Matrix-Matrix Multiplication – A More Complex Kernel

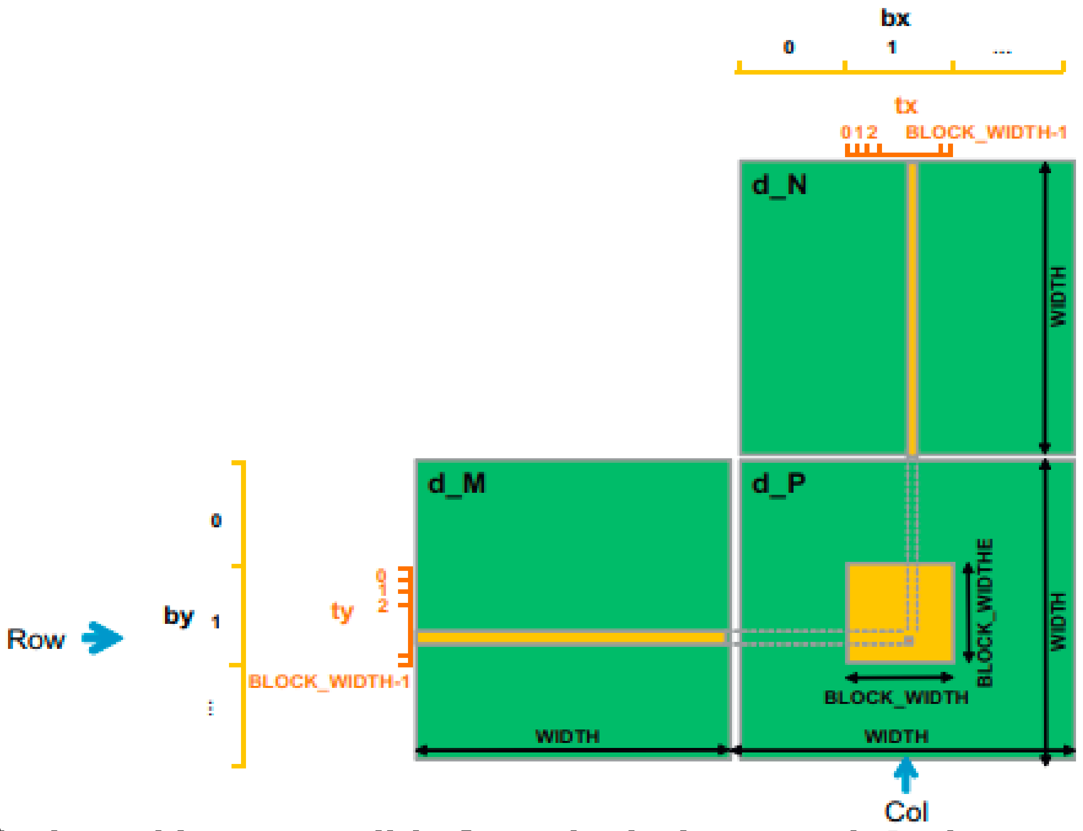
- We can map every valid data element in a 2D array to a unique thread using threadIdx, blockIdx, blockDim, and gridDim variables:

```
// Calculate the row #
int  Row = blockIdx.y*blockDim.y + threadIdx.y;
// Calculate the column #
int  Col = blockIdx.x*blockDim.x + threadIdx.x;
```

- Matrix-Matrix multiplication between an **I x J** matrix **d_M** and a **J x K** matrix **d_N** produces an **I x K** matrix **d_P**.

Matrix-Matrix Multiplication – A More Complex Kernel

- When performing a matrix-matrix multiplication, ***each element of the product matrix d_P is an inner product of a row of d_M and a column of d_N***. The inner product between two vectors is the sum of products of corresponding elements. That is, **$d\_P_{Row,Col} = \Sigma d\_M_{Row,k} * d\_N_{k,Col}$, for k = 0, 1, ... Width-1**.



- We design a kernel where **each thread is responsible for calculating one d_P element**.

- The d_P element calculated by a thread is in **row blockIdx.y * blockDim.y + threadIdx.y** and in **column blockIdx.x * blockDim.x + threadIdx.x**.

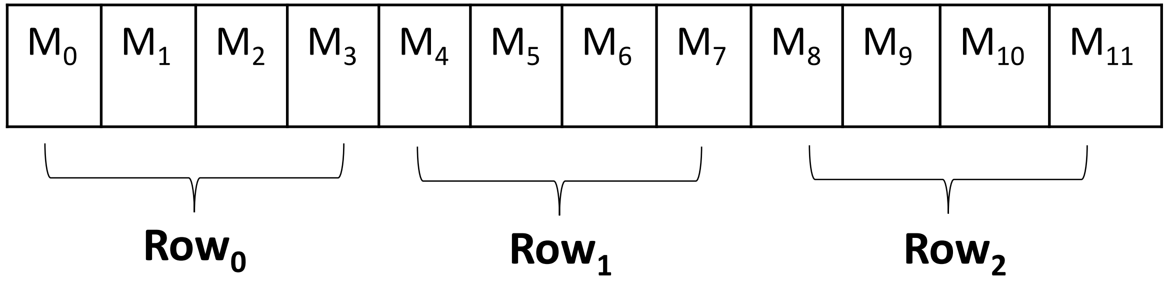
Matrix-Matrix Multiplication – three variants

We will write program in CUDA to multiply **matrix-A (ha $\times$ wa)** and **matrix-B (hb $\times$ wb)** in 3 variations as follows:

1. Each row of resultant matrix to be computed by one thread.
2. Each column of resultant matrix to be computed by one thread.
3. Each element of resultant matrix to be computed by one thread.

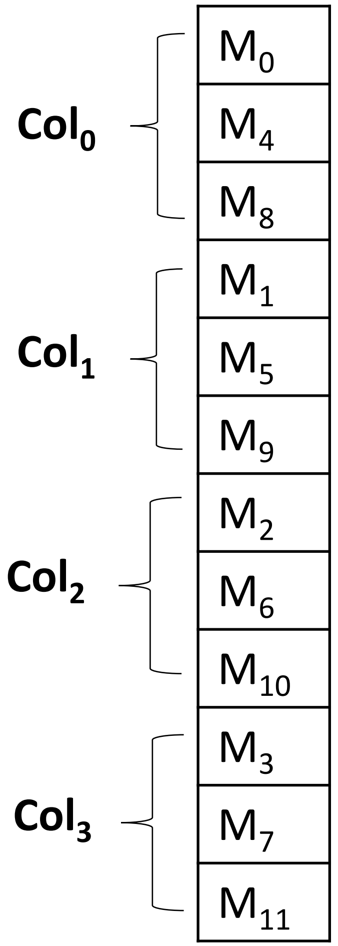
Accessing Row & Col of a matrix

Accessing Row


M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11}
$\mathbf{Row_0}$
$\mathbf{Row_1}$
$\mathbf{Row_2}$

- Recall that a matrix M is linearized into an equivalent 1D array where the rows of M are placed one after another in the memory space, starting with the 0 row.
- For example, the beginning element of the 1 row is M[1 * Width] because we need to account for all elements of the 0 row.
- In general, the beginning element of the Row row is M[Row * Width].
- Since all elements of a row are placed in consecutive locations, the k element of the Row row is at **M[Row * Width + k]**.

Accessing Col



- The beginning element of the Col column is the Col element of the **0** row, which is **M[Col].**
- Accessing each additional element in the **Col** column requires skipping over entire rows. This is because the next element of the same column is actually the same element in the next row.
- Therefore, the k element of the Col column is **M[k * Width + Col]**.

Matrix *M*

M_0	M_1	M_2	M_3
M_4	M_5	M_6	M_7
M_8	M_9	M_{10}	M_{11}

1. Each row of resultant matrix to be computed by one thread

```
multiplyKernel_a<<<1,ha>>>(d_a, d_b, d_c, wa, wb);

__global__ void multiplyKernel_rowwise(int * a, int * b, int * c, int wa, int wb)
{
        int ridA = threadIdx.x;
        int sum;
        for(int cidB = 0; cidB < wb; cidB++)
        {
                sum= 0;
                for(k = 0; k< wa; k++)
                {
                        sum += (a[ridA * wa + k] * b[k * wb + cidB]);
                }
        c[ridA * wb+ cidB]  = sum;
        }
}
```

2. Each column of resultant matrix to be computed by one thread

```
multiplyKernel_b<<<1, wb>>>(d_a, d_b, d_c, ha,wa);

 __global__ void multiplyKernel_colwise(int * a, int * b, int * c, int ha, int wa)
{
        int cidB = threadIdx.x;
        int wb = blockDim.x;
        int sum, k;
          for(ridA = 0; ridA < ha; ridA++)
          {
                    sum = 0;
                    for( k=0; k< wa; k++)
                    {
                              sum += (a[ridA * wa + k] * b[k * wb + cidB]);
                    }
        c[ridA * wb + cidB] =sum;
          }
}
```

3. Each element of resultant matrix to be computed by one thread

```
multiplyKernel_b<<<(1, 1), (wb,ha)>>>(d_a, d_b, d_c, wa);

__global__ void multiplyKernel_elementwise(int * a, int * b, int * c,  int wa)
{
        int ridA = threadIdx.y;
        int cidB= threadIdx.x;
        int wb = blockDim.x;
        int sum=0, k;

         for( k = 0; k < wa; k++)
          {
              sum += (a[ridA * wa + k] * b[k * wb + cidB]);
          }
         c[ridA * wb + cidB] =sum;

}
```

Matrix-Matrix Multiplication– Kernel for thread-to-data mapping

- In the following kernel, we assume **square matrices** having dimension **Width*Width**.
- Throughout the source code, instead of using a numerical value 16 for the block-width, the programmer can use the name BLOCK_WIDTH by defining it. It helps in *autotuning*.

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
 {
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column index of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((Row < Width) && (Col < Width))
   {
      float Pvalue = 0;
      // each thread computes one element of the block sub-matrix
              for (int k = 0; k < Width; ++k)
                   Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];

              d_P[Row*Width+Col] = Pvalue;
    }
 }
```

Kernel code

```
#define BLOCK_WIDTH 16

// Setup the execution configuration
int NumBlocks = Width/BLOCK_WIDTH;
if (Width % BLOCK_WIDTH)
                NumBlocks++;

dim3 dimGrid(NumBlocks, NumbBlocks);
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

// Launch the device computation threads
 matrixMulKernel<<dimGrid, dimBlock>>(M, N, P, Width);
```

Host code