Chapter 3

Scalable Parallel Execution

With special contribution from Mark Ebersole

Keywords: execution configuration parameters, order, multi-dimensional arrays, multi-dimensional grids, multi-dimensional blocks, row-major order, column-major, linearization, flattening, thread scheduling, transparent scalability, device property query, barrier synchronization, latency tolerance, zero-overhead thread scheduling

CHAPTER OUTLINE

In Chapter 2, we learned to write a simple CUDA C program that launches a kernel and a grid of threads to operate on elements one-dimensional arrays. The kernel specifies the C statements that are executed by each individual thread. As we unleash such massive execution activity, we need to be able to control these activities to achieve desired results, efficiency and speed. In this chapter, we will study several 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. We then 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 to understand high-performance parallel applications.

3.1. CUDA Thread Organization

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. All threads in a block share the same block index, which is available as the value of the blockIdx variable in a kernel. Each thread also 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 available as the values of variables gridDim and blockDim in kernel functions.

Hierarchical Organizations:

Like CUDA threads, many real-world systems are organized hierarchically. The U.S. 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 is sometimes larger than a city. For example, many counties and cities of 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.

One can think of each phone line as a CUDA thread, the area code as the value of blockIdx and the seven-digital local number as the value of threadIdx. This hierarchical organization allows the system to have a very large number of phone lines while preserving "locality" for calling the same area. That is, 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 a 1.) The hierarchical organization of CUDA threads also offers a form of locality. We will study this locality soon.

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 number of blocks. The second specifies the dimensions of each block in number of threads. Each such parameter is of 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.

For example, 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 the 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 dim3 type and the kernel launch uses the appropriate names. For example, 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. For example, the kernel launch in Figure 2.15 can be written as:

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

This allows the number of blocks to vary with the size of the vectors so that the grid will have enough 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 dimension of the grid. If n is equal to 1,000, the grid will consists of four blocks. If n is equal to 4,000, the grid will have 16 blocks. In

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

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.

For convenience, CUDA C provides a special shortcut for launching a kernel with one-dimensional grids and blocks. Instead of using dim3 variables, one can use arithmetic expressions 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. This gives us the kernel launch statement shown in Figure 2.15:

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

Readers who are familiar with the use of structures in C would realize that this “short-hand” 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 allows the compiler to conveniently initialize the x fields of gridDim and blockDim with the values provided in the execution configuration parameters when the shortcut is used.

Within the kernel function, the x field of variables gridDim and blockDim are pre-initialized according to the values of the execution configuration parameters. For example, if n is equal to 4,000, references to gridDim.x and blockDim.x in the vectAddkernel kernel will result in 16 and 256 respectively. Note that 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. That is, the gridDim and blockDim variables 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, the blockIdx.z value from 0 to gridDim.z-1.

We now turn our attention to 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-dimension blocks can be created by setting both blockDim.y and blockDim.z to 1, as was in the vectorAddkernel example. As we mentioned before, 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 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, blockDim(512, 1, 1), blockDim(8, 16, 4) and blockDim(32, 16, 2) are all allowable blockDim values but blockDim(32, 32, 2) is not allowable since the total number of threads would exceed 1,024[2].

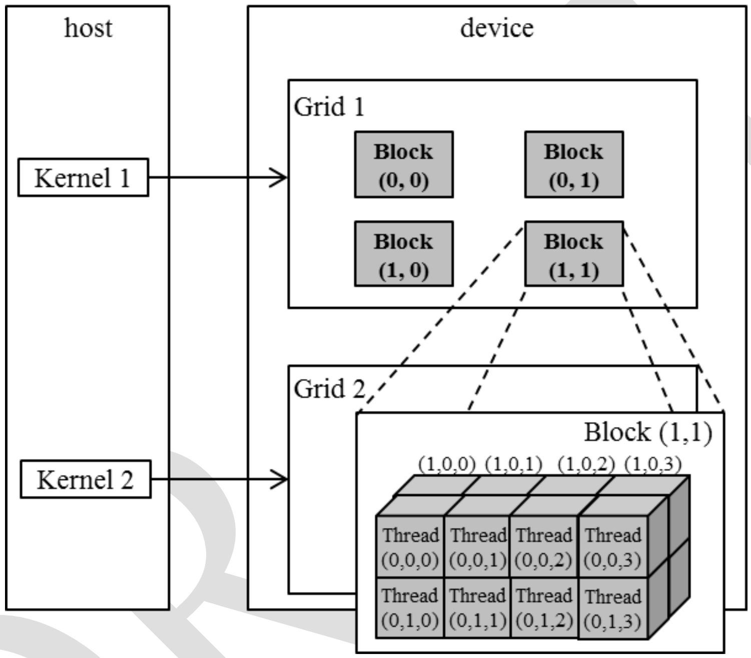

***Figure 3.1** A multi-dimensional example of CUDA grid organization*

Note that the grid can have higher dimensionality than its blocks and vice versa. For example, Figure 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>>>(…);
```

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

The grid consists of four blocks organized into a 2×2 array. Each block in Figure 3.1 is labeled with (blockIdx.y, blockIdx.x). For example, Block(1,0) has blockIdx.y=1 and blockIdx.x=0. Note that the ordering of the labels is such that highest dimension comes first. **Note that this block labeling notation is in 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 better when we illustrate the mapping of thread coordinates into data indexes in accessing multi-dimensional data.

Each threadIdx also consists of three fields: the x coordinate threadId.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z. Figure 3.1 illustrates the organization of threads within a block. In this example, each block is organized into 4×2×2 arrays of threads. Since all blocks within a grid have the same dimensions, we only need to show one of them. Figure 3.1 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. Note that in this example, we have 4 blocks of 16 threads each, with a grand 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 Multi-Dimensional Data

The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data. For example, 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. Figure 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, which results in 5×4=20 blocks as shown in Figure 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:

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

Note that we have 4 extra threads in the x direction and 2 extra threads in the y direction. That is, we will generate 80×64 threads to process 76×62 pixels. This is similar to the situation where a 1,000-element vector is processed by the 1D kernel vecAddKernel in Figure 2.11 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 indices threadIdx.x and threadIdx.y fall within the valid range of pixels.

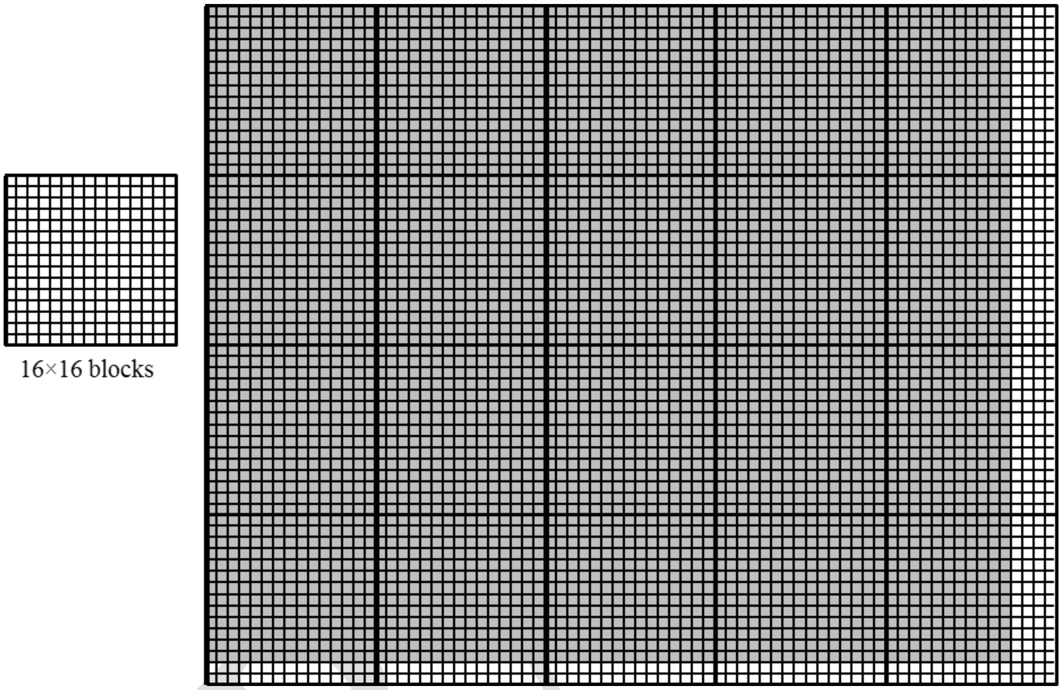

Figure 3.2 *Using a 2D thread grid to process a 76x62 picture P.*

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 the number of pixels in the y direction. We further assume that the input picture data has 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. The dimensions of the grid, on the other hand, depend on the dimensions of the picture. To process a 2,000×1,500 (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. A memory space 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. Each location typically 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 gives 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 like 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 that range from 0 to the largest number. Since there is only one address for every location, we say that the memory space has a "flat" organization. As a result, all multi-dimensional arrays are ultimately "flattened", into equivalent one-dimensional arrays. While a C programmer can use a multi-dimensional syntax to access an element of a multi-dimensional array, the compiler translates these accesses into a base pointer that points to the beginning element of the array, along with an offset calculated from these multi-dimensional indices.

Before we show the kernel code, we need to first understand how C statements access elements of dynamically allocated multi-dimensional arrays. Ideally, we would like to access d_Pin as a two-dimensional array where an element at the row j and column i can be accessed as d_Pin[j][i]. However, the ANSI C standard based on which CUDA C was developed requires that the number of columns in Pin be known at compile time for 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 why one uses dynamically allocated arrays 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 not known at compile time by design. As a result, programmers need to explicitly linearize, or "flatten", a dynamically allocated two-dimensional array into an equivalent one-dimensional array in the current CUDA C. Note that newer C99 standard allows multi-dimensional syntax for dynamically allocated arrays. It is likely that future CUDA C versions may support multi-dimensional syntax for dynamically allocated arrays.

In reality, all multi-dimensional arrays in C are linearized. This is due to the use of a “flat” memory space in modern computers (see “Memory Space” sidebar). In the case of 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 compiler linearizes them into an equivalent one-dimensional array and translates the multi-dimensional indexing syntax into a one-dimensional offset. In the case of dynamically allocated arrays, the current CUDA C compiler leaves the work of such translation to the programmers due to lack of dimensional information at compile time.

There are at least two ways one can linearize a two-dimensional array. One 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 illustrated in Figure 3.3. To increase the readability, we will use $M_{j,i}$ to denote a M element at the jth row and the ith column. $P_{j,i}$ is equivalent to the C expression M[j][i] but slightly more readable. Figure 3.3 shows an example where 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, etc. Therefore, the one-dimensional equivalent index for M element in row j and column i is j*4+i. The j*4 term skips over all elements of the rows before row j. The i term then selects the right element within the section for row j. For example, the one-dimensional index for $M_{2,1}$ is 2*4+1=9. This is illustrated in Figure 3.3, where M_9 is the one-dimensional equivalent to $M_{2,1}$. This is the way C compilers linearize two-dimensional arrays.

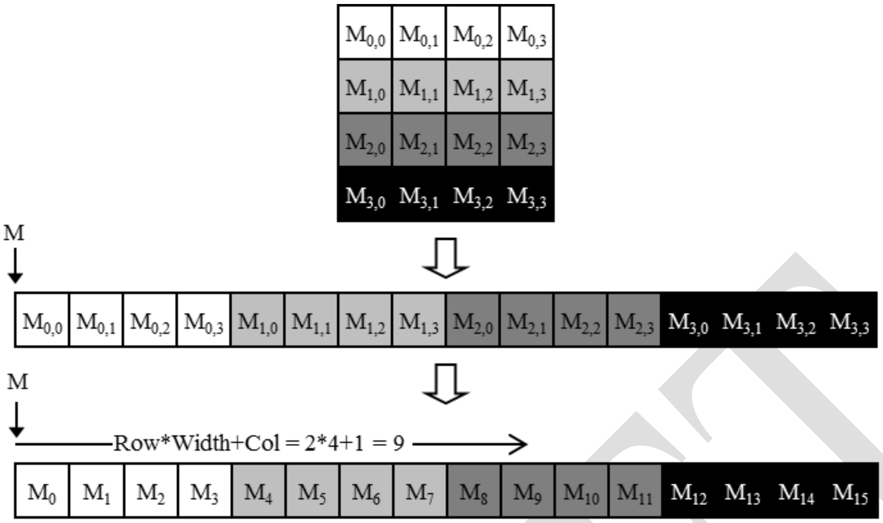
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
Row*Width+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

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.*

Another way 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 *column-major layout* is

Linear Algebra Functions

Linear algebra operations are widely used in science and engineering applications. In the Basic Linear Algebra Subprograms (BLAS), a de facto standard for publishing libraries that perform basic algebra operations, there are three levels of linear algebra functions. As the level increases, the amount of operations performed by the function increases. Level-1 functions performs vector operations of the form $\mathbf{y}= \alpha\mathbf{x}+\mathbf{y}$, *where* **x** *and* **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* **A** *is a matrix,* **x** *and* **y** *are vectors, and α, β are scalars. We will be studying a form of level-2 function in sparse linear algebra. Level-3 functions perform matrix-matrix operations in the form of* $\mathbf{C}=\alpha\mathbf{AB}+\beta\mathbf{C}$, *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 important because they 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.*

used by FORTRAN compilers. Note that column-major layout of a two-dimensional array is equivalent to the row-major layout of its transposed form. We will not spend more time on this except mentioning that readers whose primary previous programming experience were with FORTRAN should be aware that CUDA C uses row-major layout rather than the column major layout. Also, many C libraries that are designed to be used by FORTRAN programs use column-major layout to match the FORTRAN compiler layout. As a result, the manual pages for these libraries such as Basic Linear Algebra Subprograms (see "Linear Algebra Functions" sidebar) usually tell the users to transpose the input arrays if they call these libraries from C programs.

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

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

to convert each color pixel to a its greyscale counterpart.

There are a total of blockDim.x*gridDim.x threads in the horizontal direction. As shown 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 also know that there are at least as many threads as the number of pixels in the vertical direction. Therefore, as long as we test and make sure only the threads with both Row and Col values within range, that is (Col<width) && (Row<height), we will be able to cover every pixel in the picture.

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colortoGreyscaleConvertion(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 gray scale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r =  Pin[rgbOffset      ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 1]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 2]; // 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*

Since there are width pixels in each row, we can 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 since each pixel in the output greyscale image is one byte (unsigned char). Using our 76x62 image example, the linearized one-dimensional index of the Pout pixel calculated by thread(0,0) of block(1,0) with the formula:

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

As for Pin, we need to multiple the grey pixel index by 3 since each pixel is stored as (r, g, b), each of which is one byte. The resulting rgbOffset is gives the starting location of the color pixel in the Pin array. We read the r, g, and b value 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 using greyOffset.

Using our 76x62 image example, the linearized one-dimensional index of the Pin pixel calculated by thread(0,0) of block(1,0) with the formula:

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

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

Figure 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 full into one of four different cases, shown as four shaded areas in Figure 3.5.

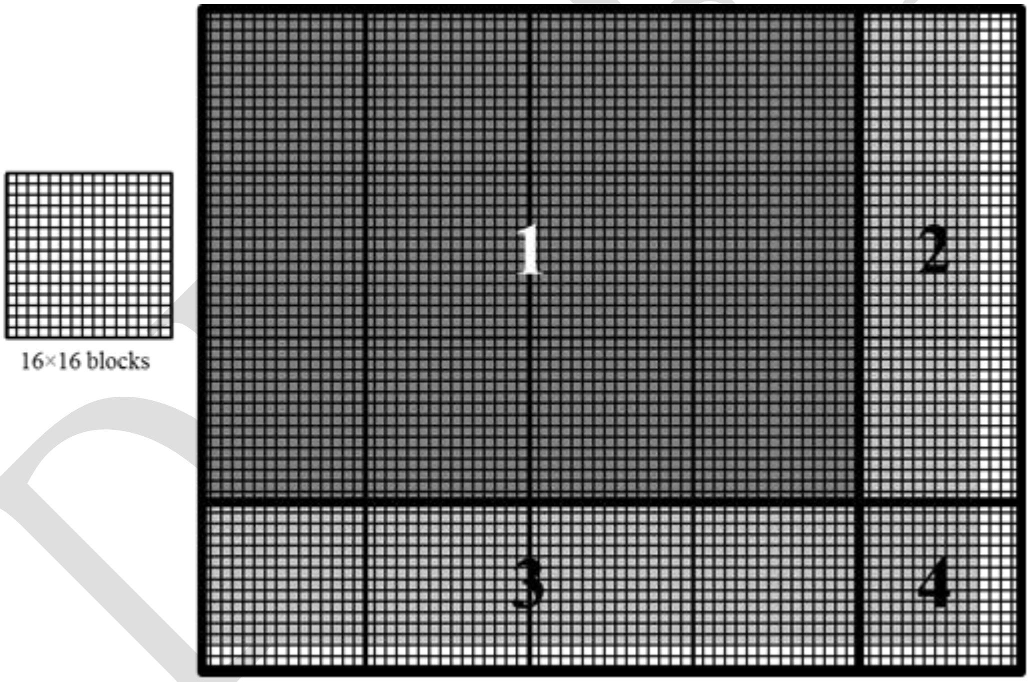

***Figure 3.5* Covering a 76×62 picture with 16×blocks**

The first area, marked as 1 in Figure 3.5, consists of the threads that belong to the 12 blocks covering the majority of pixels in the picture. Both Col and Row values of these threads are within range; all these threads will pass the if-statement test and process pixels in the dark shaded area of the picture. That is all 16×16=256 threads in each block will process pixels. The second area, marked as 2 in Figure 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, the Col values of some of them exceed the m value (76). This is because 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. As a result, 12 threads in each row will find their Col values within range and will process pixels. On the other hand, 4 threads in each row will find their Col values out of range, and thus fail the if-statement condition. These threads will not process any pixels. Overall, 12×16=192 out of the 16×16=256 threads in each of these blocks will process pixels.

The third area, marked as 3 in Figure 3.5, accounts for the 3 lower left blocks covering the medium shaded area of the picture. Although the Col values of these threads are always within range, the Row values of some of them exceed the m value (62). This is because the number of threads in the vertical direction is always multiples of the blockDim.y value chosen by the programmer (16 in this case). The smallest multiple of 16 to cover 62 is 64. As a result, 14 threads in each column will find their Row values within range and will process pixels. On the other hand, 2 threads in each column will fail the if-statement of area 2, and will not process any pixels. 16×14=224 out of the 256 threads will process pixels. The fourth area, marked as 4 in Figure 3.5, contains the threads that cover the lower right lightly shaded area of the picture. Similar to Area 2, 4 threads in each of the top 14 rows will find their Col values out of range. Similar to Area 3, the entire bottom two rows of this block will find their Row values out of range. So, only 14×12=168 of the 16×16=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 done by placing each "plane" of the array one after another into the address space. Assume 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 in the form of P[Plane*m*n + Row * m + Col]. A kernel processing the 3D P array needs to check whether all the three global indices, 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 where 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 each other. 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.

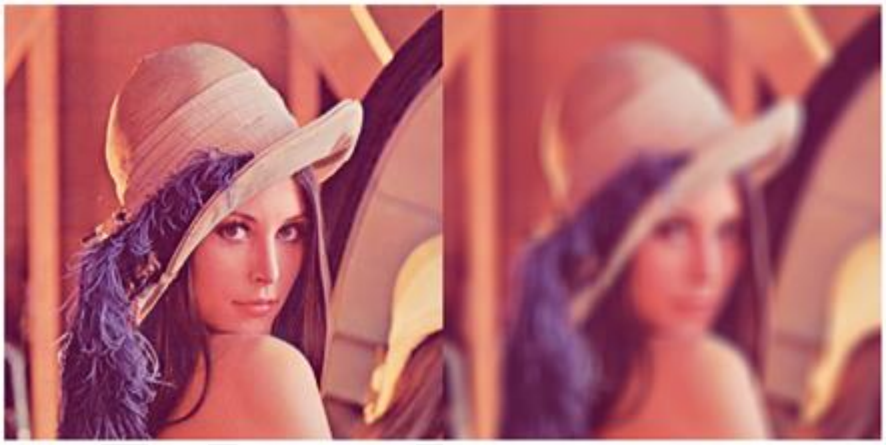

Figure 3.6 An original image and a blurred version.

Image blurring smooths out abrupt variation of pixel values while preserving the edges that are essential for recognizing the key features of the image. Figure 3.6 illustrates the effect of image blurring. Simply stated, we make the image blurry. To human eyes, a blurred image tends to obscure the fine details and present 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 cane be used to allow edge detection and object recognition algorithms to focus on thematic objects rather than being bogged down 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 patterns: 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 NxN 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, as is common in a convolution blurring approach such as Gaussian Blur.

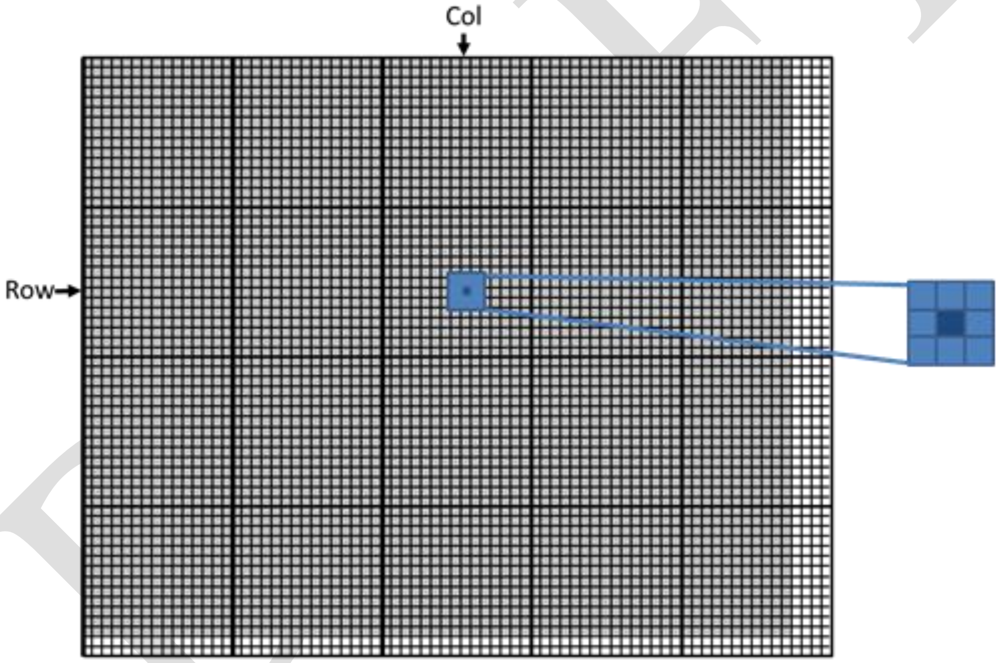

Figure 3.7 Each output pixel is the average of a patch of pixels in the input image.

Figure 3.7 shows an example using a 3x3 patch. When calculating an output pixel value at (Row, Col) position, we see that the patch is centered at the input pixel located at the (Row, Col) position. The 3x3 patch spans three rows (Row-1, Row, Row+1) and three columns (Col-1, Col, Col+1). For example, 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).

```
  __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.

Figure 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 indices. We also see the familiar if-statement that verifies that 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 indices are both within value ranges will be allowed to participate in the execution.

As shown in Figure 3.7, the Col and Row values also gives the central pixel location of the patch used for calculating the output pixel for the thread. The nested for-loops at Lines 3 and 4 in Figure 3.8 iterate through all the 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 example, for a 3x3 patch, BLUR_SIZE is set to 1 whereas for a 7x7 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 3x3 patch example, the BLUR_SIZE is 1. For the thread that calculates 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 column Col-BLUR_SIZE = 50-1 = 49 to Col+BLUR_SIZE = 51 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, the 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 fact that one more pixel value has been added into the running sum by incrementing the pixels variable. After all the pixels in the patch are processed, Line 10 calculates the average value of the pixels in the patch by diving 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 Figure 3.9 assuming 3x3 patches. In Case 1, the pixel at the upper left corner is being blurred. Five out of the nine pixels in the intended patch do not exist in the input image. In this case, the Row and Col value of the output pixel is 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), (1,1). Note that for the five pixels that are outside the image, at least one of the values is less than 0. The CurRow<0 and CurCol<0 conditions of the if-statement catch these values and skips 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.

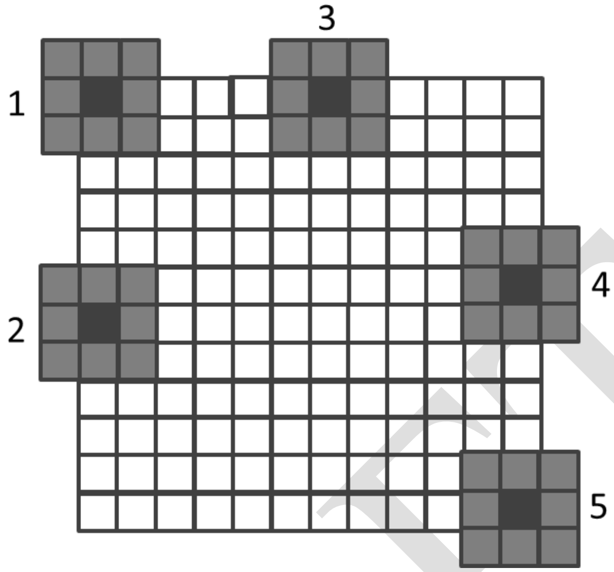

Figure 3.9 Handling boundary conditions for pixels near the edges of the image

The readers should work through the other cases in Figure 3.9 and analyze the execution behavior of the nested loop in the blurKernel. Note that most of the threads will find all the pixels in their assigned 3x3 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. These variations necessitate keeping track of the actual number of pixels accumulated with variable pixels.

3.4. Synchronization and Transparent Scalability

Thus far, we have discussed 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 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 ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase.

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

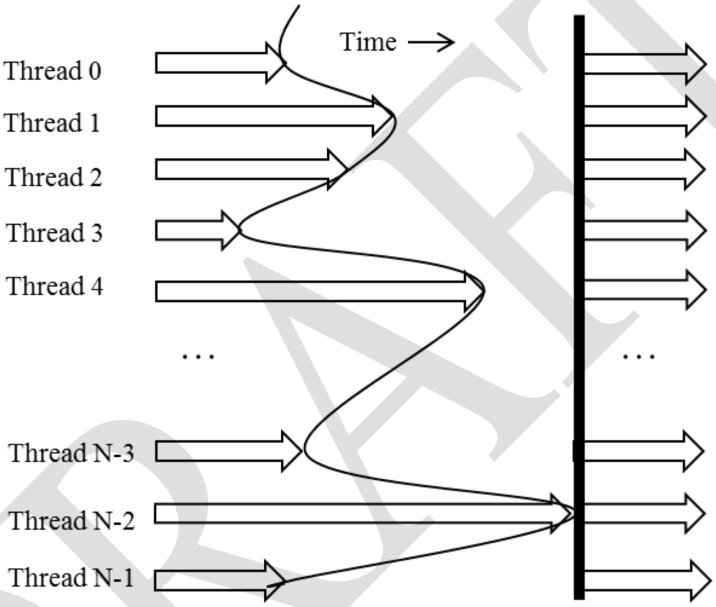

Figure 3.10 An example execution timing of barrier synchronization

Figure 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 that 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 threads in a block execute the path that includes the __syncthreads() or none of them does. For an `if-the-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 time proximity with each other to avoid excessively long waiting times. In fact, one needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier. Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to wait forever. CUDA run-time 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 run-time system has secured all the resources needed for all threads in the block to complete execution. When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization.

This leads us to 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 run-time 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 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 at the same time; portrayed as executing two blocks a time on the left hand side of Figure 3.11. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time; shown as four blocks at a time on the right hand side of Figure 3.11.

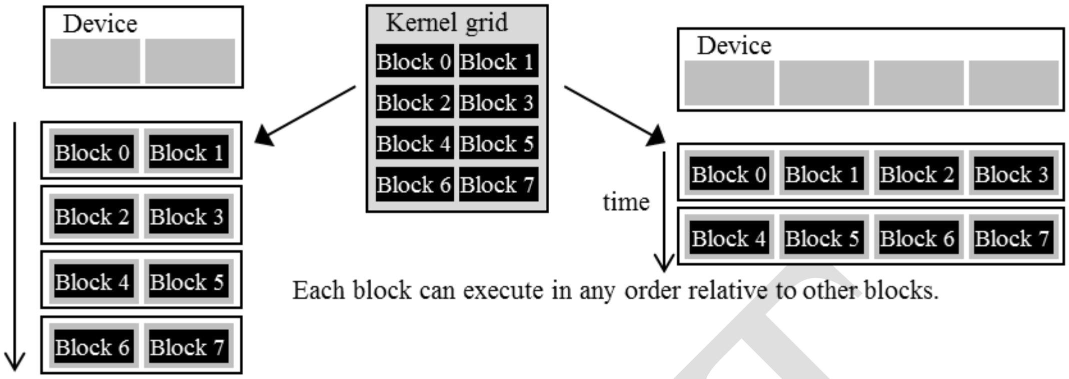

Figure 3.11 Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs

The ability to execute the same application code within a wide range of speeds allows the production of a wide range of implementations according to the cost, power, and performance requirements of particular market segments. For example, 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 while consuming more power. Both execute exactly the same application program with no change to the code. The ability to execute the same application code on hardware with different number of execution resources is referred to as transparent scalability, which reduces the burden on application developers and improves the usability of applications.

3.5. Resource Assignment

Once a kernel is launched, the CUDA run-time system generates the corresponding grid of threads. As we discussed in the previous section, these threads are assigned to execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into Streaming Multiprocessors (SMs). Figure 3.12 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 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 under the limit. With limited numbers of SMs and 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 run-time system maintains a list of blocks that need to execute and assigns new blocks to SMs as previously assigned blocks complete execution.

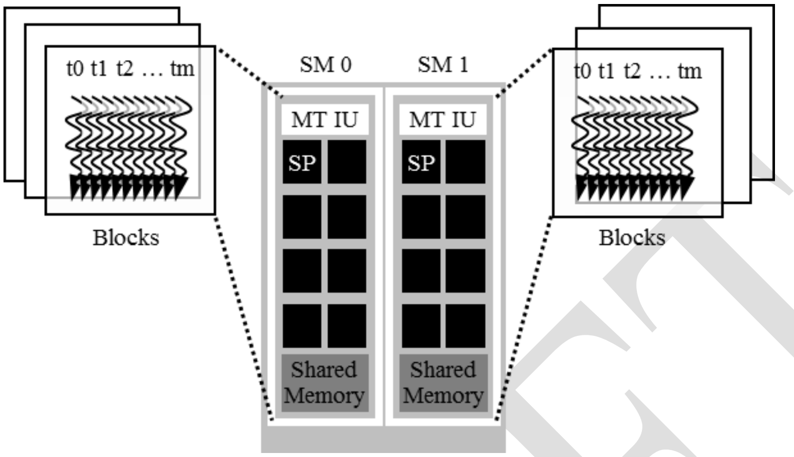

Figure 3.12 Thread block assignment to Streaming Multiprocessors (SMs)

Figure 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 indices 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. In more recent CUDA device designs, up to 8 blocks and 1,536 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, etc. 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 1,536 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 find out the number of SMs in a device and the number of blocks and threads that can be assigned to each SM? Obviously, there are also other resources that we have not discussed so far but 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. There is often a need for the application to *query* the available resources and capabilities of the

Resource and Capability Queries

In everyday life, we often query the resources and capabilities in an environment. For example, 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 while many hotels in other regions do not.

Some Asian and European hotels provide tooth pastes and even tooth brushes while most American hotels do not. Many American hotels provide both shampoo and conditioner while 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 second day. If the hotel has a pool, we can bring swim suits 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.

underlying hardware in order to take advantage of the more capable systems while compensating for the less capable systems.

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

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

While it may not be obvious, a modern PC system often have two or more CUDA devices. This is because many PC systems come with one or more "integrated" GPUs. These GPUs are the default graphics units and provide rudimentary capabilities and hardware resources to perform minimal graphics functionalities for modern window-based user interfaces. Most CUDA applications will not perform very well on these integrated devices. This would be a reason for the host code to iterate through all the available devices, query their resources and capabilities, and choose the ones that have enough resources to execute the application with satisfactory performance.

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

```
cudaDeviceProp  dev_prop;

for (i = 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 that represent the properties of a CUDA device. The reader is referred to the CUDA C Programming Guide for all the 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 gives 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. It is possible that future devices may even allow more than 1024 threads per block. Therefore, it is a good idea to query the available devices and determine which ones will allow sufficient number of threads in each block as far as the application is concerned.

The number of SMs in the device is given in dev_prop.multiProcessorCount. As we discussed earlier, some devices have only a small number of SMs, e.g. two, and some have much larger number of SMs, e.g. 30. If the application requires a large number of SMs 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 the clock rate and the number of SMs gives a good indication of the hardware execution capacity of the device.

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

There are many more fields in the cudaDeviceProp type. 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 and thus must be discussed in the context of specific hardware implementations. In most implementations to date, once a block is assigned to a Streaming Multiprocessor, it is further divided into 32-thread units called warps. The size of warps is implementation specific. In fact, 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. Figure 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 second warp, and so on. In this example, there are three blocks, Block 1, Block 2 and Block 3, all assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.

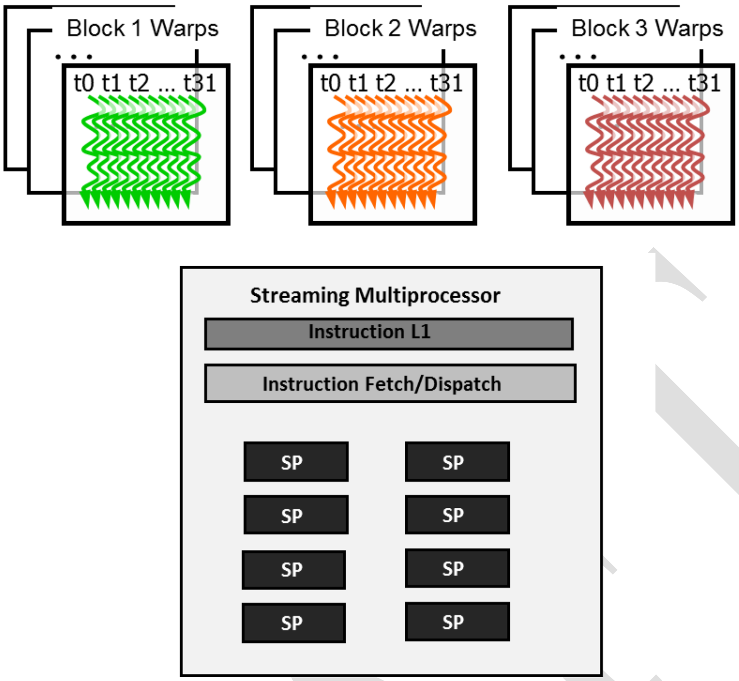

Figure 3.13 Blocks are partitioned into warps for thread scheduling

We can calculate the number of warps that reside in a SM for a given block size and a given number of blocks assigned to each SM. For example, in Figure 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*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. That is, at any instant in time, one instruction is fetched and executed for all threads in the warp. This is illustrated in Figure 3.13 with a single instruction fetch/dispatch shared among execution units (SPs) in the SM. Note that these threads will apply the same instruction to different portions of the data. As a result, all threads in a warp will always have the same execution timing.

Figure 3.13 also shows a number of hardware Streaming Processors (SPs) that actually execute instructions. In general, there are fewer streaming processors than threads assigned to each SM. That is, 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 earlier GPU design, each SM can execute only one instruction for a single warp at any given instant. In more recent designs, each SM can execute instructions for a small number of warps at any given 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).

Latency Tolerance

Latency tolerance is also needed in many everyday situations. For example, in post offices, each person trying to ship a package should ideally have filled out all the forms and labels before going to the service counter. However, as we all have experienced, 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, it is important to maximize the productivity of the service clerks. Letting a person fill out the form in front of the clerk while everyone waits is not a good approach. The clerk should be helping the next 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.

This is why 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 most cases, the first customer will be served as soon as he finishes the form and the clerk finishes serving the current customer, instead of 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.

Note that warp scheduling is also used for tolerating other types of operation latencies such as pipelined floating-point arithmetic and branch instructions. With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long latency operations. The selection of ready warps for execution does not introduce any idle 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 operation latencies is the main reason why GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as CPUs. As a result, GPUs can dedicate more of its chip area to floating-point execution resources.

We are now ready to do 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, since there is a limitation of up to 8 blocks in each SM, we will end up with only 64×8 = 512 threads in each SM. This means that the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long latency operations.

The 16×16 blocks give 256 threads per block. This means that each SM can take 1024/256 = 4 blocks. This is within the 8-block limitation. This is a good configuration since we will have full thread capacity in each SM and 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 the device. Only 16x16 blocks allow maximal number of threads 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 programmer’s responsibility 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

[3] Note that this is an over-simplified exercise. As we will explain in Chapter 4, 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.

programmer to organize threads and their data into hierarchical and multi-dimensional organizations.

Once a grid is launched, its blocks can be assigned to SMs (Streaming Multiprocessors) in arbitrary order, resulting in transparent scalability of CUDA applications. The transparent scalability comes with a limitation: threads in different blocks cannot synchronize with each other. To allow a kernel to maintain transparent scalability, the simple way for threads in different blocks to synchronize with each other is to terminate the kernel and start a new kernel for the activities after the synchronization point.

Threads are assigned to SMs for execution on a block-by-block basis. Each CUDA device imposes a potentially different limitation on the amount of resource available in each SM. For example, each CUDA device has 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 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 number. 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 number. Write a matrix-vector multiplication kernel and the 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 a CUDA device's SM (streaming multiprocessor) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the most 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
(C) 2048
(D) 2096

5. If 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 400x900 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. For 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 micro seconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, 2.9 and spend the rest of their time waiting for the barrier. What percentage of the threads' total execution time 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 1024X1024 matrices 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?