion degrees of parallelism. Even for less ambitious goals we are going to need a million degrees of parallelism. So the first question is: "do we have enough parallelism in the problem to be solved?" While there are three distinct levels of parallelism, both the message passing

and the threading should be at a high level. Both have relatively high overhead, so granularity must be large enough to overcome the overhead of the parallel region and benefit from the parallelization. The low level parallel structures could take advantage of the SIMD instructions and hyper-threading.

If the application works on a 3D grid there may be several ways of dividing the grid across the distributed nodes on the inter-connect. Recently, application developers have seen the advantage of dividing the grid into cubes rather than planes to increase the locality within the nodes and reduce the surface area of the portion of the grid that resides on the node. Communication off the node is directly proportional to the surface area of the domain contained on the node. Within the node, the subdomain may be subdivided into tiles, once again to increase the locality and attempt to utilize the caches as well as possible.

A very well designed major application is the Weather Research and Forecasting (WRF) Model, a next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs. Recent modifications for the optimization of the WRF application are covered in a presentation by John Michalakes given at Los Alamos Scientific Laboratories in June, 2016 [22]. These techniques were further refined for running on KNL, which will be covered in Chapter 8. The salient points are:

1. MPI decomposition is performed over two-dimensional patches.
2. Within the MPI task the patch is divided into two-dimensional tiles, each having a two-dimensional surface and depth which represents the third dimension.
3. The dimensions of the tile were adjustable to most effectively utilize the cache.

When identifying parallelism within the application, care must be taken to avoid moving data from the local caches of one processor to the local caches of another processor. For the past 15 years, using MPI across all the cores in a distributed multicore system has out-performed employing OpenMP on the node with MPI only between nodes. The principal advantage that MPI had was that it forced data locality within a multicore's NUMA memory hierarchy. On the other hand, OpenMP has no notion of data locality. On all Intel architectures the level-1 and level-2 caches are local to the processor. However, the level-3 cache tends to be distributed around the socket, and then there tends to be multiple sockets on a node (2 to 4). If the data being accessed with the OpenMP region extends to level-3 cache, then there will be cache interference between the threads. Examples will illustrate this in Chapter 6. Additionally, some looping structures may be parallelized with a different decomposition than other looping structures. Such an approach would necessitate moving data from one thread's low-level caches to another thread's caches. To get the most out of the supercomputers of the future, one must minimize this data

motion. Recently, several applications have employed OpenMP successfully in a SPMD style like MPI. While this approach, discussed in Chapter 8, is more difficult to implement, it performs extremely well on multi/manycore NUMA architectures.

Since most large applications generate a lot of data, we should design the application to perform efficient I/O. The decomposition of the problem across the nodes and within the node can be done to allow asynchronous parallel I/O. See Appendix E for an I/O discussion.

Your strategy depends upon the state of the existing application. When writing an application from scratch, one has complete flexibility in designing an optimized application. When a legacy application is the starting point, the task can be significantly more difficult. In this book we will concentrate on the legacy application.

Your strategy depends upon the target problem, which hopefully is large enough to utilize the million degrees of parallelism on the new system. There are two approaches for scaling a problem to larger processor counts. First, there is the idea of weak scaling where the problem size grows as the number of processors grow. Consequently, each processor has a constant amount of work. This is a great way to scale the problem size if the increased size represents good science. For example, in S3D, a combustion and computational fluid dynamics application, the increase in problem size gives finer resolution and therefore better science. S3D will be discussed further in Chapter 9. On the other hand, many weather/climate codes at some point cannot take advantage of finer resolution, since the input data is sparse. If the finer grid cannot be initialized, then often times the increased refinement is of no help. For the weather/climate application, strong scaling is employed. With strong scaling, the problem size is fixed, and as the number of processors are increased, the amount of work performed by each processor is reduced. Strong scaling problem sets are the most difficult to make effectively utilize large parallel systems. Strong scaling does have a "sweet spot" when the mesh size within each MPI task is small enough to fit into the low level cache. A super-linear performance boost will be seen when this point is reached. This is particularly true on KNL when the problem size can fit within the MCDRAM, allowing the application to run completely out of the high bandwidth memory. However, exercise caution with this line of thinking. Once the problem is using enough nodes to run within MCDRAM, one should determine if the increase in system resources required to obtain the faster results comes with a similar decrease in compute time. For example, if a strong scaling application obtains a factor of 1.5 decrease in wall clock time when scaled from 120 to 240 nodes, the decrease does not account for the factor of two increase in system resources. Of course some credit needs to be given to decreased turn-around which results in more productivity.

Given the problem size, the most important task is to find a parallel decomposition that divides the problem up across the processors in a way that not only has equal work on each processor, but also minimizes the commu-

nication between the processors. As mentioned earlier, the hope is to achieve locality within a domain for both MPI domains as well as threaded domains. With multidimensional problems, the surface area of the MPI domain should be minimized, which is achieved by using cube domains as depicted in Figure 2.3.1. For threaded regions, locality is achieved by threading across tiles. The size of the tile should allow the computation within the tile to be conducted within the cache structure of the processor. To achieve this, the working set (the total amount of memory utilized) should be less than the size of the level-2 cache. If multiple cores are sharing level-2 cache, then the working set per core must be reduced to account for the sharing.

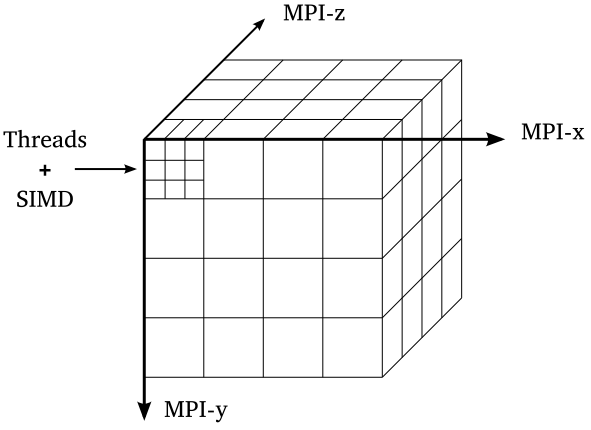


FIGURE 2.3.1 3D grid decomposition minimizing MPI surface area.

In large, multiphysics, irregular, unstructured grid codes, decomposing the computational grid can be a challenge. Poor decompositions can introduce excessive load imbalance and excessive communication. Many three-dimensional decomposition packages exist for addressing this problem, and the decomposition itself can take a significant amount of time.

In the decomposition across MPI tasks and threads, the processor characteristics must be taken into account. The state-of-the-art Xeon tends to have a modest number of cores within the node and few NUMA regions, so the number of MPI and threading domains are relatively small compared to what is required to run efficiently on a GPU. GPUs tend to have a similar number of symmetric processors (SM), and the SIMD units have a longer length (32 eight-byte words). However, the GPU requires thousands of independent threads for amortizing the latency to memory. Knight's Landing is closer to the Xeon, in that it has 60 to 70 cores on the node with a SIMD length of 8 for double-precision. KNL does not need as many independent threads since it only has four hyper-threads per core. This is a significant difference between the GPU and other systems; the GPU requires significantly more shared memory threading than the other systems, and it could require a sig-

nificantly different strategy for the threaded domains. While the Xeon and KNL can benefit from having multiple MPI tasks running within the node, within the GPU, all of the threads must be shared memory threads as MPI cannot be employed within the GPU.

The decomposition is also coupled with the data layout utilized in the program. While the MPI domain is contained within the MPI task's memory space, the application developer has some flexibility in organizing the data within the MPI task. Once again, irregular, unstructured grids tend to introduce unavoidable indirect addressing. Indirect addressing is extremely difficult and inefficient in today's architectures. Not only does operand fetching require fetching of the address prior to the operand, cache utilization can be destroyed by randomly accessing a large amount of memory with the indirect addressing. The previously discussed tile structure can be helpful, if and only if the memory is allocated so that the indirect addresses are accessing data within the caches. If the data is stored without consideration of the tile structure, then cache thrashing can result.

Without a doubt, the most important aspect of devising a strategy for moving an existing application to these new architectures is to design a memory layout that supplies addressing flexibility without destroying the locality required to effectively utilize the cache architecture.

2.4 DEGREE OF HYBRIDIZATION REQUIRED

For the past 15 years we have seen the number of cores on the node grow from 1 to 2 to 4 and now on to 40 to 70. Many believed employing threading on the node and MPI between nodes was the best approach for using such a system, and as the number of cores on the node grew that belief grew stronger. With the advent of the GPU, a hybrid combination of MPI and threading was a necessity. However, performance of all-MPI (no threading) on the Xeon and now even the KNL has been surprisingly good, and the need for adding threading on those systems has become less urgent.

Figure 2.4.1 shows the performance of S3D on 16 KNL nodes running a problem set that fits into KNL's high bandwidth memory. In this chart the lines represent the number of MPI tasks run across the 16 nodes. The ordinate is the number of OpenMP threads employed under each MPI task. The performance is measured in timesteps per second, and higher is better. The graph illustrates that using 64 MPI tasks on the node for a total of 1024 MPI tasks gives the best performance and in this case two hyper threads per MPI task increases the performance and four hyper threads decreases the performance. S3D is very well threaded; however, it does suffer from not using a consistent threading strategy across all of the computational loops. More will be discussed about S3D in Chapter 9.

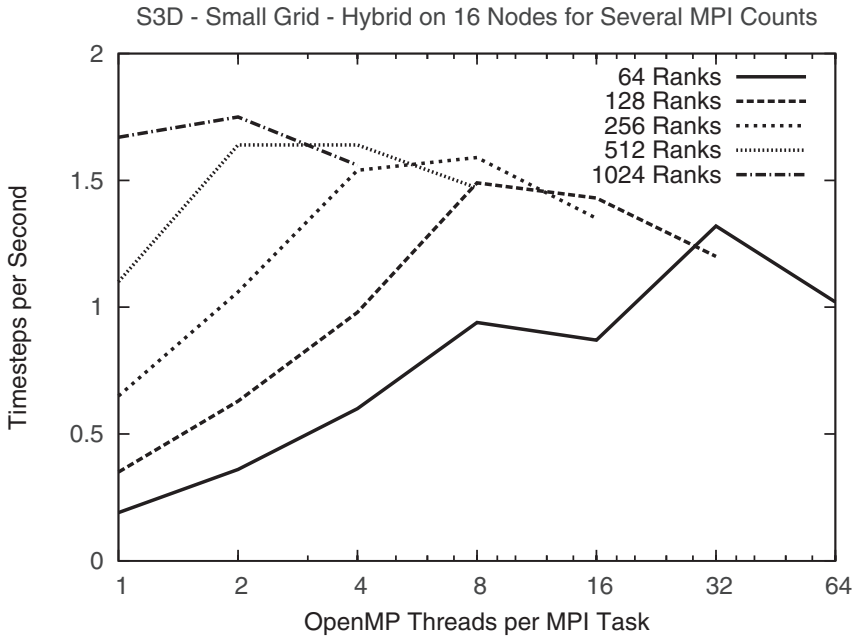


FIGURE 2.4.1 Performance of S3D on KNL with different MPI rank counts.

The two primary reasons for the superior performance of MPI on these systems are the locality forced by MPI and the fact that MPI allows the tasks to run asynchronously, which allows for better utilization of available memory bandwidth. When MPI is run across all the cores on a node, the MPI task is restricted to using the closest memory and cache structure to its cores. On the other hand, threading across cores allows the threads to access memory that may be further away, and multiple threads running across multiple cores have a chance to interfere with each other's caches. On KNL, running MPI across all the cores and threading for employing the hyper-threads seems to be a good starting point in a hybrid MPI+OpenMP approach. The performance of a hybrid application is directly proportional to the quality of the threading. In fact, as seen in the OpenMP chapter, the SPMD threading approach, which mimics the operation of MPI tasks, performs very well. OpenMP has one tremendous advantage over MPI: it can redistribute work within a group of threads more efficiently than MPI can, since OpenMP does not have to move the data. There is and will always be a place for well-written OpenMP threading. Of course, threading is a necessity on the GPU; one cannot run separate MPI tasks on each of the symmetric processors within the GPU.

2.5 DECOMPOSITION AND I/O

The incorporation of a good decomposition and efficient parallel I/O go hand in hand. While we do not talk about efficient parallel I/O in great detail in this book, some discussion is provided in Appendix E, and it should always be considered when developing a good strategy for moving to new systems.

2.6 PARALLEL AND VECTOR LENGTHS

When the application has to scale to millions of degrees of parallelism, the division of the dimensions of the problem across the MPI ranks, threads, and SIMD units is an important design criteria. In Chapter 7, we will see that vector performance on the Xeon and KNL systems top out around 80 to 100 iterations, even though the vector unit is much smaller. Additionally, we want the vector dimension to be aligned on cache boundaries and contiguous in memory – non-unit striding will significantly degrade performance. On the GPU, the vectors are executed in chunks of 32 contiguous operands, and since the GPU wants a lot of parallel work units, one way is by having very long vector lengths.

Beyond the vector length, the next question is how to divide parallelism between MPI and threading. As was discussed earlier, the Xeon and KNL seem to do well with a lot of MPI. Running MPI across 64 cores within each node and across 10,000 nodes gives 640,000 MPI tasks and, in this case, on the order of 4 to 8 threads with hyper-threads. On the GPU system, one would want the number of MPI tasks to be equal to the number of GPUs on each node, and one would want thousands of threads on the node. Much more threaded parallelism is required to effectively utilize the GPU system.

2.7 PRODUCTIVITY AND PERFORMANCE PORTABILITY

An important consideration when moving to the next generation of multi/manycore systems is striving to create a refactored application that can run well on available systems. There are numerous similarities between the multi/manycore systems of today. Today's systems have very powerful nodes, and the application must exploit a significant amount of parallelism on the node, which is a mixture of MIMD (multiple instruction, multiple data) and SIMD (single instruction, multiple data). The principal difference is how the application utilizes the MIMD parallelism on the node. Multicore Xeons and manycore Intel Phi systems can handle a significant amount of MPI on the node, whereas GPU systems cannot. There is also a difference in the size of the SIMD unit. The CPU vector unit length is less than 10, and the GPU is 32. Since longer vectors on the multi/manycore systems do run better than shorter vectors this is less of an issue. All systems must have good vectorized code to run well on the target systems. However, there is a problem.

"Software is getting slower more rapidly than hardware becomes faster."

Niklaus Wirth
Chief Designer of Pascal, 1984 Turing Award Winner

Prior to the advent of GPU systems and KNL, application developers had a free ride just using MPI and scalar processing, benefiting from the increased number of cores on the node. Today we have a situation similar to the movement to distributed memory programming which required the incorporation of message passing. Now, to best utilize all the hardware threads (including hyper-threads) applications have to be threaded and to harness the vector processing capability the applications must vectorize. Given the tremendous movement to C++, away from the traditional HPC languages Fortan and C, the modifications to achieve vectorization and threading are a tremendous challenge. Unless the developers really accept the challenge and refactor their codes to utilize threading and vectorization they will remain in the gigaflop performance realm and realize little improvement on the new HPC systems. The cited reference to the COSMOS weather code is an example of how that challenge can be realized with some hard work.

The "other p" – there is a trend in the industry that goes against creating a performant portable application: the attempt to utilize high-level abstractions intended to increase the productivity of the application developers. The movement to C++ over the past 10 to 15 years has created applications that achieve a lower and lower percentage of peak performance on today's supercomputers. Recent extensions to both C++ and Fortran to address productivity have significantly contributed to this movement. Even the developer of C++, Bjarne Stroustrup, has indicated that C++ can lure the application developer into writing inefficient code.

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

"Within C++, there is a much smaller and cleaner language struggling to get out."

Bjarne Stroustrup
Chief Designer of C++

The productivity argument is that the cost of talented labor is greater than the cost of the high performance computing system being utilized, and it is

too time consuming to improve the performance of the application. On the other hand, time spent optimizing application performance not only makes better use of expensive machines, it also reduces operational costs as energy costs continue to rise for future systems.

Several years ago, a team lead by Thomas Schulthess and Oliver Fuhrer of ETH Zurich refactored the production version of COSMOS, the climate modeling application used by MetroSwiss, and found that not only did the application run significantly faster on their current GPU system, the cost of the effort would be more than repaid by the savings in energy costs over a couple of years [12]. Figure 2.7.1 shows the performance increase and Figure 2.7.2 shows the power consumption decrease for the refactored application.

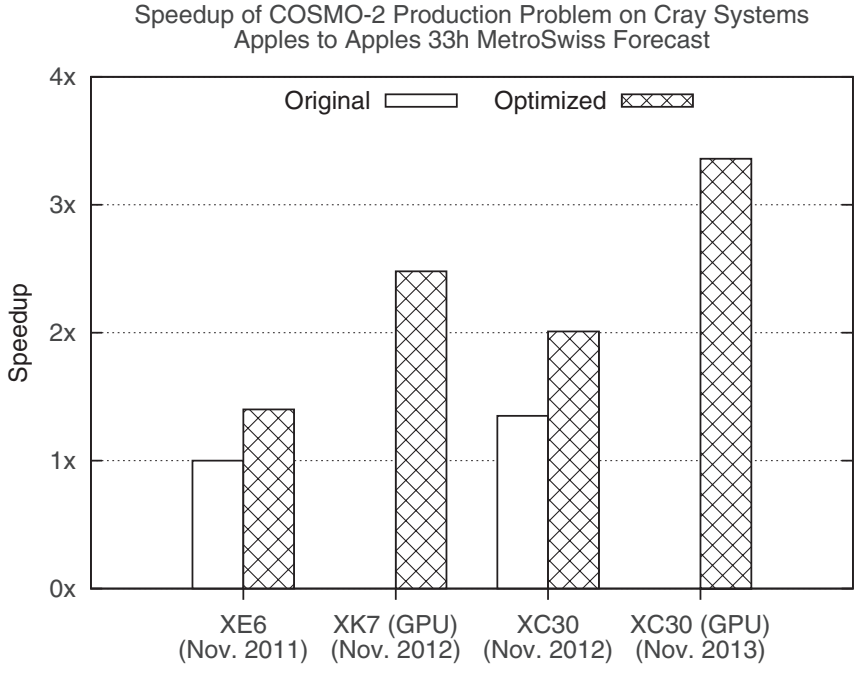


FIGURE 2.7.1 Performance increase of refactored COSMOS code.

The work on COSMOS employed C++ meta-programming templates for the time-consuming dynamical core, which resulted in the instantiation of CUDA kernels on the GPU and optimized assembly on the x86 systems. This work is an example of the developers' understanding the architecture and restructuring the application to utilize its features with the high-level C++ abstractions.

The bulk of the code – the physics – was Fortran, and OpenACC was used for the port to the accelerator. This is an excellent example that shows how an investment of several person-years of effort can result in an optimized application that more than pays for the investment in the development cost.

This work does show that a well-planned design can benefit from C++ high-level abstraction. However, there has to be significant thought put into the performance of the generated code.

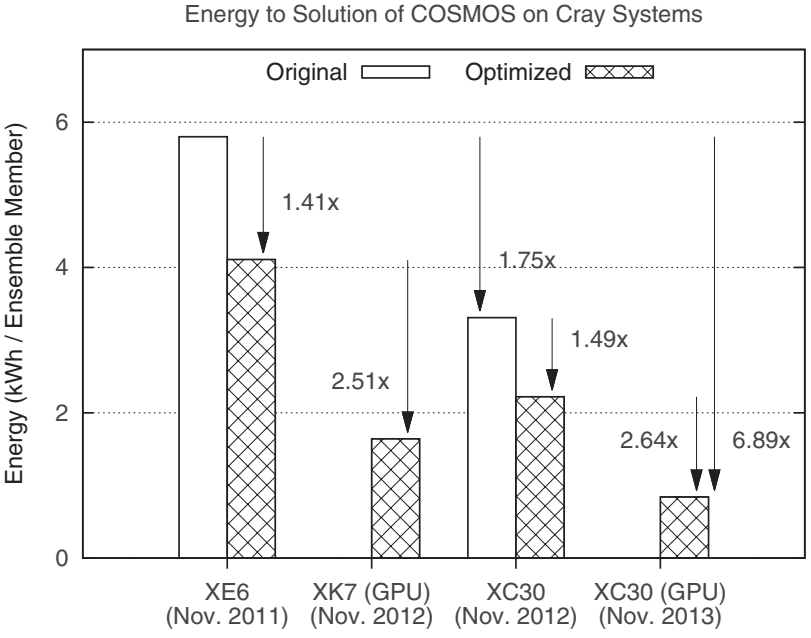


FIGURE 2.7.2 Energy reduction of refactored COSMOS code.

Data motion is extremely expensive today and will be more expensive in the future, both in energy and time, and many of the high-level abstractions in the language can easily introduce excessive memory motion in an application. Retrofitting a large C++ framework with a high-level abstraction requires application developers to move data into a form acceptable to the abstractions and/or refactor their applications to have the abstraction manage their data structures. The first approach introduces too much data motion, and the second approach often requires a significant rewrite. Once such a rewrite has been performed, the application is dependent upon those interfaces making efficient use of the underlying architecture. Additionally, most complex multi-physics applications are a combination of computations that flow from one set of operations to another, and breaking that flow up into calls to the low-level abstractions could result in poor cache utilization and increased data motion.

Much of the blame of this productivity movement has to be placed on the language standards committees that introduces semantics that make the application developer more productive without thinking about compilation issues or the efficiencies of executing the compiled code on the target system. Considering the recent additions, both Fortran and C++, it seems that the

committee really does not care about how difficult it might be to optimize the language extensions and that their principal goal is to make programmers more productive. When users see an interesting new feature in the language, they assume that the feature will be efficient; after all, why would the language committee put the feature in the language if it wouldn't run efficiently on the target systems?

At some point this trend to productivity at the expense of performance has to stop. Most, if not all of the large applications that have taken the productivity lane have implemented MPI and messaging outside of the abstraction, and they have obtained an increase in performance from the parallelism across nodes. Increased parallelism must be obtained on the node in the form of threading and/or vectorization, with special attention paid to minimizing the movement of data within the memory hierarchy of the node. At some point, application developers have to put in extra work to ensure that data motion on the node is minimized and that threading and vectorization are being well utilized.

2.8 CONCLUSION

The primary strategy for designing a well-performing application for these systems is to first get the memory decomposition correct. We must have good locality to achieve a decent performance on the target system. This memory layout imposes a threading decomposition on the node, which could also be employed for the MPI decomposition on the node. Ideally, a legacy code may already have found a good decomposition for MPI usage across all the nodes. In review, we would like the vector dimension to be contiguous, and we would like the threads to operate on a working set that fits into the cache structure for a single core. Then we want to ensure that the tasks do not interfere with each other, by either doing OpenMP right or by using MPI.

2.9 EXERCISES

2.1 Compare and contrast weak and strong scaling.

2.2 When choosing a parallel decomposition which equalizes computational load, what other aspect of the decomposition is of critical importance?

2.3 What is a good decomposition strategy to use with multidimensional (e.g., 3D) problems to minimize communication between MPI domains?

2.4 *Construct a case study*: Consider an application and a target system with respect to the amount of parallelism available (the critical issue of data motion will be covered in more detail in later chapters).

a. Select a target system and characterize it in terms of:
   i. Number of nodes.

ii. Number of MIMD processors per node.
iii. Number of SIMD elements per processor.

b. Select an application for further examination:
   i. Identify three levels of parallelism in the application (e.g., grid decomposition, nested loops, independent tasks).
   ii. Map the three levels of parallelism identified in the application to the three levels available on the target system: MPI, threads, and vectors.

c. For each level of parallelism identified in the application and mapped to the target system, compare the amount of parallelism available in the application at that level to the amount of parallelism available on the system:
   i. Compare the number of MPI ranks the application could utilize to the number of nodes available on the system as well as the number of MIMD processors per node.
   ii. Compare the number of threads per MPI rank the application could utilize to the number of MIMD processors on a node as well as the number of hardware threads per processor.
   iii. Compare the trip count of the chosen vectorizable loops to the length and number of the system's SIMD units.

   Do these comparisons show a good fit between the parallelism identified in the application and the target system? If not, try to identify alternative parallelization strategies.

2.5 Consider the execution of HPL on the KNL system and assume with full optimization it sustains 80% of peak performance on the system:

a. What would the percentage of peak be if we only used one of the 68 processors on the node?

b. What would the percentage of peak be if we did not employ SIMD instructions?

c. What would the percentage of peak be if we did not employ SIMD instructions and we only used one of the 68 processors on the node?

Chapter 1

Contemporary High Performance Computing

Jeffrey S. Vetter

Oak Ridge National Laboratory and Georgia Institute of Technology

1.1 High Performance Computing

High Performance Computing (HPC) is used to solve a number of complex questions in computational and data-intensive sciences. These questions include the simulation and modeling of physical phenomena, such as climate change, energy production, drug design, global security, and materials design; the analysis of large data sets, such as those in genome sequencing, astronomical observation, and cybersecurity; and the intricate design of engineered products, such as airplanes.

It is clear and well-documented that HPC can be used to generate insight that would not otherwise be possible. Simulations can augment or replace expensive, hazardous, or impossible experiments. Furthermore, in the realm of simulation, HPC has the potential to suggest new experiments that escape the parameters of the observable.

Although much of the excitement about HPC focuses on the largest architectures and on specific benchmarks, such as TOP500, there is a much deeper and broader commitment from the international scientific and engineering community than is first apparent. In fact, it is easy to lose track of history in terms of the broad uses of HPC and the communities that design, deploy, and operate HPC systems and facilities. Many of these sponsors and organizations have spent decades developing scientific simulation methods and software, which serves as the foundation of HPC today. During this time, this community has worked closely with countless vendors to foster the sustained development and deployment of HPC systems internationally.

1.2 Terascale to Petascale: The Past 15 Years in HPC

By any measure, the past 15 years have witnessed dramatic increases in both the use and scale of HPC. Thinking back, it was 1997, only 15 years ago, when the ASCI Red system at Sandia National Laboratories in Albuquerque, New Mexico, broke the 1 TFlop/s barrier on TOP500 Linpack using 7264 Pentium P6 processors over a proprietary interconnection network. In 2012, the Sequoia system, a Blue Gene/Q with its system-on-a-chip (SoC) architecture at Lawrence Livermore surpassed 16 PFlop/s on the same benchmark: an increase of 16,000 times in 15 years! It is impressive, indeed, when considering the well-known fact that the performance of commodity microprocessors has slowed due to power and thermal constraints [KBB+08, FPT+07, HTD11].

Although much of the focus is on the #1 system on TOP500, and its architecture, it is important to promote the fact that there are dozens, in fact, hundreds of systems around the world in daily use for solving HPC problems. As shown in Table 1.1, the architectures of these systems span the range from a fully customized processor and interconnection network to a complete commodity solution. In contrast, many of these systems share a tremendous amount of software on their software stack, as listed in Table 1.3. In fact, much of this software is open source software that organizations can download, port, and install on any system.

Aside from the popular TOP500 Linpack benchmark, this period has also witnessed a dramatic performance increase in the Gordon Bell Prizes. As illustrated in Table 1.2, since 1993 (the year that TOP500 started), the increase in the Gordon Bell performance prize winners has been almost 5 orders of magnitude (from 60 GFlop/s to 3,080,000 GFlop/s).

In this book, we have selected contributions from a combination of sites, systems, applications, and sponsors. Rather than focus simply on the architecture or the application, we focus on *HPC ecosystems* that have made this dramatic progress possible. Though the very word ecosystem can be a broad, all-encompassing term, it aptly describes high performance computing. That is, HPC is far more than one sponsor, one site, one application, one software system, or one architecture. Indeed, it is a community of interacting entities in this environment that sustains the community over time. As Table 1.1 illustrates, we have 21 chapters from authors around the world describing their ecosystem, and often focused on their existing flagship system. We not only included the largest systems in the world, but also innovative systems that advance a particular idea, such as the Gordon system at San Diego Supercomputer Center with its focus on data-intensive computing. Likewise, with a growing use of HPC internationally, we have included sites from Europe, China, and Japan. Ultimately, we would have liked to include many more chapters, but we simply ran out of room in the book.

Chapter authors were asked to address the following topics in their chapters:

1. Sponsor and site history
2. Highlights of applications, workloads, and benchmarks
3. Systems overview
4. Hardware architecture
5. System software
6. Programming systems
7. Storage, visualization, and analytics

TABLE 1.1: Significant systems in HPC.

System	Type	Organization	Location	Country
Blacklight	SGI UV	Pittsburgh Supercomputing Center	Pittsburgh	USA
Blue Waters	Cray XE6, XK6	National Center for Supercomputing Applications	Urbana	USA
JUGENE	Blue Gene/P	Jülich Research Centre, Forschungszentrum Jülich	Jülich	Germany
Gordon	x86/IB Cluster	San Diego Supercomputing Center	San Diego	USA
HA-PACS	x86/IB/GPU cluster	University of Tsukuba	Tsukuba	Japan
Keeneland	x86/IB/GPU cluster	Georgia Institute of Technology	Atlanta	USA
Kraken	Cray XT5	National Institute for Computational Science	Knoxville	USA
Lomonosov	T-Platforms	Moscow State University	Moscow	Russia
Mole-8.5	x86/IB/GPU cluster	Chinese Academy of Sciences, Institute of Process Engineering	Beijing	China
Monte Rosa	Cray XE6	Swiss National Supercomputing Centre	Lugano	Switzerland
Numerous	Cray XE6	DOD High Performance Modernization Project	Numerous	USA
Pleiades	x86/IB Cluster	NASA Ames	Mountain View	USA
Roadrunner	x86/Cell/IB cluster	Los Alamos National Laboratory	Los Alamos	USA
Sequoia, Mira	Blue Gene/Q	Lawrence Livermore National Laboratory and Argonne National Laboratory	Livermore and Argonne	USA
TERA 100	x86/IB Cluster	CEA	Arpajon	France
Tianhe-1A	x86/IB/GPU cluster	National University of Defense Technology	Tianjin	China
Titan	Cray XK6	Oak Ridge National Laboratory	Oak Ridge	USA
Tsubame 2.0	x86/IB/GPU cluster	Tokyo Institute of Technology	Tokyo	Japan
Future Grid	Grid/Cloud	Indiana University	Bloomington	USA
Magellan	Grid/Cloud	Argonne National Laboratory and Lawrence Berkeley National Laboratory	Argonne and Berkeley	USA
LLGrid	Grid/Cloud	MIT Lincoln Laboratory	Boston	USA

8. Data center/facility
9. Site HPC statistics

1.3 Performance

The most prominent HPC benchmark is the TOP500 Linpack benchmark. TOP500 has qualities that make it valuable and successful: easily scaled problem size, straightforward validation, an open source implementation, almost 20 years of historical data, a large audience familiar with the software, and well managed guidelines and submission procedures. As shown in Table 1.2, the TOP500 prize for the #1 system has been awarded since 1993. In that time, performance for this prize has grown from 124 GFlops to 17,590,000 GFlops. This increase is 5 orders of magnitude!

Although TOP500 Linpack is a formidable and long-lived benchmark, it does not fully capture the spectrum of applications across HPC as is often pointed out. TOP500 Linpack aside, the HPC community has created many metrics and benchmarks for tracking the success of different HPC solutions. These alternatives include the Gordon Bell Prize, the HPC Challenge benchmark (cf. Ch. 2.1), the NAS Parallel Benchmarks, the Green500 benchmark (cf. Ch. 3.1), the SHOC benchmarks (cf. Ch. 7.8), along with a large number of procurement benchmarks from DoE, DoD, NASA, and many other organizations.

1.3.1 Gordon Bell Prize

Aside from standard benchmarks, another indicator of growth in HPC is their performance on real scientific problems. The most well-known scientific accomplishment for these types of problems is the annual ACM Gordon Bell Prize, which is presented at the ACM/IEEE SC Conference. The prize requires authors to submit descriptions, scientific results, and performance results for real-world applications. These submissions are then judged by a group of peers, and one submission is awarded a prize for sustained performance. In one year, the committee can also award other prizes for exemplary submissions in price-performance, or other special categories. In general, the award is meant to reward innovation in applying high performance computing to applications in science. As illustrated in Table 1.2, over the years, these prizes have been awarded to teams with applications ranging from computational fluid dynamics to nanoparticle design to climate modeling to seismic modeling.

Since 1987, the year that this prize started, the sustained performance category has shown an increase of nearly seven orders of magnitude on systems ranging from 8 cores to 663,552 cores. Meanwhile, the price-performance prize has increased 5.5 orders of magnitude from 1989 to 2009. Note that until recently the Gordon Bell submission procedure has not had strict guidelines for submitting performance results, so some of the performance results are not completely documented. For example, it is known that the floating point rate listed in the table is a mix of single precision, double precision, and mixed precision. Nevertheless, in each year of the award, the committee recognized a winning application with exemplary performance and science.

More surprisingly, since 1993 (the same year that TOP500 started), the increase in the performance of Gordon Bell awards, from 60 Gflop/s to 4.45 Pflop/s, is nearly identical to the increase in performance of the TOP500 #1 system of 5 orders of magnitude: 4.87 to 5.15, respectively.

TABLE 1.2: Gordon Bell Prize winners for sustained performance compared with the #1 system on the TOP500 ranking since their inception.

Year	Type	Application	System	Cores	Increase Log10	GB Prize Gflop/s	Increase Log10	TOP500 #1 Gflop/s	Increase Log10
1987	PDE	Structures	N-CUBE	1,024		0.45			
1988	PDE	Structures	Cray Y-MP	8		1			
1989	PDE	Seismic	CM-2	2,048		5.6			
1990	PDE	Seismic	CM-2	2,048		14			
1991		*NO PRIZE AWARDED*							
1992	NB	Gravitation	Delta	512		5.4			
1993	MC	Boltzmann	CM-5	1,024	-	60	-	124	-
1994	IE	Structures	Paragon	1,904	0.27	143	0.38	170	0.14
1995	MC	QCD	NWT	128	-0.90	179	0.47	170	0.14
1996	PDE	CFD	NWT	160	-0.81	111	0.27	368	0.47
1997	NB	Gravitation	ASCI Red	4,096	0.60	170	0.45	1,338	1.03
1998	DFT	Magnetism	T3E-1200	1,536	0.18	1,020	1.23	1,338	1.03
1999	PDE	CFD	ASCI Blue Pacific	5,832	0.76	627	1.02	2,379	1.28
2000	NB	Gravitation	Grape-6	96	-1.03	1,349	1.35	4,938	1.60
2001	NB	Gravitation	Grape-6	1,024	0.00	11,550	2.28	7,226	1.77
2002	PDE	Climate	Earth Simulator	5,120	0.70	26,500	2.65	35,860	2.46
2003	PDE	Seismic	Earth Simulator	1,944	0.28	5,000	1.92	35,860	2.46
2004	PDE	CFD	Earth Simulator	4,096	0.60	15,200	2.40	70,720	2.76
2005	MD	Solidification	BG/L	131,072	2.11	101,700	3.23	280,600	3.35
2006	DFT	Electronic structures	BG/L	131,072	2.11	207,000	3.54	280,600	3.35
2007	MD	Kelvin-Helmholtz	BG/L	131,072	2.11	115,000	3.28	478,200	3.59
2008	DFT	Crystal structures	Jaguar/XT-5	150,000	2.17	1,352,000	4.35	1,105,000	3.95
2009	DFT	Nanoscale systems	Jaguar/XT-5	147,464	2.16	1,030,000	4.23	1,759,000	4.15
2010	FMM	Blood flow	Jaguar/XT-5	196,608	2.28	780,000	4.11	2,566,000	4.32
2011	RSDFT	Nanowires	K/Fujitsu	442,368	2.64	3,080,000	4.71	10,510,000	4.93
2012	NB	Astrophysics	K/Fujitsu	663,552	2.81	4,450,000	4.87	17,590,000	5.15

Note: For a specific year, the system occupying the TOP500 #1 rank may be different from the system listed as the Gordon Bell Prize winner.
Source: This table was compiled from a number of sources including ACM and IEEE documents and including conversations with the following scientists: Jack Dongarra, David Keyes, Alan Karp, John Gustafson, and Bill Gropp.

1.3.2 HPC Challenge

The HPC Challenge benchmark (cf. Ch. 2.1) was initially designed in 2005 to provide a more diverse set of kernels than those provided by TOP500 HPL [PWDC08]. In addition to kernels with high locality, such as matrix multiply or STREAM TRIAD, the HPC Challenge benchmark suite added several benchmarks to emphasize data movement in the memory subsystem and in the interconnection network: Random Access, FFT, and a global transpose operation. In addition, HPCC was created to enable comparisons of different programming approaches beyond MPI and FORTRAN or C. Since the HPCC kernels are relatively small when compared to real applications, it is easier for scientists to recode and optimize these kernels on new architectures. In fact, for the past six years, the HPCC organizing group has sponsored an annual competition at the ACM/IEEE SC conference to evaluate HPCC results in terms of productivity and performance. Judging the performance prizes is relatively straightforward as long as the submitters stay within the benchmark guidelines. On the other hand, a committee judges the *productivity* of the submitted approaches for the most "elegant" implementation. Given that productivity is subjective and often impractical to measure, the committee often must rely on brief descriptions of the proposed approaches and results to determine the winners. Nevertheless, HPCC has received a substantial number of submissions over the years that provides for an interesting debate in the community. These submissions have included Chapel, Cilk, Co-array FORTRAN, Parallel MATLAB®, Star-P, UPC, X10, XcalableMP, and others.

1.3.3 Green500

In 2007, the Green500 benchmark and list (cf. Ch. 3.1) were created in order to recognize the growing importance of energy efficiency in HPC. The list requires submitters to submit both the performance of their system on the TOP500 HPL benchmark *and* the empirically measured power consumption during the benchmark test. Using this information, Green500 calculates and ranks each system by their megaFLOPS/Watt metric. All recent HPC reports predict that energy efficiency will continue to drive the design of HPC systems in the foreseeable future [KBB+08, FPT+07, HTD11], so the Green500 list will allow the community to continue to track progress on this important topic.

1.3.4 SHOC

Most recently, heterogeneous systems have become a viable commodity option for providing high performance in this new era of limited power and facility budgets. During this time, several new programming models (e.g., CUDA, OpenCL, OpenACC) and architectural features, such as accelerators attached via PCIe, have emerged that have made it difficult to use existing benchmarks effectively. The Scalable Heterogeneous Computing benchmark suite (cf. Ch. 7.8) was created to facilitate benchmarking of scalable heterogeneous clusters for computational and data-intensive computing. In contrast to most existing GPU and consumer benchmarks, SHOC focuses on scalability so that it can run on 1 or 1000s of nodes, and it focuses on scientific kernels prioritized by their importance in existing applications.

1.4 Trends

Looking at the contributions in this book and at industry more broadly, it is clear that dominant trends have emerged over the past 15 years that have directly impacted contemporary HPC. These trends span hardware, software, and business models.

For example, Linux was used only as a research operating system in HPC in the 1990s, while now it is the operating system running on nearly all HPC systems. In another example, in the late 1990s, MPI (Message Passing Interface) was just emerging as a new de facto standard that is now ubiquitous. Meanwhile, other trends including multicore processors and graphics processors were not even imagined outside of a few scientists in research communities. Finally, perhaps most important of all, open-source software has grown to be a very strong component of HPC, even resulting in international planning exercises for the path toward Exascale [KBB+08, FPT+07, HTD11]. In the following sections, we examine these trends in more detail.

1.4.1 Architectures

Recent architectures for HPC can be categorized into a few classes. First, commodity-based clusters dominate the TOP500 list. These clusters typically have an x86 commodity processor from Intel or AMD, and a commodity-based interconnect, which today is InfiniBand. These clusters offer significant capability at a very competitive price because they are high volume products for vendors. Standard HPC software stacks, much of it open-source, make these clusters easy to install, run, and maintain.

Second, GPU-accelerated commodity-based clusters have quickly emerged over the past three years as viable solutions for HPC applications [OLG+05b]. Two important aspects of these systems often go understated. First, because the GPU leverages multiple markets such as gaming and professional graphics, these GPGPU architectures are *commodity* solutions. Second, the very quick adoption of CUDA and OpenCL for programming these architectures lowered the switching costs for users to port their applications. Recently, the move toward directive-based compilation, with tools like PGI Accelerate, CAPS HMPP, and OpenACC, demonstrates even more support and interest for easing this transition.

Third, customized architectures represent a significant fraction of the top systems. Take, for example, the K Computer [ASS09] and the Blue Gene Q systems [bgp]. These systems have customized logic for both their compute nodes and interconnection networks. They have demonstrated excellent scalability, performance, and energy efficiency.

Finally, even more specialized systems, such as DE Shaw's Anton [SDD+07], have been designed that show excellent performance on specialized problems like protein folding but are likewise inflexible such that they cannot run any of the aforementioned benchmarks like TOP500 or HPCC.

1.4.2 Software

Although HPC systems share many hardware components with servers in enterprise and data centers, the HPC software stack is dramatically different from an enterprise or cloud software stack and is unique to HPC. Generally speaking, an HPC software stack has multiple levels: system software, development environments, system management software, and scientific data management and visualization systems. Nearest to the hardware, system software typically includes operating systems, runtime systems, and low level I/O software, like filesystems. Next, development environment is a broad area that facilitates application design and development. In our framework, it includes programming models, compilers, scientific frameworks and libraries, and correctness and performance tools. Then, system management software coordinates, schedules, and monitors the system and the applications running on that system. Finally, scientific data management and visualization software provides users with domain specific tools for generating, managing, and exploring data for their science. This data may include empirically measured data from sensors in the real world that is used to calibrate and validate simulation models, or output from simulations

TABLE 1.3: HPC software summary.

Category	Item
Operating Systems	Linux (Multiple versions), CNK
Languages	C, C++, FORTRAN
Compilers	CAPS, Cray, GNU, IBM, Intel, Pathscale, PGI
Scripting Languages	Java, Perl Python, Ruby, Tcl/Tk
Distributed Memory Programming Models	Charm++, Co-Array Fortran, Global Arrays, Hadoop/MapReduce MPC, MPI (OpenMPI, MVAIPICH, Intel MPI, Cray MPI, MPICH), MPT, SHMEM, Unified Parallel C, XMP
Shared Memory Programming Models	OpenMP, Pthreads, TBB
Heterogeneous Programming Models	CAPS HMPP, CUDA, OpenACC, OpenCL, PGI Accelerate,
Performance Tools	BPMON, Cray CPMAT, Extrae/Paraver, HPCToolkit, HWLOC, IHPCT, IPM, Intel Trace Analyzer, MPIP, MPInside, NVIDIA Visual Profiler, Ocelot, oprofile, PAPI, PDToolkit, PerfSuite, SCALASCA, TAU, VampirTrace/Vampir, Vtune
Correctness Tools	DDT, GNU GDB, STAT, Threadchecker, Threadspotter, Totalview, Valgrind
Scientific Libraries	ACML, ARPACK, BLAS, Boost, CASE, CRAFFT, cuBLAS, cuFFT, cuLA, cuRAND, cuSP, cuSPARSE, ESSL, FFTW, GNU GSL, Gridgen, hypre, LAPACK, MAGMA, MASS, MKL, MUMPS, PARPACK, ParMetis, SPRNG, SUNDIALS, ScaLAPACK, Scotch, SuperLU, Thrust
Scientific Frameworks	Arcane, GraphLab, JASMIN, PETSc, Trilinos
Parallel Filesystems and Storage	GPFS, GridFtp, HPSS, Lustre, Panasas, StorNext
Job Schedulers and Resource Managers	ALPS, GangliaMole, LoadLeveler, Moab, PBSPro, SLURM, Sun Grid Engine, Torque
System Management	ACE, ClusterShell, Ganglia, Inca, NFS-Ganesh, NHC, Netlogger, NodeKARE, Robinhood, Rocks, SEC, Shine, TEAL, THRMS, xCAT
I/O Libraries and Software	HDF5, Hercule, pnetCDF
Visualization	AVS/Express, EnSight, FieldView, Grace, IDL, POV-Ray, ParaView, Tecplot360, VTK, VisIt
Integrated Development Environments	Eclipse+PTP
Integrated Problem Solving Environnments	MATLAB, Octave, R
Virtualization	Eucalyptus, HPUC, Shadowfax, vSMP, Xen

per se. As Table 1.3 shows, the systems described in this book have a tremendous amount of common software, even though some of the systems are very diverse in terms of hardware. Moreover, a considerable amount of this software is open-source, and is funded by a wide array of sponsors.

Over the past 15 years, HPC software has had to adapt and respond to several challenges. First, the concurrency in applications and systems has grown over three orders of magnitude. The primary programming model, MPI, has had to grow and change to allow this scale. Second, the increase in concurrency has on a per core basis driven lower the memory and I/O capacity, and the memory, I/O, and interconnect bandwidth. Third, in the last five years, heterogeneity and architectural diversity have placed a new emphasis on application and software portability.

1.4.3 Clouds and Grids in HPC

Outside of HPC, in the data center and enterprise markets, Clouds and Grids continue to be increasingly popular and important. Both externally visible clouds, like Amazon's

EC2, and internal corporate clouds continue to grow dramatically. IDC indicates that total worldwide revenue from public IT cloud services exceeded $21.5 billion in 2010 (http://www.idc.com/prodserv/idc_cloud.jsp), and they predict that it will reach $72.9 billion in 2015. With this tremendous growth rate – (CAGR) of 27.6% – Clouds and Grids will most likely influence the HPC marketplace, even if indirectly, so we include three chapters on Cloud and Grid systems being used and tested for scientific computing markets. These chapters highlight both the strengths and weaknesses of existing cloud and grid systems.

CHAPTER 1

Introduction to Computational Modeling

1.1 THE IMPORTANCE OF COMPUTATIONAL SCIENCE

Advances in science and engineering have come traditionally from the application of the scientific method using theory and experimentation to pose and test our ideas about the nature of our world from multiple perspectives. Through experimentation and observation, scientists develop theories that are then tested with additional experimentation. The cause and effect relationships associated with those discoveries can then be represented by mathematical expressions that approximate the behavior of the system being studied.

With the rapid development of computers, scientists and engineers translated those mathematical expressions into computer codes that allowed them to imitate the operation of the system over time. This process is called simulation. Early computers did not have the capability of solving many of the complex system simulations of interest to scientists and engineers. This led to the development of supercomputers, computers with higher level capacity for computation compared to the general-purpose computers of the time. In 1982, a panel of scientists provided a report to the U.S. Department of Defense and the National Science Foundation urging the government to aid in the development of supercomputers (Lax, 1982). They indicated that "the primacy of the U.S. in science, engineering, and computing technology could be threatened relative to that of other countries with national efforts in supercomputer access

and development." They recommended both investments in research and development and in the training of personnel in science and engineering computing.

The capability of the computer chips in your cell phone today far exceeds that of the supercomputers of the 1980s. The Cray-1 supercomputer released in 1975 had a raw computing power of 80 million floating-point operations per second (FLOPS). The iPhone 5s has a graphics processor capable of 76.8 Gigaflops, nearly one thousand times more powerful (Nick, 2014). With that growth in capability, there has been a dramatic expansion in the use of simulation for engineering design and research in science, engineering, social science, and the humanities. Over the years, that has led to many efforts to integrate computational science into the curriculum, to calls for development of a workforce prepared to apply computing to both academic and commercial pursuits, and to investments in the computer and networking infrastructure required to meet the demands of those applications. For example, in 2001 the Society for Industrial and Applied Mathematics (SIAM) provided a review of the graduate education programs in science and engineering (SIAM, 2001). They defined computational science and engineering as a multidisciplinary field requiring expertise in computer science, applied mathematics, and a subject field of science and engineering. They provided examples of emerging research, an outline of a curriculum, and curriculum examples from both North America and Europe.

Yasar and Landau (2001) provided a similar overview of the interdisciplinary nature of the field. They also describe the possible scope of programs at the both the undergraduate and graduate levels and provide a survey of existing programs and their content. More recently, Gordon et al. (2008) described the creation of a competency-based undergraduate minor program in computational science that was put into place at several institutions in Ohio. The competencies were developed by an interdisciplinary group of faculty and reviewed by an industry advisory committee from the perspective of the skills that prospective employers are looking for in students entering the job market. The competencies have guided the creation of several other undergraduate programs. They have also been updated and augmented with graduate-level computational science competencies and competencies for data-driven science. The most recent version of those competencies can be found on the HPC University website (HPC University, 2016).

More recently, there have been a number of national studies and panels emphasizing the need for the infrastructure and workforce

required to undertake large-scale modeling and simulation (Council on Competitiveness, 2004; Joseph et al., 2004; Reed, 2005; SBES, 2006). This book provides an introduction to computational science relevant to students across the spectrum of science and engineering. In this chapter, we begin with a brief review of the history or computational modeling and its contributions to the advancement of science. We then provide an overview of the modeling process and the terminology associated with modeling and simulation.

As we progress through the book, we guide students through basic programming principles using two of the widely used simulation environments—MATLAB® and Python. Each chapter introduces either a new set of programming principles or applies them to the solution of one class of models. Each chapter is accompanied by exercises that help to build both basic modeling and programming skills that will provide a background for more advanced modeling courses.

1.2 HOW MODELING HAS CONTRIBUTED TO ADVANCES IN SCIENCE AND ENGINEERING

There are a myriad of examples documenting how modeling and simulation has contributed to research and to the design and manufacture of new products. Here, we trace the history of computation and modeling to illustrate how the combination of advances in computing hardware, software, and scientific knowledge has led to the integration of computational modeling techniques throughout the sciences and engineering. We then provide a few, more recent examples of advances to further illustrate the state-of-the-art. One exercise at the end of the chapter provides an opportunity for students to examine additional examples and share them with their classmates.

The first electronic programmable computer was the ENIAC built for the army toward the end of World War II as a way to quickly calculate artillery trajectories. Herman Goldstine (1990), the project leader, and two professors from the University of Pennsylvania, J. Presper Eckert, and John Mauchly sold the idea to the army in 1942 (McCartney, 1999). As the machine was being built and tested, a large team of engineers and mathematicians was assembled to learn how to use it. That included six women mathematicians who were recruited from colleges across the country. As the machine was completed in 1945, the war was near an end.

ENIAC was used extensively by the mathematician John von Neumann not only to undertake its original purposes for the army but also to create the first weather model in 1950. That machine was capable of 400 floating-point operations per second and needed 24 hours to calculate the simple

daily weather model for North America. To provide a contrast to the power of current processors, Peter and Owen Lynch (2008) created a version of the model that ran on a Nokia 6300 mobile phone in less than one second!

It is impossible to document all of the changes in computational power and its relationship to the advancements in science that have occurred since this first computer. Tables 1.1 and 1.2 show a timeline

TABLE 1.1 Timeline of Advances in Computer Power and Scientific Modeling (Part 1)

Example Hardware	Max. Speed	Date	Weather and Climate Modeling
ENIAC	400 Flops	1945	
		1950	First automatic weather forecasts
UNIVAC		1951	
IBM 704	12 KFLOP	1956	
		1959	Ed Lorenz discovers the chaotic behavior of meteorological processes
IBM7030 Stretch; UNIVAC LARC	500-500 KFLOP	~1960	
		1965	Global climate modeling underway
CDC6600	1 Megaflop	1966	
CDC7600	10 MFLOP	1975	
CRAY1	100 MFLOP	1976	
CRAY-X-MP	400 MFLOP		
		1979	Jule Charney report to NAS
CRAY Y-MP	2.67 GFLOP		
		1988	Intergovernmental Panel on Climate Change
		1992	UNFCCC in Rio
IBM SP2	10 Gigaflop	1994	
ASCII Red	2.15 TFLOP	1995	Coupled Model Intercomparison Project (CMIP)
		2005	Earth system models
Blue Waters	13.34 PFLOP	2014	

Sources: Bell, G., Supercomputers: The amazing race (a history of supercomputing, 1960–2020), 2015, http://research.microsoft.com/en-us/um/people/gbell/MSR-TR-2015-2_Supercomputers-The_Amazing_Race_Bell.pdf (accessed December 15, 2016).
Bell, T., Supercomputer timeline, 2016, https://mason.gmu.edu/~tbell5/page2.html (accessed December 15, 2016).
Esterbrook, S., Timeline of climate modeling, 2015, https://prezi.com/pakaaiek3nol/timeline-of-climate-modeling/ (accessed December 15, 2016).

TABLE 1.2 Timeline of Advances in Computer Power and Scientific Modeling (Part 2)

Date	Theoretical Chemistry	Aeronautics and Structures	Software and Algorithms
1950	Electronic wave functions		
1951	Molecular orbital theory (Roothan)		
1953	One of the first molecular simulations (Metropolis et al.)		
1954			Vector processing directives
1956	First calculation of multiple electronic states of a molecule on EDSAC (Boys)		
1957			FORTRAN created
1965	Creation of ab initio molecular modeling (People)		
1966		2D Navier-Stokes simulations; FLO22; transonic flow over a swept wing	
1969			UNIX created
1970		2D Inviscid Flow Models; design of regional jet	
1971		Nastran (NASA Structural Analysis)	
1972			C programming language created
1973			Matrix computations and errors (Wilkinson)
1975		3D Inviscid Flow Models; complete airplane solution	
1976	First calculation of a chemical reaction (Warshel)	DYNA3D which became LS-DYNA (mid-70s)	
1977	First molecular dynamics of proteins (Karplus) First calculation of a reaction transition state (Chandler)	Boeing design of 737-500	

(*Continued*)

TABLE 1.2 (*Continued*) Timeline of Advances in Computer Power and Scientific Modeling (Part 2)

Date	Theoretical Chemistry	Aeronautics and Structures	Software and Algorithms
1979			Basic Linear Algebra Subprograms (BLAS) library launched
1980s	*Journal of Computational Chemistry* first published	800,000 mesh cells around a wing, FLO107	
1984			MATLAB created
1985		Design of Boeing 767,777	GNU project launched (free Software foundation)
1991			Linux launched
1993			Message passing interface (MPI) specification
1994			Python created
1995	First successful computer-based drug design (Kubinyi)		
1997			Open multiprocessing (OpenMP) specification
2000		Discontinuous finite element methods; turbulent flow; design of airbus	
2007			CUDA launched
2014			Open accelerator (OpenACC) specification

Sources: Bartlett, B.N., The contributions of J.H. Wilkinson to numerical analysis. In S.G. Nash, (Ed.), *A History of Scientific Computing*, ACM Press, New York, pp. 17–30, 1990.
Computer History Museum, Timeline of computer history, software and languages, 2017, http://www.computerhistory.org/timeline/software-languages/ (accessed January 2, 2017).
Dorzolamide, 2016, https://en.wikipedia.org/wiki/Dorzolamide (accessed December 15, 2016).
Jameson, A., Computational fluid dynamics, past, present, and future, 2016, http://aero-comlab.stanford.edu/Papers/NASA_Presentation_20121030.pdf (accessed December 15, 2016).
Prat-Resina, X., A brief history of theoretical chemistry, 2016, https://sites.google.com/a/r.umn.edu/prat-resina/divertimenti/a-brief-history-of-theoretical-chemistry (accessed December 15, 2016).
Vassberg, J.C., A brief history of FLO22, http://dept.ku.edu/~cfdku/JRV/Vassberg.pdf (accessed December 15, 2016).

of the development of selected major hardware advances, software and algorithm development, and scientific applications from a few fields. Looking at the first column in Table 1.1, one can see the tremendous growth in the power of the computers used in large-scale scientific computation. Advances in electronics and computer design have brought us from the ENIAC with 400 flops to Blue Waters with 13.34 petaflops, an increase in the maximum number of floating-point operations per second of more than 1015!

Tracing weather and climate modeling from von Neumann's first model on ENIAC, we can see that the computational power has allowed scientists to make rapid progress in the representation of weather and climate. In 1959, Lorenz laid the foundation for the mathematics behind weather events. By 1965, further advances in computing power and scientific knowledge provided the basis for the first global climate models. These have grown in scope to the present day to earth system models that couple atmospheric and ocean circulation that provide for the basis for the climate change forecasts of the international community.

Table 1.2 documents similar developments in computational chemistry, aeronautics and structures, and selected achievements in software and algorithms. The scientific advances were made possible not only by improvements in the hardware but also by the invention of programming languages, compilers, and the algorithms that are used to make the mathematical calculations underlying the models. As with weather modeling, one can trace the advancement of computational chemistry from the first simulation of molecules to the screening of drugs by modeling their binding to biomolecules. In aeronautics, the simulation of airflow over a wing in two dimensions has advanced to the three-dimensional simulation of a full airplane to create a final design. Similar timelines could be developed for every field of science and engineering from various aspects of physics and astronomy to earth and environmental science, to every aspect of engineering, and to economics and sociological modeling.

For those just getting introduced to these concepts, the terminology is daunting. The lesson at this point is to understand that computation has become an essential part of the design and discovery process across a wide range of scientific fields. Thus, it is essential that everyone understands the basic principles used in modeling and simulation, the mathematics underlying modeling efforts, and the tools of modeling along with their pitfalls.

1.2.1 Some Contemporary Examples

Although this book will not involve the use of large-scale models on supercomputers, some contemporary examples of large-scale simulations may provide insights into the need for the computational power described in Table 1.1. We provide four such examples.

Vogelsberger et al. created a model of galaxy formation comprised of 12 billion resolution elements showing the evolution of the universe from 12 million years after the Big Bang evolving over a period of 13.8 billion years (Vogelsberger et al., 2014). The simulation produced a large variety of galaxy shapes, luminosities, sizes, and colors that are similar to observed population. The simulation provided insights into the processes associated with galaxy formation. This example also illustrates how computation can be applied to a subject where experimentation is impossible but where simulation results can be compared with scientific observations.

Drug screening provides an example of how computer modeling can shorten the time to discovery. The drug screening pipeline requires a model of a target protein or macromolecular structure that is associated with a specific disease mechanism. A list of potential candidate compounds is then tested to see which have the highest affinity to bind to that protein, potentially inhibiting the medical problem. Biesiada et al. (2012) provide an excellent overview of the workflow associated with this process and the publically available software for accomplishing those tasks. The use of these tools allows researchers to screen thousands of compounds for their potential use as drugs. The candidate list can then be pared down to only a few compounds where expensive experimental testing is used.

The reports on global warming use comprehensive models of the earth's climate including components on the atmosphere and hydrosphere (ocean circulation and temperature, rainfall, polar ice caps) to forecast the long-term impacts on our climate and ecosystems (Pachauri and Meyer, 2014). The models:

> reproduce observed continental-scale surface temperature patterns and trends over many decades, including the more rapid warming since the mid-20th century and the cooling immediately following large volcanic eruptions (very high confidence) (IPC, 2013, p. 15).

Modeling and simulation has also become a key part of the process and designing, testing, and producing products and services. Where the building of physical prototypes or the completion of laboratory experiments

may take weeks or months and cost millions of dollars, industry is instead creating virtual experiments that can be completed in a short time at greatly reduced costs. Proctor and Gamble uses computer modeling to improve many of its products. One example is the use of molecular modeling to test the interactions among surfactants in their cleaning products with a goal of producing products that are environmentally friendly and continue to perform as desired (Council on Competiveness, 2009).

Automobile manufacturers have substituted modeling for the building of physical prototypes of their cars to save time and money. The building of physical prototypes called *mules* is expensive, costing approximately $500,000 for each vehicle with 60 prototypes required before going into production (Mayne, 2005). The design of the 2005 Toyota Avalon required no mules at all—using computer modeling to design and test the car. Similarly, all of the automobile manufacturers are using modeling to reduce costs and get new products to market faster (Mayne, 2005).

These examples should illustrate the benefits of using modeling and simulation as part of the research, development, and design processes for scientists and engineers. Of course, students new to modeling and simulation cannot be expected to effectively use complex, large-scale simulation models on supercomputers at the outset of their modeling efforts. They must first understand the basic principles for creating, testing, and using models as well as some of the approaches to approximating physical reality in computer code. We begin to define those principles in Section 1.3 and continue through subsequent chapters.

1.3 THE MODELING PROCESS

Based on the examples discussed earlier, it should be clear that a model is an abstraction or simplification of a real-world object or phenomenon that helps us gain insights into the state or behavior of a complex system. Each of us creates informal, mental models all the time as an aid to making decisions. One example may be deciding on a travel route that gets us to several shopping locations faster or with the fewest traffic headaches. To do this, we analyze information from previous trips to make an informed decision about where there may be heavy traffic, construction, or other impediments to our trip.

Some of our first formal models were physical models. Those include simplified prototypes of objects used to evaluate their characteristics and behaviors. For example, auto manufacturers built clay models of new car designs to evaluate the styling and to test the design in wind-tunnel experiments.

Mississippi Basin Model
Vertical scale - 1:100; horizontal scale - 1:2000. Looking upstream on the Ohio River from Evansville. Indiana, Tennessee, and Cumberland Rivers are in the foreground showing the site of the Kentucky and Barkley Dams. Tradewater and Green Rivers are shown center. File No. 1270–4

FIGURE 1.1 Photo of portion of Mississippi River Basin model.

One of the most ambitious physical models ever built was a costly 200 acre model of the Mississippi River Basin used to simulate flooding in the watershed (U.S. Army Corps of Engineers, 2006). A photo of a portion of this model is shown in Figure 1.1. It included replicas of urban areas, the (Fatherree, 2006) stream bed, the underlying topography, levees, and other physical characteristics. Special materials were used to allow flood simulations to be tested and instrumented.

Through theory and experimentation, scientists and engineers also developed mathematical models representing aspects of physical behaviors. These became the basis of computer models by translating the mathematics into computer codes. Over time, mathematical models that started as very simplistic representations of complex systems have evolved into systems of equations that more closely approximate real-world phenomena such as the large-scale models discussed earlier in this chapter.

Creating, testing, and applying mathematical models using computation require an iterative process. The process starts with an initial set of simplifying assumptions and is followed by testing, alteration, and application of the model. Those steps are discussed in Section 1.3.1.

1.3.1 Steps in the Modeling Process

A great deal of work must be done before one can build a mathematical model on a computer. Figure 1.2 illustrates the steps in the modeling process. The first step is to analyze the problem and define the objectives of the model. This step should include a review of the literature to uncover previous research on the topic, experimental or field-measured data showing various states of the system and the measured outcomes, mathematical representations of the system derived from theories, and previous modeling efforts.

As that information is being gathered, it is also important to define the objectives of the modeling effort. There are several questions that should be addressed while considering the model objectives: What are the outcomes that we would like the model to predict? Are we interested in every possible outcome or is there a subset of conditions that would satisfy our model objectives? For example, we could be interested in just

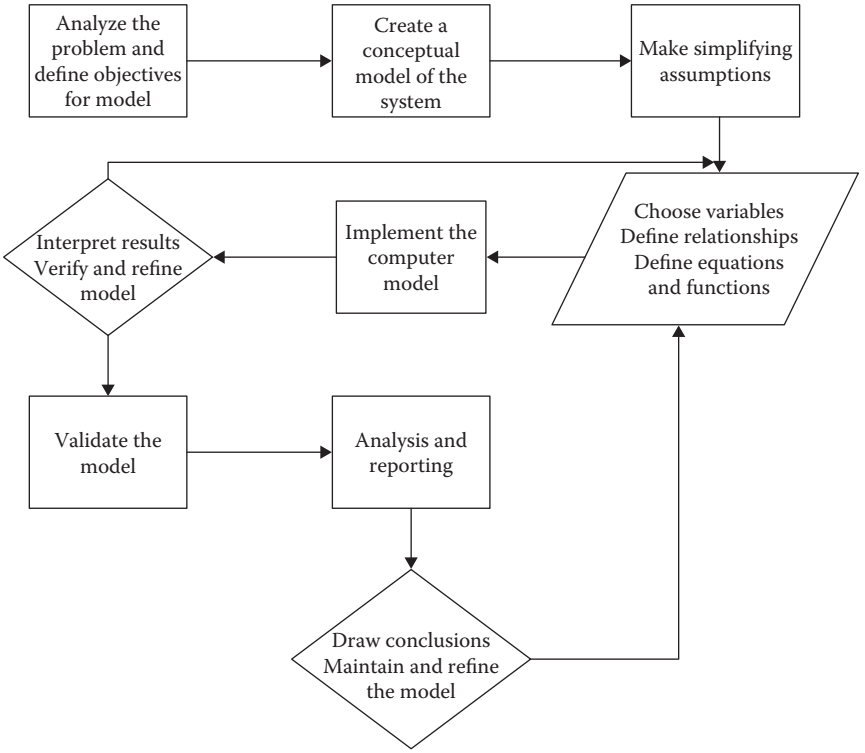


FIGURE 1.2 Major steps in the modeling process.

the *average* or *normal* state of affairs associated with a phenomenon or potential extreme events may be critical for our analysis. What level of accuracy is required for the predicted outcomes? This will impact the nature of the simplifying assumptions, input data, and computing algorithms that are required to build the model.

The second step in the process is to create a conceptual model of the system based on the analysis in the first step. A conceptual model will begin to specify all of the cause and effect relationships in the system, information on the data required and available to implement a model, and references to documents that were found in the initial analysis. The conceptual model should include a concept map showing the cause and effect relationships associated with the model and tables showing the different variables, data sources, and references. This can be done on a whiteboard, pencil and paper, or using a formal flowcharting or concept-mapping tool. There are several free tools for concept mapping. Cmap provides a free concept-mapping tool developed by the Florida Institute for Human and Machine Cognition. It creates nodes representing major components of a concept and labels the links between nodes with their relationships (Cmap, 2016). Mind Map Maker is a free mind-mapping tool provided as an app for Google Chrome users (Mindmapmaker, 2016). This tool allows one to create links between associated items. There are also a number of commercial packages in both categories.

Figures 1.3 and 1.4 are examples of a partially completed concept map and mind map showing the components of a model of the time it takes to make a car trip between two points.

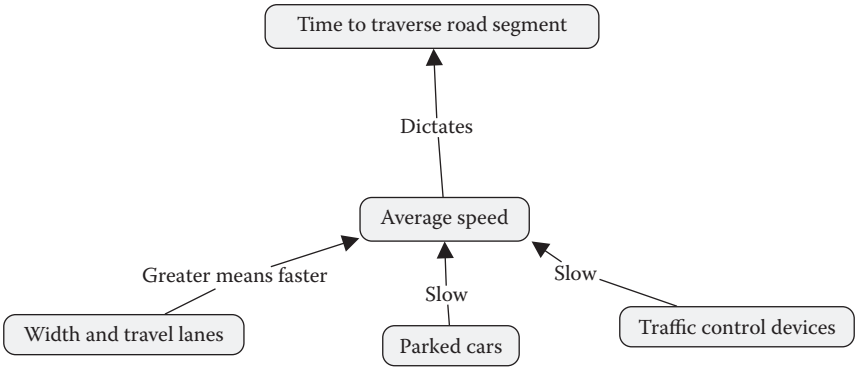


FIGURE 1.3 Partial concept map of model to calculate travel time using Cmap.

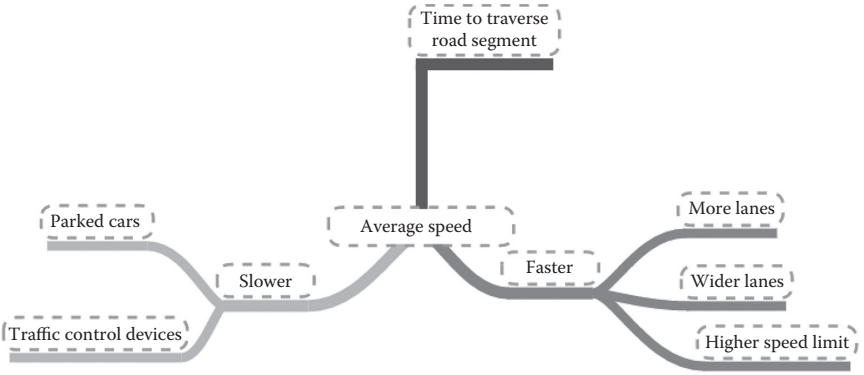


FIGURE 1.4 Partial mind map of model to calculate travel time using mind map maker.

The average speed across a road segment is slowed by parked cars and traffic control devices while wider lanes and higher speed limits take less time. The total time for a trip would need to add the average times associated with traversing each road segment. Thus, data on each segment will be needed as input to the model. Simple versions of such estimates are provided by global positioning satellite (GPS) equipment or the Internet mapping services that are available online. There are many other conditions that would impact this system. Modeling traffic conditions are a topic of one of the exercises at the end of the chapter.

Going back to Figure 1.2, one must choose which simplifying assumptions can be made in a model. This, in turn, leads to a selection of the data that would be needed, the variables that will drive the model, and the equations and mathematical functions that will comprise the model.

Once these items have been defined, a computer version of the model can be created and tested. The results must be verified to ascertain that the code is working properly. If the model is giving unexpected results with the code working properly, there may be a need to reexamine the simplifying assumptions and to reformulate the model. Thus, one may go through several iterations until the model is providing sufficiently accurate results. This can be validated against available experimental or field data to provide a quantitative assessment of model accuracy. Finally, the model can be used to undertake more detailed analysis and the results reported. As time goes on, the model must be maintained and may be improved by relaxing more of the assumptions and/or improving the input data. It should be noted that the judgment of whether a model is giving *reasonable*

results is sometimes as much an art as a science. Confidence in that judgment is a function of the experience of the modeler and the breadth and depth of the previous research about the system under study. Of course the best validation of modeling results comes from comparisons with real data gathered from observations or experiments.

1.3.2 Mathematical Modeling Terminology and Approaches to Simulation

Similar to all scientific disciplines, mathematical modeling has its own unique vocabulary. Modeling novices may believe that the language used just creates a smoke screen that hides any problems associated with a model's development and use. Unfortunately, sometimes there is truth in that belief. Nevertheless, it is important to learn that language to enable a critical understanding of the modeling literature. We will begin with some basic definitions of modeling terms in this section.

It is also important to begin to understand the variety of approaches to modeling different types of systems. We will use some of the terminology we introduce to provide a few examples of different modeling approaches to simulate a variety of situations. We will then conclude this chapter with some exercises that let you delve deeper into the world of modeling and simulation.

1.3.3 Modeling and Simulation Terminology

By now, you should have your own concept of what constitutes a mathematical or computer model. A more formal definition is provided here.

> A mathematical model is a representation of a phenomenon or system that is used to provide insights and predictions about system behavior.

> Simulation is the application of a model to imitate the behavior of the system under a variety of circumstances.

There are several different ways to classify models. Models can be deterministic or probabilistic. Another term for probabilistic is stochastic meaning a random process or a process, which occurs by chance. A probabilistic model includes one or more elements that might occur by chance or at random while a deterministic model does not. A deterministic model applies a set of inputs or initial conditions and uses one or more equations to produce

model outputs. The outputs of a deterministic model will be the same for each execution of the computer code with the same inputs. A probabilistic model will exhibit random effects that will produce different outputs for each model run.

Models can also be characterized as static or dynamic. A dynamic model considers the state of a system over time while a static model does not. For example, one could have a model of a material like a steel beam that considered its ability to bear weight without bending under a set of standard environmental conditions. This would be considered to be a static model of that system. A dynamic model of the same structure would simulate how the bearing strength and possible deformation of the beam would change under stresses over time such as under high temperatures, vibration, and chemical corrosion.

> A steady-state model is a model that has gone through a transient state such as a start-up or warm-up period and arrived at an observed behavior that remains constant.

An example of the steady-state model is the flow of fluid through a pipe. In the initial, transient state period, the pipe is empty and will fill with fluid under pressure until the capacity of the pipe is reached. This will be its steady-state condition. In economics, a steady-state economy is one that has reached a relatively stable size.

Perhaps making things more confusing, a dynamic model can have deterministic components. Such a model would track the state of a system over time and/or space. Given a current state, a deterministic function may be used to predict the future state of the system. Alternatively, the future state may be stochastic, which is impacted by random events.

Finally, dynamic models may be characterized as being discrete or continuous. A continuous model would represent time as a continuous function, whereas a discrete model divides time into small increments and calculates its state for each time period. In computer modeling, most (all?) dynamic models divide time into discrete increments to facilitate rapid calculations that mimic continuous systems.

1.3.4 Example Applications of Modeling and Simulation

In order to gain insights into system behavior, simulations are used to ask *what if* questions about how the system changes under different circumstances. How these questions are addressed depends in part on the type

of model and its underlying mathematical structure. Solving those mathematical equations on a computer also leads to differences in programming logic or the algorithms that are used to calculate the most accurate answer most efficiently. We will discuss some of those algorithms as we go through the rest of this book. For now, it may help to provide some examples of different simulation approaches as they relate to various model types.

Deterministic models consist of one or more equations that characterize the behavior of a system. Most such models simplify the system by assuming that one or more causal variables or parameters are constant for a single calculation of the model outcomes.

For example, models of people's car trip behavior assume that the willingness to make a trip is inversely proportional to the trip distance. That is, people are more likely to make a trip from home to get to a destination that is closer than the one that is far away. Empirical studies have shown that this friction of distance changes depending on the nature of the trip. People are much more willing to make a longer trip to get to work than they are to do a convenience shopping trip. To simplify the system, these models assume a constant value of this friction of distance factor for each type of trip. When such a model is applied to a new urban area, there is some uncertainty that the constants found in previous studies in different places match the area where the model is being applied. Thus, a study is done where the model is run with different but reasonable variations in the constants to ascertain the impact of those changes on the predicted trips. Those can then be compared with a sample of real data to calibrate and validate the model.

Other examples of parametric studies include models of structures where different environmental conditions will alter system behavior, air and water pollution models where assumptions are made about the rate of dispersion of contaminants, and models of drug absorption into the blood stream where assumptions are made about absorption rates and excretion rates of the drug within the body. Many models include components that are both stochastic and deterministic where parametric studies are done on the deterministic components.

For dynamic models, the focus is on the behavior of the system over time and sometimes over space. For one group of such models called systems dynamics models, the state of the system at any time period

is dependent, in part, on the state of the system at the previous time period. Simulations calculate the changes in the state of the system over time. An example is a model of ball being dropped from a bridge. As it is dropped the ball accelerates due to the force of gravity. At each time increment, the model will calculate the velocity of the ball and its position in space. That position will depend on where it was in the previous time period and how far it was dropped related to its velocity during that time period. The model will then predict when the ball will hit the water and at what velocity.

Stochastic models typically will have characteristics in common with dynamic models. The difference is that one or more of the governing parameters are probabilistic or could happen by random chance. One example is a model of the spread of a disease that is passed by human contact. A susceptible person may make contact with an infected person but will not necessarily become infected. There is a probability of being infected that is related to the virility of the disease, the state of health of the susceptible person, and the nature of the contact. A model of this system would simulate those probabilities to project the potential spread of a disease outbreak.

As we go through the rest of this book, we will describe the mathematical representation of each of these types of models and the programming steps needed to implement them on the computer. Exercises will involve the completion of example programs, the use of the model to make predictions, the analysis of model outcomes, and, in some cases, validation of model results. The exercises for this chapter focus on the modeling process and examples of how models have been used to solve research and production problems.

EXERCISES

1. Using a graphics program or one of the free concept-mapping or mind-mapping tools, create a complete conceptual map of the traffic model introduced earlier in the chapter. You should include all of the other factors you can think of that would contribute either to the increase or decrease in the traffic speed that might occur in a real situation.

2. Insert another concept mapping example here.

3. Read the executive summary of one of the following reports and be prepared to discuss it in class:
   a. PITAC report to the president
   b. Simulation-based engineering science report
   c. World Technology Evaluations Center
4. Using the student website for the book at http://www.intromodeling.com, choose an example model project in the document *example models for discovery and design* as assigned by your instructor. Read through the available material and then write a brief summary of the modeling effort and its characteristics using the summary template provided.

REFERENCES

Bartlett, B. N. 1990. The contributions of J.H. Wilkinson to numerical analysis. In *A History of Scientific Computing*, ed. S. G. Nash, pp. 17–30. New York: ACM Press.

Bell, G. 2015. Supercomputers: The amazing race. (A History of Supercomputing, 1960–2020). http://research.microsoft.com/en-us/um/people/gbell/MSR-TR-2015-2_Supercomputers-The_Amazing_Race_Bell.pdf (accessed December 15, 2016).

Bell, T. 2016. Supercomputer timeline, 2016. https://mason.gmu.edu/~tbell5/page2.html (accessed December 15, 2016).

Biesiada, J., A. Porollo, and J. Meller. 2012. On setting up and assessing docking simulations for virtual screening. In *Rational Drug Design: Methods and Protocols, Methods in Molecular Biology*, ed. Yi Zheng, pp. 1–16. New York: Springer Science and Business Media.

Cmap website. http://cmap.ihmc.us/ (accessed February 22, 2016).

Computer History Museum. 2017. Timeline of computer history, software and languages. http://www.computerhistory.org/timeline/software-languages/ (accessed January 2, 2017).

Council on Competitiveness. 2004. First Annual High Performance Computing Users Conference. http://www.compete.org/storage/images/uploads/File/PDF%20Files/2004%20HPC%2004%20Users%20Conference%20Final.pdf.

Council on Competitiveness. 2009. Procter & gamble's story of suds, soaps, simulations and supercomputers. http://www.compete.org/publications/all/1279 (accessed January 2, 2017).

Dorzolamide. 2016. https://en.wikipedia.org/wiki/Dorzolamide (accessed December 15, 2016).

Esterbrook, S. 2015. Timeline of climate modeling. https://prezi.com/pakaaiek3nol/timeline-of-climate-modeling/ (accessed December 15, 2016).

Fatherree, B. H. 2006. U.S. Army corps of engineers, Chapter 5 hydraulics research giant, 1949–1963, Part I: River modeling, potamology, and hydraulic structures. In *The First 75 Years: History of Hydraulics Engineering at the Waterways Experiment Station*, http://chl.erdc.usace.army.mil/Media/8/5/5/Chap5.htm. Vicksburg, MS: U.S. Army Engineer Research and Development Center (accessed October 15, 2016).

Goldstine, H. 1990. Remembrance of things past. In *A History of Scientific Computing*, ed. S. G. Nash, pp. 5–16. New York: ACM Press.

Gordon, S. I., K. Carey, and I. Vakalis. 2008. A shared, interinstitutional undergraduate minor program in computational science. *Computing in Science and Engineering*, 10(5): 12–16.

HPC University website. http://hpcuniversity.org/educators/competencies/ (accessed January 15, 2016).

International Panel on Climate Change. 2013. Climate change 2013—The physical science basis contribution of working Group I to the fifth assessment report of the IPCC, New York: Cambridge University Press. http://www.climatechange2013.org/images/report/WG1AR5_ALL_FINAL.pdf (accessed December 15, 2016).

Jameson, A. 2016. Computational fluid dynamics, past, present, and future. http://aero-comlab.stanford.edu/Papers/NASA_Presentation_20121030.pdf (accessed December 15, 2016).

Joseph, E., A. Snell, and C. Willard. 2004. Study of U.S. industrial HPC users. Council on Competitiveness. http://www.compete.org/publications/all/394 (accessed December 15, 2016).

Lax, P. D. 1982. Report of the panel on large scale computing in science and engineering. Report prepared under the sponsorship of the Department of Defense and the National Science Foundation. Washington, D.C.: National Science Foundation.

Lynch, P. and O. Lynch. 2008. Forecasts by PHONIAC. *Weather*, 63(11): 324–326.

Mayne, E. Automakers trade mules for computers, Detroit News, January 30, 2005, http://www.jamaicans.com/forums/showthread.php?1877-Automakers-Trade-Mules-For-Computers (accessed January 25, 2016).

McCartney, S. 1999. *ENIAC*. New York: Walker and Company.

Mindmapmaker website. http://mindmapmaker.org/ (accessed February 22, 2016).

National Science Foundation. 2006. Simulation-based engineering science: Report of the NSF blue ribbon panel on simulation-based engineering science. http://www.nsf.gov/pubs/reports/sbes_final_report.pdf.

Nick, T. 2014. A modern smartphone or a vintage supercomputer: Which is more powerful? http://www.phonearena.com/news/A-modern-smartphone-or-a-vintage-supercomputer-which-is-more-powerful_id57149 (accessed January 15, 2016).

Pachauri, R. K. and L. A. Meyer (ed.). 2014. Climate change 2014: Synthesis report. Contribution of working groups I, II and III to the fifth assessment report of the intergovernmental panel on climate change. http://www.ipcc.ch/report/ar5/syr/.

Prat-Resina, X. 2016. A brief history of theoretical chemistry. https://sites.google.com/a/r.umn.edu/prat-resina/divertimenti/a-brief-history-of-theoretical-chemistry (accessed December 15, 2016).

Reed, D. 2005. Computational science: America's competitive challenge. President's information technology advisory committee subcommittee on computational science. http://www.itrd.gov/pitac/meetings/2005/20050414/20050414_reed.pdf.

Society of Industrial and Applied Mathematics (SIAM). 2001. Graduate education in computational science and engineering. *SIAM Review*, 43(1): 163–177.

Vassberg, J. C. A brief history of FLO22. http://dept.ku.edu/~cfdku/JRV/Vassberg.pdf (accessed December 15, 2016).

Vogelsberger, M., S. Genel, V. Springel, et al. 2014. Properties of galaxies reproduced by a hydrodynamic simulation. *Nature*, 509(8): 177–182. doi:10.1038/nature13316.

Yasar, O. and R. H. Landau. 2001. Elements of computational science and engineering education. *SIAM Review*, 45(4): 787–805.