CUDA - Matrix Multiplication

We have learnt how threads are organized in CUDA and how they are mapped to multi-dimensional data. Let us go ahead and use our knowledge to do matrix-multiplication using CUDA. But before we delve into that, we need to understand how matrices are stored in the memory. The manner in which matrices are stored affect the performance by a great deal.

2D matrices can be stored in the computer memory using two layouts − **row-major** and **column-major**. Most of the modern languages, including C (and CUDA) use the row-major layout. Here is a visual representation of the same of both the layouts −

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

Matrix to be stored

M0,0	M0,1	M0,2	M0,3	M1,0	M1,1	M1,2	M1,3	M2,0	M2,1	M2,2	M2,3	M3,0	M3,1	M3,2	M3,3

Row-major layout

Actual organization in memory −

M0,0	M1,0	M2,0	M3,0	M0,1	M1,1	M2,1	M3,1	M0,2	M1,2	M2,2	M3,2	M0,3	M1,3	M2,3	M3,3

Actual organization in memory −

Column-major layout

Actual organization in memory

Note that a 2D matrix is stored as a 1D array in memory in both the layouts. Some languages like FORTRAN follow the column-major layout.

Addressing

In row-major layout, element(x,y) can be addressed as: x*width + y. In the above example, the width of the matrix is 4. For example, element (1,1) will be found at position −

1*4 + 1 = 5 in the 1D array.

We will be mapping each data element to a thread. The following mapping scheme is used to map data to thread. This gives each thread its unique identity.

```
row=blockIdx.x*blockDim.x+threadIdx.x;
col=blockIdx.y*blockDim.y+threadIdx.y;
```

We know that a grid is made-up of blocks, and that the blocks are made up of threads. All threads in the same block have the same block index. Each coloured chunk in the above figure represents a block (the yellow one is block 0, the red one is block 1, the blue one is block 2 and the green one is block 3). So, for each block, we have blockDim.x=4 and blockDim.y=1. Let us find the unique identity of thread M(0,2). Since it lies in the yellow array, blockIdx.x=0 and threadIdx.x=2. So, we get: 0*4+2=2.

In the previous chapter, we noted that we often launch more threads than actually needed. To ensure that the extra threads do not do any work, we use the following 'if' condition −

```
if(row<width && col<width) {
   then do work
}
```

The above condition is written in the kernel. It ensures that extra threads do not do any work.

Matrix multiplication between a (IxJ) matrix d_M and (JxK) matrix d_N produces a matrix d_P with dimensions (IxK). The formula used to calculate elements of d_P is −

```
d_Px,y = Σ d_Mx,,k*d_Nk,y, for k=0,1,2,....width
```

A d_P element calculated by a thread is in 'blockIdx.y*blockDim.y+threadIdx.y' row and 'blockIdx.x*blockDim.x+threadIdx.x' column. Here is the actual kernel that implements the above logic.

```
__global__ void simpleMatMulKernell(float* d_M, float* d_N, float* d_P, int width)
{
```

This helps to calculate row and col to address what element of d_P will be calculated by this thread.

```
int row = blockIdx.y*width+threadIdx.y;
int col = blockIdx.x*width+threadIdx.x;
```

This ensures that the extra threads do not do any work.

```
if(row<width && col <width) {
   float product_val = 0
   for(int k=0;k<width;k++) {
      product_val += d_M[row*width+k]*d_N[k*width+col];
   }
   d_p[row*width+col] = product_val;
}
```

Let us now understand the above kernel with an example −

Let d_M be −

2	4	1
8	7	4
7	4	9

The above matrix will be stored as −

2	4	1	8	7	4	7	4	9

And let d_N be −

4	8	9
1	7	0
2	5	4

The above matrix will be stored as −

4	8	9	1	7	0	2	5	4

Since d_P will be a 3x3 matrix, we will be launching 9 threads, each of which will compute one element of d_P.

d_P matrix

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Let us compute the (2,1) element of d_P by doing a dry-run of the kernel −

```
row=2;
col=1;
```

When (k=0)

```
product_val = 0 + d_M[2*3+0] * d_N[0*3+1]
product_val = 0 + d_M[6]*d_N[1] = 0+7*8=56
```

1st Iteration

```
product_val = 56 + d_M[2*3+1]*d_N[1*3+1]
product_val = 56 + d_M[7]*d_N[4] = 84
```

Final Iteration

```
product_val = 84+d_M[2*3+2]*d_N[2*3+1]
product_val = 84+d_M[8]*d_N[7] = 129
```

Now, we have −

```
d_P[7] = 129
```