GPU Programming

Week # 15

CUDA Programming Syllabus

#3 **Week # 14**

Introduction to Data Parallelism and CUDA C: Data Parallelism, CUDA program structure, Vector Addition kernel, Device global memory and data transfer, kernel functions and threading.

#4 **Week # 15**

Data-Parallel Execution Model: Cuda thread organization, mapping threads to multidimensional, synchronization and transparent scalability, assigning resources to blocks, query device properties.

All CUDA threads in a grid execute the same kernel function and they rely on 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: a grid consists of one or more blocks and each block in turn consists of one or more threads.

GPU Grid for Kernel execution: Blocks + Threads

- When a host code launches a kernel:
  - the CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy.
  - Each grid is organized into an array of thread blocks, which will be referred to as blocks for brevity.
  - All blocks of a grid are of the same size; each block can contain up to 1,024 threads (or more based on the generation of GPU).
- The number of threads in each thread block is specified by the host code when a kernel is launched. The same kernel can be launched with different numbers of threads at different parts of the host code.
- By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with n or more threads, one can process vectors of length n.

GPU Grid for Kernel execution: Blocks + Threads

- threads in a block is available in the blockDim variable. In **Figure 3.10**, its is is 256. **In general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons.**
- Each thread in a block has a unique threadIdx value. For example, thread 0-255 in block 0, thread 256-511 in block 1, thread 212-767, etc.
- Therefore, a unique global index i is calculated as i = blockIdx.x * blockDim.x + threadIdx.x.

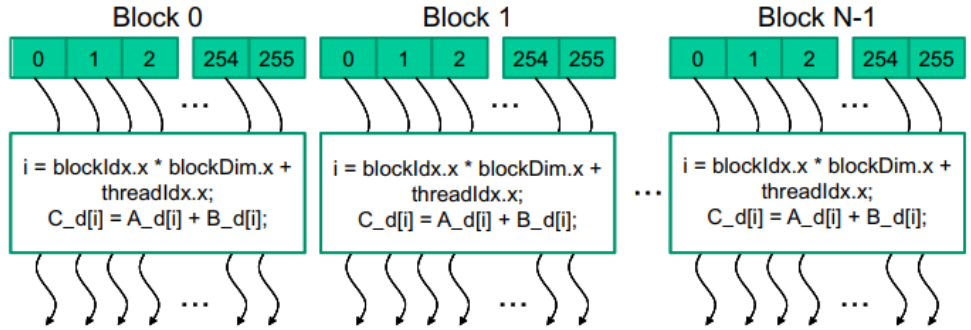
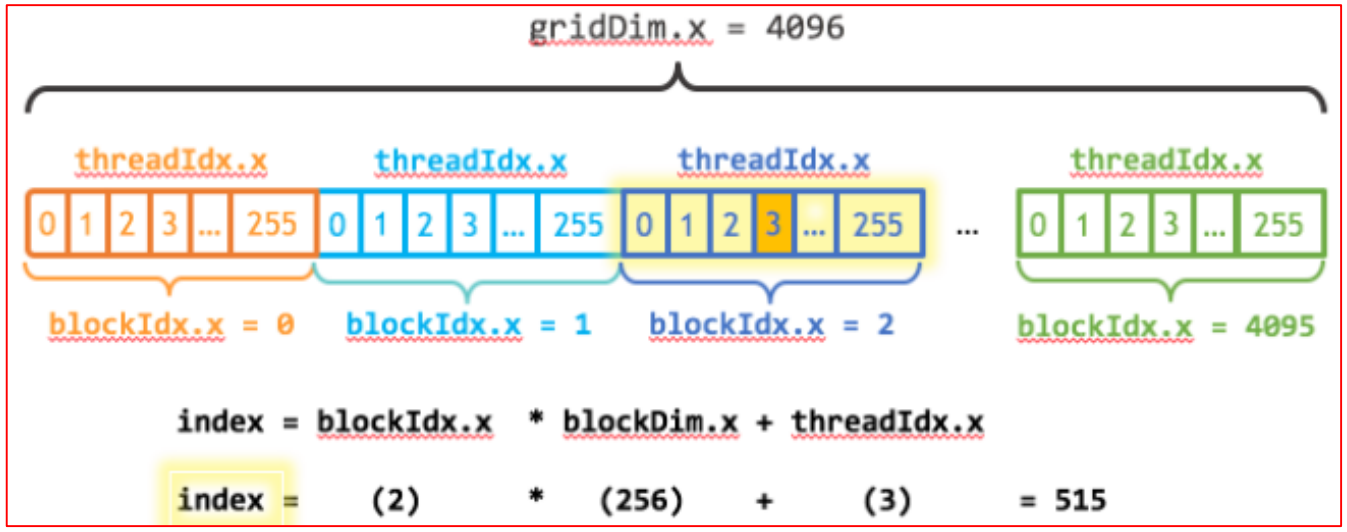


FIGURE 3.10

All threads in a grid execute the same kernel code.




```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Kernel executed as a Grid of Blocks of Threads

```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

No. of thread blocks

No. of thread per blocks

- To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly.

- For example, if we have n=1,000 elements, we would launch ceil(1,000/256.0) = ceil(3.9) = 4 thread blocks. As a result, the statement will launch 4 x 256 = 1,024 threads.

GPU Kernels in real application do much more work

- It is important to point out that the vector addition example is used for its simplicity.

- In practice, the overhead of:
  - allocating device memory,
  - Input data transfer from host to device,
  - output data transfer from device to host, and
  - de-allocating device memory

  will likely make the resulting code slower than sequential code.

- Why? … because kernel is doing small computation relative to the amount of data processed. Only one addition is performed for two floating-point input operands and one floating-point output operand.

- Kernels in real applications do much more work relative to the amount of data processed, which makes the additional overhead worthwhile.
- They also tend to keep the data in the device memory across multiple kernel invocations so that the overhead can be amortized.

Data-Parallel Execution Model

4

CHAPTER OUTLINE

This chapter presents more details on the organization, resource assignment, synchronization, and scheduling of threads in a grid. A CUDA programmer who understands these details is well equipped to express and understand the parallelism in high-performance CUDA applications.

4.1 CUDA THREAD ORGANIZATION

In general, a grid is a 3D array of blocks[1] and each block is a 3D array of threads. The programmer can choose to use fewer dimensions by setting the unused dimensions to 1. 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

For 1D or 2D grids and blocks, the unused dimension fields should be set to 1 for clarity. For example, the following host code can be used to

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(32, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

```
dim3 dog(128, 1, 1);
dim3 cat(32, 1, 1);
vecAddKernel<<<dog, cat>>>(...);
```

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

If n is equal to 1,000, the grid will consist of four blocks. If n is equal to 4,000, the grid will have 16 blocks. In each case, there will be enough threads to cover all the vector elements. Once vecAddKernel() is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

Shorthand version

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

- If n is equal to 4,000, references to gridDim.x and blockDim.x in the vectAddkernel kernel function will result in 16 and 256, respectively.

- 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.y value between 0 and gridDim.y-1, and the blockIdx.z value between 0 and gridDim.z-1.

- Blocks are organized into 3D arrays of threads. 2D can be created by setting the z dimension to 1. 1D 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.
- (32, 32, 2) is not allowable since the total number of threads would exceed 1,024.

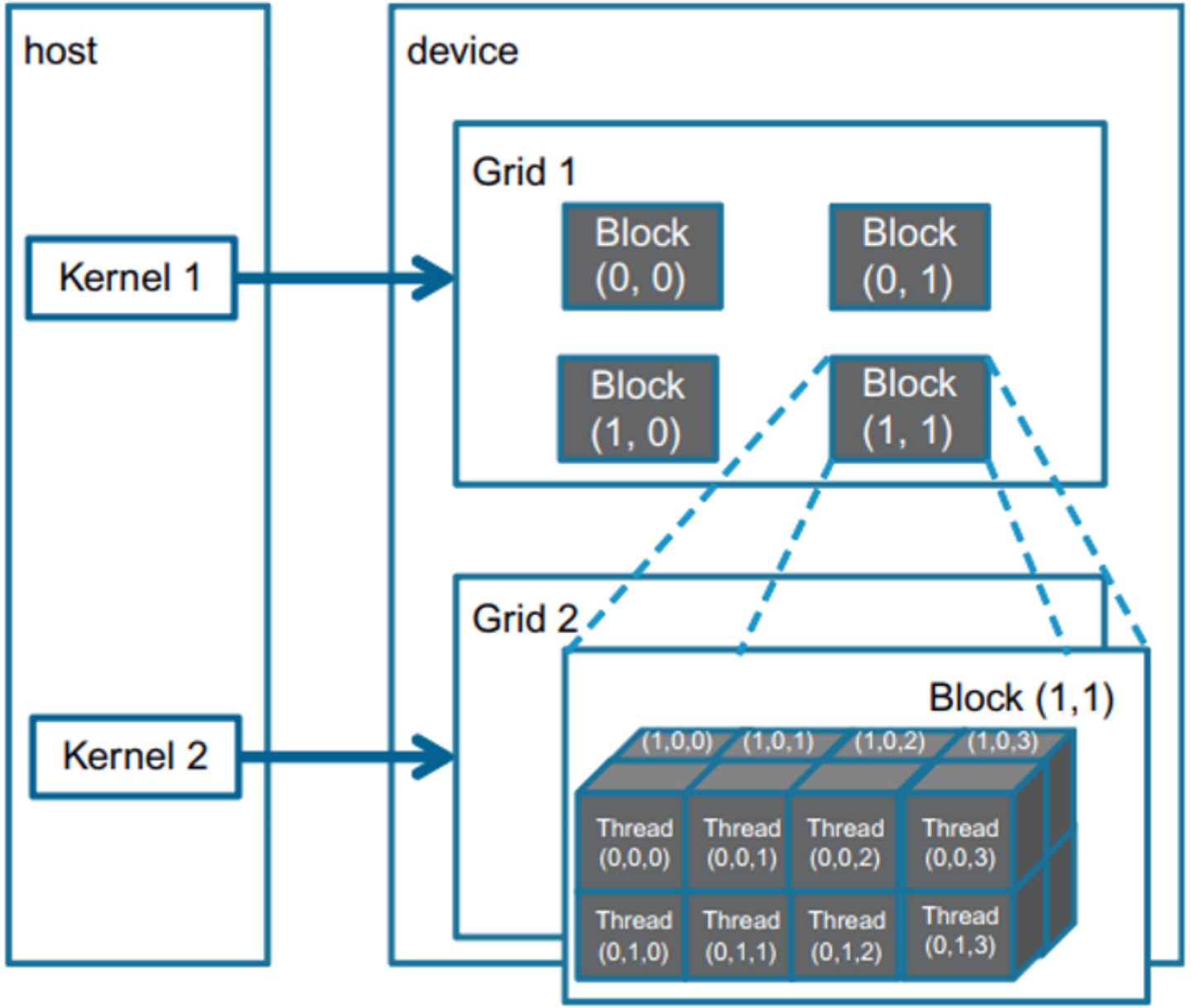


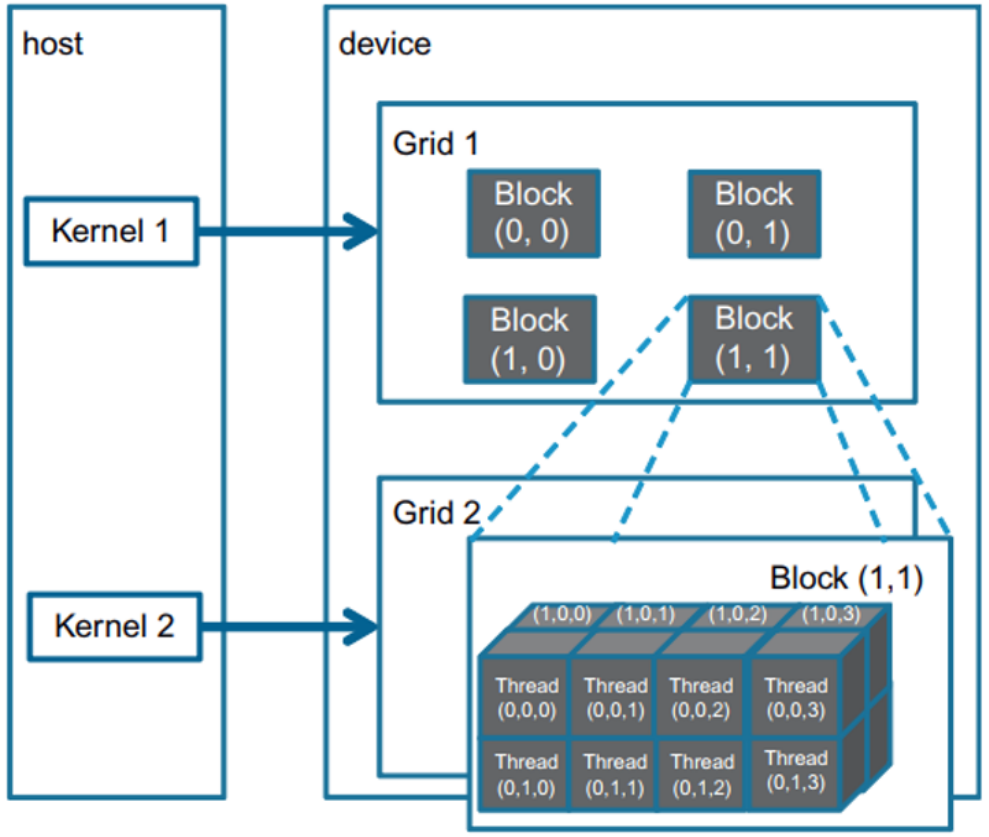
FIGURE 4.1

A multidimensional example of CUDA grid organization.

2D (2, 2, 1) grid that consists of 3D (4, 2, 2) blocks.

```
dim3 dimBlock(2, 2, 1);
dim3 dimGrid(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

- The grid consists of four blocks organized into a 2 x 2 array. Each block is labeled with (blockIdx.y, blockIdx.x).
- Each threadIdx also consists of three fields: the x coordinate threadId.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z.
- The figure expands block(1,1) to show its 16 threads. For example, thread(1,0,2) has threadIdx.z = 1, threadIdx. y = 0, and threadIdx.x = 2.

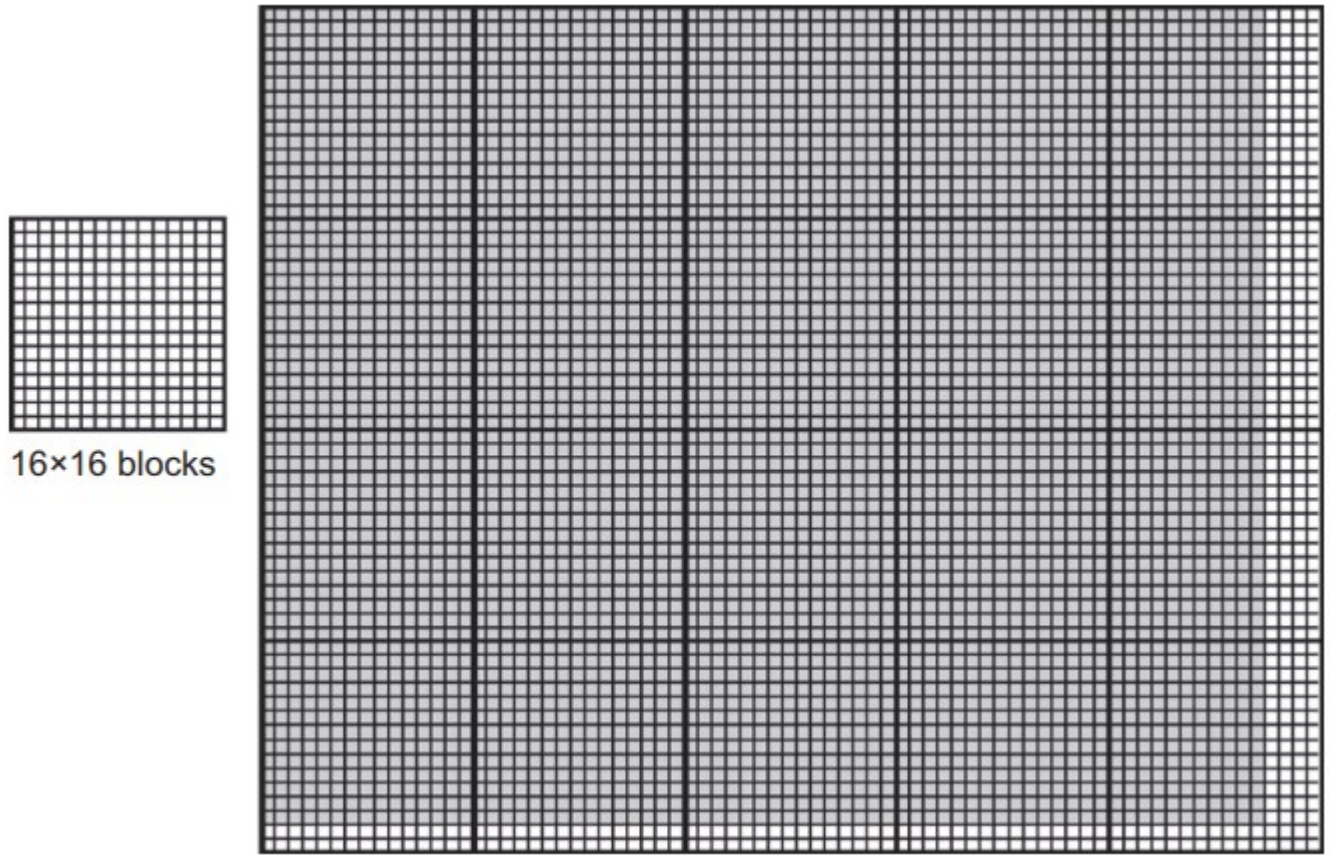


4.2 MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Thread organizations is usually based on the nature of the data. Pictures are a 2D array of pixels. It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture.
- Figure 4.2 shows such an arrangement for processing a 76 x 62 picture (76 pixels in the horizontal or x direction and 62 pixels in the vertical or y direction).
- We will need five blocks in the x direction and four blocks in the y direction, which results in 5 x 4 = 20 blocks as shown in Figure 4.2. The heavy lines mark the block boundaries. The shaded area depicts the threads that cover pixels.
- Note that we have four extra threads in the x direction and two extra threads in the y direction. That is, we will generate 80 x 64 threads to process 76 x 62 pixels.

FIGURE 4.2

Using a 2D grid to process a picture.




```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0), 1);
dim3 dimGrid(16, 16, 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

FIGURE 4.4

Source code of `pictureKernel()` showing a 2D thread mapping to a data pattern.

```
1   __global__ void PictureKernell(float *d_Pin, float *d_Pout, int n, int m) {
2     // Calculate the row # of the d_Pin and d_Pout element to process
3     int Row = blockIdx.y * blockDim.y + threadIdx.y;
4     // Calculate the column # of the d_Pin and d_Pout element to process
5     int Col = blockIdx.x * blockDim.x + threadIdx.x;
6     // each thread computes one element of d_Pout if in range
7     if ((Row < m) && (Col < n)) {
8       d_Pout[Row * n + Col] = 2 * d_Pin[Row * n + Col];
9     }
10  }
```

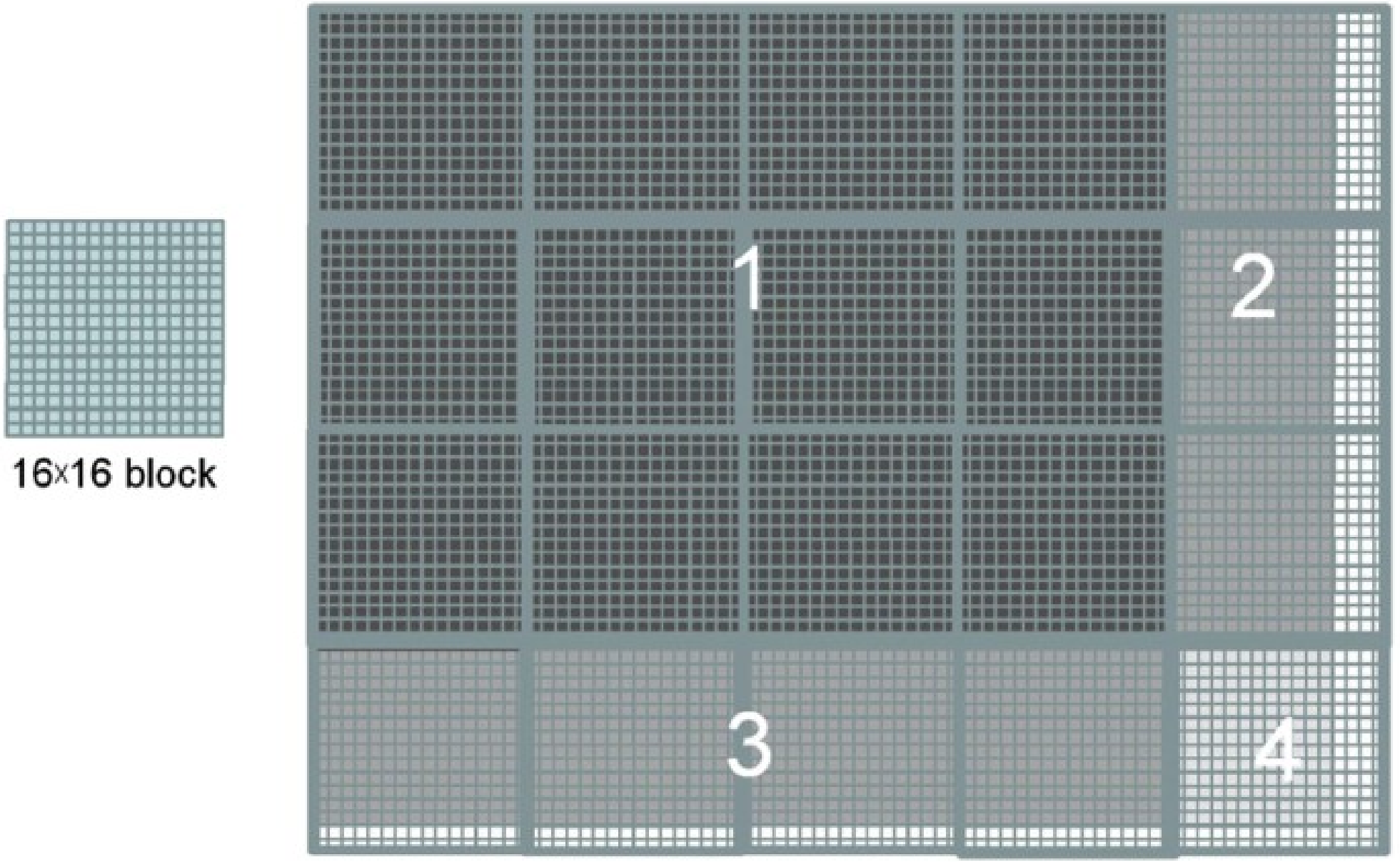


FIGURE 4.5

Covering a 76×62 picture with 16×16 blocks.

Linearize/flatten a dynamically allocated 2D array

programmers need to explicitly linearize, or "flatten," a dynamically allocated 2D array into an equivalent 1D array in the current CUDA C. Note that the newer C99 standard allows multidimensional syntax for dynamically allocated arrays. It is likely that future CUDA C versions may support multidimensional syntax for dynamically allocated arrays.

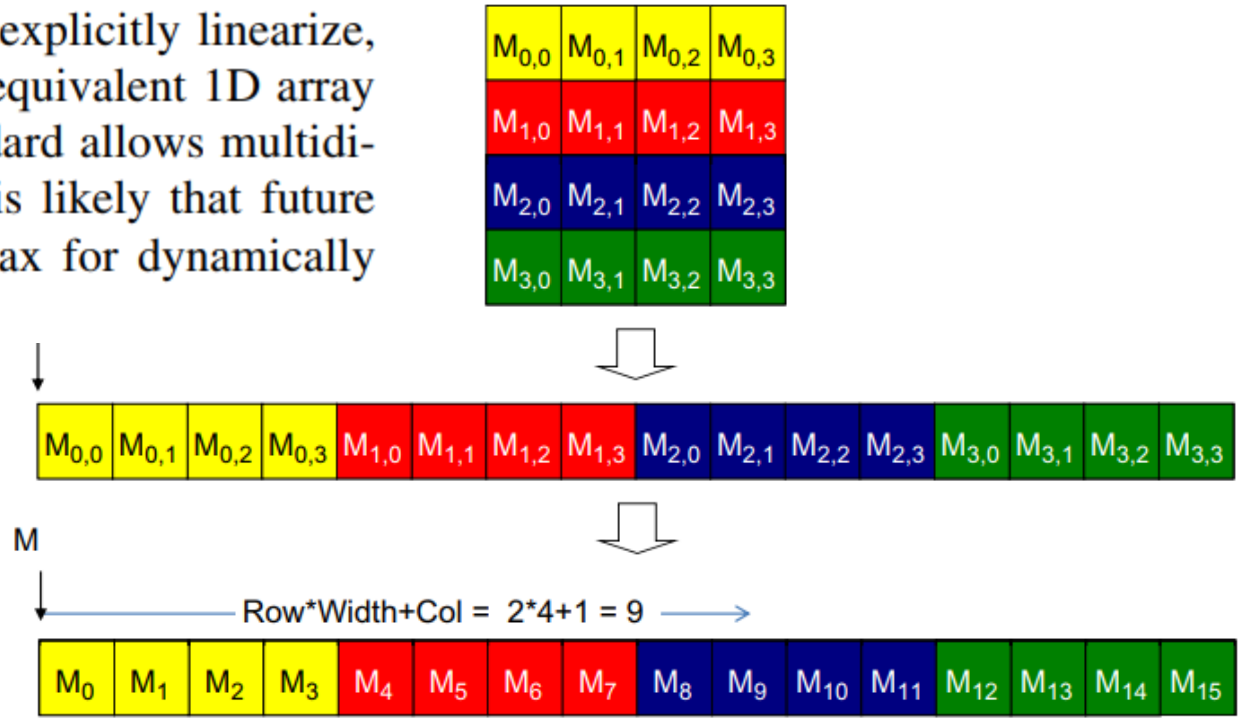


FIGURE 4.3

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression `Row*Width + Col` for an element that is in the `Row`th row and `Col`th column of an array of `Width` elements in each row.

4.4 SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- How to coordinate the execution of multiple threads?

  - 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.

__syncthreads()

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. It is the responsibility of the programmers to write their code so that these requirements are satisfied.

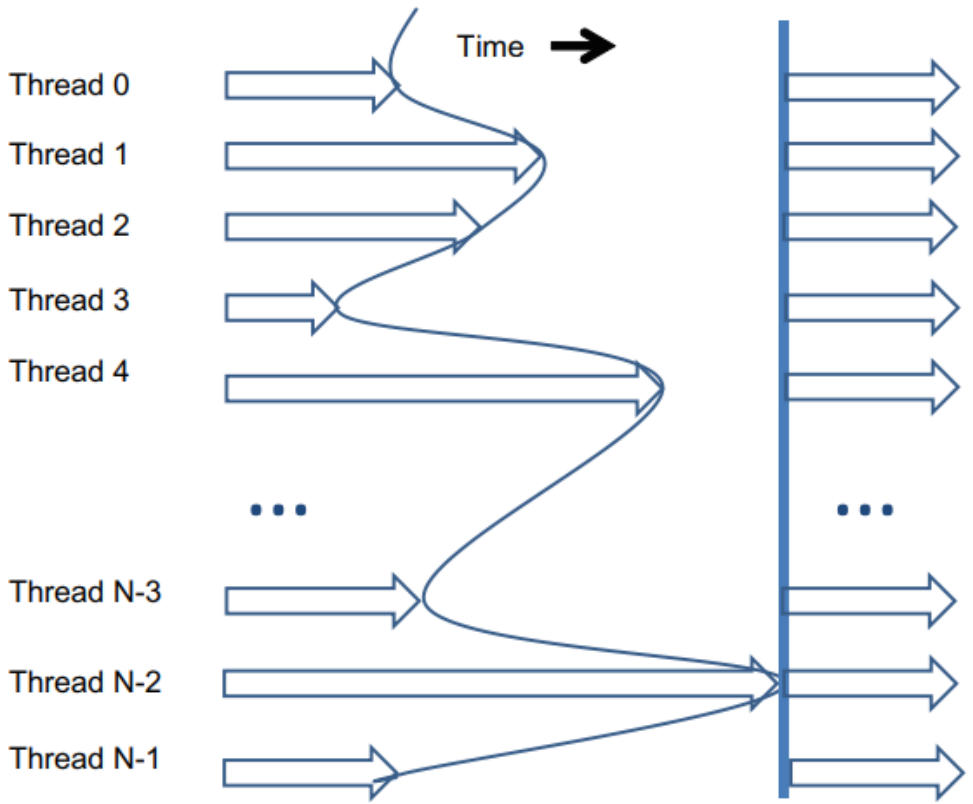


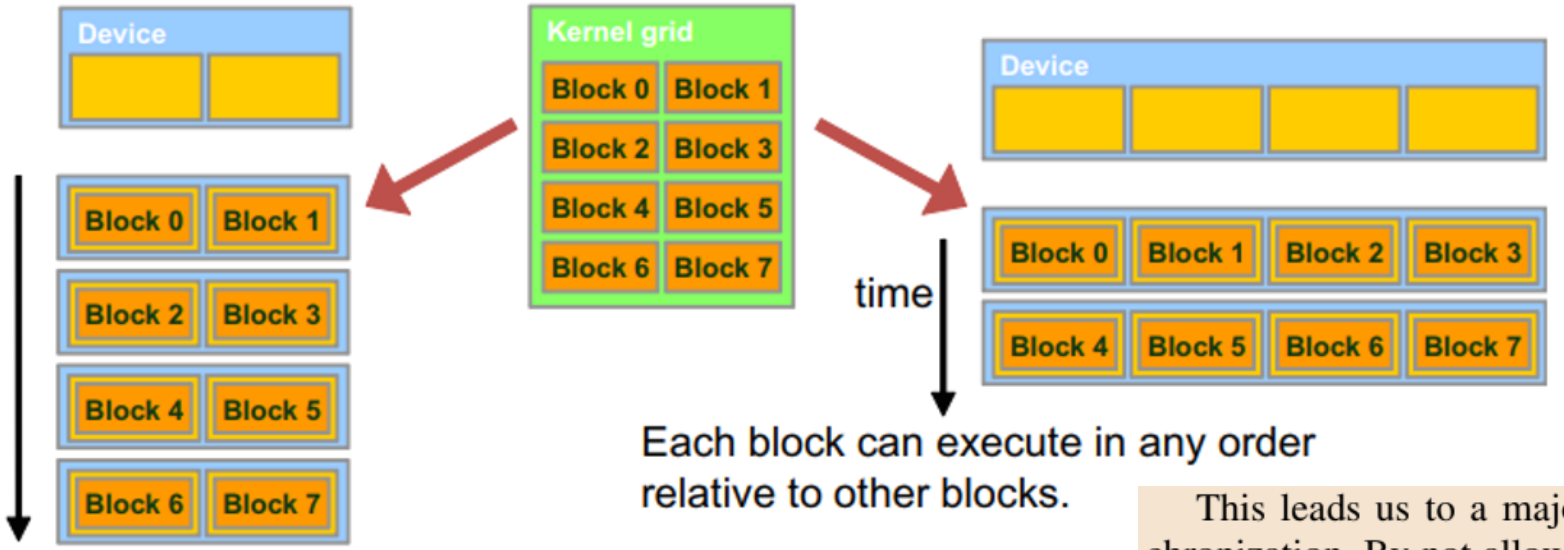
FIGURE 4.11

An example execution timing of barrier synchronization.

- CUDA runtime systems assigns 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.

FIGURE 4.12

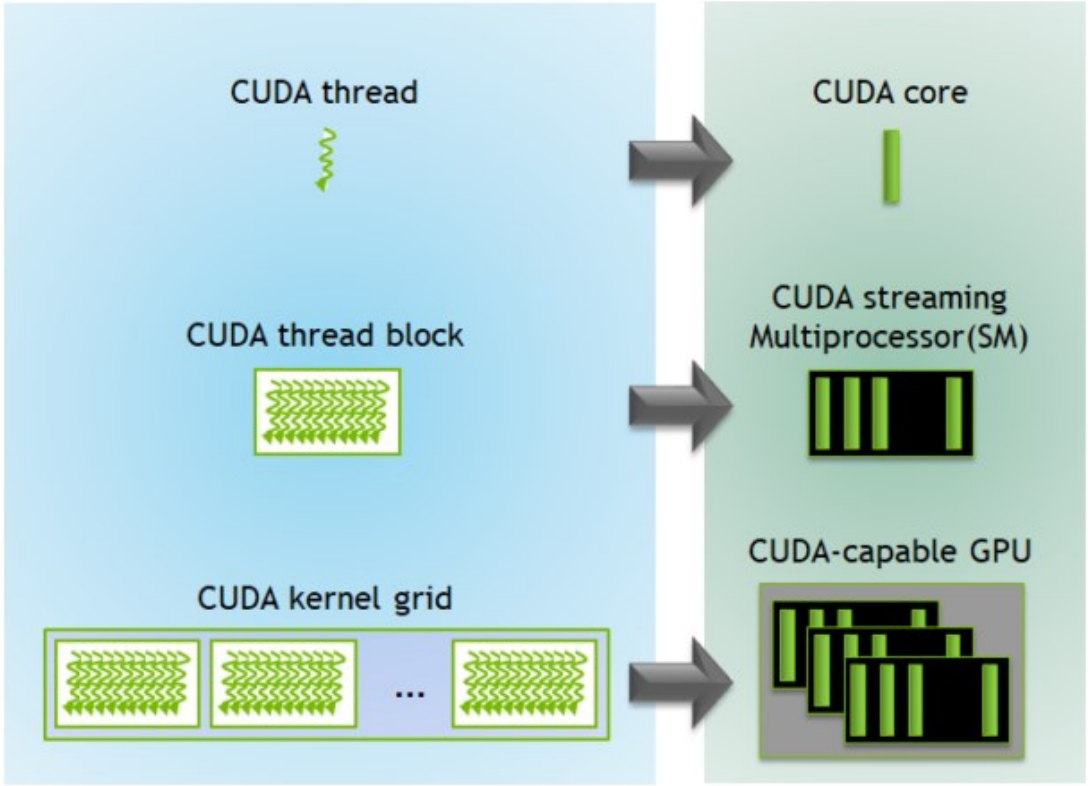
Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.



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. In a low-cost system with only a few execution resources, one can execute a small number of blocks at the same time; two blocks executing at a time is shown on the left side of Figure 4.12. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time; four blocks executing at a time is shown on the right side of Figure 4.12.

4.5 ASSIGNING RESOURCES TO BLOCKS

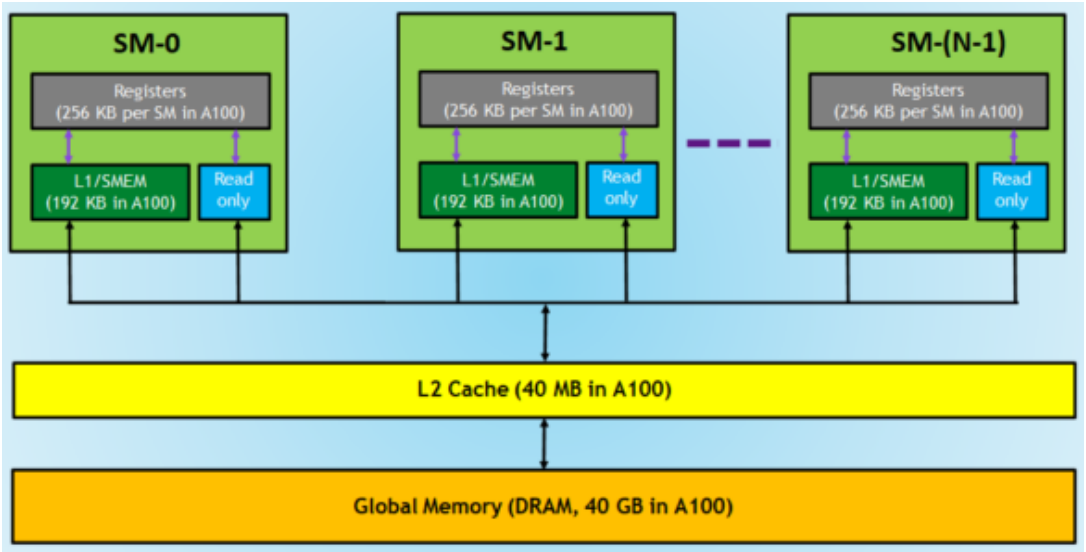
Kernel execution on GPU.



CUDA core is also called a Streaming Processor (SP)

https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Memory Hierarchy in NVIDIA GPUs



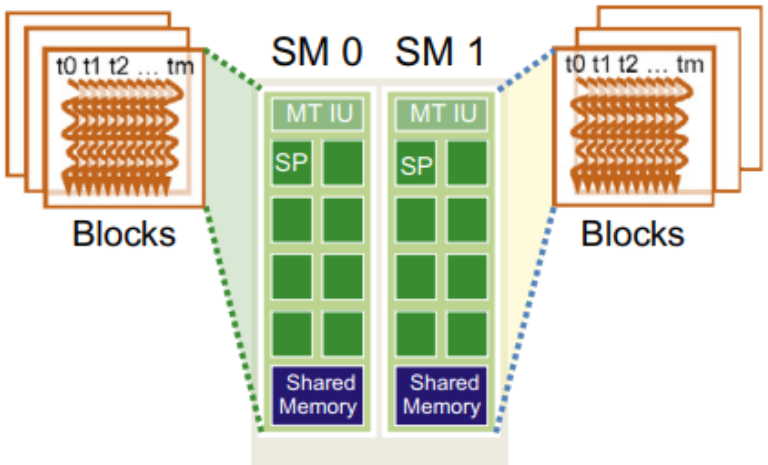
- **Registers**—These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
- **L1/Shared memory (SMEM)**—Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and shared memory. All threads in a CUDA block can share shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM..
- **Read-only memory**—Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.
- **L2 cache**—The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory. The NVIDIA A100 GPU has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.
- **Global memory**—This is the framebuffer size of the GPU and DRAM sitting in the GPU.

4.5 ASSIGNING RESOURCES TO BLOCKS

In the current generation of hardware, the execution resources are organized into streaming multiprocessors (SMs). Figure 4.13 illustrates that multiple thread blocks can be assigned to each SM. 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 numbers 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

Thread block assignment to SMs.



How to get properties from active CUDA device?

```
int nDevices;
cudaGetDeviceCount(&nDevices);

printf("Number of devices: %d\n", nDevices);

for (int i = 0; i < nDevices; i++) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, i);
```

```
Number of devices: 1
Device Number: 0
  Device name: Quadro K5200
  Memory Clock Rate (MHz): 2933
  Memory Bus Width (bits): 256
  Peak Memory Bandwidth (GB/s): 192.3
  Total global memory (Gbytes) 7.4
  Shared memory per block (Kbytes) 48.0
  minor-major: 5-3
  Warp-size: 32
  Concurrent kernels: yes
  Concurrent computation/communication: yes
```

Querying device properties on a CUDA platform can provide valuable information about the characteristics and capabilities of the GPU that can be used for optimizing and configuring your CUDA applications.

- Device Capabilities
- Memory Information
- Concurrent Execution and Multiprocessing
- Warp Size and Thread Execution
- Driver and CUDA Runtime Version
- Asynchronous Execution
- Occupancy and Performance Metrics
- Error Handling

Parallel execution on GPU using blocks & Threads

CUDA C Code

adds the elements of two arrays.
(size = 1 million elements each)

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void) {
  int N = 1<<20;
  float *x, *y;
  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f; y[i] = 2.0f;
  }
  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();
  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;
  // Free memory
  cudaFree(x);  cudaFree(y);
  return 0;
}
```

```
> nvcc add.cu -o add_cuda
> ./add_cuda
Max error: 0.000000
```

RUN

profile

```
$ nvprof ./add_cuda
==3355== NVPROF is profiling process 3355, command: ./add_cuda
Max error: 0
==3355== Profiling application: ./add_cuda
==3355== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  463.25ms         1  463.25ms  463.25ms  463.25ms  add(int, float*, float*)
```

About half a second on an NVIDIA Tesla K80 accelerator, and about the same time on an NVIDIA GeForce GT 740M.

CUDA's <<<1, 1>>> Syntax

- This is called the execution configuration, and it tells the CUDA runtime how many parallel threads to use for the launch on the GPU.

- There are two parameters here, but let's start by changing the second one: the number of threads in a thread block.
  - CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size, so 256 threads is a reasonable size to choose.
  - **add<<<1, 256>>>(N, x, y);**
  - If I run the code with only this change, it will do the computation once per thread, rather than spreading the computation across the parallel threads. To do it properly, I need to modify the kernel.

Adding > 1 threads per block in Kernel Config.

- CUDA C++ provides keywords that let kernels get the indices of the running threads. Specifically, **threadIdx.x** contains the index of the current thread within its block, and **blockDim.x** contains the number of threads in the block.

```
add<<<1, 256>>>(N, x, y);
```

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

- Time reduced from 463ms down to 2.7ms.

Time(%)	Time	Calls	Avg	Min	Max	Name
100.00%	2.7107ms	1	2.7107ms	2.7107ms	2.7107ms	add(int, float*, float*)

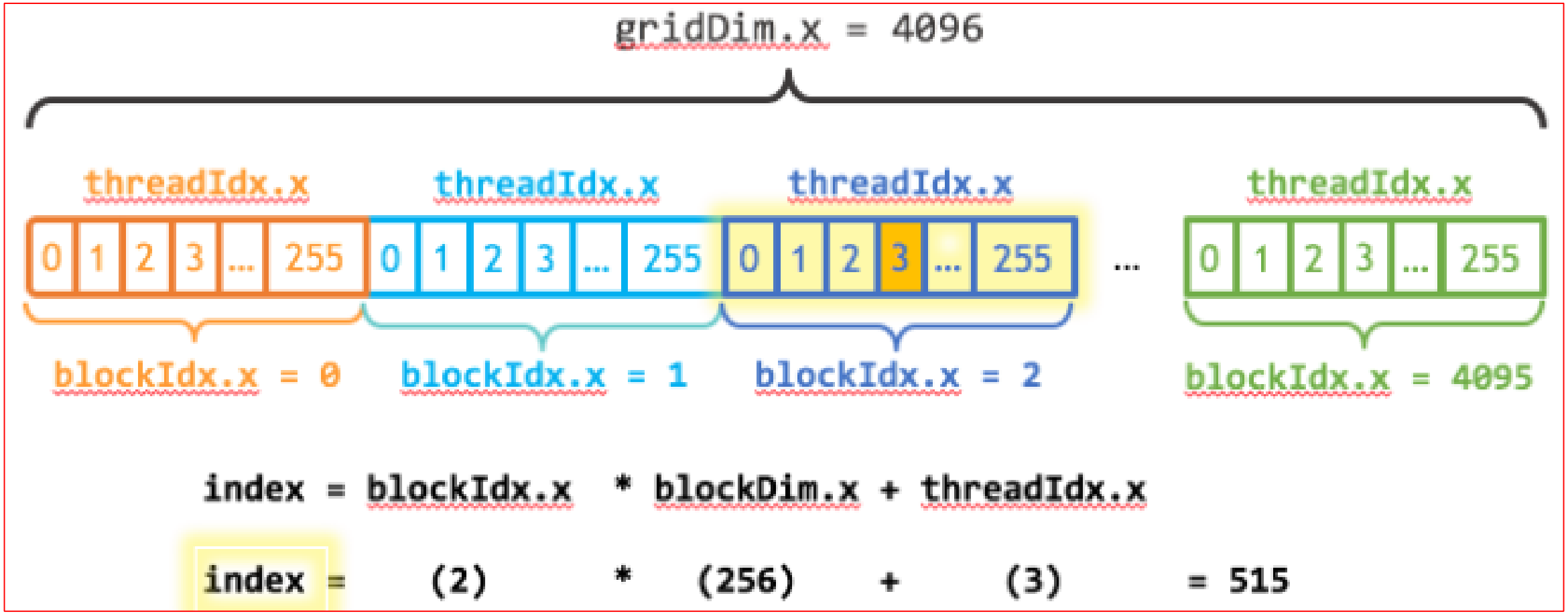
Adding > 1 blocks in Kernel Config. (1)

- CUDA GPUs have many parallel processors grouped into **Streaming Multiprocessors**, or **SMs**. Each SM can run multiple concurrent thread blocks.
  - As an example, a Tesla P100 GPU based on the Pascal GPU Architecture has 56 SMs, each capable of supporting up to 2048 active threads.

- To take full advantage of all these threads, I should launch the kernel with multiple thread blocks.

- The first parameter of the execution configuration specifies the number of thread blocks. Together, the blocks of parallel threads make up what is known as the grid. Since I have N elements to process, and 256 threads per block, I just need to calculate the number of blocks to get at least N threads. I simply divide N by the block size (being careful to round up in case N is not a multiple of blockSize).

Adding > 1 blocks in Kernel Config. (2)



Adding > 1 blocks in Kernel Config. (3)

- CUDA provides **gridDim.x**, which contains the number of blocks in the grid, and **blockIdx.x**, which contains the index of the current thread block in the grid.
  - Now indexing into an 1-D array in CUDA using blockDim.x, gridDim.x, and threadIdx.x. The idea is that each thread gets its index by computing the offset to the beginning of its block (the block index times the block size: blockIdx.x * blockDim.x) and adding the thread's index within the block (threadIdx.x).

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

- Time reduced from 2.7ms down to 94microseconds.

```
Time(%)      Time      Calls       Avg       Min       Max  Name
100.00%  94.015us          1  94.015us  94.015us  94.015us  add(int, float*, float*)
```

CUDA C Code

adds the elements of two arrays.
(size = 1 million elements each)

	Laptop (GeForce GT 750M)		Server (Tesla K80)	
Version	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411ms	30.6 MB/s	463ms	27.2 MB/s
1 CUDA Block	3.2ms	3.9 GB/s	2.7ms	4.7 GB/s
Many CUDA Blocks	0.68ms	18.5 GB/s	0.094ms	134 GB/s