Chapter 4
CUDA Memories

So far, we have learned to write a CUDA kernel function which can be invoked by a massive number of threads. The data to be processed by these threads are first transferred from the host memory to the device global memory. The threads then access their portion of the data from the global memory using block and thread IDs. We have also learned the more details of the assignment and scheduling of threads for execution. Although this is a very good start, these simple CUDA kernels will likely achieve only a small fraction of the potential speed of the underlying hardware. This is due to the fact that global memory, which is typically implemented with Dynamic Random Access Memory (DRAM), tends to have long access latencies (hundreds of clock cycles) and limited 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 multiple Streaming Multiprocessors idle. In order to circumvent such congestion, CUDA provides a plethora of additional types of memories that can filter out a majority of data requests to the global memory. In this chapter, you will learn to use such memories to boost the execution efficiency of CUDA kernels.

4.1. Importance of Memory Access Efficiency

The effect of memory access efficiency can be illustrated by calculating the expected performance level of the simple matrix multiplication kernel code in Figure 3.4, replicated in Figure 4.1. The most important part of the kernel in terms of execution time is the *for* loop that performs inner product calculation. In every iteration of this loop, two global memory accesses are performed for one multiplication and one addition. 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 (CGMA) ratio*, defined as the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

```
global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockId.y * TILE_WIDTH + threadId.y;
// Calculate the column idenx of Pd and N
Int Col = blockId.x * TILE_WIDTH + threadId.x;

Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row][k] * Nd[k][Col];

Pd[Row][Col] = Pvalue;
}
```

Figure 4.1 Revised Matrix Multiplication Kernel using multiple blocks.

CGMA has major implications on the performance of a CUDA kernel. For example, the GeForce 8800GTX processor supports 86.4 Giga (10^9) Bytes per second, or 86.4 GB/s, of global memory access bandwidth. With a CGMA of 1.0 and 4 bytes in each single-precision floating-point datum, one can expect that the matrix multiplication kernel will execute at no more than 21.6 Giga Floating Point Operations per Cycle (GFLOPS), since each floating point operation requires four bytes of global memory data and 86.4/4=21.6. While 21.6 GFLOPS is a respectable number, it is only a tiny fraction of the peak performance of 367 GFLOPS for GeForce 8800GTX. We will need to increase the CGMA ratio in order to achieve a higher level of performance for the kernel.

4.2. CUDA Device Memory Types

Each CUDA device has several memories that can be used by programmers to achieve high CGMA ratio and thus high execution speed in their kernels. Figure 4.2 shows these CUDA device memories as implemented in the GeForce 8800GTX hardware. At the bottom of the picture, we see global memory and constant memory. These are the memories that the host code can write (W) and read (R) by calling API functions. We have already introduced global memory in Chapter 2. The constant memory allows read-only access by the device and provides faster and more parallel data access paths for CUDA kernel execution than the global memory.

Above the thread execution boxes in Figure 4.2 are registers and shared memories. Variables that reside in these memories 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 memories are allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block. Shared memories are efficient means for threads to cooperate by sharing the results of their work.

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread local memory
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**

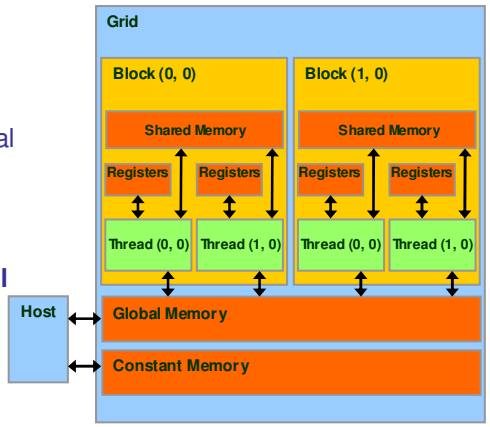

Figure 4.2 GeForce 8800GTX Implementation of CUDA Memories

Table 1 shows the CUDA syntax for declaring program variables into the various device memories. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of the entire grid. If a variable's scope is a single thread, a private version of the variable will be created for each and every thread; every thread can only access its own local 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.

Table 1. CUDA Variable Type Qualifiers

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

Lifetime specifies the portion of program execution duration when the variable is available for use: either within a kernel's invocation or throughout the entire application. If a variable's lifetime is within a kernel invocation, 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 contents of the variable are 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 the variable are maintained throughout the execution of the application and available to all kernels.

As shown in Table 1, all automatic variables except for arrays declared in kernel and device functions are placed into registers. We will refer to variables that are not arrays as *scalar* variables. 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 tx, ty, and Pvalue 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. We will address this point in Chapter 5.

Automatic array variables are not stored in registers. Instead, they are stored into the global memory and incur long access delays and potential access congestions. The scopes of these arrays are, same as automatic scalar variable, within individual threads. That is, a private version of such array is created and used for every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist. Due to the slow nature of automatic array variables, one should avoid using such variables. From our experience, one seldom needs to use automatic array variables in kernel functions and device functions.

If a variable declaration is preceded by keywords "__shared__'' (each "__'' consists of two "_'' characters), 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 must reside within a kernel function or a device function. 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. Shared variables are an efficient means for threads within a block to collaborate with each other. Accessing to shared memory is extremely fast and highly parallel. CUDA programmers often use shared memory 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 shortly with matrix multiplication.

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 "__device__" in front of "__constant__" to achieve the same effect. Declaration of constant variables must reside 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 variable 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 5.

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 global memory. Accesses to a global variable are very slow. However, global variable 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 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. Therefore, global variables are often used to pass information from one kernel execution to another kernel execution.

Note that there is a limitation on the use of pointers with CUDA variables declared into device memories. Pointers can only be used to point to data object in the global memory. There are two typical ways in which pointers usages arise 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 Md, Nd, and Pd 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;}` assigns the address of GlobalVar into an automatic pointer variable **`ptr`**.

4.3. A Strategy to Reduce Global Memory Traffic

We have an intrinsic tradeoff in the use of device memories in CUDA: global memory is large but slow whereas the shared memory is small but fast. A common strategy is 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 often be covered 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 structure can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example. Figure 4.3 shows a small example of matrix multiplication using multiple blocks in Figure 4.1. This example assumes that we use four 2X2 blocks to compute the Pd matrix. Figure 4.3 highlights the computation done by the four threads of block(0,0). These four threads compute $Pd_{0,0}$, $Pd_{1,0}$, $Pd_{0,1}$, and $Pd_{1,1}$.

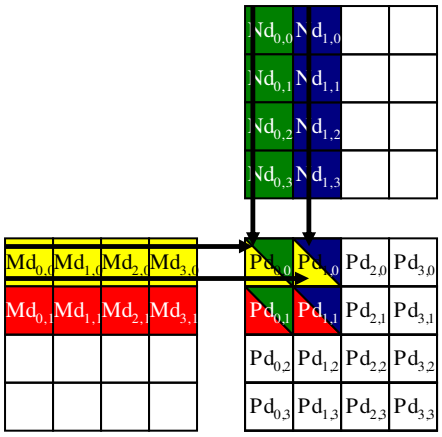

Figure 4.3 A small example of matrix multiplication using multiple blocks

Figure 4.4 shows the global memory accesses done by all threads in $block_{0,0}$. Note that each thread accesses four elements of Md and four elements of Nd during its execution. Among the four threads highlighted, there is a significant overlap of their accesses to Md and Nd. For example, $thread_{0,0}$ and $thread_{1,0}$ both access $Md_{1,0}$ as well as the rest of row 0 of Md. In Figure 4.1, the kernel is written so that both threads access these Md elements from the global memory. If we manage to have $thread_{0,0}$ and $thread_{1,0}$ to collaborate so that these Md elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In general, we can see that every Md and Nd element are accessed exactly twice during the execution of $block_{0,0}$. Therefore, if we can have all the four threads to collaborate in their accesses to global memory, we can reduce the traffic to the global memory by half.

Access order

$Pd_{0,0}$ $thread_{0,0}$	$Pd_{1,0}$ $thread_{1,0}$	$Pd_{0,1}$ $thread_{0,1}$	$Pd_{1,1}$ $thread_{1,1}$
$Md_{0,0} * Nd_{0,0}$	$Md_{0,0} * Nd_{1,0}$	$Md_{0,1} * Nd_{0,0}$	$Md_{0,1} * Nd_{1,0}$
$Md_{1,0} * Nd_{0,1}$	$Md_{1,0} * Nd_{1,1}$	$Md_{1,1} * Nd_{0,1}$	$Md_{1,1} * Nd_{1,1}$
$Md_{2,0} * Nd_{0,2}$	$Md_{2,0} * Nd_{1,2}$	$Md_{2,1} * Nd_{0,2}$	$Md_{2,1} * Nd_{1,2}$
$Md_{3,0} * Nd_{0,3}$	$Md_{3,0} * Nd_{1,3}$	$Md_{3,1} * Nd_{0,3}$	$Md_{3,1} * Nd_{1,3}$

Figure 4.4 Global memory accesses performed by threads in $block_{0,0}$

The reader should be able to verify that the potential reduction of global memory traffic in matrix multiplication is proportional to the dimension of the blocks used. With NxN blocks, the potential reduction of global memory traffic would be N. That is, if we use

16x16 blocks, one can potentially reduce the global memory traffic to 1/16 through collaboration between threads.

We now present an algorithm where threads collaborate to reduce the traffic to the global memory. The basic idea is to have the threads to collaboratively load Md and Nd elements into the shared memory before they individually use these elements in their dot product calculation. Keep 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 Md and Nd elements into the shared memory. This can be accomplished by dividing the Md and Nd 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 4.5.

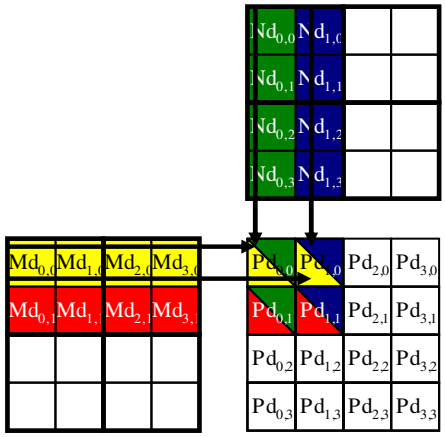

Figure 4.5 Tiling Md and Nd to utilize shared memory

In Figure 4.5, we further divide Md and Nd into 2X2 tiles. 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 Md and a tile of Nd into the shared memory. This is done by having every thread in a block to load one Md element and one Nd element into the shared memory, as illustrated in Figure 4.6. Each row of Figure 4.6 shows the execution activities of a thread. We only need to show the activities of threads in $block_{0,0}$; the other blocks all have similar behavior. The shared memory locations for the Md elements are Mds and Nd elements Nds. At the beginning of Phase 1, the four threads of $block_{0,0}$ collaboratively loads the a tile of Md into shared memory: $thread_{0,0}$ loads $Md_{0,0}$ into $Mds_{0,0}$, $thread_{1,0}$ loads $Md_{1,0}$ into $Mds_{1,0}$, $thread_{0,1}$ loads $Md_{0,1}$ into $Mds_{0,1}$, and $thread_{1,1}$ loads $Md_{1,1}$ into $Mds_{1,1}$. A tile of Nd is also loaded in a similar manner.

After the two tiles of Md and Nd are loaded into the shared memory, these values are used in the calculation of the dot product. Note that each value in the shared memory is used twice. For example, the $Md_{1,1}$ value, loaded by $Thread_{1,1}$ into $Mds_{1,1}$, is used twice, once by $thread_{0,1}$ 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 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 NxN elements.

	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Figure 4.6 Execution phases of a tiled matrix multiplication algorithm

Note that the calculation of each dot product in Figure 4.6 is now performed in two phases. In each phase, products of two pairs of the input matrix elements are accumulated into the Pvalue variable. In this example, the dot products are done in 2 phases. In an arbitrary case where the input matrix is of dimension N and the tile size is TILE_WIDTH, the dot product would be performed in N/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 the input needs of the phase of calculations.

Note also that the Mds and Nds locations are re-used to hold the input values. In each phase, the same locations are used to hold the subset of Md and Nd elements used in the phase. This allows a much smaller shared memory to screen away 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 exhibit locality, there is an opportunity to use small, high-speed memories to screen away most accesses to the global memory. We will return to the concept of locality in Chapter 5.

We are now ready to present the tiled kernel function that uses shared memory to reduce the traffic to global memory. This kernel shown in Figure 4.7 implements the phases illustrated in Figure 4.6. In Figure 4.7, Line 1 and Line 2 declare Mds as a shared memory variable. Recall that the scope of shared memory variables is a block. Thus, all threads of a block have access to the same Mds and Nds arrays. This is important since all threads in a

block must have access to the Md and Nd values loaded into Mds and Nds by each other so that they can avoid accessing global memory.

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, 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 Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.   int Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Coolaborative loading of Md and Nd tiles into shared memory
9.    Mds[tx][ty] = Md[m*TILE_WIDTH + tx][Row];
10.   Nds[tx][ty] = Nd[Col][m*TILE_WIDTH + ty];

11.   for (int k = 0; k < TILE_WIDTH; ++k)
12.       Pvalue += Mds[tx][k] * Nds[k][ty];

13.   Pd[Row][Col] = Pvalue;
    }
}
```

Figure 4.1 Tiled Matrix Multiplication Kernel using shared memories.

Lines 3 and 4 save the threadId and blockId values into automatic variables and thus into registers for fast access. Recall that automatic non-array 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. They will reside in registers that are accessible by one thread. They are initialized with the threaded and blockId values and used many times during the lifetime of thread. Once the thread ends, the values of these variables also cease to exist.

Lines 5 and 6 identify the row index and column index of the Pd element that the thread is to produce. As shown in Figure 4.8, the column (x) index of the Pd 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 x dimension. A thread in block bx would have bx blocks before it that will cover bx*TILE_WIDTH elements of Pd. Another tx trheads within the same block would cover another tx elements of Pd. Thus the thread with bx and tx should be responsible for covering the Pd element whose x index is bx*TILE_WIDTH+tx. For the example of Figure 4.5, the x index of the Pd element to be calculated by $thread_{1,0}$ of $block_{0,1}$ is 0*2+1 = 1. Similarly, the y index can be calculated as by*TILE_WIDTH+ty. In Figure 4.5, the y index of the Pd element to be calculated by $thread_{1,0}$ of $block_{0,1}$ is 1*2+0 = 2. Thus, the Pd element to be produced by this thread is $Pd_{1,2}$.

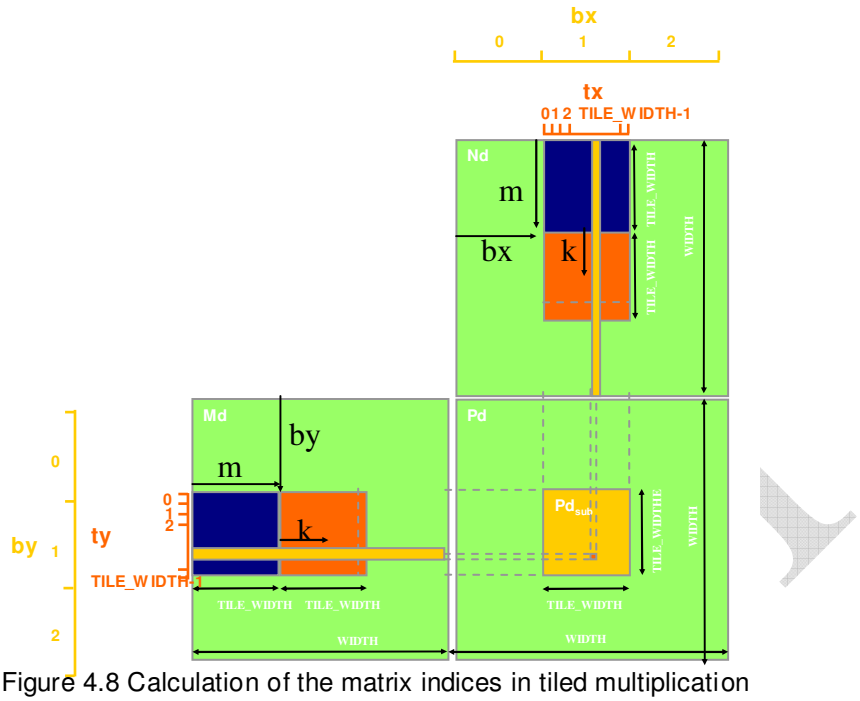

Figure 4.8 Calculation of the matrix indices in tiled multiplication

Line 8 of Figure 4.7 shows the loop that iterates through all the phases of calculating the final Pd element. Each iteration of the loop corresponds to one phase of the calculation shown in Figure 4.6. The `m` variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of Md and one tile of Nd elements. Therefore, at the beginning of each phase, m*TILE_WIDTH pairs of Md and Nd elements have been processed by previous phases.

Recall that all threads in a grid execute the same kernel function. The threadId variable allows them to identify the part of the data they are to process. Also recall that the thread with by=blockId.y and ty=threaded.y is to process row (by*TILE_WIDTH+ty) of Md, as shown at the left side of in Figure 4.8. Line 5 stores this number into the `Row` variable of each thread. Likewise, the thread with bx=blockId.x and tx=threadId.x is to process column (bx*TILE_WIDTH+tx) of Nd, as shown at the top side of Figure 4.8. Line 6 stores this number into the `Col` variable of each trhead. This will be used when the threads load Md and Nd elements into the shared memory.

In each phase, Line 9 loads the appropriate Md element into the shared memory. Since we already know the row index of Md and column index of Nd elements to be processed by the thread, we will focus on the column index of Md and row index of Nd. As shown in Figure 4.8, each block has $TILE\_WIDTH^2$ threads that will collaborate to load $TILE\_WIDTH^2$ Md elements into the shared memory. Thus, all we need to do is to assign each thread to load one Md element. This is conveniently done using the block and thread IDs. Note that the beginning index of the section of Md elements to be loaded is

m*TILE_WIDTH. Therefore, an easy approach is to have every thread to load an element from that point on identified by the thread ID. This is precisely what we have in Line 9, where each thread loads Md[m*TILE_WIDTH+tx][Row]. Since the value of Row is a linear function of ty, each of the TILE_WIDTH^2 threads will load a unique Md element into the shared memory. Altogether, these threads will load the orange square subset of Md shown in Figure 4.8. The reader should use the small example in Figure 4.5 and Figure 4.6 to verify that the address calculation works correctly.

Once the tiles of Md and Nd are loaded in Mds and Nds, the loop in Line 11 performs the phase of the dot product based on these elements. The progression of the loop for thread(tx,ty) is shown in Figure 4.8, with the direction of the Md and Nd data usage marked with `k`, the loop variable in Line 11. Note that the data will be accessed from Mds and Nds, the shared memory location holding these Md and Nd elements.

The benefit of the tiled algorithms is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of TILE_WIDTH. If one uses 16X16 tiles, we can reduce the global memory accesses by a factor of 16. This reduction allows the 86.4GB/s global memory bandwidth to serve a much larger floating point computation rate than the original algorithm. More specifically, the global memory bandwidth can now support ((86.4/4)*16) = 345.6 GFLOPS, very close to the peak floating-point performance of the GeForce 8800 GTX processor. This effectively removes the global memory bandwidth as the major limiting factor of matrix multiplication performance.

4.4. Memory as a Limiting Factor of Paralleism

While CUDA registers, shared memories, and constant memories can be extremely effective in reducing the number of accesses to the global memory, one must be careful not to exceed the capacity of these memories. Each processor implementation offers a limited amount of CUDA memories, which limits the number threads that can simultaneously reside in the Streaming Multiprocessors for a given application. In general, the more memory locations each thread requires, the fewer the number of threads can reside in each SM, and thus the fewer number of threads that can reside in the entire processor.

In the GeForce 8800 GTX implementation, each SM has 8K registers, which amounts to 128K registers for the entire processor. While this is a very large number, it only allows each thread to use a very limited number of registers. Recall that each SM can accommodate up to 768 threads. In order to achieve this maximal, each thread can use only 8K/768= 10 registers. If each thread uses 11 registers, the number of threads in each SM will be reduced. Such reduction is done at the block granularity. For example, if each block contains 256 threads, the reduction of threads will be done by reducing 256 threads at a time. Thus, the next lower number of threads from 768 would be 512, 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.

Shared memories can also limit the number of threads assigned to each SM. In the GeForce 8800 GTX processor, there are 16K bytes of shared memory in each SM. Keep in mind that shared memory is used by blocks. Recall that each SM 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 such that the total number of shared memories used by these blocks cannot 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, the shared memory can become a limiting factor. For a tile size of 16X16, each block needs a 16X16X4 = 1K bytes of storage of Mds. Another 1KB is needed for Nds. Thus each block uses 2K bytes of shared memory. The 16K bytes of shared memory allows 8 blocks to simultaneous reside in an SM. Since this is the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size. If we chose 32X32 tiles, each block needs 32*32*4*2 = 8K bytes of shared memory. Thus, only two blocks would be allowed to reside in each SM.

4.5. Summary

In summary, CUDA defines registers, shared memory, and constant memory that can be accessed at higher speed and in a more parallel manner than the global memory. Using these memories effectively will likely require re-design of the algorithm, We use matrix multiplication as an example to illustrate tiled algorithms, a popular strategy to enable effective use of shared memories. We demonstrate that with 16X16 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 special memories. Their capacities are implementation dependent. Once their capacities are exceeded, they become limiting factors for the number of threads that can be assigned to each SM.