Chapter 3

CUDA Threads

Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA. As we explained in Chapter 2, launching a CUDA kernel creates a grid of threads that all execute the kernel function. That is, the kernel function specifies the statements that are executed by each individual thread created when the kernel is launched at run-time. This chapter presents more details on the organization, resource assignment, and scheduling of threads in a grid. A CUDA programmer who understands these details are well equipped to writing and understanding efficiency CUDA applications.

3.1. CUDA Thread Organization

Since all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates, called blockId and threadId, assigned to them by the CUDA runtime system. The blockId and threadId appear as built-in variables that are initialized by the run-time system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the blockId and threadId variables return the appropriate values that form coordinates of the thread.

At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch. For the purpose of our discussions, we will refer to the special parameters that specify the number of blocks in each dimension as a `struct` variable gridDim, with gridDim.x specifying the number of blocks in the x dimension and gridDim.y the y dimension. The values of gridDim.x and gridDim.y can be anywhere between 1 and 65,536. The values of gridDim.x and gridDim.y can be supplied by run-time variables at kernel launch time. Once a kernel is launched, its dimensions cannot change in the current CUDA run-time implementation. All threads in a block share the same blockId values. The blockId.x value ranges between 0 and gridDim.x-1 and the blockId.y value between 0 and gridDim.y-1.

Figure 3.1 shows a small grid that consists of four blocks organized into a 2X2 array. Each block in the array is labeled with (blockId.x, blockId.y). For example, Block(1,0) has its blockId.x=1 and blockId.y=0. It should be clear to the reader that the grid was generated by launching the kernel with both grdiDim.x and gridDim.y set to 2. We will show the code that does so momentarily.
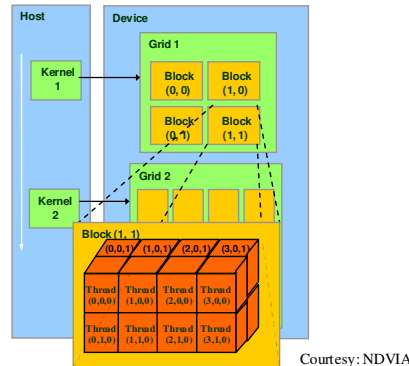

Figure 3.1 An example of CUDA Thread Organization

At the bottom level of the hierarchy, all blocks of a grid are organized into a three-dimensional array of threads. All blocks in a grid have the same dimensions. Each threadId consists of three components: the x coordinate threadId.x, the y coordinate threadId.y, and the z coordinate threaded.z. The number of threads in each dimension of a block is specified by the second special parameter given at the kernel launch. For the purpose of our discussion, we refer to the second special parameter as blockDim variable given at the launch of a kernel. The total size of a block is limited at 512 threads, with total flexibility of distributing these elements into the three dimensions as long as the total number of threads does not exceed 512. For example, (512,1,1), (8, 16, 2) and (16,16, 2) are all allowable dimensions but (32, 32, 1) is not allowable since the total number of threads would be 1024.

Figure 3.1 also illustrates the organization of threads within a block. Since all blocks within a grid have the same dimensions, we only need to show one of them. In this example, each block is organized into 4X2X2 arrays of threads. Figure 3.1 expands blcck(1,1) by showing this organization of all 16 threads in block(1,1). For example, thread(2,1,0) has its threadId.x=2, threadId.y=1, and threadId.z=0. Note that in this example, we have 4 blocks of 16 threads each, with a grand total of 64 threads in the grid. Note that we use these small numbers to keep the illustration simple. Typical CUDA grids contain thousands to millions of threads.

We now come back to the point that the exact organization of a grid is determined by the special parameters provided during kernel launch. The first special parameter of a kernel

launch specifies the dimensions of the grid in terms of number of blocks. The second specifies the dimensions of each block in terms of number of threads. Each such parameter is a dim3 type, which is essentially a `struct` with three fields. Since grids are 2D array of block dimensions, the third field of the grid dimension parameter is ignored; one should set it to one for clarity. At this point, the reader should be able to tell that the thread organization shown in Figure 3.1 is created through a kernel launch of the following form:

> **dim3 dimBlock(4, 2, 2);**
> **dim3 dimGrid(2, 2, 1);**
> **KernelFunction<<<dimGrid, dimBlock>>>(…);**

The first two statements initialize the dimension parameters. The third statement is the actual kernel launch.

In situations where a kernel does not need one of the dimensions given, the programmer can simply initialize that field of the dimension parameter to 1. For example, if a kernel is to have a 1D grid of 100 blocks and each block has 16X16 threads, the kernel launch sequence can be done as follows:

> **dim3 dimBlock(16, 16, 1);**
> **dim3 dimGrid(100, 1, 1);**
> **KernelFunction<<<dimGrid, dimBlock>>>(…);**

Note that the dimension variables can be given as contents of variables. They do not need to be compile-time constants.

3.2. More on BlockId and ThreadId

From the programmer's point of view, the main functionality of blockId and threaded variables is to provide threads with a means to distinguish among themselves when executing the same kernel. One common usage for threadId and blockId is to determine the area of data that a thread is to work on. This was exemplified by the simple matrix multiplication code in Figure 2.8, where the dot product loop uses threadId.x and threadId.y to identify the row of Md and column of Nd to work on. We will now cover more sophisticated usage of these variables.

One limitation of the simple code in Figure 2.8 is that it can only handle matrices of up to 16 elements in each dimension. This limitation comes from the fact the code uses only one block of threads to calculate Pd. Since the kernel function does not use blockId, all threads implicitly belong to the same block. With each thread calculating one element of Pd, we can calculate up to 512 Pd elements with the code. For square matrices, we are limited to 16X16 since 32X32 results in more than 512 Pd elements.

In order to accommodate larger matrices, we need to use multiple blocks. Figure 3.2 shows the basic idea of such an approach. Conceptually, we break Pd into square tiles. All the Pd elements of a tile are computed by a block of threads. By keeping the dimensions of these Pd tiles small, we keep the total number of threads in each block under 512, the maximal allowable block size. In Figure 3.2, for simplicity, we abbreviate threaded.x and threaded.y into tx and ty. Similarly, we abbreviate blockId.x and blockId.y into bx and by.

Figure 3.2 Matrix Multiplication using multiple blocks by using blockId and tiling Pd.
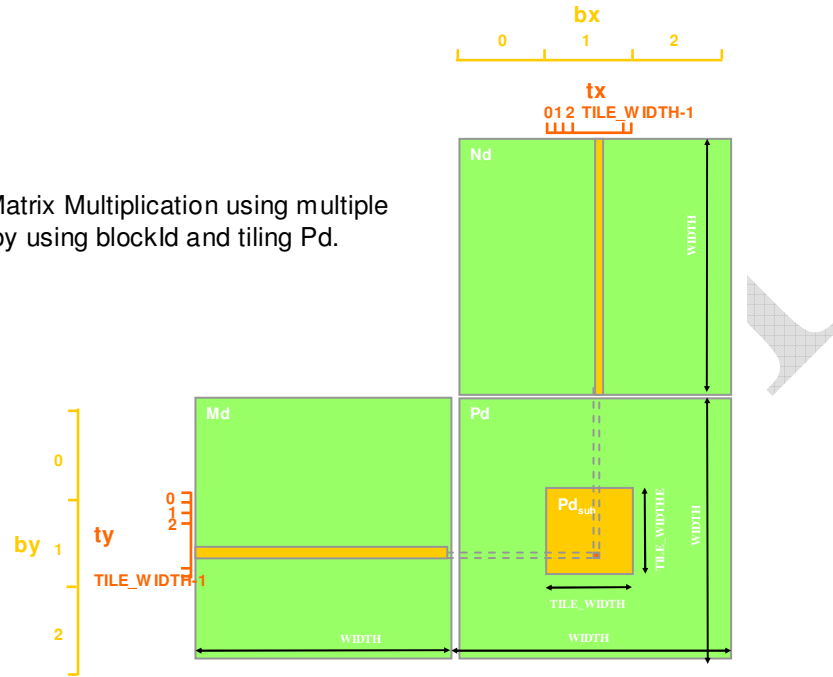
Each thread still calculates one Pd element. The difference is that it needs to use its blockId values to identify the tile that contains its element before it uses its threadId values to identify its element inside the tile. That is, each thread now uses both threadId and blockId to identify the Pd element to work on. This is portrayed in Figure 3.2 with the blockId and threadId values of threads calculating the Pd elements marked in both x and y dimensions. All threads calculating the Pd elements within a tile have the same blockId values. Assume that the dimensions of a block are specified by variable tile_width. Each dimension of Pd is now divided into sections of tile_width elements each, as shown on the left and top edges of Figure 3.2. Each block handles such a section. Thus, a thread can find the x index of its Pd element as (bx*tile_width + tx) and the y index as (by*tile + ty). That is, thread(tx,ty) in block(bx,by) is to calculate Pd[bx*tile_width+tx][by*tile_didtch+ty].
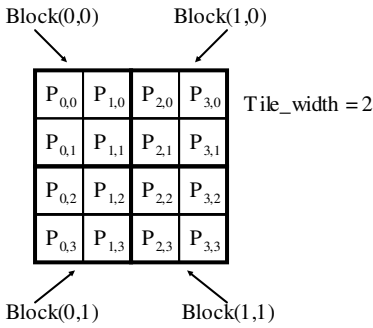
Figure 3.3 A simple example of using multiple blocks to calculate Pd

Figure 3.3 shows an example of using multiple blocks to calculate Pd. For simplicity, we use a very small tile_width value (2). The Pd matrix is now divided into 2X2 tiles. Each dimension of Pd is now divided into sections of two elements. Each block now needs to calculate four Pd elements. We can do so by creating blocks of four threads, organized into a 2X2 array each; with each thread calculating one Pd element. In the example, thread(0,0) of block(0,0) calculates Pd[0][0] whereas thread(0,0) of block(1,0) calculates Pd[2][0]. It is easy to verify that one can identify the Pd element calculated by thread(0,0) of block(1,0) with the formula given above: P[bx*tile_width+tx][by*tile_width+ty] = P[1*2+0][0*2+0] = P[2][0]. The reader should work through the index derivation for as many threads as it takes to become comfortable with the concept.

Once we identified the indices for the Pd element calculated by a thread, we also have identified the row (y) index of Md and the column (x) index of Nd as input values needed for calculating the Pd element. As shown in Figure 3.2, the row index of Md used by thread(tx,ty) of block(bx,by) is (by*tile_width+ty). The column index of Nd used by the same thread is (bx*tile_width+tx). We are now ready to revise the kernel of Figure 2.8 into a version that uses multiple blocks to calculate Pd. Figure 3.4 shows such a revised matrix multiplication kernel function.

5

```
global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockId.y * TILE_WIDTH + threadId.y;
// Calculate the column idenx of Pd and N
Int Col = blockId.x * TILE_WIDTH + threadId.x;

Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row][k] * Nd[k][Col];

Pd[Row][Col] = Pvalue;
}
```

Figure 3.4 Revised Matrix Multiplication Kernel using multiple blocks.

In Figure 3.4, each thread uses its blockId and threadId values to identify the row index and the column index of the Pd element it is responsible for. It then performs a dot product on the row of Md and column of Nd to generate the value of Pd element. It eventually writes the Pd value to the appropriate global memory location. Note that this kernel can handle matrices of up to 16*65,536 elements in each dimension. In the unlikely situation where matrices larger than this new limit are to be multiplied, one can divide up the Pd matrix into sub-matrices whose size is permissible by the kernel. Each sub-matrix would still have ample number of blocks (65,536*65,536) to fully utilize parallel execution resources of any processors in the foreseeable future.

3.3. Transparent Scalability

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function syncthreads(). When a kernel function calls syncthreads(), all threads in a block will be held at the calling location until everyone else in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before they all move on to the next phase.

Barrier synchronization is a popular method of coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities of multiple persons. For example, assume that four friends go to a shopping mall in a car. They can all go to different stores to buy their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit the stores for their clothes. However, barrier synchronization is needed before they leave the mall. They have to wait until all four friends have returned to the car before they can leave. Without the barrier synchronization, one or more persons can be left in the mall when the car leaves, which can seriously damage the friendship!

The ability of synchronizing with each other also imposes execution constraints on threads within a block. These threads should execute in close time proximity with each other to avoid excessively long waiting times. CUDA run-time systems satisfy this constraint by assigning execution resources to all threads in a block as a unit. That is, when a thread is of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive waiting time during barrier synchronization.

This leads us to a major tradeoff in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, CUDA run-time system does not need to deal with any constraint while executing different blocks. That is, blocks can execute in any order relative to each other since none of them need to wait for each other. This flexibility enables scalable implementations as shown in Figure 3.5. In a low-cost implementation with only few execution resources, one can execute a small number of blocks at the same time, shown as executing two blocks a time on the left hand side of Figure 3.5. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time, shown as four blocks a time on the right hand side of Figure 3.5. The ability to execute the same application code at a wide range of speeds allows one to produce a wide range of implementations according the cost, power, and performance requirements of particular market segments. For example, one can produce a mobile processor that execute an application slowly but at extremely low power consumption and a desktop processor that executes the same application at a higher speed while consuming more power. Both execute exactly the same application program with no change to the code. The ability to execute the same application code at different speeds is referred to as transparent scalability, which reduces the burden on application developers and improves the usability of applications.
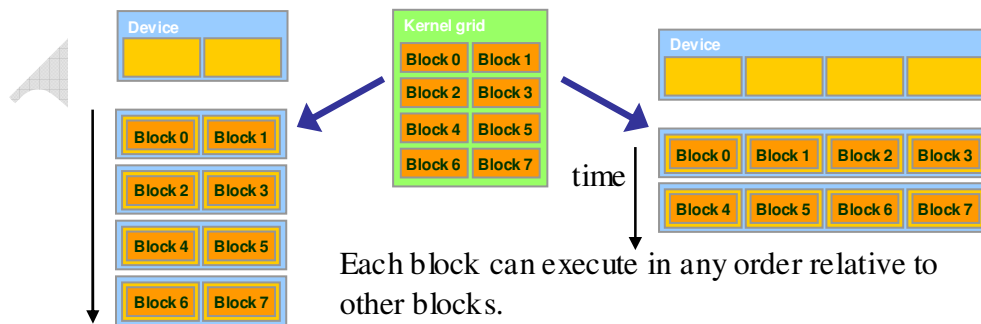

Figure 3.5 Lack of synchronization across blocks enables transparent scalability of CUDA programs

3.4. Thread Assignment

Once a kernel is launched, the CUDA run-time system generates the corresponding grid of threads. These threads are assigned to execution resources on a block by block basis. In the GeForce-8 series hardware, the execution resources are organized into Streaming Multiprocessors. For example, the GeForce 8800GTX implementation has 16 Streaming Multiprocessors, two of which are shown in Figure 3.6. Up to 8 blocks can be assigned to each SM in the GeForce 8800GTX design as long as there are enough resources to satisfy the needs of all the blocks. In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA runtime automatically reduces the number of blocks assigned to each Streaming Multiprocessor until the resource usage is under the limit. With 16 Streaming Multiprocessors in a GeForce 8800 GTX processor, up to 128 blocks can be simultaneously assigned to Streaming Multiprocessors. Most grids contain much more than 128 blocks. The run-time system maintains a list of blocks that need to execute and assigns new blocks to Streaming Multiprocessors as they complete the execution of blocks previously assigned to them.
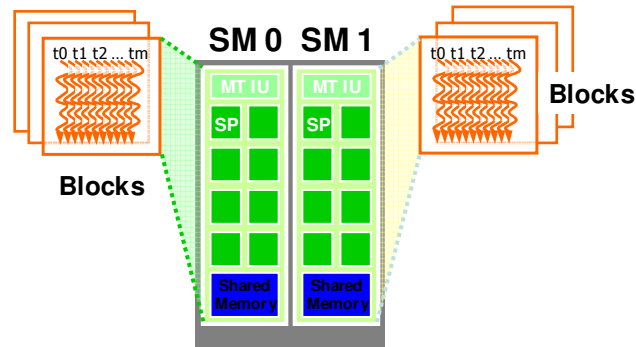

Figure 3.6 Thread Assignment in GeForce-8 Series GPU Devices.

One of the Streaming Multiprocessor (SM) resource limitations is the number of threads that can be simultaneously tracked and scheduled. It takes hardware resources for Streaming Multiprocessors to maintain the thread and block IDs and track their execution status. In the GeForce 8800GTX design, up to 768 threads can be assigned to each SM. This could be in the form of 3 blocks of 256 threads each, 6 blocks of 128 threads each, etc. It should be obvious that 12 blocks of 64 threads each are not a viable option since each SM can only accommodate up to 8 blocks. With 16 SMs in GeForce 8800 GTX, there can be up to 12,288 threads simultaneously residing in SMs for execution.

3.5. Thread Scheduling

Thread scheduling is strictly an implementation concept and thus must be discussed in the context of specific implementations. In the GeForce 8800GTX, once a block is assigned to a Streaming Multiprocessor, it is further divided into 32-thread units called Warps. The size of warps is implementation specific and can vary from one implementation to another.

In fact, warps are not even part of the CUDA language definition. However, knowledge of the warps can be helpful in understanding and optimizing the performance of CUDA applications on GeForce-8 series processors. These warps are the unit of thread scheduling in SMs. Figure 3.7 shows the division of blocks into warps in GeForce 8800GTX. Each warp consists of 32 threads of consecutive threadId values: thread 0 through 31 form the first warp, 32 through 63 the second warp, and so on. In this example, there are three blocks, Block 1 in green, Block 2 in orange, and Block 3 in blue, all assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.
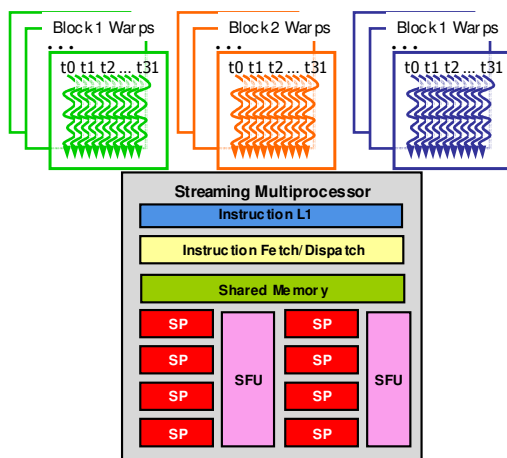

Figure 3.7 Warp-Based thread scheduling in GeForce 8800GTX.

At this point, the reader should be able to calculate the number warps that reside in a SM for a given size of blocks and a given number of blocks assigned to each SM. For example, in Figure 3.7, if each block has 256 threads, we should be able to determine the number of warps that reside in each Streaming Multiprocessor. Each block has 256/32 or 8 warps. With three blocks in each SM, we have 8*3 = 24 warps in each SM. This is in fact the maximal number of warps that can reside in each SM in GeForce 8800GTX, since there can be no more than 768 threads in each SM and this amounts to 768/32 = 24 warps.

To summarize, for the GeForce-8 series processors, there can be up to 24 warps residing in each Streaming Multiprocessor at any point in time. We should also point out that the SMs are designed such that only one of these warps will be actually executed by the hardware at any point in time. A legitimate question is why we need to have so many warps in an SM considering the fact that it executes only one of them at any point in time. The answer is that this is how these processors efficiently execute long latency operations such as access to the global memory. When an instruction executed by threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is placed into a waiting area. One of the other resident warps who are no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.
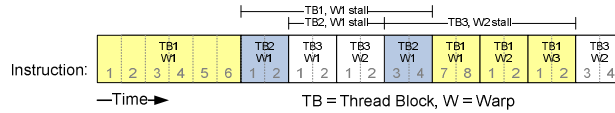
Figure 3.8 Timing example of warp-based thread scheduling

Figure 3.8 illustrates the operation of the warp-based thread scheduling scheme. It shows a snapshot of execution timeline in a Streaming Multiprocessor, where time increases from left to right. At the beginning of the snapshot, Warp 1 of Block 1 is selected for execution. Instruction 7 needs to wait for a result of a long latency operation so the warp is placed into a waiting area. Next, the scheduling hardware selects Warp 1 of Block 2 for execution. Instruction 3 needs to wait for a long latency operation so the warp is placed into a waiting area. During this time, the operation that will ultimately provide value to Instruction 7 of Warp 1 of Block 1 continue to make progress, shown as the stall time marked as "TB1, W1 stall" on top of the timeline in Figure 3.8. When the long-latency operation completes, Instruction 7 of Warp 1 or Block 1 will be ready for execution and will eventually be selected for execution, as shown in the Figure. With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long latency operations. The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as zero-overhead thread scheduling.

3.5. Summary

To summarize, special parameters at a kernel launch define the dimensions of a grid and its blocks. Unique coordinates in blockId and threadId variables allow threads of a grid to distinguish among them. It is the programmer's responsibility to use these variables in the kernel functions so that the threads can properly identify the portion of the data to process. These variables compel the programmers to organize threads and their data into hierarchical and multi-dimensional organizations.

Once a grid is launched, its blocks are assigned to Streaming Multiprocessors in arbitrary order, resulting in transparent scalability of applications. The transparent scalability comes with a limitation: threads in different blocks cannot synchronize with each other. The only safe way for threads in different blocks to synchronize with each other is to terminate the kernel and start a new kernel for the activities after the synchronization point.

Threads are assigned to SM for execution on a block-by-block basis. For GeForce-8 processors, each SM can accommodate up to 8 blocks or 768 threads, which ever becomes a limitation first. Once a block is assigned to SM, it is further partitioned into warps. At any time, the SM executes only one of its resident warps. This allows the other warps to wait for long latency operations without slowing down the overall execution throughput of the massive number of execution units.