
This is a draft chapter from an upcoming CUDA textbook by David Kirk from NVIDIA and Prof. Wen-mei Hwu from UIUC.

Please send any comment to dkirk@nvidia.com and w-hwu@uiuc.edu

---

This material is also part of the IAP09 CUDA@MIT (6.963) course. You may direct your questions about the class to Nicolas Pinto (pinto@mit.edu).

Chapter 2
CUDA Programming Model

To a CUDA programmer, the computing system consists of a *host* that is a traditional Central Processing Unit (CPU), such an Intel Architecture microprocessor in personal computers today, and one or more *devices* that are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, there are often program sections that exhibit rich amount of data parallelism, a property where many arithmetic operations can 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. Since 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.

2.1. Data Parallelism

Many software applications that process a large amount of data, and thus incur long execution time on today's computers, are designed to model real-world, physical phenomena. Images and video frames are snap shots 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 where 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 multiplication example in Figure 2.1. 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. This is illustrated in Figure 2.1, where the highlighted element of P is generated by taking the dot product of the highlighted row of M and the highlighted column of N. Note that the dot product operations for computing different P elements can be simultaneously performed. That is, none of these dot products will affect the results of each other. For large matrices, the number of dot products can be very large. For example, for a 1,000 X

1,000 matrix multiplication, there are 1,000,000 independent dot products, each involves 1,000 multiply and 1,000 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 tradition 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.

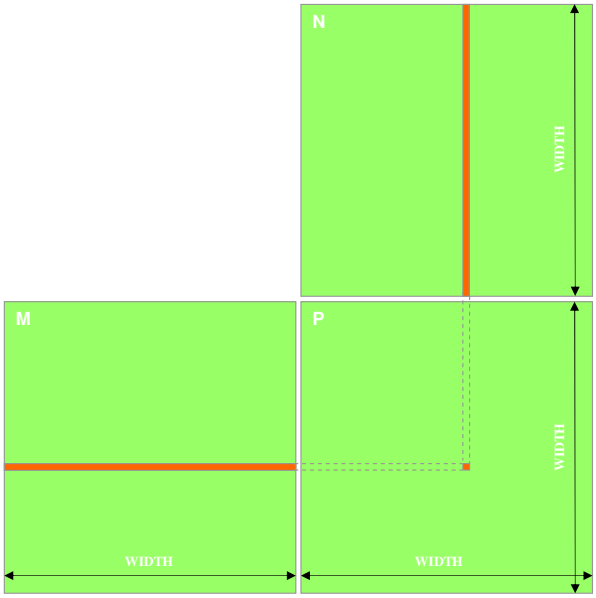

Figure 2.1. Data parallelism in matrix multiplication.

2.2. 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. The program supplies a single source code encompassing both host and device code. The NVIDIA C Compiler (NVCC) separates the two. The host code is straight ANSI C code and is compiled with the host's standard C compilers and runs as an ordinary 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 there is no device available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU. We will discuss this option in more detail in Chapter mCUDA.

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 the output (P) matrix. In this example, the number of threads used by the kernel is a function of the matrix dimension. For a 1,000 X 1,000 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 take thousands of clock cycles to generate and schedule.

The execution of a typical CUDA program is illustrated in Figure 2.2. The execution starts with host (CPU) execution. When a kernel function is invoked, 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 2.2 shows the execution of two Girds of threads. We will discuss how these grids are organized soon. When all threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is invoked.

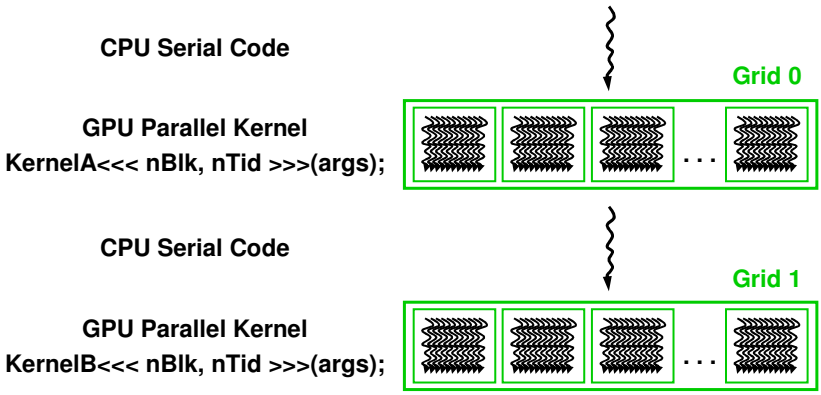

Figure 2.2. Execution of a CUDA program.

2.3 A Matrix Multiplication Example

At this point, it is worthwhile to introduce a code example that concretely illustrates the CUDA program structure. Figure 2.3 shows a simple host code skeleton for matrix multiplication. For simplicity, we assume that the matrices are square in their shapes with the dimension of each matrix specified by a parameter *width*.


```
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
     MatrixMulOnDevice(M, N, P, width);


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

Figure 2.3 A simple CUDA host code skeleton for matrix multiplication.

The main program first allocates the M, N, and P matrices 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 simplicity. The detailed code of the main function and some user defined ANSI C function is shown in Appendix I. Similarly, after completing the matrix multiplication, the main function performs I/O to write the product matrix P and the free all the allocated matrices. The details of Part 2 are also shown in Appendix I. Part 2 is the main focus of our example. It calls a function, MatrixMulOnDevice() to perform matrix multiplication on a device. We will use more details of MatrixMulOnDevice() to explain the basic CUDA programming model.

2.4. Device Memories and Data Transfer

In CUDA, 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 GeForce 8800 GTX card that we will use as the device through the book comes with 768 MB (million bytes, or mega-bytes) of DRAM. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer the pertinent data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer result data from device back to the host and free up the device memory that is no longer needed. The CUDA runtime system provides Application Programming Interface (API) function calls to perform these activities for use by the programmer.

Figure 2.4 shows an overview of the CUDA device memory model for programmers to reason about the allocation, movement, and usage of the various memory types available on a device. At the bottom of the picture, we see global memory and constant memory. These are the memories that the host code can write (W) to and read (R) from. Constant memory allow read-only access by the device; we will describe them in Chapter [CUA-Memoy]. For now, we will focus on the use of Global memory. Note that the host memory is not explicitly shown in Figure 2.4, but assumed to be contained in the host.

- 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
  - R/W per grid global and constant memories

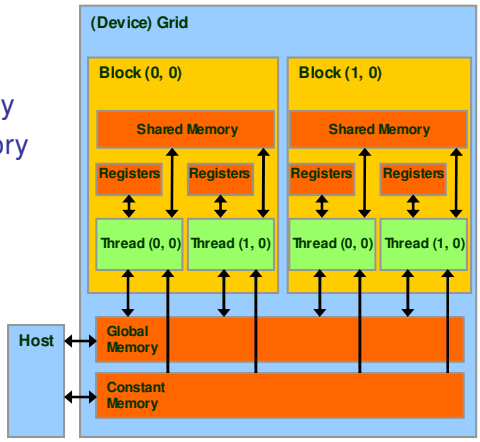

Figure 2.4 Overview of the CUDA device memory model.

The concept CUDA memory model is supported by the API functions that can be called by CUDA programmers. Figure 2.5 shows the two most important API functions for allocating and de-allocating device Global Memory. Function cudaMalloc() can be called from the host code to allocate a piece of Global Memory for an object. The first parameter for the cudaMalloc() function is the address of a pointer that needs to point to the allocated object after a piece of Global Memory is allocated to it. The second parameter gives the size of the object to be allocated.

The use of cudaMalloc() can be illustrated with a simple code example that continues from Figure 2.3. Let's assume that a programmer wishes to perform a 64x64 single-precision matrix multiplication on the device and have a pointer variable Md that can point to the first element of a single precision array. For clarity, we will end a variable with letter "d" to indicate that the variable is used as an object in the device memory space. In the following code example, we assume that Width is a variable or constant that is set to 64. The programmer specifies that Md is the pointer that should point to the data region allocated for the 64x64 matrix. Since Width is set at 64, the size of the allocated array will be 64*64*(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 64x64 matrix from the Global Memory.

```
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**)&Md, size);
…
cudaFree(Md);
```

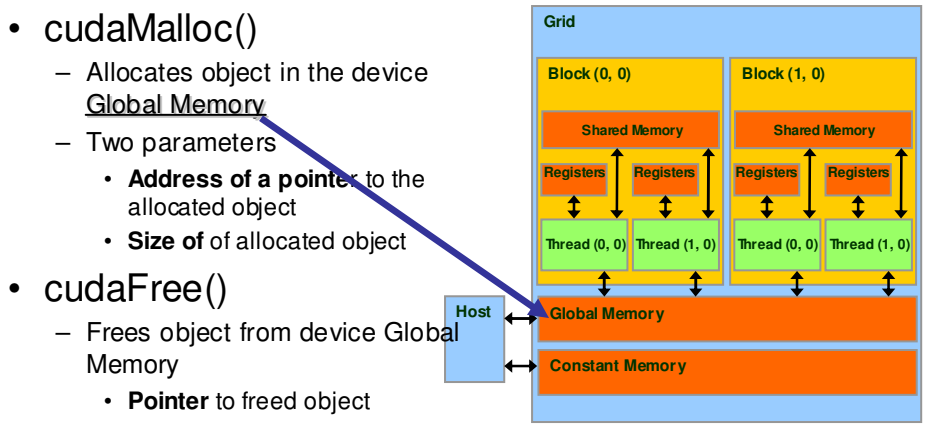

Figure 2.5 CUDA API Functions for Device Global Memory Management.

Once a program has allocated device memory for the data objects, it can request that data be transferred from the host to the device memory. This is accomplished by calling one of the CUDA API functions for data transfer between memories. Figure 2.6 shows the API function for such data transfer. The cudaMemcpy() function requires four parameters. The first parameter is a pointer to the source data object to be copied. The second parameter points to the destination location for the copy operation. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the types of memory involved in the copy: from host memory to host memory, from host memory to devise memory, from device memory to host memory, and from device memory to device memory. For example, the memory copy function can use used to copy data from one location of the device memory to another location of the device memory.

For the matrix multiplication example, the host code calls the cudaMemcpy() function to copy M and N matrices from the host memory to the device memory before the multiplication and then to copy the P matrix from the device memory to the host memory after the multiplication is done. Assume that M, P, Md, Pd and size have already been set as we discussed before, the two function calls are shown below. Note that the two symbolic constants, cudaMemcopyHostToDevice and cudaMemcopyDeviceToHost, are recognized, predefined constants of the CUDA programming environment. Note that the same function can be used to transfer data in both directions by properly ordering the source and destination pointers and using the appropriate constant for the transfer type.

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Transfer is asynchronous

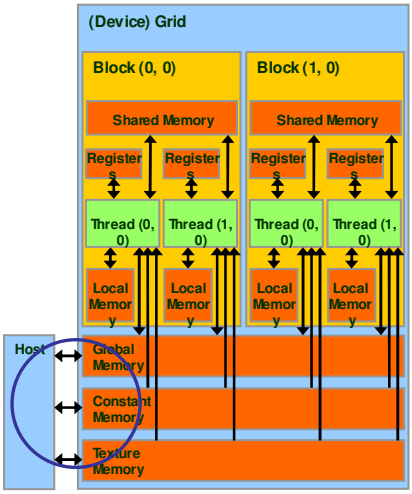

Figure 2.6 CUDA API Functions for Data Transfer Between Memories.

Now we are ready to complete the details of invoking a kernel in the matrix multiplication example. As shown in Figure 2.3, the host code calls matrixMulOnDevice(), which is also executed on the host. It is responsible for allocating device memory, performing data transfers, and then activating the kernel that performs the actual matrix multiplication. We often refer to this type of host code as the *stub function* for invoking a kernel. After the matrix multiplication, matrixMulOnDevice() also At this point, the reader should be able to write this function. We show the the function in Figure 2.7. The code consists of three parts. The first part allocates device memory for Md, Nd, and Pd, the device counter part of M, N, and P and transfer M to Md and N to Nd. The second part actually invokes the kernel and will be described later. The third part reads the product from device memory variable Pd to host memory variable P so that the value will be available to main(). It then frees Md, Nd, and Pd from the device memory.

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and N to device memory
   cudaMalloc(Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc(Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(Pd, size);

2.  // Kernel invocation code – to be shown later
    …
3.  // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

Figure 2.7 The MatrixMulOnDevice() Function.

2.5. Kernel Functions and Threading

We now discuss the CUDA kernel functions and the organizations of threads generated by the invocation of kernel functions. In CUDA, a kernel function specifies the code to be executed by all threads of a parallel phase. Since all threads of a parallel phase execute the same code, CUDA programming is an instance of the well-known Single-Program Multiple-Data (SPMD) [algorithms:98:crc] parallel programming style, a popular programming style for massively parallel computing systems.

Figure 2.8 shows the kernel function for matrix multiplication. The syntax is ANSI C with some notable extensions. First, there is a CUDA specific keyword "__global__" in front of the declaration of MatrixMulKernel(). This keyword indicates that the function is a kernel and that it can be called from a host functions to generate a grid of threads.

The second notable extension to ANSI C is the keywords "threadIdx.x" and "threadIdx.y" that refer to the thread indices of a thread. Note that all threads execute the same kernel code. There needs to be a mechanism to allow them to distinguish themselves and direct themselves towards the particular parts of the data structure they are designated to work on. These keywords allow a thread to access the hardware registers associated with it at runtime that provides the identity to the thread. For simplicity, we will refer to a thread as $\text{Thread}_{\text{threadIdx.x, threadIdx.y}}$. Note that the indices reflect a multi-dimensional organization for the threads. We will come back to this point soon.

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
       float Mdelement = Md[ty * Md.width + k];
       float Ndelement = Nd[k * Nd.width + tx];
       Pvalue += Mdelement * Ndelement;
     }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

Figure 2.8 The Matrix Multiplication Kernel Function.

In Figure 2.8, each thread uses the two indices to identify the row of Md and the column of Nd to perform dot product operation in the *for* loop and to select the Pd element that it is responsible for. It does so by calculating the starting positions in the input matrices based on their unique block and thread coordinates. They then iterate through a loop to calculate the result, and store it to memory. For example, $Thread_{2,3}$ will perform a dot product between row 2 of Md and column 3 of Nd and write the result into element (2,3) of Pd. This way, the threads collectively generate all the elements of the Pd matrix.

When a kernel is invoked, *or launched*, it is executed as *grid* of parallel threads. In Figure 2.9, the launch of Kernel 1 creates Grid 1. Each CUDA thread grid typically comprises thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example each element of a large array might be computed in a separate thread.

Threads in a grid are organized into a two-level hierarchy, as illustrated in Figure 2.9. For simplicity, the number of threads shown in Figure 2.9 is set to be small. In reality, a grid will typically consist of many more threads. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. In Figure 2.9, Grid 1 consists of 6 thread blocks that are organized into a 2x3 two-dimensional array of threads. Each thread block has a unique two dimensional coordinate given by the CUDA specific keywords blockIdx.x and blockIdx.y. All thread blocks must have the same number of threads organized in the same manner. For simplicity, we assume that the kernel

in Figure 2.8 is launched with only one thread block. It will become clear soon that a practical kernel will create much large number of thread blocks.

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

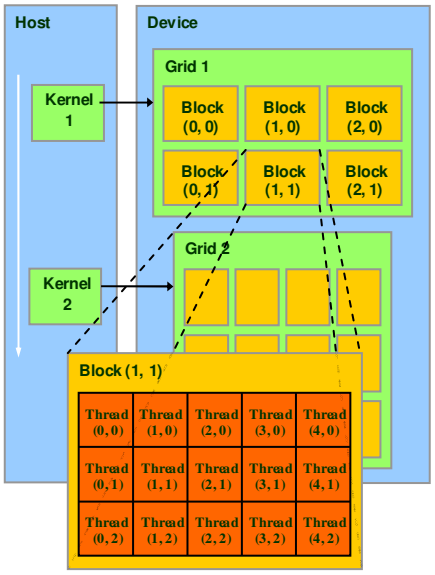

Figure 2.9 CUDA Thread Organization.

Courtesy: NDVIA

Each thread block is in turn organized as a three dimensional array of threads with a total size of up to 512 threads. The coordinates of threads in a block are uniquely defined by three thread indices: threadIdx.x, threadIdx.y, and threadIdx.z. Not all applications will use all the three dimensions of a thread block. In Figure 2.9, each thread block uses only two of the dimensions and is organized into a 3x5 array of threads. This gives Grid 1 a total of 15*6=90 threads. This is obviously a toy example.

In the matrix multiplication example, a grid is invoked to compute the product matrix. The code in Figure 2.8 can use only one thread block organized as a 2-dimensional array of threads in the grid. Since a thread block can have only up to 512 threads and each thread is to calculate one element of the product matrix, the code can only calculate a product matrix of up to 512 elements. This is obviously not acceptable. As we explained before, the product matrix needs to have millions of elements in order to have sufficient amount of data parallelism to benefit from execution on a device. We will come back to this point in Chapter [CUDA threading model] and discuss the use of multiple blocks.


```
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
```

Figure 2.10 Example of host code that launches a kernel.

When the host code invokes a kernel, it sets the grid and thread block dimensions by passing them as parameters. This is illustrated in Figure 2.10. Two structures of type *dim3* are declared: the first is for blocks, which are defined as 16x16 groups of threads. There is a thread computing each element of the result matrix. The final line of code invokes the kernel. The special syntax between the name of the kernel function and the traditional C parameters of the function is a CUDA extension to ANSI C. It provides the dimensions of grids in terms of number of blocks and the dimensions of blocks in terms of number of threads.

2.6 Summary

We have now finished an overview tour of the CUDA programming model. The matrix multiplication program developed through the chapter is a fully functional CUDA program. You can now compile the code and run the code using the CUDA runtime system. In the next few chapters, we will give more complete description of each of the main aspects of CUDA and begin to learn about the techniques for writing high performance CUDA applications.