Chapter 4

Memory and Data Locality

Keywords: memory bandwidth, memory-bound, on-chip memory, tiling, strip-mining, shared memory, private memory, scope, lifetime, occupancy

CHAPTER OUTLINE

4.1. Importance of Memory Access Efficiency
4.2. Matrix Multiplication
4.3. CUDA Memory Types
4.4. Tiling for Reduced Memory Traffic
4.5. A Tiled Matrix Multiplication Kernel
4.6. Boundary Checks
4.7. Memory as a Limiting Factor to Parallelism
4.8. Summary
4.9. Exercises

So far, we have learned how to write a CUDA kernel function and how to configure and coordinate its execution by a massive number of threads. In this chapter, we will begin to study how one can organize and position the data for efficient access by a massive number of threads. We learned in Chapter 2 that the data are first transferred from the host memory to the device global memory. In Chapter 3, we studied how to direct the threads to access their portion of the data from the global memory using their block indexes and thread indexes. We have also learned more details about resource assignment and thread scheduling. Although this is a very good start, the CUDA kernels that we have learned so far will likely achieve only a tiny fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that global memory, which is typically implemented with DRAM, tends to have long access latencies (hundreds of clock cycles) and finite access bandwidth. While having many threads available for execution can theoretically tolerate long memory access latencies, one can easily run into a situation where traffic congestion in the global memory access paths prevents all but very few threads from making progress, thus rendering some of the Streaming Multiprocessors (SMs) idle. In order to circumvent such congestion, CUDA provides a number of additional resources and methods for accessing memory that can remove the majority of traffic to and from the global memory. In this chapter, you will learn to use different memory types to boost the execution efficiency of CUDA kernels.

4.1. Importance of Memory Access Efficiency

We can illustrate the effect of memory access efficiency by calculating the expected performance level of the most executed portion of the image blur kernel code in Figure 3.8, replicated in Figure 4.1. The most important part of the kernel in terms of execution time is the nested for-loop that performs pixel value accumulation with the blurring patch.

```
    for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
      // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol];
          pixels++; // Keep track of number of pixels in the avg
        }
      }
    }
```

Figure 4.1*: The most executed part of the image blurring kernel in Figure 3.8.*

In every iteration of the inner loop, one global memory access is performed for one floating-point addition. The global memory access fetches an in[] array element. The floating-point add operation accumulates the value of the in[] array element into pixVal. Thus, the ratio of floating-point calculation to global memory access operation is 1 to 1, or 1.0. We will refer to this ratio as the *compute-to-global-memory-access ratio*, defined as the number of floating-point calculations performed for each access to the global memory within a region of a program.

The compute-to-global-memory-access ratio has major implications on the performance of a CUDA kernel. In a high-end device today, the global memory bandwidth is around 1000 GB/s, or 1 TB/s. With four bytes in each single-precision floating-point value, one can expect to load no more than 1000/4=250 giga single-precision operands per second. With a compute-to-global-memory ratio of 1.0, the execution of the image blur kernel will be limited by the rate at which the operands (e.g., the elements of in[])can be delivered to the GPU. We will refer to programs whose execution speed is limited by memory access throughput as *memory bound* programs. In our example, the kernel will achieve no more than 250 giga floating-point operations per second (GFLOPS).

While 250 GFLOPS is a respectable number, it is only a tiny fraction (2%) of the peak single-precision performance of 12 TFLOPS or higher for these high-end devices. In order to achieve a higher level of performance for the kernel, we need to increase the ratio by reducing the number of global memory accesses. To achieve the peak 12 TFLOPS rating of the processor, we need a ratio of 48 or higher. In general, the desired ratio has been increasing in the past few generations of devices as computational throughput has been increasing faster than memory bandwidth. The rest of this chapter introduces a commonly used technique for reducing the number of global memory accesses.

4.2. Matrix Multiplication

Matrix-matrix multiplication, or matrix multiplication in short, between an i×j (i rows by j columns) matrix M and a j×k matrix N produces an i×k matrix P. Matrix multiplication is an important component of the Basic Linear Algebra Subprograms (BLAS) standard (see "Linear Algebra Functions" sidebar in Chapter 3). It is the basis of many linear algebra solvers such as LU decomposition. As we will see, matrix multiplication presents opportunities for reduction of global memory accesses that can be captured with relatively simple techniques. The execution speed of matrix multiplication functions can vary by orders of magnitude depending on the level of reduction of global memory accesses. Therefore, matrix multiplication provides an excellent initial example for such techniques.

When performing a matrix multiplication, each element of the output matrix P is an inner product of a row of M and a column of N. We will continue to use the convention where $P_{Row,Col}$ is the element at Row^{th} position in the vertical direction and Col^{th} position in the horizontal direction. As shown in Figure 4.2, $P_{Row, Col}$ (the small square in P) is the inner product of the vector formed from the Row^{th} row of M (shown as a horizontal strip in M) and the vectorformed from the Col^{th} column of N (shown as a vertical strip in N). The inner product, sometimes called the dot product, of two vectors is the sum of products of the individual vector elements. That is, $P_{Row,Col}=\sum M_{Row,k} * N_{k,Col,}$ for k = 0, 1, … Width-1. For example,

$$P_{1,5} = M_{1,0}*N_{0,5} + M_{1,1}*N_{1,5} + M_{1,2}*N_{2,5} + ... + M_{1,Width-1}*N_{Width-1,5}$$

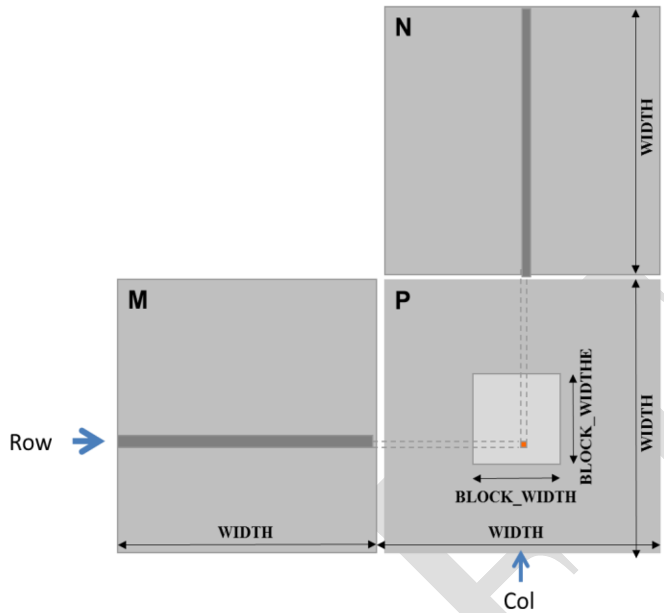

Figure 4.2: *Matrix multiplication using multiple blocks by tiling P*

In our initial matrix multiplication implementation, we map threads to elements of P with the same approach that we used for colorToGreyscaleConversion. That is, each thread is responsible for calculating one P element. The row and column indices for the P element to be calculated by each thread are:

```
Row = blockIdx.y*blockDim.y+threadIdx.y
```

and

```
Col = blockIdx.x*blockDim.x+threadIdx.x.
```

With this one-to-one mapping, the Row and Col thread indices are also the row and column indices for output array. Figure 4.3 shows the source code of the kernel based on this thread-to-data mapping. The reader should immediately see the familiar pattern of calculating Row, Col, and the if-statement testing if Row and Col are both within range. These statements are almost identical to their counter parts in colorToGreyscaleConversion. The only significant difference is that we are assuming square matrices for matrixMulKernel so we replace both width and height with Width.

The thread-to-data mapping effectively divides P into tiles, one of which is shown as a large square in Figure 4.2. Each block is responsible for calculating one of these tiles.


```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

Figure 4.3: *A simple matrix multiplication kernel using one thread to compute one P element*

We now turn our attention to the work done by each thread. Recall that $P_{Row, Col}$ is the inner product of the Row^{th} row of M and the Col^{th} column of N. In Figure 4.3, we use a for-loop to perform this inner product operation. Before we enter the loop, we initialize a local variable Pvalue to 0. Each iteration of the loop accesses an element from the Row^{th} row of M and one from the Col^{th} column of N, multiplies the two elements together, and accumulates the product into Pvalue.

Let's first focus on accessing the M element within the for-loop. Recall that M is linearized into an equivalent 1D array where the rows of M are placed one after another in the memory space, starting with the 0^{th} row. Therefore, the beginning element of the 1^{st} row is M[1*Width] because we need to account for all elements of the 0^{th} row. In general, the beginning element of the Row^{th} row is M[Row*Width]. Since all elements of a row is placed in consecutive locations, the k^{th} element of the Row^{th} row is at M[Row*Width+k]. This is what we used in Figure 4.3.

We now turn our attention to N. As shown in Figure 4.3, the beginning element of the Col^{th} column is the Col^{th} element of the 0^{th} row, which is N[Col]. Accessing each additional element in Col^{th} column requires skipping over entire rows. This is because the next element of the same column is actually the same element in the next row. Therefore, the k^{th} element of the Col^{th} column is N[k*Width+Col].

After the execution exits the for-loop, all threads have their P element value in the Pvalue variables. Each thread then uses the one-dimensional equivalent index expression

Row*Width+Col to write its P element. Again, this index pattern is similar to that used in the colorToGreyscaleConversion kernel.

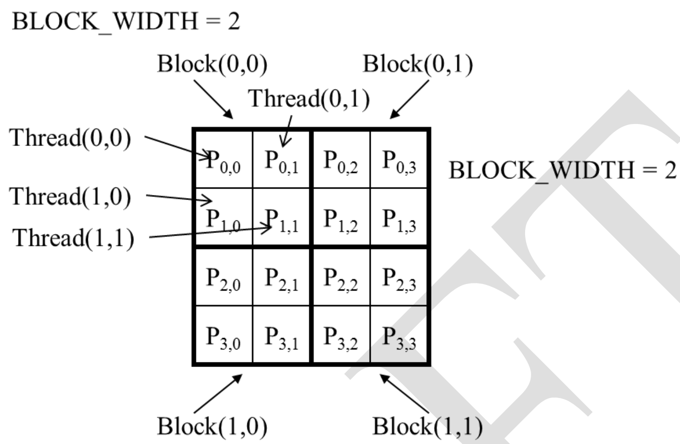

Figure 4.4: *A small execution example of matrixMulKernel*

We now use a small example to illustrate the execution of the matrix multiplication kernel. Figure 4.4 shows a 4×4 P with BLOCK_WIDTH = 2. The small sizes allow us to fit the entire example in one picture. The P matrix is now divided into four tiles and each block calculates one tile. We do so by creating blocks that are 2x2 arrays of threads, with each thread calculating one P element. In the example, thread(0,0) of block(0,0) calculates $P_{0,0}$ whereas thread(0,0) of block(1,0) calculates $P_{2,0}$.

Row and Col in the matrixMulKernel identify the P element to be calculated by a thread. Row also identifies the row of M and Col identifies the column of N as input values for the thread. Figure 4.5 illustrates the multiplication actions in each thread block. For the small matrix multiplication example, threads in block (0,0) produce four dot products. The Row and Col variables of thread(1,0) in block(0,0) are 0*0+1=1 and 0*0+0=0. It maps to $P_{1,0}$ and calculates the dot product of row 1 of M and column 0 of N.

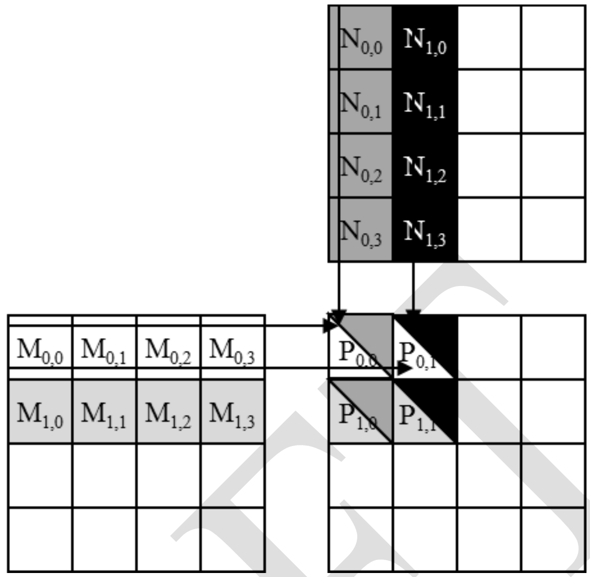

Figure 4.5: *Matrix multiplication actions of one thread block.*

Let's walk through the execution of the for-loop of Figure 4.3 for thread(0,0) in block(0,0). During the 0th iteration (k=0), Row*Width+k = 0*4+0 = 0 and k*Width+Col = 0*4+0 = 0. Therefore the we are accessing M[0] and N[0], which according to Figure 3.3 are the 1D equivalent of $M_{0,0}$ and $N_{0,0}$. Note that these are indeed the 0th elements of row 0 of M and column 0 of N. During the 1st iteration (k=1), Row*Width+k = 0*4+1 = 1 and k*Width+Col = 1*4+0 = 4. We are accessing M[1] and N[4], which according to Figure 3.3 are the 1D equivalent of $M_{0,1}$ and $N_{1,0}$. These are the 1st elements of row 0 of M and column 0 of N.

During the 2nd iteration (k=2), Row*Width+k = 0*4+2 = 2 and k*Width+Col = 8, which results in M[2] and N[8]. Therefore, the elements accessed are the 1D equivalent of $M_{0,2}$ and d_$N_{2,0}$. Finally, during the 3rd iteration (k=3), Row*Width+k = 0*4+3 and k*Width+Col = 12, which results in M[3] and N[12], the 1D equivalent of $M_{0,3}$ and $N_{3,0}$. We now have verified that the for-loop performs inner product between the 0th row of M and the 0th column of N. After the loop, the thread writes P[Row*Width+Col], which is P[0]. This is the 1D equivalent of $P_{0,0}$ so thread(0,0) in block(0,0) successfully calculated the inner product between the 0th row of M and the 0th column of N and deposited the result in $P_{0,0}$.

We will leave it as an exercise for the reader to hand-execute and verify the for-loop for other threads in block(0,0) or in other blocks.

Note that matrixMulKernel can handle matrices of up to 16×65,535 elements in each dimension. In the situation where matrices larger than this limit are to be multiplied, one can divide up the P matrix into sub-matrices whose size can be covered by a grid. We can then use the host code to iteratively launch kernels and complete the P matrix. Alternatively, we can change the kernel code so that each thread calculates more P elements.

We can estimate the effect of memory access efficiency by calculating the expected performance level of the matrix multiplication kernel code in Figure 4.3. The dominating part of the kernel in terms of execution time is the for-loop that performs inner product calculation:

```
for (int k = 0; k < Width; ++k)  Pvalue += M[Row*Width+k] * N[k*Width+Col];
```

In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition. One global memory access fetches an M element and the other fetches an N element. One floating-point operation multiplies the M and N elements fetched and the other accumulates the product into Pvalue. Thus, the compute-to-global-memory-access ratio of the loop is 1.0. From our discussion in Chapter 3, this ratio will likely result in less than 3% utilization of the peak execution speed of the modern GPUs. We need to increase it by at least an order of magnitude in order to achieve good utilization of the computation throughput of modern devices. In the next section, we will show that we can use the special memory types in CUDA devices to accomplish this goal.

4.3. CUDA Memory Types

A CUDA device contains several types of memory that can help programmers to improve compute-to-global-memory-access ratio and thus achieve high execution speed. Figure 4.6 shows these CUDA device memories. At the bottom of the picture, we see global memory and constant memory. These types of memory can be written (W) and read (R) by the host by calling API functions[1]. We have already introduced global memory in Chapter 2. The global memory can be written and read by the device. The constant memory supports short-latency, high bandwidth **read-only access** by the device.

Registers and shared memory in Figure 4.6 are on-chip memories. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual threads; each thread can only access its own registers. A kernel function typically uses registers to hold frequently accessed variables that are private to each thread. Shared memory locations are allocated to thread blocks; all threads

[1] See CUDA Programming Guide for zero-copy access to the global memory.

in a block can access shared-memory variables allocated to the block. Shared memory is an efficient means for threads to cooperate by sharing their input data and intermediate results. By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

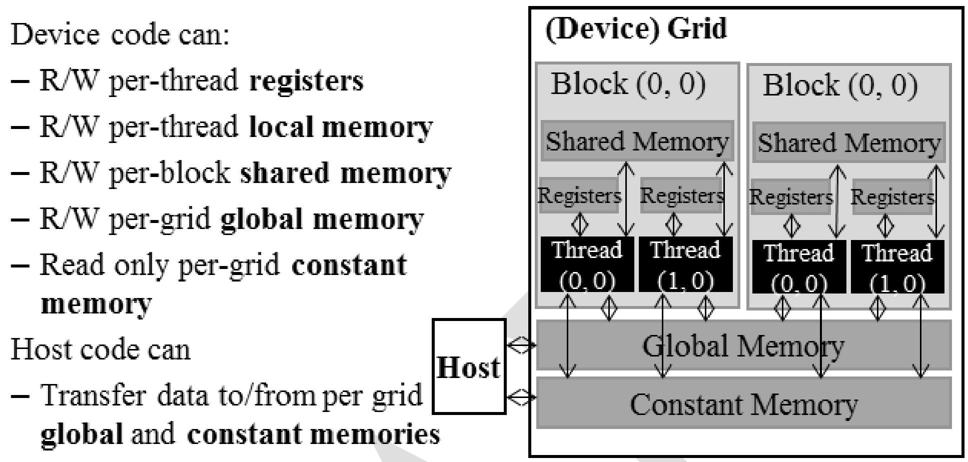

Figure 4.6: *Overview of the CUDA device memory model*

In order to fully appreciate the difference between registers, shared memory and global memory, we need to go into a little more details of how these different memory types are realized and used in modern processors. Virtually all modern processors find their root in the model proposed by John von Neumann in 1945, which is shown in Figure 4.7. The CUDA devices are no exception. The Global Memory in a CUDA device maps to the Memory box in Figure 4.7. The processor box corresponds to the processor chip boundary that we typically see today. The Global Memory is off the processor chip and is implemented with DRAM technology, which implies long access latencies and relatively low access bandwidth. The Registers correspond to the “Register File” of the von Neumann model. The Register File is on the processor chip, which implies very short access latency and drastically higher access bandwidth when compared to the global memory. In a typical device, the aggregated access bandwidth of the register files is at least two orders of magnitude higher than that of the global memory. Furthermore, whenever a variable is stored in a register, its accesses no longer consume off-chip global memory bandwidth. This will be reflected as an increased compute-to-global-memory-access ratio.

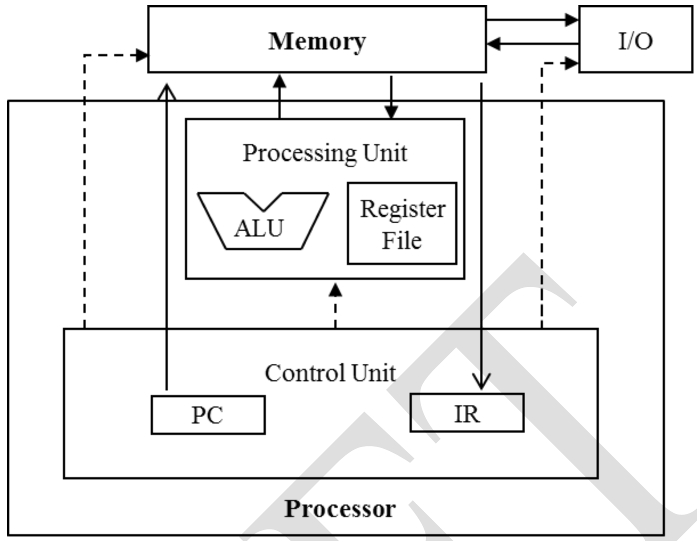

Figure 4.7*: Memory vs. Registers in a modern computer based on the von Neumann model*

A subtler point is that each access to registers involves fewer instructions than an access to the global memory. Arithmetic instructions in most modern processors have "built-in" register operands. For example, a floating-point addition instruction might be of the form:

```
fadd    r1, r2, r3
```

where r2 and r3 are the register numbers that specifies the location in the register file where the input operand values can be found. The location for storing the floating-point addition result value is specified by r1. Therefore, when an operand of an arithmetic instruction is in

The von Neumann Model

In his seminal 1945 report, John von Neumann described a model for building electronic computers, which is based on the design of the pioneering EDVAC computer. This model, now commonly referred to as the von Neumann Model, has been the foundational blueprint for virtually all modern computers.

The von Neumann Model is illustrated in the Figure 4.7. The computer has an I/O (input/output) that allows both programs and data to be provided to and generated from the system. In order to execute a program, the computer first inputs the program and its data into the Memory.

The program consists of a collection of instructions. The Control Unit maintains a Program Counter (PC), which contains the memory address of the next instruction to be executed. In each "instruction cycle," the Control Unit uses the PC to fetch an instruction into the Instruction Register (IR). The instruction bits are then used to determine the action to be taken by all components of the computer. This is the reason why the model is also called the "stored program" model, which means that a user can change the behavior of a computer by storing a different program into its memory.

a register, there is no additional instruction required to make the operand value available to the arithmetic and logic unit (ALU), where the arithmetic calculation is done.

Meanwhile, if an operand value is in the global memory, the processor needs to perform a memory load operation in order to make the operand value available to the ALU. For example, if the first operand of a floating-point addition instruction is in the global memory, the instructions involved will likely be

```
load r2, r4, offset
fadd r1, r2, r3
```

where the load instruction adds an offset value to the contents of r4 to form an address for the operand value. It then accesses the global memory and places the value into register r2. Once the operand value is in r2, the `fadd` instruction performs the floating-point addition using the values in r2 and r3 and places the result into r1. Since the processor can only fetch and execute a limited number of instructions per clock cycle, the version with an additional load will likely take more time to process than the one without. This is another reason why placing the operands in registers can improve execution speed.

Finally, there is another subtle reason why placing an operand value in registers is preferable. In modern computers, the energy consumed for accessing a value from the register file is at least an order of magnitude lower that for accessing a value from the global memory. We will look at more details of the speed and energy difference in accessing these two hardware structures in modern computers soon. However, as we will soon learn, the number of registers available to each thread (see "Processing Units and Threads" sidebar) is quite limited in today's GPUs. We need to be careful not oversubscribe to this limited resource.

Figure 4.8 shows the shared memory and registers in a CUDA device. Although both are on-chip memories, they differ significantly in functionality and cost of access. Shared memory is designed as part of the memory space that resides on the processor chip. When the processor accesses data that reside in the shared memory, it needs to perform a memory load operation, just like accessing data in the global memory. However, because shared memory resides on-chip, it can be accessed with much lower latency and much higher throughput than the global memory. Because of the need to perform a load operation, share memory has longer latency and lower bandwidth than registers. In computer architecture terminology, the shared memory is a form of *scratchpad memory*.

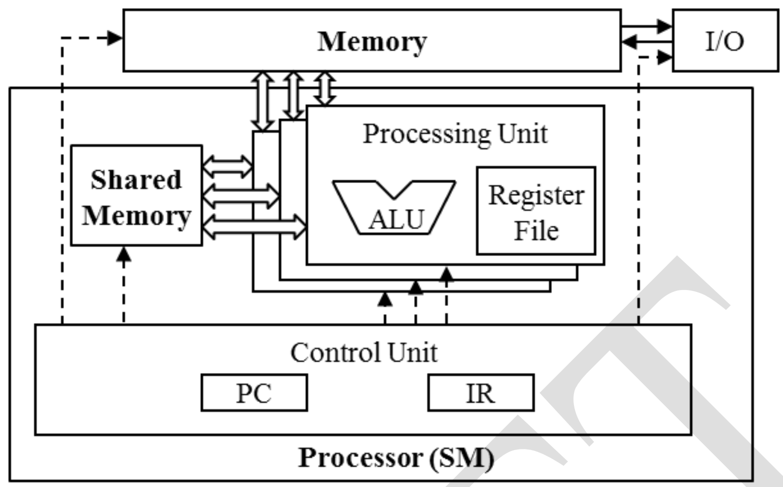

Figure 4.8: *Shared memory vs. registers in a CUDA device SM*

One important difference between the share memory and registers in CUDA is that variables that reside in the shared memory are accessible by all threads in a block. This is in contrast to register data, which is private to a thread. That is, shared memory is designed to support

Processing Units and Threads

Now that we have introduced the von Neumann model, we are ready to discuss how threads are implemented. A thread in modern computers is the state of executing a program on a von Neumann Processor. Recall that a thread consists of the code of a program, the particular point in the code that is being executed, and value of its variables and data structures.

In a computer based on the von Neumann model, the code of the program is stored in the memory. The PC keeps track of the particular point of the program that is being executed. The IR holds the instruction that is fetched from the point execution. The register and memory hold the values of the variables and data structures.

Modern processors are designed to allow context-switching, where multiple threads can time-share a processor by taking turns to make progress. By carefully saving and restoring the PC value and the contents of registers and memory, we can suspend the execution of a thread and correctly resume the execution of the thread later.

Some processors provide multiple processing units, which allow multiple threads to make simultaneous progress. Figure 4.8 shows a Single-Instruction, Multiple-Data (SIMD) design style where multiple processing units share a PC and IR. Under this design, all threads make simultaneous progress by executing the same instruction in the program.

efficient, high-bandwidth sharing of data among threads in a block. As shown in Figure 4.8, a CUDA device SM typically employs multiple processing units, to allow multiple threads to make simultaneous progress (see "Processing Units and Threads" sidebar). Threads in a block can be spread across these processing units. Therefore, the hardware implementations of the shared memory in these CUDA devices are typically designed to allow multiple processing units to simultaneously access its contents to support efficient data sharing among threads in a block. We will be learning several important types of parallel algorithms that can greatly benefit from such efficient data sharing among threads.

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	kernel
Automatic array variables	local	thread	kernel
`__device__ __shared__ int SharedVar;`	shared	block	kernel
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstVar;`	constant	grid	application

Table 4.1: *CUDA Variable Type Qualifiers*

It should be clear by now that registers, shared memory and global memory all have different functionalities, latencies, and bandwidth. It is therefore important to understand how to declare a variable so that it will reside in the intended type of memory. Table 4.1 presents the CUDA syntax for declaring program variables into the various memory types. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: a single thread only, all threads of a block, or all threads of all grids. If a variable's scope is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable. For example, if a kernel declares a variable whose scope is a thread and it is launched with one million threads, one million versions of the variable will be created so that each thread initializes and uses its own version of the variable.

Lifetime tells the portion of the program's execution duration when the variable is available for use: either within a kernel's execution or throughout the entire application. If a variable's lifetime is within a kernel execution, it must be declared within the kernel function body and will be available for use only *by the kernel's code. If the kernel is invoked several times, the value of the variable is not* maintained across these invocations. Each invocation must initialize the variable in order to use them. On the other hand, if a variable's lifetime is throughout the entire application, it must be declared outside of any function body. The contents of these variables are maintained throughout the execution of the application and available to all kernels.

We refer to variables that are not arrays or matrices as *scalar* variables. As shown in Table 4.1, all automatic scalar variables declared in kernel and device functions are placed into registers. The scopes of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all its automatic variables also cease to exist. In Figure 4.1, variables `blurRow`, `blurCol`, `curRow`, `curCol`, `pixels` and `pixVal` are all automatic variables and fall into this category. Note that accessing these variables is extremely fast and parallel but one must be careful not to exceed the limited capacity of the register storage in the hardware implementations. Using a large number of registers can negatively impact the number of active threads assigned to each SM. We will address this point in Chapter 5.

Automatic array variables are not stored in registers[2]. Instead, they are stored into the global memory and may incur long access delays and potential access congestions. The scope of these arrays is, like automatic scalar variables, limited to individual threads. That is, a private version of each automatic array is created for and used by every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist. From our experience, one seldom needs to use automatic array variables in kernel functions and device functions.

If a variable declaration is preceded by the "`__shared__`" (each "__" consists of two "_" characters) keyword, it declares a shared variable in CUDA. One can also add an optional "`__device__`" in front of "`__shared__`" in the declaration to achieve the same effect. Such declaration typically resides within a kernel function or a device function. Shared variables reside in shared memory. The scope of a shared variable is within a thread block, that is, all threads in a block see the same version of a shared variable. A private version of the shared variable is created for and used by each thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel. When a kernel terminates its execution, the contents of its shared variables cease to exist. As we discussed earlier, shared variables are an efficient means for threads within a block to collaborate with each other. Accessing shared variables from the shared memory is extremely fast and highly parallel. CUDA programmers often use shared variables to hold the portion of global memory data that are heavily used in an execution phase of kernel. One may need to adjust the algorithms used in order to create execution phases that heavily focus on small portions of the global memory data, as we will demonstrate with matrix multiplication in Section 4.4.

If a variable declaration is preceded by keywords "`__constant__`" (each "__" consists of two "_" characters) it declares a constant variable in CUDA. One can also add an optional

[2] There are some exceptions to this rule. The compiler may decide to store an automatic array into registers if all accesses are done with constant index values.

"__device__" in front of "__constant__" to achieve the same effect. Declaration of constant variables must be outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution. Constant variables are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at 65,536 bytes. One may need to break up the input data volume to fit within this limitation, as we will illustrate in Chapter 7.

A variable whose declaration is preceded only by the keyword "__device__" (each "__'' consists of two "_'' characters), is a global variable and will be placed in the global memory. Accesses to a global variable are slow. Latency and throughput of accessing global variables have been improved with caches in more recent devices. One important advantage of global variables is that they are visible to all threads of all kernels. Their contents also persist through the entire execution. Thus, global variables can be used as a means for threads to collaborate across blocks. One must, however, be aware of the fact that there is currently no easy way to synchronize between threads from different thread blocks or to ensure data consistency across threads when accessing global memory other than terminating the current kernel execution[3]. Therefore, global variables are often used to pass information from one kernel invocation to another kernel invocation.

In CUDA, pointers are used to point to data objects in the global memory. There are two typical ways in which pointer usage arises in kernel and device functions. First, if an object is allocated by a host function, the pointer to the object is initialized by cudaMalloc and can be passed to the kernel function as a parameter. For example, the parameters M, N, and P in Figure 4.1 are such pointers. The second type of usage is to assign the address of a variable declared in the global memory to a pointer variable. For example, the statement {float* ptr = &GlobalVar;} in a kernel function assigns the address of GlobalVar into an automatic pointer variable ptr. The reader should refer to the CUDA Programming Guide for using pointers in other memory types.

4.4. Tiling for Reduced Memory Traffic

We have an intrinsic tradeoff in the use of device memories in CUDA: the global memory is large but slow whereas the shared memory is small but fast. A common strategy is to partition the data into subsets called *tiles* so that each tile fits into the shared memory. The term tile draws on the analogy that a large wall (i.e., the global memory data) can be covered

[3] Note that one can use CUDA memory fencing to ensure data coherence between thread blocks if the number of thread blocks is smaller than the number of SMs in the CUDA device. See the CUDA programming guide for more details.

by tiles (i.e., subsets that each can fit into the shared memory). An important criterion is that the kernel computation on these tiles can be done independently of each other. Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example. Figure 4.5 shows a small example of matrix multiplication. It corresponds to the kernel function in Figure 4.3. We replicate the example in Figure 4.9 for convenient reference by the reader. For brevity, we abbreviate P[y*Width+x], M[y*Width+x], and N[y*Width+x] into $P_{y,x}$, $M_{y,x}$, and $N_{y,x}$. This example assumes that we use four 2×2 blocks to compute the P matrix. Figure 4.5 highlights the computation done by the four threads of block(0,0). These four threads compute $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$. The accesses to the M and N elements by thread(0,0) and thread(0,1) of block(0,0) are highlighted with black arrows. For example, thread(0,0) reads $M_{0,0}$ and $N_{0,0}$, followed by $M_{0,1}$ and $N_{1,0}$ followed by $M_{0,2}$ and $N_{2,0}$, followed by $M_{0,3}$ and $N_{3,0}$.

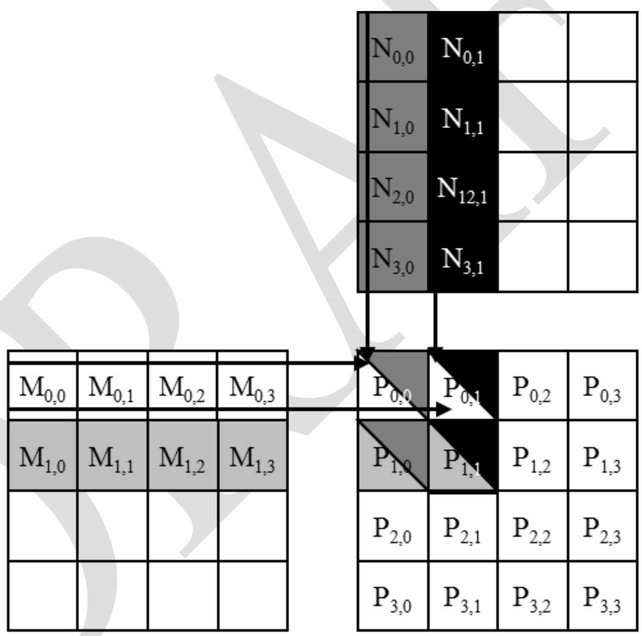

Figure 4.9 *A small example of matrix multiplication. For brevity, We show M[y*Width+x], N[y*Width+x], P[y*Width+x] as* $M_{y,x}$, $N_{y,x}$ $P_{y,x}$.

Figure 4.10 shows the global memory accesses done by all threads in $block_{0,0}$. The threads are listed in the vertical direction, with time of access increasing to the right in the horizontal direction. Note that each thread accesses four elements of M and four elements of N during its execution. Among the four threads highlighted, there is a significant overlap in the M and

N elements they access. For example, $thread_{0,0}$ and $thread_{0,1}$ both access $M_{0,0}$ as well as the rest of row 0 of M. Similarly, $thread_{0,1}$ and $thread_{1,1}$ both access $N_{0,1}$ as well as the rest of column 1 of N.

The kernel in Figure 4.3 is written so that both $thread_{0,0}$ and $thread_{0,1}$ access row 0 elements of M from the global memory. If we can somehow manage to have $thread_{0,0}$ and $thread_{1,0}$ to collaborate so that these M elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In fact, we can see that every M and N element is accessed exactly twice during the execution of $block_{0,0}$. Therefore, if we can have all four threads to collaborate in their accesses to global memory, we can reduce the traffic to the global memory by half.

Readers should verify that the potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used. With Width×Width blocks, the potential reduction of global memory traffic would be Width. That is, if we use 16×16 blocks, one can potentially reduce the global memory traffic to 1/16 through collaboration between threads.

Access order →

$thread_{0,0}$	$\boxed{M_{0,0}} * N_{0,0}$	$M_{0,1} * \boxed{N_{1,0}}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$\boxed{M_{0,0}} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * \boxed{N_{1,0}}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

Figure 4.10*: Global memory accesses performed by threads in* $block_{0,0}$

Traffic congestion obviously does not only arise in computing. Most of us have experienced traffic congestion in highway systems, as illustrated in Figure 4.11. The root cause of highway traffic congestion is that there are too many cars all squeezing through a road that is designed for a much smaller number of vehicles. When congestion occurs, the travel time for each vehicle is greatly increased. Commute time to work can easily double or triple during traffic congestion.

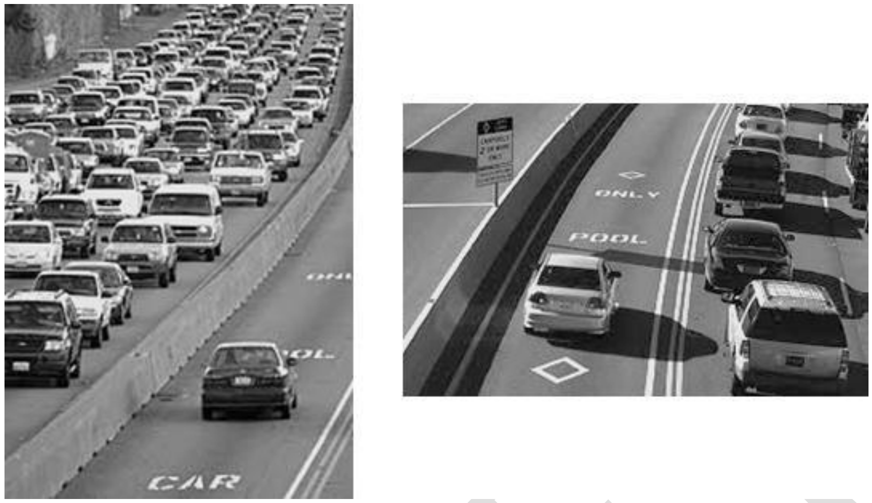

***Figure 4.11**: Reducing traffic congestion in highway systems*

Most solutions for reduced traffic congestion involve reduction of cars on the road. Assuming that the number of commuters is constant, people need to share rides in order to reduce the number of cars on the road. A common way to share rides in the U.S. is carpooling, where a group of commuters take turns to drive the group to work in one vehicle. The government usually need to have policies to encourages carpooling. In some countries, the government simply disallows certain classes of cars to be on the road on a daily basis. For example, cars with odd license plates may not be allowed on the road on Monday, Wednesday, or Friday. This encourages people whose cars are allowed on different days to form a carpool group. There are also countries where the government makes gasoline so expensive that people form carpools to save money. In other countries, the government may provide incentives for behavior that reduces the number of cars on the road. In the U.S., some lanes of congested highways are designated as carpool lanes; only cars with more than 2 or 3 people are allowed to use these lanes. All these measures for encouraging carpooling are designed to overcome the fact that carpooling requires extra effort, as we show in Figure 4.12.

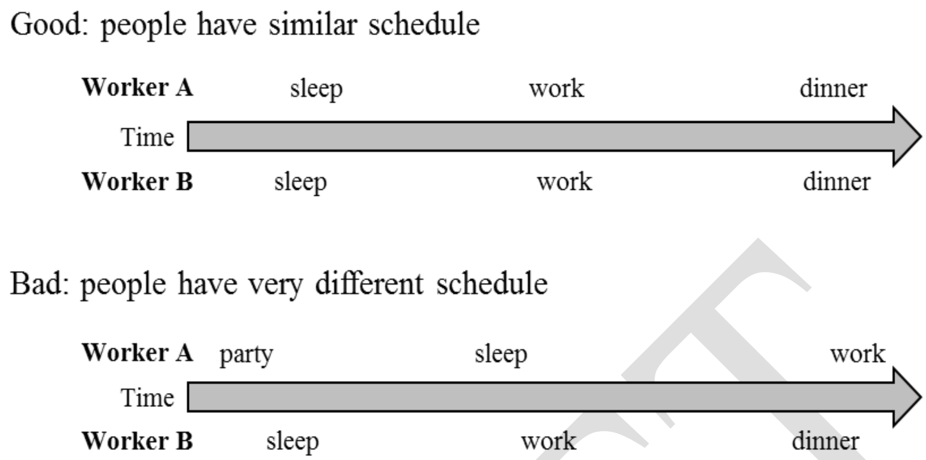

Figure 4.12: *Carpooling requires synchronization among people*

Carpooling requires workers who wish to carpool to compromise and agree on a common commute schedule. The top half of Figure 4.12 shows a good schedule pattern for carpooling. Time goes from left to right. Worker A and Worker B have similar schedule for sleep, work, and dinner. This allows these two workers to easily go to work and return home in one car. Their similar schedules allow them to more easily agree on a common departure time and return time. This is however, not the case of the schedules shown in the bottom half of Figure 4.12. Worker A and Worker B have very different habit in this case. Worker A parties until sunrise, sleeps during the day, and goes to work in the evening. Worker B sleeps at night, goes to work in the morning, and returns home for dinner at 6pm. The schedules are so wildly different that there is no way these two workers can coordinate a common time to drive to work and return home in one car. In order for these workers to form a carpool, they need to negotiate a common schedule similar what is shown in the top half of Figure 4.12.

Tiled algorithms are very similar to carpooling arrangements. We can think of threads accessing data values as commuters and DRAM access requests as vehicles. When the rate of DRAM requests exceeds the provisioned access bandwidth of the DRAM system, traffic congestion arises and the arithmetic units become idle. If multiple threads access data from the same DRAM location, they can potentially form a “carpool” and combine their accesses into one DRAM request. This, however, requires the threads to have similar execution schedule so that their data accesses can be combined into one. This is shown in Figure 4.13, where the cells in the middle of the picture depicts DRAM locations. An arrow going from a DRAM location to a thread represent an access by the thread to that location at the time marked by the head of the arrow. Note that the time goes from left to right. The top portion shows two threads that access the same data elements with similar timing. The bottom half shows two threads that access their common data in very different times. That is, the accesses by Thread 2 lag significantly behind their corresponding accesses by Thread 1. The reason why the bottom is a bad arrangement is that data elements brought back from the DRAM need to be kept in the on-chip memory for a long time, waiting for Thread 2 to consume them. This will likely require a large number of data elements to be kept around, resulting in excessive on-chip memory requirements.

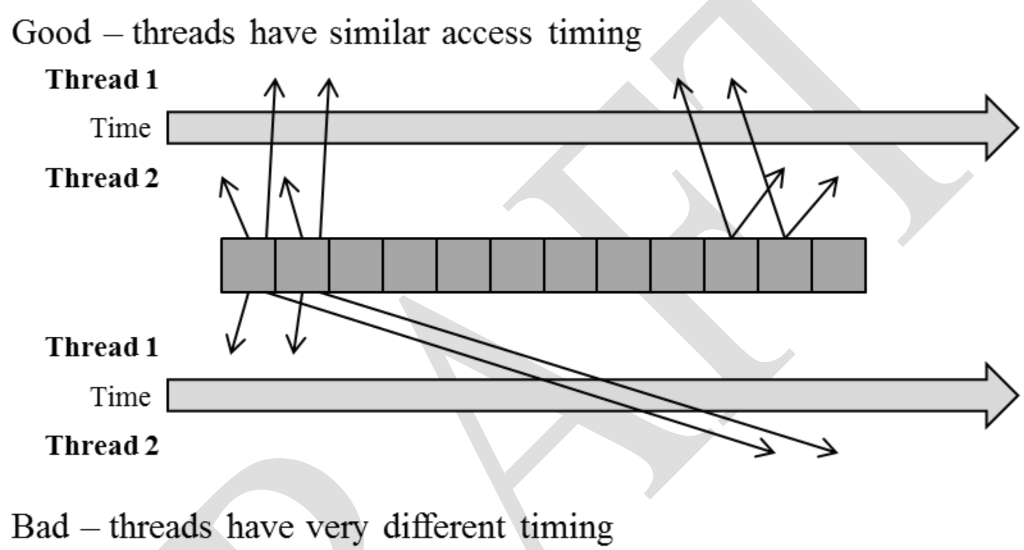

Figure 4.13 *Tiled Algorithms require synchronization among threads*

In the context of parallel computing, tiling is a program transformation technique that localizes the memory locations accessed among threads and the timing of their accesses. It breaks long access sequences of each thread into phases and use barrier synchronization to keep the timing of accesses to each section by the threads close to each other. It controls the amount of on-chip memory required by localizing the accesses both in time and in space. In terms of our carpool analogy, we keep the threads that form the “carpool” group to follow approximately the same execution timing.

We now present a tiled matrix-multiplication algorithm. The basic idea is to have the threads to collaboratively load subsets of the M and N elements into the shared memory before they individually use these elements in their dot product calculation. Keep in mind that the size of the shared memory is quite small and one must be careful not to exceed the capacity of the shared memory when loading these M and N elements into the shared memory. This can be accomplished by dividing the M and N matrices into smaller tiles. The size of these tiles is chosen so that they can fit into the shared memory. In the simplest form, the tile dimensions equal those of the block, as illustrated in Figure 4.11.

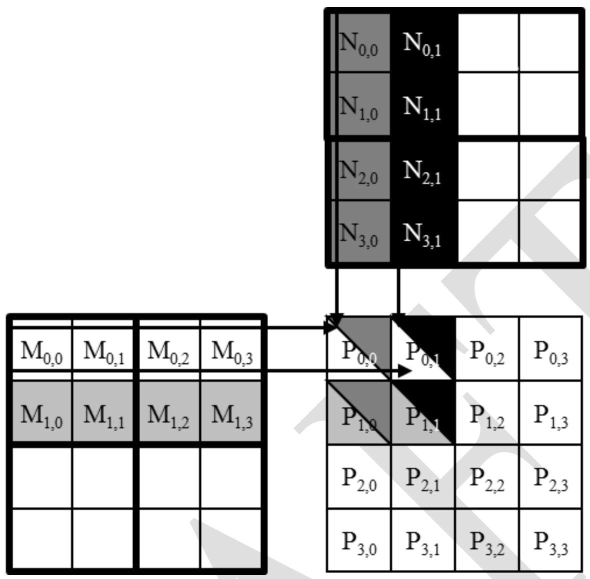

Figure 4.14 *Tiling M and N to utilize shared memory*

In Figure 4.14, we divide M and N into 2×2 tiles, as delineated by the thick lines. The dot product calculations performed by each thread are now divided into phases. In each phase, all threads in a block collaborate to load a tile of M and a tile of N into the shared memory. This can be done by having every thread in a block to load one M element and one N element into the shared memory, as illustrated in Figure 4.15. Each row of Figure 4.15 shows the execution activities of a thread. Note that time progresses from left to right. We only need to show the activities of threads in $block_{0,0}$; the other blocks all have the same behavior. The shared memory array for the M elements is called Mds. The shared memory array for the N elements is called Nds. At the beginning of Phase 1, the four threads of $block_{0,0}$ collaboratively load a tile of M into shared memory: $thread_{0,0}$ loads $M_{0,0}$ into $Mds_{0,0}$, $thread_{0,1}$ loads $M_{0,1}$ into $Mds_{0,1}$, $thread_{1,0}$ loads $M_{1,0}$ into $Mds_{1,0}$, and $thread_{1,1}$ loads $M_{1,1}$ into $Mds_{1,1}$. These loads are shown in the second column of Figure 4.15. A tile of N is also loaded in a similar manner, shown in the third column of Figure 4.15.

After the two tiles of M and N are loaded into the shared memory, these elements are used in the calculation of the dot product. Note that each value in the shared memory is used twice. For example, the $M_{1,1}$ value, loaded by $thread_{1,1}$ into $Mds_{1,1}$, is used twice, once by $thread_{1,0}$ and once by $thread_{1,1}$. By loading each global memory value into shared memory so that it can be used multiple times, we reduce the number of accesses to the global memory. In this case, we reduce the number of accesses to the global memory by half. The reader should verify that the reduction is by a factor of N if the tiles are N×N elements.

	Phase 1			Phase 2		
$\text{thread}_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $\text{Mds}_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $\text{Nds}_{0,0}$	$\text{PValue}_{0,0}$ += $\text{Mds}_{0,0}$*$\text{Nds}_{0,0}$ + $\text{Mds}_{0,1}$*$\text{Nds}_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $\text{Mds}_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $\text{Nds}_{0,0}$	$\text{PValue}_{0,0}$ += $\text{Mds}_{0,0}$*$\text{Nds}_{0,0}$ + $\text{Mds}_{0,1}$*$\text{Nds}_{1,0}$
$\text{thread}_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $\text{Mds}_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $\text{Nds}_{1,0}$	$\text{PValue}_{0,1}$ += $\text{Mds}_{0,0}$*$\text{Nds}_{0,1}$ + $\text{Mds}_{0,1}$*$\text{Nds}_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $\text{Mds}_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $\text{Nds}_{0,1}$	$\text{PValue}_{0,1}$ += $\text{Mds}_{0,0}$*$\text{Nds}_{0,1}$ + $\text{Mds}_{0,1}$*$\text{Nds}_{1,1}$
$\text{thread}_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $\text{Mds}_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $\text{Nds}_{1,0}$	$\text{PValue}_{1,0}$ += $\text{Mds}_{1,0}$*$\text{Nds}_{0,0}$ + $\text{Mds}_{1,1}$*$\text{Nds}_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $\text{Mds}_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $\text{Nds}_{1,0}$	$\text{PValue}_{1,0}$ += $\text{Mds}_{1,0}$*$\text{Nds}_{0,0}$ + $\text{Mds}_{1,1}$*$\text{Nds}_{1,0}$
$\text{thread}_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $\text{Mds}_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $\text{Nds}_{1,1}$	$\text{PValue}_{1,1}$ += $\text{Mds}_{1,0}$*$\text{Nds}_{0,1}$ + $\text{Mds}_{1,1}$*$\text{Nds}_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $\text{Mds}_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $\text{Nds}_{1,1}$	$\text{PValue}_{1,1}$ += $\text{Mds}_{1,0}$*$\text{Nds}_{0,1}$ + $\text{Mds}_{1,1}$*$\text{Nds}_{1,1}$

time →

Figure 4.15: *Execution phases of a tiled matrix multiplication*

Note that the calculation of each dot product in Figure 4.3 is now performed in two phases, shown as Phase 1 and Phase 2 in Figure 4.15. In each phase, products of two pairs of the input matrix elements are accumulated into the Pvalue variable. Note that Pvalue is an automatic variable so a private version is generated for each thread. We added subscripts just to clarify that these are different instances of the Pvalue variable created for each thread. The first phase calculation is shown in the fourth column of Figure 4.15; the second phase the seventh column. In general, if an input matrix is of dimension Width and the tile size is TILE_WIDTH, the dot product would be performed in Width/TILE_WIDTH phases. The creation of these phases is key to the reduction of accesses to the global memory. With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy their overlapping input needs in the phase.

Note also that Mds and Nds are re-used to hold the input values. In each phase, the same Mds and Nds are used to hold the subset of M and N elements used in the phase. This allows a much smaller shared memory to serve most of the accesses to global memory. This is due to the fact that each phase focuses on a small subset of the input matrix elements. Such focused access behavior is called locality. When an algorithm exhibits locality, there is an opportunity to use small, high-speed memories to serve most of the accesses and remove these accesses from the global memory. Locality is as important for achieving high-performance in multi-core CPUs as in many-thread GPUs We will return to the concept of locality in Chapter 5.

4.5. A Tiled Matrix Multiplication Kernel

We are now ready to present a tiled matrix multiplication kernel that uses shared memory to reduce traffic to the global memory. The kernel shown in Figure 4.16 implements the phases illustrated in Figure 4.15. In Figure 4.16, Line 1 and Line 2 declare Mds and Nds as shared memory variables. Recall that the scope of shared memory variables is a block. Thus, one pair of Mds and Nds will be created for each block and all threads of a block have access to the same Mds and Nds. This is important since all threads in a block must have access to the M and N elements loaded into Mds and Nds by their peers so that they can use these values to satisfy their input needs.

```
#define TILE_WIDTH 16
```

```
    __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 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 4.16*: A tiled Matrix Multiplication Kernel using shared memory*

Lines 3 and 4 save the threadIdx and blockIdx values into automatic variables and thus into registers for fast access. Recall that automatic scalar variables are placed into registers. Their scope is in each individual thread. That is, one private version of tx, ty, bx, and by is created by the run-time system for each thread and will reside in registers that are accessible by the thread. They are initialized with the threadIdx and blockIdx values and used many times during the lifetime of thread. Once the thread ends, the values of these variables cease to exist.

Lines 5 and 6 determine the row index and column index of the P element that the thread is to produce. The code assumes that each thread is responsible for calculating one P element. As shown in Line 6, the horizontal (x) position, or the column index of the P element to be produced by a thread can be calculated as bx*TILE_WIDTH+tx. This is because each block covers TILE_WIDTH elements in the horizontal dimension. A thread in block bx would have bx blocks of threads, or (bx*TILE_WIDTH) threads, before it; they cover bx*TILE_WIDTH elements of P. Another tx threads within the same block would cover another tx elements. Thus the thread with bx and tx should be responsible for calculating the P element whose x index is bx*TILE_WIDTH+tx. This horizontal index is saved in the variable Col for the thread and is also illustrated in Figure 4.17.

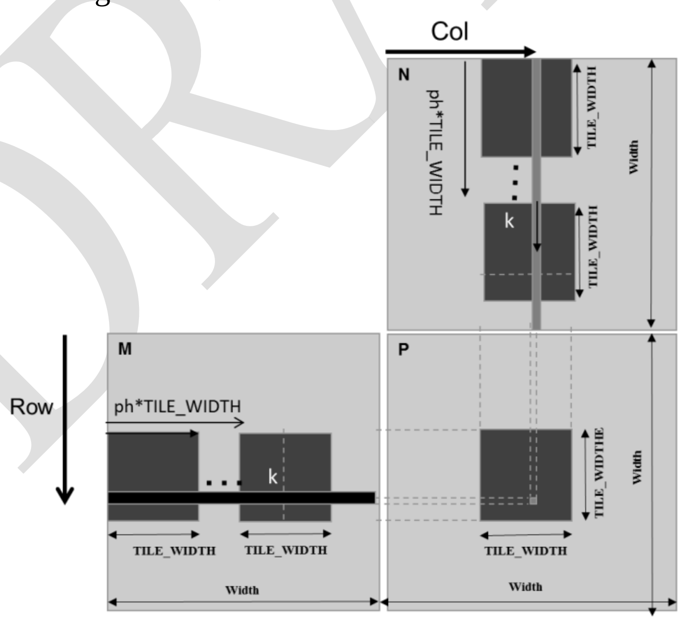

Figure 4.17*: Calculation of the matrix indices in tiled multiplication*

For the example in Figure 4.14, the x index of the P element to be calculated by $thread_{0,1}$ of $block_{1,0}$ is 0*2+1 = 1. Similarly, the y index can be calculated as by*TILE_WIDTH+ty. This vertical index is saved in the variable Row for the thread. Thus, as shown in Figure 4.17, each thread calculates the P element at the Col^{th} column and the Row^{th} row. Going back to the example in Figure 4.14, the y index of the P element to be calculated by $thread_{1,0}$ of $block_{0,1}$ is 1*2+0 = 2. Thus, the P element to be calculated by this thread is $P_{2,1}$.

Line 8 of Figure 4.16 marks the beginning of the loop that iterates through all the phases of calculating the P element. Each iteration of the loop corresponds to one phase of the calculation shown in Figure 4.15. The ph variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of M and one tile of N elements. Therefore, at the beginning of each phase, ph*TILE_WIDTH pairs of M and N elements have been processed by previous phases.

In each phase, Line 9 loads the appropriate M element into the shared memory. Since we already know the row of M and column of N to be processed by the thread, we now turn our focus to the column index of M and row index of N. As shown in Figure 4.17, each block has $TILE\_WIDTH^2$ threads that will collaborate to load $TILE\_WIDTH^2$ M elements into the shared memory. Thus, all we need to do is to assign each thread to load one M element. This is conveniently done using the blockIdx and threadIdx. Note that the beginning column index of the section of M elements to be loaded is ph*TILE_WIDTH. Therefore, an easy approach is to have every thread load an element that is tx (the threadIdx.x value) positions away from that beginning point.

This is precisely what we have in Line 9, where each thread loads M[Row*Width + ph*TILE_WIDTH + tx], where the linearized index is formed with the row index Row and column index ph*TILE_WIDTH + tx. Since the value of Row is a linear function of ty, each of the $TILE\_WIDTH^2$ threads will load a unique M element into the shared memory. Together, these threads will load a dark square subset of M in Figure 4.17. The reader should use the small example in Figure 4.14 and Figure 4.15 to verify that the address calculation works correctly for individual threads.

The barrier __syncthreads() in Line 11 ensures that all threads have finished loading the tiles of M and N into Mds and Nds before any of them can move forward. The loop in Line 12 then performs one phase of the dot product based on these tile elements. The progression of the loop for $thread_{ty,tx}$ is shown in Figure 4.17, with the access direction of the M and N elements along the arrow marked with k, the loop variable in Line 12. Note that these elements will be accessed from Mds and Nds, the shared memory arrays holding these M and N elements. The barrier __syncthreads() in Line 14 ensures that all threads have finished using the M and N elements in the shared memory before any of them move on to the next iteration and load the elements from the next tiles. This way, none of the threads would load the elements too early and corrupt the input values for other threads.

The nested loop from Line 8 to Line 14 illustrates a technique called *strip-mining*, which takes a long-running loop and break it into phases. Each phase consists of an inner loop that executes a number of consecutive iterations of the original loop. The original loop becomes an outer loop whose role is to iteratively invoke the inner loop so that all the iterations of the original loop are executed in their original order. By adding barrier synchronizations before and after the inner loop, we force all threads in the same block to all focus their work on a section of their input data. Strip-mining is an important means to creating the phases needed by tiling in data parallel programs. [4]

After all phases of the dot product are complete, the execution exits the loop of Line 8. All threads write to their P element using the linearized index calculated from Row and Col.

The benefit of the tiled algorithm is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of TILE_WIDTH. If one uses 16×16 tiles, we can reduce the global memory accesses by a factor of 16. This increases the compute-to-global-memory-access ratio from 1 to 16. This improvement allows the memory bandwidth of a CUDA device to support a computation rate close to its peak performance. For example, this improvement allows a device with 150 GB/s global memory bandwidth to approach ((150/4)*16) = 600 GFLOPS!

While the performance improvement of the tiled matrix multiplication kernel is impressive, it does make a few simplifying assumptions. First, the width of the matrices are assumed to be a multiple of the width of thread blocks. This prevents the kernel from correctly processing arbitrary sized matrices. The second assumption is that the matrices are square matrices. This is not always true in practice. In the next section, we will present a kernel with boundary checks that removes these assumptions.

4.6. Boundary Checks

We now extend the tiled matrix multiplication kernel to handle matrices with arbitrary width. The extensions will have to allow the kernel to correctly handle matrices whose width is not

[4] Interested reader should note that strip-mining has long been used in programming CPUs. Strip-mining followed by loop interchange is often used to enable tiling for improved locality in sequential programs. Strip-mining is also the main vehicle for vectorizing compilers to generate vector or SIMD instructions for CPU programs.

a multiple of the tile width. Let's change the small example in Figure 4.14 to 3x3 M, N, and P matrices. The revised example is shown in Figure 4.18. Note that the width of the matrices is 3, which is not a multiple of the tile width (2). Figure 4.18 shows the memory access pattern during phase 1 of $block_{0,0}$. We see that $thread_{0,1}$ and $thread_{1,1}$ will attempt to load M elements that do not exist. Similarly, we see that $thread_{1,0}$ and $thread_{1,1}$ will attempt to access N elements that do not exist.

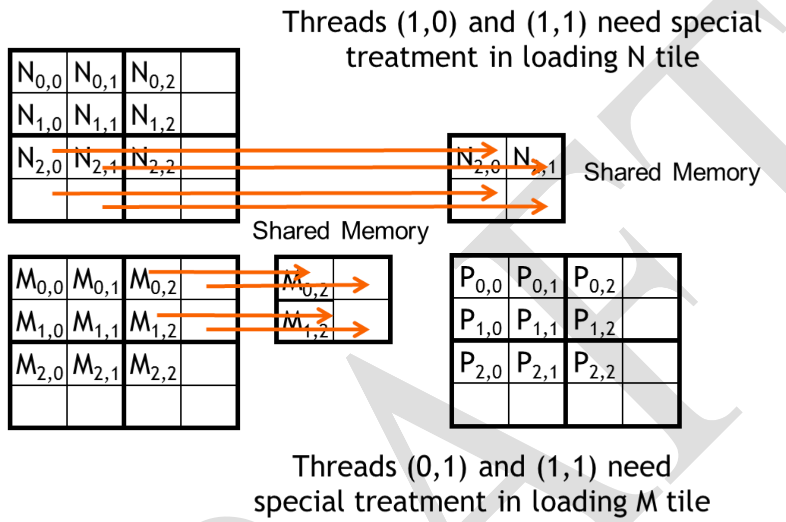

Figure 4.18: *Loading input matrix elements that are close to the edge – phase 1 of $Block_{0,0}$*

Accessing non-existing elements is problematic in two ways. In the case of accessing a non-existing elements past the end of a row (M accesses by $thread_{1,0}$ and $thread_{1,1}$ in Figure 4.18), these accesses will be done to incorrect elements. In our example, the threads will attempt to access $M_{0,3}$ and $M_{1,3}$, which do not exist. So, what will happen to these memory loads? In order to answer this question, we need to go back to the linearized layout of 2D matrices. The element after $M_{0,2}$ in the linearized layout is $M_{1,0}$. Although thead0,1 is attempting to access M0,3, it will end up getting $M_{1,0}$. The use of this value in the subsequent inner product calculation will obviously corrupt the output value.

A similar problem arises when accessing an element past the end of a column (N accesses by $thread_{1,0}$ and $thread_{1,1}$ in Figure 4.18). These accesses are to memory locations outside the allocated area for the array. In some systems, they will return random values from other data structures. Other systems will reject these accesses and cause the program to abort. Either way, the outcome of such accesses is undesirable.

From our discussion so far, it may seem that the problematic accesses only arise in the last phase of execution of the threads. This would suggest that we can deal with it by taking special actions during the last phase of the tiled kernel execution. Unfortunately this is not true. Problematic accesses can arise in all phases. Figure 4.19 shows the memory access pattern of $block_{1,1}$ during phase 0. We see that $thread_{1,0}$ and $thread_{1,1}$ attempt to access non-existing M elements $M_{3,0}$ and $M_{3,1}$ whereas thread0,1and thread1,1 attempt to access $N_{0,3}$ and $N_{1,3}$, which do not exist.

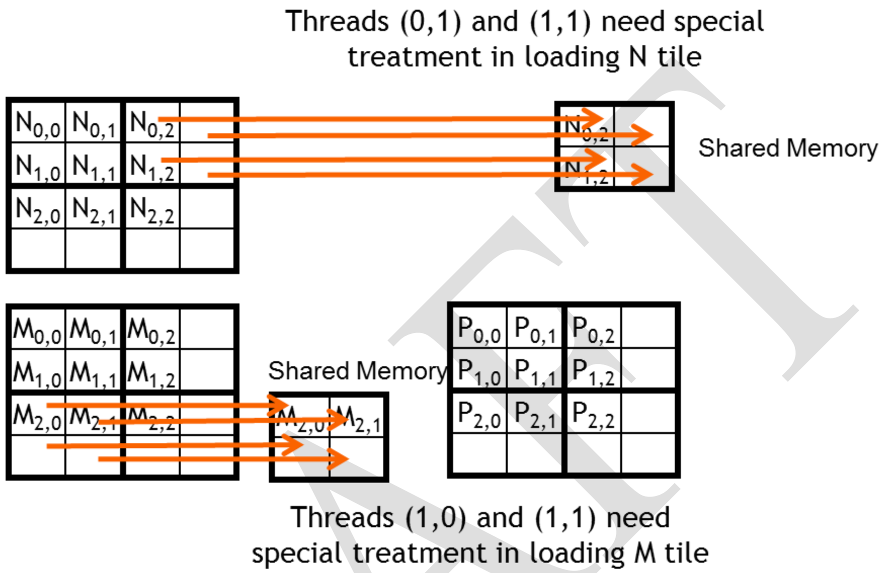

Figure 4.19: *Loading input elements during phase 0 of block1,0.*

Note that these problematic accesses cannot be prevented by excluding the threads that do not calculate valid P elements. For example, $thread_{1,0}$ in $block_{1,1}$ does not calculate any valid P element. However, it needs to load $M_{2,1}$ during phase 0. Further note that some threads that calculate valid P elements will attempt to access M or N elements that do not exist. For example, as we have seen in Figure 4.18, $thread_{0,1}$ of block 0,0 calculates a valid P element $P_{0,1}$. However, it attempts to access a non-existing $M_{0,3}$ during phase 1. These two facts indicate that we will need to use different boundary condition tests for loading M tiles, loading N tiles, and calculating/storing P elements.

Let's start with the boundary test condition for loading input tiles. When a thread is to load an input tile element, it should test whether the input element it is attempting to load is a valid element. This is easily done by examining the y and x indices. For example, at Line 9 in Figure 4.16, the linearized index is a derived from a y index of Row and an x index of ph*TILE_WIDTH + tx. The boundary condition test would be that both of indices are smaller than Width: (Row<Width) && (ph*TILE_WIDTH+tx)<Width. If the condition is true, the thread should go ahead and load the M element. The reader should verify that the condition test for loading the N element is (ph*TILE_WIDTH+ty)<Width && Col<Width.

If the condition is false, the thread should not load the element. The question is what should be placed into the shared memory location. The answer is 0.0, a value that will not cause any harm if it is used in the inner product calculation. If any thread uses this 0.0 value in the calculation of its inner product, there will not be any change in the inner product value.

Finally, a thread should only store its final inner product value if it is responsible for calculating a valid P element. The test for this condition is (Row<Width) && (Col<Width). The kernel code with the additional boundary condition checks is shown in Figure 4.20.

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

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

     for (int k = 0; k < TILE_WIDTH; ++k) {
       Pvalue += Mds[ty][k] * Nds[k][tx];
     }
     __syncthreads();
   }
   if ((Row<Width) && (Col<Width)P[Row*Width + Col] = Pvalue;
```

Figure 4.20*: Tiled matrix multiplication kernel with boundary condition checks.*

With the boundary condition checks, the tile matrix multiplication kernel is just one more step away from being a general matrix multiplication kernel. In general, matrix multiplication is defined for rectangular matrices: a jxk M matrix multiplied with a kxl N matrix results in a jxl P matrix. Our kernel can only handle square matrices so far.

Fortunately, it is quite easy to further extend our kernel into a general matrix multiplication kernel. We need to make a few simple changes. First, the Width argument is replaced by three unsigned integer arguments: j, k, l. Where Width is used to refer to the height of M or height of P, replace it with j. Where Width is used to refer to the width of M or height of N, replace it with k. Where Width is used to refer to the width of N or width of P, replace it with l. The revision of the kernel with these changes is left as an exercise.

4.7. Memory as a Limiting Factor to Parallelism

While CUDA registers and shared memory can be extremely effective in reducing the number of accesses to global memory, one must be careful to stay within the capacity of these memories. These memories are forms of resources that are needed for thread execution. Each CUDA device offers a limited amount of resources, which limits the number threads that can simultaneously reside in the SM for a given application. In general, the more resources each thread requires, the fewer the number of threads can reside in each SM, and thus the fewer number of threads that can run in parallel in the entire device.

Let's use an example to illustrate the interaction between register usage of a kernel and the level of parallelism that a device can support. Assume that in a current generation device D, each SM can accommodate up to 1,536 threads and has 16,384 registers. While 16,384 is a large number, it only allows each thread to use a very limited number of registers considering the number of threads that can reside in each SM. In order to support 1,536 threads, each thread can use only 16,384/1,536= 10 registers. If each thread uses 11 registers, the number of threads that are able to be executed concurrently in each SM will be reduced. Such reduction is done at the block granularity. For example, if each block contains 512 threads, the reduction of threads will be done by reducing 512 threads at a time. Thus, the next lower number of threads from 1,536 would be 1,024, a 1/3 reduction of threads that can simultaneously reside in each SM. This can greatly reduce the number of warps available for scheduling, thus reducing the processor's ability to find useful work in the presence of long-latency operations.

Note that the number of registers available to each SM varies from device to device. An application can dynamically determine the number of registers available in each SM of the device used and choose a version of the kernel that uses the number of registers appropriate for the device. This can be done by calling the `cudaGetDeviceProperties` function, whose use was discussed in Section 3.6. Assume that variable `&dev_prop` is passed to the function for the device property, the field `dev_prop.regsPerBlock` gives the number of registers available in each SM. For current generation devices, the returned value for this field should be 16,384. The application can then divide this number by the target number of threads to reside in each SM to determine the number of registers that can be used in the kernel.

Shared memory usage can also limit the number of threads assigned to each SM. Assume device D mentioned above has 16,384 (16K) bytes of shared memory in each SM. Keep in mind that shared memory is allocated to thread blocks. Assume that each SM in D can accommodate up to 8 blocks. In order to reach this maximum, each block must not use more than 2K bytes of shared memory. If each block uses more than 2K bytes of memory, the number of blocks that can reside in each SM is reduced such that the total amount of shared memory used by these blocks does not exceed 16K bytes. For example, if each block uses 5K bytes of shared memory, no more than three blocks can be assigned to each SM.

For the matrix multiplication example, shared memory can become a limiting factor. For a tile size of 16 × 16, each block needs 16×16×4 = 1K bytes of storage for Mds. (Keep in mind that each element is a float type, which is 4 bytes.) Another 1KB is needed for Nds. Thus each block uses 2K bytes of shared memory. The 16K-byte shared memory allows 8 blocks to simultaneous reside in an SM. Since this is the same as the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size. In this case, the real limitation is the threading hardware limitation that only 1,536 threads are allowed in each SM. This limits the number of blocks in each SM to six. As a result, only 6*2KB=12KB of the shared memory will be used. These limits do change from device to device but can be determined at runtime with device queries.

Note that the size of shared memory in each SM can also vary from device to device. Each generation or model of device can have different amount of shared memory in each SM. It is often desirable for a kernel to be able to use different amount of shared memory according to the amount available in the hardware. That is, we may want have a host code to dynamically determine the size of the shared memory and adjust the amount of shared memory used by a kernel. This can be done by calling the cudaGetDeviceProperties function. Assume that variable &dev_prop is passed to the function, the field dev_prop.sharedMemPerBlock gives the number of registers available in each SM. The programmer can then determine the number of amount of shared memory that should be used by each block.

Unfortunately, the kernel in Figure 4.16 does not support this. The declarations used in Figure 4.16 hardwire the size of its shared memory usage to a compile-time constant:

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

That is, the size of Mds and Nds is set to be TILE_WIDTH^2 elements, whatever the value of TILE_WIDTH is set to be at compile time. For example, assume that the file contains

```
#define TILE_WIDTH 16
```

Both Mds and Nds will have 256 elements. If we want to change the size of Mds and Nds, we need to change the value of TILE_WIDTH and recompile the code. The kernel cannot easily adjust its shared memory usage at runtime without recompilation.

We can enable such adjustment with a different style of declaration in CUDA. We can add a C “extern” keyword in front of the shared memory declaration and omit the size of the array in the declaration. Based on this style, the declarations for Mds and Nds become:

```
extern __shared__ Mds[];
extern __shared__ Nds[];
```

Note that the arrays are now one-dimensional. We will need to use a linearized index based on the vertical and horizontal indices.

At run time, when we launch the kernel, we can dynamically determine the amount of shared memory to be used according to the device query result and supply that as a **third** configuration parameter to the kernel launch. For example, the revised kernel could be launched with the following statements:

```
size_t size =
  calculate_appropriate_SM_usage(dev_prop.sharedMemPerBlock,...);

matrixMulKernel<<<dimGrid, dimBlock, size>>>(Md, Nd, Pd, Width);
```

where size_t is a built-in type for declaring a variable to holds the size information for dynamically allocated data structures. The size is expressed in number of bytes. In our matrix multiplication example, for a 16x16 tile, we have size to be 16x16x4=1024 bytes. We have omitted the details of the calculation for setting the value of size at run time.

4.8. Summary

In summary, the execution speed of a program in modern processors can be severely limited by the speed of the memory. To achieve good utilization of the execution throughput of CUDA devices, one needs to achieve a high compute-to-global-memory-access ratio in the kernel code. If the ratio is low, the kernel is memory-bound. That is, its execution speed is limited by the rate at which its operands are accessed from memory.

CUDA defines registers, shared memory, and constant memory. These memories are much smaller than the global memory but can be accessed at much higher speed. Using these memories effectively requires re-design of the algorithm. We use matrix multiplication as an example to illustrate tiling, a popular strategy to enhance locality of data access and enable effective use of shared memory. In parallel programming, tiling forces multiple threads to jointly focus on a subset of the input data at each phase of the execution so that the subset data can be placed into these special memory types to enable much higher access speed. We demonstrate that with 16×16 tiling, global memory accesses are no longer the major limiting factor for matrix multiplication performance.

It is, however, important for CUDA programmers to be aware of the limited sizes of these types of memory. Their capacities are implementation dependent. Once their capacities are exceeded, they limit the number of threads that can be simultaneously executing in each SM.

The ability to reason about hardware limitations when developing an application is a key aspect of computational thinking.

Although we introduced tiled algorithms in the context of CUDA programming, it is an effective strategy for achieving high-performance in virtually all types of parallel computing systems. The reason is that an application must exhibit locality in data access in order to make effective use of high-speed memories in these systems. For example, in a multi-core CPU system, data locality allows an application to effectively use on-chip data caches to reduce memory access latency and achieve high performance. Therefore, the reader will find the tiled algorithm useful when he/she develops a parallel application for other types of parallel computing systems using other programming models.

Our goal for this chapter is to introduce the concept of locality, tiling, and different CUDA memory types. We introduced a tiled matrix multiplication kernel using shared memory. We have not discussed the use of registers and constant memory in tiling. We will explain the use of these memory types in tiled algorithms when we discussed parallel algorithm patterns.

4.9. Exercises

1. Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: analyze the elements accessed by each thread and see if there is any commonality between threads.

2. Draw the equivalent of Figure 4.14 for a 8×8 matrix multiplication with 2×2 tiling and 4×4 tiling. Verify that the reduction in global memory bandwidth is indeed proportional to the dimension size of the tiles.

3. What type of incorrect execution behavior can happen if one forgot to use one or both __syncthreads() in the kernel of Figure 4.16?

4. Assuming capacity were not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.

5. For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?
   (A) 1/8 of the original usage
   (B) 1/16 of the original usage
   (C) 1/32 of the original usage

(D) 1/64 of the original usage

6. Assume that a CUDA kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?
   (A) 1
   (B) 1000
   (C) 512
   (D) 512000

7. In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
   (A) 1
   (B) 1000
   (C) 512
   (D) 51200

8. Consider performing a matrix multiplication of two input matrices with dimensions N×N. How many times is each element in the input matrices requested from global memory when:

   a) There is no tiling?

   b) Tiles of size TxT are used?

9. A kernel performs 36 floating point operations and 7 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.

   a) Peak FLOPS = 200 GFLOPS, Peak Memory Bandwidth = 100 GB/s

   b) Peak FLOPS = 300 GFLOPS, Peak Memory Bandwidth = 250 GB/s

10. To manipulate tiles, a new CUDA programmer has written the following device kernel which will transpose each tile in a matrix. The tiles are of size BLOCK_WIDTH by BLOCK_WIDTH, and each of the dimensions of matrix A is known to be a multiple of BLOCK_WIDTH. The kernel invocation and code are shown below. BLOCK_WIDTH is known at compile-time, but could be set anywhere from 1 to 20.

```
dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void
BlockTranspose(float* A_elements, int A_width, int A_height)
{
  __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];

  int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
  baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;

  blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];

  A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
}
```

a. Out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will this kernel function execute correctly on the device?

b. If the code does not execute correctly for all BLOCK_SIZE values, suggest a fix to the code to make it work for all BLOCK_SIZE values.