Chapter 5

Performance Considerations

Although a CUDA kernel can run correctly on any CUDA device, its execution speed can vary greatly depending on the resource constraints of each device. In this chapter, we will discuss the major dimensions of resource constraints in an actual CUDA device and how they can constrain the level of parallel execution in this device. In order to achieve his/her goals, a programmer often has to find ways to achieve a required level of performance that is higher than that of an initial version of the application. In different applications, different constraints may dominate and become the limiting factors. One can improve the performance of an application on a particular CUDA device, sometimes dramatically, by trading one resource usage for another. This strategy works well if the resource constraint thus alleviated was actually the dominating constraint before the strategy was applied and the one thus exacerbated does not have worse effects on parallel execution. Without such understanding, performance tuning would be a guess work; plausible strategies may or may not lead to performance enhancements. Beyond insights into these resource constraints, this chapter further offers principles and case studies designed to cultivate intuition about the type of algorithms that can result in high performance execution CUDA devices. It is also establishes idioms and ideas that will likely lead to good performance improvements during your performance tuning efforts.

5.1. More on Thread Execution

Let's first discuss the aspects of thread execution that can limit performance. Recall that launching a CUDA kernel generates grid of threads that are organized into a two-level hierarchy. At the top level, a grid consists of a one- or two-dimensional array of blocks. At the bottom level, each block, in turn, consists of a one-, two-, or three-dimensional array of threads. In Chapter 3, we discussed the fact that blocks can execute in any order relative to each other, which allows for transparent scalability in parallel execution of CUDA kernels. However, we did not say much about the execution timing of threads within each block.

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

The G80/G280 implementation bundles several threads for execution. Each block is partitioned into *warps*. This implementation technique helps to reduce hardware cost and enable some optimizations in servicing memory accesses. In the foreseeable future, we expect that warp partitioning will remain as a popular implementation technique. However, the size of warp can easily vary from implementation to implementation. In G80/G280, each warp consists of 32 threads. We will use the G80/G280 implementation to explain warp partitioning for the rest of this chapter.

Thread blocks are partitioned into warps based on thread IDs. If a thread block is organized into a one-dimensional array, i.e., only threadIdx.x is used, the partition is straightforward. Thread IDs within a warp are consecutive and increasing. For warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63. In general, warp n starts with thread 32*n and ends with thread 32(n+1)-1. For a block whose number of threads is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32 threads. For example, if a block has 48 threads, it will be partitioned into 2 warps, and its warp 1 will be padded with 16 extra threads.

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

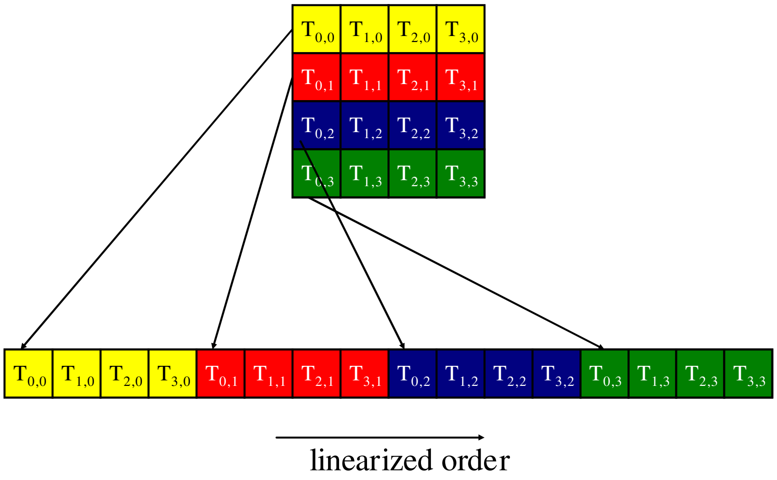

$T_{0,0}$ $T_{1,0}$ $T_{2,0}$ $T_{3,0}$
$T_{0,1}$ $T_{1,1}$ $T_{2,1}$ $T_{3,1}$
$T_{0,2}$ $T_{1,2}$ $T_{2,2}$ $T_{3,2}$
$T_{0,3}$ $T_{1,3}$ $T_{2,3}$ $T_{3,3}$
$T_{0,0}$ $T_{1,0}$ $T_{2,0}$ $T_{3,0}$ $T_{0,1}$ $T_{1,1}$ $T_{2,1}$ $T_{3,1}$ $T_{0,2}$ $T_{1,2}$ $T_{2,2}$ $T_{3,2}$ $T_{0,3}$ $T_{1,3}$ $T_{2,3}$ $T_{3,3}$
linearized order

Figure 5.1. Placing threads into linear order

Figure 5.1, shows an example of placing threads of a two dimensional block into linear order. The upper part shows the two-dimensional view of the block. Each thread is shown as $T_{x,y}$, x being the threadIdx.x and y being threadIdx.y for the thread. The lower part shows the linear view of the block. The first four threads are those threads whose trheadIdx.y is 0; they are placed with increasing threadIdx.x values. The next four threads are those trheads whose threadIdx.y is 1. They are also placed with their increasing threadIdx.x values. For this example, all 16 threads form half a warp. The warp will be padded with another 16 threads to complete a 32-thread warp. Imagine a 2 dimensional block with 8X8 threads. The 64 threads will form 2 warps. The first warp start from $T_{0,0}$ and ends with $T_{3,7}$. The second warp starts with $T_{4,0}$ and ends with $T_{7,7}$. It would be a useful exercise to draw out the picture as an exercise.

For a three dimensional block, we first place all threads whose threadIdx.z is 0 into the linear order. Among these threads, they are treated as a 2-dimensional block as shown in Figure 5.1. All threads whose threadIdx.z is 1 will then be placed into the linear order, and so on. A three dimensional 4X8X2 (4 in the x dimension, 8 in the y dimension, and 2 in the z dimension), the 64 threads will be partitioned into 2 warps, with $T_{0,0,0}$ through $T_{3,7,0}$ in the first warp and $T_{0,0,1}$ through $T_{3,7,1}$ in the second warp.

At any point in time, the hardware selects and executes one warp at a time. An instruction is run for all threads in the same warp, before moving to the next instruction. This style of execution is motivated by hardware cost constraints: it allows the cost of fetching and processing an instruction to be amortized among a large number of threads. It works well when all threads within a warp follow the same control flow path when working their data. For an if-then else construct, the execution works well when either all threads execute the then part or all execute the else part. When threads within a warp take different control flow paths, that is when some threads execute the then part and others execute the else part, the simple execution style no longer works well. In such situation, the execution of the warp will require multiple passes through these divergent paths. One pass will be needed for those threads that follow the then part and another pass for those that follow the else part. These passes are sequential to each other, thus will add to the execution time.

When threads in the same warp follow different paths of control flow, we say that these threads *diverge* in their execution. Divergence can arise in other constructs. For example, if threads execute a loop whose number of iterations can vary across threads, one additional pass must be taken for each case among the threads. For example, if threads in a warp execute the same for loop whose loop bound can be 6, 7, 8, or 9 iterations. All threads will finish the first 6 iterations together. One pass will be used to execute all threads that need the 7th iteration. One more pass will be used to execute all threads that need the 8th iteration. Yet another pass will be used to execute those that need the 9th iteration.

An if-the-else construct can result in thread divergence is when its decision condition is based on thread ID. For example, the statement "if (threadIdx.x > 2) {}" causes the threads to follow two divergent control flow paths. Threads 0, 1, and 2 follow a different path than threads 3, 4, 5, etc. Similarly, if a loop can cause thread divergence if its loop condition is based on thread ID. Such usages arise naturally in some important parallel algorithms. We will use a reduction algorithm to illustrate this point.

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

When there are a large number of elements in the array, the time needed to visit all elements of an array becomes large enough to motivate parallel execution. A parallel reduction algorithm typically resembles that of a soccer tournament. In fact, the elimination process of the world cup is a reduction of "maximal" where the maximal is defined as the team that "beats" all other teams. The tournament "reduction" is done by multiple rounds. The teams are divided into pairs. During the first round, all pairs play in parallel. Winners of the first round advance to the second round, whose winners advance to the third round, etc. With 16 teams entering a tournament, the 8 winners will emerge from the first round, 4 winners the second round, 2 winners the third round, and 1 final winner the fourth round. It should be easy to see that even with 1024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough soccer fields to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on. With enough fields, even with sixty thousand teams, we can determine the final winner in just 16 rounds. Of course, one would need to have enough soccer fields and enough officials to accommodate the thirty thousand games in the first round, etc.

Figure 5.2 shows a kernel function that performs sum reduction. The original array is in the global memory. Each thread block reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction. The reduction in done in place, which means the elements in the shared memory will be replaced by partial sums. Each iteration of the while loop in the kernel function implements a round of reduction. The syncthreads() statement in the while loop ensures that all threads are ready to enter the next iteration before any thread is allowed to do so. Therefore, all threads that enter the second iteration will be using the values produced in the first iteration. After the first round, the even elements will be replaced by the partial sums generated in the first round. After the second round, the elements whose indices are multiples of four will be replaced with the partial sums. After the final round, the total sum of the entire section will be in element 0.

```
1. __shared__ float partialSum[]

2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = 1;
4.        stride < blockDim.x; stride *= 2)
5. {
6.  __syncthreads();
7.  if (t % (2*stride) == 0)
8.   partialSum[t] += partialSum[t+stride];
9  }
```

Figure 5.2 A simple sum reduction kernel.

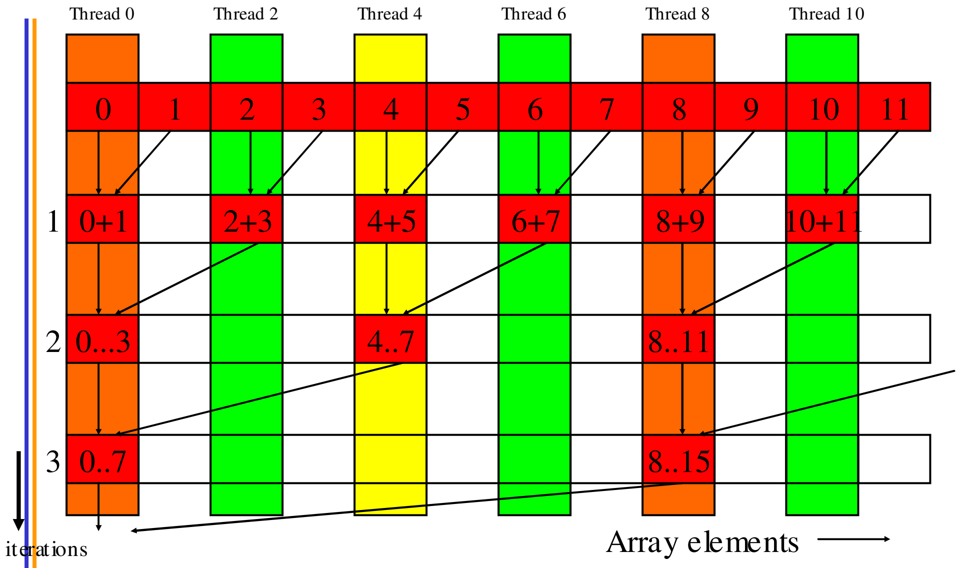


Figure 5.3 Execution of the sum reduction kernel.

In Figure 5.2, the Line 3 initializes the stride variable to 1. During the first iteration, the if statement in Line 7 is used to select only the even threads to perform addition between two neighboring elements. The execution of the kernel is illustrated in Figure 5.3. The threads and array elements are shown in the as columns and the iterations are shown as rows. As shown in row 1 in Figure 5.3, the even elements of the array now hold the pair-wise partial sums. Before the second iteration, the value of the stride variable in doubled to 2. During the second iteration, only those threads whose indices are multiples of four, shown as orange and yellow columns in Figure 5.3, will execute the add statement. Each thread generates a partial sum that covers four elements, as shown in the row 2 in Figure 5.3. With 512 elements in each section, the kernel function will generate the total for the entire section after 9 iterations.

The kernel in Figure 5.3 clearly has thread divergence. During the first iteration of the loop, only those threads whose threadIdx.x are even will execute the add statement. One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute the add statement. In each successive iteration, fewer threads will execute the add statement but two passes will be still needed to execute all the threads. This divergence can be reduced with a slight change to the algorithm.

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
  stride > 1;  stride >> 1)
{
 __syncthreads();
  if (t < stride)
   partialSum[t] += partialSum[t+stride];
}
```

Figure 5.4 A kernel with fewer thread divergence

Figure 5.4 shows a modified kernel with a slightly different algorithm for sum reduction. Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other during the first round. It does so by initializing the stride to be half the size of the section. All pairs added during the first round are half the section size away from each other. After the first iteration, all the pair-wise sums are stored in the first half of the array. The loop divides the stride by 2 before entering the next iteration. This is done by shifting the stride value to the right by one bit, a much less expensive way to implement divide by 2 than a real integer division.

Figure 5.5 illustrates the execution of the revised kernel. During the first iteration, all threads whose threadIdx.x value are less than half of the size of the section execute the add statement. For a section of 512 elements, Threads 0 through 255 execute the add statement during the first iteration while threads 256 through 511 do not. The pair-wise sums are stored in elements 0 through 255 after the first iteration. Since the warps consists of 32 threads with consecutive threadIdx.x values, all threads in warps 1 through warp 8 execute the add statement whereas warps 9 through warp 15 execute all skip the add statement. Since all threads in each warp take the same path, there is no thread divergence!

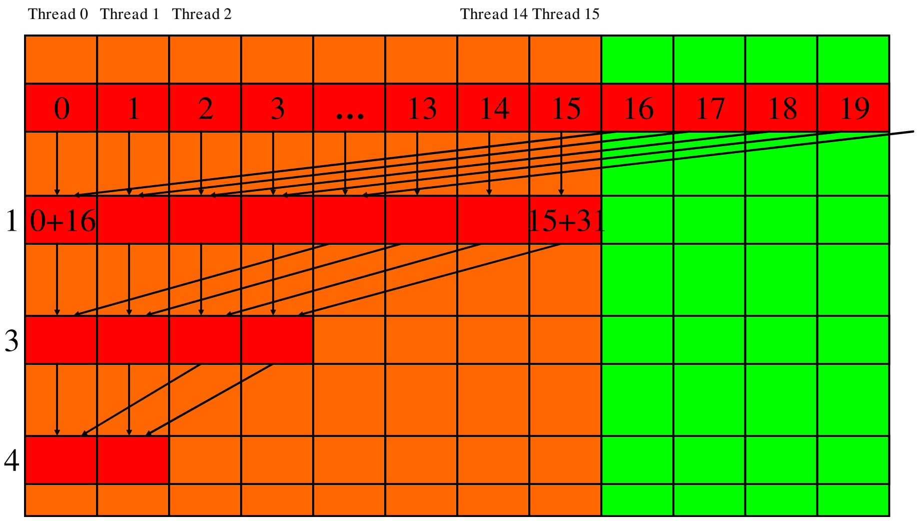


Figure 5.5 Execution of the revised algorithm

5.2. Global Memory Performance

One of the most important dimensions of CUDA kernel performance is accessing data in the global memory. CUDA applications exploit massive data parallelism that comes from processing as massive amount of data simultaneously. Therefore, CUDA applications typically process a massive amount of data within a short period of time. In particular, a CUDA kernel must be able to access a massive amount of data from the global memory within a very short period of time. In Chapter 4, we discussed tiling techniques that utilize shared memories to reduce the total amount of data that must be accessed by a collection of threads in the thread block. In this Chapter, we will further discuss memory coalescing techniques that can more effectively move data from the global memory into shared memories and registers. Memory coalescing techniques are often used in conjunction with tiling techniques to allow CUDA devices to reach their performance potential in the presence of limited data access bandwidth of the global memory.

Global memory in a CUDA system is typically implemented with Dynamic Random Access Memories, or DRAMs. Data bits are stored in DRAM cells that are very weak capacitors, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1. Reading data from a DRAM cell that contains a 1 requires the weak capacitor to share its tiny amount of charge to a sensor and set off a detection mechanism that determines whether a sufficient amount of charge is present in the capacitor. Because this is a very slow process, modern DRAMs use a parallel process to increase their rate of data access. Each time a location is to be accessed, many consecutive locations that includes the requested location are accessed. Many sensors are provided in each DRAM chip and they work in parallel, each sensing the contents of a bit location within these consecutive locations. Once detected by the sensors, the data from all these consecutive locations can then be transferred at very high speed to the processor. If an application can make use of data from multiple, consecutive locations before moving on to other locations, the DRAMs can supply the data at much higher rate than if a truly random sequence of locations were accessed. In order to achieve anywhere close to the advertised 84.6GB/sec global memory bandwidth for G80, a kernel must arrange its data accesses so that each request to the DRAMs is for a large number of consecutive DRAM locations.

Recognizing the organization of modern DRAMs, G80/280 designs employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads to exhibit favorable access patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive global memory locations. That is, the most favorable access pattern is achieved when the same instruction for all threads in a warp accesses consecutive global memory locations. In this case, the hardware combines, or coalesces, all these accesses into a consolidated access to the DRAMs that requests all consecutive locations involved. For example, for a given load instruction of a warp, if thread 0 accesses global memory location N, thread 1 location N+1, thread 2 location N+2, etc, all these accesses will be coalesced, or combined into a single request for all consecutive locations when accessing the DRAMs. Such coalesced access allows the DRAMs to deliver data at a rate close to the maximal global memory bandwidth.

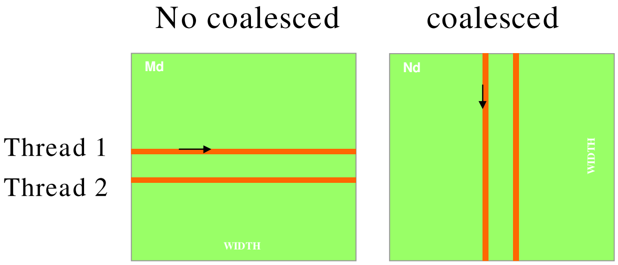


Figure 5.6 Memory access patterns for coalescing.

Figure 5.6 illustrates the favorable vs. unfavorable C program matrix data access patterns for memory coalescing. In part (a), illustrates the data access pattern of a loop where each thread reads a row of matrix Md. Assume that threads in a warp read adjacent rows. That is, during iteration 0, threads in warp 0 read element 0 of rows 0 through 31. During iteration 1, these same threads read element 1 of rows 0 through 31. None of the accesses will be coalesced. A more favorable access pattern is shown in Figure 5.6(b), where each thread reads a column of Nd. During iteration 0, threads in warp 0 reads element 1 of columns 0 through 31. All these accesses will be coalesced. In order to understand why the pattern in 5.6(b) is more favorable than that in 5.6(b), we need to understand how these matrix elements are placed into the global memory.

All locations in the global memory form a single, consecutive address space. That is, every location in the global memory has a unique address. This is analogous to a very long street where every house has a unique address. For example, if the global memory contains 1,024 locations, these locations will be accessed by address 0 through 1023. The G208 can have up to 4GB (2^{32}) locations; the addresses range from 0 to $2^{32} – 1$. All variables of a CUDA program are placed into this linear address space and will be assigned an address.

Matrix elements are placed into the linearly addressed locations according to the *row major* convention. That is, the elements of row 0 of a matrix are first placed in order into consecutive locations. They are followed by the elements of row 1 of the matrix, and so on. In other words, all elements in a row are placed into consecutive locations and entire rows are placed one after another. The term row major refers to the fact that the placement of data preserves the structure of rows, all adjacent element in a row are placed into consecutive locations in the address space. This is illustrated with an example in Figure 5.7, where the 16 elements of a 4X4 matrix M are placed into linearly addressed locations. The four elements of row 0 are first placed in their order of appearance in the row. Elements in row 1 are then placed, followed by elements of row 2, followed by elements of row 3. It should be clear that M0,0 and M0,1, though appear to be consecutive in the two dimensional matrix, are placed four locations away in the linearly addressed memory.

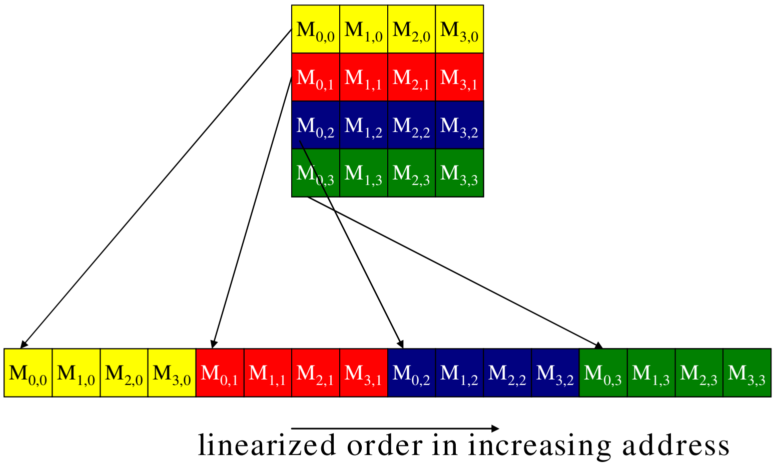

$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
linearized order in increasing address

Figure 5.7 Placing matrix elements into linear order

Now that we understand the placement of matrix elements into global memory, we are ready to understand the favorable vs. unfavorable matrix data access patterns in Figure 5.6. Figure 5.8 shows an example of the favorable access pattern in accessing a 4X4 matrix. The arrow in the top portion of the Figure shows the access pattern of the kernel code for one thread. The accesses are generated by a loop where threads in a warp access element 0 of the columns in the first iteration. As shown in the bottom portion of Figure 5.8, these elements are in consecutive locations in the global memory. The hardware detects that these accesses are to consecutive locations in the global memory and coalesces these accesses into a consolidated access. This allows the DRAMs to supply data at high rate.

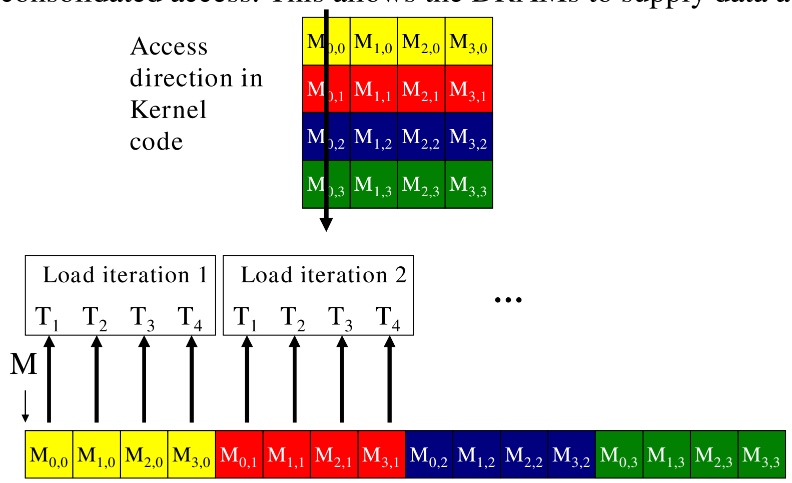

Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Load iteration 1
T_1 T_2 T_3 T_4
Load iteration 2
T_1 T_2 T_3 T_4
…
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Figure 5.8 A coalesced access pattern.

Figure 5.9 shows an example of matrix data access pattern that are not coalesced. The arrow in the top portion of the figure shows that the kernel code for each thread accesses elements of a row in sequence. The accesses are generated by a loop where threads in Warp 0 access element 0 of the columns during the first iteration. As shown in the bottom portion of Figure 5.9, these elements are in locations that are four elements away from each other. As a result, the hardware will determine that accesses to these elements cannot be coalesced. As a result, when a kernel loop iterates through a row, the accesses to global memory are much less efficient than the case where a kernel iterates through a column.

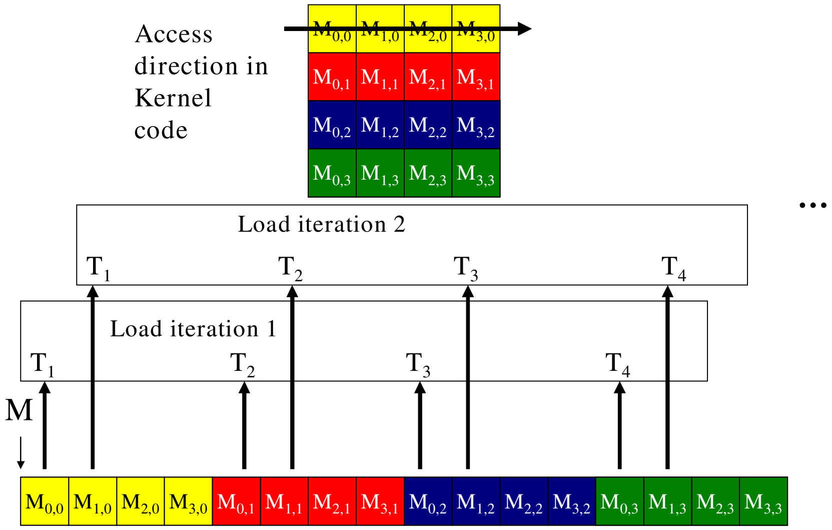

Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Load iteration 2
T_1 T_2 T_3 T_4
…
Load iteration 1
T_1 T_2 T_3 T_4
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Figure 5.9 An un-coalesced access pattern.

If the algorithm intrinsically requires a kernel code to iterate through data within rows, one can use the shared memory to enable memory coalescing. The technique is illustrated in Figure 5.10 for matrix multiplication. Each thread reads a row from Md, a pattern that cannot be coalesced. A tiled algorithm can be used to enable coalescing. As described in Section 4, threads of a block first cooperatively load the tiles into the shared memory. Care can be taken to ensure that these tiles are loaded in a coalesced pattern. Once the data is in shared memory, they can be accessed either on a row basis or a column basis without any performance penalty because the shared memories are implemented as intrinsically high-speed on-chip memory that does not require coalescing to achieve high data access rate.

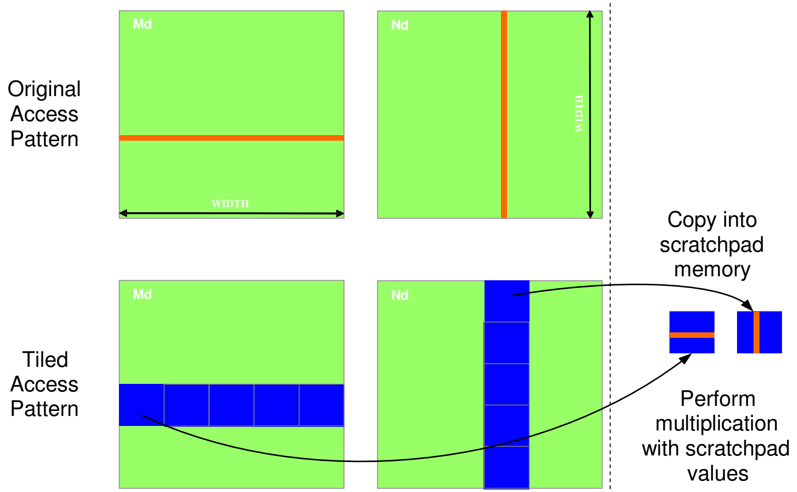
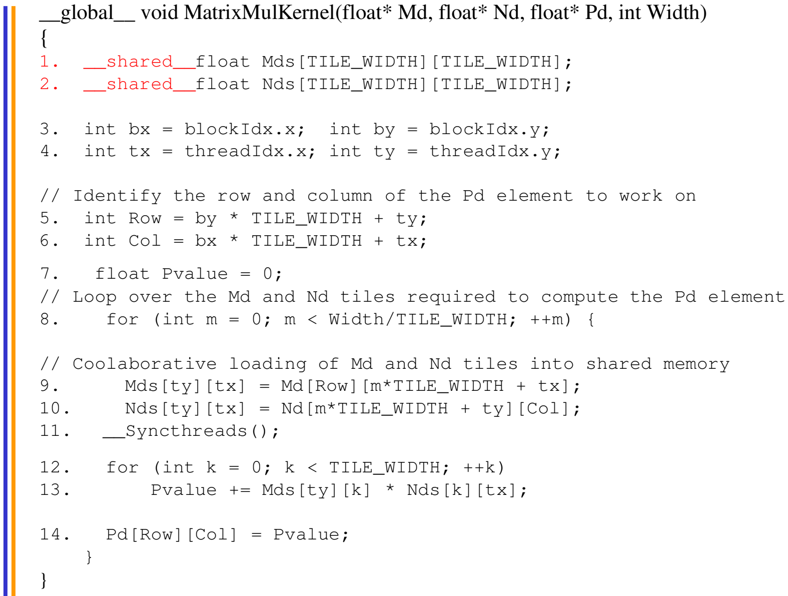

Original Access Pattern
Md
Nd
WIDTH
WIDTH
Copy into scratchpad memory
Tiled Access Pattern
Md
Nd
Perform multiplication with scratchpad values

Figure 5.10 Using shared memory to enable coalescing.

However, one has to take care so that the loading of tiles from global memory to shared memory is coalesced. We replicate Figure 4.7 here as Figure 5.11, where the matrix multiplication kernel loads two tiles of matrix Md and Nd into the shared memory. Note that each thread in a thread block is responsible for loading one Md element and one Nd element into Mds and Nds in each iteration of the for loop defined in line 8. Recall that there are TILE_WIDTH^2 threads involved in each tile. The threads use threadIdx.y and threadIdx.y to determine the element of each matrix to load.

For Md, the index calculation for each thread uses m to locate the left end of the tile. Each row of the tile is then loaded by TILE_WIDTH threads whose thread IDs differ in the x dimension. Since these threads have consecutive threadIdx.x values, they are in the same warp. Also, recall that elements in the same row are placed into consecutive locations of the global memory. The hardware detects that these threads in the same warp access consecutive locations in the global memory and combine them into a coalesced access.

In the case of Nd, the index calculation for each thread uses m to locate the upper end of the tile. Each row of the thread is then also loaded by TILE_WIDTH threads whose thread IDs differ only in the x dimension. Once again, these threads are in the same warp because they have consecutive trheadIdx.x values. The hardware detects that these threads in the same warp access consecutive location in the global memory and combine them into a coalesced access.



```
__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.   float 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[ty][tx] = Md[Row][m*TILE_WIDTH + tx];
10.     Nds[ty][tx] = Nd[m*TILE_WIDTH + ty][Col];
11.   __Syncthreads();

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

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

Figure 5.10 Tiled Matrix Multiplication Kernel using shared memories.

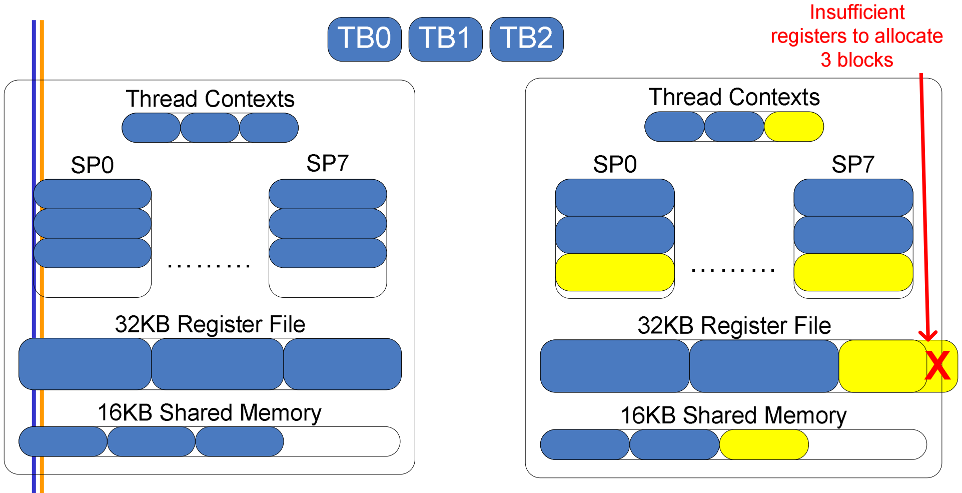
The reader shall find it useful to draw a picture base don the kernel code in Figure 5.10 and identify the threadIdx.y and threadIdx.x values of the thread that loads each element of the tile. Lines 5, 6, 9, 10 in Figure 5.10 form a frequently used programming pattern for loading elements into shared memory in tiled algorithms. We would also like to encourage the reader to analyze the data access pattern by the dot-product loop in lines 12 and 13. Note that the threads in a warp do not access consecutive location of Mds. This is not a problem since Mds is in shared memory, which does not require coalescing to achieve high speed data access.

5.3. Dynamic Partitioning of SM Resources

The execution resources in a Streaming Multiprocessor, or SM, include registers, thread block slots, and thread slots. These resources are dynamically partitioned and assigned to threads to support their execution. In Chapter 3, we have seen that each SM has 768 thread slots, each of which can accommodate one thread. These thread slots are partitioned and assigned to thread blocks during runtime. If each thread block consists of 256 threads, the 768 threads slots are partitioned and assigned to three blocks. In this case, each SM can accommodate up to three thread blocks due to limitations on thread slots. If each thread block contains 128 threads, the 768 thread slots are partitioned and assigned to 6 thread blocks. The ability to dynamically partition the thread slots among thread blocks makes the streaming multiprocessors versatile. They can either execute many thread blocks each of which consists of few threads or execute few thread blocks each of which consists of many threads. This is in contrast to a fixed partitioning method where each block receives a fixed amount of resource regardless of their real needs. Such fixed partitioning results in wasted thread slots when a block has few threads and fails to support blocks that require more thread slots than the fixed partition allows.

Dynamic partitioning of resources can result in subtle interactions between resource limitations, which in turn cause underutilization of resources. Such interactions can occur between block slots and thread slots. If each block has 64 threads, the 768 thread slots can be partitioned and assigned to 12 blocks. However, since there are only 8 block slots in each SM, only 8 blocks will be allowed. This means that only 512 of the thread slots will be utilized. Therefore, to fully utilize both the block slots and thread slots, one needs at least 96 threads in each block.

The register file is another dynamically partitioned resource. The number of registers in each CUDA device is not specified in the language and varies across implementations. In G80, there is an 8192-entry register file in each SM. These registers are used to hold frequently used programmer and compiler-generated variables to reduce their access latency and to conserve memory bandwidth. As we mentioned in Chapter 4, the automatic variables declared in a CUDA kernel are placed into registers. Some kernels may use lots of automatic variables and others may use few of them. Thus, one should expect that some kernels require many registers and some require fewer. By dynamically partitioning the registers among blocks, the SM can accommodate more blocks if they require few registers and fewer blocks if they require more registers. One does, however, need to be aware of potential interactions between register limitations and other resource limitations.



(a) Pre-“optimization”

(b) Post-“optimization”

Figure 5.12 Interaction of resource limitations

In the matrix multiplication example, assume that the kernel code uses 10 registers per thread. If we have 16X16 thread blocks, how many threads can run on each SM? We can answer this question by first calculate the number of registers needed for each block, which is 10*16*16= 2560. The number of registers required by three blocks is 7680, which is under the 8,120 limit. Adding another block would require 10240 registers, which exceeds the limit. Therefore, as shown in Figure 5.12(a), the register limitation allows 3 blocks that altogether have 768 threads to run on each SM, which also fits within the limit of 8 block slots and 768 thread slots.

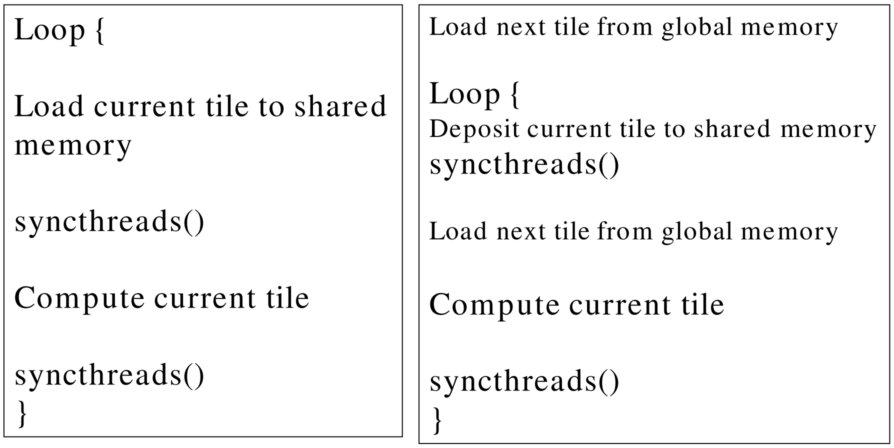
Now assume that the programmer declares another automatic variable in the kernel and bump the number of registers used by each thread to 11. Assuming the same 16X16 blocks, each block now requires 11*16*16 = 2,816 registers. The number of registers required by three blocks is now 8,448, which exceeds the register limitation. As shown in Figure 5.12(b), the SM deals with this situation by reducing the number of blocks by one, thus reducing the number of registered required to 5,632. This, however, reduces the number of threads running on an SM from 768 to 512. That is, by using one extra automatic variable, the program saw a 1/3 reduction in the thread-level parallelism in G80 execution! This is sometimes a referred to as a "performance cliff" where a slight increase in resource usage can result in dramatic reduction in parallelism and performance achieved.

In some cases, adding an automatic variable may allow the programmer to improve the execution speed of individual threads by initiating time consuming memory accesses early, as we will explain in more detail later in this chapter. The improvement within each thread may be sufficient to overcome the loss of thread-level parallelism. For example, assume that in the original kernel, there are four independent instructions between a global memory load and its use. In G80, each instruction takes 4 clock cycles to process. So the 4 independent instructions give a 16 cycle slack for the memory access. With a 200-cycle global memory latency, we need to have at least 200/(4*4) = 14 warps available for zero-overhead scheduling to keep the execution units fully utilized.

Assume that the additional register allows the programmer or the compiler to use a program transformation technique to increase the number of independent instructions from 4 to 8. These independent instructions give 1 32 cycle slack for the memory access. With the same 200-cycle global memory latency, we now only need 200/(4*8) = 7 warps available for zero-overhead scheduling to keep the execution units fully utilized. That is, even though we just reduced the number of blocks from 3 to 2, and thus the number of warps from 24 to 16, we may have enough warps to fully utilize the execution units in each SM. Thus, the performance may actually increase! A programmer typically needs to experiment with each alternative and choose the best performing code. This can be a labor intensive, tedious process. Ryoo et al have proposed a methodology for automating the experimentation process to reduce the programming efforts required to reach an optimal arrangement for each variety of CUDA hardware [RyooCGO2008].

5.4. Data Prefetching

One of the most important resource limitations for parallel computing in general is that global memory has limited bandwidth in serving data accesses and these accesses take a long time to complete. The tiling techniques for using shared memory address the problem of limited global memory bandwidth. The CUDA threading model tolerates long memory access latency by allowing some warps to make progress while others wait for their memory access results. While this is a very powerful mechanism, it may not be sufficient in some cases where all threads are waiting for their memory access results. Such a situation can arise if all threads have very small number of independent instructions between memory access instructions and the consumer of the data accessed.



```
Loop {
    Load current tile to shared memory
    syncthreads()
    Compute current tile
    syncthreads()
}
```

(a) Without prefetching

```
Load next tile from global memory
Loop {
    Deposit current tile to shared memory
    syncthreads()
    Load next tile from global memory
    Compute current tile
    syncthreads()
}
```

(b) With prefetching

Figure 5.13 Data Prefetching.

A useful, complementary solution to the problem is to prefetch the next data elements while consuming the current data elements, which increases the number of independent instructions between the memory accesses and the consumers of the data accessed. Prefetch techniques are often combined with tiling to simultaneously address the problems of limited bandwidth and long latency. We show such a combined approach in Figure 5.13.

The algorithm in Figure 5.13(a) corresponds to the tiled matrix multiplication kernel in Figure 5.10. Lines 9 and 10 in Figure 5.10 correspond to “load current tile to shared memory” in Figure 5.13(a). This is the part that loads data from global memory into shared memory. The dot-product code (lines 12 and 13) in Figure 5.10 correspond to “compute current tile” in Figure 5.13(a). This is the part that consumes the loaded data. Note that there is no substantial activity between the two parts. That is, there are few independent instructions between these two parts.

Figure 5.13(b) shows a prefetch version of matrix multiplication kernel. This technique allocates twice the amount of shared memory for each tile, one holds the tile currently being processed and one holds the tile to be processed next. Before we enter the while loop, we load the first tile into the registers. Once we enter the loop, we move the loaded data into shared memory. Since this is a consumer of the loaded data, the threads will likely need to be put to sleep, waiting for its loaded data while other threads make progress. When the first tile of data arrive, threads in the block pass the barrier synchronization and deposit the tile data from their registers to the shared memory. When all threads of a block complete depositing their data, they pass the barrier synchronization point and begin to load the next tile into their registers. The key is that the next tile data loaded is not immediately consumed. Rather, the current block is processed from the shared memory by the dot-product loop of lines 12 and 13 in Figure 5.10.

When the loop iterates, the “next tile” in the current iteration becomes “current tile” of the next iteration. Thus, the deposit of the “current tile” into the shared memory corresponds to the “next tile” loaded in the previous iteration. The execution of the dot-product loop provides many independent instructions between the two parts. This reduces the amount of time the threads will need to wait for their global memory access data.

We would like to encourage the reader to revise the kernel in Figure 5.10 to use prefetch. A cost of the data prefetch is that that it uses two additional automatic variables (registers). As we discussed in Section 5.3, using additional registers can reduce the number of blocks that can run on an SM. However, this technique can still win if it significantly reduces the amount of time each thread waits for its global memory load data.

5.5. Instruction Mix

In current generation CUDA GPUs, each processor core has limited instruction processing bandwidth. Every instruction consumes instruction processing bandwidth, whether it is a floating point calculation instruction, a load instruction, or a branch instruction. Figure 5.14(a) shows the dot-product loop of the matrix multiplication kernel. The loop incurs extra instructions to update loop counter k and performs conditional branch at the end of each iteration. Furthermore, the use of loop counter k to index the Ms and Ns matrices incurs address arithmetic instructions. These instructions compete against floating point calculation instructions for the limited instruction processing bandwidth.

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

(a) loop incurs overhead instructions

```
Pvalue += Ms[ty][k] * Ns[k][tx] + …
              Ms[ty][k+15] * Ns[k+15][tx];
```

(b) loop unrolling eliminates overhead

Figure 5.14 Loop unrolling improves instruction mix

For example, the kernel loop in 5.14(a) executes 2 floating point arithmetic instructions, one loop branch instruction, two address arithmetic instructions, and one loop counter increment instruction. That is, only 1/3 of the instructions executed are floating-point calculation instructions. With limited instruction processing bandwidth, this instruction mixture limits the achievable performance to no more than 1/3 of the peak bandwidth.

A common way to improve the instruction mix is to unroll the loop, as shown in Figure 5.14(b). Given a tile size, one can simply unroll all the iterations and simply express the dot product computation as one long multiply-add expression. This eliminates the branch instruction and the loop counter update. Furthermore, since the indices are constants rather than K, the compiler can use the addressing mode offsets of the load instructions to eliminate address arithmetic instructions. As a result, the long expression can execute at close to peak performance!

Ideally, loop unrolling should be automatically done by the compiler. This is one of the areas where compiler technology will likely be improved rapidly in the near future. Until the tools mature, many programmers will still unroll loops in their source code to achieve high performance.

5.6. Thread Granularity

An important algorithmic decision in performance tuning is the granularity of threads. It is often advantageous to put more work into each thread and use fewer threads. Such advantage arises when some redundant work exists between threads. Figure 5.14 illustrates such an opportunity in matrix multiplication. The tiled algorithm in Figure 4.8 uses one thread to compute one element of the output Pd matrix. This requires a dot product between on row of Md and one column of Nd.

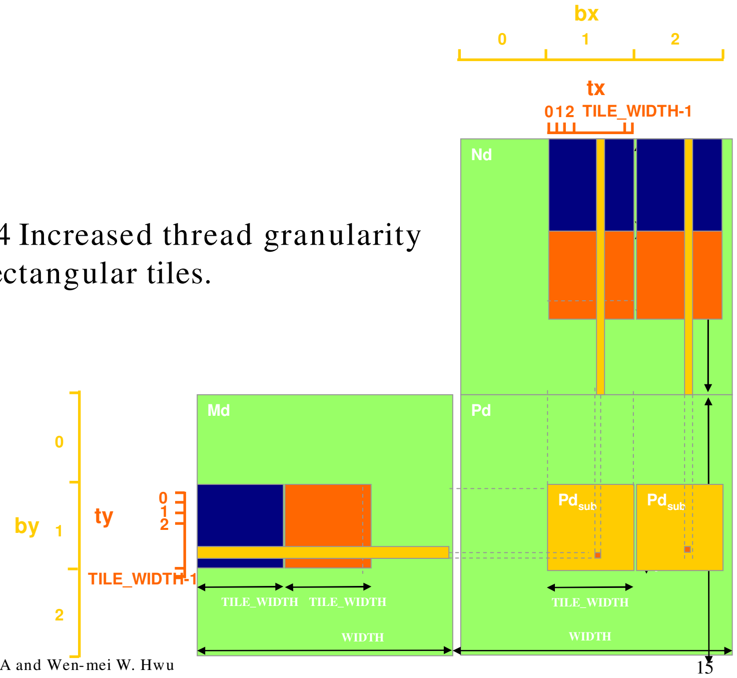


Figure 5.14 Increased thread granularity with rectangular tiles.

© David Kirk/NVIDIA and Wen-mei W. Hwu 15

The opportunity of thread granularity adjustment comes from the fact that multiple threads redundantly load each Md row. As shown in Figure 5.14, the calculation of two Pd elements in adjacent tiles uses the same Md row. With the original tiled algorithm, the same Md row is redundantly loaded by the two thread blocks assigned to generate these two Pd tiles. One can eliminate this redundancy by merging the two thread blocks into one. Each thread in the new thread block now calculates two Pd elements. This is done by revising the kernel so that each two dot-products are computed by the kernel. Both dot products use the same Mds row but different Nds columns. This reduces the global memory access by ¼. It also increases the number of independent instructions in the case of a prefetch algorithm in Figure 5.13 since there are two dot-products calculated between the loading of the tiles into registers and depositing these tiles into shared memories.

The potential downside is that the new kernel now uses more registers and shared memory. Thus the number of blocks that can be running on each SM may decrease. It also reduces the total number of thread blocks by half, which may result in insufficient amount of parallelism for matrices of smaller dimensions. For G80/G280, we found that combining four adjacent horizontal blocks to compute for adjacent horizontal tiles gives the best performance for a 2048x2048 matrix multiplication.

5.7. Experimental Performance Tuning

The combined effects of various performance enhancement techniques on the matrix multiplication kernel are shown in Figure 5.14. The dimensions covered are tiling size, loop unrolling, data prefetching, and thread granularity. We can make at least four observations.

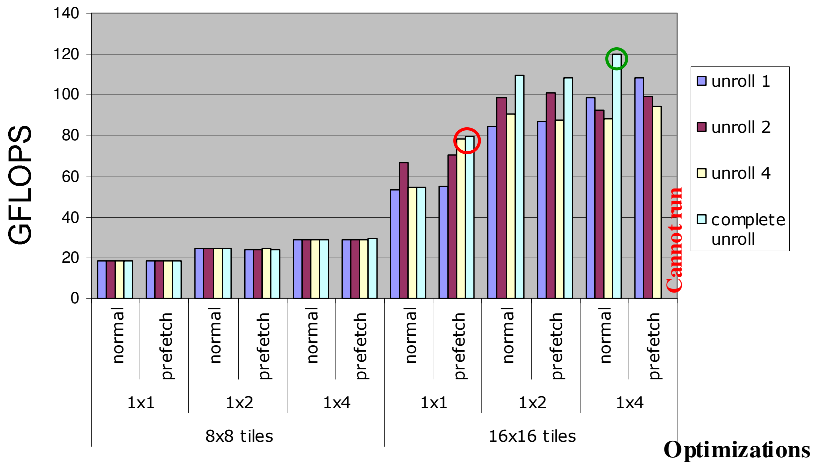


Figure 5.15 Effects of Performance Improvement Techniques

First, the tile size plays a major role in the performance. Until the tile size reaches 16X16, neither loop unrolling nor data prefetch helps. This is reflected by the fact that all eight bars in granularity bracket are of the same height. For small tile sizes such as 8x8, the saturated global memory bandwidth so severely limits the execution performance that transformations such as loop unrolling and data prefetching simply do not matter. On the other hand, since rectangular tiling can reduce global memory accesses, one would expect that it should improve performance. That is, 1x2 rectangular tiling reduces the global memory access by ¼ and resulted and 1X4 tiling by 3/8. Note that 1X8 would have reduced the global traffic by only 7/16, a diminishing return that makes it much less attractive than using a larger tile size. The reductions in global memory accesses indeed help improve the performance shown in Figure 5.15.

Second, once the tile size becomes sufficiently large, 16X16 in this case, to alleviate the saturation of global memory bandwidth, loop unrolling and data prefetching become much more important. In most cases, complete unrolling the loop can result in more than 20% performance improvement.

The third observation is that while data prefetching is very beneficial for 1x1 tiling, it does not help much for 1x2 rectangular tiling. In fact, for 1x4 rectanular tiling, the register usage by one block of data prefetching kernel exceeded the total number of registers in the SM. This makes the code not executable in G80! This is a good illustration that as one applies multiple techniques to a kernel, these techniques will likely interact by reducing the resources available to other techniques.

Finally, the appropriate combinations of performance tuning techniques can make a tremendous difference in the performance achieved by the matrix multiplication kernel. In Figure 5.15, the speed of the kernel executing on G80 increased from 18 GFOPS to 120 GLOPS! However, the programming efforts required to search through these combinations is currently quite large. Much work is being done in both academia and industry to reduce the amount of programming efforts needed to achieve these performance improvements with automation tools.