CHAPTER

Scalable parallel execution 3

Mark Ebersole

CHAPTER OUTLINE

3.1 CUDA Thread Organization....43
3.2 Mapping Threads to Multidimensional Data....47
3.3 Image Blur: A More Complex Kernel....54
3.4 Synchronization and Transparent Scalability....58
3.5 Resource Assignment....60
3.6 Querying Device Properties....61
3.7 Thread Scheduling and Latency Tolerance....64
3.8 Summary....67
3.9 Exercises....67

In Chapter 2, Data parallel computing, we learned to write a simple CUDA C program that launches a kernel and a grid of threads to operate on elements in one-dimensional arrays. The kernel specifies the C statements executed by each thread. As we unleash such a massive execution activity, we need to control these activities to achieve desired results, efficiency, and speed. In this chapter, we will study important concepts involved in the control of parallel execution. We will start by learning how thread index and block index can facilitate processing multidimensional arrays. Subsequently, we will explore the concept of flexible resource assignment and the concept of occupancy. We will then advance into thread scheduling, latency tolerance, and synchronization. A CUDA programmer who masters these concepts is well-equipped to write and understand high-performance parallel applications.

3.1 CUDA THREAD ORGANIZATION

All CUDA threads in a grid execute the same kernel function; they rely on coordinates to distinguish themselves from one another and identify the appropriate portion of data to process. These threads are organized into a two-level hierarchy: a grid consists of one or more blocks, and each block consists of one or more threads. All

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00003-0

threads in a block share the same block index, which is the value of the `blockIdx` variable in a kernel. Each thread has a thread index, which can be accessed as the value of 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 the values of the variables `gridDim` and `blockDim` in kernel functions.

HIERARCHICAL ORGANIZATIONS

Similar to CUDA threads, many real-world systems are organized hierarchically. The United States telephone system is a good example. At the top level, the telephone system consists of "areas," each of which corresponds to a geographical area. All telephone lines within the same area have the same 3-digit "area code". A telephone area can be larger than a city; e.g., many counties and cities in Central Illinois are within the same telephone area and share the same area code 217. Within an area, each phone line has a seven-digit local phone number, which allows each area to have a maximum of about ten million numbers.

Each phone line can be considered as a CUDA thread, the area code as the value of `blockIdx`, and the seven-digit local number as the value of `threadIdx`. This hierarchical organization allows the system to accommodate a considerably large number of phone lines while preserving "locality" for calling the same area. When dialing a phone line in the same area, a caller only needs to dial the local number. As long as we make most of our calls within the local area, we seldom need to dial the area code. If we occasionally need to call a phone line in another area, we dial "1" and the area code, followed by the local number. (This is the reason why no local number in any area should start with "1.") The hierarchical organization of CUDA threads also offers a form of locality, which will be examined here.

In general, a grid is a three-dimensional array of blocks[1], and each block is a three-dimensional array of threads. When launching a kernel, the program needs to specify the size of the grid and blocks in each dimension. The programmer can use fewer than three dimensions *by setting the size of the unused dimensions to 1*. The exact organization of a grid is determined by the execution configuration parameters (within <<< >>>) of the kernel launch statement. The first execution configuration parameter specifies the dimensions of the grid in the number of blocks. The second specifies the dimensions of each block in the number of threads. Each such parameter is of the `dim3` type, which is a C `struct` with three unsigned integer fields: *x*, *y*, and *z*. These three fields specify the sizes of the three dimensions.

[1] Devices with compute capability less than 2.0 support grids with up to two-dimensional arrays of blocks.

To illustrate, the following host code can be used to launch the `vecAddkernel()` kernel function and generate a 1D grid that consists of 32 blocks, each of which consists of 128 threads. The total number of threads in the grid is 128*32 = 4096.

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

Note that `dimBlock` and `dimGrid` are host code variables defined by the programmer. These variables can have any legal C variable names as long as they are of the `dim3` type and the kernel launch uses the appropriate names. For instance, the following statements accomplish the same as the statements above:

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

The grid and block dimensions can also be calculated from other variables. The kernel launch in Fig. 2.15 can be written as follows:

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

The number of blocks may vary with the size of the vectors for the grid to have sufficient threads to cover all vector elements. In this example, the programmer chose to fix the block size at 256. The value of variable `n` at kernel launch time will determine the dimension of the grid. If `n` is equal to 1000, the grid will consist of four blocks. If `n` is equal to 4000, the grid will have 16 blocks. In each case, there will be enough threads to cover all of the vector elements. Once `vecAddKernel` is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

For convenience, CUDA C provides a special shortcut for launching a kernel with one-dimensional grids and blocks. Instead of `dim3` variables, arithmetic expressions can be used to specify the configuration of 1D grids and blocks. In this case, the CUDA C compiler simply takes the arithmetic expression as the *x* dimensions and assumes that the *y* and *z* dimensions are 1. Thus, the kernel launch statement is as shown in Fig. 2.15:

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

Readers familiar with the use of structures in C would realize that this "shorthand" convention for 1D configurations takes advantage of the fact that the *x* field is the first field of the `dim3` structures `gridDim(x, y, z)` and `blockDim{x, y, z)`. This shortcut allows the compiler to conveniently initialize the *x* fields of `gridDim` and `blockDim` with the values provided in the execution configuration parameters.

Within the kernel function, the *x* field of the variables `gridDim` and `blockDim` are pre-initialized according to the values of the execution configuration parameters.

If n is equal to 4000, references to gridDim.x and blockDim.x in the vectAddkernel kernel will obtain 16 and 256, respectively. Unlike the dim3 variables in the host code, the names of these variables within the kernel functions are part of the CUDA C specification and cannot be changed—i.e., gridDim and blockDim in a kernel always reflect the dimensions of the grid and the blocks.

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 blocks, the blockIdx.x value ranges from 0 to gridDim.x-1, the blockIdx.y value from 0 to gridDim.y-1, and the blockIdx.z value from 0 to gridDim.z-1.

Regarding the configuration of blocks, each block is organized into a three-dimensional array of threads. Two-dimensional blocks can be created by setting blockDim.z to 1. One-dimensional blocks can be created by setting both blockDim.y and blockDim.z to 1, as was the case in the vectorAddkernel example. As previously mentioned, all blocks in a grid have the same dimensions and sizes. 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 blockDim.

The total size of a block is limited to 1024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1024. For instance, blockDim(512, 1, 1), blockDim(8, 16, 4), and blockDim(32, 16, 2) are allowable blockDim values, but blockDim(32, 32, 2) is not allowable because the total number of threads would exceed 1024.[2]

The grid can have higher dimensionality than its blocks and vice versa. For instance, Fig. 3.1 shows a small toy grid example of gridDim(2, 2, 1) with blockDim(4, 2, 2). The grid can be generated with the following host code:

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

The grid consists of four blocks organized into a 2 × 2 array. Each block in Fig. 3.1 is labeled with (blockIdx.y, blockIdx.x), e.g., Block(1,0) has blockIdx.y=1 and blockIdx.x=0. The labels are ordered such that the highest dimension comes first. *Note that this block labeling notation is the reversed ordering of that used in the C statements for setting configuration parameters where the lowest dimension comes first*. This reversed ordering for labeling blocks works more effectively when we illustrate the mapping of thread coordinates into data indexes in accessing multidimensional data.

Each threadIdx also consists of three fields: the *x* coordinate threadId.x, the *y* coordinate threadIdx.y, and the *z* coordinate threadIdx.z. Fig. 3.1 illustrates the organization of threads within a block. In this example, each block is organized into 4 × 2 × 2 arrays of threads. All blocks within a grid have the same dimensions; thus, we

[2] Devices with capability less than 2.0 allow blocks with up to 512 threads.

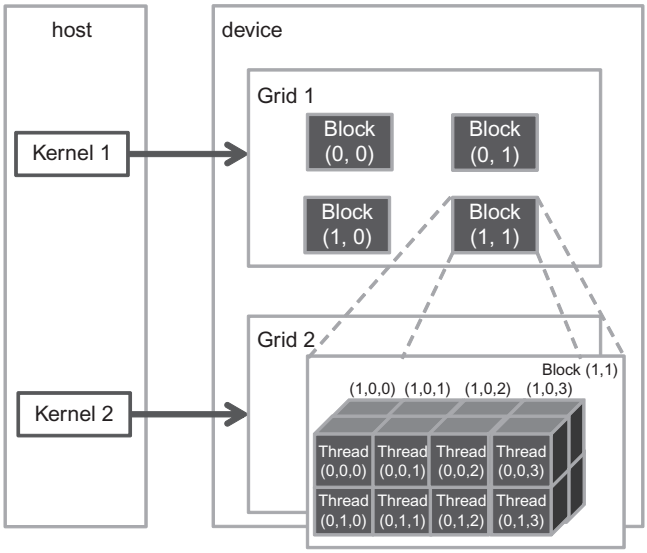

FIGURE 3.1

A multidimensional example of CUDA grid organization.

only need to show one of them. Fig. 3.1 expands Block(1,1) to show its 16 threads. For instance, Thread(1,0,2) has `threadIdx.z=1`, `threadIdx.y=0`, and `threadIdx.x=2`. This example shows 4 blocks of 16 threads each, with a total of 64 threads in the grid. We use these small numbers to keep the illustration simple. Typical CUDA grids contain thousands to millions of threads.

3.2 MAPPING THREADS TO MULTIDIMENSIONAL DATA

The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data. Pictures are 2D array of pixels. Using a 2D grid that consists of 2D blocks is often convenient for processing the pixels in a picture. Fig. 3.2 shows such an arrangement for processing a 76 × 62 picture `P` (76 pixels in the horizontal or *x* direction and 62 pixels in the vertical or *y* direction). Assume that we decided to use a 16 × 16 block, with 16 threads in the *x* direction and 16 threads in the *y* direction. We will need 5 blocks in the *x* direction and 4 blocks in the *y* direction, resulting in $5 \times 4 = 20$ blocks, as shown in Fig. 3.2. The heavy lines mark the block boundaries. The shaded area depicts the threads that cover pixels. It is easy to verify that one can identify the `Pin` element processed by thread(0,0) of block(1,0) with the formula:

$$P_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}} = P_{1*16+0,0*16+0} = P_{16,0}.$$

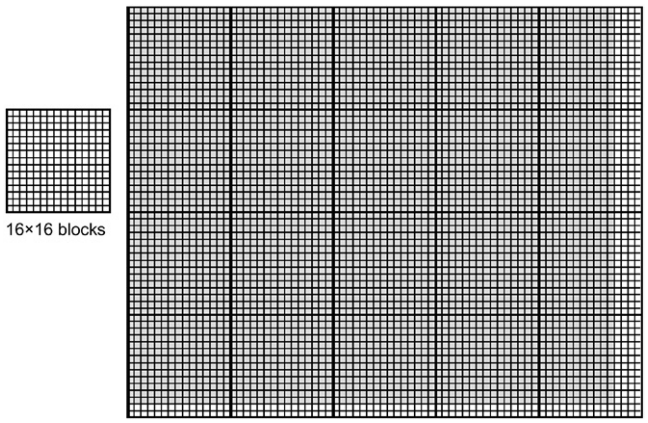

FIGURE 3.2

Using a 2D thread grid to process a 76 × 62 picture *P*.

Note that we have 4 extra threads in the *x* direction and 2 extra threads in the *y* direction—i.e., we will generate 80 × 64 threads to process 76 × 62 pixels. This case is similar to the situation in which a 1000-element vector is processed by the 1D kernel `vecAddKernel` in Fig. 2.11 by using four 256-thread blocks. Recall that an if statement is needed to prevent the extra 24 threads from taking effect. Analogously, we should expect that the picture processing kernel function will have if statements to test whether the thread indexes `threadIdx.x` and `threadIdx.y` fall within the valid range of pixels.

Assume that the host code uses an integer variable `m` to track the number of pixels in the *x* direction and another integer variable `n` to track the number of pixels in the *y* direction. We further assume that the input picture data have been copied to the device memory and can be accessed through a pointer variable `d_Pin`. The output picture has been allocated in the device memory and can be accessed through a pointer variable `d_Pout`. The following host code can be used to launch a 2D kernel `colorToGreyscaleConversion` to process the picture, as follows:

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

In this example, we assume, for simplicity, that the dimensions of the blocks are fixed at 16 × 16. Meanwhile, the dimensions of the grid depend on the dimensions of the picture. To process a 2000 × 1500 (3-million-pixel) picture, we will generate 11,750 blocks—125 in the *x* direction and 94 in the *y* direction. Within the kernel function, references to `gridDim.x`, `gridDim.y`, `blockDim.x`, and `blockDim.y` will result in 125, 94, 16, and 16, respectively.

MEMORY SPACE

Memory space is a simplified view of how a processor accesses its memory in modern computers. It is usually associated with each running application. The data to be processed by an application and instructions executed for the application are stored in locations in its memory space. Typically, each location can accommodate a byte and has an address. Variables that require multiple bytes—4 bytes for float and 8 bytes for double—are stored in consecutive byte locations. The processor generates the starting address (address of the starting byte location) and the number of bytes needed when accessing a data value from the memory space.

The locations in a memory space are similar to phones in a telephone system where everyone has a unique phone number. Most modern computers have at least 4G byte-sized locations, where each G is 1,073,741,824 (2^{30}). All locations are labeled with an address ranging from 0 to the largest number. Every location has only one address; thus, we say that the memory space has a "flat" organization. As a result, all multidimensional arrays are ultimately "flattened" into equivalent one-dimensional arrays. Whereas a C programmer can use a multidimensional syntax to access an element of a multidimensional array, the compiler translates these accesses into a base pointer that points to the initial element of the array, along with an offset calculated from these multidimensional indexes.

Before we show the kernel code, we need to first understand how C statements access elements of dynamically allocated multidimensional arrays. Ideally, we would like to access `d_Pin` as a two-dimensional array where an element at row *j* and column *i* can be accessed as `d_Pin[j][i]`. However, the ANSI C standard on which the development of CUDA C was based requires that the number of columns in `d_Pin` be known at compile time for `d_Pin` to be accessed as a 2D array. Unfortunately, this information is not known at compiler time for dynamically allocated arrays. In fact, part of the reason dynamically allocated arrays are used is to allow the sizes and dimensions of these arrays to vary according to data size at run time. Thus, the information on the number of columns in a dynamically allocated two-dimensional array is unknown at compile time by design. Consequently, programmers need to explicitly linearize or "flatten" a dynamically allocated two-dimensional array into an equivalent one-dimensional array in the current CUDA C. The newer C99 standard allows multidimensional syntax for dynamically allocated arrays. Future CUDA C versions may support multidimensional syntax for dynamically allocated arrays.

In reality, all multidimensional arrays in C are linearized because of the use of a "flat" memory space in modern computers (see "Memory Space" sidebar). In statically allocated arrays, the compilers allow the programmers to use higher-dimensional indexing syntax such as `d_Pin[j][i]` to access their elements. Under the hood, the

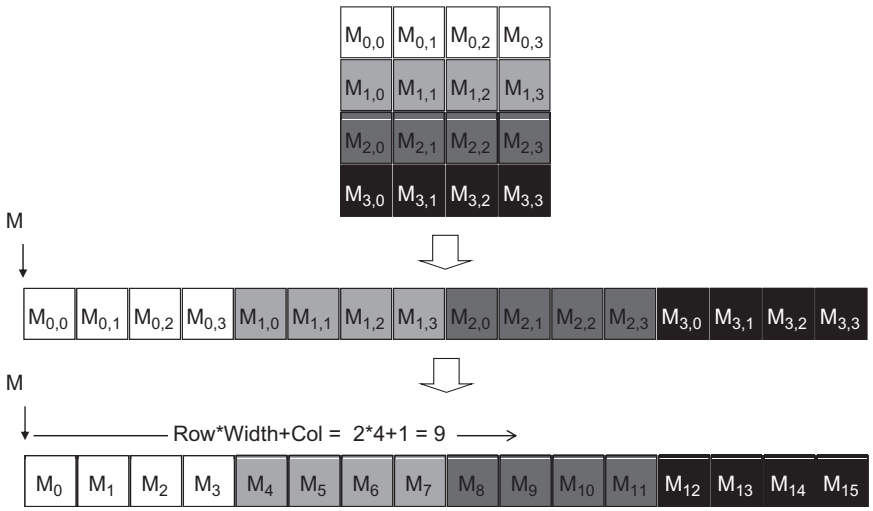

FIGURE 3.3

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

compiler linearizes them into an equivalent one-dimensional array and translates the multidimensional indexing syntax into a one-dimensional offset. In dynamically allocated arrays, the current CUDA C compiler leaves the work of such translation to the programmers because of the lack of dimensional information at compile time.

A two-dimensional array can be linearized in at least two ways. One way is to place all elements of the same row into consecutive locations. The rows are then placed one after another into the memory space. This arrangement, called *row-major layout*, is depicted in Fig. 3.3. To improve readability, we will use $M_{j,i}$ to denote the M element at the j^{th} row and the i^{th} column. $P_{j,i}$ is equivalent to the C expression $M[j][i]$ but is slightly more readable. Fig. 3.3 illustrates how a 4×4 matrix M is linearized into a 16-element one-dimensional array, with all elements of row 0 first, followed by the four elements of row 1, and so on. Therefore, the one-dimensional equivalent index for M in row j and column i is $j*4 + i$. The $j*4$ term skips all elements of the rows before row j. The i term then selects the right element within the section for row j. The one-dimensional index for $M_{2,1}$ is $2*4 + 1 = 9$, as shown in Fig. 3.3, where M_9 is the one-dimensional equivalent to $M_{2,1}$. This process shows the way C compilers linearize two-dimensional arrays.

Another method to linearize a two-dimensional array is to place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space. This arrangement, called the *column-major layout* is used by FORTRAN compilers. The column-major layout of a two-dimensional

array is equivalent to the row-major layout of its transposed form. Readers whose primary previous programming experience were with FORTRAN should be aware that CUDA C uses the row-major layout rather than the column-major layout. In addition, numerous C libraries that are designed for FORTRAN programs use the column-major layout to match the FORTRAN compiler layout. Consequently, the manual pages for these libraries, such as Basic Linear Algebra Subprograms (BLAS) (see "Linear Algebra Functions" sidebar), usually instruct the users to transpose the input arrays if they call these libraries from C programs.

LINEAR ALGEBRA FUNCTIONS

Linear algebra operations are widely used in science and engineering applications. BLAS, a de facto standard for publishing libraries that perform basic algebraic operations, includes three levels of linear algebra functions. As the level increases, the number of operations performed by the function increases as well. Level-1 functions perform vector operations of the form $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. Our vector addition example is a special case of a level-1 function with α=1. Level-2 functions perform matrix–vector operations of the form $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$, where $\mathbf{A}$ is a matrix, $\mathbf{x}$ and $\mathbf{y}$ are vectors, and α, β are scalars. We will be examining a form of level-2 function in sparse linear algebra. Level-3 functions perform matrix–matrix operations of the form $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, where **A, B, C** are matrices and α, β are scalars. Our matrix–matrix multiplication example is a special case of a level-3 function, where α=1 and β=0. These BLAS functions are used as basic building blocks of higher-level algebraic functions such as linear system solvers and eigenvalue analysis. As we will discuss later, the performance of different implementations of BLAS functions can vary by orders of magnitude in both sequential and parallel computers.

We are now ready to study the source code of `colorToGreyscaleConversion` shown in Fig. 3.4. The kernel code uses the formula

$$L = r * 0.21 + g * 0.72 + b * 0.07$$

to convert each color pixel to its greyscale counterpart.

A total of `blockDim.x*gridDim.x` threads can be found in the horizontal direction. As in the `vecAddKernel` example, the expression

`Col=blockIdx.x*blockDim.x+threadIdx.x` generates every integer value from 0 to `blockDim.x*gridDim.x-1`. We know that `gridDim.x*blockDim.x` is greater than or equal to `width` (`m` value passed in from the host code). We have at least as many threads as the number of pixels in the horizontal direction. Similarly, we know that

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * Pout,  unsigned
             char * Pin, int width, int height) {,
int Col = threadIdx.x + blockIdx.x * blockDim.x;
int Row = threadIdx.y + blockIdx.y * blockDim.y;
if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

FIGURE 3.4

Source code of colorToGreyscaleConversion showing 2D thread mapping to data.

at least as many threads as the number of pixels in the vertical direction are present. Therefore, as long as we test and make sure only the threads with both `Row` and `Col` values are within range—i.e., (`Col<width`) && (`Row<height`)—we can cover every pixel in the picture.

Given that each row has `width` pixels, we can thus generate the one-dimensional index for the pixel at row `Row` and column `Col` as `Row*width+Col`. This one-dimensional index `greyOffset` is the pixel index for `Pout` as each pixel in the output greyscale image is one byte (unsigned char). By using our 76 × 62 image example, the linearized one-dimensional index of the `Pout` pixel is calculated by thread(0,0) of block(1,0) with the formula:

$$Pout_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} = Pout_{1*16+0,0*16+0}$$
$$= Pout_{16,0} = Pout[16 * 76 + 0] = Pout[1216]$$

As for `Pin`, we multiply the gray pixel index by 3 because each pixel is stored as (*r*, *g*, *b*), with each equal to one byte. The resulting `rgbOffset` gives the starting location of the color pixel in the `Pin` array. We read the *r*, *g*, and *b* values from the three consecutive byte locations of the `Pin` array, perform the calculation of the greyscale pixel value, and write that value into the `Pout` array by using `greyOffset`. With our 76 × 62 image example, the linearized one-dimensional index of the `Pin` pixel is calculated by thread(0,0) of block(1,0) with the following formula:

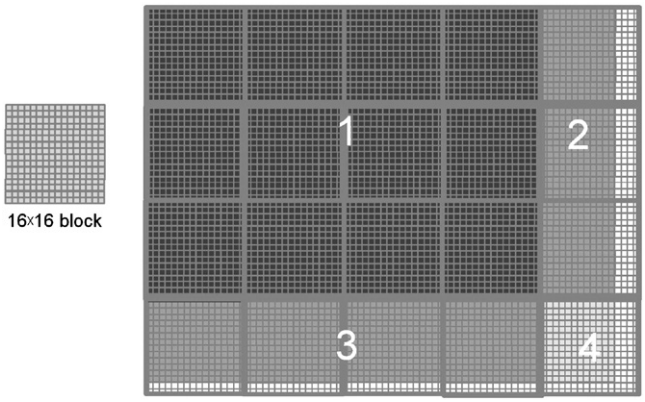

FIGURE 3.5

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

$$Pin_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}} = Pin_{1*16+0,0*16+0}$$
$$= Pin_{16,0} = Pin[16 * 76 * 3 + 0] = Pin[3648]$$

The data being accessed are the three bytes, starting at byte index 3648.

Fig. 3.5 illustrates the execution of `colorToGreyscaleConversion` when processing our 76 × 62 example. Assuming that we use 16 × 16 blocks, launching `colorToGreyscaleConvertion` generates 80 × 64 threads. The grid will have 20 blocks—5 in the horizontal direction and 4 in the vertical direction. The execution behavior of blocks will fall into one of four different cases, depicted as four shaded areas in Fig. 3.5.

The first area, marked as "1" in Fig. 3.5, consists of threads that belong to the 12 blocks covering the majority of pixels in the picture. Both the `Col` and `Row` values of these threads are within range; all these threads will pass the `if`-statement test and process pixels in the heavily shaded area of the picture—i.e., all 16 × 16 = 256 threads in each block will process pixels. The second area, marked as "2" in Fig. 3.5, contains the threads that belong to the three blocks in the medium-shaded area covering the upper right pixels of the picture. Although the `Row` values of these threads are always within range, some `Col` values exceed the `m` value (76). The reason is that the number of threads in the horizontal direction is always a multiple of the `blockDim.x` value chosen by the programmer (16 in this case). The smallest multiple of 16 needed to cover 76 pixels is 80. Thus, 12 threads in each row will have `Col` values that are within range and will process pixels. Meanwhile, 4 threads in each row will have `Col` values that are out of range and thus fail the `if`-statement condition. These threads will not process any pixels. Overall, 12 × 16 = 192 of the 16 × 16 = 256 threads in each of these blocks will process pixels.

The third area, marked "3" in Fig. 3.5, accounts for the 3 lower left blocks covering the medium-shaded area in the picture. Although the `Col` values of these threads are always within range, some `Row` values exceed the m value (62). The reason is that the number of threads in the vertical direction is always a multiple of the `blockDim.y` value chosen by the programmer (16 in this case). The smallest multiple of 16 to cover 62 is 64. Thus, 14 threads in each column will have `Row` values that are within range and will process pixels. Meanwhile, 2 threads in each column will fail the `if`-statement of area 2 and will not process any pixels. Of the 256 threads, $16 \times 14 = 224$ will process pixels. The fourth area, marked "4" in Fig. 3.5, contains threads that cover the lower right, lightly shaded area of the picture. In each of the top 14 rows, 4 threads will have `Col` values that are out of range, similar to Area 2. The entire bottom two rows of this block will have `Row` values that are out of range, similar to area "3". Thus, only $14 \times 12 = 168$ of the $16 \times 16 = 256$ threads will process pixels.

We can easily extend our discussion of 2D arrays to 3D arrays by including another dimension when we linearize arrays. This is accomplished by placing each "plane" of the array one after another into the address space. The assumption is that the programmer uses variables `m` and `n` to track the number of columns and rows in a 3D array. The programmer also needs to determine the values of `blockDim.z` and `gridDim.z` when launching a kernel. In the kernel, the array index will involve another global index:

```
int Plane = blockIdx.z*blockDim.z + threadIdx.z
```

The linearized access to a three-dimensional array `P` will be of the form `P[Plane*m*n+Row*m+Col]`. A kernel processing the 3D `P` array needs to check whether all the three global indexes—`Plane`, `Row`, and `Col`—fall within the valid range of the array.

3.3 IMAGE BLUR: A MORE COMPLEX KERNEL

We have studied `vecAddkernel` and `colorToGreyscaleConversion` in which each thread performs only a small number of arithmetic operations on one array element. These kernels serve their purposes well: to illustrate the basic CUDA C program structure and data parallel execution concepts. At this point, the reader should ask the obvious question—do all CUDA threads perform only such simple, trivial amount of operation independently of each other? The answer is no. In real CUDA C programs, threads often perform complex algorithms on their data and need to cooperate with one another. For the next few chapters, we are going to work on increasingly more complex examples that exhibit these characteristics. We will start with an image blurring function.

Image blurring smooths out the abrupt variation of pixel values while preserving the edges that are essential for recognizing the key features of the image. Fig. 3.6 illustrates the effect of image blurring. Simply stated, we make the image appear blurry. To the human eye, a blurred image tends to obscure the fine details and present

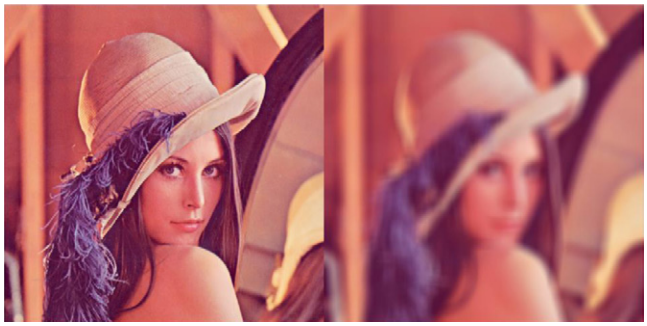

FIGURE 3.6

An original image and a blurred version.

the "big picture" impression or the major thematic objects in the picture. In computer image processing algorithms, a common use case of image blurring is to reduce the impact of noise and granular rendering effects in an image by correcting problematic pixel values with the clean surrounding pixel values. In computer vision, image blurring can be used to allow edge detection and object recognition algorithms to focus on thematic objects rather than being impeded by a massive quantity of fine-grained objects. In displays, image blurring is sometimes used to highlight a particular part of the image by blurring the rest of the image.

Mathematically, an image blurring function calculates the value of an output image pixel as a weighted sum of a patch of pixels encompassing the pixel in the input image. As we will learn in Chapter 7, Parallel pattern: convolution, the computation of such weighted sums belongs to the *convolution* pattern. We will be using a simplified approach in this chapter by taking a simple average value of the `N×N` patch of pixels surrounding, and including, our target pixel. To keep the algorithm simple, we will not place a weight on the value of any pixels based on its distance from the target pixel, which is common in a convolution blurring approach such as Gaussian blur.

Fig. 3.7 shows an example using a 3×3 patch. When calculating an output pixel value at the (`Row, Col`) position, we see that the patch is centered at the input pixel located at the (`Row, Col`) position. The 3×3 patch spans three rows (`Row-1, Row, Row+1`) and three columns (`Col-1, Col, Col+1`). To illustrate, the coordinates of the nine pixels for calculating the output pixel at (25, 50) are (24, 49), (24, 50), (24, 51), (25, 49), (25, 50), (25, 51), (26, 49), (26, 50), and (26, 51).

Fig. 3.8 shows an image blur kernel. Similar to that in `colorToGreyscaleConversion`, we use each thread to calculate an output pixel. That is, the thread to output data mapping remains the same. Thus, at the beginning of the kernel, we see the familiar calculation of the `Col` and `Row` indexes. We also see the familiar `if`-statement that verifies whether both `Col` and `Row` are within the valid range according to the height and width of the image. Only the threads whose `Col` and `Row` indexes are within the value ranges will be allowed to participate in the execution.

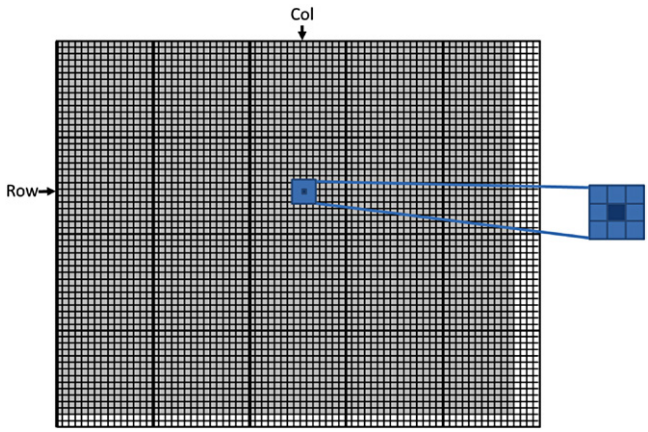

FIGURE 3.7

Each output pixel is the average of a patch of pixels in the input image.

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
    {
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
1.    int pixVal = 0;
2.    int pixels = 0;

    // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.    for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
     {

5.        int curRow = Row + blurRow;
6.        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
7.        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {

8.          pixVal += in[curRow * w + curCol];
9.          pixels++; // Keep track of number of pixels in the avg
          }
        }
      }

    // Write our new pixel value out
10.  out[Row * w + Col] = (unsigned char)(pixVal / pixels);
  }
 }
```

FIGURE 3.8

An image blur kernel.

As shown in Fig. 3.7, the `Col` and `Row` values also generate the central pixel location of the patch used to calculate the output pixel for the thread. The nested `for`-loop Lines 3 and 4 of Fig. 3.8 iterate through all pixels in the patch. We assume that the program has a defined constant, `BLUR_SIZE`. The value of `BLUR_SIZE` is set such that 2*`BLUR_SIZE` gives the number of pixels on each side of the patch. For a 3 × 3 patch, `BLUR_SIZE` is set to 1, whereas for a 7 × 7 patch, `BLUR_SIZE` is set to 3. The outer loop iterates through the rows of the patch. For each row, the inner loop iterates through the columns of the patch.

In our 3 × 3 patch example, the `BLUR_SIZE` is 1. For the thread that calculates the output pixel (25, 50), during the first iteration of the outer loop, the `curRow` variable is `Row-BLUR_SIZE` = (25 − 1) = 24. Thus, during the first iteration of the outer loop, the inner loop iterates through the patch pixels in row 24. The inner loop iterates from the column `Col-BLUR_SIZE` = 50 − 1 = 49 to `Col+BLUR_SIZE` = 51 by using the `curCol` variable. Therefore, the pixels processed in the first iteration of the outer loop are (24, 49), (24, 50), and (24, 51). The reader should verify that in the second iteration of the outer loop, the inner loop iterates through pixels (25, 49), (25, 50), and (25, 51). Finally, in the third iteration of the outer loop, the inner loop iterates through pixels (26, 49), (26, 50), and (26, 51).

Line 8 uses the linearized index of `curRow` and `curCol` to access the value of the input pixel visited in the current iteration. It accumulates the pixel value into a running sum variable `pixVal`. Line 9 records the addition of one more pixel value into the running sum by incrementing the `pixels` variable. After all pixels in the patch are processed, Line 10 calculates the average value of the pixels in the patch by dividing the `pixVal` value by the `pixels` value. It uses the linearized index of `Row` and `Col` to write the result into its output pixel.

Line 7 contains a conditional statement that guards the execution of Lines 9 and 10. For output pixels near the edge of the image, the patch may extend beyond the valid range of the picture. This is illustrated in Fig. 3.9 assuming 3 × 3 patches. In Case 1, the pixel at the upper left corner is being blurred. Five of the nine pixels in the intended patch do not exist in the input image. In this case, the `Row` and `Col` values of the output pixel are 0 and 0. During the execution of the nested loop, the `CurRow` and `CurCol` values for the nine iterations are (−1,−1), (−1,0), (−1,1), (0,−1), (0,0), (0,1), (1,−1), (1,0), and (1,1). Note that for the five pixels outside the image, at least one of the values is less than 0. The `curRow<0` and `curCol<0` conditions of the `if`-statement capture these values and skip the execution of Lines 8 and 9. As a result, only the values of the four valid pixels are accumulated into the running sum variable. The `pixels` value is also correctly incremented four times so that the average can be calculated properly at Line 10.

The readers should work through the other cases in Fig. 3.9 and analyze the execution behavior of the nested loop in the `blurKernel`. Note that most of the threads will find all pixels in their assigned 3 × 3 patch within the input image. They will accumulate all the nine pixels in the nested loop. However, for the pixels on the four corners, the responsible threads will accumulate only 4 pixels. For other pixels on the four edges, the responsible threads will accumulate 6 pixels in the nested loop.

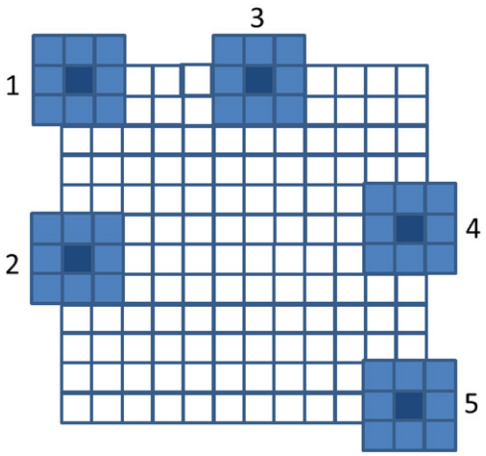

FIGURE 3.9

Handling boundary conditions for pixels near the edges of the image.

These variations necessitate keeping track of the actual number of pixels accumulated with variable `pixels`.

3.4 SYNCHRONIZATION AND TRANSPARENT SCALABILITY

We have discussed thus far how to launch a kernel for execution by a grid of threads and how to map threads to parts of the data structure. However, we have not yet presented any means to coordinate the execution of multiple threads. We will now study a basic coordination mechanism. CUDA allows threads in the same block to coordinate their activities by using a barrier synchronization function `__syncthreads()`. Note that "__" consists of two "_" characters. When a thread calls `__syncthreads()`, it will be held at the calling location until every thread in the block reaches the location. This process ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can proceed to the next phase.

Barrier synchronization is a simple and popular method for coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities of multiple persons. To illustrate, 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 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 ahead of others need to wait for those who finish later. Without the barrier synchronization, one or more persons can be left behind in the mall when the car leaves, which can seriously damage their friendship!

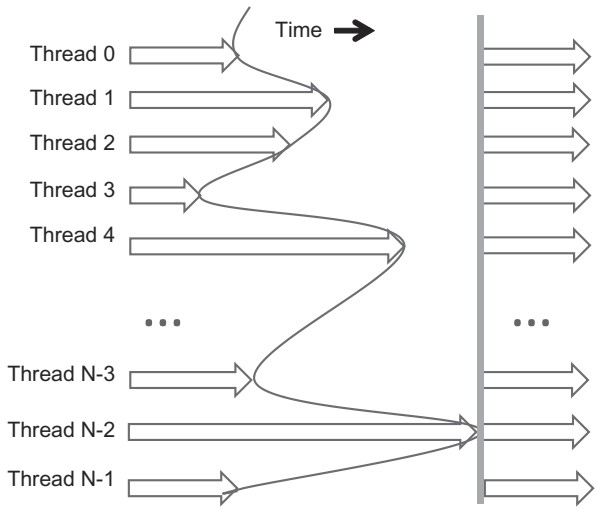

FIGURE 3.10

An example execution timing of barrier synchronization.

Fig. 3.10 illustrates the execution 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 who reach the barrier early will wait for those who arrive late. When the latest one arrives at the barrier, everyone can continue their execution. With barrier synchronization, "No one is left behind."

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 or none of the threads in a block execute the path that includes the `__syncthreads()`. For an `if-then-else` statement, if each path has a `__syncthreads()` statement, either all threads in a block execute 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.

The ability to synchronize also imposes execution constraints on threads within a block. These threads should execute in close temporal 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 arrives 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 resources needed for

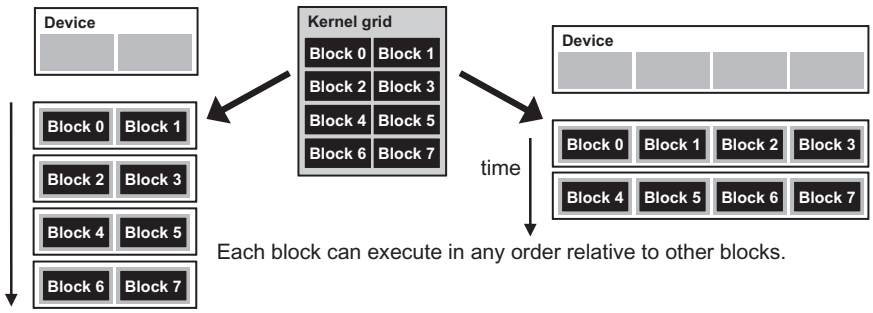

FIGURE 3.11

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

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 condition ensures the temporal proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization.

This leads us to an important tradeoff 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 because none of them need to wait for each other. This flexibility enables scalable implementations as shown in Fig. 3.11, 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 simultaneously, portrayed as executing two blocks at a time on the left hand side of Fig. 3.11. In a high-end implementation with more execution resources, one can execute a large number of blocks simultaneously, shown as four blocks at a time on the right hand side of Fig. 3.11.

The ability to execute the same application code within a wide range of speeds allows the production of a wide range of implementations in accordance with the cost, power, and performance requirements of particular market segments. For instance, a mobile processor may execute an application slowly but at extremely low power consumption, and a desktop processor may execute the same application at a higher speed but at increased power consumption. Both execute exactly the same application program with no change to the code. The ability to execute the same application code on hardware with different numbers of execution resources is referred to as *transparent scalability*. This characteristic reduces the burden on application developers and improves the usability of applications.

3.5 RESOURCE ASSIGNMENT

Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. As discussed in the previous section, these threads are assigned to

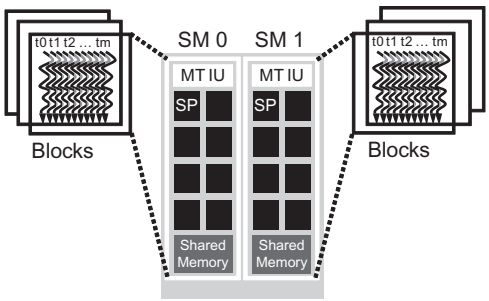

FIGURE 3.12

Thread block assignment to Streaming Multiprocessors (SMs).

execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into Streaming Multiprocessors (SMs). Fig. 3.12 illustrates that multiple thread blocks can be assigned to each SM. Each device sets a limit on the number of blocks that can be assigned to each SM. For instance, let us consider a CUDA device that may allow up to 8 blocks to be assigned to each SM. In situations where there is shortage of 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 SM until their combined resource usage falls below the limit. With limited numbers of SMs and limited numbers of blocks that can be assigned to each SM, the number of blocks that can be actively executing in a CUDA device is limited as well. 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 previously assigned blocks complete execution.

Fig. 3.12 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 (built-in registers) for SMs to maintain the thread and block indexes and track their execution status. Therefore, each generation of hardware sets a limit on the number of blocks and number of threads that can be assigned to an SM. For instance in the Fermi architecture, up to 8 blocks and 1536 threads can be assigned to each SM. This could be in the form of 6 blocks of 256 threads each, 3 blocks of 512 threads each, and so on. If the device only allows up to 8 blocks in an SM, it should be obvious that 12 blocks of 128 threads each is not a viable option. If a CUDA device has 30 SMs, and each SM can accommodate up to 1536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution.

3.6 QUERYING DEVICE PROPERTIES

Our discussions on assigning execution resources to blocks raise an important question. How do we find out the amount of resources available? When a CUDA

application executes on a system, how can it determine the number of SMs in a device and the number of blocks and threads that can be assigned to each SM? Other resources have yet to be discussed that can be relevant to the execution of a CUDA application. In general, many modern applications are designed to execute on a wide variety of hardware systems. The application often needs to *query* the available resources and capabilities of the underlying hardware in order to take advantage of the more capable systems while compensating for the less capable systems.

In CUDA C, a built-in mechanism exists for a host code to query the properties of the devices available in the system. The CUDA runtime system (device driver) has an API function `cudaGetDeviceCount` that returns the number of available CUDA devices in the system. The host code can determine the number of available CUDA devices by using the following statements:

```
int dev_count;
cudaGetDeviceCount(&dev_count);
```

RESOURCE AND CAPABILITY QUERIES

In everyday life, we often query the resources and capabilities available in an environment. When we make a hotel reservation, we can check the amenities that come with a hotel room. If the room comes with a hair dryer, we do not need to bring one. Most American hotel rooms come with hair dryers; many hotels in other regions do not.

Some Asian and European hotels provide toothpastes and even toothbrushes, whereas most American hotels do not. Many American hotels provide both shampoo and conditioner, whereas hotels in other continents often only provide shampoo.

If the room comes with a microwave oven and a refrigerator, we can take the leftover from dinner and expect to eat it the following day. If the hotel has a pool, we can bring swimsuits and take a dip after business meetings. If the hotel does not have a pool but has an exercise room, we can bring running shoes and exercise clothes. Some high-end Asian hotels even provide exercise clothing!

These hotel amenities are part of the properties, or resources and capabilities, of the hotels. Veteran travelers check these properties at hotel web sites, choose the hotels that better match their needs, and pack more efficiently and effectively given these details.

While it may not be obvious, a modern PC system often has two or more CUDA devices. The reason is that 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 Windows-based user interfaces. Most CUDA applications will not perform very well on these integrated devices. This weakness would be a reason for the host code to iterate through all the available devices, query their resources and capabilities, and choose the ones with adequate resources to execute the application satisfactorily.

The CUDA runtime numbers all available devices in the system from 0 to `dev_count-1`. It provides an API function `cudaGetDeviceProperties` that returns the properties of the device whose number is given as an argument. We can use the following statements in the host code to iterate through the available devices and query their properties:

```
cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
  //decide if device has sufficient resources and capabilities
}
```

The built-in type `cudaDeviceProp` is a C struct type with fields representing the properties of a CUDA device. The reader is referred to the CUDA C Programming Guide for all fields of the type. We will discuss a few of these fields that are particularly relevant to the assignment of execution resources to threads. We assume that the properties are returned in the `dev_prop` variable whose fields are set by the `cudaGetDeviceProperties` function. If the reader chooses to name the variable differently, the appropriate variable name will obviously need to be substituted in the following discussion.

As the name suggests, the field `dev_prop.maxThreadsPerBlock` indicates the maximal number of threads allowed in a block in the queried device. Some devices allow up to 1024 threads in each block and other devices allow fewer. Future devices may even allow more than 1024 threads per block. Therefore, the available devices should be queried, and the ones that will allow a sufficient number of threads in each block should be determined.

The number of SMs in the device is given in `dev_prop.multiProcessorCount`. As we discussed earlier, some devices have only a small number of SMs (e.g., two) and some have a much larger number of SMs (e.g., 30). If the application requires a large number of SMs in order to achieve satisfactory performance, it should definitely check this property of the prospective device. Furthermore, the clock frequency of the device is in `dev_prop.clockRate`. The combination of the clock rate and the number of SMs provides 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 fields `dev_prop.maxThreadsDim[0]`, `dev_prop.maxThreadsDim[1]`, and `dev_prop.maxThreadsDim[2]` (for the *x*, *y*, and *z* dimensions). Such information can be used for an automated tuning system to set the range of block dimensions when evaluating the best performing block dimensions for the

underlying hardware. Similarly, it can determine the maximal number of blocks allowed along each dimension of a grid in `dev_prop.maxGridSize[0]`, `dev_prop.maxGridSize[1]`, and `dev_prop.maxGridSize[2]` (for the *x*, *y*, and *z* dimensions). This information is typically used to determine whether a grid can have sufficient threads to handle the entire data set or whether some iteration is needed.

The `cudaDeviceProp` type has many more fields. We will discuss them as we introduce the concepts and features that they are designed to reflect.

3.7 THREAD SCHEDULING AND LATENCY TOLERANCE

Thread scheduling is strictly an implementation concept. Thus, it must be discussed in the context of specific hardware implementations. In the majority of implementations to date, a block assigned to an SM is further divided into 32 thread units called *warps*. The size of warps is implementation-specific. Warps are not part of the CUDA specification; however, knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices. The size of warps is a property of a CUDA device, which is in the `warpSize` field of the device query variable (`dev_prop` in this case).

The warp is the unit of thread scheduling in SMs. Fig. 3.13 shows the division of blocks into warps in an implementation. Each warp consists of 32 threads of consecutive `threadIdx` values: thread 0 through 31 form the first warp, 32 through 63 the

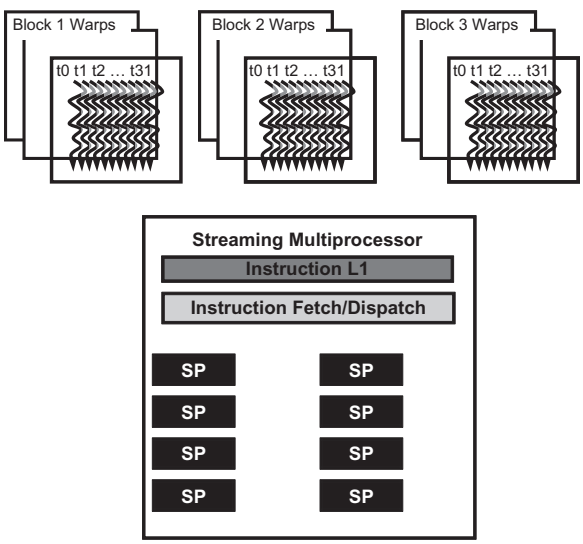

FIGURE 3.13

Blocks are partitioned into warps for thread scheduling.

second warp, and so on. In this example, three blocks—Block 1, Block 2, and Block 3—are assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.

We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM. In Fig. 3.13, if each block has 256 threads, we can determine that each block has 256/32 or 8 warps. With three blocks in each SM, we have $8 \times 3 = 24$ warps in each SM.

An SM is designed to execute all threads in a warp following the Single Instruction, Multiple Data (SIMD) model—i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. This situation is illustrated in Fig. 3.13 with a single instruction fetch/dispatch shared among execution units (SPs) in the SM. These threads will apply the same instruction to different portions of the data. Consequently, all threads in a warp will always have the same execution timing.

Fig. 3.13 also shows a number of hardware Streaming Processors (SPs) that actually execute instructions. In general, there are fewer SPs than the threads assigned to each SM; i.e., each SM has only enough hardware to execute instructions from a small subset of all threads assigned to the SM at any point in time. In early GPU designs, each SM can execute only one instruction for a single warp at any given instant. In recent designs, each SM can execute instructions for a small number of warps at any point in time. In either case, the hardware can execute instructions for a small subset of all warps in the SM. A legitimate question is why we need to have so many warps in an SM if it can only execute a small subset of them at any instant. The answer is that this is how CUDA processors efficiently execute long-latency operations, such as global memory accesses.

When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Instead, another resident warp that is no longer waiting for results will be selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution. This mechanism of filling the latency time of operations with work from other threads is often called “latency tolerance” or “latency hiding” (see “Latency Tolerance” sidebar).

Warp scheduling is also used for tolerating other types of operation latencies, such as pipelined floating-point arithmetic and branch instructions. Given a sufficient number of warps, 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 avoids introducing idle or wasted time into the execution timeline, which is referred to as zero-overhead thread scheduling. With warp scheduling, the long waiting time of warp instructions is “hidden” by executing instructions from other warps. This ability to tolerate long-latency operations is the main reason GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as do CPUs. Thus, GPUs can dedicate more of its chip area to floating-point execution resources.

LATENCY TOLERANCE

Latency tolerance is also needed in various everyday situations. For instance, in post offices, each person trying to ship a package should ideally have filled out all necessary forms and labels before going to the service counter. Instead, some people wait for the service desk clerk to tell them which form to fill out and how to fill out the form.

When there is a long line in front of the service desk, the productivity of the service clerks has to be maximized. Letting a person fill out the form in front of the clerk while everyone waits is not an efficient approach. The clerk should be assisting the other customers who are waiting in line while the person fills out the form. These other customers are "ready to go" and should not be blocked by the customer who needs more time to fill out a form.

Thus, a good clerk would politely ask the first customer to step aside to fill out the form while he/she can serve other customers. In the majority of cases, the first customer will be served as soon as that customer accomplishes the form and the clerk finishes serving the current customer, instead of that customer going to the end of the line.

We can think of these post office customers as warps and the clerk as a hardware execution unit. The customer that needs to fill out the form corresponds to a warp whose continued execution is dependent on a long-latency operation.

We are now ready for a simple exercise.[3] Assume that a CUDA device allows up to 8 blocks and 1024 threads per SM, whichever becomes a limitation first. Furthermore, it allows up to 512 threads in each block. For image blur, should we use 8×8, 16×16, or 32×32 thread blocks? To answer the question, we can analyze the pros and cons of each choice. If we use 8×8 blocks, each block would have only 64 threads. We will need $1024/64 = 12$ blocks to fully occupy an SM. However, each SM can only allow up to 8 blocks; thus, we will end up with only $64 \times 8 = 512$ threads in each SM. This limited number implies that the SM execution resources will likely be underutilized because fewer warps will be available to schedule around long-latency operations.

The 16×16 blocks result in 256 threads per block, implying that each SM can take $1024/256 = 4$ blocks. This number is within the 8-block limitation and is a good configuration as it will allow us a full thread capacity in each SM and a maximal number of warps for scheduling around the long-latency operations. The 32×32 blocks would give 1024 threads in each block, which exceeds the 512 threads per block limitation of this device. Only 16×16 blocks allow a maximal number of threads assigned to each SM.

[3] Note that this is an over-simplified exercise. As we will explain in Chapter 4, Memory and data locality, the usage of other resources such as registers and shared memory must also be considered when determining the most appropriate block dimensions. This exercise highlights the interactions between the limit on number of blocks and the limit on the number of threads that can be assigned to each SM.

3.8 SUMMARY

The kernel execution configuration parameters define the dimensions of a grid and its blocks. Unique coordinates in `blockIdx` and `threadIdx` allow threads of a grid to identify themselves and their domains of data. It is the responsibility of the programmer to use these variables in kernel functions so that the threads can properly identify the portion of the data to process. This model of programming compels the programmer to organize threads and their data into hierarchical and multidimensional organizations.

Once a grid is launched, its blocks can be assigned to SMs in an arbitrary order, resulting in the transparent scalability of CUDA applications. The transparent scalability comes with a limitation: threads in different blocks cannot synchronize with one another. To allow a kernel to maintain transparent scalability, the simple method 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 SMs for execution on a block-by-block basis. Each CUDA device imposes a potentially different limitation on the amount of resources available in each SM. Each CUDA device sets a limit on the number of blocks and the number of threads each of its SMs can accommodate, whichever becomes a limitation first. For each kernel, one or more of these resource limitations can become the limiting factor for the number of threads that simultaneously reside in a CUDA device.

Once a block is assigned to an SM, it is further partitioned into warps. All threads in a warp have identical execution timing. At any time, the SM executes instructions of only a small subset of its resident warps. This condition allows the other warps to wait for long-latency operations without slowing down the overall execution throughput of the massive number of execution units.

3.9 EXERCISES

1. A matrix addition takes two input matrices A and B and produces one output matrix C. Each element of the output matrix C is the sum of the corresponding elements of the input matrices A and B, i.e., C[i][j] = A[i][j] + B[i][j]. For simplicity, we will only handle square matrices whose elements are single-precision floating-point numbers. Write a matrix addition kernel and the host stub function that can be called with four parameters: pointer-to-the-output matrix, pointer-to-the-first-input matrix, pointer-to-the-second-input matrix, and the number of elements in each dimension. Follow the instructions below:

A. Write the host stub function by allocating memory for the input and output matrices, transferring input data to device; launch the kernel, transferring the output data to host and freeing the device memory for the input and output data. Leave the execution configuration parameters open for this step.

B. Write a kernel that has each thread to produce one output matrix element. Fill in the execution configuration parameters for this design.
C. Write a kernel that has each thread to produce one output matrix row. Fill in the execution configuration parameters for the design.
D. Write a kernel that has each thread to produce one output matrix column. Fill in the execution configuration parameters for the design.
E. Analyze the pros and cons of each kernel design above.

2. A matrix–vector multiplication takes an input matrix B and a vector C and produces one output vector A. Each element of the output vector A is the dot product of one row of the input matrix B and C, i.e., $A[i] = \sum^{j} B[i][j] + C[j]$. For simplicity, we will only handle square matrices whose elements are single-precision floating-point numbers. Write a matrix–vector multiplication kernel and a host stub function that can be called with four parameters: pointer-to-the-output matrix, pointer-to-the-input matrix, pointer-to-the-input vector, and the number of elements in each dimension. Use one thread to calculate an output vector element.

3. If the SM of a CUDA device can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the largest number of threads in the SM?
A. 128 threads per block
B. 256 threads per block
C. 512 threads per block
D. 1024 threads per block

4. For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?
A. 2000
B. 2024
D. 2048
D. 2096

5. With reference to the previous question, how many warps do you expect to have divergence due to the boundary check on vector length?
A. 1
B. 2
C. 3
D. 6

6. You need to write a kernel that operates on an image of size 400×900 pixels. You would like to assign one thread to each pixel. You would like your thread blocks to be square and to use the maximum number of threads per block possible on the device (your device has compute capability 3.0). How would you select the grid dimensions and block dimensions of your kernel?

7. With reference to the previous question, how many idle threads do you expect to have?

8. Consider a hypothetical block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in microseconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, and 2.9 and to spend the rest of their time waiting for the barrier. What percentage of the total execution time of the thread is spent waiting for the barrier?

9. Indicate which of the following assignments per multiprocessor is possible. In the case where it is not possible, indicate the limiting factor(s).
   - **A.** 8 blocks with 128 threads each on a device with compute capability 1.0
   - **B.** 8 blocks with 128 threads each on a device with compute capability 1.2
   - **C.** 8 blocks with 128 threads each on a device with compute capability 3.0
   - **D.** 16 blocks with 64 threads each on a device with compute capability 1.0
   - **E.** 16 blocks with 64 threads each on a device with compute capability 1.2
   - **F.** 16 blocks with 64 threads each on a device with compute capability 3.0

10. A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the `__syncthreads()` instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.

11. A student mentioned that he was able to multiply two 1024×1024 matrices by using a tiled matrix multiplication code with 32×32 thread blocks. He is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

This page intentionally left blank