CHAPTER

Performance considerations 5

CHAPTER OUTLINE

The execution speed of a parallel program can vary greatly depending on the resource constraints of the computing hardware. While managing the interaction between parallel code and hardware resource constraints is important for achieving high performance in virtually all parallel programming models, it is a practical skill that is best learned with hands-on exercises in a parallel programming model designed for high performance. In this chapter, we will discuss the major types of resource constraints in a CUDA device and how they can affect the kernel execution performance [Ryoo 2008][CUDA C Best Practice]. In order to achieve his/her goals, a programmer often has to find ways to achieve a required level of performance that is higher than that of an initial version of the application. In different applications, different constraints may dominate and become the limiting factors, commonly referred to as *bottlenecks*. One can often dramatically improve the performance of an application on a particular CUDA device by trading one resource usage for another. This strategy works well if the resource constraint thus alleviated was actually the dominating constraint before the strategy was applied, and the one thus exacerbated does not have negative effects on parallel execution. Without such understanding, performance tuning would be guesswork; plausible strategies may or may not lead to performance enhancements. Beyond insights into these resource constraints, this chapter further offers principles and case studies designed to cultivate intuition about the type of algorithm patterns that can result in high performance execution. It also establishes idioms and ideas that will likely lead to good performance improvements during your performance tuning efforts.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00005-4

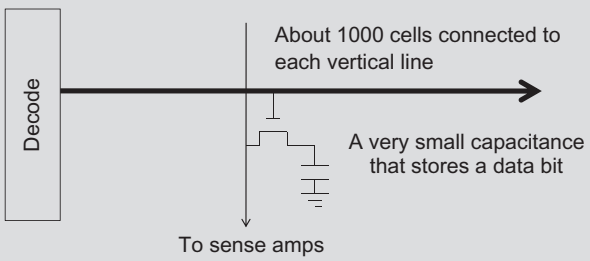
5.1 GLOBAL MEMORY BANDWIDTH

One of the most important factors of CUDA kernel performance is accessing data in the global memory. CUDA applications exploit massive data parallelism. Naturally, CUDA applications tend to process a massive amount of data from the global memory within a short period of time. In Chapter 4, Memory and data locality, we studied tiling techniques that utilize shared memories to reduce the total amount of data that must be accessed from the global memory by a collection of threads in each thread block. In this chapter, we will further discuss memory coalescing techniques that can more effectively move data from the global memory into shared memories and registers. Memory coalescing techniques are often used in conjunction with tiling techniques to allow CUDA devices to reach their performance potential by more efficiently utilizing the global memory bandwidth.[1]

The global memory of a CUDA device is implemented with DRAMs. Data bits are stored in DRAM cells that are small capacitors, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1. Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to a sensor and set off its detection mechanism that determines whether a sufficient amount of charge is present in the capacitor to qualify as a "1" (see "Why is DRAM so slow?" sidebar). This process takes 10s of nanoseconds in modern DRAM chips. This is in sharp contrast with the sub-nanosecond clock cycle time of modern computing devices. Because this is a very slow process relative to the desired data access speed (sub-nanosecond access per byte), modern DRAMs use parallelism to increase their rate of data access, commonly referred to as memory access throughput.

WHY ARE DRAMS SO SLOW?

The following figure shows a DRAM cell and the path for accessing its content. The decoder is an electronic circuit that uses a transistor to drive a line connected to the outlet gates of thousands of cells. It can take a long time for the line to be fully charged or discharged to the desired level.



[1] Recent CUDA devices use on-chip caches for global memory data. Such caches automatically coalesce more of the kernel access patterns and somewhat reduce the need for programmer to manually rearrange their access patterns. However, even with caches, coalescing techniques will continue to have significant effect on kernel execution performance in the foreseeable future.

A more formidable challenge is for the cell to drive the vertical line to the sense amplifiers and allow the sense amplifier to detect its content. This is based on electrical charge sharing. The gate lets out the tiny amount of electrical charge stored in the cell. If the cell content is "1", the tiny amount of charge must raise the electrical potential of the large capacitance of the long bit line to a sufficiently high level that can trigger the detection mechanism of the sense amplifier. A good analogy would be for someone to hold a small cup of coffee at one end of a long hallway for another person to smell the aroma propagated through the hallway to determine the flavor of the coffee.

One could speed up the process by using a larger, stronger capacitor in each cell. However, the DRAMs have been going in the opposite direction. The capacitors in each cell have been steadily reduced in size and thus reduced in their strength over time so that more bits can be stored in each chip. This is why the access latency of DRAMs has not decreased over time.

Each time a DRAM location is accessed, a range of consecutive locations that includes the requested location are actually accessed. Many sensors are provided in each DRAM chip and they work in parallel. Each senses the content of a bit within these consecutive locations. Once detected by the sensors, the data from all these consecutive locations can be transferred at very high-speed to the processor. These consecutive locations accessed and delivered are referred to as DRAM *bursts*. If an application makes focused use of data from these bursts, the DRAMs can supply the data at a much higher rate than if a truly random sequence of locations were accessed.

Recognizing the burst organization of modern DRAMs, current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. That is, the most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations. In this case, the hardware combines, or *coalesces*, all these accesses into a consolidated access to consecutive DRAM locations. For example, for a given load instruction of a warp, if thread 0 accesses global memory location N[2], thread 1 location N+1, thread 2 location N+2, and so on, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAMs. Such coalesced access allows the DRAMs to deliver data as a burst.[3]

[2] Different CUDA devices may also impose alignment requirements on N. For example, in some CUDA devices, N is required to be aligned to 16-word boundaries. That is, the lower 6 bits of N should all be 0 bits. Such alignment requirements have been relaxed in recent CUDA devices due to the presence of 2nd-level caches.

[3] Note that modern CPUs also recognize the DRAM burst organization in their cache memory design. A CPU cache line typically maps to one or more DRAM bursts. Applications that make full use of bytes in each cache line they touch tend to achieve much higher performance than those that randomly access memory locations. The techniques presented in this chapter can be adapted to help CPU programs to achieve high performance.

In order to understand how to effectively use the coalescing hardware, we need to review how the memory addresses are formed in accessing C multidimensional array elements. Recall from Chapter 3, Scalable parallel execution (Fig. 3.3, replicated as Fig. 5.1 for convenience) that multidimensional array elements in C and CUDA are placed into the linearly addressed memory space according to the row-major convention. The term *row major* refers to the fact that the placement of data preserves the structure of rows: all adjacent elements in a row are placed into consecutive locations in the address space. In Fig. 5.1, the four elements of row 0 are first placed in their order of appearance in the row. Elements in row 1 are then placed, followed by elements of row 2, followed by elements of row 3. It should be clear that $M_{0,0}$ and $M_{1,0}$, though appear to be consecutive in the two-dimensional matrix, are placed four locations away in the linearly addressed memory.

Fig. 5.2 illustrates the favorable vs. unfavorable CUDA kernel 2D row-major array data access patterns for memory coalescing. Recall from Fig. 4.7 that in our simple matrix multiplication kernel, each thread accesses a row of the `M` array and

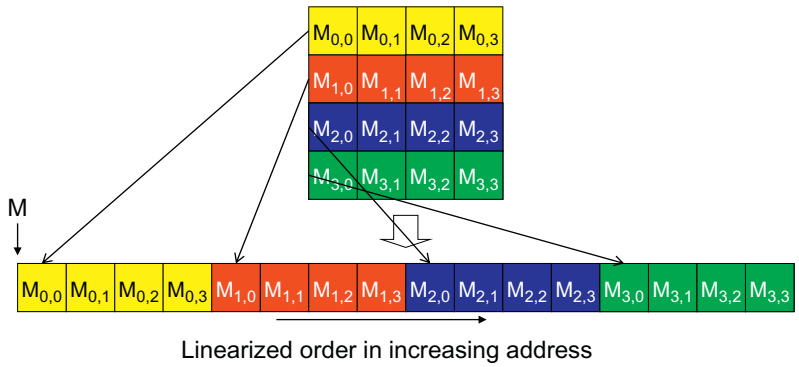


FIGURE 5.1

Placing matrix elements into linear order.

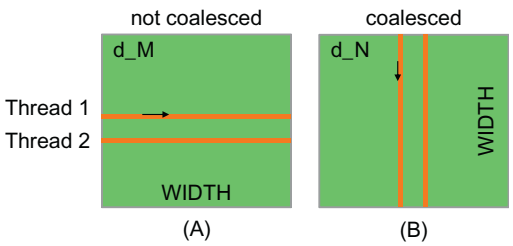


FIGURE 5.2

Memory access patterns in C 2D arrays for coalescing.

a column of the N array. The reader should review Section 4.3 before continuing. Figure 5.2(A) illustrates the data access pattern the M array, where threads in a warp read adjacent rows. That is, during iteration 0, threads in a warp read element 0 of rows 0 through 31. During iteration 1, these same threads read element 1 of rows 0 through 31. None of the accesses will be coalesced. A more favorable access pattern is shown in Fig. 5.2(B), where each thread reads a column of N. During iteration 0, threads in warp 0 read element 1 of columns 0 through 31. All these accesses will be coalesced.

In order to understand why the pattern in Fig. 5.2(B) is more favorable than that in Fig. 5.2(A), we need to review how these matrix elements are accessed in more detail. Fig. 5.3 shows a small example of the favorable access pattern in accessing a 4×4 matrix. The arrow in the top portion of Fig. 5.3 shows the access pattern of the kernel code. This access pattern is generated by the access to N in Fig. 4.3:

```
N[k*Width + Col]
```

Within a given iteration of the k loop, the k*Width value is the same across all threads. Recall that Col=blockIdx.x*blockDim.x+threadIdx.x. Since the value of blockIndx.x and blockDim.x are of the same value for all threads in the same block, the only part of k*Width+Col that varies across a thread block is threadIdx.x. Since adjacent threads have consecutive threadIdx.x values, their accessed elements will have consecutive addresses. For example, in Fig. 5.3, assume that we are using 4×4 blocks and that the warp size is 4. That is, for this toy example, we are using only 1 block to calculate the entire P matrix. The values of Width, blockDim.x,

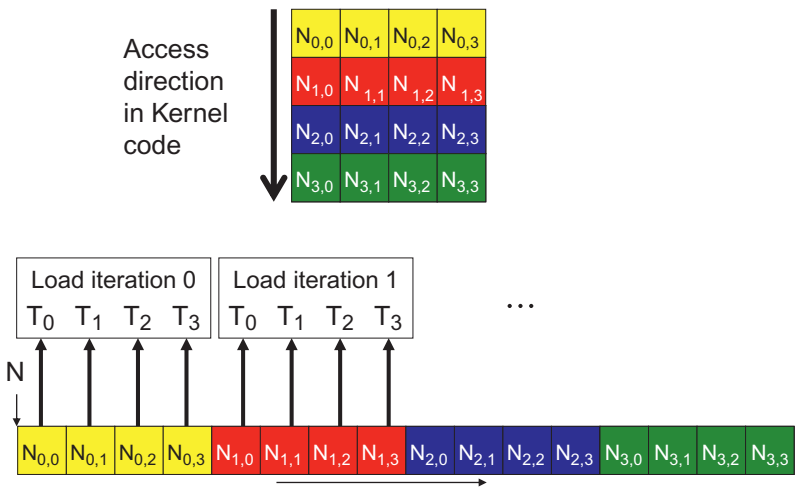


FIGURE 5.3

A coalesced access pattern.

`blockIdx.x` are 4, 4, and 0 for all threads in the block. In iteration 0, the `k` value is 0. The index used by each thread for accessing `N` is

```
N[k*Width+Col]=N[k*Width+blockIdx.x*blockDim.x+threadIdx.x]
              =N[0*4 + 0*4 + threadidx.x]
              =N[threadIdx.x]
```

That is, within this thread block, the index for accessing N is simply the value of `threadIdx.x`. The *N* elements accessed by T_0, T_1, T_2, T_3 are `N[0]`, `N[1]`, `N[2]`, and `N[3]`. This is illustrated with the "Load iteration 0" box of Fig. 5.3. These elements are in consecutive locations in the global memory. The hardware detects that these accesses are made by threads in a warp and to consecutive locations in the global memory. It coalesces these accesses into a consolidated access. This allows the DRAMs to supply data at a high rate.

During the next iteration, the `k` value is 1. The index used by each thread for accessing `N` becomes:

```
N[k*Width+Col] =N[k*Width+blockIdx.x*blockDim.x+threadIdx.x]
               =N[1*4 + 0*4 + threadidx.x]
               =N[4+threadIdx.x]
```

The *N* elements accessed by T_0, T_1, T_2, T_3 in this iteration are `N[5]`, `N[6]`, `N[7]`, and `N[8]`, as shown with the "Load iteration 1" box in Fig. 5.3. All these accesses are again coalesced into a consolidated access for improved DRAM bandwidth utilization.

Fig. 5.4 shows an example of a matrix data access pattern that is not coalesced. The arrow in the top portion of the figure shows that the kernel code for each thread accesses elements of a row in sequence. The arrow in the top portion of Figure 5.4 shows the access pattern of the kernel code for one thread. This access pattern is generated by the access to `M` in Fig. 4.3:

```
M[Row*Width+k]
```

Within a given iteration of the `k` loop, `k*Width` value is the same across all threads. Recall from Fig. 4.3 that `Row=blockIdx.y*blockDim.y+threadIdx.y`. Since the value of `blockIndx.y` and `blockDim.y` are of the same value for all threads in the same block, the only part of `Row*Width+k` that can vary across a thread block is `threadIdx.y`. In Fig. 5.4, we assume again that we are using 4×4 blocks and that the warp size is 4. The values of `Width`, `blockDim.y`, `blockIdx.y` are 4, 4, and 0 for all threads in the block. In iteration 0, the `k` value is 0. The index used by each thread for accessing `M` is:

```
M[Row*Width+k] =M[(blockIdx.y*blockDim.y+threadIdx.y)*Width+k]
               =M[((0*4+threadIdx.y)*4 + 0]
               =M[threadIdx.x*4]
```

That is, the index for accessing `M` is simply the value of `threadIdx.x*4`. The `M` elements accessed by T_0, T_1, T_2, T_3 are `M[0]`, `M[4]`, `M[8]`, and `M[12]`. This is illustrated with the "Load iteration 0" box of Fig. 5.4. These elements are not in

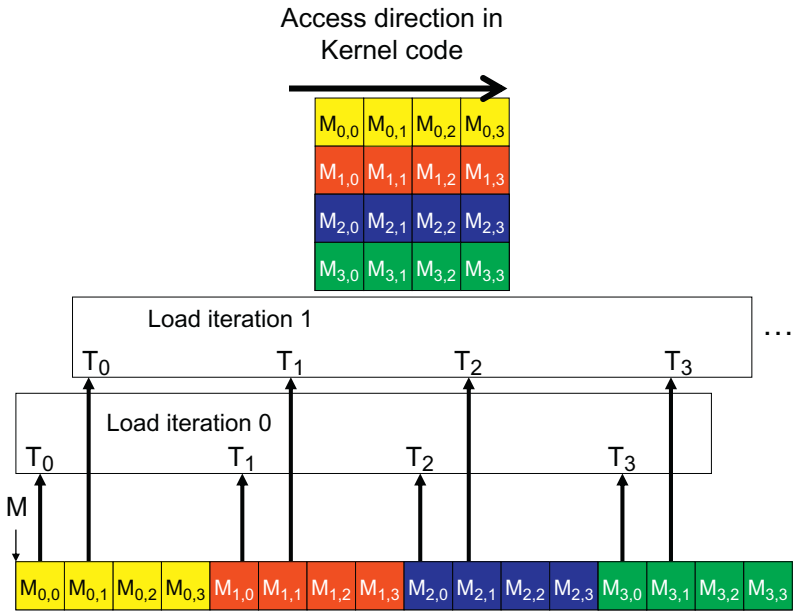


FIGURE 5.4

An un-coalesced access pattern.

consecutive locations in the global memory. The hardware cannot coalesce these accesses into a consolidated access.

During the next iteration, the `k` value is 1. The index used by each thread for accessing `M` becomes:

```
M[Row*Width+k]  =M[(blockIdx.y*blockDim.y+threadIdx.y)*Width+k]
                =M[(0*4+threadidx.x)*4+1]
                =M[threadIdx.x*4+1]
```

The `M` elements accessed by T_0, T_1, T_2, T_3 are `M[1]`, `M[5]`, `M[9]`, and `M[13]`, as shown with the "Load iteration 1" box in Fig. 5.4. Again, these accesses cannot be coalesced into a consolidated access.

For a realistic matrix, there are typically hundreds or even thousands of elements in each dimension. The `M` elements accessed in each iteration by neighboring threads can be hundreds or even thousands of elements apart. The "Load iteration 0" box in the bottom portion shows how the threads access these nonconsecutive locations in the 0^{th} iteration. The hardware will determine that accesses to these elements are far away from each other and cannot be coalesced. As a result, when a kernel loop iterates through a row, the accesses to global memory are much less efficient than the case where a kernel iterates through a column.

If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the shared memory to enable memory coalescing. The

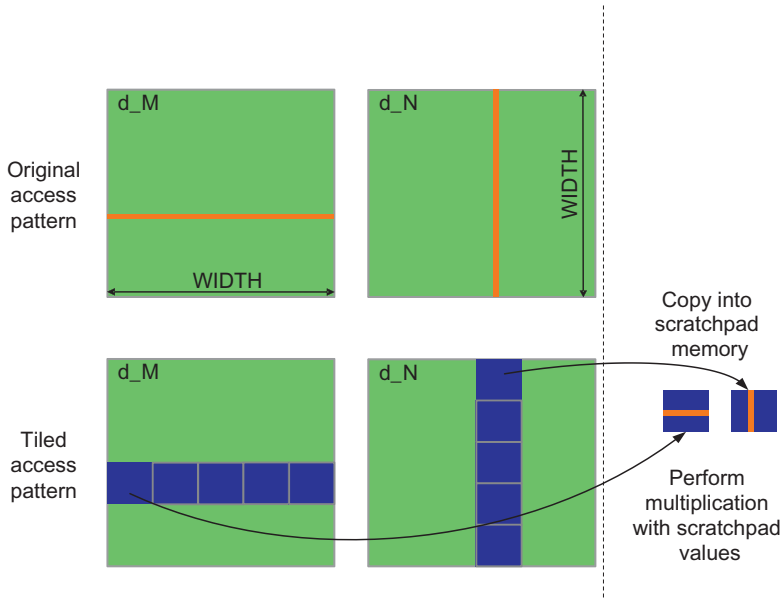


FIGURE 5.5

Using shared memory to enable coalescing.

technique, called *corner turning*, is illustrated in Fig. 5.5 for matrix multiplication. Each thread reads a row from `M`, a pattern that cannot be coalesced. Fortunately, a tiled algorithm can be used to enable coalescing. As we discussed in Chapter 4, Memory and data locality, threads of a block can first cooperatively load the tiles into the shared memory. *Care must be taken to ensure that these tiles are loaded in a coalesced pattern*. Once the data is in shared memory, they can be accessed either on a row basis or a column basis with much less performance variation because the shared memories are implemented as intrinsically high-speed on-chip memory that does not require coalescing to achieve high data access rate.

We replicate Fig. 4.16 here as Fig. 5.6, where the matrix multiplication kernel loads two tiles of matrix `M` and `N` into the shared memory. Recall that at the beginning of each phase (Lines 9–11) each thread in a thread block is responsible for loading one `M` element and one `N` element into `Mds` and `Nds`. Note that there are `TILE_WIDTH`^{2} threads involved in each tile. The threads use `threadIdx.y` and `threadIdx.y` to determine the elements to load.

The `M` elements are loaded in line 9, where the index calculation for each thread uses `ph` to locate the left end of the tile. The linearized index calculation is equivalent to the two-dimensional array access expression `M[Row][ph*TILE_SIZE+tx]`. Note that the column index used by the threads only differs in terms of `threadIdx`. The row index is determined by `blockIdx.y` and `threadIdx.y` (Line 5), which means that threads in the same thread block with identical `blockIdx.y/threadIdx.y` and adjacent `threadIdx.x` values will access adjacent `M` elements. That is, each row of the tile

```
 __global__ void  MatrixMulKernel(float* M, float* N, float* P, int Width)
{

 1.   __shared__ float  Mds[TILE_WIDTH][TILE_WIDTH];
 2.   __shared__ float  Nds[TILE_WIDTH][TILE_WIDTH];

 3.   int bx = blockIdx.x;  int by = blockIdx.y;
 4.   int tx = threadIdx.x; int ty = threadIdx.y;

      // Identify the row and column of the P element to work on
 5.   int Row = by * TILE_WIDTH + ty;
 6.   int Col = bx * TILE_WIDTH + tx;

 7.   float Pvalue = 0;
      // Loop over the M and N tiles required to compute the P element
 8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

        // Collaborative loading of M and N tiles into shared memory
 9.     Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

12.       for (int k = 0; k < TILE_WIDTH; ++k) {
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
          }
14.       __syncthreads();
      }
15.   P[Row*Width + Col] = Pvalue;
    }
```

FIGURE 5.6

Tiled Matrix Multiplication Kernel using shared memory.

is loaded by `TILE_WIDTH` threads whose `threadIdx` are identical in the y dimension and consecutive in the x dimension. The hardware will coalesce these loads.

In the case of `N`, the row index `ph*TILE_SIZE+ty` has the same value for all threads with the same `threadIdx.y` value. The question is whether threads with adjacent `threadIdx.x` values access adjacent `N` elements of a row. Note the column index calculation for each thread, `Col=bx*TILE_SIZE+tx` (see line 6). The first term, `bx*TILE_SIZE`, is the same for all threads in the same block. The second term, `tx`, is simply the `threadIdx.x` value. Therefore, threads with adjacent `threadIdx.x` values access adjacent `N` elements in a row. The hardware will coalesce these loads.

Note that in the simple algorithm, threads with adjacent `threadIdx.x` values access vertically adjacent elements that are not physically adjacent in the row major layout. The tiled algorithm "transformed" this into a different access pattern where threads with adjacent `threadIdx.x` values access horizontally adjacent elements. That is, we turned a vertical access pattern into a horizontal access pattern, which is sometimes referred to as *corner turning*. Corner turning could be also used to turn a horizontal access pattern into a vertical access pattern, which is beneficial in languages such as FORTRAN where 2D arrays are laid out in column-major order.

In the tiled algorithm, loads to both the `M` and `N` elements are coalesced. Therefore, the tiled matrix multiplication algorithm has two advantages over the simple matrix

multiplication. First, the number of memory loads is reduced due to the reuse of data in the shared memory. Second, the remaining memory loads are coalesced so the DRAM bandwidth utilization is further improved. These two improvements have multiplicative effect on each other and result in very significant increased execution speed of the kernel. On a current generation device, the tiled kernel can run more than 30x faster than the simple kernel.

Lines 5, 6, 9, 10 in Fig. 5.6 form a frequently used programming pattern for loading matrix elements into shared memory in tiled algorithms. We would also like to encourage the reader to analyze the data access pattern by the dot-product loop in lines 12 and 13. Note that the threads in a warp do not access consecutive locations of Mds. This is not a problem since Mds is in shared memory, which does not require coalescing to achieve high-speed data access.

5.2 MORE ON MEMORY PARALLELISM

As we explained in Section 5.1, DRAM bursting is a form of parallel organization: multiple locations around are accessed in the DRAM core array in parallel. However, bursting alone is not sufficient to realize the level of DRAM access bandwidth required by modern processors. DRAM systems typically employ two more forms of parallel organization – banks and channels. At the highest level, a processor contains one or more channels. Each channel is a memory controller with a bus that connects a set of DRAM banks to the processor. Fig. 5.7 illustrates a processor that contains four channels, each with a bus that connects four DRAM banks to the processor. In real systems, a processor typically has one to eight channels and each channel is connected to a large number of banks.

The data transfer bandwidth of a bus is defined by its width and clock frequency. Modern *double data rate* (DDR) busses perform two data transfers per clock cycle, one at the rising edge and one at the falling edge of each clock cycle. For example, a 64-bit DDR bus with a clock frequency of 1 GHz has a bandwidth of 8B*2*1 GHz = 16 GB/sec. This seems to be a large number but is often too small for modern CPUs

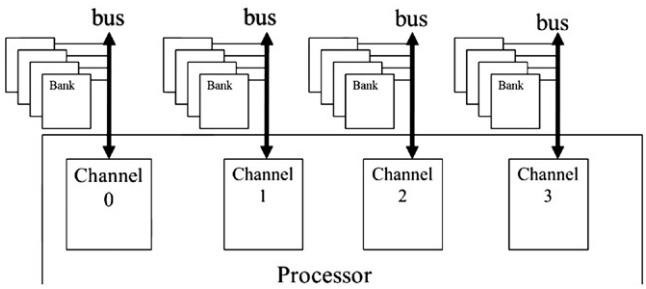


FIGURE 5.7

Channels and banks in DRAM systems.

and GPUs. A modern CPU might require a memory bandwidth of at least 32 GB/s, whereas a modern GPU might require 128 GB/s. For this example, the CPU would require 2 channels and the GPU would require 8 channels.

For each channel, the number of banks connected to it is determined by the number of banks required to fully utilize the data transfer bandwidth of the bus. This is illustrated in Fig. 5.8. Each bank contains an array of DRAM cells, the sensing amplifiers for accessing these cells, and the interface for delivering bursts of data to the bus (Section 5.1).

Fig. 5.8(A) illustrates the data transfer timing when a single bank is connected to a channel. It shows the timing of two consecutive memory read accesses to the DRAM cells in the bank. Recall from Section 5.1 that each access involves a long latency for the decoder to enable the cells and for the cells to share their stored charge with the sensing amplifier. This latency is shown as the light gray section at the left end of the time frame. Once the sensing amplifier completes its work, the burst data is delivered through the bus. The time for transferring the burst data through the bus is shown as the left dark section of the time frame in Fig. 5.8(A). The second memory read access will incur a similar long access latency (light section between the dark sections of the time frame) before its burst data can be transferred (right dark section).

In reality, the access latency (light sections) is much longer than the data transfer time (dark section). It should be apparent that the access-transfer timing of a one-bank organization would grossly underutilize the data transfer bandwidth of the channel bus. For example, if the ratio of DRAM cell array access latency to the data transfer time is 20:1, the maximal utilization of the channel bus would be 1/21 = 4.8%. That is a 16 GB/s channel would deliver data to the processor at a rate no more than 0.76 GB/s. This would be totally unacceptable. This problem is solved by connecting multiple banks to a channel bus.

When two banks are connected to a channel bus, an access can be initiated in the second bank while the first bank is serving another access. Therefore, one can overlap the latency for accessing the DRAM cell arrays. Fig. 5.8(B) shows the timing of a two-bank organization. We assume that the bank 0 started at a time earlier than the window shown in Fig. 5.8(B). Shortly after the first bank starts accessing its cell

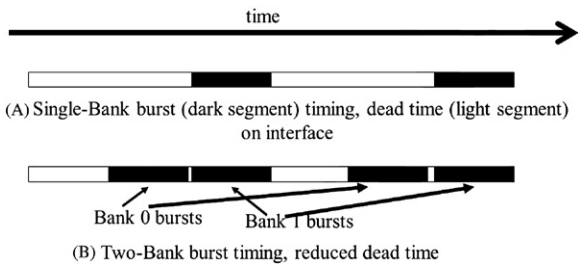


FIGURE 5.8

Banking improves the utilization of data transfer bandwidth of a channel.

array, the second bank also starts accessing its cell array. When the access in bank 0 is complete, it transfers the burst data (leftmost dark section of the time frame). Once bank 0 completes its data transfer, bank 1 can transfer its burst data (second dark section). This pattern repeats for the next accesses.

From Fig. 5.8(B), we can see that by having two banks, we can potentially double the utilization of the data transfer bandwidth of the channel bus. In general, if the ratio of the cell array access latency and data transfer time is `R`, we need to have at least `R+1` banks if we hope to fully utilize the data transfer bandwidth of the channel bus. For example, if the ratio is 20, we will need at least 21 banks connected to each channel bus. In reality, the number of banks connected to each channel bus needs to be larger than `R` for two reasons. One is that having more banks reduces the probability of multiple simultaneous accesses targeting the same bank, a phenomenon called bank conflict. Since each bank can serve only one access at a time, the cell array access latency can no longer be overlapped for these conflicting accesses. Having a larger number of banks increases the probability that these accesses will be spread out among multiple banks. The second reason is that the size of each cell array is set to achieve reasonable latency and manufacturability. This limits the number of cells that each bank can provide. One may need a large number of banks just to be able to support the memory size required.

There is an important connection between the parallel execution of threads and the parallel organization of the DRAM system. In order to achieve the memory access bandwidth specified for device, there must be a sufficient number of threads making simultaneous memory accesses. Furthermore, these memory accesses must be evenly distributed to the channels and banks. Of course, each access to a bank must also be a coalesced access, as we studied in Section 5.1.

Fig. 5.9 shows a toy example of distributing array `M` elements to channels and banks. We assume a small burst size of two elements (eight bytes). The distribution is done by hardware design. The addressing of the channels and banks is such that the first eight bytes of the array (`M[0]` and `M[1]`) are stored in bank 0 of channel 0, the next eight bytes (`M[2]` and `M[3]`) in bank 0 of Channel 1, the next eight bytes (`M[4]` and `M[5]`) in bank 0 of Channel 2, and the next eight bytes (`M[6]` and `M[7]`) in bank 0 of Channel 3.

At this point, the distribution wraps back to Channel 0 but will use bank 1 for the next eight bytes (`M[8]` and `M[9]`). This way, elements `M[10]` and `M[11]` will be in bank 1 of Channel 1, `M[12]` and `M[13]` in bank 1 of Channel 2, and `M[14]` and `M[15]` in bank 1 of Channel 3. Although not shown in the figure, any additional elements will be wrapped around and start with bank 0 of Channel 0. For example, if there are more elements, `M[16]` and `M[17]` will be stored in bank 0 of Channel 0, `M[18]` and `M[19]` will be in bank 0 of Channel 1, and so on.

The distribution scheme illustrated in Fig. 5.9, often referred to as *interleaved data distribution*, spreads the elements across the banks and channels in the system. This scheme ensures that even relatively small arrays are spread out nicely. That is, we only assign enough elements to fully utilize the DRAM burst of bank 0 of Channel 0 before moving on to bank 0 of Channel 1. In our toy example, as long as

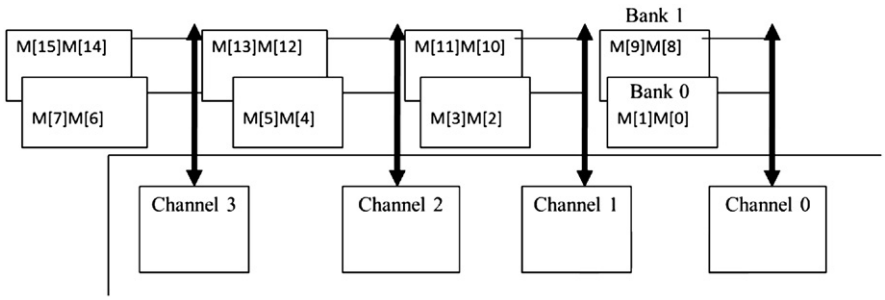


FIGURE 5.9

Distributing array elements into channels and banks.

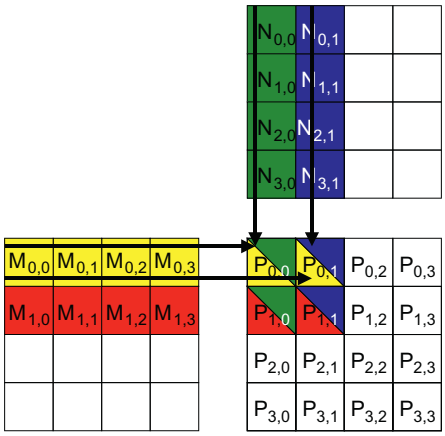


FIGURE 5.10

A small example of matrix multiplication (replicated from Fig. 4.9).

we have at least 16 elements, the distribution will involve all the channels and banks for storing the elements.

We now illustrate the interaction between parallel thread execution and the parallel memory organization. We will use the example in Fig. 4.9, replicated as Fig. 5.10. We assume that the multiplication will be performed with 2×2 thread blocks and 2×2 tiles.

During the phase 0 of the kernel's execution, all four thread blocks will be loading their first tile. The `M` elements involved in each tile are shown in Fig. 5.11. Row 2 shows the `M` elements accessed in Phase 0, with their 2D indices. Row 3 shows the same `M` elements with their linearized indices. Assume that all thread blocks are executed in parallel. We see that each block will make two coalesced accesses.

Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

FIGURE 5.11

M elements loaded by thread blocks in each phase.

According to the distribution in Fig. 5.9, these coalesced accesses will be made to the two banks in channel 0 as well as the two banks in channel 2. These four accesses will be done in parallel to take advantage of two channels as well as improving the utilization of the data transfer bandwidth of each channel.

We also see that $Block_{0,0}$ and $Block_{0,1}$ will load the same `M` elements. Most of the modern devices are equipped with caches that will combine these accesses into one as long as the execution timing of these blocks are sufficiently close to each other. In fact, the cache memories in GPU devices are mainly designed to combine such accesses and reduce the number of accesses to the DRAM system.

Rows 4 and 5 show the `M` elements loaded during phase 1 of the kernel execution. We see that the accesses are now done to the banks in channel 1 and channel 3. Once again, these accesses will be done in parallel. It should be clear to the reader that there is a symbiotic relationship between the parallel execution of the threads and the parallel structure of the DRAM system. On one hand, good utilization of the potential access bandwidth of the DRAM system requires that many threads simultaneously access data that reside in different banks and channels. On the other hand, the execution throughput of the device relies on good utilization of the parallel structure of the DRAM system. For example, if the simultaneously executing threads all access data in the same channel, the memory access throughput and the overall device execution speed will be greatly reduced.

The reader is invited to verify that multiplying two larger matrices, such as 8×8 with the same 2×2 thread block configuration, will make use of all the four channels in Fig. 5.9. Also, an increased DRAM burst size would require multiplication of even larger matrices to fully utilize the data transfer bandwidth of all the channels.

5.3 WARPS AND SIMD HARDWARE

We now turn our attention to aspects of the thread execution that can limit performance. Recall that launching a CUDA kernel generates a grid of threads that is organized as a two-level hierarchy. At the top level, a grid consists of a one-, two-, or three-dimensional array of blocks. At the bottom level, each block, in turn, consists of a one-, two-, or three-dimensional array of threads. In Chapter 3, Scalable parallel execution, we saw that blocks can execute in any order relative to each other, which allows for transparent scalability across different devices. However, we did not say much about the execution timing of threads within each block.

Conceptually, one should assume that threads in a block can execute in any order with respect to each other. In algorithms with phases, barrier synchronizations should be used whenever we want to ensure that all threads have completed a common phase of their execution before any of them start the next phase. We saw such an example in the tiled matrix multiplication kernel. The correctness of executing a kernel should not depend on the fact that certain threads will execute in synchrony with each other. Having said this, we also want to point out that due to various hardware cost considerations, current CUDA devices actually bundle multiple threads for execution. Such implementation strategy leads to performance limitations for certain types of kernel code constructs. It is advantageous for application developers to change these types of constructs to other equivalent forms that perform better.

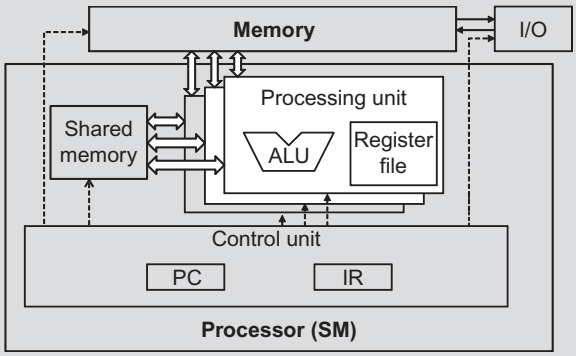
As we discussed in Chapter 3, Scalable parallel execution, each thread block is partitioned into *warps*. The execution of warps is implemented by an SIMD hardware (see "Warps and SIMD Hardware" sidebar). This implementation technique helps to reduce hardware manufacturing cost, lower run-time operation electricity cost, and enable coalescing of memory accesses. In the foreseeable future, we expect that warp partitioning will remain as a popular implementation technique. However, the size of warps can easily vary from implementation to implementation. Up to this point in time, all CUDA devices have used similar warp configurations where each warp consists of 32 threads.

WARPS AND SIMD HARDWARE

The motivation for executing threads as warps is illustrated in the following picture (Same as Fig. 4.8). The processor has only one control unit that fetches and decodes instructions. The same control signal goes to multiple processing units, each of which executes one of the threads in a warp. Since all processing units are controlled by the same instruction, their execution differences are due to the different data operand values in the register files. This is called Single-Instruction-Multiple-Data (SIMD) in processor design. For example, although all processing units are controlled by an instruction:

```
add r1, r2, r3
```

the r2 and r3 values are different in different processing units.



Control units in modern processors are quite complex, including sophisticated logic for fetching instructions and access ports to the instruction memory. They include on-chip instruction caches to reduce the latency of instruction fetch. Having multiple processing units to share a control unit can result in significant reduction in hardware manufacturing cost and power consumption.

As the processors are increasingly power-limited, new processors will likely use SIMD designs. In fact, we may see even more processing units sharing a control unit in the future.

Thread blocks are partitioned into warps based on thread indices. If a thread block is organized into a one-dimensional array, i.e., only `threadIdx.x` is used, the partition is straightforward. `ThreadIdx.x` values within a warp are consecutive and increasing. For warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63. In general, warp n starts with thread $32*n$ and ends with thread $32(n+1) - 1$. For a block whose size is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32-thread positions. For example, if a block has 48 threads, it will be partitioned into two warps, and its warp 1 will be padded with 16 extra threads.

For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linearized row-major order before partitioning into warps. The linear order is determined by placing the rows with larger y and z coordinates after those with lower ones. That is, if a block consists of two dimensions of threads, one would form the linear order by placing all threads whose `threadIdx.y` is 1 after those whose `threadIdx.y` is 0. Threads whose `threadIdx.y` is 2 will be placed after those whose `threadIdx.y` is 1, and so on.

Fig. 5.12 shows an example of placing threads of a two-dimensional block into linear order. The upper part shows the two-dimensional view of the block. The reader

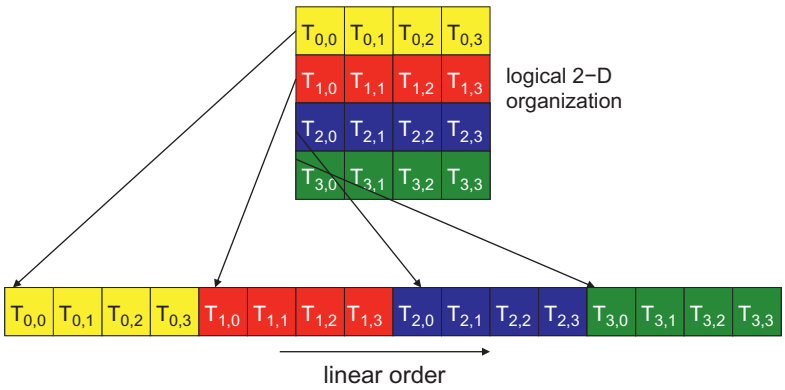


FIGURE 5.12

Placing 2D threads into linear order.

should recognize the similarity with the row-major layout of two-dimensional arrays. Each thread is shown as $T_{y,x}$, x being `threadIdx.x` and y being `threadIdx.y`. The lower part of Fig. 5.12 shows the linearized view of the block. The first four threads are those threads whose `threadIdx.y` value is 0; they are ordered with increasing `threadIdx.x` values. The next four threads are those threads whose `threadIdx.y` value is 1. They are also placed with increasing `threadIdx.x` values. For this example, all 16 threads form half a warp. The warp will be padded with another 16 threads to complete a 32-thread warp. Imagine a two-dimensional block with 8×8 threads. The 64 threads will form two warps. The first warp starts with $T_{0,0}$ and ends with $T_{3,7}$. The second warp starts with $T_{4,0}$ and ends with $T_{7,7}$. It would be a useful exercise to draw out the picture.

For a three-dimensional block, we first place all threads whose `threadIdx.z` value is 0 into the linear order. Among these threads, they are treated as a two-dimensional block as shown in Fig. 5.12. All threads whose `threadIdx.z` value is 1 will then be placed into the linear order, and so on. For a three-dimensional thread block of dimensions 2×8×4 (four in the x dimension, eight in the y dimension, and two in the z dimension), the 64 threads will be partitioned into two warps, with $T_{0,0,0}$ through $T_{0,7,3}$ in the first warp and $T_{1,0,0}$ through $T_{1,7,3}$ in the second warp.

The SIMD hardware executes all threads of a warp as a bundle. An instruction is run for all threads in the same warp. It works well when all threads within a warp follow the same execution path, or more formally referred to as control flow, when working their data. For example, for an if-else construct, the execution works well when either all threads execute the if part or all execute the else part. When threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these divergent paths. One pass executes those threads that follow the if part and another pass executes those that follow the else part. During each pass, the threads that follow the other path are not allowed to take effect. These passes are sequential to each other, thus will add to the execution time.

The multipass approach to divergent warp execution extends the SIMD hardware's ability to implement the full semantics of CUDA threads. While the hardware executes the same instruction for all threads in a warp, it selectively lets the threads take effect in only each pass, allowing every thread to take its own control flow path. This preserves the independence of threads while taking advantage of the reduced cost of SIMD hardware.

When threads in the same warp follow different execution paths, we say that these threads *diverge* in their execution. In the if-else example, divergence arises if some threads in a warp take the if path and some the else path. The cost of divergence is the extra pass the hardware needs to take in order to allow the threads in a warp to make their own decisions.

Divergence also can arise in other constructs, for example, if threads in a warp execute a `for`-loop which can iterate six, seven, or eight times for different threads. All threads will finish the first six iterations together. Two passes will be used to execute the 7th iteration, one for those that take the iteration and one for those that do not. Two passes will be used to execute the 8th iteration, one for those that take the iteration and one for those that do not.

One can determine if a control construct can result in thread divergence by inspecting its decision condition. If the decision condition is based on `threadIdx` values, the control statement can potentially cause thread divergence. For example, the statement `if (threadIdx.x > 2) {}` causes the threads to follow two divergent control flow paths. Threads 0, 1, and 2 follow a different path than threads 3, 4, 5, etc. Similarly, a loop can cause thread divergence if its loop condition is based on thread index values.

A prevalent reason for using a control construct with thread divergence is handling boundary conditions when mapping threads to data. This is usually because the total number of threads needs to be a multiple of the block size whereas the size of the data can be an arbitrary number. Starting with our vector addition kernel in Fig. 2.12, we had an `if (i<n)` statement in `addVecKernel`. This is because not all vector lengths can be expressed as multiples of the block size. For example, assume that the vector length is 1003. Assume that we picked 64 as block size. One would need to launch 16 thread blocks to process all the 1003 vector elements. However, the 16 thread blocks would have 1024 threads. We need to disable the last 21 threads in thread block 15 from doing work not expected/allowed by the original program. Keep in mind that these 16 blocks are partitioned into 32 warps. Only the last warp will have control divergence.

Note that the performance impact of control divergence decreases with the size of the vectors being processed. For a vector length of 100, one of the four warps will have control divergence, which can have significant impact on performance. For a vector size of 1000, only one out of the 32 warps will have control divergence. That is, control divergence will affect only about 3% of the execution time. Even if it doubles the execution time of the warp, the net impact to the total execution time will be about 3%. Obviously, if the vector length is 10,000 or more, only one of the 313 warps will have control divergence. The impact of control divergence will be much less than 1%!

For two-dimensional data, such as the color-to-greyscale conversion example, if-statements are also used to handle the boundary conditions for threads that operate at the edge of the data. In Fig. 3.2, to process the 76×62 picture, we used 20 = 5*4 two-dimensional blocks that consist of 16×16 threads each. Each block will be partitioned into 8 warps, each one consists of two rows of a block. There are a total 160 warps (8 warps per block) involved.

To analyze the impact of control divergence, refer to Fig. 3.5. None of the warps in the 12 blocks in region 1 will have control divergence. There are 12*8 = 96 warps in region 1. For region 2, all the 24 warps will have control divergence. For region 3, note that all the bottom warps are mapped to data that are completely outside the picture. As a result, none of them will pass the if-condition. The reader should verify that these warps would have had control divergence if the picture had an odd number of pixels in the vertical dimension. Since they all follow the same control flow path, none of these 32 warps will have control divergence! In region 4, the first seven warps will have control divergence but the last warp will not. All in all, 31 out of the 160 warps will have control divergence.

Once again, the performance impact of control divergence decreases as the number of pixels in the horizontal dimension increases. For example, if we process a 200×150 picture with 16×16 blocks, there will be a total 130 = 13*10 thread blocks or 1040 warps. The number of warps in regions 1 through 4 will be 864 (12*9*8), 72 (9*8), 96 (12*8), and 8 (1*8). Only 80 of these warps will have control divergence. Thus, the performance impact of control divergence will be less than 8%. Obviously, if we process a realistic picture with more than 1000 pixels in the horizontal dimension, the performance impact of control divergence will be less than 2%.

Control divergence also naturally arises in some important parallel algorithms where the number of threads participating in the computation varies over time. We will use a reduction algorithm to illustrate such behavior.

A reduction algorithm derives a single value from an array of values. The single value could be the sum, the maximal value, the minimal value, etc., among all elements. All these types of reductions share the same computation structure. A reduction can be easily done by sequentially going through every element of the array. When an element is visited, the action to take depends on the type of reduction being performed. For a sum reduction, the value of the element being visited at the current step, or the current value, is added to a running sum. For a maximal reduction, the current value is compared to a running maximal value of all the elements visited so far. If the current value is larger than the running maximal, the current element value becomes the running maximal value. For a minimal reduction, the value of the element currently being visited is compared to a running minimal. If the current value is smaller than the running minimal, the current element value becomes the running minimal. The sequential algorithm ends when all the elements are visited.

The sequential reduction algorithm is work efficient in that every element is only visited once and only a minimal amount of work is performed when each element is visited. Its execution time is proportional to the number of elements involved. That is, the computational complexity of the algorithm is $O(N)$, where N is the number of elements involved in the reduction.

The time needed to visit all elements of a large array motivates parallel execution. A parallel reduction algorithm typically resembles the structure of a soccer tournament. In fact, the elimination process of the World Cup is a reduction of "maximal" where the maximal is defined as the team that "beats" all other teams. The tournament "reduction" is done in multiple rounds. The teams are divided into pairs. During the first round, all pairs play in parallel. Winners of the first round advance to the second round, whose winners advance to the third round, etc. With 16 teams entering a tournament, eight winners will emerge from the first round, four from the second round, two from the third round, and one final winner from the fourth round.

It should be easy to see that even with 1024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough soccer fields to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on. With enough fields, even with sixty thousand teams, we can determine the final winner in just 16 rounds. Of course, one would need to have enough soccer fields and enough officials to accommodate the thirty thousand games in the first round, etc.

Fig. 5.13 shows a kernel function that performs parallel sum reduction. The original array is in the global memory. Each thread block reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction on these elements. The code loads the elements of the input array `X` from global memory into the shared memory. The reduction is done *in place*, which means some of the elements in the shared memory will be replaced by partial sums. Each iteration of the for-loop in the kernel function implements a round of reduction.

The `__syncthreads()` statement (Line 5) in the for-loop ensures that all partial sums for the previous iteration have been generated and before any one of the threads is allowed to begin the current iteration. This way, all threads that enter the second iteration will be using the values produced in the first iteration. After the first round, the even elements will be replaced by the partial sums generated in the first round. After the second round, the elements whose indices are multiples of four will be replaced with the partial sums. After the final round, the total sum of the entire section will be in `partialSum[0]`.

```
1. __shared__ float partialSum[SIZE];
   partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
4. {
5.   __syncthreads();
6.   if (t % (2*stride) == 0)
7.     partialSum[t] += partialSum[t+stride];
8. }
```

FIGURE 5.13

A simple sum reduction kernel.

In Fig. 5.13, Line 3 initializes the `stride` variable to 1. During the first iteration, the if-statement in Line 6 is used to select only the even threads to perform addition between two neighboring elements. The execution of the kernel is illustrated in Fig. 5.14. The threads and the array element values are shown in the horizontal direction. The iterations taken by the threads are shown in the vertical direction with time progressing from top to bottom. Each row of Fig. 5.14 shows the contents of the array elements after an iteration of the for-loop.

As shown in Fig. 5.16, the even elements of the array hold the pair-wise partial sums after iteration 1. Before the second iteration, the value of the `stride` variable is doubled to 2. During the second iteration, only those threads whose indices are multiples of four will execute the add-statement in Line 7. Each thread generates a partial sum of four elements, as shown in row 2. With 512 elements in each section, the kernel function will generate the sum of the entire section after 9 iterations. By using `blockDim.x` as the loop bound in Line 4, the kernel assumes that it is launched with the same number of threads as the number of elements in the section. That is, for section size of 512, the kernel needs to be launched with 512 threads.[4]

Let us analyze the total amount of work done by the kernel. Assume that the total number of elements to be reduced is N. The first round requires $N/2$ additions. The second round requires $N/4$ additions. The final round has only one addition. There are $\log_2(N)$ rounds. The total number of additions performed by the kernel is $N/2 + N/4 + N/8 + \ldots + 1 = N-1$. Therefore, the computational complexity of the reduction algorithm is $O(N)$. The algorithm is work efficient. However, we also need to make sure that the hardware is efficiently utilized while executing the kernel.

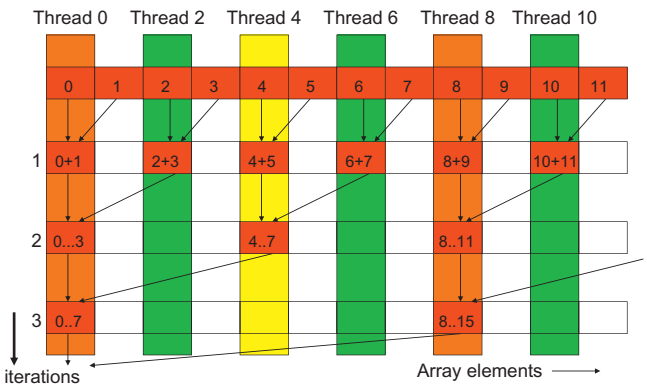


FIGURE 5.14

Execution of the sum reduction kernel.

[4] Note that using the same number of threads as the number of elements in a section is wasteful. Half of the threads in a block will never execute. The reader is encouraged to modify the kernel and the kernel launch execution configuration parameters to eliminate this waste (Exercise 5.1).

The kernel in Fig. 5.13 clearly has thread divergence. During the first iteration of the loop, only those threads whose `threadIdx.x` are even will execute the add-statement. One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute Line 7. In each successive iteration, fewer threads will execute Line 7 but two passes will be still needed to execute all the threads during each iteration. This divergence can be reduced with a slight change to the algorithm.

Fig. 5.15 shows a modified kernel with a slightly different algorithm for sum reduction. Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other. It does so by initializing the stride to be half the size of the section. All pairs added during the first round are half the section size away from each other. After the first iteration, all the pair-wise sums are stored in the first half of the array, as shown in Fig. 5.16. The loop divides the stride by 2 before entering the next iteration. Thus for the second iteration, the `stride` variable

```
1.  __shared__ float partialSum[SIZE];
    partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];

2.  unsigned int t = threadIdx.x;
3.  for (unsigned int stride = blockDim.x/2; stride >= 1; stride = stride>>1)
4.  {
5.    __syncthreads();
6.    if (t < stride)
7.      partialSum[t] += partialSum[t+stride];
8.  }
```

FIGURE 5.15

A kernel with fewer thread divergence.

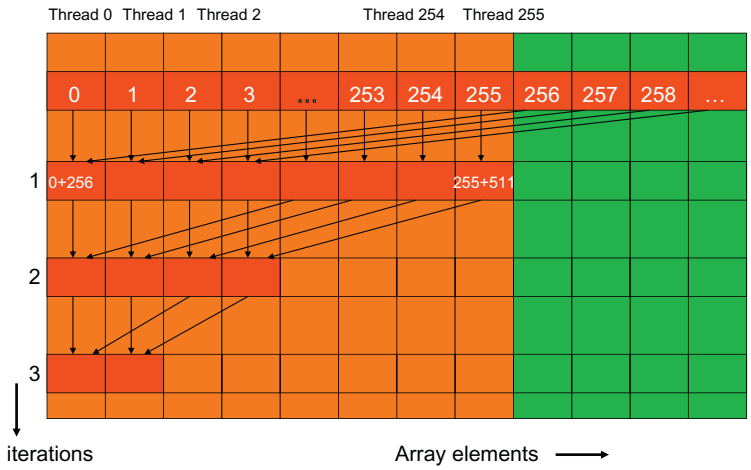


FIGURE 5.16

Execution of the revised algorithm.

value is one-quarter of the section size. That is, the threads add elements that are quarter a section away from each other during the second iteration.

Note that the kernel in Fig. 5.15 still has an if-statement (Line 6) in the loop. The number of threads that execute Line 7 in each iteration is the same as in Fig. 5.13. So, why should there be a performance difference between the two kernels? The answer lies in the positions of threads that execute Line 7 relative to those that do not.

Fig. 5.16 illustrates the execution of the revised kernel in Fig. 5.15. During the first iteration, all threads whose `threadIdx.x` values are less than half of the size of the section execute Line 7. For a section of 512 elements, Threads 0 through 255 execute the add-statement during the first iteration while threads 256 through 511 do not. The pair-wise sums are stored in elements 0 through 255 after the first iteration. Since the warps consists of 32 threads with consecutive `threadIdx.x` values, all threads in warp 0 through warp 7 execute the add-statement, whereas warp 8 through warp 15 all skip the add-statement. Since all threads in each warp take the same path, there is no thread divergence!

The kernel in Fig. 5.15 does not completely eliminate the divergence caused by the if-statement. The reader should verify that starting with the 5th iteration, the number of threads that execute Line 7 will fall below 32. That is, the final five iterations will have only 16, 8, 4, 2, and 1 thread(s) performing the addition. This means that the kernel execution will still have divergence in these iterations. However, the number of iterations of the loop that has divergence is reduced from ten to five.

The difference between Figs. 5.13 and 5.15 is small but has very significant performance impact. It requires someone with clear understanding of the execution of threads on the SIMD hardware of the device to be able to confidently make such adjustments.

5.4 DYNAMIC PARTITIONING OF RESOURCES

The execution resources in an SM include registers, shared memory, thread block slots, and thread slots. These resources are dynamically partitioned and assigned to threads to support their execution. In Chapter 3, Scalable parallel execution, we have seen that Fermi generation of devices have 1536 thread slots. These thread slots are partitioned and assigned to thread blocks during runtime. If each thread block consists of 512 threads, the 1536 thread slots are partitioned and assigned to three blocks. In this case, each SM can accommodate up to three thread blocks due to limitations on thread slots.

If each thread block contains 256 threads, the 1536 thread slots are partitioned and assigned to 6 thread blocks. The ability to dynamically partition the thread slots among thread blocks makes SMs versatile. They can either execute many thread blocks each having few threads, or execute few thread blocks each having many threads. This is in contrast to a fixed partitioning method where each block receives a fixed amount of resources regardless of their real needs. Fixed partitioning results in wasted thread slots when a block has few threads and fails to support blocks that require more thread slots than the fixed partition allows.

Dynamic partitioning of resources can lead to subtle interactions between resource limitations, which can cause underutilization of resources. Such interactions can occur between block slots and thread slots. For example, if each block has 128 threads, the 1536 thread slots can be partitioned and assigned to 12 blocks. However, since there are only 8 block slots in each SM, only 8 blocks will be allowed. This means that in the end, only 1024 of the thread slots will be utilized. Therefore, to fully utilize both the block slots and thread slots, one needs at least 256 threads in each block.

As we mentioned in Chapter 4, Memory and data locality, the automatic variables declared in a CUDA kernel are placed into registers. Some kernels may use lots of automatic variables and others may use few of them. Thus, one should expect that some kernels require many registers and some require fewer. By dynamically partitioning the registers among blocks, the SM can accommodate more blocks if they require few registers, and fewer blocks if they require more registers. One does, however, need to be aware of potential interactions between register limitations and other resource limitations.

In the matrix multiplication example, assume that each SM has 16,384 registers and the kernel code uses 10 registers per thread. If we have 16×16 thread blocks, how many threads can run on each SM? We can answer this question by first calculating the number of registers needed for each block, which is 10*16*16 = 2560. The number of registers required by six blocks is 15,360, which is under the 16,384 limit. Adding another block would require 17,920 registers, which exceeds the limit. Therefore, the register limitation allows six blocks that altogether have 1536 threads to run on each SM, which also fits within the limit of 8 block slots and 1536 thread slots.

Now assume that the programmer declares an additional two automatic variables in the kernel and bumps the number of registers used by each thread to 12. Assuming the same 16×16 blocks, each block now requires 12*16*16 = 3072 registers. The number of registers required by six blocks is now 18,432, which exceeds the register limitation for some CUDA hardware. The CUDA runtime system deals with this situation by reducing the number of blocks assigned to each SM by one, thus reducing the number of registers required to 15,360. This, however, reduces the number of threads running on an SM from 1536 to 1280. That is, by using two extra automatic variables, the program saw a 1/6 reduction in the warp parallelism in each SM. This is sometimes a referred to as a "performance cliff" where a slight increase in resource usage can result in significant reduction in parallelism and performance achieved [RRS 2008].

Shared memory is another resource that is dynamically partitioned at run-time. Tiled algorithms often require a large amount of shared memory to be effective. Unfortunately, large shared memory usage can reduce the number of thread blocks running on an SM. As we discussed in Section 5.3, reduced thread parallelism can negatively affect the utilization of the memory access bandwidth of the DRAM system. The reduced memory access throughput, in turn, can further reduce the thread execution throughput. This is a pitfall that can result in disappointing performance of tiled algorithms and should be carefully avoided.

It should be clear to the reader that the constraints of all the dynamically partitioned resources interact with each other in a complex manner. Accurate determination of the number of threads running in each SM can be difficult. The reader is

referred to the CUDA Occupancy Calculator [NVIDIA], which is a downloadable Excel sheet that calculates the actual number of threads running on each SM for a particular device implementation given the usage of resources by a kernel.

5.5 THREAD GRANULARITY

An important algorithmic decision in performance tuning is the granularity of threads. It is sometimes advantageous to put more work into each thread and use fewer threads. Such advantage arises when some redundant work exists between threads. In the current generation of devices, each SM has limited instruction processing bandwidth. Every instruction consumes instruction processing bandwidth, whether it is a floating-point calculation instruction, a load instruction, or a branch instruction. Eliminating redundant work can ease the pressure on the instruction processing bandwidth and improve the overall execution speed of the kernel.

Fig. 5.17 illustrates such an opportunity in matrix multiplication. The tiled algorithm in Fig. 5.6 uses one thread to compute one element of the output `P` matrix. This requires a dot-product between one row of `M` and one column of `N`.

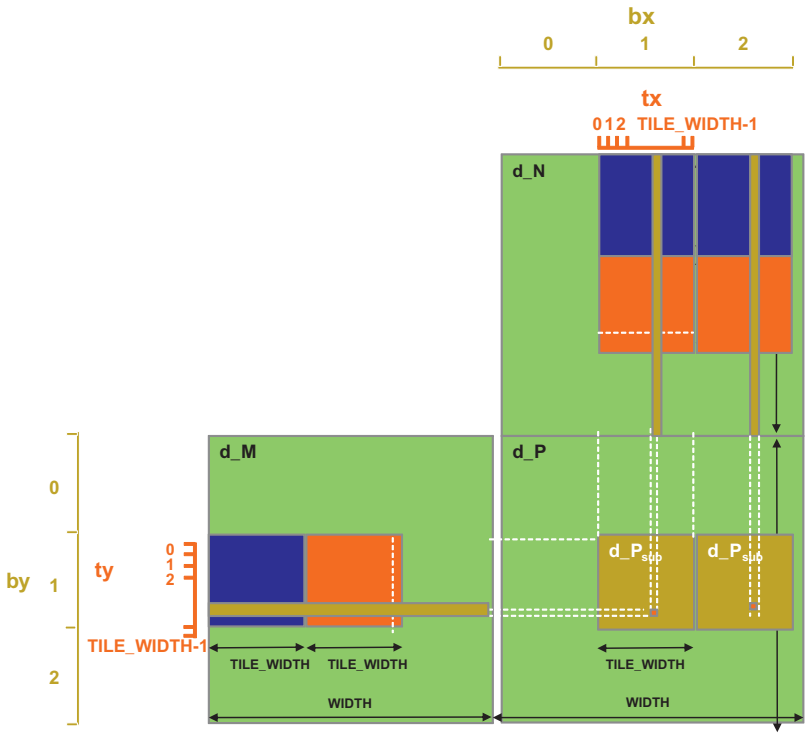


FIGURE 5.17

Increased thread granularity with rectangular tiles.

The opportunity of thread granularity adjustment comes from the fact that multiple blocks redundantly load each M tile. This was also demonstrated in Fig. 5.11. As shown in Fig. 5.17, the calculation of two P elements in adjacent tiles uses the same M row. With the original tiled algorithm, the same M row is redundantly loaded by the two blocks assigned to generate these two P tiles. One can eliminate this redundancy by merging the two thread blocks into one. Each thread in the new thread block now calculates two P elements. This is done by revising the kernel so that two dot-products are computed by the innermost loop of the kernel. Both dot-products use the same Mds row but different Nds columns. This reduces the global memory access by one-quarter. The reader is encouraged to write the new kernel as an exercise.

The potential downside is that the new kernel now uses even more registers and shared memory. As we discussed in the previous section, the number of blocks that can be running on each SM may decrease. For a given matrix size, this also reduces the total number of thread blocks by half, which may result in an insufficient amount of parallelism for matrices of smaller dimensions. In practice, combining up to four adjacent horizontal blocks to compute adjacent horizontal tiles significantly improves the performance of large (2048×2048 or more) matrix multiplication.

5.6 SUMMARY

In this chapter, we reviewed the major aspects of application performance on a CUDA device: global memory access coalescing, memory parallelism, control flow divergence, dynamic resource partitioning and instruction mixes. Each of these aspects is rooted in the hardware limitations of the devices. With these insights, the reader should be able to reason about the performance of any kernel code he/she comes across.

More importantly, we need to be able to convert poor performing code into well performing code. As a starting point, we presented practical techniques for creating good program patterns for these performance aspects. We will continue to study practical applications of these techniques in the parallel computation patterns and application case studies in the next few chapters.

5.7 EXERCISES

1. The kernels in Figs. 5.13 and 5.15 are wasteful in their use of threads; half of the threads in each block never execute. Modify the kernels to eliminate such waste. Give the relevant execute configuration parameter values at the kernel launch. Is there a cost in terms of extra arithmetic operation needed? Which resource limitation can be potentially addressed with such modification?

(Hint: (1) Line 2 and/or Line 3 can be adjusted in each case. (2) The number of elements in a section may need to increase.)

2. Compare the modified kernels you wrote for Exercise 5.1. Which kernel incurred fewer additional arithmetic operations from the modification?

3. Write a complete kernel based on Exercise 5.1 by (1) adding the statements that load a section of the input array from global memory to shared memory, (2) using `blockIdx.x` to allow multiple blocks to work on different sections of the input array, (3) writing the reduction value for the section to a location according to the `blockIdx.x` so that all blocks will deposit their section reduction value to the lower part of the input array in global memory.

4. Design a reduction program based on the kernel you wrote for Exercise 5.3. The host code should (1) transfer a large input array to the global memory, (2) use a loop to repeatedly invoke the kernel you wrote for Exercise 5.3 with adjusted execution configuration parameter values so that the reduction result for the input array will eventually be produced.

5. For the tiled matrix multiplication kernel in Fig. 5.6, draw the access patterns of threads in a warp of Lines 9 and 10 for a small 16×16 matrix size. Calculate the `tx` values and `ty` values for each thread in a warp and use these values in the `M` and `N` index calculations in Lines 9 and 10. Show that the threads indeed access consecutive `M` and `N` locations in global memory during each iteration.

6. For the simple matrix multiplication ($P = M*N$) based on row-major layout, which input matrix will have coalesced accesses?

A. `M`
B. `N`
C. `M, N`
D. Neither

7. For the tiled matrix–matrix multiplication ($M*N$) based on row-major layout, which input matrix will have coalesced accesses?

A. `M`
B. `N`
C. `M, N`
D. Neither

8. For the simple reduction kernel, if the block size is 1024 and warp size is 32, how many warps in a block will have divergence during the 5th iteration?

A. 0
B. 1
C. 16
D. 32

9. For the improved reduction kernel, if the block size is 1024 and warp size is 32, how many warps will have divergence during the 5th iteration?
   A. 0
   B. 1
   C. 16
   D. 32

10. Write a matrix multiplication kernel function that corresponds to the design illustrated in Figure 5.17.

11. For tiled matrix multiplication out of the possible range of values for `BLOCK_SIZE`, for what values of `BLOCK_SIZE` will the kernel completely avoid un-coalesced accesses to global memory? (You need to consider only square blocks.)

12. In an attempt to improve performance, a bright young engineer changed the reduction kernel into the following. (A) Do you believe that the performance will improve? Why or why not? (B) Should the engineer receive a reward or a lecture? Why?

```
__shared__ float partialSum[];
unsigned int tid=threadIdx.x;
for (unsigned int stride=n>>1; stride >= 32; stride >>= 1) {
     __syncthreads();
     if (tid < stride)
          shared[tid] += shared[tid + stride];
}
__syncthreads();
if (tid < 32) {    // unroll last 5 predicated steps
     shared[tid] += shared[tid + 16];
     shared[tid] += shared[tid + 8];
     shared[tid] += shared[tid + 4];
     shared[tid] += shared[tid + 2];
     shared[tid] += shared[tid + 1];
}
```

REFERENCES

CUDA C Best Practices Guide v. 4.2, January 2012.

CUDA Occupancy Calculator. Web search using keywords “CUDA Occupancy Calculator”.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W. February 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming.

Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., Hwu, W. April 6–9, 2008. Program optimization space pruning for a multithreaded GPU. Proceedings of the 6th ACM/IEEE international symposium on code generation and optimization.