CHAPTER

Introduction to CUDA

3

CHAPTER CONTENTS

INTRODUCTION

To a CUDA™ programmer, the computing system consists of a *host*, which is a traditional central processing unit (CPU), such as an Intel® architecture microprocessor in personal computers today, and one or more *devices*, which are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Because data parallelism plays such an important role in CUDA, we will first discuss the concept of data parallelism before introducing the basic features of CUDA.

3.1 DATA PARALLELISM

Many software applications that process a large amount of data and thus incur long execution times on today's computers are designed to model real-world, physical phenomena. Images and video frames are snapshots of a physical world where different parts of a picture capture simultaneous, independent physical events. Rigid body physics and fluid dynamics model natural forces and movements that can be independently evaluated within small time steps. Such independent evaluation is the basis of data parallelism in these applications.

As we mentioned earlier, data parallelism refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner. We illustrate the concept of data parallelism with a matrix–matrix multiplication (matrix multiplication, for brevity) example in Figure 3.1. In this example, each element of the product matrix **P** is generated by performing a dot product between a row of input matrix **M** and a column of input matrix **N**. In Figure 3.1, the highlighted element of matrix **P** is generated by taking the dot product of the highlighted row of matrix **M** and the highlighted column of matrix **N**. Note that the dot product operations for computing different matrix **P** elements can be simultaneously performed. That is, none of these dot products will affect

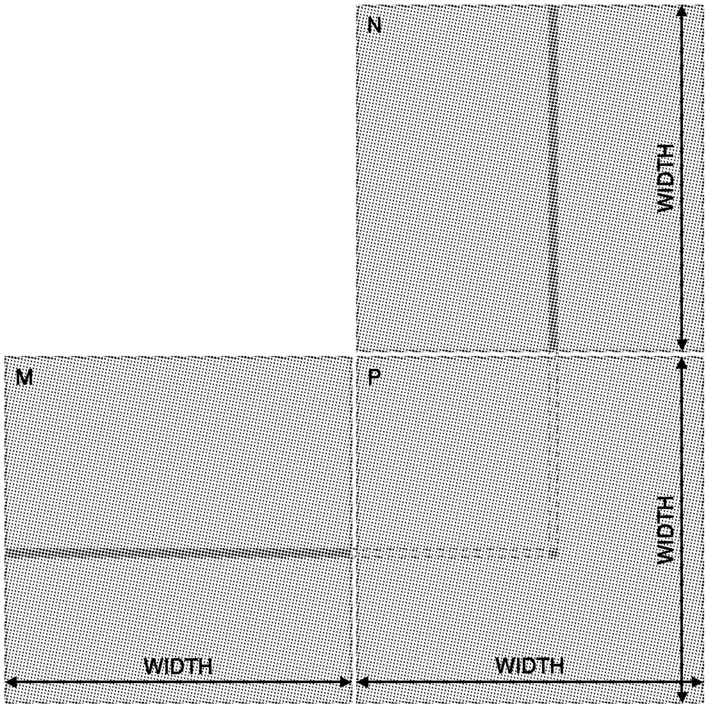


FIGURE 3.1

Data parallelism in matrix multiplication.

the results of each other. For large matrices, the number of dot products can be very large; for example, a 1000×1000 matrix multiplication has 1,000,000 independent dot products, each involving 1000 multiply and 1000 accumulate arithmetic operations. Therefore, matrix multiplication of large dimensions can have very large amount of data parallelism. By executing many dot products in parallel, a CUDA device can significantly accelerate the execution of the matrix multiplication over a traditional host CPU. The data parallelism in real applications is not always as simple as that in our matrix multiplication example. In a later chapter, we will discuss these more sophisticated forms of data parallelism.

3.2 CUDA PROGRAM STRUCTURE

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. A CUDA program is a unified source code encompassing both host and device code. The NVIDIA® C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. The device code is typically further compiled by the nvcc and executed on a GPU device. In situations where no device is available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU using the emulation features in CUDA software development kit (SDK) or the MCUDA tool [Stratton 2008].

The kernel functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism. In the matrix multiplication example, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix **P**. In this example, the number of threads used by the kernel is a function of the matrix dimension. For a 1000×1000 matrix multiplication, the kernel that uses one thread to compute one **P** element would generate 1,000,000 threads when it is invoked. It is worth noting that CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically require thousands of clock cycles to generate and schedule.

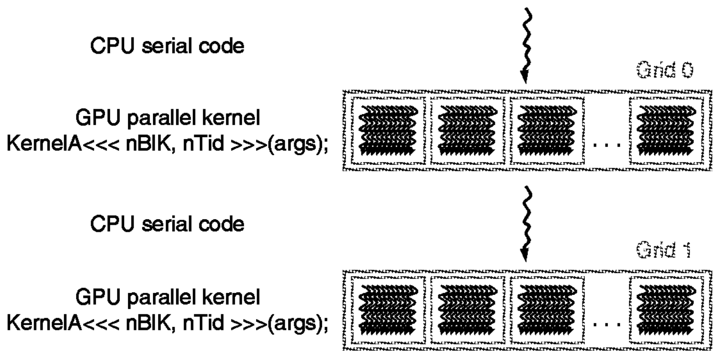


FIGURE 3.2

Execution of a CUDA program.

The execution of a typical CUDA program is illustrated in Figure 3.2. The execution starts with host (CPU) execution. When a kernel function is invoked, or *launched*, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a *grid*. Figure 3.2 shows the execution of two grids of threads. We will discuss how these grids are organized soon. When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.

3.3 A MATRIX–MATRIX MULTIPLICATION EXAMPLE

At this point, it is worthwhile to introduce a code example that concretely illustrates the CUDA program structure. Figure 3.3 shows a simple main function skeleton for the matrix multiplication example. For simplicity, we assume that the matrices are square in shape, and the dimension of each matrix is specified by the parameter `Width`.

The main program first allocates the **M**, **N**, and **P** matrices in the host memory and then performs I/O to read in **M** and **N** in Part 1. These are ANSI C operations, so we are not showing the actual code for the sake of brevity. The detailed code of the main function and some user-defined ANSI C functions is shown in Appendix A. Similarly, after completing the matrix multiplication, Part 3 of the main function performs I/O to write the product matrix **P** and to free all the allocated matrices. The details of Part 3 are also shown in Appendix A. Part 2 is the main focus of our

```
int main(void) {
1.  // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

2.  // M * N on the device
    MatrixMultiplication(M, N, P, Width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

FIGURE 3.3

A simple main function for the matrix multiplication example.

example. It calls a function, `MatrixMultiplication()`, to perform matrix multiplication on a device.

Before we explain how to use a CUDA device to execute the matrix multiplication function, it is helpful to first review how a conventional CPU-only matrix multiplication function works. A simple version of a CPU-only matrix multiplication function is shown in Figure 3.4. The `MatrixMultiplication()` function implements a straightforward algorithm that consists of three loop levels. The innermost loop iterates over variable k and steps through one row of matrix **M** and one column of matrix **N**. The loop calculates a dot product of the row of **M** and the column of **N** and generates one element of **P**. Immediately after the innermost loop, the **P** element generated is written into the output **P** matrix.

The index used for accessing the **M** matrix in the innermost loop is `i*Width+k`. This is because the **M** matrix elements are placed into the system memory that is ultimately accessed with a linear address. That is, every location in the system memory has an address that ranges from 0 to the largest memory location. For C programs, the placement of a 2-dimensional matrix into this linear addressed memory is done according to the row-major convention, as illustrated in Figure 3.5.[1] All elements of a row are placed into consecutive memory locations. The rows are then placed one after another. Figure 3.5 shows an example where a 4×4 matrix is

[1]Note that FORTRAN adopts the column–major placement approach: All elements of a column are first placed into consecutive locations, and all columns are then placed in their numerical order.

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         float sum = 0;
         for (int k  = 0; k < Width; ++k) {
            float a = M[i * width + k];
            float b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```

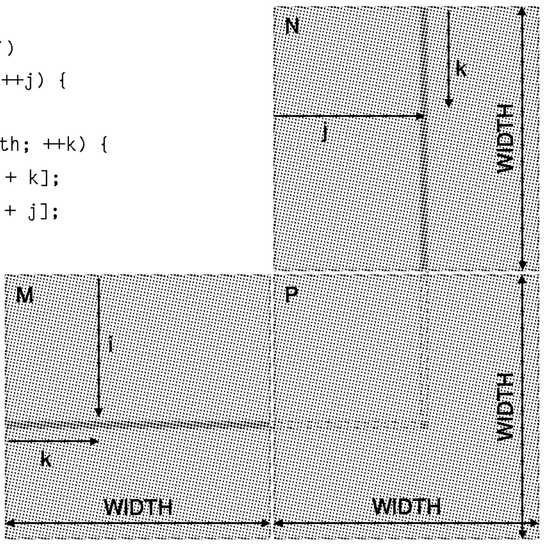


FIGURE 3.4

A simple matrix multiplication function with only host code.

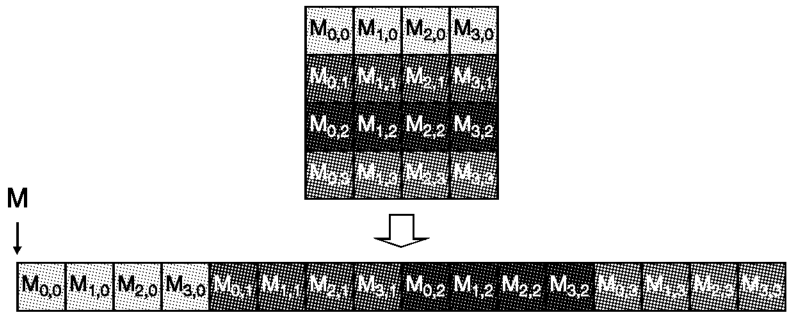

M0,0 M1,0 M2,0 M3,0
M0,1 M1,1 M2,1 M3,1
M0,2 M1,2 M2,2 M3,2
M0,3 M1,3 M2,3 M3,3
M
M0,0 M1,0 M2,0 M3,0 M0,1 M1,1 M2,1 M3,1 M0,2 M1,2 M2,2 M3,2 M0,3 M1,3 M2,3 M3,3

FIGURE 3.5

Placement of two-dimensional array elements into the linear address system memory.

placed into 16 consecutive locations, with all elements of row 0 first followed by the four elements of row 1, etc. Therefore, the index for an **M** element in row *i* and column *k* is `i*Width+k`. The `i*Width` term skips over all elements of the rows before row *i*. The *k* term then selects the proper element within the section for row *i*.

The outer two (*i* and *j*) loops in Figure 3.4 jointly iterate over all rows of **M** and all columns of **N**; each joint iteration performs a row–column dot product to generate one **P** element. Each *i* value identifies a row. By systematically iterating all **M** rows and all **N** columns, the function generates all **P** elements. We now have a complete matrix multiplication function that executes solely on the CPU. Note that all of the code that we have shown so far is in standard C.

Assume that a programmer now wants to port the matrix multiplication function into CUDA. A straightforward way to do so is to modify the `MatrixMultiplication()` function to move the bulk of the calculation to a CUDA device. The structure of the revised function is shown in Figure 3.6. Part 1 of the function allocates device (GPU) memory to hold copies of the **M**, **N**, and **P** matrices and copies these matrices over to the device memory. Part 2 invokes a kernel that launches parallel execution of the actual matrix multiplication on the device. Part 3 copies the product matrix **P** from the device memory back to the host memory.

Note that the revised `MatrixMultiplication()` function is essentially an outsourcing agent that ships input data to a device, activates the calculation on the device, and collects the results from the device. The agent does so in such

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M and N to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

FIGURE 3.6

Outline of a revised host code `MatrixMultiplication()` that moves the matrix multiplication to a device.

a way that the main program does not have to even be aware that the matrix multiplication is now actually done on a device. The details of the revised function, as well as the way to compose the kernel function, will serve as illustrations as we introduce the basic features of the CUDA programming model.

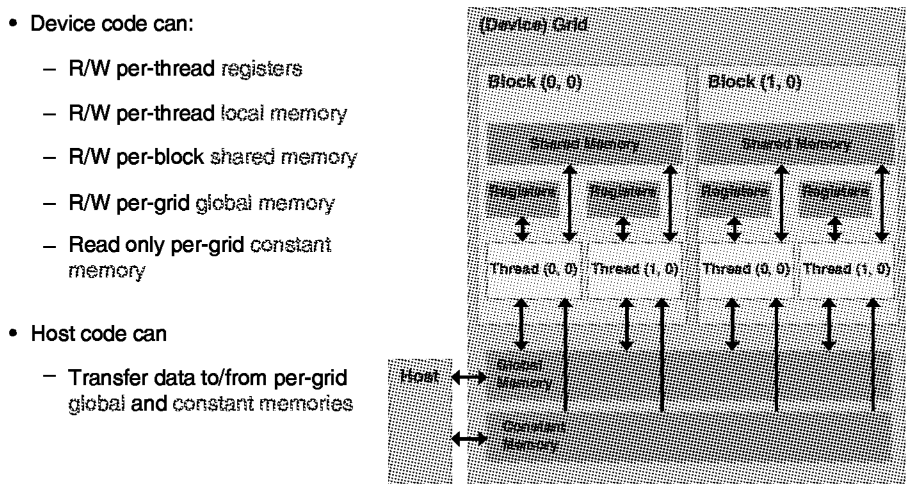
3.4 DEVICE MEMORIES AND DATA TRANSFER

In CUDA, the host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own dynamic random access memory (DRAM). For example, the NVIDIA T10 processor comes with up to 4 GB (billion bytes, or gigabytes) of DRAM. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 of Figure 3.6. Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed. This corresponds to Part 3 of Figure 3.6. The CUDA runtime system provides application programming interface (API) functions to perform these activities on behalf of the programmer. From this point on, we will simply say that a piece of data is transferred from host to device as shorthand for saying that the piece of data is transferred from the host memory to the device memory. The same holds for the opposite data transfer direction.

Figure 3.7 shows an overview of the CUDA device memory model for programmers to reason about the allocation, movement, and usage of the various memory types of a device. At the bottom of the figure, we see global memory and constant memory. These are the memories that the host code can transfer data to and from the device, as illustrated by the bidirectional arrows between these memories and the host. Constant memory allows read-only access by the device code and is described in Chapter 5. For now, we will focus on the use of global memory. Note that the host memory is not explicitly shown in Figure 3.7 but is assumed to be contained in the host.[2]

The CUDA memory model is supported by API functions that help CUDA programmers to manage data in these memories. Figure 3.8 shows the API functions for allocating and deallocating device global memory. The function `cudaMalloc()` can be called from the host code to allocate

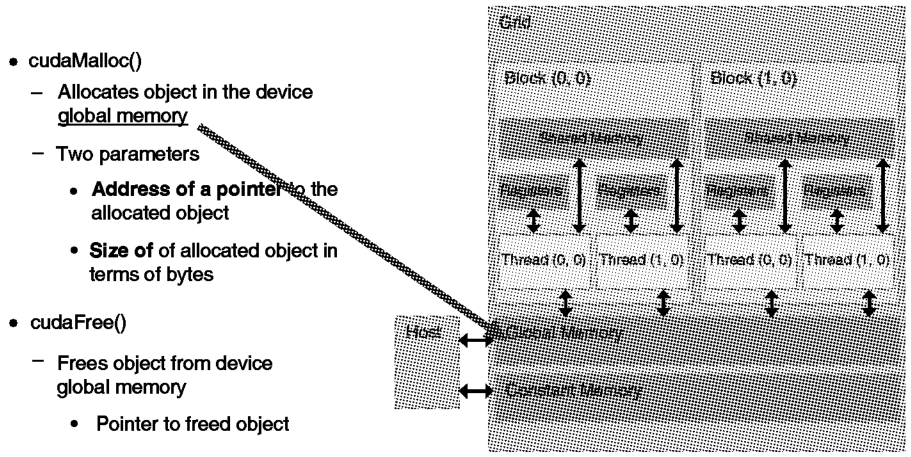
[2]Note that we have omitted the texture memory from Figure 3.7 for simplicity. We will introduce texture memory later.



- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories

FIGURE 3.7

Overview of the CUDA device memory model.



- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object

FIGURE 3.8

CUDA API functions for device global memory management.

a piece of global memory for an object. The reader should be able to notice the striking similarity between `cudaMalloc()` and the standard C runtime library `malloc()`. This is intentional; CUDA is C with minimal extensions. CUDA uses the standard C runtime library `malloc()` function to manage the host memory and adds `cudaMalloc()` as an extension to the C runtime library. By keeping the interface as close to the original C runtime libraries as possible, CUDA minimizes the time that a C programmer needs to relearn the use of these extensions.

The first parameter of the `cudaMalloc()` function is the address of a pointer variable that must point to the allocated object after allocation. The address of the pointer variable should be cast to `(void **)` because the function expects a generic pointer value; the memory allocation function is a generic function that is not restricted to any particular type of objects. This address allows the `cudaMalloc()` function to write the address of the allocated object into the pointer variable.[3] The second parameter of the `cudaMalloc()` function gives the size of the object to be allocated, in terms of bytes. The usage of this second parameter is consistent with the size parameter of the C `malloc()` function.

We now use a simple code example illustrating the use of `cudaMalloc()`. This is a continuation of the example in Figure 3.6. For clarity, we will end a pointer variable with the letter "d" to indicate that the variable is used to point to an object in the device memory space. The programmer passes the address of **Md** (i.e., `&Md`) as the first parameter after casting it to a void pointer; that is, **Md** is the pointer that points to the device global memory region allocated for the **M** matrix. The size of the allocated array will be `Width*Width*4` (the size of a single-precision floating number). After the computation, `cudaFree()` is called with pointer **Md** as input to free the storage space for the **M** matrix from the device global memory:

```
float *Md
int size = Width * Width * sizeof(float);
cudaMalloc((void**)&Md, size);
...
cudaFree(Md);
```

The reader should complete Part 1 of the `MatrixMultiplication()` example in Figure 3.6 with similar declarations of an **Nd** and a **Pd** pointer variable as

[3]Note that `cudaMalloc()` has a different format from the C `malloc()` function. The `C Malloc()` function returns a pointer to the allocated object. It takes only one parameter that specifies the size of the allocated object. The `cudaMalloc()` function writes to the pointer variable whose address is given as the first parameter. As a result, the `cudaMalloc()` function takes two parameters. The two-parameter format of `cudaMalloc()` allows it to use the return value to report any errors in the same way as other CUDA API functions.