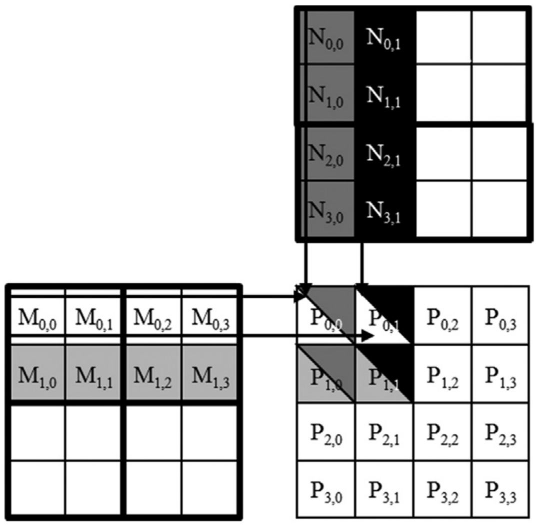
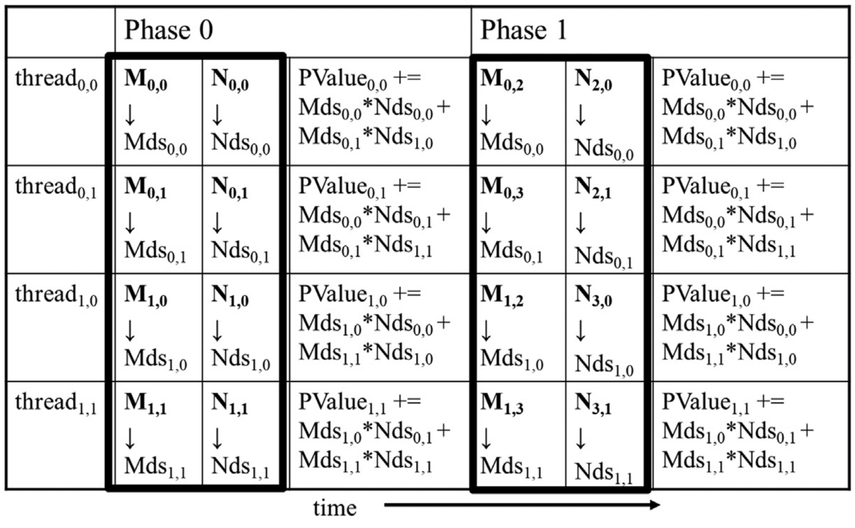


FIGURE 5.7

Tiling M and N to utilize shared memory.



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

FIGURE 5.8

Execution phases of a tiled matrix multiplication.

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 Fig. 5.8, and the second phase is shown in 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 reused across phases. In each phase, the same Mds and Nds are reused 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 because 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 multicore CPUs as in many-thread GPUs We will return to the concept of locality in Chapter 6, Performance Considerations.

5.4 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 Fig. 5.9 implements the phases illustrated in Fig. 5.8. In Fig. 5.9, lines 04 and 05 declare `Mds` and `Nds`, respectively, as shared memory arrays. Recall that the scope of shared memory variables is a block. Thus one version of the `Mds` and `Nds` arrays will be created for each block, and all threads of a block have access to the same `Mds` and `Nds` version. This is important because all threads in a block must have access to the M and N elements that are loaded into `Mds` and `Nds` by their peers so that they can use these values to satisfy their input needs.

Lines 07 and 08 save the `threadIdx` and `blockIdx` values into automatic variables with shorter names to make the code more concise. 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 runtime 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 11 and 12 determine the row index and column index, respectively, 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 12, 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 of P in the horizontal dimension. A thread in block `bx` would have before it `bx` blocks of threads, or `(bx*TILE_WIDTH)` threads; they cover

```
01    #define TILE_WIDTH 16
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07        int bx = blockIdx.x;  int by = blockIdx.y;
08        int tx = threadIdx.x; int ty = threadIdx.y;
09
10        // Identify the row and column of the P element to work on
11        int Row = by * TILE_WIDTH + ty;
12        int Col = bx * TILE_WIDTH + tx;
13
14        // Loop over the M and N tiles required to compute P element
15        float Pvalue = 0;
16        for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18            // Collaborative loading of M and N tiles into shared memory
19            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21            __syncthreads();
22
23            for (int k = 0; k < TILE_WIDTH; ++k) {
24                Pvalue += Mds[ty][k] * Nds[k][tx];
25            }
26            __syncthreads();
27
28        }
29        P[Row*Width + Col] = Pvalue;
30
31    }
```

FIGURE 5.9

A tiled matrix multiplication kernel using shared memory.

`bx*TILE_WIDTH` elements of P. Another `tx` thread 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`. For the example in Fig. 5.7, the horizontal (x) index of the P element to be calculated by $\text{thread}_{0,1}$ of $\text{block}_{1,0}$ is 0*2+1=1. This horizontal index is saved in the variable `Col` for the thread and is also illustrated in Fig. 5.10.

Similarly, the vertical (y) position, or the row index, of the P element to be processed by a thread is calculated as `by*TILE_WIDTH+ty`. Going back to the example in Fig. 5.7, the y index of the P element to be calculated by $\text{thread}_{0,1}$ of $\text{block}_{1,0}$ is 1*2+0=2. This vertical index is saved in the variable `Row` for the thread. As shown in Fig. 5.10, each thread calculates the P element at the `Col`th column and the `Row`th row. Thus the P element to be calculated by $\text{thread}_{0,1}$ of $\text{block}_{1,0}$ is $P_{2,1}$.

Line 16 of Fig. 5.9 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 Fig. 5.8. 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

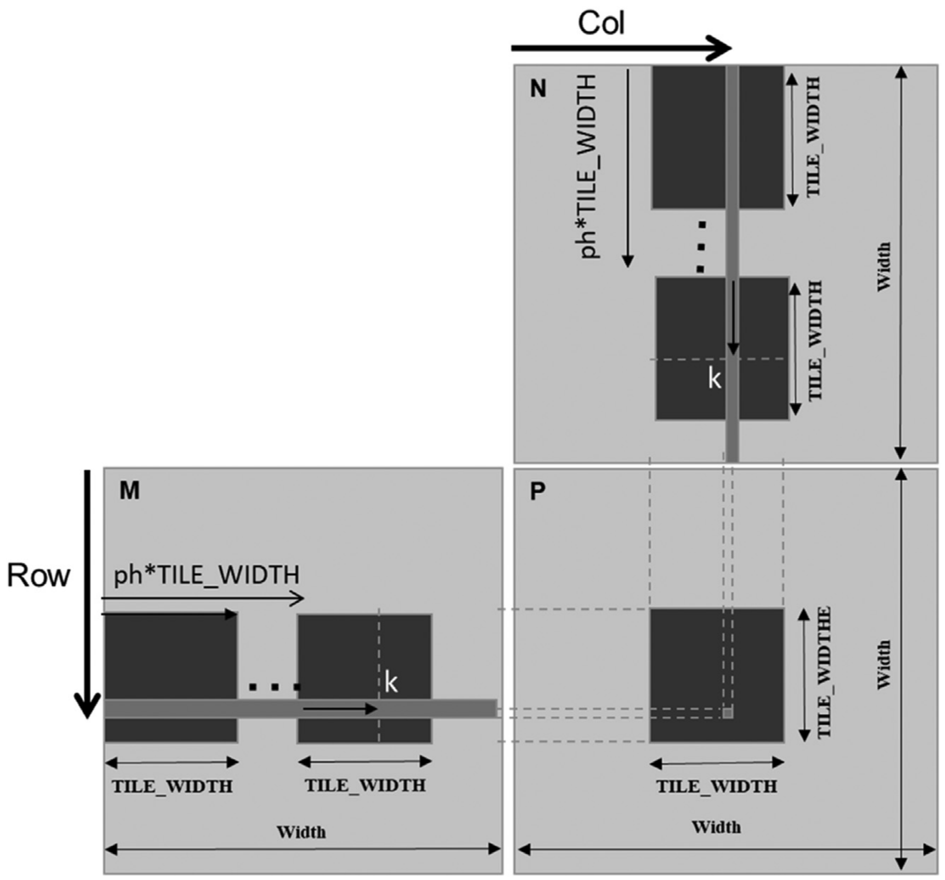


FIGURE 5.10

Calculation of the matrix indices in tiled multiplication.

of each phase, ph*TILE_WIDTH pairs of M and N elements have been processed by previous phases.

In each phase, lines 19 and 20 in Fig. 5.9 load the appropriate M and N elements, respectively, 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 Fig. 5.10, each block has TILE_WIDTH^2 threads that will collaborate to load TILE_WIDTH^2 M elements and TILE_WIDTH^2 N elements into the shared memory. Thus all we need to do is to assign each thread to load one M element and one N element. This is conveniently done by 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. Similarly, the beginning row index of the section of N elements to be loaded is also ph*TILE_WIDTH. Therefore every thread loads an element that is ty (the threadIdx.y value) positions away from that beginning point.

This is precisely what we have in lines 19 and 20. In line 19, 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 because each thread has a unique combination of `tx` and `ty`. Together, these threads will load a dark square subset of M in Fig. 5.10. In a similar way, in line 20, each thread loads the appropriate N element to shared memory using the linearized index `(ph*TILE_WIDTH + ty)*Width + Col`. The reader should use the small example in Figs. 5.7 and 5.8 to verify that the address calculation works correctly for individual threads.

The barrier `__syncthreads()` in line 21 ensures that all threads have finished loading the tiles of M and N into `Mds` and `Nds` before any of them can move forward. Recall from Chapter 4, Compute Architecture and Scheduling, that the call to `__syncthreads()` can be used to make all threads in a block wait for each other to reach the barrier before any of them can proceed. This is important because the M and N elements to be used by a thread can be loaded by other threads. One needs to ensure that all elements are properly loaded into the shared memory before any of the threads start to use the elements. The loop in line 23 then performs one phase of the dot product based on the tile elements. The progression of the loop for thread$_{ty,\ tx}$ is shown in Fig. 5.10, with the access direction of the M and N elements along the arrow marked with k, the loop variable in line 23. 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 26 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. Thus none of the threads would load the elements too early and corrupt the input values of other threads.

The two `__syncthreads()` calls in lines 21 and 26 demonstrate two different types of data dependence that parallel programmers often have to reason about when they are coordinating between threads. The first is called a *read-after-write dependence* because threads must wait for data to be written to the proper place by other threads before they try to read it. The second is called a *write-after-read dependence* because a thread must wait for the data to be read by all threads that need it before overwriting it. Other names for read-after-write and write-after-read dependences are true and false dependences, respectively. A read-after-write dependence is a *true dependence* because the reading thread truly needs the data supplied by the writing thread, so it has no choice but to wait for it. A write-after-read dependence is a *false dependence* because the writing thread does not need any data from the reading thread. The dependence is caused by the fact that they are reusing the same memory location and would not exist if they used different locations.

The loop nest from line 16 to line 28 illustrates a technique called *strip-mining*, which takes a long-running loop and break it into phases. Each phase involves an inner loop that executes a few 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 focus their work on the same section of input data during each phase. Strip-mining is an important means to creating the phases that are needed by tiling in data parallel programs.[3]

After all phases of the dot product are complete, the execution exits the outer loop. In Line 29, 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`. With 16×16 tiles, one can reduce the global memory accesses by a factor of 16. This increases the compute to global memory access ratio from 0.25 OP/B to 4 OP/B. This improvement allows the memory bandwidth of a CUDA device to support a higher computation rate. For example, in the A100 GPU which has a global memory bandwidth of 1555 GB/second, this improvement allows the device to achieve (1555 GB/second)*(4 OP/B)=6220 GFLOPS, which is substantially higher than the 389 GFLOPS achieved by the kernel that did not use tiling.

Although tiling improves throughput substantially, 6220 GFLOPS is still only 32% of the device's peak throughput of 19,500 GFLOPS. One can further optimize the code to reduce the number of global memory accesses and improve throughput. We will see some of these optimizations later in the book, while other advanced optimizations will not be covered. Because of the importance of matrix multiplication in many domains, there are highly optimized libraries, such as cuBLAS and CUTLASS, that already incorporate many of these advanced optimizations. Programmers can use these libraries to immediately achieve close to peak performance in their linear algebra applications.

The effectiveness of tiling at improving the throughput of matrix multiplication in particular and applications in general is not unique to GPUs. There is a long history of applying tiling (or blocking) techniques to improve performance on CPUs by ensuring that the data that is reused by a CPU thread within a particular time window will be found in the cache. One key difference is that tiling techniques on CPUs rely on the CPU cache to keep reused data on-chip implicitly, whereas tiling techniques on GPUs use shared memory explicitly to keep the data on-chip. The reason is that a CPU core typically runs one or two threads at a time, so a thread can rely on the cache keeping recently used data around. In contrast, a GPU SM runs many threads simultaneously to be able to hide latency. These threads may compete for cache slots, which makes the GPU cache less reliable, necessitating the use of shared memory for important data that is to be reused.

[3]The 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.

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 is assumed to be a multiple of the width of thread blocks. This prevents the kernel from correctly processing matrices with arbitrary width. 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.

5.5 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 a multiple of the tile width. Let's change the small example in Fig. 5.7 to use 3 × 3 M, N, and P matrices. The revised example is shown in Fig. 5.11. Note that the width of the matrices is 3, which is not a multiple of the tile width (which is 2). Fig. 5.11 shows the memory access pattern during the second phase of $\text{block}_{0,0}$. We see that $\text{thread}_{0,1}$ and $\text{thread}_{1,1}$ will attempt to load M elements that do not exist. Similarly, we see that $\text{thread}_{1,0}$ and $\text{thread}_{1,1}$ will attempt to access N elements that do not exist.

Accessing nonexisting elements is problematic in two ways. In the case of accessing a nonexisting element that is past the end of a row (M accesses by $\text{thread}_{0,1}$ and $\text{thread}_{1,1}$ in Fig. 5.11), 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? To answer this question, we need to go back to the linearized layout of two-dimensional matrices.

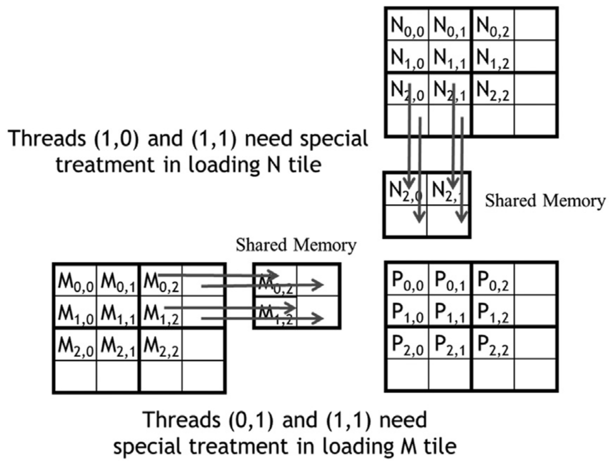

$N_{0,0}$ $N_{0,1}$ $N_{0,2}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$
Threads (1,0) and (1,1) need special treatment in loading N tile
$N_{2,0}$ $N_{2,1}$
Shared Memory
Shared Memory
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$
$M_{0,2}$
$M_{1,2}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$
Threads (0,1) and (1,1) need special treatment in loading M tile

FIGURE 5.11

Loading input matrix elements that are close to the edge: phase 1 of $\text{block}_{0,0}$.

The element after $M_{0,2}$ in the linearized layout is $M_{1,0}$. Although thead$_{0,1}$ is attempting to access $M_{0,3}$, 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 in accessing an element that is past the end of a column (N accesses by thread$_{1,0}$ and thread$_{1,1}$ in Fig. 5.11). 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. In other systems these accesses will be rejected, causing 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 arise only 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. Fig. 5.12 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 nonexisting M elements $M_{3,0}$ and $M_{3,1}$, whereas thread$_{0,1}$ and thread$_{1,1}$ attempt to access $N_{0,3}$ and $N_{1,3}$, which do not exist.

Note that these problematic accesses cannot be prevented by simply 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 for other threads in block$_{1,1}$ to use. Furthermore, 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 saw in Fig. 5.11, thread$_{0,1}$ of block 0,0 calculates a valid P element $P_{0,1}$. However, it attempts to access a nonexisting $M_{0,3}$ 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/

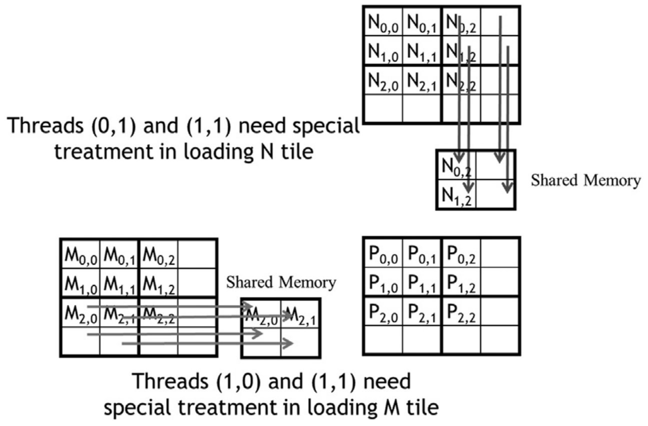


FIGURE 5.12

Loading input elements during phase 0 of block$_{1,1}$.

storing P elements. A rule of thumb to follow is that every memory access needs to have a corresponding check that ensures that the indices used in the access are within the bounds of the array being accessed.

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, in line 19 in Fig. 5.9, 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 store its final inner product value only 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 Fig. 5.13.

With the boundary condition checks, the tile matrix multiplication kernel is just one step away from being a general matrix multiplication kernel. In general,

```
    // Loop over the M and N tiles required to compute P element
    float Pvalue = 0;
    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];
        else Mds[ty][tx] = 0.0f;
        if ((ph*TILE_WIDTH+ty) < Width && Col < Width)
            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
        else Nds[ty][tx] = 0.0f;
        __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 5.13

Tiled matrix multiplication kernel with boundary condition checks.

matrix multiplication is defined for rectangular matrices: a j × k M matrix multiplied with a k × l N matrix results in a j × l P matrix. Our kernel can handle only square matrices so far.

Fortunately, it is quite easy to extend our kernel further 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.

5.6 Impact of memory usage on occupancy

Recall that in Chapter 4, Compute Architecture and Scheduling, we discussed the importance of maximizing the occupancy of threads on SMs to be able to tolerate long latency operations. The memory usage of a kernel plays an important role in occupancy tuning. While CUDA registers and shared memory can be extremely effective at reducing the number of accesses to global memory, one must be careful to stay within the SM's capacity of these memories. Each CUDA device offers limited 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 that can reside in each SM.

We saw in Chapter 4, Compute Architecture and Scheduling, how register usage can be a limiting factor for occupancy. Shared memory usage can also limit the number of threads that can be assigned to each SM. For example, the A100 GPU can be configured to have up to 164 KB of shared memory per SM and supports a maximum of 2048 threads per SM. Thus for all 2048 thread slots to be used, a thread block should not use more than an average of (164 KB)/(2048 threads)=82 B/thread. In the tiled matrix multiplication example, every block has `TILE_WIDTH`^{2} threads, and uses `TILE_WIDTH`^{2}*4B of shared memory for `Mds` and `TILE_WIDTH`^{2}*4B of shared memory for `Nds`. Thus the thread block uses an average of (`TILE_WIDTH`^{2}*4B + `TILE_WIDTH`^{2}*4B)/(`TILE_WIDTH`^{2} threads)=8 B/thread of shared memory. Therefore the tiled matrix multiplication kernel's occupancy is not limited by the shared memory.

However, consider a kernel that has thread blocks that use 32 KB of shared memory, each of which has 256 threads. In this case, the kernel uses an average of (32 KB)/(256 threads)=132 B/thread of shared memory. With such shared memory usage, the kernel cannot achieve full occupancy. Each SM can host a maximum of only (164 KB)/(132 B/thread)=1272 threads. Therefore the maximum achievable occupancy of this kernel will be (1272 assigned threads)/(2048 maximum threads)=62%.

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

Unfortunately, the kernels in Figs. 5.9 and 5.13 do not support any dynamic adjustment of shared memory usage by the host code. The declarations that are used in Fig. 5.9 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. Since the code 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 by adding a `C extern` keyword in front of the shared memory declaration and omitting the size of the array in the declaration. Based on this style, the declarations for `Mds` and `Nds` need to be merged into one dynamically allocated array:

```
extern __shared__ Mds_Nds[];
```

Since there is only one merged array, we will also need to manually define where the `Mds` section of the array starts and where the `Nds` section starts. Note that the merged array is one-dimensional. We will need to access it by using a linearized index based on the vertical and horizontal indices.

At runtime, when we call the kernel, we can dynamically configure the amount of shared memory to be used for each block according to the device query result and supply that as a third configuration parameter to the kernel call. For example, the revised kernel could be launched with the following statements:

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

  matrixMulKernel<<<dimGrid,dimBlock,size>>>(Md, Nd, Pd,
Width, size/2, size/2);
```

where `size_t` is a built-in type for declaring a variable to hold the size information for dynamically allocated data structures. The size is expressed in number of bytes. In our matrix multiplication example, for a 16×16 tile, we have a size of $2 \times 16 \times 16 \times 4 = 2048$ bytes to accommodate both `Mds` and `Nds`. We have omitted the details of the calculation for setting the value of size at runtime and leave it as an exercise for the reader.

In Fig. 5.14 we show how one can modify the kernel code in Figs. 5.9 and 5.11 to use dynamically sized shared memory for the `Mds` and `Nds` arrays. It may also be useful to pass the sizes of each section of the array as arguments into the kernel function. In this example we added two arguments: The first argument is the size of the `Mds` section, and the second argument is the size of the `Nds` section, both in terms of bytes. Note that in the host code above, we passed size/2 as the values of these arguments, which is 1024 bytes. With the assignments in lines 06 and 07, the rest of the kernel code can use `Mds` and `Nds` as the base of the array and use a linearized index to access the `Mds` and `Nds` elements. For example, instead of using `Mds[ty][tx]`, one would use `Mds [ty*TILE_WIDTH+tx]`.

```
01    #define TILE_WIDTH 16
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width,
                                       unsigned Mdz_sz, unsigned Nds_sz) {
03
04       extern __shared__ char float Mds_Nds[];
05
06       float *Mds = (float *) Mds_Nds;
07       float *Nds = (float *) Mds_Nds + Mds_sz;
```

FIGURE 5.14

Tiled matrix multiplication kernel with dynamically sized shared memory usage.

5.7 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 a CUDA devices, one needs to strive for 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 provides access to 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 redesign 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 uses barrier synchronization to force 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.

However, it is important for CUDA programmers to be aware of the limited sizes of these special types of memory. Their capacities are implementation dependent. Once their capacities have been exceeded, they limit the number of threads that can be executing simultaneously in each SM and can negatively affect the GPU's computation throughput as well as its ability to tolerate latency. The ability to reason about hardware limitations when developing an application is a key aspect of parallel programming.

Although we introduced tiled algorithms in the context of CUDA C 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 to make effective use of high-speed memories in these systems. For example, in a multicore CPU system, data locality allows an application to effectively use on-chip data caches to reduce memory access latency and achieve high performance. These on-chip data caches are also of limited size and require the computation to exhibit locality. Therefore the reader will also find the tiled algorithm useful when developing a parallel application for other types of parallel computing systems using other programming models.

Our goal for this chapter was to introduce the concept of locality, tiling, and different CUDA memory types. We introduced a tiled matrix multiplication kernel using shared memory. We further studied the need for boundary test conditions to allow for arbitrary data dimensions in applying tiling techniques. We also briefly discussed the use of dynamically sized shared memory allocation so that the kernel can adjust the size of shared memory that is used by each block according to the hardware capability. We did not discuss the use of registers in tiling. We will explain the use of registers in tiled algorithms when we discuss parallel algorithm patterns in Part II of the book.