# Chapter 7

# Parallel Patterns: Convolution

An introduction to stencil computation

**Keywords:** convolution, stencil, tiling, ghost cells, halo cells, tiling efficiency, constant memory, cache, finite difference method

## CHAPTER OUTLINE

In the next several chapters, we will discuss a set of important patterns of parallel computation. These patterns are the basis of a wide range of parallel algorithms that appear in many parallel applications. We will start with convolution, which is a popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision. In these application areas, convolution is often performed as a filter that transforms signals and pixels into more desirable values. Our image blur kernel is such a filter that smooths out the signal values so that one can see the big-picture trend. For another example, Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images.

In high-performance computing, the convolution pattern is often referred to as stencil computation, which appears widely in numerical methods for solving differential equations. It also forms the basis of many force calculation algorithms in simulation models. Convolution typically involves a significant number of arithmetic operations on each data element. For large data sets such as high-definition images and videos, the amount of computation can be very large. Each output data element can be calculated independently of each other, a desirable trait for parallel computing. On the other hand, there is substantial level of input data sharing among output data elements with somewhat challenging boundary conditions. This makes convolution an important use case of sophisticated tiling methods and input data staging methods.

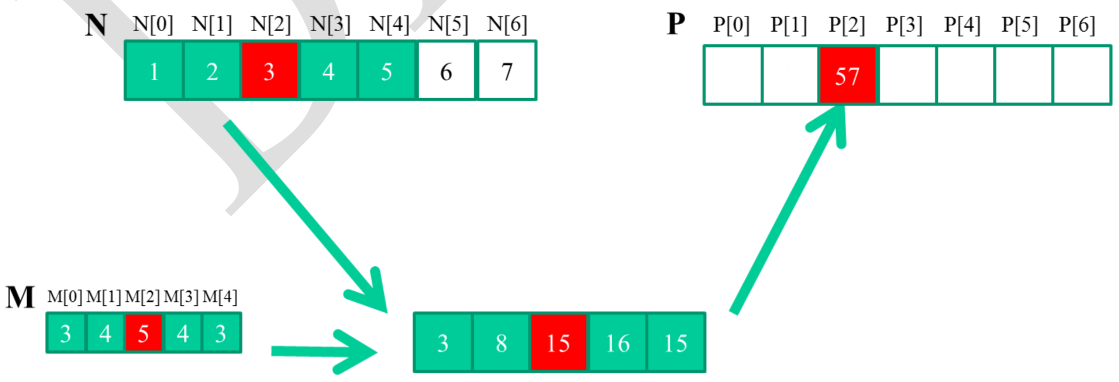
## 7.1. Background

Convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements. The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the *convolution kernel*. Since there is an unfortunate name conflict between the CUDA kernel functions and convolution kernels, we will refer to these mask arrays as *convolution masks* to avoid confusion. The same convolution mask is typically used for all elements of the array.

In audio digital signal processing, the input data are in 1D form and represent sampled signal volume as a function of time. Figure 7.1 shows a convolution example for 1D data where a 5-element convolution mask array M is applied to a 7-element input array N. We will follow the C language convention where N and P elements are indexed from 0 to 6 and M elements are indexed from 0 to 4. The fact that we use a 5-element mask M means that each P element is generated by a weighted sum of the N element at the corresponding position, two N elements to the left and two N elements to the right.

For example, the value of P[2] is generated as the weighted sum of N[0] (i.e., N[2-2]) through N[4] (i.e., N[2+2]). In this example, we arbitrarily assume that the values of the N elements are 1, 2, 3, …,7. The M elements define the weights, whose values are 3, 4, 5, 4, 3 in this example. Each weight value is multiplied to the corresponding N element values before the products are summed together. As shown in Figure 7.1, the calculation for P[2] is as follows:

```
P[2]    = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]
        = 1*3 + 2*4 + 3*5 + 4*4 + 5*3
        = 57
```



N
N[0] N[1] N[2] N[3] N[4] N[5] N[6]
1 2 3 4 5 6 7
P
P[0] P[1] P[2] P[3] P[4] P[5] P[6]
57
M
M[0] M[1] M[2] M[3] M[4]
3 4 5 4 3
3 8 15 16 15

*Figure 7.1 A 1D convolution example, inside elements.*

In general, the size of the mask tends to be an odd number, which makes the weighted sum calculation symmetric around the element being calculated. That is, an odd number of mask elements defines the weighted sum to include the same number of elements on each side of the element being calculated. In Figure 7.1, the mask size is 5 elements. Each output element is calculated as the weighted sum of the corresponding input element, two elements on the left, and two elements on the right.

In Figure 7.1, the calculation for P[i] can be viewed as an inner product between the sub array of N that starts at N[i-2] and the M array. Figure 7.2 shows the calculation for P[3]. The calculation is shifted by one N element from that of Figure 7.1. That is the value of P[3] is the weighted sum of N[1] (i.e., N[3-2]), through N[5] (i.e., 3+2). We can think of the calculation for P[3] is as follows:

```
P[3]  = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]
      = 2*3 + 3*4 + 4*5 + 5*4 + 6*3
      = 76
```

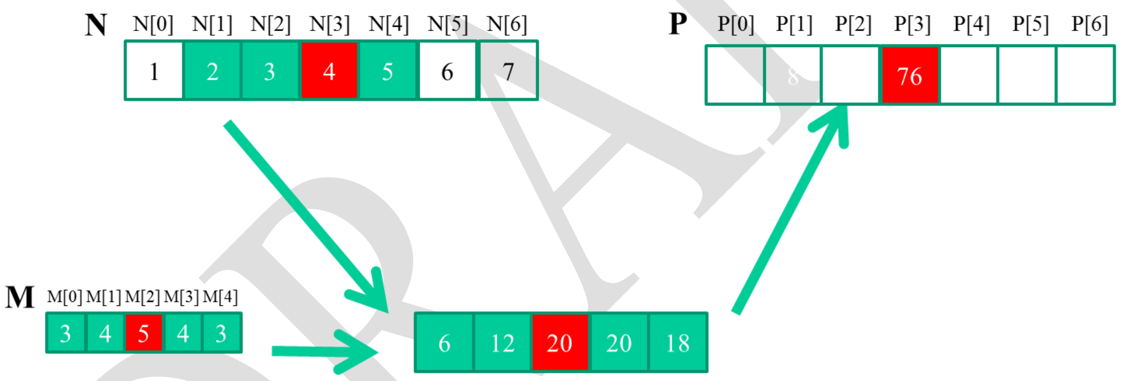


*Figure 7.2 1D convolution, calculation of P[3]*

Because convolution is defined in terms of neighboring elements, boundary conditions naturally arise for output elements that are close to the ends of an array. As shown in Figure 7.3, when we calculate P[1], there is only one N element to the left of N[1]. That is, there are not enough N elements to calculate P[1] according to our definition of convolution. A typical approach to handling such boundary condition is to define a default value to these missing N elements. For most applications, the default value is 0, which is what we used in Figure 7.3. For example, in audio signal processing, we can assume that the signal volume is 0 before the recording starts and after it ends. In this case, the calculation of P[1] is as follows.

```
P[1]  = 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4]
      = 0 * 3 + 1*4 + 2*5 + 3*4 + 4*3
      = 38
```

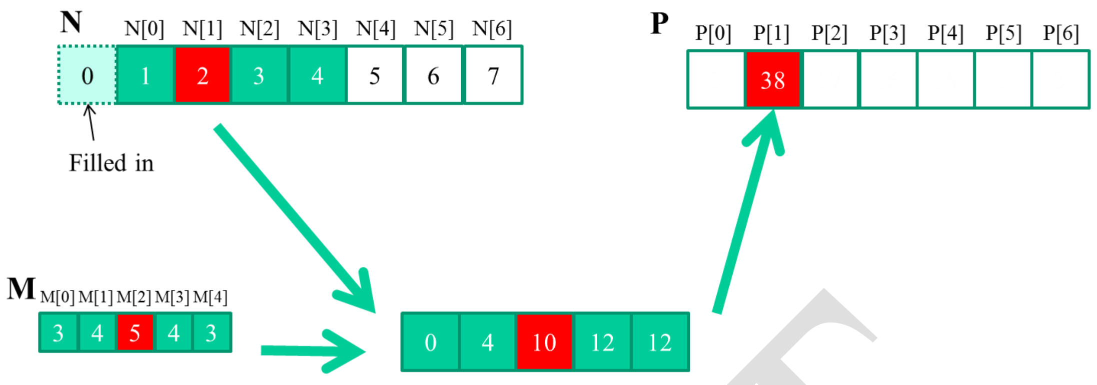


*Figure 7.3 1D convolution boundary condition.*

The N element that does not exist in this calculation is illustrated as a dashed box in Figure 7.3. It should be clear that the calculation of P[0] will involve two missing N elements, both will be assumed to be 0 for this example. We leave the calculation of P[0] as an exercise. These missing elements are typically referred to as "ghost cells" or "halo cells" in literature. There are also other types of ghost cells due to the use of tiling in parallel computation. These ghost cells can have significant impact on the effectiveness and/or efficiency of tiling. We will come back to this point soon.

Also, not all application assume that the ghost cells contain 0. For example, some applications might assume that the ghost cells contain the same value as the closest valid data element.

For image processing and computer vision, input data is typically two-dimensional arrays, with pixels in an x-y space. Image convolutions are therefore 2D convolution, as illustrated in Figure 7.4. In a 2D convolution, the mask M is a 2D array. Its x- and y-dimensions determine the range of neighbors to be included in the weighted sum calculation. In Figure 7.4, we use a 5×5 mask for simplicity. In general, the mask does not have to be a square array. To generate an output element, we take the sub-array whose center is at the corresponding location in the input array N. We then perform pairwise multiplication between elements of the mask array and those of the image array. For our example, the result is shown as the 5×5 product array below N and P in Figure 7.4. The value of the output element is the sum of all elements of the product array.

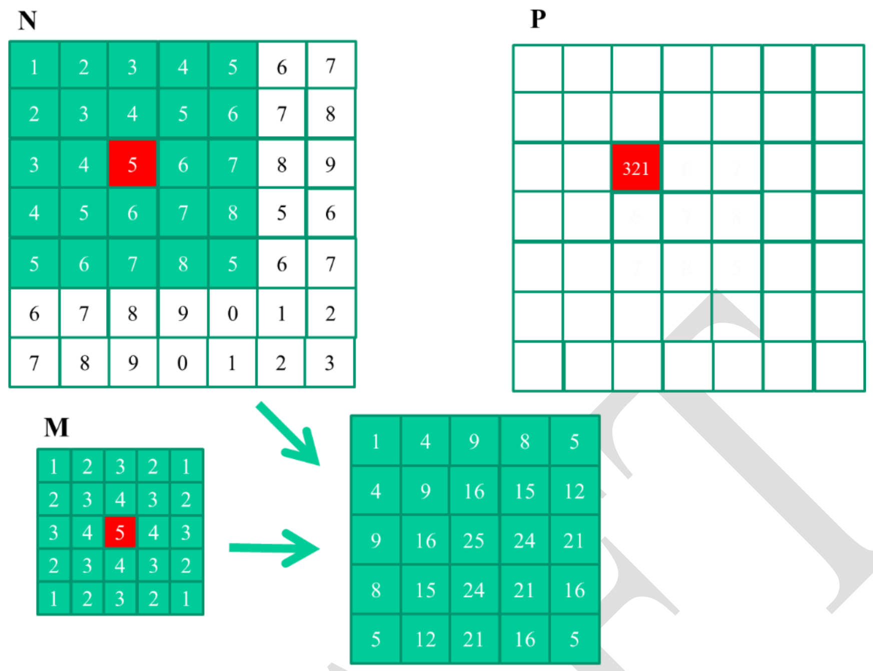


*Figure 7.4 A 2D convolution example.*

The example in Figure 7.4 shows the calculation of $P_{2,2}$. For brevity, we will use $N_{y,x}$ to denote N[y][x] in addressing a C array. Since N and P are most likely dynamically allocated arrays, we will be using linearized indices in our actual code examples. The sub-array of N for calculating the value of $P_{2,2}$ span from $N_{0,0}$ to $N_{0,4}$ in the x or horizontal direction and $N_{0,0}$ to $N_{4,0}$ in the y or vertical direction. The calculation is as follows:

$$\begin{aligned}
P_{2,2} &= N_{0,0}*M_{0,0} + N_{0,1}*M_{0,1} + N_{0,2}*M_{0,2} + N_{0,3}*M_{0,3} + N_{0,4}*M_{0,4} \\
&\quad + N_{1,0}*M_{1,0} + N_{1,1}*M_{1,1} + N_{1,2}*M_{1,2} + N_{1,3}*M_{1,3} + N_{1,4}*M_{1,4} \\
&\quad + N_{2,0}*M_{2,0} + N_{2,1}*M_{1,1} + N_{2,2}*M_{2,2} + N_{2,3}*M_{2,3} + N_{2,4}*M_{2,4} \\
&\quad + N_{3,0}*M_{3,0} + N_{3,1}*M_{3,1} + N_{3,2}*M_{3,2} + N_{3,3}*M_{3,3} + N_{3,4}*M_{3,4} \\
&\quad + N_{4,0}*M_{4,0} + N_{4,1}*M_{4,1} + N_{4,2}*M_{4,2} + N_{4,3}*M_{4,3} + N_{4,4}*M_{4,4} \\
&= 1*1 + 2*2 + 3*3 + 4*2 + 5*1 \\
&\quad + 2*2 + 3*3 + 4*4 + 5*3 + 6*2 \\
&\quad + 3*3 + 4*4 + 5*5 + 6*4 + 7*3 \\
&\quad + 4*2 + 5*3 + 6*4 + 7*3 + 8*2 \\
&\quad + 5*1 + 6*2 + 7*3 + 8*2 + 5*1 \\
&= 1 + 4 + 9 + 8 + 5 \\
&\quad + 4 + 9 + 16 + 15 + 12 \\
&\quad + 9 + 16 + 25 + 24 + 21 \\
&\quad + 8 + 15 + 24 + 21 + 16 \\
&\quad + 5 + 12 + 21 + 16 + 5 \\
&= 321
\end{aligned}$$

Like 1D convolution, 2D convolution must also deal with boundary conditions. With boundaries in both the x and y dimensions, there are more complex boundary conditions: the calculation of an output element may involve boundary conditions along a horizontal boundary, a vertical boundary, or both. Figure 7.5 illustrates the calculation of a P element that involves both boundaries. From Figure 7.5, the calculation of $P_{1,0}$ involves two missing columns and one missing horizontal row in the sub-array of N. Like in 1D convolution, different applications assume different default values for these missing N elements. In our example, we assume that the default value is 0. These boundary conditions also affect the efficiency of tiling. We will come back to this point soon.

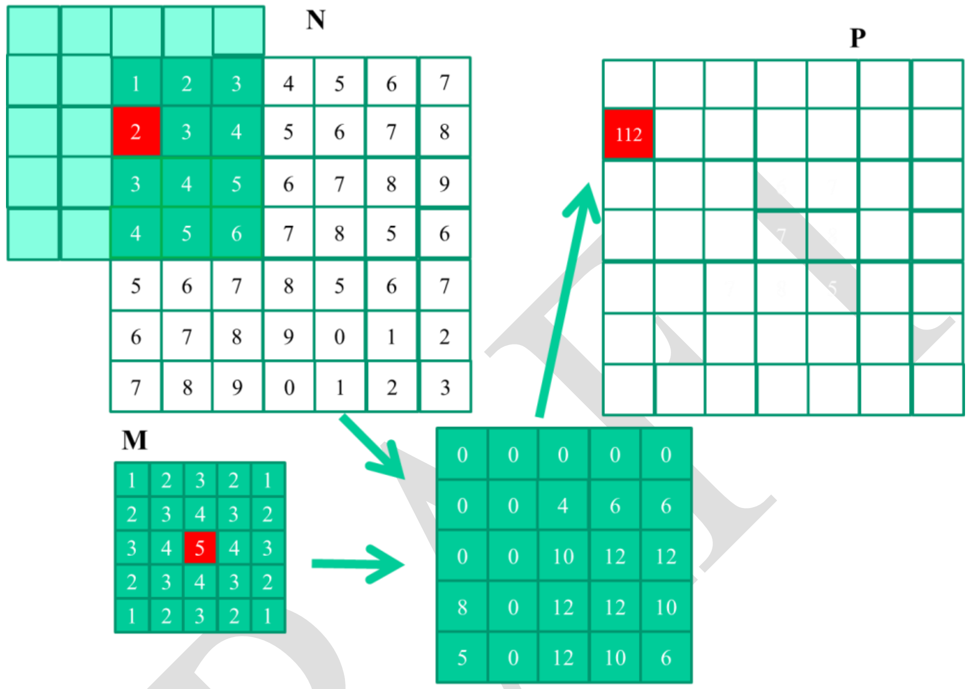


*Figure 7.5 A 2D convolution boundary condition.*

## 7.2. 1D Parallel Convolution – A Basic Algorithm

As we mentioned in Section 7.1, the calculation of all output (P) elements can be done in parallel in a convolution. This makes convolution an ideal problem for parallel computing. Based on our experience in matrix-matrix multiplication, we can quickly write a simple parallel convolution kernel. For simplicity, we will work on 1D convolution.

The first step is to define the major input parameters for the kernel. We assume that the 1D convolution kernel receives five arguments: pointer to input array N, pointer to input mask M, pointer to output array P, size of the mask Mask_Width, and size of the input and output arrays Width. Thus, we have the following set up.

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {
  // kernel body
}
```

The second step is to determine and implement the mapping of threads to output elements. Since the output array is 1D, a simple and good approach is to organize the threads into a 1D grid and have each thread in the grid to calculate one output element. The reader should recognize that this is the same arrangement as the vector addition example as far as output elements are concerned. Therefore, we can use the following statement to calculate an output element index from the block index, block dimension, and thread index for each thread:

```
    int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Once we determined the output element index, we can access the input N elements and the mask M elements using offsets to the output element index. For simplicity, we assume that Mask_Width is an odd number and the convolution is symmetric, i.e., Mask_Width is 2*n+1 where n is an integer. The calculation of P[i] will use N[i-n], N[i-n+1],…, N[i-1], N[i], N[i+1], N[i+n-1], N[i+n]. We can use a simple loop to do this calculation in the kernel:

```
    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
      if (N_start_point + j >= 0 && N_start_point + j < Width) {
        Pvalue += N[N_start_point + j]*M[j];
      }
    }
    P[i] = Pvalue;
```

The variable Pvalue will allow all intermediate results be accumulated in a register to save DRAM bandwidth. The for loop accumulates all the contributions from the neighboring elements to the output P element. The if statement in the loop tests in any of the input N elements used are ghost cells, either on the left side or the right side of the N array. Since we assume that 0 values will be used for ghost cells, we can simply skip the multiplication and accumulation of the ghost cell element and its corresponding N element. After the end of the loop, we release the Pvalue into the output P element. We now have a simple kernel in Figure 7.6.

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;
}
```

*Figure 7.6 A 1D convolution kernel with boundary condition handling.*

We can make two observations about the kernel in Figure 7.6. First, there will be control flow divergence. The threads that calculate the output P elements near the left end or the right end of the P array will handle ghost cells. As we showed in Section 7.1, each of these neighboring threads will encounter a different number of ghost cells. Therefore, they will all be somewhat different decisions in the if statement. The thread that calculates P[0] will skip the multiply-accumulate statement about half of the time whereas the one that calculates P[1] will skip one fewer times, and so on. The cost of control divergence will depend on Width the size of the input array and Mask_Width the size of the masks. For large input arrays and small masks, the control divergence only occurs to a small portion of the output elements, which will keep the effect of control divergence small. Since convolution is often applied to large images and spatial data, we typically expect that the effect of convergence to be modest or insignificant.

A more serious problem is memory bandwidth. The ratio of floating-point arithmetic calculation to global memory accesses is only about 1.0 in the kernel. As we have seen in the matrix-matrix multiplication example, this simple kernel can only be expected to run at a small fraction of the peak performance. We will discuss two key techniques for reducing the number of global memory accesses in the next two sections.

## 7.3. Constant Memory and Caching

There are three interesting properties of the way the mask array M is used in convolution. First, the size of the M array is typically small. Most convolution masks are less than 10 elements in each dimension. Even in the case of a 3D convolution, the mask typically contains only less than 1000 elements. Second, the contents of M are not changed throughout the execution of the kernel. Third, all threads need to access the mask elements. Even better, all threads access the M elements in the same order, starting from M[0] and move by one element a time through the iterations of the for loop in Figure 7.6. These two properties make the mask array an excellent candidate for constant memory and caching.

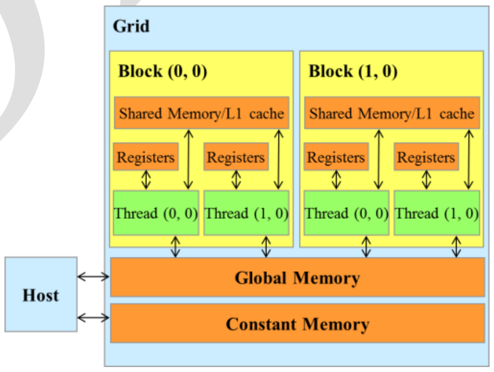


*Figure 7.7 A review of the CUDA Memory Model*

As we discussed in Chapter 5, the CUDA programming model allows programmers to declare a variable in the constant memory. Like global memory variables, constant memory variable are also visible to all thread blocks. The main difference is that a constant memory variable cannot be changed by threads during kernel execution. Furthermore, the size of the constant memory is quite small, currently at 64KB.

In order to use constant memory, the host code needs to allocate and copy constant memory variables in a different way than global memory variables. To declare an M array in constant memory, the host code declares it as a global variable as follows:

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

This is a global variable declaration and should be outside any function in the source file. The keyword **__constant__** (two underscores on each side) tells the compiler that array M should be placed into the device constant memory.

Assume that the host code has already allocated and initialized the mask in a mask h_M array in the host memory with Mask_Width elements. The contents of the h_M can be transferred to M in the device constant memory as follows:

```
  cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

Note that this is a special memory copy function that informs the CUDA runtime that the data being copied into the constant memory will not be changed during kernel execution. In general, the use of cudaMemcpyToSymble() function is as follows:

```
cudaMemcpyToSymbol(dest, src, size)
```

where dest is a pointer to the destination location in the constant memory, src is a pointer to the source data in the host memory, and size is the number of bytes to be copied.

Kernel functions access constant memory variables as global variables. Thus, their pointers do not need to be passed to the kernel as parameters. We can revise our kernel to use the constant memory as shown in Figure 7.8. Note that the kernel looks almost identical to that in Figure 7.6. The only difference is that M is no longer accessed through a pointer passed in as a parameter. It is now accessed as a global variable declared by the host code. Keep in mind that all the C language scoping rules for global variables apply here. If the host code and kernel code are in different files, the kernel code file must include the relevant external declaration information to ensure that the declaration of M is visible to the kernel.

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int
Mask_Width,
  int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;
```

*Figure 7.8 A 1D convolution kernel using constant memory for M.*

Like global memory variables, constant memory variables are also located in DRAM. However, because the CUDA runtime knows that constant memory variables are not modified during kernel execution, it directs the hardware to aggressively cache the constant memory variables during kernel execution.  In order to understand the benefit of constant memory usage, we need to first understand more about modern processor memory and cache hierarchies.

In virtually all modern processors, accessing a variable from DRAM takes hundreds if not thousands of clock cycles. Also, the rate at which variables can be accessed from DRAM is typically much lower than the rate at which processors can perform arithmetic operation. The long latency and limited bandwidth of DRAM has been a major bottleneck in virtually all modern processors commonly referred to as the Memory Wall. In order to mitigate the effect of memory bottleneck, modern processors commonly employ on-chip cache memories, or caches, to reduce the number of variables that need to be accessed from DRAM.

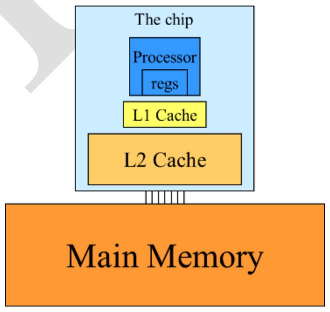


*Figure 7.9 A simplified view of the cache hierarchy of modern processors.*

Unlike CUDA shared memory, or scratch memories in general, caches are “transparent” to programs. That is, in order to use CUDA shared memory, a program needs to declare variables as __shared__ and explicitly move a global memory variable into a shared memory variable. On the other hand, when using caches, the program simply accesses the original variables. The processor hardware will automatically retain some of the most recently or frequently used variables in the cache and remember their original DRAM address. When one of the retained variables is used later, the hardware will detect from their addresses that a copy of the variable is available in cache. The value of the variable will then be provided from the cache, eliminating the need to access DRAM.

There is a tradeoff between the size of a memory and the speed of a memory. As a result, modern processors often employ multiple levels of caches. The numbering convention for these cache levels reflects the distance to the processor. The lowest level, L1 or Level 1, is the cache that is directly attached to a processor core. It runs at a speed very close to the processor in both latency and bandwidth. However, an L1 cache is small in size, typically between 16KB and 64KB. L2 caches are larger, in the range of 128KB to 1MB, but can take tens of cycles to access. They are typically shared among multiple processor cores, or SMs in a CUDA device. In some high-end processors today, there are even L3 caches that can be of several MB in size.

A major design issue with using caches in a massively parallel processor is cache coherence, which arises when one or more processor cores modify cached data. Since L1 caches are typically directly attached to only one of the processor cores, changes in its contents are not easily observed by other processor cores. This causes a problem if the modified variable is shared among threads running on different processor cores. A *cache coherence mechanism* is needed to ensure that the contents of the caches of the other processor cores are updated. Cache coherence is difficult and expensive to provide in massively parallel processors. However, their presence typically simplifies parallel software development. Therefore, modern CPUs typically support cache coherence among processor cores. While modern GPUs provide two levels of caches, they typically do without cache coherence to maximize hardware resources available to increase the arithmetic throughput of the processor.

Constant memory variables play an interesting role in using caches in massively parallel processors. Since they are not changed during kernel execution, there is no cache coherence issue during the execution of a kernel. Therefore, the hardware can aggressively cache the constant variable values in L1 caches. Furthermore, the design of caches in these processors is typically optimized to broadcast a value to a large number of threads. As a result, when all threads in a warp access the same constant memory variable, as is the case of M, the caches can provide tremendous amount of bandwidth to satisfy the data needs of threads. Also, since the size of M is typically small, we can assume that all M elements are effectively always accessed from caches. Therefore, we can simply assume that no DRAM bandwidth is spent on M accesses. With the use of DRAM, we have effectively doubled the ratio of floating-point arithmetic to memory access to 2.

As it turns out, the accesses to the input N array elements can also benefit from caching in more recent GPUs. We will come back to this point in Section 7.5.

## 7.4. Tiled 1D Convolution with Halo Cells

We will now address the memory bandwidth issue in accessing N array element with a tiled convolution algorithm. Recall that in a tiled algorithm, threads collaborate to load input elements into an on-chip memory and then access the on-chip memory for their subsequent use of these elements. For simplicity, we will continue to assume that each thread calculates one output P element. With up to 1024 threads in a block we can process up to 1024 data elements. We will refer to the collection of output elements processed by each block as an *output tile*. Figure 7.10 shows a small example of 16-element 1D convolution using four thread blocks of four threads each. In this example, there are four output tiles. The first output tile covers N[0] through N[3], the second tile N[4] through N[7], the third tile N[8] through N[11], and the fourth tile N[12] through N[15]. Keep in mind that we use four threads per block to keep the example small. In practice, there should be at least 32 threads per block for the current generation of hardware. From this point on, we will assume that M elements are in the constant memory.

We will discuss two input data tiling strategy for reducing the total number of global memory accesses. The first one is the most intuitive and involves loading all input data elements needed for calculating all output elements of a thread block into the shared memory. The number of input elements to be loaded depends on the size of the mask. For simplicity, we will continue to assume that the mask size is an odd number equal to 2*n+1. That is each output element P[i] is a weighted sum of the input element at the corresponding input element N[i], the n input elements to the left (N[i-n], … N[i-1]), and the n input elements to the right (N[i+1], … N[i+n]). Figure 7.10 shows an example where K=5 and n=2.

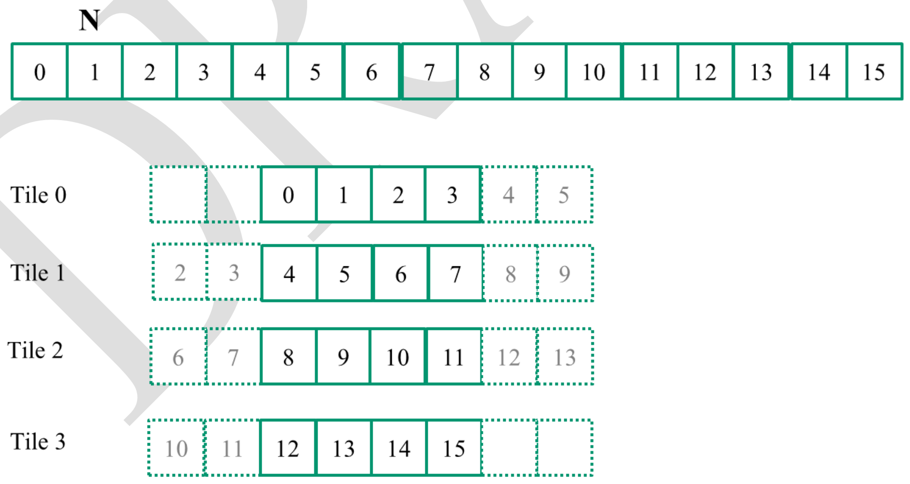


*Figure 7.10 A 1D tiled convolution example.*

Threads in the Block 0 calculate output elements P[0] through P[3]. They collectively require input elements N[0] through N[5]. Note that the calculation also requires two ghost cell elements to the left of N[0]. This is shown as two dashed empty elements on the left end of

Tile 0 of Figure 7.6. These ghost elements will be assumed have default value of 0. Tile 3 has a similar situation at the right end of input array N. In our discussions, we will refer to tiles like Tile 0 and Tile 3 as boundary tiles since they involve elements at or outside the boundary of the input array N.

Threads in Block 1 calculate output elements P[4] through P[7]. They collectively require input elements N[2] through N[9], also shown in Figure 7. Note that elements N[2] and N[3] belong to two tiles and are loaded into the shared memory twice, once to the shared memory of Block 0 and once to the shared memory of Block 1. Since the contents of shared memory of a block are only visible to the threads of the block, these elements need to be loaded into the respective shared memories for all involved threads to access them. The elements that are involved in multiple tiles and loaded by multiple blocks are commonly referred to as *halo cells* or *skirt cells* since they "hang" from the side of the part that is used solely by a single block. We will refer to the center part of an input tile that solely by a single block the *internal cells* of that input tile. Tile 1 and Tile 2 are commonly referred to as *internal tiles* since they do not involve any ghost elements at our outside the boundaries of the input array N.

We now show the kernel code that loads the input tile into shared memory. We first declare a shared memory array N_ds to hole the N tile for each block. The size of the shared memory array must be large enough to hold the left halo cells, the center cells, and the right halo cells of an input tile. We assume that Mask_Size is an odd number. The total is TILE_SIZE + MAX_MASK_WIDTH -1, which is used in the following declaration in the kernel.

```
 __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
```

We then load the left halo cells, which include the last n = Mask_Width/2 center elements of the previous tile. For example, in Figure 7.6, the left halo cells of Tile 1 consist of the last 2 center elements of Tile 0. In C, assuming that Mask_Width is an odd number, the expression Mask_Width/2 will result in an integer value that is the same as (Mask_Wdith-1)/2. We will use the last (Mask_Width/2) threads of the block to load the left halo element. This is done with the following two statements.

```
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
if (threadIdx.x >= blockDim.x - n) {
  N_ds[threadIdx.x - (blockDim.x - n)] =
    (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

In the first statement, we map the thread index to element index into the previous tile with the expression (blockIdx.x-1)*blockDim.x+threadIdx.x. We then pick only the last n threads to load the needed left halo elements using the condition in the if statement. For example, in Figure 7.6, blockDim.x equals 4 and n equals 2; only thread 2 and thread 3 will be used. Thread 0 and thread 1 will not load anything due to the failed condition.

For the threads used, we also need to check if their halo cells are actually ghost cells. This can be checked by testing if the calculated halo_index_left value is negative. If so, the halo cells are actually ghost cells since their N indices are negative, outside the valid range of the N indices. The conditional C assignment will choose 0 for threads in this situation. Otherwise, the conditional statement will use the halo_index_left to load the appropriate N elements into the shared memory. The shared memory index calculation is such that left halo cells will be loaded into the shared memory array starting at element 0. For example, in Figure 7.6, blockDim.x-n equals 2. So for block 1, thread 2 will load the left most halo element into N_ds[0] and thread 3 will load the next halo element into N_ds[1]. However, for block 0, both thread 2 and thread 3 will load value 0 into N_ds[0] and N_ds[1].

The next step is to load the center cells of the input tile. This is done by mapping the blockIdx.x and threadIdx.x values into the appropriate N indices, as shown in the following statement. The reader should be familiar with the N index expression used.

```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
```

Since the first n elements of the N_ds array already contain the left halo cells, the center elements need to be loaded into the next section of N_ds. This is done by n to threadIdx.x as the index for each thread to write its loaded center element into N_ds.

We now load the right halo elements, which is quite similar to loading the left halo. We first map the blockIdx.x and threadIdx.x to the elements of next output tile. This is done by adding (blockIdx.x+1)*blockDim.x to the thread index to form the N index for the right halo cells. In this case, we are loading the beginning Mask_Width

```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
if (threadIdx.x < n) {
  N_ds[n + blockDim.x + threadIdx.x] =
    (halo_index_right >= Width) ? 0 : N[halo_index_right];
}
```

Now that all the input tile elements are in N_ds, each thread can calculate their output P element value using the N_ds elements. Each thread will use a different section of the N_ds. Thread 0 will use N_ds[0] through N_ds[Mask_Width-1]; thread 1 will use N_ds[1] through N[Mask_Width]. In general, each thread will use N_ds[threadIdx.x] through N[threadIdx.x+Mask_Width-1]. This is implemented in the following for loop for calculate the P element assigned to the thread:

```
float Pvalue = 0;
for(int j = 0; j < Mask_Width; j++) {
  Pvalue += N_ds[threadIdx.x + j]*M[j];
}
P[i] = Pvalue;
```

However, one must not forget to do a barrier synchronization using synchtrheads() to make sure that all threads in the same block have completed loading their assigned N elements before anyone should start using them from the shared memory.

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int
Mask_Width,
  int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
  int n = Mask_Width/2;
  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }
  __syncthreads();
  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
```

*Figure 7.11 A tiled 1D convolution kernel using constant memory for M.*

Note that the code for multiply and accumulate is simpler than the base algorithm. The conditional statements for loading the left and right halo cells have placed the 0 values into the appropriate `N_ds` elements for the first and last thread block.

The tiled 1D convolution kernel is significantly longer and more complex than the basic kernel. We introduced the additional complexity in order to reduce the number of DRAM accesses for the N elements. The goal is improve the arithmetic to memory access ratio so that the achieved performance is not limited or less limited by the DRAM bandwidth. We will evaluate improvement by comparing the number of DRAM accesses performed by each thread block for the kernels in Figure 7.8 and Figure 7.11.

In Figure 7.8, there are two cases. For thread blocks that do not handle ghost cells, the number of N elements accessed by each thread is Mask_Wdith. Thus, the total number of N elements accessed by each thread block is blockDim.x*Mask_Width or blockDim.x*(2n+1). For example, if Mask_Width is equal to 5 and each block contains 1024 threads, each block access a total of 5120 N elements.

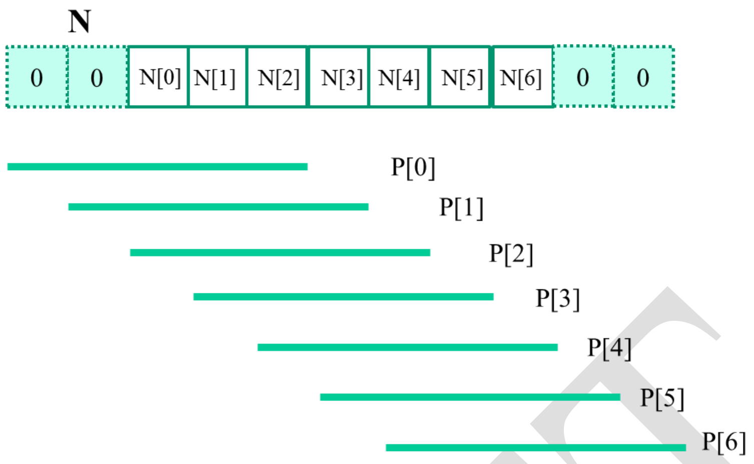


*Figure 7.12 A small example of accessing N elements and ghost cells.*

For the first and the last blocks, the threads that that handle ghost cells, no memory access is done for the ghost cells. This reduces the number of memory accesses. We can calculate the reduced number of memory accesses by enumerating the number of threads that use each ghost cell. This is illustrated with a small example in Figure 7.12. The leftmost ghost cell is used by one thread. The second left ghost cell is used by two threads. In general, the number of ghost cells is n and the number of threads that use each of these ghost cells, from left to right is 1, 2, … n. This is a simple series with sum n(n+1)/2, which is the total number of accesses that were avoided due to ghost cells. For our simple example where Mask_Width is equal to 5 and n is equal to 2, the number of accesses avoided due to ghost cells is 2*3/2 = 3. A similar analysis gives the same results for the right ghost cells. It should be clear that for large thread blocks, the effect of ghost cells for small mask sizes will be insignificant.

We now calculate the total number of memory accesses for N elements by the tiled kernel in Figure 7.11. All the memory accesses have been shifted to the code that loads the N elements into the shared memory. In the tiled kernel, each N element is only loaded by one thread. However, 2n halo cells will also be loaded, n from the left and n from the right, for blocks that do not handle ghost cells. Therefore, we have the blockDim.x+2n elements in for the internal thread blocks and blockDim+n for boundary thread blocks.

For internal thread blocks, the ratio of memory accesses between the basic and the tiled 1D convolution kernel is:

```
(blockDim.x*(2n+1)) / (blockDim.x+2n)
```

whereas the ratio for boundary blocks is:

```
(blockDim.x*(2n+1) – n(n+1)/2) / (blockDim.x+n)
```

For most situations, blockDim.x is much larger than n. Both ratios can be approximated by eliminating the small terms n(n+1)/2 and n:

```
(blockDim.x*(2n+1)/ blockDim.x = 2n+1 = Mask_Width
```

This should be quite an intuitive result. In the original algorithm, each N element is redundantly loaded by approximately Mask_Width threads. For example, in Figure 7.12, N[2] is loaded by the 5 threads that calculate P[2], P[3], P[4], P[5], and P[6]. That is, the ratio of memory access reduction is approximately proportional to the mask size.

However, in practice, the effect of the smaller terms may be significant and cannot be ignored. For example, if blockDim.x is 128 and n is 5, the ratio for the internal blocks is

```
(128*11 – 10) / (128 + 10) = 1398 / 138 = 10.13
```

whereas the approximate ratio would be 11. It should be clear that as the blockDim.x becomes smaller, the ratio also becomes smaller. For example, if blockDim is 32 and n is 5, the ratio for the internal blocks becomes

```
(32*11 – 10) / (32+10) = 8.14
```

The readers should always be careful when using smaller block and tile sizes. They may result in significantly less reduction in memory accesses than expected. In practice, smaller tile sizes are often used due to insufficient amount of on-chip memory, especially for 2D and 3D convolution where the amount of on-chip memory needed grow quickly with the dimension of the tile.

## 7.5. A Simpler Tiled 1D Convolution - General Caching

In Figure 7.11, much of the complexity of the code has to do with loading the left and right halo cells in addition to the internal elements into the shared memory. More recent GPUs such as Fermi provide general L1 and L2 caches, where L1 is private to each streaming multiprocessor and L2 is shared among all streaming multiprocessors. This leads to an opportunity for the blocks to take advantage of the fact that their halo cells may be available in the L2 cache.

Recall that the halo cells of a block are also internal cells of a neighboring block. For example, in Figure 7.10, the halo cells N[2] and N[3] of Tile 1 are also internal elements of Tile 0. There is a significant probability that by the time Block 1 needs to use these halo cells, they are already in L2 cache due to the accesses by Block 0. As a result, the memory accesses to these halo cells may be naturally served from L2 cache without causing additional DRAM traffic. That is, we can leave the accesses to these halo cells in the original N elements rather than loading them into the N_ds. We now present a simpler tiled 1D convolution algorithm that only loads the internal elements of each tile into the shared memory.

In the simpler tiled kernel, the shared memory N_ds array only needs to hold the internal elements of the tile. Thus, it is declared with the TILE_SIZE, rather than TILE_SIZE+Mask_Width-1.

```
    __shared__ float  N_ds[TILE_SIZE];
```

Loading the tile becomes very simple with only one line of code:

```
    N_ds[threadIdx.x] = N[blockIdx.x*blockDim.x+threadIdx.x];
```

We still need a barrier synchronization before using the elements in N_ds. The loop that calculates P elements, however, becomes more complex. It needs to add conditions to check for use of both halo cells and ghost cells.  The handling of ghost cells are done with the same conditional statement as that in Figure 7.6. The multiply-accumulate statement becomes more complex:

```
    __syncthreads();

    int This_tile_start_point = blockIdx.x * blockDim.x;
    int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
    int N_start_point = i - (Mask_Width/2);
    float Pvalue = 0;
    for (int j = 0; j < Mask_Width; j ++) {
      int N_index = N_start_point + j;
      if (N_index >= 0  && N_index < Width) {
        if ((N_index >= This_tile_start_point)
          && (N_index < Next_tile_start_point)) {
          Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
        } else {
          Pvalue += N[N_index] * M[j];
        }
      }
    }
    P[i] = Pvalue;
```

*Figure 7.13 Using general caching for halo cells.*

The variables This_tile_start_point and Next_tile_start_point hold the starting position index of the tile processed by the current block and that of the tile processed by the next in the next block. For example, in Figure 7.10, the value of This_tile_start_point for Block 1 is 4 and the value of Next_tile_start_point is 8.

The new if statement tests if the current access to the N element fulls within tile by testing it against This_tile_start_point and Next_tile_start_point. If the element falls within the tile, that is, it is an internal element for the current block, it is accssed from the N_ds array in the shared memory. Otherwise, it is accessed from the N array, which is hopefully in the L2 cache.

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int
Mask_Width,
  int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE];
  N_ds[threadIdx.x] = N[i];
  __syncthreads();
  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j ++) {
     int N_index = N_start_point + j;
     if (N_index >= 0  && N_index < Width) {
       if ((N_index >= This_tile_start_point)
         && (N_index < Next_tile_start_point)) {
         Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
       } else {
         Pvalue += N[N_index] * M[j];
       }
     }
  }
```

*Figure 7.14 A simpler tiled 1D convolution kernel using constant memory and general caching.*

## 7.6. Tiled 2D Convolution with Halo Cells

Now that we have learned how to tile a parallel 1D convolution computation, we can extend our knowledge to 2D quite easily. For a little more fun, we will use an example based on a class of 2D image format that is frequently encountered in image libraries and applications.

As we have seen in Chapter 3, real-world images are represented as 2D matrices and come in all sizes and shapes. Image processing libraries typically store these images in row-major layout when reading them from files into memory. If the width of the image in terms of bytes is not a multiple of the DRAM burst size, the starting point of row 1 and beyond can be misaligned from the DRAM burst boundaries. As we have seen in Chapter 5, such misalignment can result in poor utilization of the DRAM bandwidth when we attempt to access data in one of the rows. As a result, image libraries often also convert images into a padded format when reading them from files into memory, as illustrated in Figure 7.15.

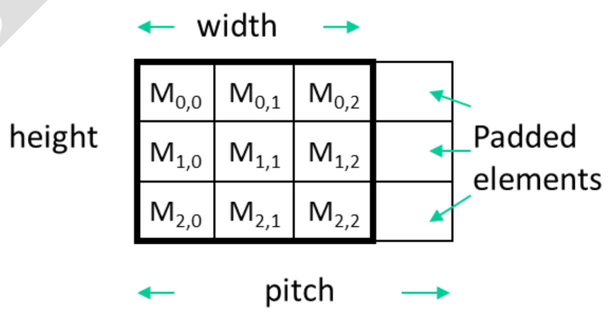

width
height
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$
Padded elements
pitch

*Figure 7.15 A padded image format and the concept of pitch.*

In Figure 7.15, we assume that the original image is 3x3. We future assume that each DRAM burst encompass 4 pixels. Without padding, $M_{1,0}$ in row 1 would reside in one DRAM burst unit whereas $M_{1,1}$ and $M_{1,2}$ would reside in the next DRAM burst unit. Accessing row 1 would require two DRAM bursts and wasting half of the memory bandwidth. To address this inefficiency, the library pads one element at the end of each row. With the padded elements, each row occupies an entire DRAM burst size. When we access row 1 or row 2, the entire row can now be accessed in one DRAM burst. In general, the images are much larger; each row can encompass multiple DRAM bursts. The padded elements will be added such that each row ends at the DRAM burst boundaries.

With padding, the image matrix has been enlarged by the padded elements. However, during computation such as image blur (Chapter 3), one should not process the padded elements. Therefore, the library data structure will indicate the original width and height of the image as shown in Figure 7.15. However, the library also has to provide the users with the information about the padded elements so that the user code can properly find the actual starting position of all the rows. This information is conveyed as the *pitch* of the padded matrix.

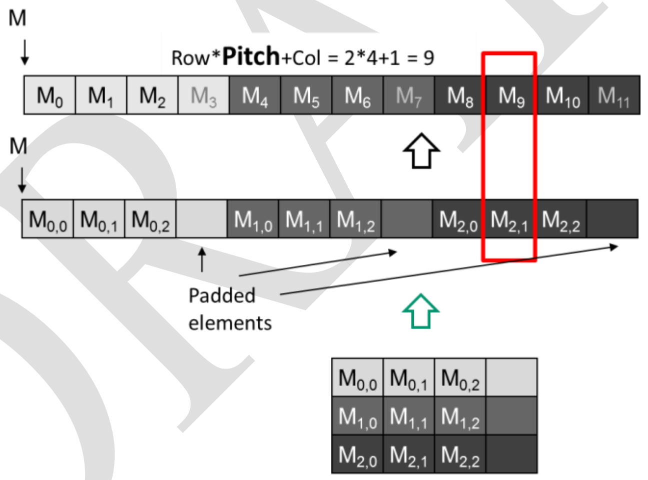

M
Row*Pitch+Col = 2*4+1 = 9
M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11
M
M0,0 M0,1 M0,2 M1,0 M1,1 M1,2 M2,0 M2,1 M2,2
Padded elements
M0,0 M0,1 M0,2
M1,0 M1,1 M1,2
M2,0 M2,1 M2,2

*Figure 7.16 Row-major layout of a 2D image matrix with padded elements*

Figure 7.16 shows how the image pixel elements can be accessed in the row-major layout of the padded image matrix. The lower layout shows the linearized order. Note that the padded elements are at the end of each row. The top layout shows the linearized 1D index of pixel elements in the padded matrix. As before, the three original elements, $M_{0,1}$, $M_{0,2}$, $M_{0,3}$ of row 0 become $M_0$, $M_1$, and $M_2$ in the linearized 1D array., Note that the padded elements become “dummy” linearized elements $M_3$, $M_7$, and $M_{11}$. The original elements of row 1, $M_{1,1}$, $M_{1,2}$, $M_{1,3}$, have their linearized 1D index as $M_4$, $M_5$, and $M_6$. That is, as shown in the top of Figure 7.16, to calculate the linearized 1D index of the pixel elements, we will use pitch instead of width in in the expression:

Linearized 1D index = row * pitch + column

However, when we iterate through a row, we will use width as the loop bound to ensure that we use only the original elements in a computation.

```
// Image Matrix Structure declaration
//
typedef struct {
   int width;
   int height;
   int pitch;
   int channels;
   float* data;
} * wbImage_t;
```

*Figure 7.17 The C type structure definition of the image pixel element.*

Figure 7.17 shows the image type that we will be using for the kernel code example. Note the channels field indicates the number of channels in the pixel: 3 for an RGB color image and 1 for a greyscale image as we have seen in Chapter 2. We assume that the value of these fields will be used as arguments when we invoke the 2D convolution kernel.

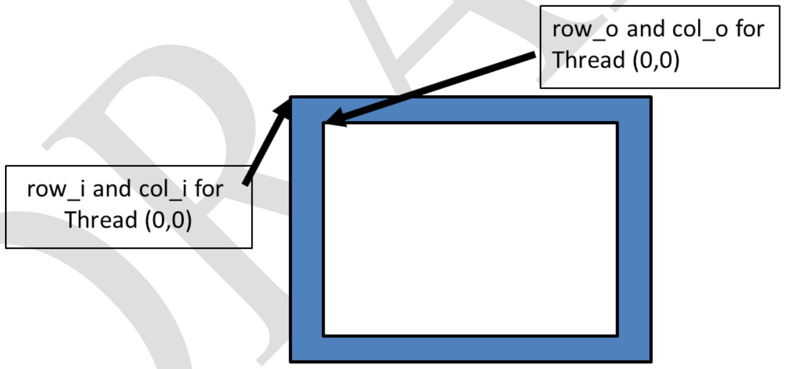

row_o and col_o for Thread (0,0)
row_i and col_i for Thread (0,0)

*Figure 7.18 starting element indices of the input tile vs. output tile.*

We are now ready to work on the design of a tiled 2D convolution kernel. In general, we will find that the design of the 2D convolution kernel is a straightforward extension of the 1D convolution kernel presented in Section 7.5. We need to first design the input and output tiles to be processed by each thread block, as shown in Figure 7.18. Note that the input tiles must include the halo cells and extend beyond their corresponding output tiles by the number of halo cells in each direction. Figure 7.19 shows the first part of the kernel:

```
__global__ void convolution_2D_kernel(float *P, float *N, int height, int width,
                           int pitch, int channels, int Mask_Width,
                           const float __restrict__ *M)
{

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row_o = blockIdx.y*O_TILE_WIDTH + ty;
  int col_o = blockIdx.x*O_TILE_WIDTH + tx;

  int row_i = row_o - Mask_Width/2 + 1;
  int col_i = col_o - Mask_Width/2 + 1;
```

*Figure 7.19 Part 1 of a 2D convolution kernel.*

Each thread of the kernel first calculates the y and x indices of its output element. These are the col_o and row_o variables of the kernel. The index values for thread$_{0,0}$ of the thread block (who is responsible for output the element at the upper left corner) is shown in Figure 7.18. Each thread then calculates the y and x indices of the input element it is to load into the shared memory by subtracting (Mask_Width/2+1) form row_o and col_o and assigning the results to row_i and col_i, also shown in Figure 7.18. Note that the input tile element to be loaded by thread$_{0,0}$ is also shown in Figure 7.18. To simply the tiling code over the kernel in Figure 7.14, we will configure each thread block to be of the same size as the input tile. In this design, we can simply have each thread to load one input N element. We will turn off some of the threads when we calculate the output since there are more threads in each block than the number of data element in each output tile.

```
if((row_i >= 0) && (row_i < height) &&
   (col_i >= 0)  && (col_i < width)) {
  Ns[ty][tx] = data[row_i * pitch + col_i];
} else{
  Ns[ty][tx] = 0.0f;
}
```

*Figure 7.20 Part 2 of a 2D convolution kernel.*

We are now ready to load the input tiles into the shared memory. All threads participate in this activity but each of them needs to check if the y and x indices of its input tile elements are within the valid range of the input. If not, the input element it is attempting to load is actually a ghost element and a 0.0 value should be placed into the shared memory. These threads belong in the thread blocks that calculate the image tiles that are close to the edge of the image. Note that we use the pitch value when we compute the linearized 1D index from the y and x index of the pixel. Also note that this code only works for the case where the number of channels is 1. In general, we should use a for-loop to load all the pixel channel values based on the number of channels present.

```
        float output = 0.0f;
        if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
           for(i = 0; i < MASK_WIDTH; i++) {
             for(j = 0; j < MASK_WIDTH; j++) {
               output += M[i][j] * Ns[i+ty][j+tx];
             }
           }

           if(row_o < height && col_o < width)
              data[row_o*width + col_o] = output;
           }
         }
```

*Figure 7.21 Part 3 of a 2D convolution kernel.*

The last part of the kernel, shown in Figure 7.21, computes the output value using the input elements in the shared memory. Keep in mind that we have more threads in the thread block than the number of pixels in the output tile. The if-statement ensures that only the threads whose indices fall are both smaller than the O_TILE_WIDTH should participate in the calculation of output pixels. The doubly nested for-loop iterates through the mask array and performs the multiply and accumulate operation on the mask element values and input pixel values. Since the input tile in the shared memory Ns includes all the halo elements, the index expressions Ns[i+ty][j+tx] gives the Ns element that should be multiplied with M[i][j]. The reader should notice that this is a straightforward extension of the index expression in corresponding for-loop in Figure 7.11. Finally, all threads whose output elements are in the valid range write their result values into their respective output elements.

To assess the benefit of the 2D tiled kernel over a basic kernel, we can also extend the analysis from 1D convolution. In a basic kernel, every thread in a thread block will perform $(\text{Mask\_Width})^2$ accesses to the image array. Thus, each thread block performs a total of $(\text{Mask\_Width})^2 * (\text{O\_TILE\_WIDTH})^2$ accesses to the image array.

In the tiled kernel, all threads in a thread block collectively load one input tile. Therefore, the total number of accesses by a thread block to the image array is $(\text{O\_TILE\_SIZE}+\text{Mask\_Width}+1)^2$. That is, the ratio of image array accesses between the basic and the tiled 2D convolution kernel is:

$$(\text{Mask\_Width})^2 * (\text{O\_TILE\_WIDTH})^2 \,/\, (\text{O\_TILE\_SIZE}+\text{Mask\_Width}+1)^2$$

The larger the ratio is, the more effective the tiled algorithm is in reducing the number of memory accesses as compared to the basic algorithm.

| TiLE_WIDTH | 8 | 16 | 32 | 64 |
|---|---|---|---|---|
| Reduction Mask_Width = 5 | 11.1 | 16 | 19.7 | 22.1 |
| Reduction Mask_Width = 9 | 20.3 | 36 | 51.8 | 64 |

*Figure 7.22 Image array access reduction ratio for different tile sizes.*

Figure 7.22 shows the trend of the image array access reduction ratio as we vary O_TILE_SIZE, the output tile size. As O_TILE_SIZE becomes very large, the size of the mask becomes negligible compared to tile size. Thus, each input element loaded will be used about $(\text{Mask\_Width})^2$ times. For Mask_Width value of 5, we expect that the ratio will approach 25 as the O_TILE_SIZE becomes much larger than 5. For example, for O_TILE_SIZE=64, the ratio is 22.1. This is significantly higher than the ratio of 11.1 for O_TILE_SIZE = 8. The important takeaway point is that we must have a sufficiently large O_TILE_SIZE in order for the tiled kernel to deliver its potential benefit. The cost of a large O_TILE_SIZE is the amount of shared memory needed to hold the input tiles.

For a larger Mask_Size, such as 9 in the bottom row of Figure 7.22, the ideal ratio should be $9^2$= 81. However, even with a large O_TILE_SIZE such as 64, the ratio is only 64. Note that O_TILE_SIZE=64 and Mask_Width=9 translates into input tile size of $72^2$=5184 elements or 20,736 bytes assuming single precision data. This is more than the about the amount of available shared memory in each SM of the current generation of GPUs. Stencil computation that is derived from finite difference methods for solving differential equation often require a Mask_Size of 9 or above to achieve numerical stability. Such stencil computation can benefit from larger amount of shared memory in future generations of GPUs.

## 7.7. Summary

In this chapter, we have studied convolution as an important parallel computation pattern. While convolution is used in many applications such as computer vision and video processing, it is also represents a general pattern that forms the basis of many parallel algorithms. For example one can view the stencil algorithms in partial differential equation (PDE) solvers as a special case of convolution. For another example, one can also view the calculation of grid point force or potential value as a special case of convolution.

We have presented a basic parallel convolution algorithm whose implementations will be limited by DRAM bandwidth for accessing both the input N and mask M elements. We then introduced the constant memory and a simple modification to the kernel and host code to take advantage of constant caching and eliminate practically all DRAM accesses for the mask elements. We further introduced a tiled parallel convolution algorithm that reduces DRAM bandwidth consumption by introducing more control flow divergence and programming complexity. Finally we presented a simpler tiled parallel convolution algorithm that takes advantage of the L2 caches.

Although we have shown kernel examples for only 1D convolution, the techniques are directly applicable to 2D and 3D convolutions. In general, the index calculation for the N and M arrays are more complex due to higher dimensionality. Also, one will have more loop nesting for each thread since multiple dimensions need to be traversed when loading tiles and/or calculating output values. We encourage the reader to complete these higher dimension kernels as homework exercises.

## 7.8. Exercises

1. Calculate the P[0] value in Figure 7.3.

2. Consider performing a 1D convolution on array N={4,1,3,2,3} with mask M={2,1,4}. What is the resulting output array?

3. What do you think the following 1D convolution masks are doing?

   (a) [0 1 0]
   (b) [0 0 1]
   (c) [1 0 0]
   (d) [-1/2 0 1/2]
   (e) [1/3 1/3 1/3]

4. Consider performing a 1D convolution on an array of size n with a mask of size m:

   (a) How many halo cells are there in total?
   (b) How many multiplications are performed if halo cells are treated as multiplications (by 0)?
   (c) How many multiplications are performed if halo cells are not treated as multiplications?

5. Consider performing a 2D convolution on a square matrix of size nxn with a square mask of size mxm:

   (a) How many halo cells are there in total?
   (b) How many multiplications are performed if halo cells are treated as multiplications (by 0)?

(c) How many multiplications are performed if halo cells are not treated as multiplications?

6. Consider performing a 2D convolution on a rectangular matrix of size n1xn2 with a rectangular mask of size m1xm2:

(a) How many halo cells are there in total?
(b) How many multiplications are performed if halo cells are treated as multiplications (by 0)?
(c) How many multiplications are performed if halo cells are not treated as multiplications?

7. Consider performing a 1D tiled convolution with the kernel shown in Figure 7.11 on an array of size n with a mask of size m using a tiles of size t.

(a) How many blocks are needed?
(b) How many threads per block are needed?
(c) How much shared memory is needed in total?
(d) Repeat the same questions if you were using the kernel in Figure 7.13.

8. Revised the 1D kernel in Figure 7.6 to perform 2D convolution. Add more width parameters to the kernel declaration as needed.

9. Revise the tiled 1D kernel in Figure 7.8 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.

10. Revise the tiled 1D kernel in Figure 7.11 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.