Matrix-Matrix Multiplication – A More Complex Kernel

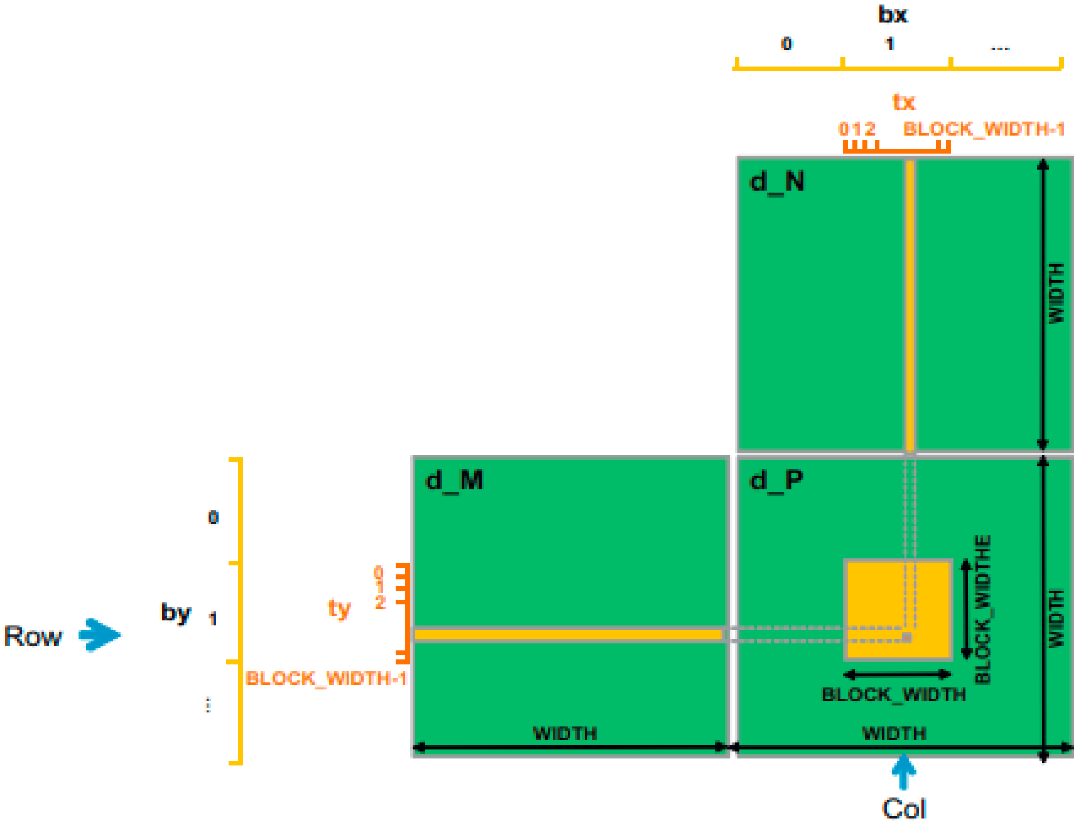
- We can map every valid data element in a 2D array to a unique thread using threadIdx, blockIdx, blockDim, and gridDim variables:

```
// 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**.

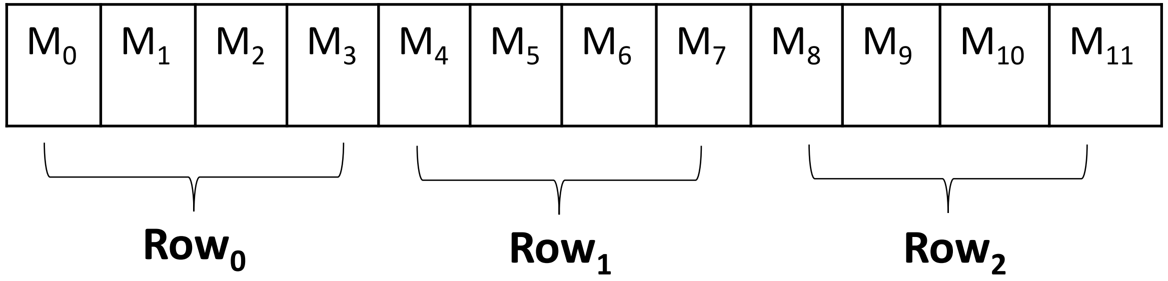
Matrix-Matrix Multiplication – three variants

We will write program in CUDA to multiply **matrix-A** **(ha × wa)** and **matrix-B** **(hb × 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.

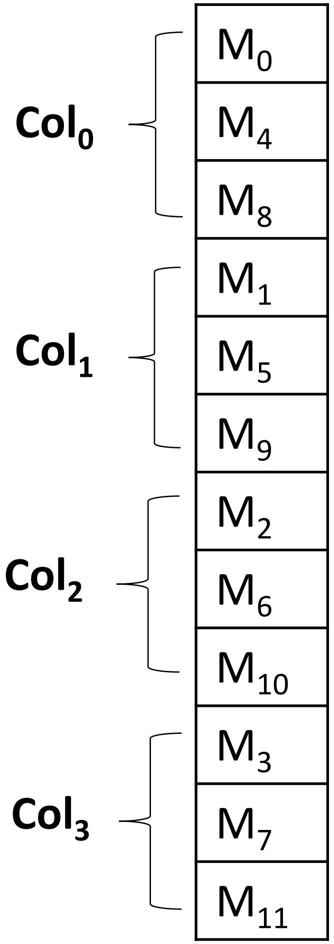
Accessing Row & Col of a matrix

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}
Row_0
Row_1
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*.

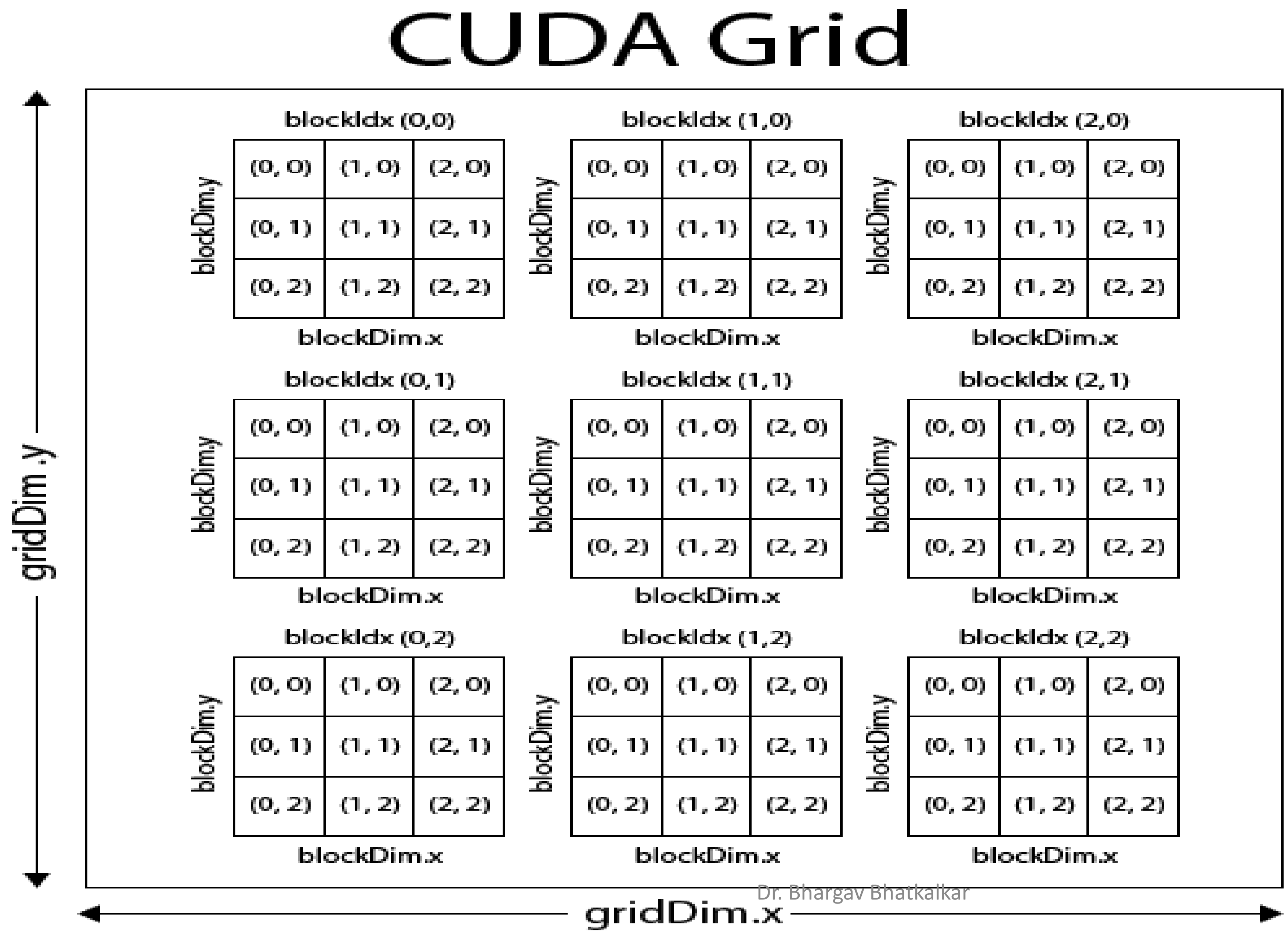
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
   // Calculate the row index of the d_P element and d_M
   int Row = blockIdx.y*blockDim.y+threadIdx.y;
   // Calculate the column index of d_P and d_N
   int Col = blockIdx.x*blockDim.x+threadIdx.x;
   if ((Row < Width) && (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

Calculations of global threadID



blockIdx (0,0)

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

blockIdx (1,0)

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

blockIdx (2,0)

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

blockIdx (0,1)

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

blockIdx (1,1)

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

blockIdx (2,1)

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

blockIdx (0,2)

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

blockIdx (1,2)

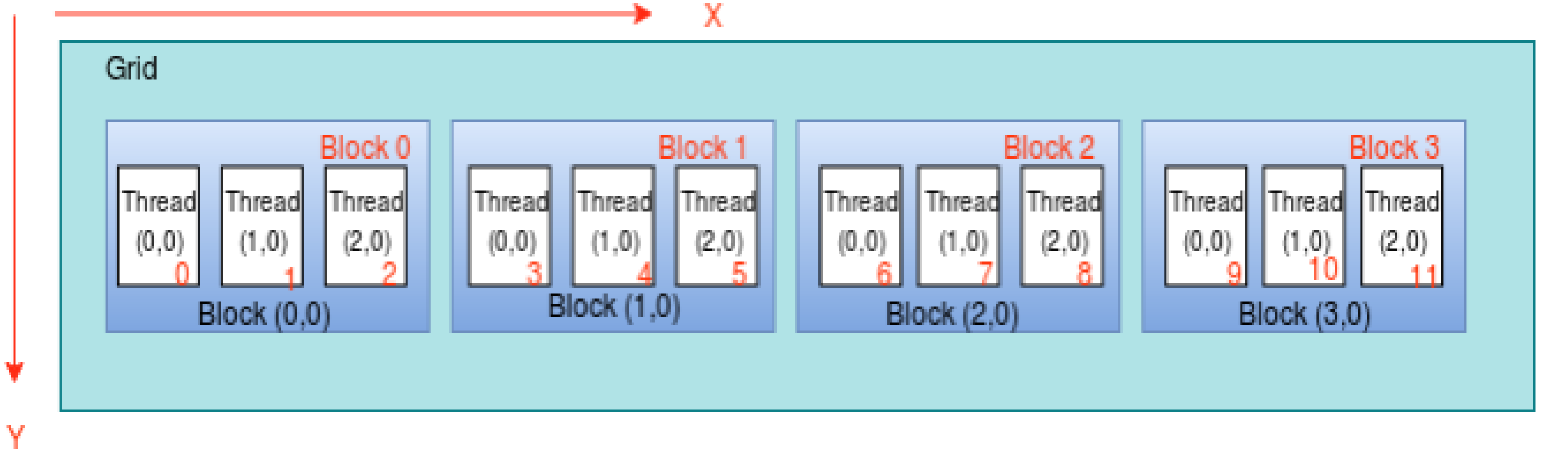
(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)

blockIdx (2,2)

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

Note: (x,y) is followed

Calculations of global threadID – ID Grid/1D Block



Indices given in RED color are the unique numbers for each block and each thread

<<< 4, 3 >>>

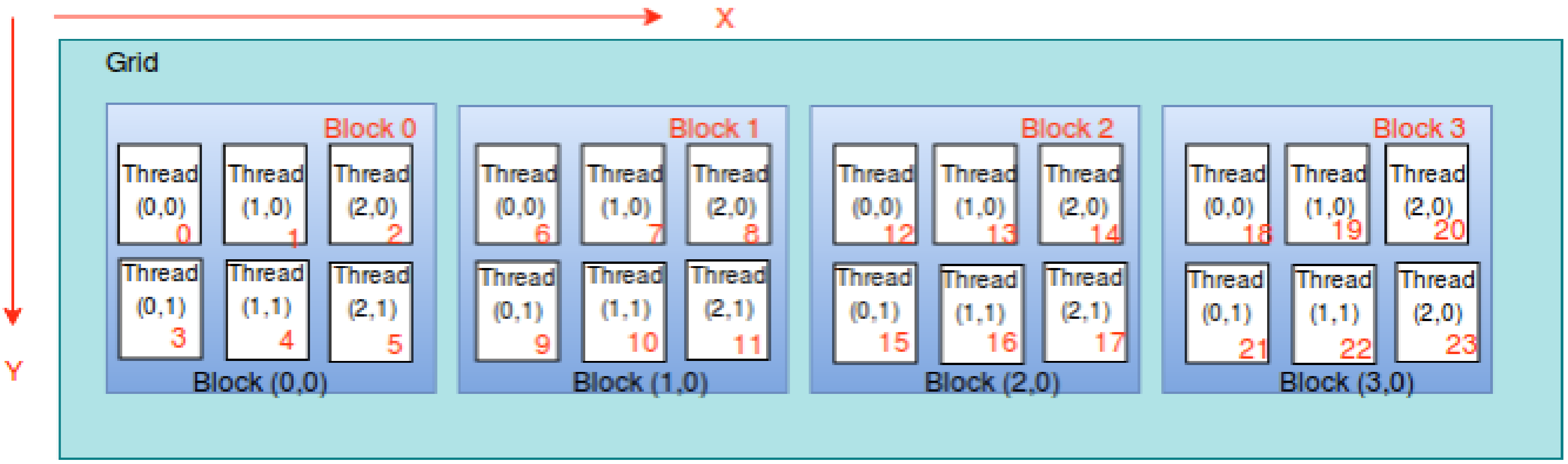
(x,y) notation used

```
threadId = (blockIdx.x * blockDim.x) + threadIdx.x
```

Let’s check the equation for Thread (2,0) in Block (1,0).

Thread ID = (1 * 3) + 2 =3+2 = 5

Calculations of global threadID – ID Grid/2D Block



<<< 4, (3,2) >>>

Indices given in RED color are the unique numbers for each block and each thread

Gtid=blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;

Let’s check the equation for Thread (2,1) in Block (1,0).

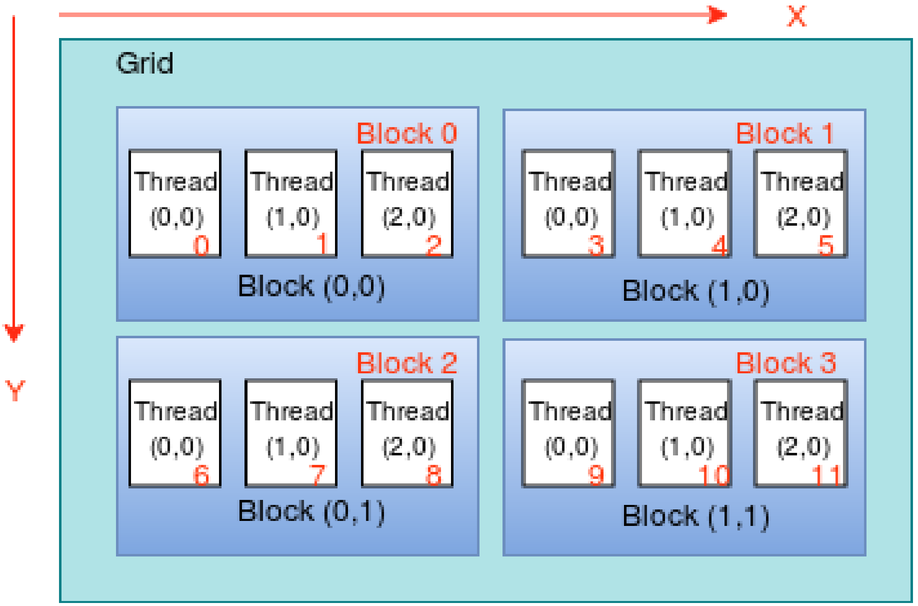
Thread ID = (1*3*2)+(1*3)+2 = 6+3+2 =11

Here (1*3*2) → count threads in block 0

(1*3) →count thread(0,0),(1,0) and (2,0) in block 1

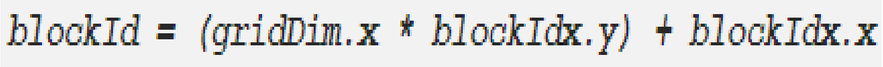
Then add the threadIdx.x of the particular thread.

Calculations of global threadID – 2D Grid/1D Block

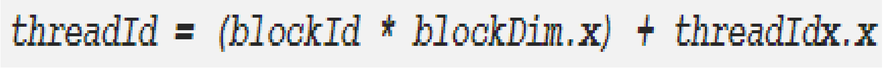


Indices given in RED color are the unique numbers for each block and each thread

<<< (2,2), 3 >>>



```
blockId = (gridDim.x * blockIdx.y) + blockIdx.x
```



```
threadId = (blockId * blockDim.x) + threadIdx.x
```

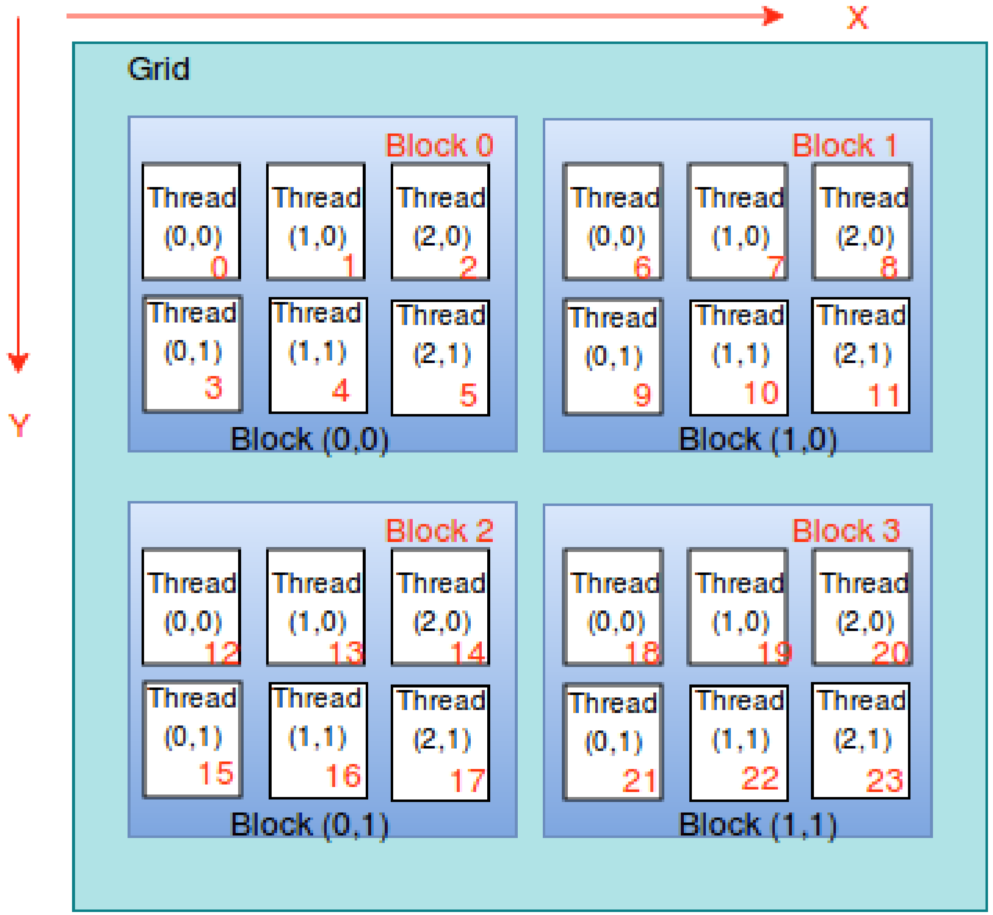
Let’s check the equation for Thread (1,0) in Block (1,1).

blockId = (2*1) + 1 =2+1=3

threadID = (3*3)+1 =9+1=10

Calculations of global threadID – 2D Grid/2D Block

<<< (2,2), (2,3) >>>



blockId = (gridDim.x * blockIdx.y) + blockIdx.x

threadId = (blockId * (blockDim.x * blockDim.y))

+

(threadIdx.y * blockDim.x) + threadIdx.x

Let's check the equation for Thread (2,1) in Block (0,1).

block Id = (2 * 1) + 0 = 2

Thread Id = (2 * (3 * 2))+(1*3) + 2 = 12+3+2 = 17

Calculations of global threadID

1D grid of 1D blocks:

```
__device__int getGlobalID_1D_1D(){
                return blockIdx.x *blockDim.x + threadIdx.x;
            }
```

1D grid of 2D blocks:

```
__device__int getGlobalID_1D_2D(){
            return blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;
}
```

1D grid of 3D blocks:

```
__device__int getGlobalID_1D_3D(){
            return blockIdx.x * blockDim.x * blockDim.y * blockDim.z + threadIdx.z * blockDim.y * blockDim.x +
                threadIdx.y * blockDim.x + threadIdx.x;
}
```

Calculations of global threadID

2D grid of 1D blocks:

```
__device__ int getGlobalID_2D_1D(){
        int blockId = blockIdx.y * gridDim.x + blockIdx.x;
        int threadId = blockId * blockDim.x + threadIdx.x;
        return threadId;
}
```

2D grid of 2D blocks:

```
__device__int getGlobalID_2D_2D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x;
        int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

2D grid of 3D blocks:

```
__device__int getGlobalID_2D_3D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x;
        int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x *
                 blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

Calculations of global threadID

3D grid of 1D blocks:

```
__device__int getGlobalID_3D_1D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x  + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * blockDim.x + threadIdx.x;
        return threadId;
}
```

3D grid of 2D blocks:

```
__device__int getGlobalID_3D_2D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

3D grid of 3D blocks:

```
__device__int getGlobalID_3D_3D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x *
                blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

Querying Device Properties

- When a CUDA application executes on a system, how can it find out the number of SMs in a device and the number of threads that can be assigned to each SM?

- The CUDA runtime system has an API function **cudaGetDeviceCount()** that returns the number of available CUDA devices in the system:

```
int dev_count;
cudaGetDeviceCount( &dev_count);
```

- The CUDA runtime system numbers all the available devices in the system from **0** to **dev_count-1**. It provides an API function **cudaGetDeviceProperties()** that returns the properties of the device of which the number is given as an argument:

```
cudaDeviceProp dev_prop;
for (i=0; i<dev_count; i++) {
cudaGetDeviceProperties( &dev_prop, i);
// decide if device has sufficient resources and capabilities
}
```

- The built-in type **cudaDeviceProp** is a *C structure* with fields that represent the properties of a CUDA device

Querying Device Properties

The built-in type **cudaDeviceProp** is a *C structure* with fields that represent the properties of a CUDA device

- The maximal number of threads allowed in a block in the queried device is given by the field **dev_prop.maxThreadsPerBlock.**

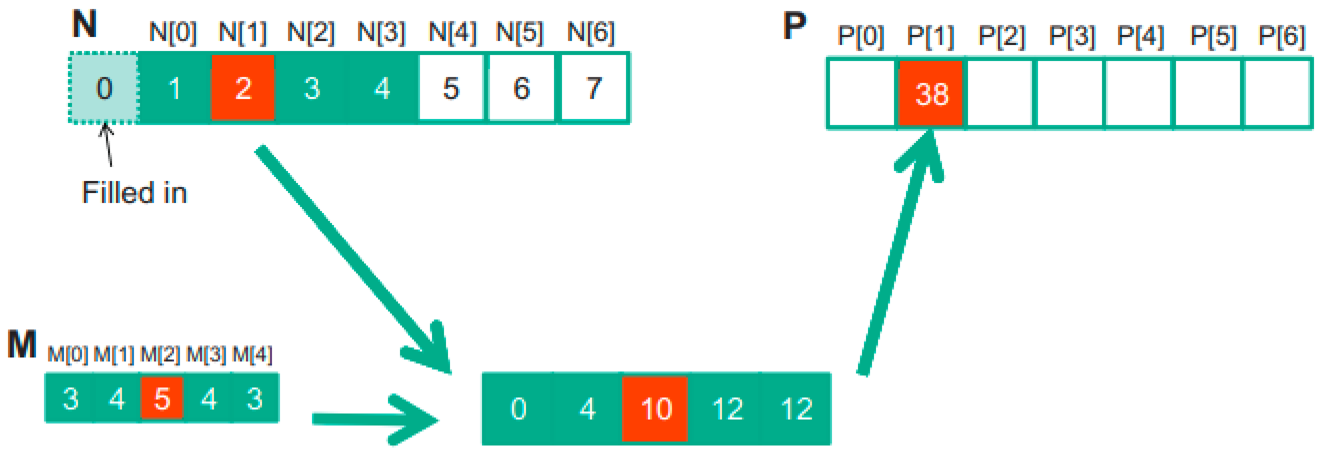
- The number of SMs in the device is given in **dev_prop. multiProcessorCount**.

- The clock frequency of the device is in **dev_prop. clockRate**.

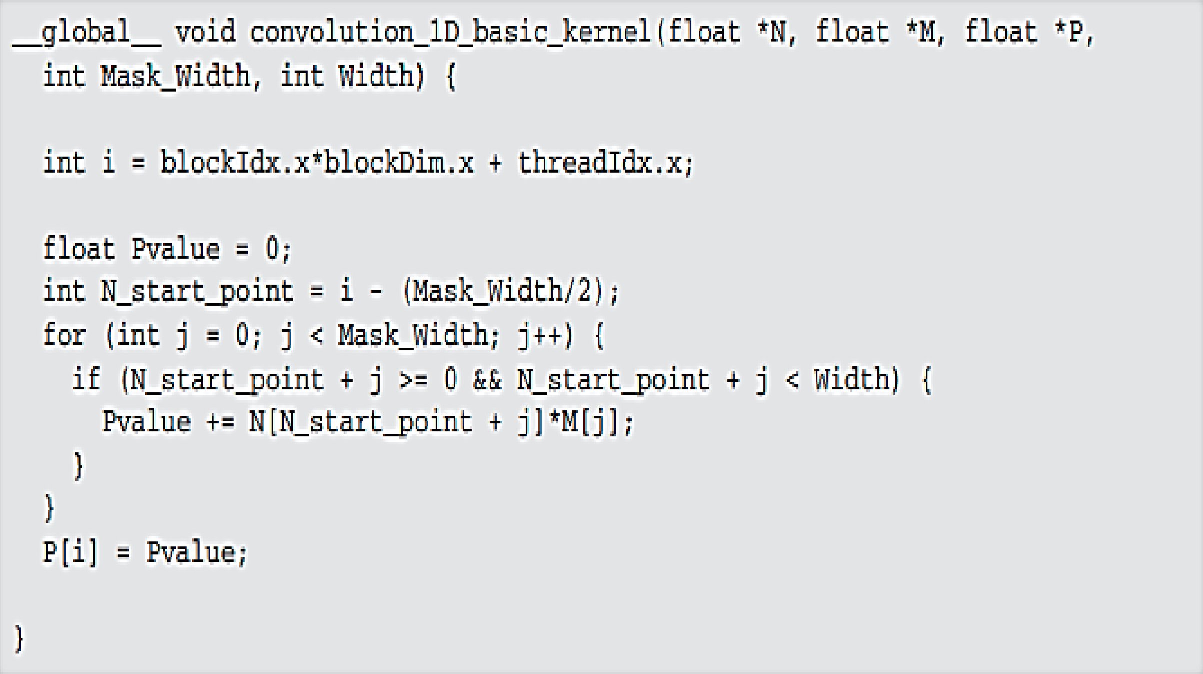
- The host code can find the *maximal number of threads allowed along each dimension of a block* in **dev_prop.maxThreadsDim[0]** (for the x dimension), **dev_prop.maxThreadsDim[1]** (for the y dimension), and **dev_prop.maxThreadsDim[2]** (for the z dimension).

- The host code can find the *maximal number of blocks allowed along each dimension of a grid* in **dev_prop.maxGridSize[0]** (for the x dimension), **dev_prop.maxGridSize[1]** (for the y dimension), and **dev_prop.maxGridSize[2]** (for the z dimension).

Constant Memory and Caching



$$\begin{aligned} P[1] &= 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4] \\ &= 0 * 3 + 1*4 + 2*5 + 3*4 + 4*3 \\ &= 38 \end{aligned}$$



```
__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;

}
```

- We can make three interesting observations about the way the mask array ***M*** is used in convolution:
  1. **First**, the size of the ***M*** array is typically small.
  2. **Second**, the contents of ***M*** are not changed throughout the execution of the kernel.
  3. **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 1D parallel convolution.

- These properties make the mask array an excellent candidate for **constant memory and caching.**

Constant Memory and Caching

- The CUDA programming model allows programmers to declare a variable in the constant memory.

- Like global memory variables, **constant memory variables** 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 can vary from device to device. The amount of constant memory available on a device can be learned with a device property query. Assume that **dev_prop** is returned by **cudaGetDeviceProperties()**. The field **dev_prop.totalConstMem** indicates the amount of constant memory available on a device is in the field.

- To declare an ***M*** array in constant memory, the host code declares it as follows:

```
#define MAX_MASK_WIDTH  5
__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.

Constant Memory and Caching

- 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, h_M, Mask_Width * sizeof(float));**

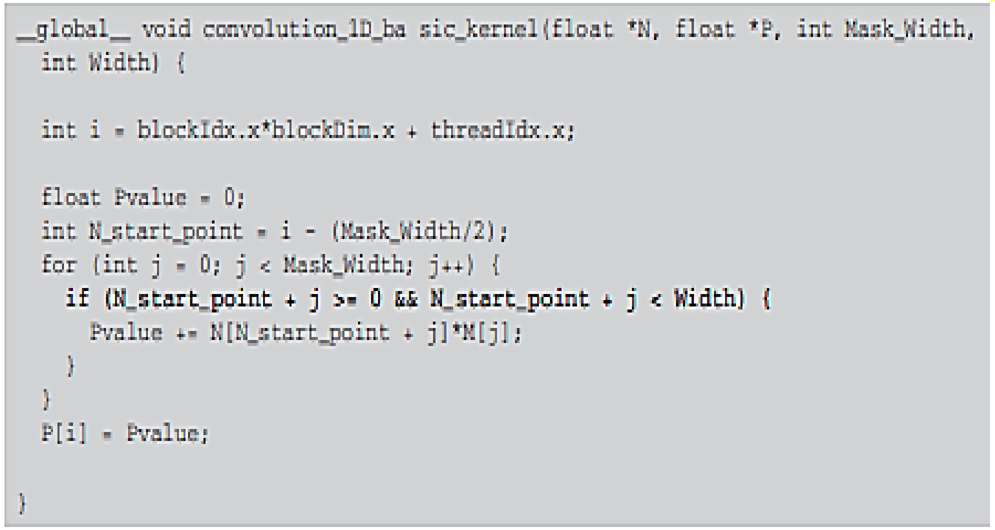
** This is a special memory copy function that informs the CUDA runtime that the data being copied into the constant memory will not be changed during kernel execution.

- In general, the use of the **cudaMemcpyToSymbol()** 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.

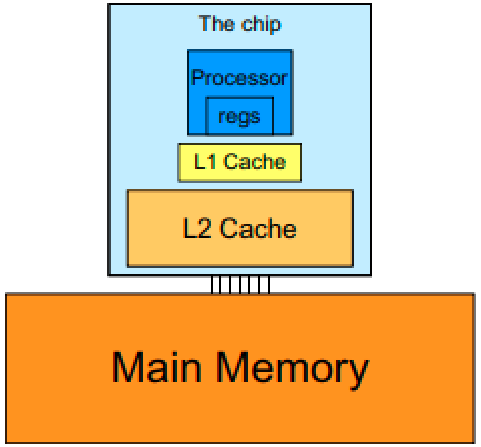


```
__global__ void convolution_1D_ba sic_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  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;

}
```

- This is a revised kernel to use the constant memory for the 1D parallel convolution.
- **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.

Constant Memory and Caching

- 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.
- 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.



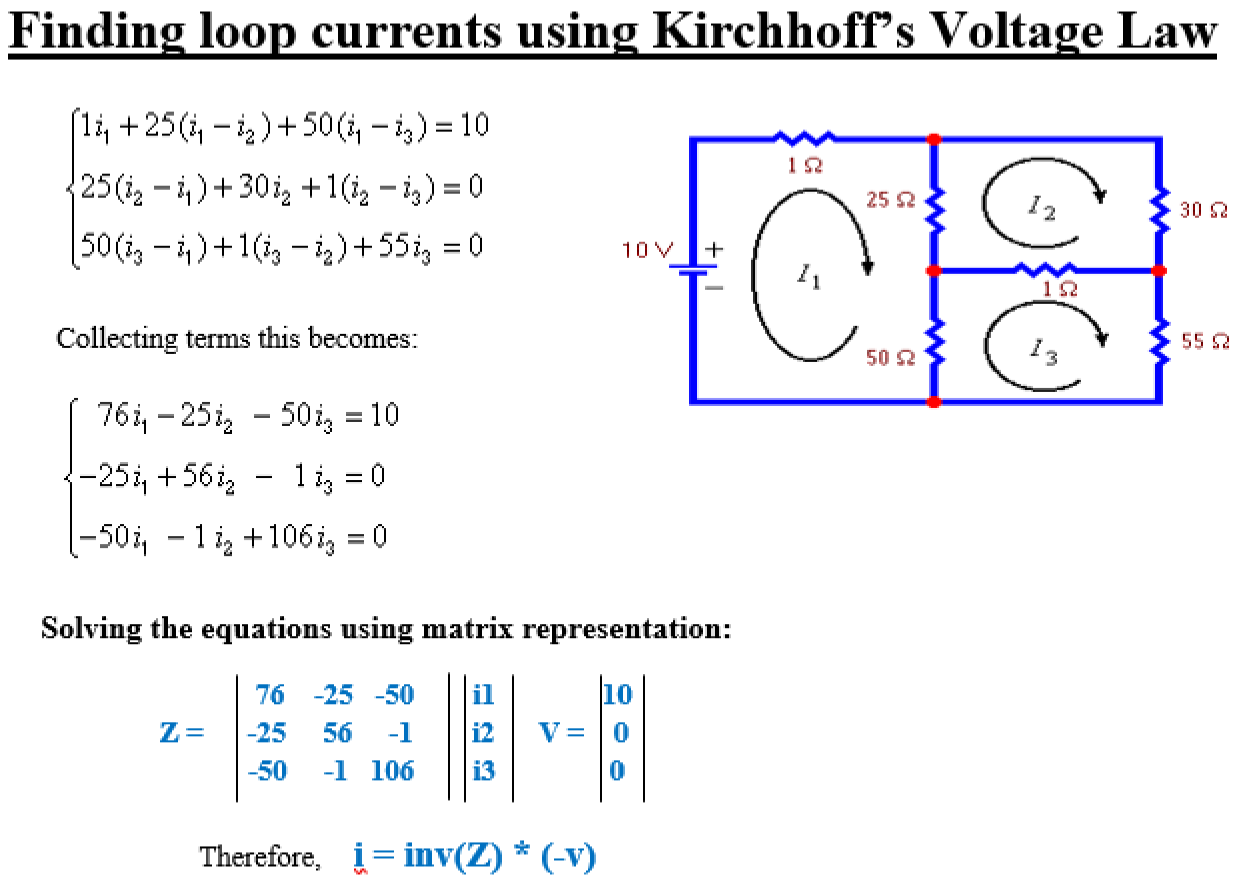
- A major design issue with using caches in a massively parallel processor is ***cache coherence***, which arises when one or more processor cores modify cached data.
- A ***cache coherence mechanism*** is needed to ensure that the contents of the caches of the other processor cores are updated.

Sequential Sparse-Matrix Vector Multiplication (SpVM)

- In a ***sparse matrix***, the **vast majority of the elements are zeros**. Storing and processing these zero elements are wasteful in terms of memory, time, and energy.

- Due to the importance of sparse matrices, several sparse matrix storage formats and their corresponding processing methods have been proposed and widely used in the field.

- Matrices are often used to represent the coefficients in a linear system of equations.



Finding loop currents using Kirchhoff's Voltage Law

$$\begin{cases} 1i_1 + 25(i_1 - i_2) + 50(i_1 - i_3) = 10 \\ 25(i_2 - i_1) + 30i_2 + 1(i_2 - i_3) = 0 \\ 50(i_3 - i_1) + 1(i_3 - i_2) + 55i_3 = 0 \end{cases}$$

1 Ω
25 Ω
30 Ω
10 V
+
–
I_1
I_2
I_3
1 Ω
50 Ω
55 Ω

Collecting terms this becomes:

$$\begin{cases} 76i_1 - 25i_2 \; - 50i_3 = 10 \\ -25i_1 + 56i_2 \; - \; 1\,i_3 = 0 \\ -50i_1 \; - 1\,i_2 + 106i_3 = 0 \end{cases}$$

Solving the equations using matrix representation:

$$Z = \begin{vmatrix} 76 & -25 & -50 \\ -25 & 56 & -1 \\ -50 & -1 & 106 \end{vmatrix} \begin{vmatrix} i1 \\ i2 \\ i3 \end{vmatrix} \quad V = \begin{vmatrix} 10 \\ 0 \\ 0 \end{vmatrix}$$

Therefore, $i = inv(Z) * (-v)$

Sequential Sparse-Matrix Vector Multiplication (SpVM)

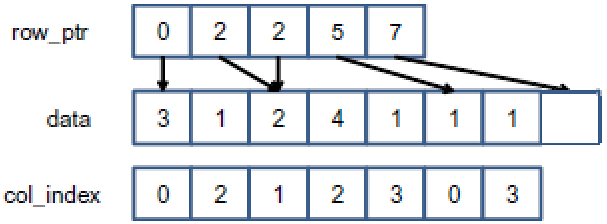
- In many science and engineering problems, there are a large number of variables and the equations involved are **loosely coupled**. That is, each equation only involves a small number of variables.

Row 0	3	0	1	0
Row 1	0	0	0	0
Row 2	0	2	4	1
Row 3	1	0	0	1

The variables x_0 and x_2 are involved in equation 0, none of the variables in equation 1, variables x_1, x_2, and x_3 in equation 2, and finally variables x0 and x3 in equation 3.

- **Compressed Sparse Row (CSR)** storage format is used to avoid storing zero elements of a sparse matrix.

	Row 0	Row 2	Row 3
Nonzero values data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers row_ptr[5]	{ 0, 2, 2, 5, 7 }		

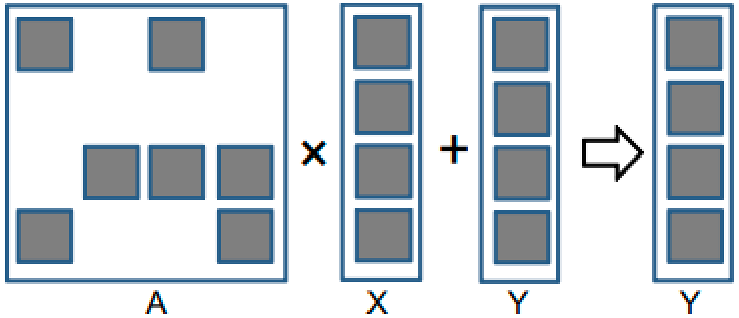


row_ptr	0	2	2	5	7			
data	3	1	2	4	1	1	1	
col_index	0	2	1	2	3	0	3	

- CSR stores only nonzero values of consecutive rows in a 1D data storage named **data[ ].** This format compresses away all zero elements.
- The two sets of markers **col_index[ ]** and **row_ptr[ ]** preserve the structure of the original sparse matrix.
- The marker **col_index[ ]** gives the column index of every nonzero value in the original sparse matrix.
- The marker **row_ptr[ ]** gives the starting location of every row in the **data[ ]** array of the compressed storage. Note that **row_ptr[4]** stores the starting location of a **non-existing row-4** as **7**. This is for convenience, as some algorithms need to use the starting location of the next row to delineate the end of the current row. This extra marker gives a convenient way to locate the ending location of row 3.

Sequential Sparse-Matrix Vector Multiplication (SpVM)

- A sequential implementation of SpMV based on CSR is quite straightforward. We assume that the code has access to the following:
  1. The **num_rows**, a function argument that specifies the number of rows in the sparse matrix.
  2. A floating-point **data[ ]** array and three integer **row_ptr[ ], col_index[ ],** and **x[]** arrays.



```
1.    for (introw = 0; row < num_rows; row++) {
2.      float dot = 0;
3.      int row_start = row_ptr[row];
4.      int row_end =   row_ptr[row+1];

5.      for (intelem = row_start; elem < row_end; elem++) {
6.          dot += data[elem] * x[col_index[elem]];
        }

7.      y[row] += dot;

    }
```

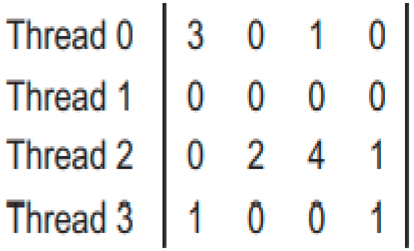
		Row 0		Row 2			Row 3		
Nonzero values data[7]	{	3,	1,	2,	4,	1,	1,	1	}
Column indices col_index[7]	{	0,	2,	1,	2,	3,	0,	3	}
Row Pointers row_ptr[5]	{	0,	2,	2,	5,	7	}		

1. **Line 1** is a loop that iterates through all rows of the matrix, with each iteration calculating a dot product of the current row and the vector *X*.
2. In each row, **Line 2** first initializes the dot product to zero.
3. **Line 3** and **Line 4** sets up the range of *data[ ]* array elements that belong to the current row.
4. **Line 5** is a loop that fetches the elements from the sparse matrix *A* and the vector *X.*
5. The loop body in **Line 6** calculates the dot product for the current row.
6. **Line 7** adda the dot product with the corresponding element of the vector *Y*.

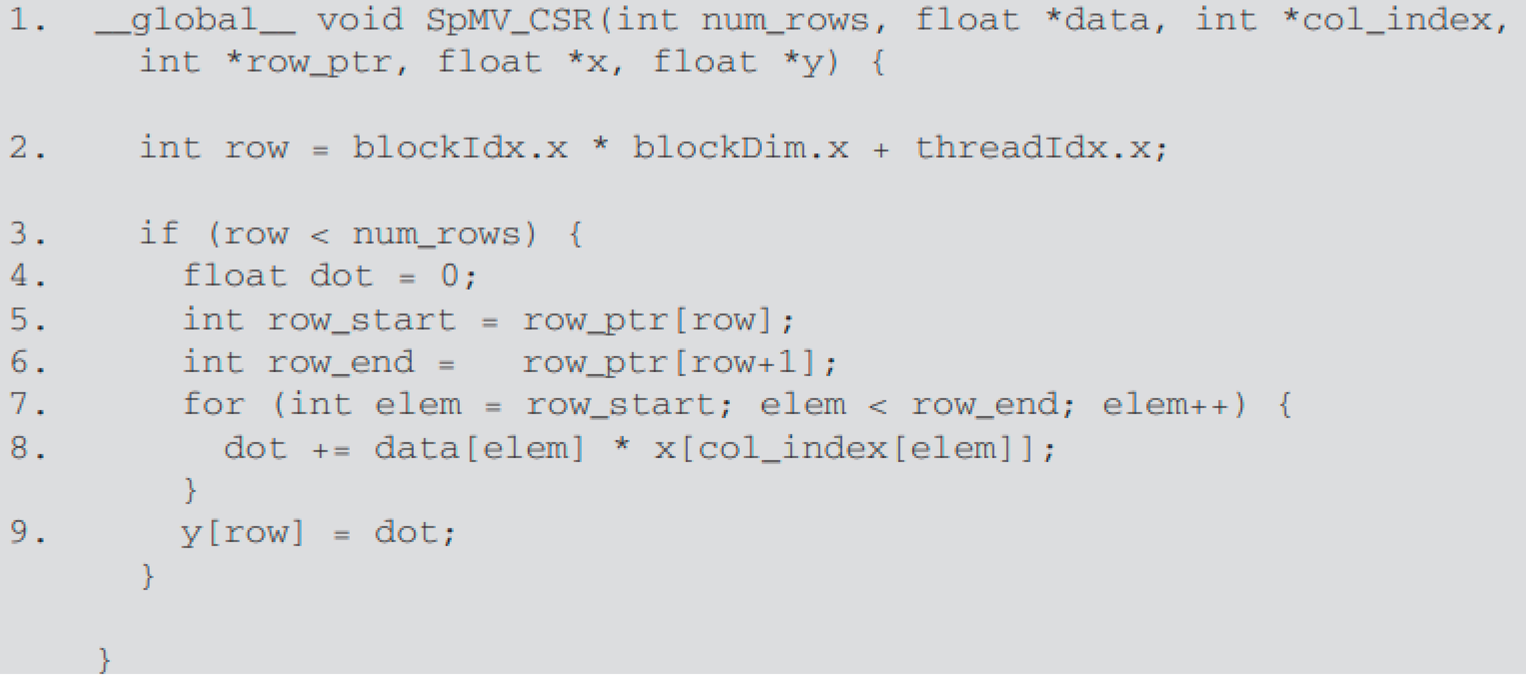
Parallel SpMV using CSR

- In a sequential implementation of SpMV the dot product calculation for each row of the sparse matrix is independent of those of other rows.

- We can easily convert this sequential SpMV/CSR into a parallel CUDA kernel by **assigning each iteration of the outer loop to a thread.**



$$\begin{array}{l|cccc|}
\text{Thread 0} & 3 & 0 & 1 & 0 \\
\text{Thread 1} & 0 & 0 & 0 & 0 \\
\text{Thread 2} & 0 & 2 & 4 & 1 \\
\text{Thread 3} & 1 & 0 & 0 & 1
\end{array}$$



```
1.  __global__ void SpMV_CSR(int num_rows, float *data, int *col_index,
      int *row_ptr, float *x, float *y) {

2.    int row = blockIdx.x * blockDim.x + threadIdx.x;

3.    if (row < num_rows) {
4.      float dot = 0;
5.      int row_start = row_ptr[row];
6.      int row_end =   row_ptr[row+1];
7.      for (int elem = row_start; elem < row_end; elem++) {
8.        dot += data[elem] * x[col_index[elem]];
        }
9.      y[row] = dot;
      }

    }
```

- The loop construct has been removed since it is replaced by the thread grid.

- The row index is calculated as the familiar expression *blockIdx.x * blockDim.x + threadIdx.x*

- Line 3 checks if the row index of a thread exceeds the number of rows.