BOOK 02 CHAPTER 4 DATA-PARALLEL EXECUTION MODEL

CUDA THREAD ORGANIZATION

- Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA.
- All CUDA threads in a grid execute the same kernel function
- These threads are organized into a two-level hierarchy: a grid consists of one or more blocks and each block in turn consists of one or more threads.
- All threads in a block share the same block index, which can be accessed as the blockIdx variable in a kernel.
- Each thread also has a thread index, which can be accessed as the threadIdx variable in a kernel.
- When a thread executes a kernel function, references to the blockIdx and threadIdx variables return the coordinates of the thread.
- The execution configuration parameters in a kernel launch statement specify the dimensions of the grid and the dimensions of each block. These dimensions are available as predefined built-in variables gridDim and blockDim in kernel functions.
- In general, a grid is a 3D array of blocks and each block is a 3D array of threads.
- The exact organization of a grid is determined by the execution configuration parameters (within <<< and >>>) of the kernel launch statement. The first execution configuration parameter specifies the dimensions of the grid in number of blocks. The second specifies the dimensions of each block in number of threads.
- For example, the following host code can be used to launch the vecAddkernel() kernel function and generate a 1D grid that consists of 128 blocks, each of which consists of 32 threads. The total number of threads in the grid is 128 3 32 5 4,096.

```
dim3 dimBlock(128, 1, 1);

dim3 dimGrid(32, 1, 1);

vecAddKernel ,, , dimGrid, dimBlock .. . (...);
```

Note that dimBlock and dimGrid are host code variables defined by the programmer.

- Once vecAddKernel() is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.
- In CUDA C, the allowed values of gridDim.x, gridDim.y, and gridDim.z range from 1 to 65,536.

- All threads in a block share the same blockIdx.x, blockIdx.y, and blockIdx.z values.
- Among all blocks, the blockIdx.x value ranges between 0 and gridDim.x-1, the blockIdx.yvalue between 0 and gridDim.y-1, and the blockIdx.z value between 0 and gridDim.z-1.

Configuration of Blocks

- Blocks are organized into 3D arrays of threads. Two-dimensional blocks can be created by setting the z dimension to 1. One-dimensional blocks can be created by setting both the y and z dimensions to 1.
- All blocks in a grid have the same dimensions.
- The number of threads in each dimension of a block is specified by the second execution configuration parameter at the kernel launch.
- Within the kernel, this configuration parameter can be accessed as the x, y, and z fields of the predefined variable blockDim.
- The total size of a block is limited to 1,024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1,024.
- For example, (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowable blockDim values, but (32, 32, 2) is not allowable since the total number of threads would exceed 1,024.
- The grid can have higher dimensionality than its blocks and vice versa.

Example

- Figure 4.1 shows a small toy example of a 2D (2, 2, 1) grid that consists of 3D (4, 2, 2) blocks.

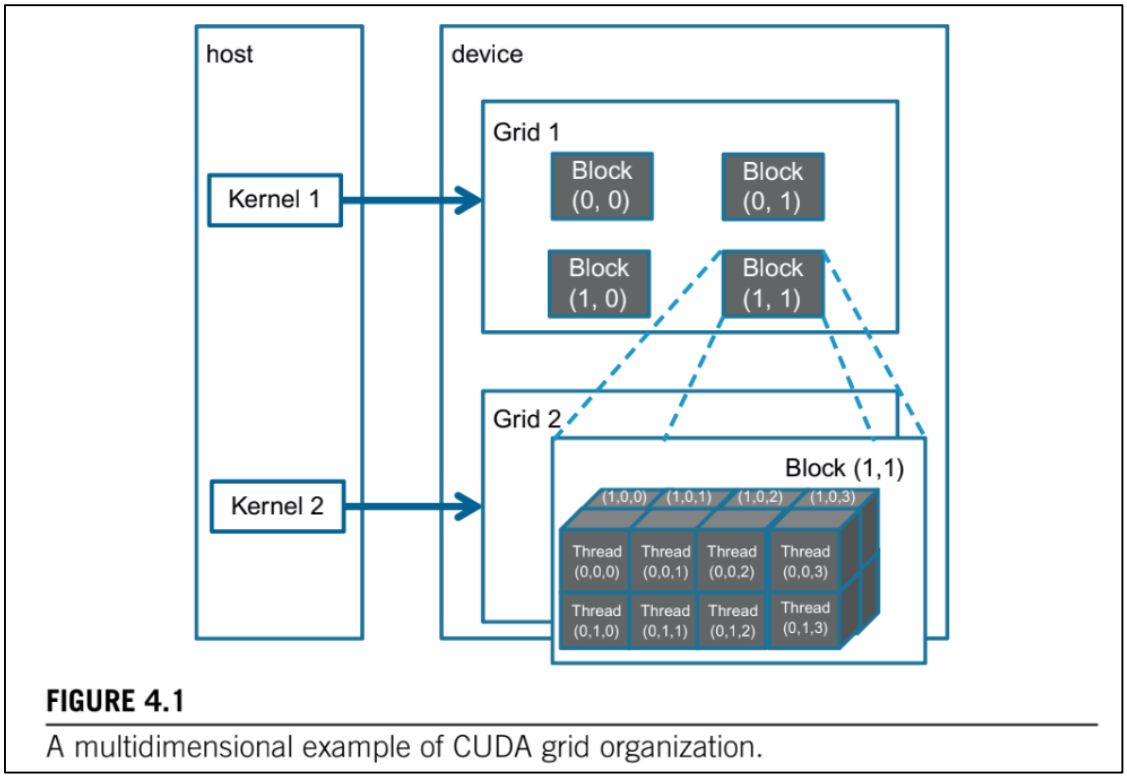

FIGURE 4.1

A multidimensional example of CUDA grid organization.

- The grid can be generated with the following host code:

  dim3 dimBlock(2, 2, 1);

  dim3 dimGrid(4, 2, 2);

  KernelFunction ,, , dimGrid, dimBlock .. . (...);

- The grid consists of four blocks organized into a 2 3 2 array. Each block in Figure 4.1 is labeled with (blockIdx.y, blockIdx.x).
- For example, block(1,0) has blockIdx.y 5 1 and blockIdx.x 5 0.
- Note that the ordering of the labels is such that the highest dimension comes first. This is reverse of the ordering used in the configuration parameters where the lowest dimension comes first. This reversed ordering for labeling threads works better when we illustrate the mapping of thread coordinates into data indexes in accessing multidimensional arrays.
- Each threadIdx also consists of three fields: the x coordinate threadId.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z.
- Figure 4.1 illustrates the organization of threads within a block. In this example, each block is organized into 4 3 2 3 2 arrays of threads. Since all blocks within a grid have the same dimensions, we only need to show one of them.

- Figure 4.1 expands block(1,1) to show its 16 threads. For example, thread(1,0,2) has threadIdx.z 5 1, threadIdx.y 5 0, and threadIdx.x 5 2. Note that in this example, we have four blocks of 16 threads each, with a grand total of 64 threads in the grid.

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Read this topic from book (Important)

SYNCHRONIZATION AND 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 every thread 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 any of them can move on to the next phase.
- Barrier synchronization is a simple and popular method of coordinating parallel activities.
- For example, assume that four friends go to a shopping mall in a car. They can all go to different stores to shop for their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit all the stores of interest. 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—the ones who finish earlier need to wait for those who finish later. Without the barrier synchronization, one or more persons can be left in the mall when the car leaves.

Execution of Barrier Synchronization

- Figure 4.11 illustrates the execution of barrier synchronization.

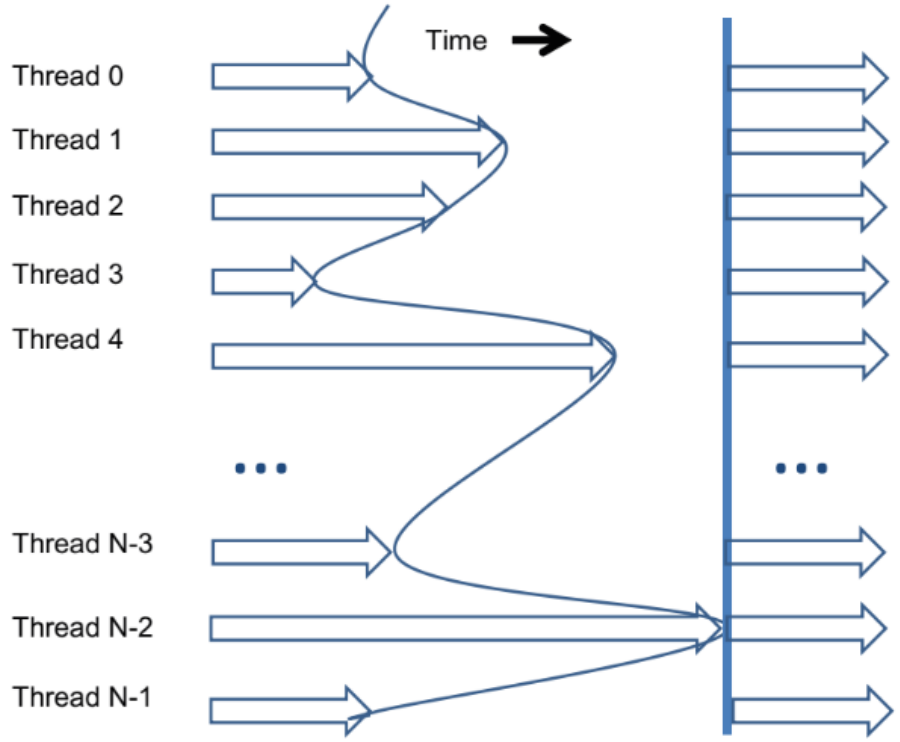

FIGURE 4.11

An example execution timing of barrier synchronization.

- There are N threads in the block.
- Time goes from left to right.
- Some of the threads reach the barrier synchronization statement early and some of them much later. The ones that reach the barrier early will wait for those that arrive late. When the latest one arrives at the barrier, everyone can continue their execution.
- In CUDA, a __syncthreads() statement, if present, must be executed by all threads in a block.
- When a __syncthread() statement is placed in an if statement, either all threads in a block execute the path that includes the __syncthreads() or none of them does.
- For an if-then-else statement, if each path has a __syncthreads() statement, either all threads in a block execute the __syncthreads() on the then path or all of them execute the else path.
- The two __syncthreads() are different barrier synchronization points. If a thread in a block executes the then path and another executes the else path, they would be waiting at different barrier synchronization points. They would end up waiting for each other forever.
- The ability to synchronize 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. In fact, one needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier. Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to wait forever.
- CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit.
- A block can begin execution only when the runtime system has secured all the resources needed for all threads in the block to complete execution.
- When a thread 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 or indefinite waiting time during barrier synchronization.
- This leads us to a major trade-off in the design of CUDA barrier synchronization.
- By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks 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 4.12, where time progresses from top to bottom.

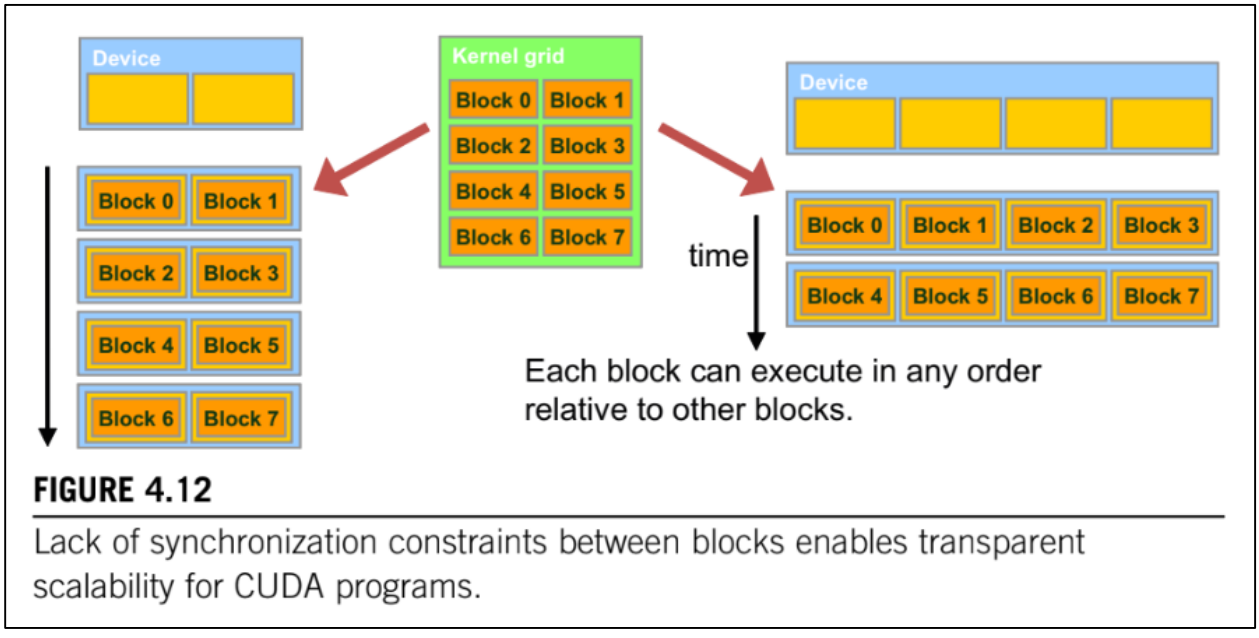

FIGURE 4.12

Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

ASSIGNING RESOURCES TO BLOCKS

- Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads.
- These threads are assigned to execution resources on a block-by-block basis.
- Figure 4.13 illustrates that multiple thread blocks can be assigned to each streaming multiprocessors (SM).

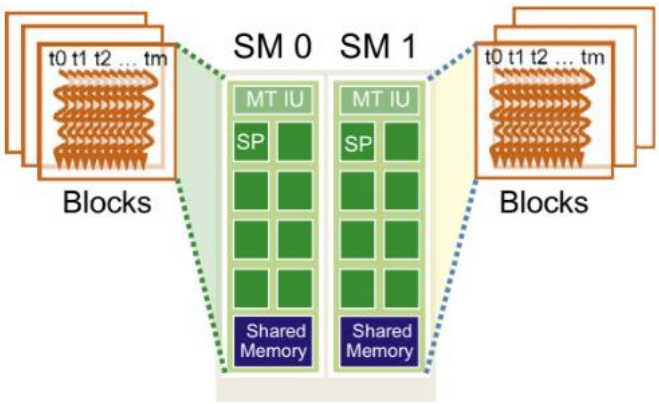

FIGURE 4.13

Thread block assignment to SMs.

- Each device has a limit on the number of blocks that can be assigned to each SM.
- For example, a CUDA device may allow up to eight blocks to be assigned to each SM.
- In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of eight blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until their combined resource usage falls under the limit.
- With a limited number of SMs and a limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in a CUDA device.
- Most grids contain many more blocks than this number.
- The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete executing the blocks previously assigned to them.
- Figure 4.13 shows an example in which three thread blocks are assigned to each SM.
- One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled.

- It takes hardware resources for SMs to maintain the thread and block indices and track their execution status.
- If a CUDA device has 30 SMs and each SM can accommodate up to1,536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution.

QUERYING DEVICE PROPERTIES

Q. How do we find out the amount of resources available?

Q. When a CUDA application executes on a system, how can it find out the number of SMs in a device and the number of threads that can be assigned to each SM?

In CUDA C, there is a built-in mechanism for host code to query the properties of the devices available in the system. The CUDA runtime system has an API function cudaGetDeviceCount() that returns the number of available CUDA devices in the system. The host code can find out the number of available CUDA devices using the following statements:

int dev_count;

cudaGetDeviceCount(&dev_count);

A modern PC system can easily have two or more CUDA devices. This is because many PC systems come with one or more “integrated” GPUs.

These GPUs are the default graphics units and provide rudimentary capabilities and hardware resources to perform minimal graphics functionalities for modern window-based user interfaces. Most CUDA applications will not perform very well on these integrated devices. This would be a reason for the host code to iterate through all the available devices, query their resources and capabilities, and choose the ones that have enough resources to execute the application with satisfactory performance.

The CUDA runtime system numbers all the available devices in the system from 0 to dev_count-1. It provides an API function **cudaGetDeviceProperties()** that returns the properties of the device of which the number is given as an argument. The built-in type **cudaDeviceProp** is a C structure with fields that represent the properties of a CUDA device. For example, we can use the following statements in the host code to iterate through the available devices and query their properties:

cudaDeviceProp dev_prop;

for (I 5 0; i , dev_count; i11) {

cudaGetDeviceProperties(&dev_prop, i); // decide if device has sufficient resources and capabilities

}

The field **dev_prop.maxThreadsPerBlock** gives the maximal number of threads allowed in a block in the queried device. Therefore, it is a good idea to query the available devices and determine which ones will allow a sufficient number of threads in each block as far as the application is concerned.

The number of SMs in the device is given in **dev_prop.multiProcessorCount**.

The clock frequency of the device is in **dev_prop.clockRate**.

The combination of the clock rate and the number of SMs gives a good indication of the hardware execution capacity of the device.

The host code can find the maximal number of threads allowed along each dimension of a block in **dev_prop.maxThreadsDim[0**] (for the x dimension), **dev_prop.maxThreadsDim[1]** (for the y dimension), and **dev_prop.maxThreadsDim[2]** (for the z dimension).

Similarly, it can find the maximal number of blocks allowed along each dimension of a grid in **dev_prop.maxGridSize[0]** (for the x dimension), **dev_prop.maxGridSize[1]** (for the y dimension), and **dev_prop.maxGridSize[2]** (for the z dimension). (A typical use of this information is to determine whether a grid can have enough threads to handle the entire data set or if some kind of iteration is needed.)