1

Heterogeneous Parallel Computing with CUDA

WHAT'S IN THIS CHAPTER?

- Understanding heterogeneous computing architectures
- Recognizing the paradigm shift of parallel programming
- Grasping the basic elements of GPU programming
- Knowing the differences between CPU and GPU programming

CODE DOWNLOAD *The wrox.com code downloads for this chapter are found at* `www.wrox.com/go/procudac` *on the Download Code tab. The code is in the Chapter 1 download and individually named according to the names throughout the chapter.*

The *high-performance computing* (HPC) landscape is always changing as new technologies and processes become commonplace, and the definition of HPC changes accordingly. In general, it pertains to the use of multiple processors or computers to accomplish a complex task concurrently with high throughput and efficiency. It is common to consider HPC as not only a computing architecture but also as a set of elements, including hardware systems, software tools, programming platforms, and parallel programming paradigms.

Over the last decade, high-performance computing has evolved significantly, particularly because of the emergence of GPU-CPU heterogeneous architectures, which have led to a fundamental paradigm shift in parallel programming. This chapter begins your understanding of heterogeneous parallel programming.

PARALLEL COMPUTING

During the past several decades, there has been ever-increasing interest in parallel computation. The primary goal of parallel computing is to improve the speed of computation.

From a pure calculation perspective, *parallel computing* can be defined as a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently.

From the programmer's perspective, a natural question is how to map the concurrent calculations onto computers. Suppose you have multiple computing resources. Parallel computing can then be defined as the simultaneous use of multiple computing resources (cores or computers) to perform the concurrent calculations. A large problem is broken down into smaller ones, and each smaller one is then solved concurrently on different computing resources. The software and hardware aspects of parallel computing are closely intertwined together. In fact, parallel computing usually involves two distinct areas of computing technologies:

- Computer architecture (hardware aspect)
- Parallel programming (software aspect)

Computer architecture focuses on supporting parallelism at an architectural level, while *parallel programming* focuses on solving a problem concurrently by fully using the computational power of the computer architecture. In order to achieve parallel execution in software, the hardware must provide a platform that supports concurrent execution of multiple processes or multiple threads.

Most modern processors implement the *Harvard architecture,* as shown in Figure 1-1, which is comprised of three main components:

- Memory (instruction memory and data memory)
- Central processing unit (control unit and arithmetic logic unit)
- Input/Output interfaces

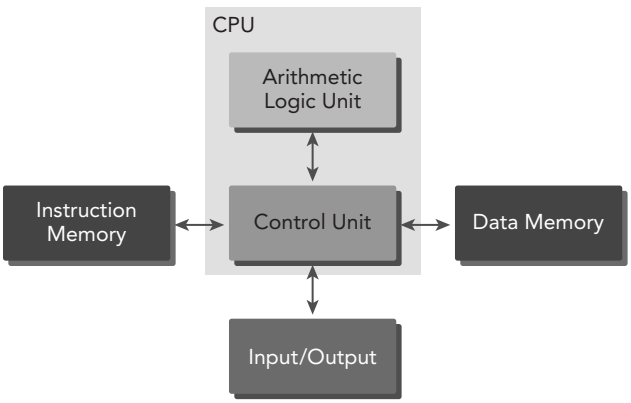


FIGURE 1-1

The key component in high-performance computing is the *central processing unit* (CPU), usually called the *core*. In the early days of the computer, there was only one core on a chip. This architecture is referred to as a *uniprocessor*. Nowadays, the trend in chip design is to integrate multiple cores onto a single processor, usually termed *multicore*, to support parallelism at the architecture level. Therefore, programming can be viewed as the process of mapping the computation of a problem to available cores such that parallel execution is obtained.

When implementing a sequential algorithm, you may not need to understand the details of the computer architecture to write a correct program. However, when implementing algorithms for multicore machines, it is much more important for programmers to be aware of the characteristics of the underlying computer architecture. Writing both correct and efficient parallel programs requires a fundamental knowledge of multicore architectures.

The following sections cover some basic concepts of parallel computing and how these concepts relate to CUDA programming.

Sequential and Parallel Programming

When solving a problem with a computer program, it is natural to divide the problem into a discrete series of calculations; each calculation performs a specified task, as shown in Figure 1-2. Such a program is called a *sequential program*.

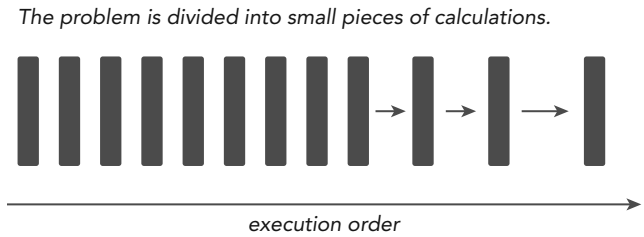


FIGURE 1-2

There are two ways to classify the relationship between two pieces of computation: Some are related by a precedence restraint and therefore must be calculated sequentially; others have no such restraints and can be calculated concurrently. Any program containing tasks that are performed concurrently is a *parallel program*. As shown in Figure 1-3, a parallel program may, and most likely will, have some sequential parts.

From the eye of a programmer, a program consists of two basic ingredients: instruction and data. When a computational problem is broken down into many small pieces of computation, each piece is called a *task*. In a task, individual instructions consume inputs, apply a function, and produce outputs. A *data dependency* occurs when an instruction consumes data produced by a preceding instruction. Therefore, you can classify the relationship between any two tasks as either dependent, if one consumes the output of another, or independent.

Analyzing data dependencies is a fundamental skill in implementing parallel algorithms because dependencies are one of the primary inhibitors to parallelism, and understanding them is necessary

to obtain application speedup in the modern programming world. In most cases, multiple independent chains of dependent tasks offer the best opportunity for parallelization.

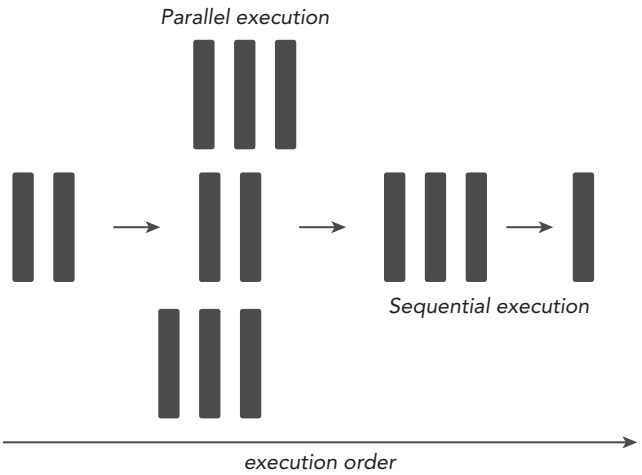


FIGURE 1-3

Parallelism

Nowadays, parallelism is becoming ubiquitous, and parallel programming is becoming mainstream in the programming world. Parallelism at multiple levels is the driving force of architecture design. There are two fundamental types of parallelism in applications:

- Task parallelism
- Data parallelism

Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel. Task parallelism focuses on distributing functions across multiple cores.

Data parallelism arises when there are many data items that can be operated on at the same time. Data parallelism focuses on distributing the data across multiple cores.

CUDA programming is especially well-suited to address problems that can be expressed as data-parallel computations. The major focus of this book is how to solve a data-parallel problem with CUDA programming. Many applications that process large data sets can use a data-parallel model to speed up the computations. Data-parallel processing maps data elements to parallel threads.

The first step in designing a data parallel program is to partition data across threads, with each thread working on a portion of the data. In general, there are two approaches to partitioning data: block partitioning and cyclic partitioning. In block partitioning, many consecutive elements of data are chunked together. Each chunk is assigned to a single thread in any order, and threads generally process only one chunk at a time. In cyclic partitioning, fewer data elements are chunked together. Neighboring threads receive neighboring chunks, and each thread can handle more than one chunk. Selecting a new chunk for a thread to process implies jumping ahead as many chunks as there are threads.

Figure 1-4 shows two simple examples of 1D data partitioning. In the block partition, each thread takes only one portion of the data to process, and in the cyclic partition, each thread takes more than one portion of the data to process. Figure 1-5 shows three simple examples of 2D data partitioning: block partitioning along the `y` dimension, block partitioning on both dimensions, and cyclic partitioning along the `x` dimension. The remaining patterns — block partitioning along the `x` dimension, cyclic partitioning on both dimensions, and cyclic partitioning along the `y` dimension — are left as an exercise.

Usually, data is stored one-dimensionally. Even when a logical multi-dimensional view of data is used, it still maps to one-dimensional physical storage. Determining how to distribute data among threads is closely related to both how that data is stored physically, as well as how the execution of each thread is ordered. The way you organize threads has a significant effect on the program's performance.

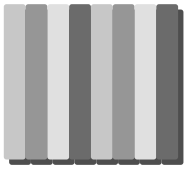
Block partition: each thread takes one data block

Cyclic partition: each thread takes two data blocks

FIGURE 1-4

Block partition on one dimension

Block partition on both dimensions



Cyclic partition on one dimension

FIGURE 1-5

DATA PARTITIONS

There are two basic approaches to partitioning data:

- **Block:** Each thread takes one portion of the data, usually an equal portion of the data.
- **Cyclic:** Each thread takes more than one portion of the data.

The performance of a program is usually sensitive to the block size. Determining an optimal partition for both block and cyclic partitioning is closely related to the computer architecture. You will learn more about this through the examples in this book.

Computer Architecture

There are several different ways to classify computer architectures. One widely used classification scheme is *Flynn's Taxonomy*, which classifies architectures into four different types according to how instructions and data flow through cores (see Figure 1-6), including:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

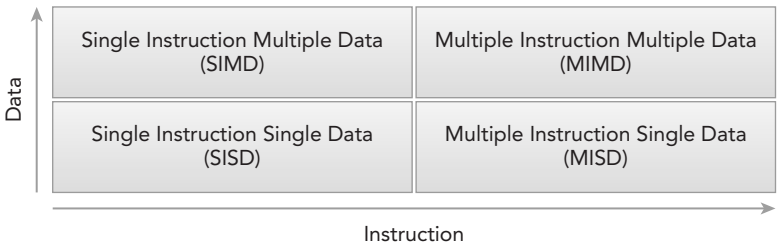


FIGURE 1-6

Single Instruction Single Data refers to the traditional computer: a serial architecture. There is only one core in the computer. At any time only one instruction stream is executed, and operations are performed on one data stream.

Single Instruction Multiple Data refers to a type of parallel architecture. There are multiple cores in the computer. All cores execute the same instruction stream at any time, each operating on different data streams. Vector computers are typically characterized as SIMD, and most modern computers employ a SIMD architecture. Perhaps the biggest advantage of SIMD is that, while writing code on the CPU, programmers can continue to think sequentially yet achieve parallel speed-up from parallel data operations because the compiler takes care of the details.

Multiple Instruction Single Data refers to an uncommon architecture, where each core operates on the same data stream via separate instruction streams.

Multiple Instruction Multiple Data refers to a type of parallel architecture in which multiple cores operate on multiple data streams, each executing independent instructions. Many MIMD architectures also include SIMD execution sub-components.

At the architectural level, many advances have been made to achieve the following objectives:

- Decrease latency
- Increase bandwidth
- Increase throughput

Latency is the time it takes for an operation to start and complete, and is commonly expressed in microseconds. *Bandwidth* is the amount of data that can be processed per unit of time, commonly expressed as megabytes/sec or gigabytes/sec. *Throughput* is the amount of operations that can be processed per unit of time, commonly expressed as *gflops* (which stands for billion floating-point operations per second), especially in fields of scientific computation that make heavy use of floating-point calculations. Latency measures the time to complete an operation, while throughput measures the number of operations processed in a given time unit.

Computer architectures can also be subdivided by their memory organization, which is generally classified into the following two types:

- Multi-node with distributed memory
- Multiprocessor with shared memory

In a multi-node system, large scale computational engines are constructed from many processors connected by a network. Each processor has its own local memory, and processors can communicate the contents of their local memory over the network. Figure 1-7 shows a typical multi-node system with distributed memory. These systems are often referred to as *clusters*.

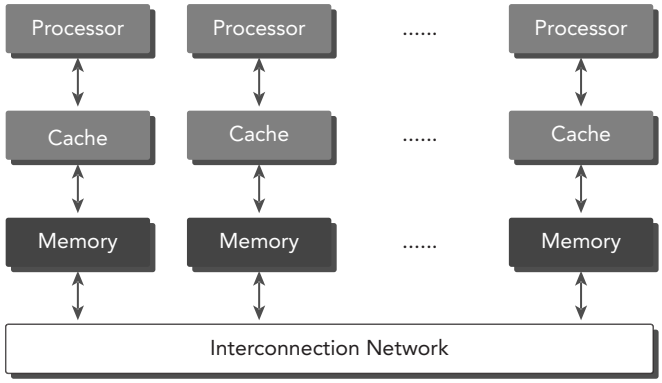


FIGURE 1-7

Multiprocessor architectures typically range in size from dual-processor to dozens or hundreds of processors. These processors are either physically connected to the same memory (as shown in Figure 1-8), or share a low-latency link (such as PCI-Express or PCIe). Although sharing memory implies a shared address space, it does not necessarily mean there is a single physical memory. Such multiprocessors include both single-chip systems with multiple cores, known as *multicore*, and computers consisting of multiple chips, each of which might have a multicore design. Multicore architectures have displaced single-core architectures permanently.

The term *many-core* is usually used to describe multicore architectures with an especially high number of cores (tens or hundreds). Recently, computer architectures have been transitioning from multicore to many-core.

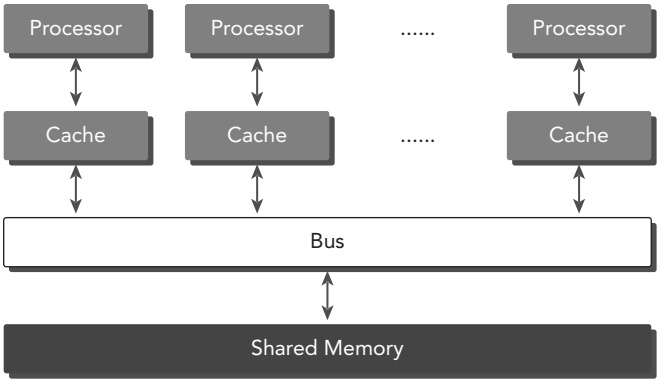


FIGURE 1-8

GPUs represent a many-core architecture, and have virtually every type of parallelism described previously: multithreading, MIMD, SIMD, and instruction-level parallelism. NVIDIA coined the phrase *Single Instruction, Multiple Thread* (SIMT) for this type of architecture.

GPUs and CPUs do not share a common ancestor. Historically, GPUs are graphics accelerators. Only recently have GPUs evolved to be powerful, general-purpose, fully programmable, task and data parallel processors, ideally suited to tackle massively parallel computing problems.

GPU CORE VERSUS CPU CORE

Even though many-core and multicore are used to label GPU and CPU architectures, a GPU core is quite different than a CPU core.

A CPU core, relatively heavy-weight, is designed for very complex control logic, seeking to optimize the execution of sequential programs.

A GPU core, relatively light-weight, is optimized for data-parallel tasks with simpler control logic, focusing on the throughput of parallel programs.

HETEROGENEOUS COMPUTING

In the earliest days, computers contained only central processing units (CPUs) designed to run general programming tasks. Since the last decade, mainstream computers in the high-performance computing community have been switching to include other processing elements. The most prevalent is the GPU, originally designed to perform specialized graphics computations in parallel. Over time, GPUs have become more powerful and more generalized, enabling them to be applied to general-purpose parallel computing tasks with excellent performance and high power efficiency.

Typically, CPUs and GPUs are discrete processing components connected by the PCI-Express bus within a single compute node. In this type of architecture, GPUs are referred to as discrete devices.

The switch from homogeneous systems to heterogeneous systems is a milestone in the history of high-performance computing. *Homogeneous computing* uses one or more processor of the same architecture to execute an application. *Heterogeneous computing* instead uses a suite of processor architectures to execute an application, applying tasks to architectures to which they are well-suited, yielding performance improvement as a result.

Although heterogeneous systems provide significant advantages compared to traditional high-performance computing systems, effective use of such systems is currently limited by the increased application design complexity. While parallel programming has received much recent attention, the inclusion of heterogeneous resources adds complexity.

If you are new to parallel programming, then you can benefit from the performance improvements and advanced software tools now available on heterogeneous architectures. If you are already a good parallel programmer, adapting to parallel programming on heterogeneous architectures is straightforward.

Heterogeneous Architecture

A typical heterogeneous compute node nowadays consists of two multicore CPU sockets and two or more many-core GPUs. A GPU is currently not a standalone platform but a co-processor to a CPU. Therefore, GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus, as shown in Figure 1-9. That is why, in GPU computing terms, the CPU is called the *host* and the GPU is called the *device*.

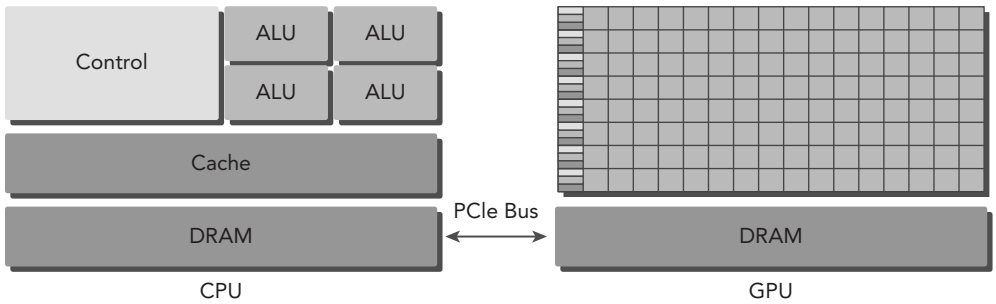


FIGURE 1-9

A heterogeneous application consists of two parts:

- Host code
- Device code

Host code runs on CPUs and *device code* runs on GPUs. An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device.

With computational intensive applications, program sections often exhibit a rich amount of data parallelism. GPUs are used to accelerate the execution of this portion of data parallelism. When a

hardware component that is physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as a *hardware accelerator*. GPUs are arguably the most common example of a hardware accelerator.

NVIDIA's GPU computing platform is enabled on the following product families:

- Tegra
- GeForce
- Quadro
- Tesla

The Tegra product family is designed for mobile and embedded devices such as tablets and phones, GeForce for consumer graphics, Quadro for professional visualization, and Tesla for datacenter parallel computing. Fermi, the GPU accelerator in the Tesla product family, has recently gained widespread use as a computing accelerator for high-performance computing applications. Fermi, released by NVIDIA in 2010, is the world's first complete GPU computing architecture. Fermi GPU accelerators have already redefined and accelerated high-performance computing capabilities in many areas, such as seismic processing, biochemistry simulations, weather and climate modeling, signal processing, computational finance, computer-aided engineering, computational fluid dynamics, and data analysis. Kepler, the current generation of GPU computing architecture after Fermi, released in the fall of 2012, offers much higher processing power than the prior GPU generation and provides new methods to optimize and increase parallel workload execution on the GPU, expecting to further revolutionize high-performance computing. The Tegra K1 contains a Kepler GPU and provides everything you need to unlock the power of the GPU for embedded applications.

There are two important features that describe GPU capability:

- Number of CUDA cores
- Memory size

Accordingly, there are two different metrics for describing GPU performance:

- Peak computational performance
- Memory bandwidth

Peak computational performance is a measure of computational capability, usually defined as how many single-precision or double-precision floating point calculations can be processed per second. Peak performance is usually expressed in `gflops` (billion floating-point operations per second) or `tflops` (trillion floating-point calculations per second). *Memory bandwidth* is a measure of the ratio at which data can be read from or stored to memory. Memory bandwidth is usually expressed in gigabytes per second, GB/s. Table 1-1 provides a brief summary of Fermi and Kepler architectural and performance features.

TABLE 1-1: Fermi and Kepler

	FERMI (TESLA C2050)	KEPLER (TESLA K10)
CUDA Cores	448	2 x 1536
Memory	6 GB	8 GB
Peak Performance*	1.03 Tflops	4.58 Tflops
Memory Bandwidth	144 GB/s	320 GB/s

* Peak single-precision floating point performance

Most examples in this book can be run on both Fermi and Kepler GPUs. Some examples require special architectural features only included with Kepler GPUs.

COMPUTE CAPABILITIES

NVIDIA uses a special term, *compute capability*, to describe hardware versions of GPU accelerators that belong to the entire Tesla product family. The version of Tesla products is given in Table 1-2.

Devices with the same major revision number are of the same core architecture.

- Kepler class architecture is major version number 3.
- Fermi class architecture is major version number 2.
- Tesla class architecture is major version number 1.

The first class of GPUs delivered by NVIDIA contains the same Tesla name as the entire family of Tesla GPU accelerators.

All examples in this book require compute capability above 2.

TABLE 1-2: Compute Capabilities of Tesla GPU Computing Products

GPU	COMPUTE CAPABILITY
Tesla K40	3.5
Tesla K20	3.5
Tesla K10	3.0
Tesla C2070	2.0
Tesla C1060	1.3

Paradigm of Heterogeneous Computing

GPU computing is not meant to replace CPU computing. Each approach has advantages for certain kinds of programs. CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks. When CPUs are complemented by GPUs, it makes for a powerful combination. The CPU is optimized for dynamic workloads marked by short sequences of computational operations and unpredictable control flow; and GPUs aim at the other end of the spectrum: workloads that are dominated by computational tasks with simple control flow. As shown in Figure 1-10, there are two dimensions that differentiate the scope of applications for CPU and GPU:

- Parallelism level
- Data size

If a problem has a small data size, sophisticated control logic, and/or low-level parallelism, the CPU is a good choice because of its ability to handle complex logic and instruction-level parallelism. If the problem at hand instead processes a huge amount of data and exhibits massive data parallelism, the GPU is the right choice because it has a large number of programmable cores, can support massive multi-threading, and has a larger peak bandwidth compared to the CPU.

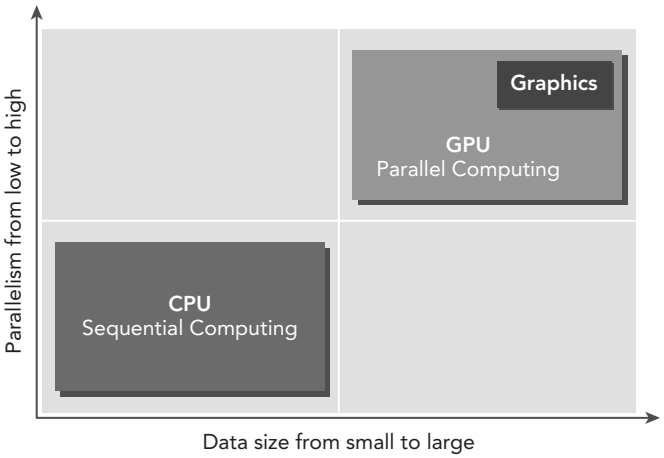


FIGURE 1-10

CPU + GPU heterogeneous parallel computing architectures evolved because the CPU and GPU have complementary attributes that enable applications to perform best using both types of processors. Therefore, for optimal performance you may need to use both CPU and GPU for your application, executing the sequential parts or task parallel parts on the CPU and intensive data parallel parts on the GPU, as shown in Figure 1-11.

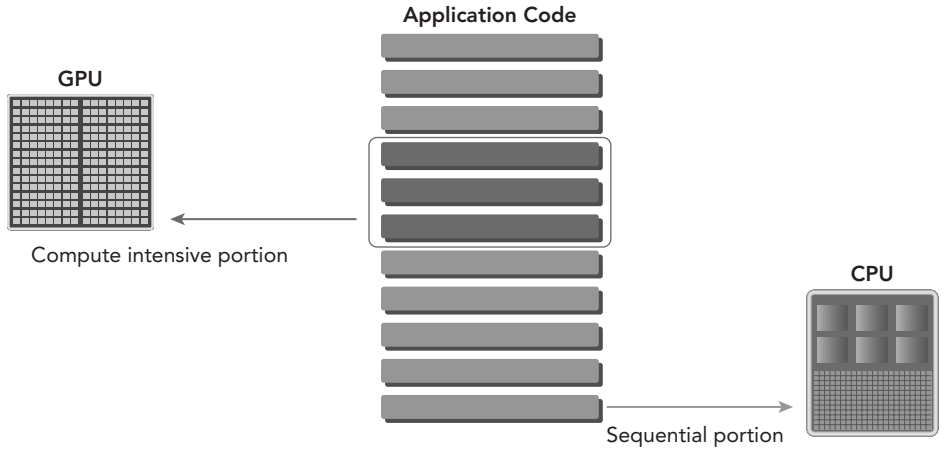


FIGURE 1-11

Writing code this way ensures that the characteristics of the GPU and CPU complement each other, leading to full utilization of the computational power of the combined CPU + GPU system. To support joint CPU + GPU execution of an application, NVIDIA designed a programming model called CUDA. This new programming model is the focus for the rest of this book.

CPU THREAD VERSUS GPU THREAD

Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.

Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.

CPU cores are designed to minimize latency for one or two threads at a time, whereas GPU cores are designed to handle a large number of concurrent, lightweight threads in order to maximize throughput.

Today, a CPU with four quad core processors can run only 16 threads concurrently, or 32 if the CPUs support hyper-threading.

Modern NVIDIA GPUs can support up to 1,536 active threads concurrently per multiprocessor. On GPUs with 16 multiprocessors, this leads to more than 24,000 concurrently active threads.

CUDA: A Platform for Heterogeneous Computing

CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way. Using CUDA, you can access the GPU for computation, as has been traditionally done on the CPU.

The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages, including C, C++, Fortran, and Python (as illustrated by Figure 1-12). This book focuses on CUDA C programming.

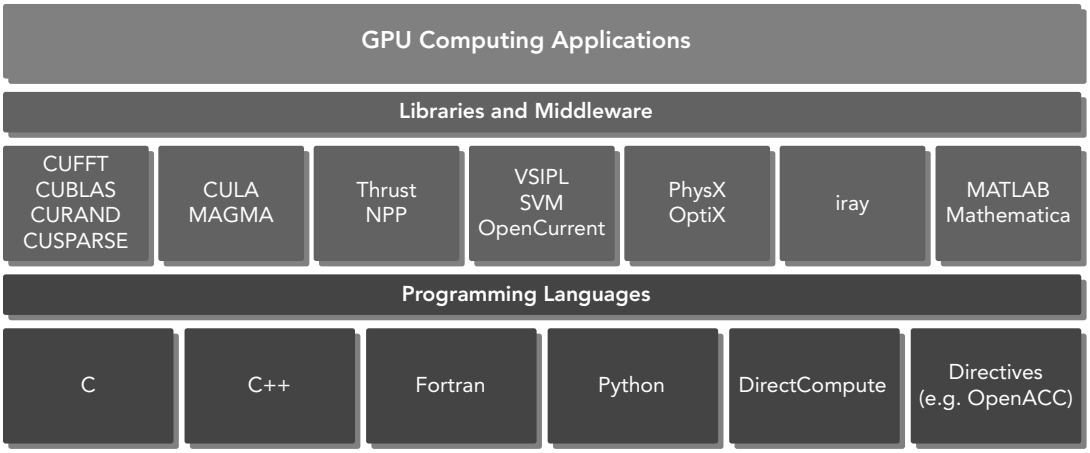


FIGURE 1-12

CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogeneous programming, and also straightforward APIs to manage devices, memory, and other tasks. CUDA is also a scalable programming model that enables programs to transparently scale their parallelism to GPUs with varying numbers of cores, while maintaining a shallow learning curve for programmers familiar with the C programming language.

CUDA provides two API levels for managing the GPU device and organizing threads, as shown in Figure 1-13.

- CUDA Driver API
- CUDA Runtime API

The *driver API* is a low-level API and is relatively hard to program, but it provides more control over how the GPU device is used. The *runtime API* is a higher-level API implemented on top of the

driver API. Each function of the runtime API is broken down into more basic operations issued to the driver API.

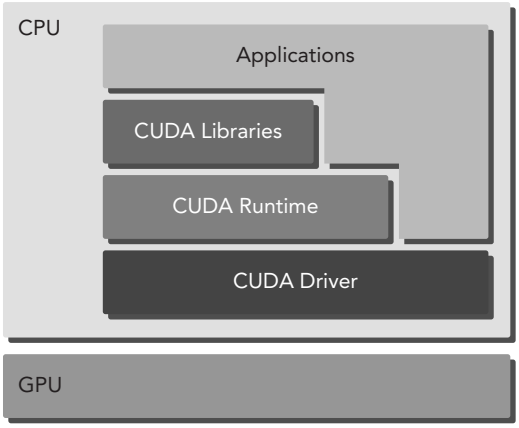


FIGURE 1-13

RUNTIME API VERSUS DRIVER API

There is no noticeable performance difference between the runtime and driver APIs. How your kernels use memory and how you organize your threads on the device have a much more pronounced effect.

These two APIs are mutually exclusive. You must use one or the other, but it is not possible to mix function calls from both. All examples throughout this book use the runtime API.

A CUDA program consists of a mixture of the following two parts:

- The host code runs on CPU.
- The device code runs on GPU.

NVIDIA's CUDA `nvcc` compiler separates the device code from the host code during the compilation process. As shown in Figure 1-14, the host code is standard C code and is further compiled with C compilers. The device code is written using CUDA C extended with keywords for labeling data-parallel functions, called *kernels*. The device code is further compiled by `nvcc`. During the link stage, CUDA runtime libraries are added for kernel procedure calls and explicit GPU device manipulation.

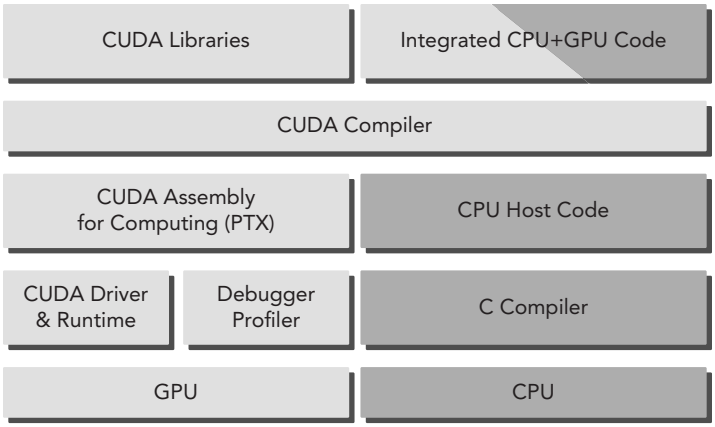


FIGURE 1-14

The CUDA `nvcc` compiler is based on the widely used *LLVM* open source compiler infrastructure. You can create or extend programming languages with support for GPU acceleration using the CUDA Compiler SDK, as shown in Figure 1-15.

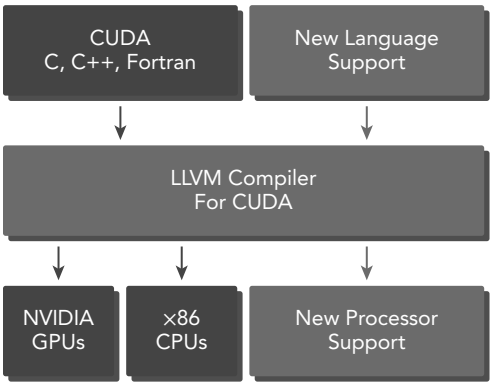


FIGURE 1-15

The CUDA platform is also a foundation that supports a diverse parallel computing ecosystem, as shown in Figure 1-16. Today, the CUDA ecosystem is growing rapidly as more and more companies provide world-class tools, services, and solutions. If you want to build your applications on GPUs, the easiest way to harness the performance of GPUs is with the CUDA Toolkit (`https://developer.nvidia.com/cuda-toolkit`), which provides a comprehensive development environment for C and C++ developers. The CUDA Toolkit includes a compiler, math libraries, and tools for debugging and optimizing the performance of your applications. You will also find code samples, programming guides, user manuals, API references, and other documentation to help you get started.

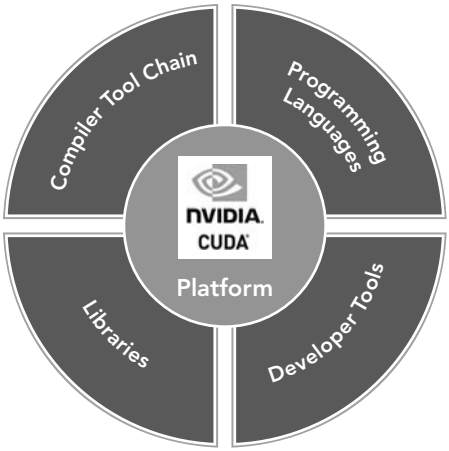


FIGURE 1-16

HELLO WORLD FROM GPU

The best way to learn a new programming language is by writing programs using the new language. In this section, you are going to write your first kernel code running on the GPU. The first program is the same for all languages: Print the string "Hello World."

If this is your first time working with CUDA, you may want to check that the CUDA compiler is installed properly with the following command on a Linux system:

```
$ which nvcc
```

A typical response would be:

```
/usr/local/cuda/bin/nvcc
```

You also need to check if a GPU accelerator card is attached in your machine. You can do so with the following command on a Linux system:

```
$ ls -l /dev/nv*
```

A typical response would be:

```
crw-rw-rw- 1 root root 195,   0 Jul  3 13:44 /dev/nvidia0
crw-rw-rw- 1 root root 195,   1 Jul  3 13:44 /dev/nvidia1
crw-rw-rw- 1 root root 195, 255 Jul  3 13:44 /dev/nvidiactl
crw-rw---- 1 root root  10, 144 Jul  3 13:39 /dev/nvram
```

In this example, you have two GPU cards installed (your configuration may be different, and may show more or fewer devices). Now you are ready to write your first CUDA C code. To write a CUDA C program, you need to:

1. Create a source code file with the special file name extension of `.cu`.
2. Compile the program using the CUDA `nvcc` compiler.
3. Run the executable file from the command line, which contains the kernel code executable on the GPU.

As a starting point, write a C program to print out "Hello World" as follows:

```
#include <stdio.h>
int main(void)
{
   printf("Hello World from CPU!\n");
}
```

Save the code into the file `hello.cu` and then compile it with `nvcc`. The CUDA `nvcc` compiler has similar semantics to `gcc` and other compilers.

```
$ nvcc hello.cu -o hello
```

If you run the executable file `hello`, it will print:

```
Hello World from CPU!
```

Next, write a kernel function, named `helloFromGPU`, to print the string of "Hello World from GPU!" as follows:

```
__global__ void helloFromGPU(void)
{
    printf("Hello World from GPU!\n");
}
```

The qualifier `__global__` tells the compiler that the function will be called from the CPU and executed on the GPU. Launch the kernel function with the following code:

```
  helloFromGPU <<<1,10>>>();
```

Triple angle brackets mark a call from the host thread to the code on the device side. A kernel is executed by an array of threads and all threads run the same code. The parameters within the triple angle brackets are the execution configuration, which specifies how many threads will execute the kernel. In this example, you will run 10 GPU threads. Putting all of these things together, you have the program shown in Listing 1-1:

LISTING 1-1: Hello World from GPU (hello.cu)

```
#include <stdio.h>

__global__ void helloFromGPU (void)
{
    printf("Hello World from GPU!\n");
```

```
}

int main(void)
{
   // hello from cpu
   printf("Hello World from CPU!\n");

   helloFromGPU <<<1, 10>>>();
   cudaDeviceReset();
   return 0;
}
```

The function `cudaDeviceReset()` will explicitly destroy and clean up all resources associated with the current device in the current process. Compile the code with the switch `-arch sm_20` on the `nvcc` command line as follows:

```
$ nvcc  -arch sm_20 hello.cu -o hello
```

The switch `-arch sm_20` causes the compiler to generate device code for the Fermi architecture. Run the executable file and it will print 10 strings of "Hello World from GPU!" as follows, from each thread.

```
$ ./hello
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

CUDA PROGRAM STRUCTURE

A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA kernel to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

In the simple program `hello.cu`, you only see the third step: Invoke the kernel. For the remainder of this book, examples will demonstrate each step in the CUDA program structure.

IS CUDA C PROGRAMMING DIFFICULT?

The main difference between CPU programming and GPU programming is the level of programmer exposure to GPU architectural features. Thinking in parallel and having a basic understanding of GPU architecture enables you to write parallel programs that scale to hundreds of cores as easily as you write a sequential program.

If you want to write efficient code as a parallel programmer, you need a basic knowledge of CPU architectures. For example, *locality* is a very important concept in parallel programming. Locality refers to the reuse of data so as to reduce memory access latency. There are two basic types of reference locality. *Temporal locality* refers to the reuse of data and/or resources within relatively small time durations. *Spatial locality* refers to the use of data elements within relatively close storage locations. Modern CPU architectures use large caches to optimize for applications with good spatial and temporal locality. It is the programmer's responsibility to design their algorithm to efficiently use CPU cache. Programmers must handle low-level cache optimizations, but have no introspection into how threads are being scheduled on the underlying architecture because the CPU does not expose that information.

CUDA exposes you to the concepts of both memory hierarchy and thread hierarchy, extending your ability to control thread execution and scheduling to a greater degree, using:

- Memory hierarchy structure
- Thread hierarchy structure

For example, a special memory, called *shared memory*, is exposed by the CUDA programming model. Shared memory can be thought of as a software-managed cache, which provides great speed-up by conserving bandwidth to main memory. With shared memory, you can control the locality of your code directly.

When writing a parallel program in ANSI C, you need to explicitly organize your threads with either *pthreads* or *OpenMP*, two well-known techniques to support parallel programing on most processor architectures and operating systems. When writing a program in CUDA C, you actually just write a piece of serial code to be called by only one thread. The GPU takes this kernel and makes it parallel by launching thousands of threads, all performing that same computation. The CUDA programming model provides you with a way to organize your threads hierarchically. Manipulating this organization directly affects the order in which threads are executed on the GPU. Because CUDA C is an extension of C, it is often straightforward to port C programs to CUDA C. Conceptually, peeling off the loops of your code yields the kernel code for a CUDA C implementation.

CUDA abstracts away the hardware details and does not require applications to be mapped to traditional graphics APIs. At its core are three key abstractions: a hierarchy of thread groups, a hierarchy

of memory groups, and barrier synchronization, which are exposed to you as a minimal set of language extensions. With each release of CUDA, NVIDIA is simplifying parallel programming. Though some still consider CUDA concepts to be low-level, raising the abstraction level any higher would damage your ability to control the interaction between your application and the platform. Without that ability, the performance of your application is beyond your control no matter what knowledge you have of the underlying architecture.

Therefore, the challenge to you is to learn the basics of GPU architecture and master the CUDA development tools and environment.

CUDA DEVELOPMENT ENVIRONMENT

NVIDIA provides a comprehensive development environment for C and C++ developers to build GPU-accelerated applications, including:

- NVIDIA Nsight™ integrated development environment
- CUDA-GDB command line debugger
- Visual and command line profiler for performance analysis
- CUDA-MEMCHECK memory analyzer
- GPU device management tools

After you become familiar with these tools, programming with CUDA C is straightforward and rewarding.

SUMMARY

As both computer architectures and parallel programming models have evolved, the design of each has intertwined to produce modern heterogeneous systems. The CUDA platform helps improve performance and programmer productivity on heterogeneous architectures.

CPU + GPU systems have become mainstream in the high-performance computing world. This change has led to a fundamental shift in the parallel programming paradigm: The data-parallel workload is executed on the GPU, while the serial and task-parallel workload is executed on the CPU.

Fermi and Kepler GPU accelerators, as complete GPU computing architectures, have already redefined the high-performance computing capabilities in many areas. After reading and understanding the concepts in this book, you will discover that writing CUDA programs that scale to hundreds or thousands of cores in a heterogeneous system is as easy as writing sequential programs.

CHAPTER 1 EXERCISES

1. Refer to Figure 1-5 and illustrate the following patterns of data partition:
   - Block partition along the *x* dimension for 2D data
   - Cyclic partition along the *y* dimension for 2D data
   - Cyclic partition along the *z* dimension for 3D data

2. Remove the `cudaDeviceReset` function from `hello.cu`, then compile and run it to see what would happen.

3. Replace the function `cudaDeviceReset` in `hello.cu` with `cudaDeviceSynchronize`, then compile and run it to see what happens.

4. Refer to the section "Hello World from GPU." Remove the device architecture flag from the compiler command line and compile it as follows to see what happens.

```
$ nvcc  hello.cu -o hello
```

5. Refer to the CUDA online document (`http://docs.nvidia.com/cuda/index.html`). Based on the section "CUDA Compiler Driver NVCC," what file suffixes does nvcc support compilation on?

6. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in *threadIdx.x* variable. Modify the kernel function in `hello.cu` with the thread index to let the output be:

```
$ ./hello
Hello World from CPU!
Hello World from GPU thread 5!
```