3.4 Matrix multiplication

Matrix-matrix multiplication, or matrix multiplication in short, is an important component of the Basic Linear Algebra Subprograms standard (see the "Linear Algebra Functions" sidebar). It is the basis of many linear algebra solvers, such as LU decomposition. It is also an important computation for deep learning using convolutional neural networks, which will be discussed in detail in Chapter 16, Deep Learning.

Linear Algebra Functions

Linear algebra operations are widely used in science and engineering applications. In the Basic Linear Algebra Subprograms (BLAS), a de facto standard for publishing libraries that perform basic algebra operations, there are three levels of linear algebra functions. As the level increases, the number of operations performed by the function increases. Level 1 functions perform vector operations of the form $\mathbf{y} = \alpha\mathbf{x}+\mathbf{y}$, *where* $\mathbf{x}$ *and* $\mathbf{y}$ *are vectors and* α *is a scalar. Our vector addition example is a special case of a level 1 function with* $\alpha = 1$. *Level 2 functions perform matrix-vector operations of the form* $\mathbf{y} = \alpha\mathbf{A}\mathbf{x}+\beta\mathbf{y}$, *where* $\mathbf{A}$ *is a matrix,* $\mathbf{x}$ *and* $\mathbf{y}$ *are vectors, and* α *and* β *are scalars. We will be studying a form of level 2 function in sparse linear algebra. Level 3 functions perform matrix-matrix operations in the form of* $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, *where* $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ *are matrices and* α and β *are scalars. Our matrix-matrix multiplication example is a special case of a level 3 function where* $\alpha = 1$ *and* $\beta = 0$. *These BLAS functions are important because they are used as basic building blocks of higher-level algebraic functions, such as linear system solvers and eigenvalue analysis. As we will discuss later, the performance of different implementations of BLAS functions can vary by orders of magnitude in both sequential and parallel computers.*

Matrix multiplication between an $I \times j$ (i rows by j columns) matrix `M` and a $j \times k$ matrix `N` produces an $I \times k$ matrix `P`. When a matrix multiplication is performed, each element of the output matrix `P` is an inner product of a row of `M` and a column of `N`. We will continue to use the convention where $P_{row,\ col}$ is the element at the rowth position in the vertical direction and the colth position in the horizontal direction. As shown in Fig. 3.10, $P_{row,col}$ (the small square in P) is the inner product of the vector formed by the rowth row of `M` (shown as a horizontal strip in `M`) and the vector formed by the colth column of `N` (shown as a vertical strip in `N`). The inner product, sometimes called the dot product, of two vectors is the sum of products of the individual vector elements. That is,

$$P_{row,col} = \sum M_{row,k} * N_{k,col} \qquad \text{for } k = 0,\ 1, \ldots \text{Width} - 1$$

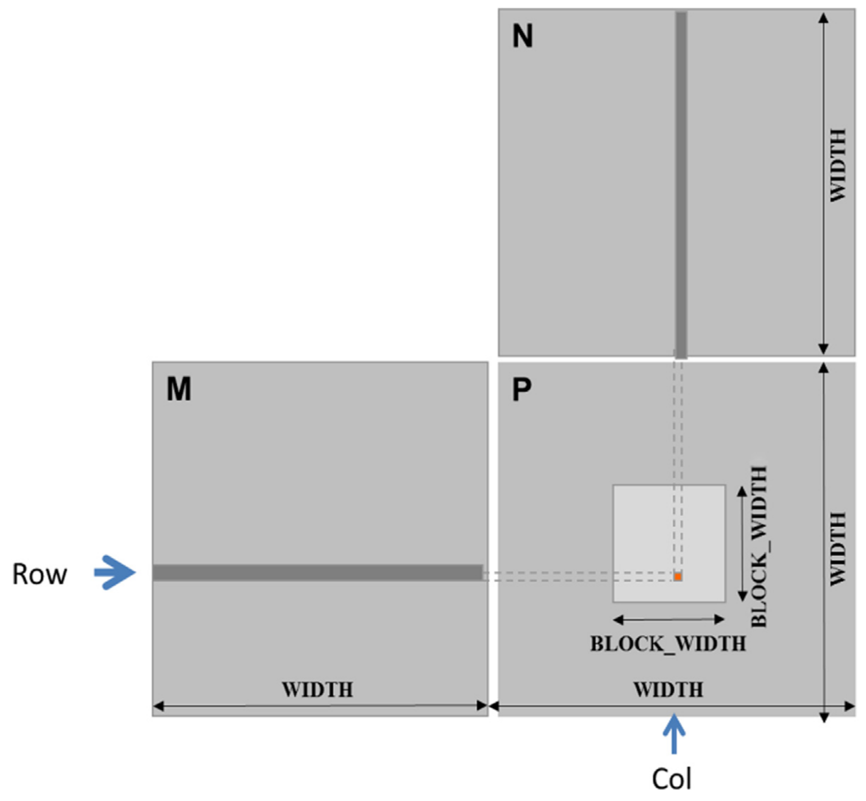

FIGURE 3.10

Matrix multiplication using multiple blocks by tiling P.

For example, in Fig. 3.10, assuming row = 1 and col = 5,

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

To implement matrix multiplication using CUDA, we can map the threads in the grid to the elements of the output matrix `P` with the same approach that we used for `colorToGrayscaleConversion`. That is, each thread is responsible for calculating one `P` element. The row and column indices for the `P` element to be calculated by each thread are the same as before:

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

and

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

With this one-to-one mapping, the `row` and `col` thread indices are also the row and column indices for their output elements. Fig. 3.11 shows the source code of

```
01    __global__ void MatrixMulKernel(float* M, float* N,
02                                    float* P, int Width) {
03        int row = blockIdx.y*blockDim.y+threadIdx.y;
04        int col = blockIdx.x*blockDim.x+threadIdx.x;
05        if ((row < Width) && (col < Width)) {
06            float Pvalue = 0;
07            for (int k = 0; k < Width; ++k) {
08                Pvalue += M[row*Width+k]*N[k*Width+col];
09            }
10            P[row*Width+col] = Pvalue;
11        }
12    }
```

FIGURE 3.11

A matrix multiplication kernel using one thread to compute one P element.

the kernel based on this thread-to-data mapping. The reader should immediately see the familiar pattern of calculating `row` and `col` (lines 03–04) and the if-statement testing if `row` and `col` are both within range (line 05). These statements are almost identical to their counterparts in `colorToGrayscaleConversion`. The only significant difference is that we are making a simplifying assumption that `matrixMulKernel` needs to handle only square matrices, so we replace both `width` and `height` with `Width`. This thread-to-data mapping effectively divides `P` into tiles, one of which is shown as a light-colored square in Fig. 3.10. Each block is responsible for calculating one of these tiles.

We now turn our attention to the work done by each thread. Recall that $P_{row,col}$ is calculated as the inner product of the rowth row of `M` and the colth column of `N`. In Fig. 3.11 we use a for-loop to perform this inner product operation. Before we enter the loop, we initialize a local variable `Pvalue` to 0 (line 06). Each iteration of the loop accesses an element from the rowth row of `M` and an element from the colth column of `N`, multiplies the two elements together, and accumulates the product into `Pvalue` (line 08).

Let us first focus on accessing the `M` element within the for-loop. `M` is linearized into an equivalent 1D array using row-major order. That is, the rows of `M` are placed one after another in the memory space, starting with the 0th row. Therefore the beginning element of row 1 is `M[1*Width]` because we need to account for all elements of row 0. In general, the beginning element of the rowth row is `M[row*Width]`. Since all elements of a row are placed in consecutive locations, the kth element of the rowth row is at `M[row*Width+k]`. This linearized array offset is what we use in Fig. 3.11 (line 08).

We now turn our attention to accessing N. As is shown in Fig. 3.11, the beginning element of the colth column is the colth element of row 0, which is `N[col]`. Accessing the next element in the colth column requires skipping over an entire row. This is because the next element of the same column is the same element in the next row. Therefore the kth element of the colth column is `N[k*Width+col]` (line 08).

After the execution exits the for-loop, all threads have their `P` element values in the `Pvalue` variables. Each thread then uses the 1D equivalent index expression row*Width+col to write its P element (line 10). Again, this index pattern is like that used in the colorToGrayscaleConversion kernel.

wWe now use a small example to illustrate the execution of the matrix multiplication kernel. Fig. 3.12 shows a 4 × 4 P with BLOCK_WIDTH = 2. Although such small matrix and block sizes are not realistic, they allow us to fit the entire example into one picture. The P matrix is divided into four tiles, and each block calculates one tile. We do so by creating blocks that are 2 × 2 arrays of threads, with each thread calculating one P element. In the example, thread (0,0) of block (0,0) calculates $P_{0,0}$, whereas thread (0,0) of block (1,0) calculates $P_{2,0}$.

The row and col indices in matrixMulKernel identify the P element to be calculated by a thread. The row index also identifies the row of M, and the col index identifies the column of N as input values for the thread. Fig. 3.13 illustrates the multiplication actions in each thread block. For the small matrix multiplication example, threads in block (0,0) produce four dot products. The row and col indices of thread (1,0) in block (0,0) are 0*0 + 1 = 1 and 0*0 + 0 = 0, respectively. The thread thus maps to $P_{1,0}$ and calculates the dot product of row 1 of M and column 0 of N.

Let us walk through the execution of the for-loop of Fig. 3.11 for thread (0,0) in block (0,0). During iteration 0 (k = 0), row*Width + k = 0*4 + 0 = 0 and k*Width + col = 0*4 + 0 = 0. Therefore the input elements accessed are M[0] and N[0], which are the 1D equivalent of $M_{0,0}$ and $N_{0,0}$. Note that these are indeed the 0th elements of row 0 of M and column 0 of N. During iteration 1 (k = 1), row*Width + k = 0*4 + 1 = 1 and k*Width + col = 1*4 + 0 = 4. Therefore we are accessing M[1] and N[4], which are the 1D equivalent of $M_{0,1}$ and $N_{1,0}$. These are the first elements of row 0 of M and column 0 of N. During iteration 2 (k = 2), row*Width + k = 0*4 + 2 = 2 and k*Width + col = 2*4 + 0 = 8, which results in M[2] and N[8]. Therefore the elements accessed are the 1D equivalent of $M_{0,2}$ and $N_{2,0}$. Finally, during iteration 3 (k = 3), row*Width + k = 0*4 + 3 = 3 and k*Width + col =

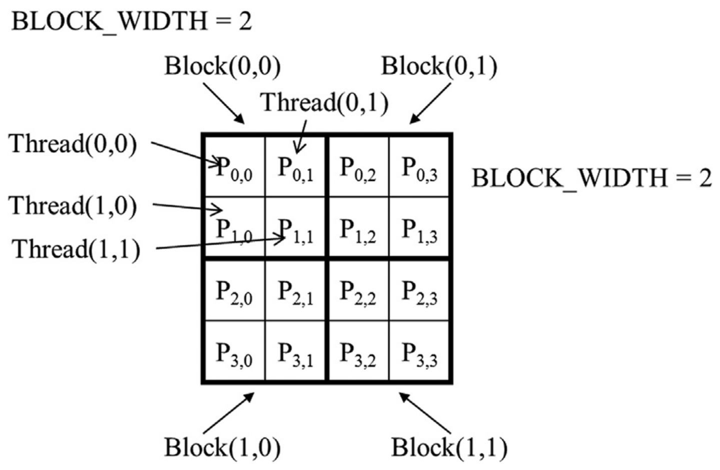

FIGURE 3.12

A small execution example of matrixMulKernel.

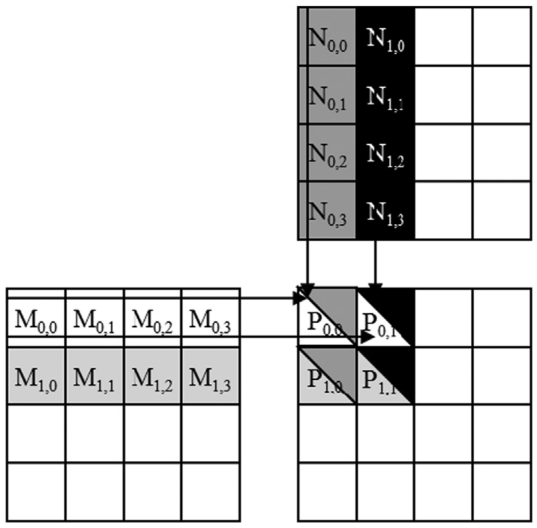

FIGURE 3.13

Matrix multiplication actions of one thread block.

3*4 + 0 = 12, which results in `M[3]` and `N[12]`, the 1D equivalent of $M_{0,3}$ and $N_{3,0}$. We have now verified that the for-loop performs the inner product between the 0th row of M and the 0th column of N for thread (0,0) in block (0,0). After the loop, the thread writes `P[row*Width+col]`, which is `P[0]`. This is the 1D equivalent of $P_{0,0}$, so thread (0,0) in block (0,0) successfully calculated the inner product between the 0th row of `M` and the 0th column of `N` and deposited the result in $P_{0,0}$.

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

Since the size of a grid is limited by the maximum number of blocks per grid and threads per block, the size of the largest output matrix `P` that can be handled by `matrixMulKernel` will also be limited by these constraints. In the situation in which output matrices larger than this limit are to be computed, one can divide the output matrix into submatrices whose sizes can be covered by a grid and use the host code to launch a different grid for each submatrix. Alternatively, we can change the kernel code so that each thread calculates more `P` elements. We will explore both options later in this book.

3.5 Summary

CUDA grids and blocks are multidimensional with up to three dimensions. The multidimensionality of grids and blocks is useful for organizing threads to be mapped to multidimensional data. The kernel execution configuration parameters define the dimensions of a grid and its blocks. Unique coordinates in `blockIdx` and `threadIdx` allow threads of a grid to identify themselves and their domains of data. It is the programmer's responsibility to use these variables in kernel functions so that the threads can properly identify the portion of the data to process. When accessing multidimensional data, programmers will often have to linearize multidimensional indices into a 1D offset. The reason is that dynamically allocated multidimensional arrays in C are typically stored as 1D arrays in row-major order. We use examples of increasing complexity to familiarize the reader with the mechanics of processing multidimensional arrays with multidimensional grids. These skills will be foundational for understanding parallel patterns and their associated optimization techniques.

Exercises

1. In this chapter we implemented a matrix multiplication kernel that has each thread produce one output matrix element. In this question, you will implement different matrix-matrix multiplication kernels and compare them.
   - **a.** Write a kernel that has each thread produce one output matrix row. Fill in the execution configuration parameters for the design.
   - **b.** Write a kernel that has each thread produce one output matrix column. Fill in the execution configuration parameters for the design.
   - **c.** Analyze the pros and cons of each of the two kernel designs.
2. A matrix-vector multiplication takes an input matrix B and a vector C and produces one output vector A. Each element of the output vector A is the dot product of one row of the input matrix B and C, that is, $A[i] = \sum^{j} B[i][j] + C[j]$. For simplicity we will handle only square matrices whose elements are single-precision floating-point numbers. Write a matrix-vector multiplication kernel and the host stub function that can be called with four parameters: pointer to the output matrix, pointer to the input matrix, pointer to the input vector, and the number of elements in each dimension. Use one thread to calculate an output vector element.
3. Consider the following CUDA kernel and the corresponding host function that calls it:

```
__global__ void foo_kernel(float* a, float* b, unsigned int M,
unsigned int N) {
    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
    if(row < M && col < N) {
        b[row*N + col] = a[row*N + col]/2.1f + 4.8f;
    }
}
void foo(float* a_d, float* b_d) {
    unsigned int M = 150;
    unsigned int N = 300;
    dim3 bd(16, 32);
    dim3 gd((N - 1)/16 + 1, (M - 1)/32 + 1);
    foo_kernel <<< gd, bd >>>(a_d, b_d, M, N);
}
```

   - **a.** What is the number of threads per block?
   - **b.** What is the number of threads in the grid?