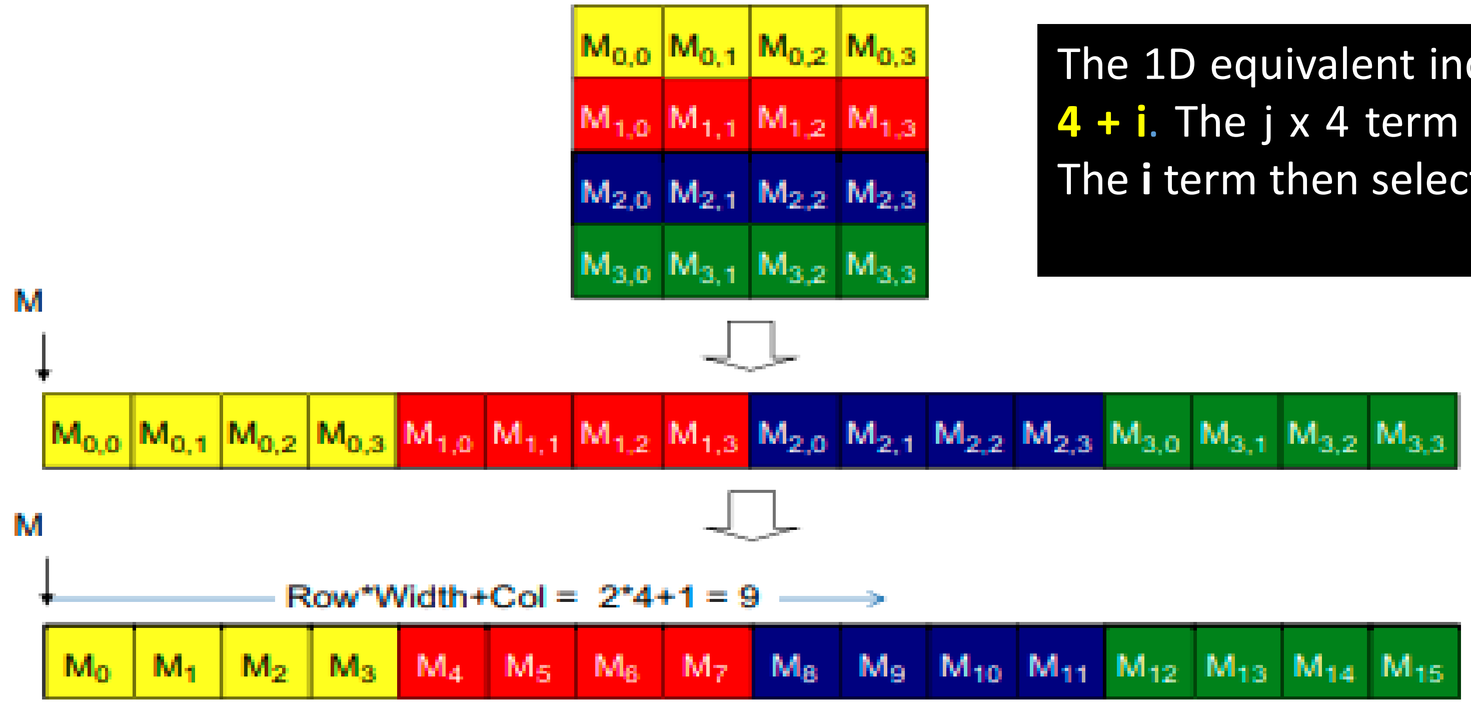
Mapping Threads to Multidimensional Data - Flattening a 2D-array

- Ideally, we would like to access **d_Pin** as a 2D array where an element at row **j** and column **i** can be accessed as **d_Pin[j][i]**.
- However, the ANSI C standard based on which CUDA C was developed requires that the number of columns in d_Pin be **known at compile time**.
- **Unfortunately**, this information is not known at compiler time for dynamically allocated arrays.
- As a result, programmers need to explicitly **linearize**, or **"flatten,"** a dynamically allocated 2D array into an equivalent 1D array in the current CUDA C.
- In reality, all multidimensional arrays in C are linearized.
- There are at least two ways one can linearize a 2D array:
  1) row-major layout
  2) column-major layout

Mapping Threads to Multidimensional Data - Flattening a 2D-array

1. **row-major layout**
   - Here we place all elements of the same row into consecutive locations. The rows are then placed one after another into the memory space.
   - **$M_{j,i}$** denote an **M** element at the **j**th row and the **i**th column.

The 1D equivalent index for the **M** element in row **j** and column **i** is **j x 4 + i**. The j x 4 term skips over all elements of the rows before row **j**. The **i** term then selects the right element within the section for row **j**.



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
M
Row*Width+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression `Row*Width + Col` for an element that is in the `Row`th row and `Col`th column of an array of `Width` elements in each row.

Mapping Threads to Multidimensional Data - Flattening a 2D-array

2. <u>column-major layout</u>

- Here we place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space.

- This is used by FORTRAN compilers.

- The column-major layout of a 2D array is equivalent to the row-major layout of its transposed form.

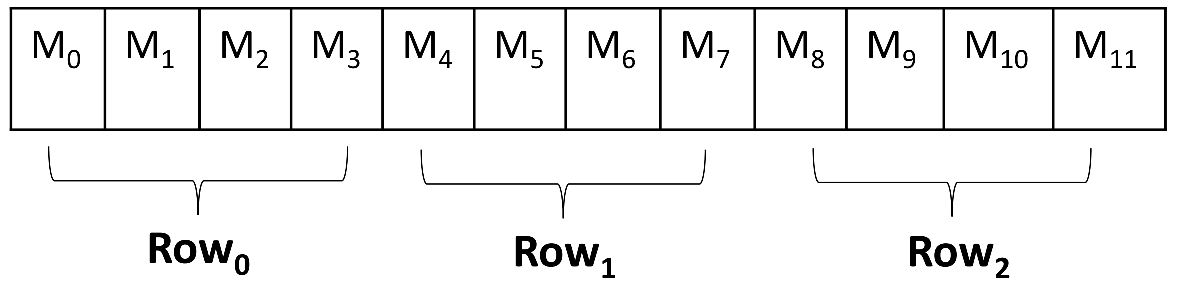
Matrix-Matrix Multiplication – three variants

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

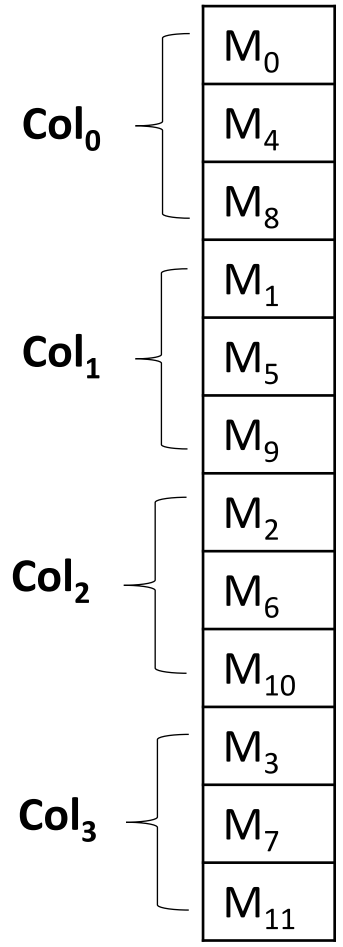
1. Each row of resultant matrix to be computed by one thread.
2. Each column of resultant matrix to be computed by one thread.
3. Each element of resultant matrix to be computed by one thread.

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}

Each row of resultant matrix to be computed by one thread

a(ha*wa) *   b(hb*wb)   c (ha*wb)

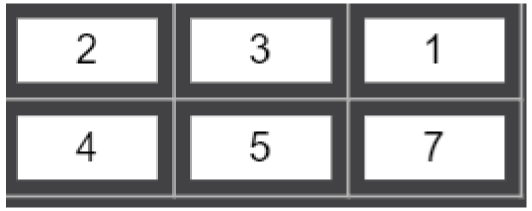
wa should be equal to hb

How many threads=no. of rows of resultant matrix=ha

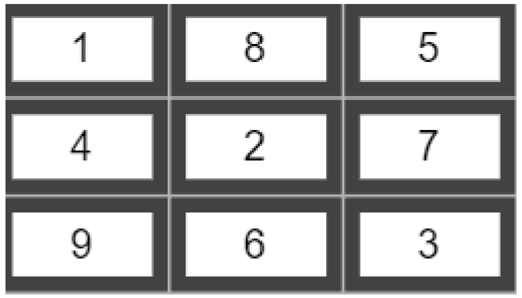
matmul_rowwise<<<1,ha>>>(a,b,c,wa,wb);

how many for loops in the kernel? 2

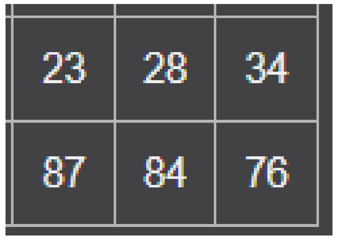
2X3 3*3 2X 3



2	3	1
4	5	7



1	8	5
4	2	7
9	6	3



23	28	34
87	84	76

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;
         }
}
```

Each col of a resultant matrix to be computed by one thread

a(ha*wa) * b(hb*wb) c (ha*wb)

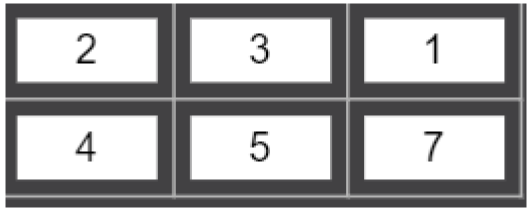
wa should be equal to hb

How many threads=no. of cols of resultant matrix=wb

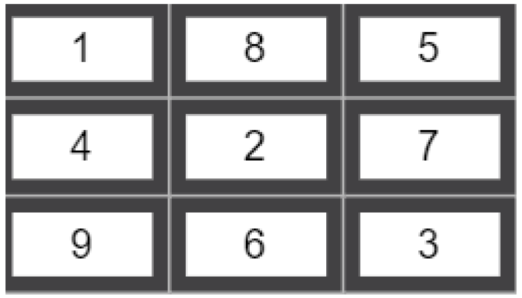
matmul_colwise<<<1,wb>>>(a,b,c,ha,wa);

how many for loops in the kernel? 2

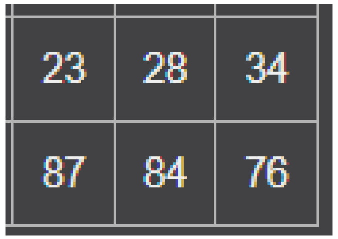
2X3 3*3 2X 3



2	3	1
4	5	7



1	8	5
4	2	7
9	6	3



23	28	34
87	84	76

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;
          }
}
```

Each element of resultant matrix to be computed by one thread

a(ha*wa) * b(hb*wb) c (ha*wb)

wa should be equal to hb

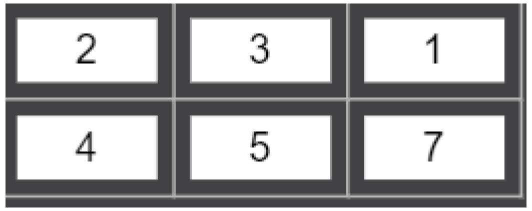
How many threads=no. of elements of resultant matrix=ha*wb

Let’s use 1D grid, 2D block

matmul_rowwise<<<1,(,)>>>(a,b,c,wa);

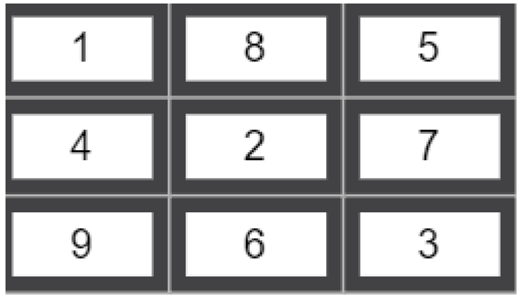
how many for loops in the kernel? 1

2X3



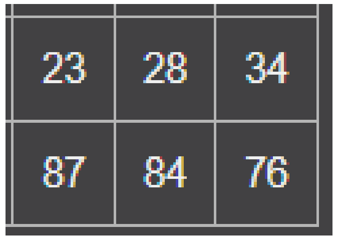
2	3	1
4	5	7

3*3



1	8	5
4	2	7
9	6	3

2X 3



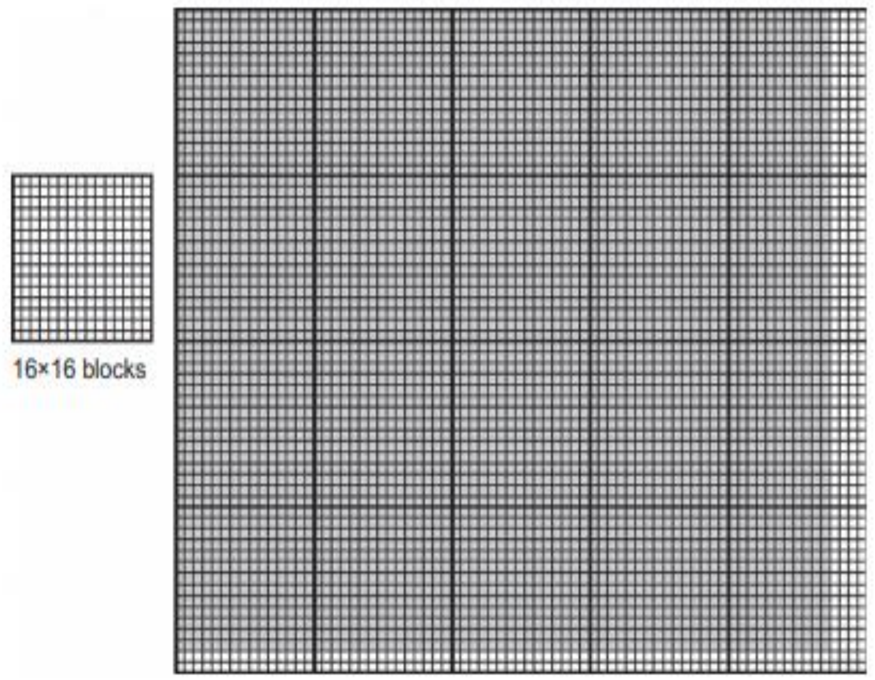
23	28	34
87	84	76

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;

}
```

Mapping Threads to Multidimensional Data

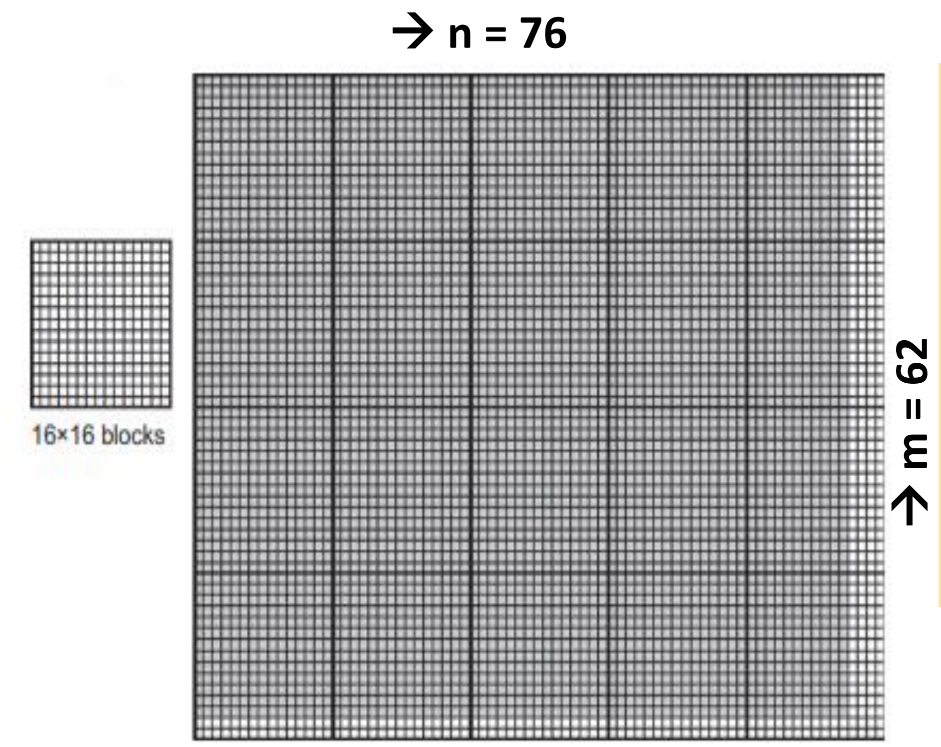
- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data.
- For example, pictures are a 2D array of pixels. It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture.



- It is a **76x62** picture.
- Assume that we decided to use a **16x16 block**, with 16 threads in the x-direction and 16 threads in the y-direction.
- We will need **five blocks** in the x-direction and **four blocks** in the y-direction, which results in **5x4=20 block**.

- In this picture example, we have **four extra threads** in the **x-direction** and **two extra threads** in the **y-direction**. That is, we will generate **80x64** threads to process **76x62** pixels.
- The picture processing kernel function will have **if statements** to test whether the thread indices **threadIdx.x** and **threadIdx.y** fall within the valid range of pixels.

Mapping Threads to Multidimensional Data



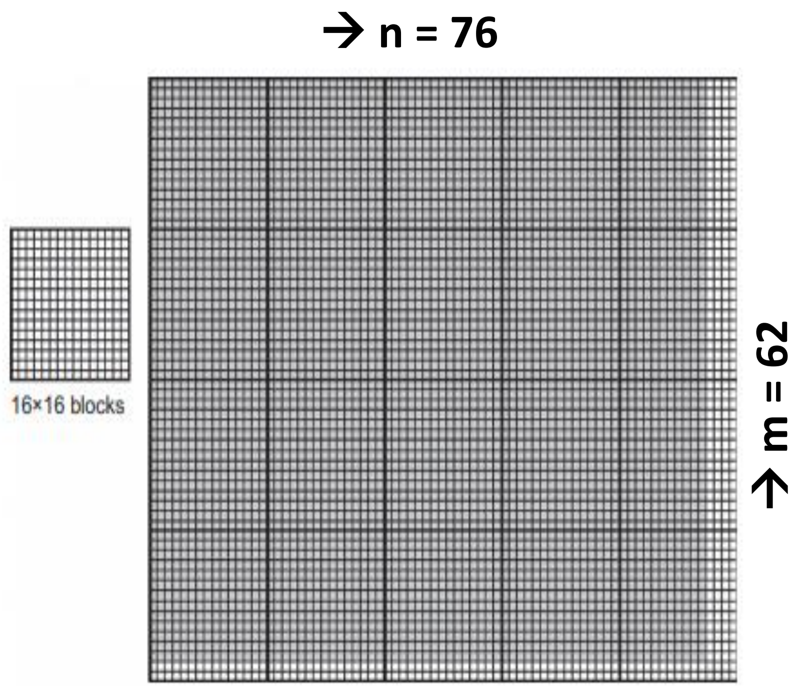
- Assume that the host code uses an integer variable **n** to track the number of pixels in the **x-direction**, and another integer variable **m** to track the number of pixels in the **y-direction**.
- Assume that the input picture data has been copied to the device memory and can be accessed through a pointer variable **d_Pin**. The output picture has been allocated in the device memory and can be accessed through a pointer variable **d_Pout**.

```
dim3 dimGrid (ceil(n/16.0), ceil(m/16.0), 1);
dim3 dimBlock(16, 16, 1);
vecAddKernel <<< dimGrid, dimBlock >>> (d_Pin, d_Pout, n, m);
```

- Within the kernel function, references to built-in variables gridDim.x, gridDim.y, blockDim.x, and blockDim.y will result in 5, 4, 16, and 16, respectively.

Mapping Threads to Multidimensional Data – the pictureKernel()

```
__global__ void pictureKernell(float* d_Pin, float* d_Pout, int n, int m)
{             // Calculate the row # of the d_Pin and d_Pout element to process
              int Row = blockIdx.y*blockDim.y + threadIdx.y;
              // Calculate the column # of the d_Pin and d_Pout element to process
              int Col = blockIdx.x*blockDim.x + threadIdx.x;
              // each thread computes one element of d_Pout if in range
              if ((Row < m) && (Col < n))
              {
                            d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
              }
}
```



- This kernel will scale every pixel value in the picture by a factor of 2.0.

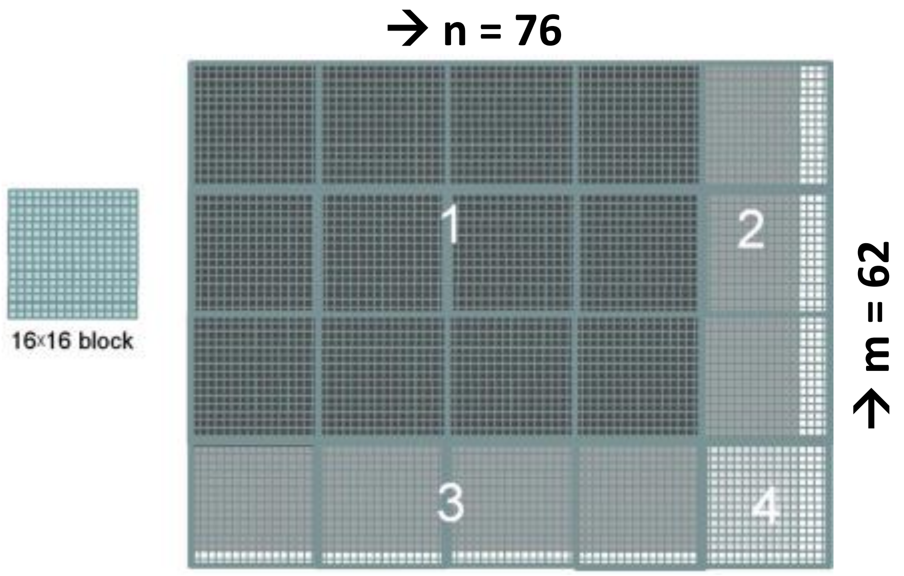
- There are a total of **blockDim.x * gridDim.x** threads in the horizontal direction
- and **blockDim.y * gridDim.y** threads in the vertical direction.

- The expression **Col=blockIdx.x*blockDim.x+threadIdx.x** generates every integer value from **0** to **blockDim.x*gridDim.x-1**.

- The condition **(Col < n) && (Row < m)** make sure that only the threads in proper range are executed.

```
dim3 dimGrid (ceil(n/16.0), ceil(m/16.0), 1); // 5,4,1
dim3 dimBlock(16, 16, 1);     vecAddKernel <<< dimGrid, dimBlock >>> (d_Pin, d_Pout, n, m);
```

Mapping Threads to Multidimensional Data – the pictureKernel()



- During the execution, the execution behaviour of blocks will fill into one of four different cases:
  1. The **first area**, marked as *1* consists of the threads that belong to the *12* blocks covering the majority of pixels in the picture. Both Col and Row values of these threads are within range (76).

  2. The **second area**, marked as *2* contains the threads that belong to the *3* blocks covering the upper-right pixels of the picture. Although the Row values of these threads are always within range, the Col values of some of them exceed the n value.

  3. The **third area**, marked as *3* contains the threads that belong to the *4* blocks covering the lower-left pixels of the picture. Although the Col values of these threads are always within range, the Row values of some of them exceed the m value (62).

  4. The **forth area**, marked as *4* contains the threads that belong to *1* block covering the lower-right pixels of the picture. Both Col and Row values exceed the *n* and *m* values.

Matrix-Matrix Multiplication – A More Complex Kernel

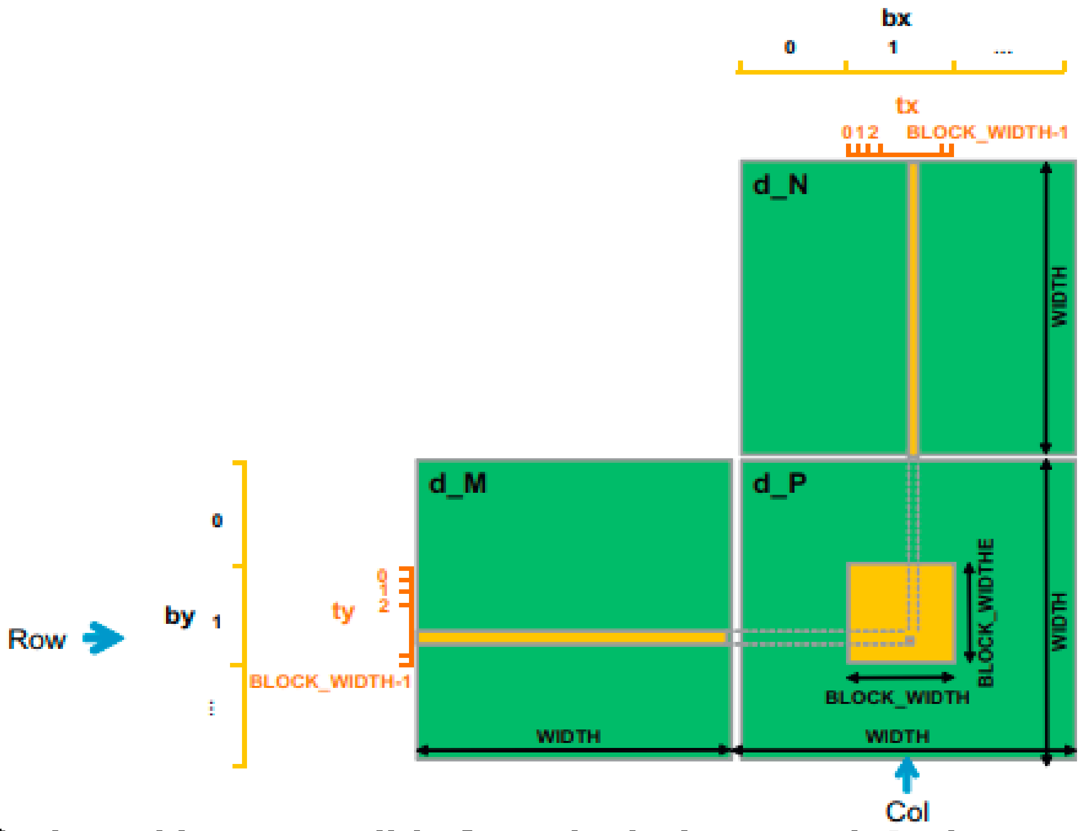
- We can map every valid data element in a 2D array to a unique thread using threadIdx, blockIdx, blockDim, and gridDim variables:

```
// 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– 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_Pelement 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);
```

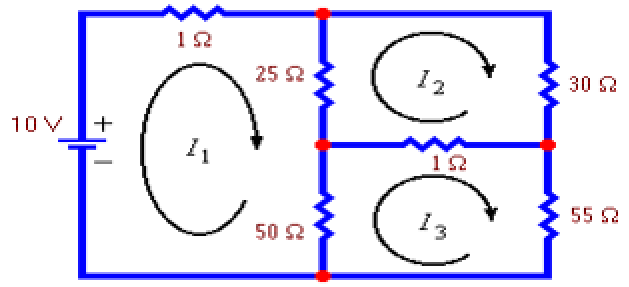
Host code

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

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



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.

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

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.

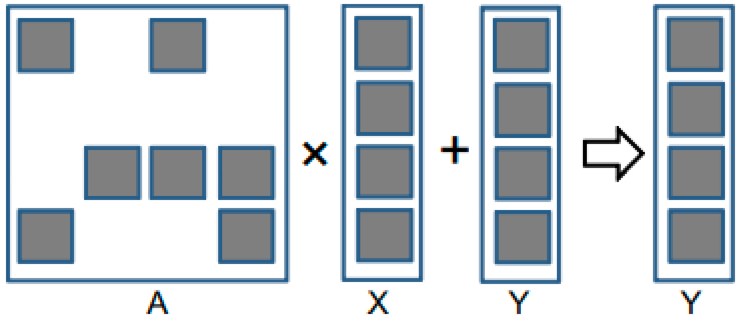
- **Compressed Sparse Row (CSR)** storage format is used to avoid storing zero elements of a sparse matrix.

		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 }			

- 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.

Importance of Memory Access Efficiency

- In CUDA programming, the data to be processed by the threads is first transferred from the host memory to the device global memory.
- The threads then access their portion of the data from the global memory using their block IDs and thread IDs.
- The simple CUDA kernels will likely achieve only a small fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that global memory, which is typically implemented with dynamic random access memory (DRAM), tends to have long access latencies (hundreds of clock cycles) and finite access bandwidth.
- In matrix multiplication kernel, the most important part of the kernel in terms of execution time is the ***for*** loop that performs inner product calculation:

```
for (int k = 0; k < Width; ++k)
        Pvalue += d_M[Row * Width + k] * d_N[k * Width + Col];
```

1. **In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition.**
2. One global memory access fetches a **d_M[ ]** element and the other fetches a **d_N[ ]** element.
3. One floating-point operation **multiplies** the **d_M[]** and **d_N[]** elements fetched and the other **accumulates** the product into **Pvalue**.
4. Thus, the ratio of floating-point calculation to global memory access operation is **1:1**, or **1.0**.

Importance of Memory Access Efficiency

- The ***compute to global memory access (CGMA)*** ratio is defined as the number of floating point calculations performed for each access to the global memory within a region of a CUDA program.

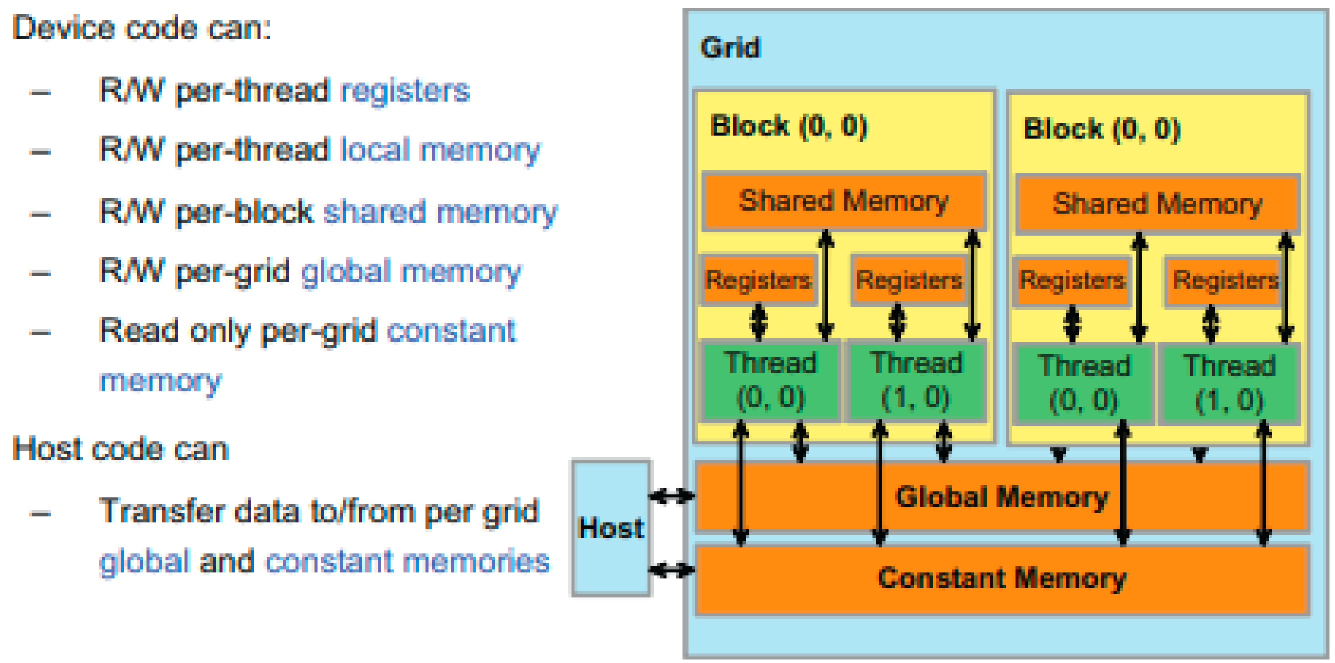
- CGMA has major implications on the performance of a CUDA kernel.

➢ In a high-end device today, the **global memory bandwidth is around 200 GB/s**. With **4 bytes** in each **single-precision floating-point value**, one can expect to load no more than **50 (200/4) giga single-precision operands per second**. With a CGMA ration of **1.0**, the matrix multiplication kernel will execute no more than **50 giga floating-point operations per second (GFLOPS)**.

➢ For the matrix multiplication code to achieve the peak **1,500 GFLOPS** rating of the processor, we need a **CGMA value of 30**.

CUDA Device Memory Types

- CUDA supports **several types of memory** that can be used by programmers to achieve a **high CGMA ratio** and thus a high execution speed in their kernels.



Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories

- At the bottom of the figure, we see **global memory** and **constant memory**. These types of memory can be **written (W)** and **read (R)** by the **host** by calling API functions.

- The **constant memory** supports *short-latency*, *high-bandwidth*, *read-only* access by the device when all threads simultaneously access the same location.

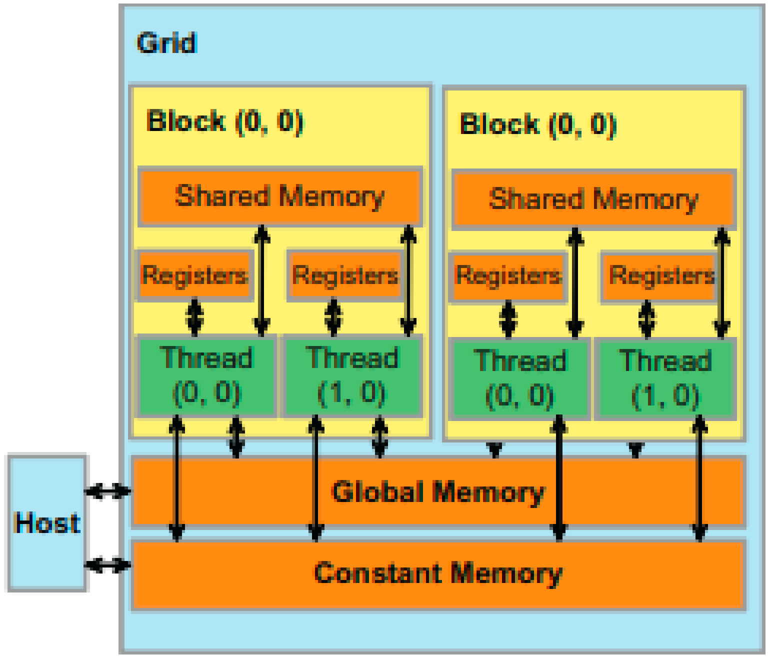
CUDA Device Memory Types

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



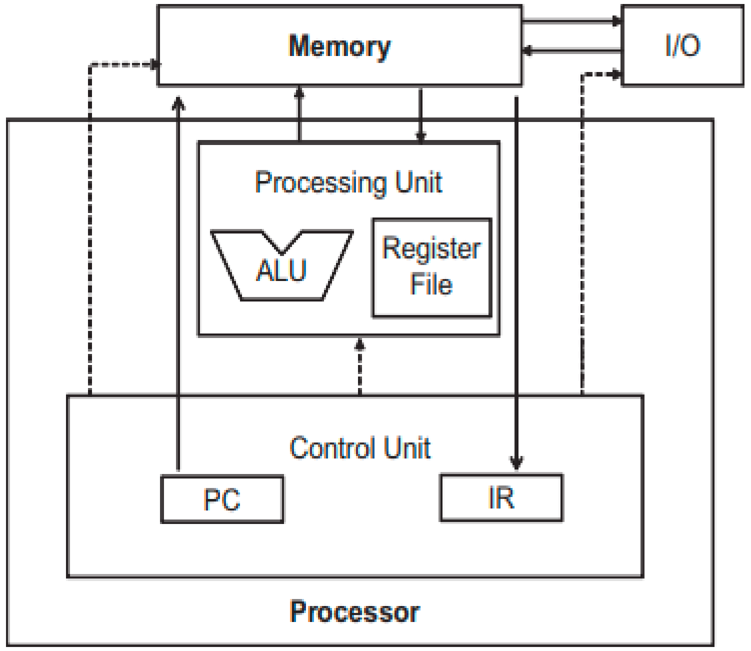
- **Registers** and **shared memory** are **on-chip memories**. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner.

- **Registers** are allocated to individual threads; each thread can only access its own registers. A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

- **Shared memory** is allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block. Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work.

CUDA Device Memory Types

- The **global memory** in the CUDA programming model **maps** to the memory of the **von Neumann model**.



- The **processor box** corresponds to the processor chip boundary.
- The **global memory** is off the processor chip and is implemented with DRAM technology, which implies long access latencies and relatively low access bandwidth.
- The registers correspond to the **"register file"** of the von Neumann model. It is on the processor chip, which implies very short access latency and drastically higher access bandwidth.
- Whenever a **variable is stored in a register**, its accesses no longer consume off-chip global memory bandwidth. This will be reflected as an increase in the CGMA ratio.
- The processor uses the PC value to fetch instructions from memory into the IR.

- **Arithmetic instructions** in most modern processors have "built-in" register operands:

  fadd r1, r2, r3 → where r2 and r3 are the register numbers that specify the location in the register file where the input operand values can be found. The location for storing the floating-point addition result value is specified by r1.

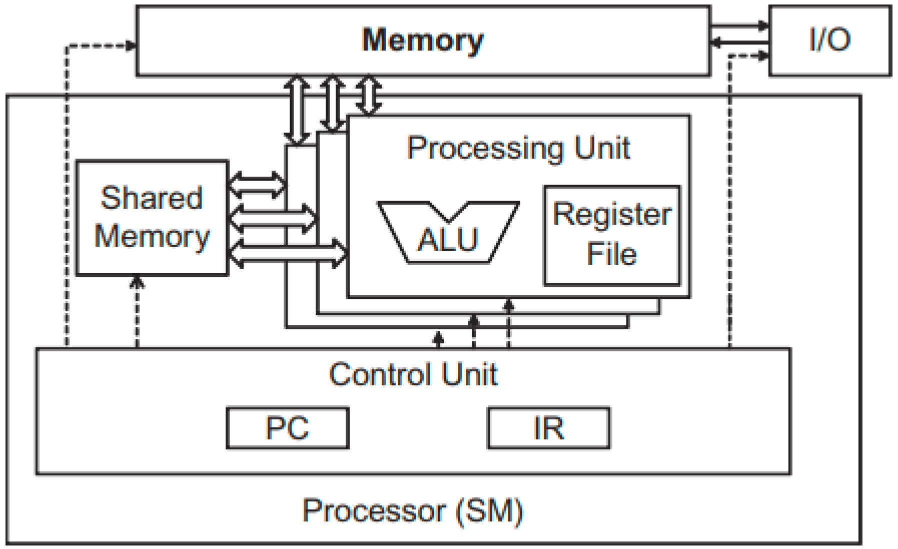
- When an operand of an arithmetic instruction is in a register, **there is no additional instruction required** to make the operand value available to the arithmetic and logic unit (ALU) where the arithmetic calculation is done.

- On the other hand, **if an operand value is in global memory**, one needs to perform a **memory load operation** to make the operand value available to the ALU. For example, if the first operand of a floating-point addition instruction is in global memory of a typical computer today, the instructions involved will likely be:

  load r2, r4, offset → where the load instruction adds an offset value to the contents of r4 to form an address for the operand value. It then accesses the global memory and places the value into register r2.
  fadd r1, r2, r3

CUDA Device Memory Types

- The following figure shows **shared memory and registers in a CUDA device**:



- Although both are on-chip memories, they differ significantly in functionality and cost of access.
- When the processor accesses data that resides in the **shared memory**, it needs to perform **a memory load operation**, just like accessing data in the global memory.
- However, because shared memory resides on-chip, it can be accessed with much **lower latency** and **much higher bandwidth** than the global memory.
- Because of the need to perform a load operation, share memory has longer latency and lower bandwidth than **registers**.
- In computer architecture, shared memory is a form of ***scratchpad memory***.

- One important difference between the share memory and registers in CUDA is that variables that reside in the **shared memory** are **accessible by all threads in a block**. This is in contrast to register data, which is **private to a thread**.

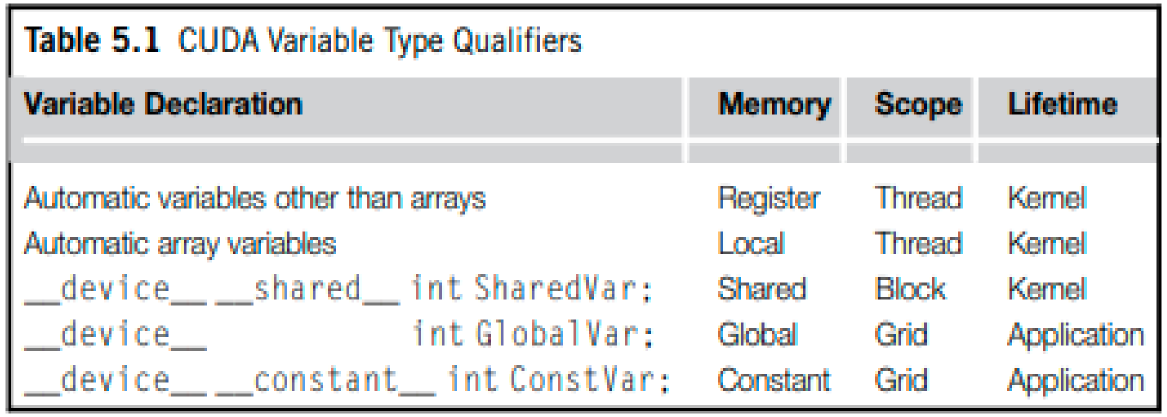
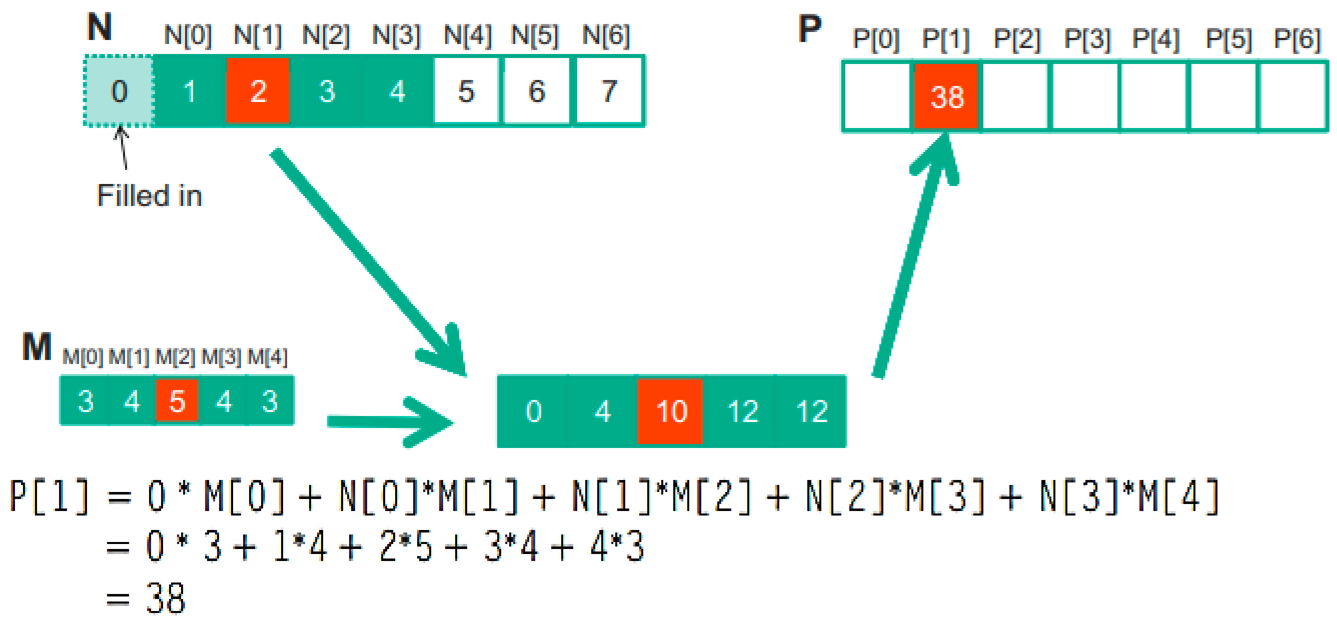


Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

- **Scope** identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of all grids.
- **Lifetime** tells the portion of the program's execution duration when the variable is available for use: either within a kernel's execution or throughout the entire application.
- **Automatic array variables** are not stored in registers. Instead, they are stored into the global memory and may incur long access delays and potential access congestions. The scope of these arrays is, like automatic scalar variables, limited to individual threads.

Constant Memory and Caching



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

```
__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 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.

Constant Memory and Caching

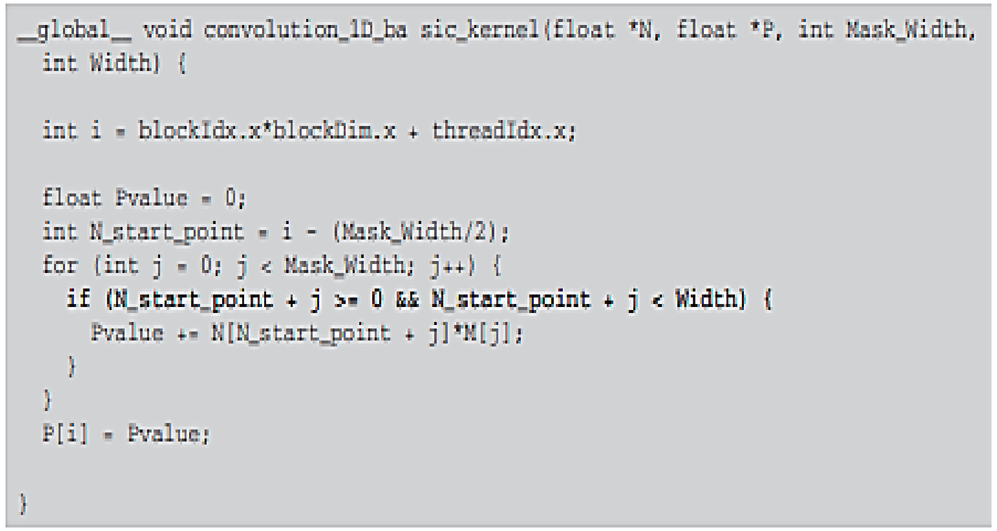
- Assume that the host code has already allocated and initialized the mask in a mask ***h_M*** array in the host memory with ***Mask_Width*** elements. The contents of the ***h_M*** can be transferred to ***M*** in the device constant memory as follows: **cudaMemcpyToSymbol(M, h_M, Mask_Width * sizeof(float));**

  ** 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.



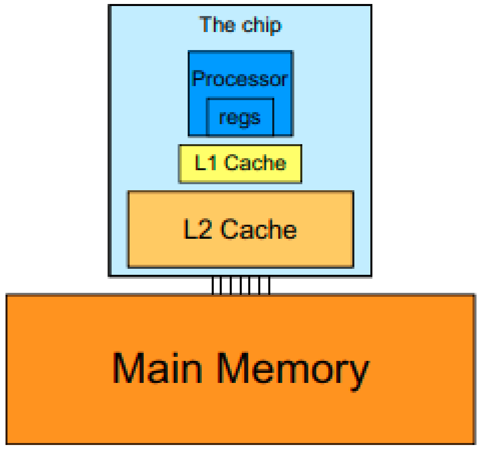
```
__global__ void convolution_1D_ba sic_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  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.



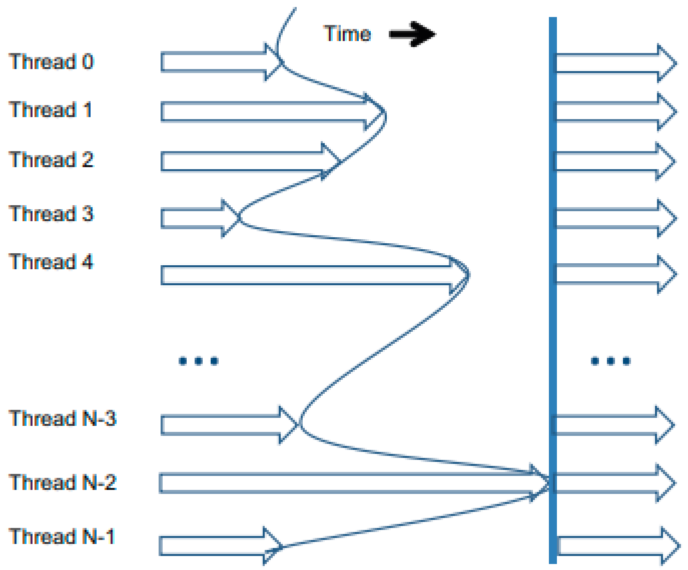
- A major design issue with using caches in a massively parallel processor is ***cache coherence***, which arises when one or more processor cores modify cached data.
- A ***cache coherence mechanism*** is needed to ensure that the contents of the caches of the other processor cores are updated.

Synchronization and Transparent Scalability

- CUDA allows threads in the same block to coordinate their activities using a **barrier synchronization function __syncthreads()**.

- When a kernel function calls __syncthreads(), *all threads in a block will be held at the calling location until every thread in the block reaches the location*.

- This ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase.



- In CUDA, a __syncthreads() statement, if present, **must be executed by all threads in a block**.

Synchronization and Transparent Scalability

- When a __syncthread() statement is placed in an **if**-statement, either all threads in a block execute the path that includes the __syncthreads() or none of them does.

- For an **if-then-else** statement, if each path has a __syncthreads() statement, either all threads in a block execute the __syncthreads() on the **then path** or all of them execute the **else path**.

- The two __syncthreads() are different barrier synchronization points. If a thread in a block executes the **then path** and another executes the else path, they would be waiting at different barrier synchronization points. They would end up waiting for each other forever.

- It is the responsibility of the programmers to write their code so that these requirements are satisfied.

- The ability to synchronize also imposes execution constraints on threads within a block. **These threads should execute in close time proximity with each other** to avoid excessively long waiting times.

Synchronization and Transparent Scalability

- In fact, one needs to make sure that all threads involved in the barrier synchronization **have access to the necessary resources** to eventually arrive at the barrier. Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to **wait forever**.

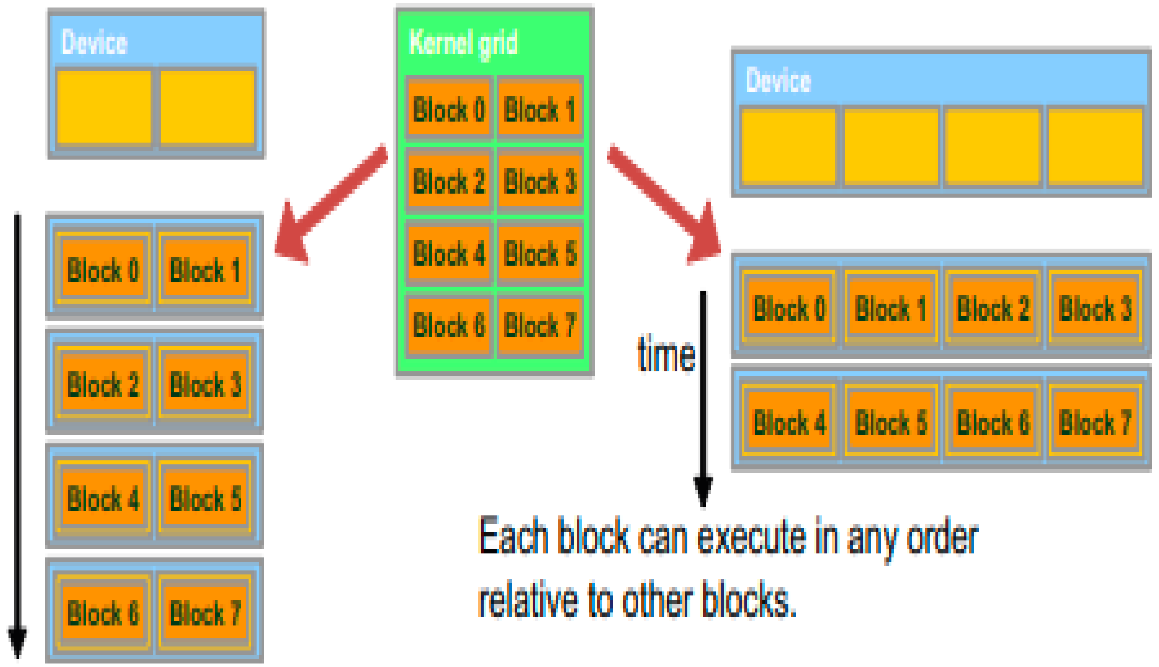
- CUDA runtime systems satisfy this constraint by **assigning execution resources to all threads in a block as a unit**. A block can begin execution only when the runtime system has secured all the resources needed for all threads in the block to complete execution.

- When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization

- By not allowing threads in different blocks to perform barrier synchronization with each other, the **CUDA runtime system can execute blocks in any order relative to each other** since none of them need to wait for each other.

Synchronization and Transparent Scalability

- In a low-cost system with only a few execution resources, one can execute a small number of blocks at the same time. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time.

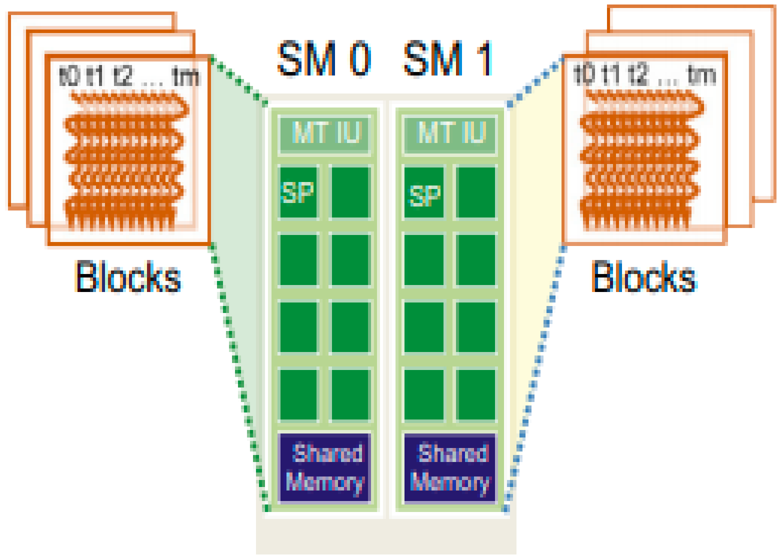


- The ability to execute the same application code on hardware with a different number of execution resources is referred to as ***transparent scalability***, which reduces the burden on application developers and improves the usability of applications.

Assigning Resources to Blocks

- Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. These threads are **assigned to execution resources on a block-by-block basis**.

- In the current generation of hardware, **the execution resources are organized into streaming multiprocessors (SMs)**.



- Multiple thread blocks can be assigned to each SM.

- Each device has a limit on the number of blocks that can be assigned to each SM. For example, a CUDA device may allow up to eight blocks to be assigned to each SM.

Assigning Resources to Blocks

- In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of eight blocks, the **CUDA runtime automatically reduces the number of blocks assigned to each SM** until their combined resource usage falls under the limit.

- With a limited numbers of SMs and a limited number of blocks that can be assigned to each SM, **there is a limit on the number of blocks that can be actively executing** in a CUDA device.

- Most grids contain many more blocks than this number. The runtime system maintains **a list of blocks** that need to execute and assigns new blocks to SMs as they complete executing the blocks previously assigned to them.

- One of the SM resource limitations is the **number of threads that can be simultaneously tracked and scheduled**.

- It takes hardware resources for SMs to maintain the thread and block indices and track their execution status.

- In more recent CUDA device designs, up to **1,536** threads can be assigned to each SM. This could be in the form of 6 blocks of 256 threads each, 3 blocks of 512 threads each, etc.

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**.

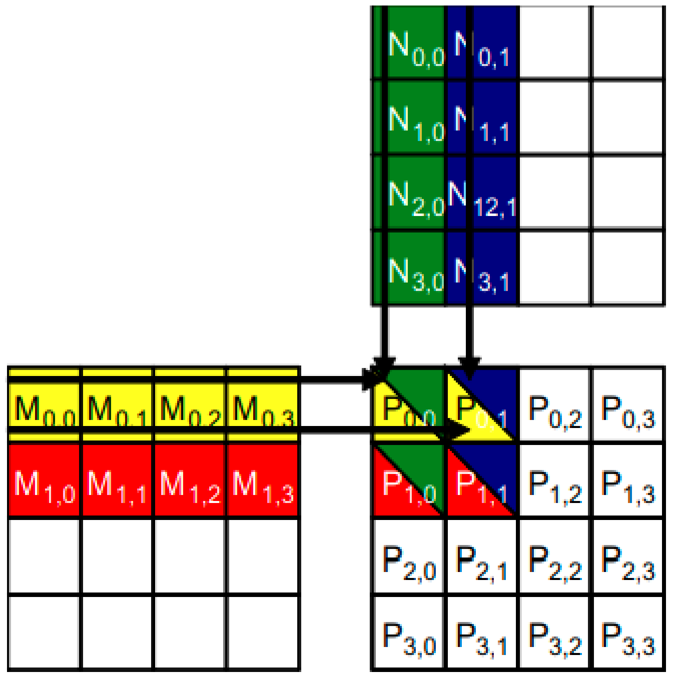
- The clock frequency of the device is in **dev_prop. clockRate**.

- 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).

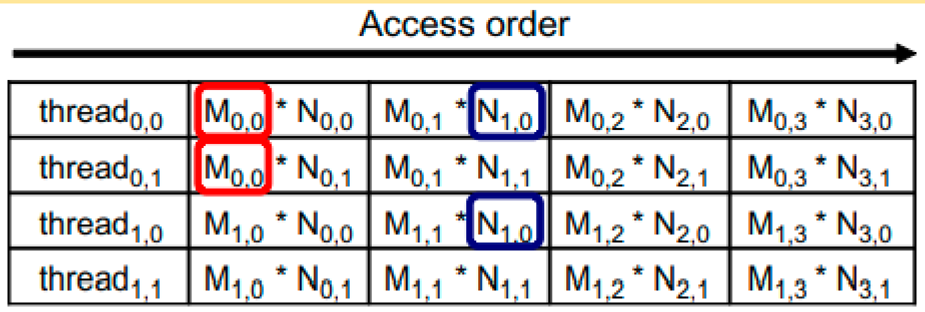
A Strategy for Reducing Global Memory Traffic

- **Global memory** *is large but slow*, whereas the **shared memory** *is small but fast*.
- A common strategy is partition the data into subsets called **tiles** so that **each tile fits into the shared memory**.
- An important criterion is that the kernel computation on these tiles can be done independently of each other.
- Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.
- The concept of tiling can be illustrated with the matrix multiplication example.



$N_{0,0}$
$N_{0,1}$
$N_{1,0}$
$N_{1,1}$
$N_{2,0}$
$N_{2,1}$
$N_{3,0}$
$N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

- Assume that we use four **2 x 2 blocks** to compute the P matrix.
- The figure highlights the computation done by the **four threads of block(0,0)** to compute $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$.



Access order $\longrightarrow$

$\text{thread}_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

- The table shows the **global memory accesses** done by all threads in **block $_{0,0}$**. The threads are listed in the vertical direction, with time of access increasing to the right in the horizontal direction.
- Among the four threads highlighted, there is a **significant overlap** in terms of the M and N elements they access.

A Strategy for Reducing Global Memory Traffic

Access order →

thread$_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread$_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread$_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread$_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, intWidth) {

  // 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;
  }

}
```

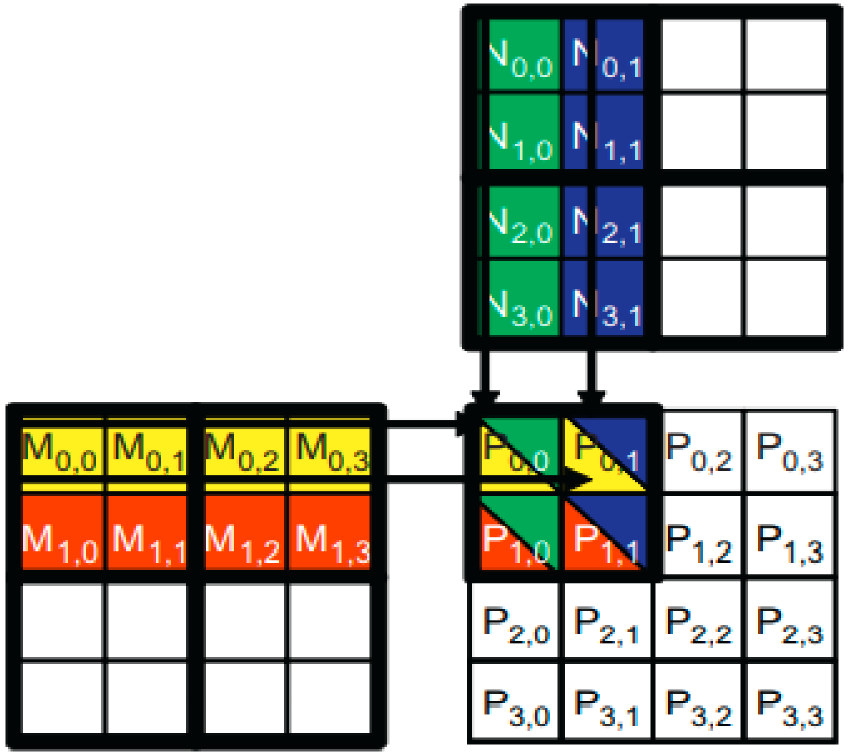
- The kernel is written so that all the threads repeatedly access elements of matrix **M** and **N** from the global memory.
- we can see that every **M** and **N** element is accessed **exactly twice** during the execution of a **block**. Therefore, if we can have all four threads to **collaborate** in their accesses to global memory, we can reduce the traffic to the global memory by half.

A Tiled Matrix-Matrix Multiplication Kernel

- In the design of a kernel for a tiled matrix-matrix multiplication, the basic idea is to have the threads to collaboratively load **M** and **N** matrix elements into the shared memory before they individually use these elements in their dot product calculation.

- Keep in mind that the **size of the shared memory is quite small** and one must be careful not to exceed the capacity of the shared memory when loading these M and N elements into the shared memory.

- This can be accomplished by **dividing the M and N matrices into smaller tiles**. The size of these tiles is chosen so that they can fit into the shared memory. In the simplest form, the tile dimensions equal those of the block:



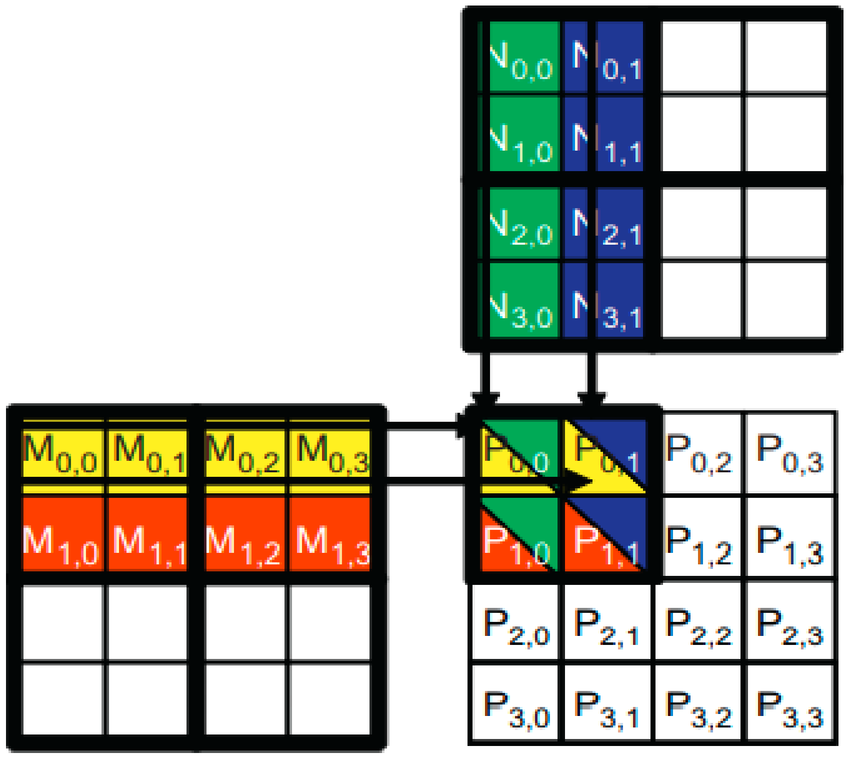
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

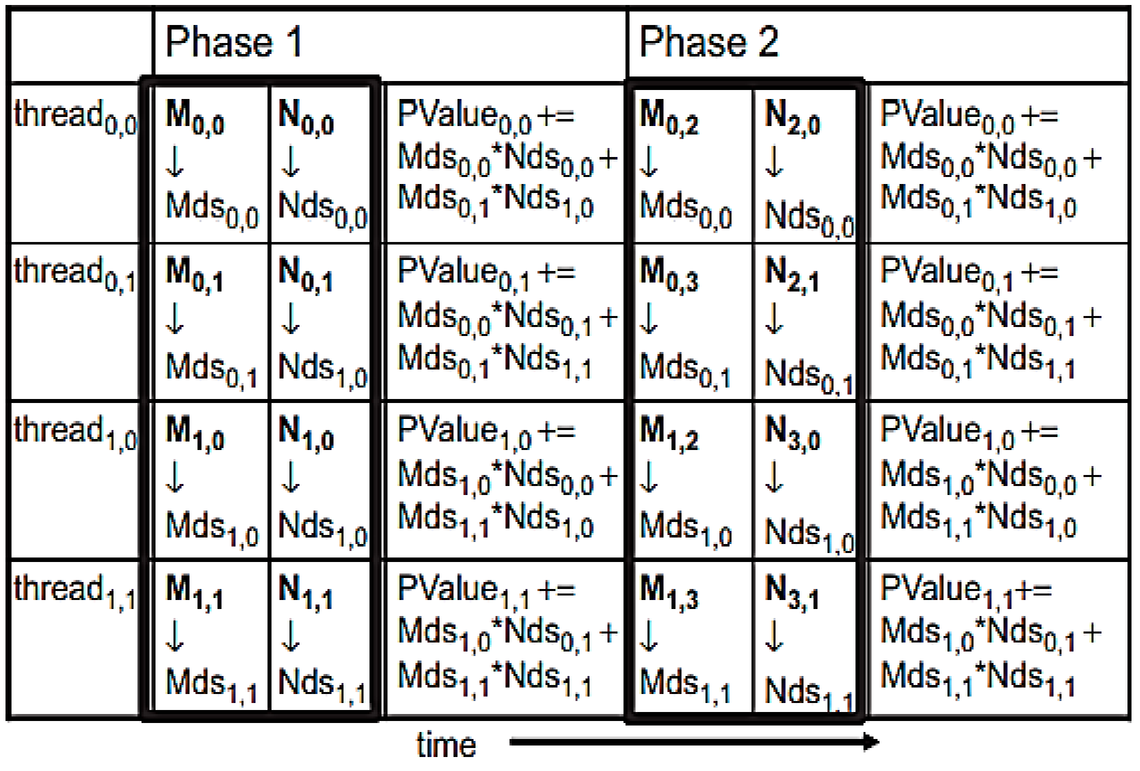
- We divide the **M** and **N** matrices into **2 x 2 tiles**.

- The **dot product calculations** performed by each thread are now divided into **phases**.

- **In each phase**, all threads in a block collaborate to load a tile of **M** elements and a tile of **N** elements into the **shared memory**. This is done by having every thread in a block to load one M element and one N element into the shared memory.

A Tiled Matrix-Matrix Multiplication Kernel


$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

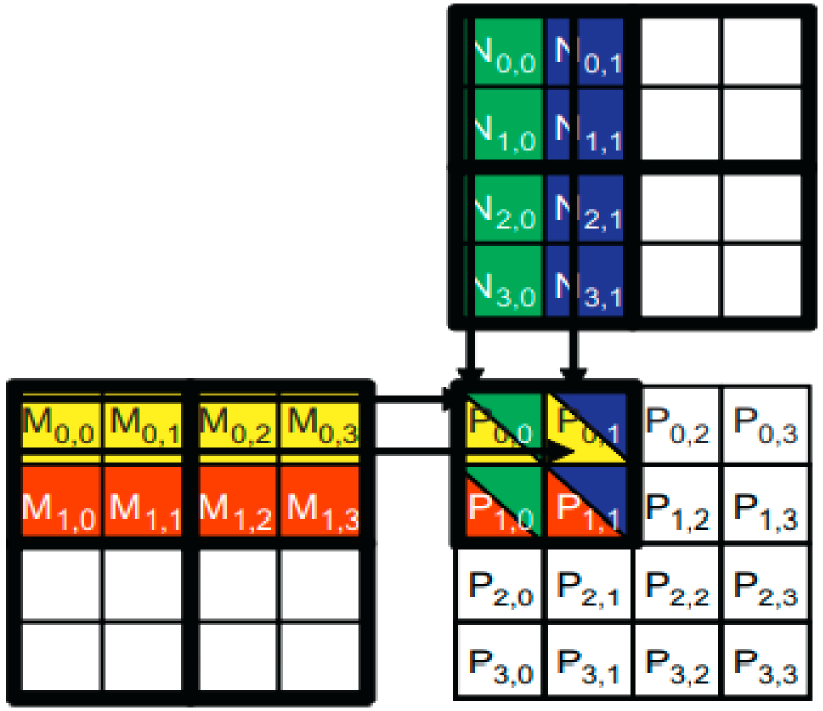


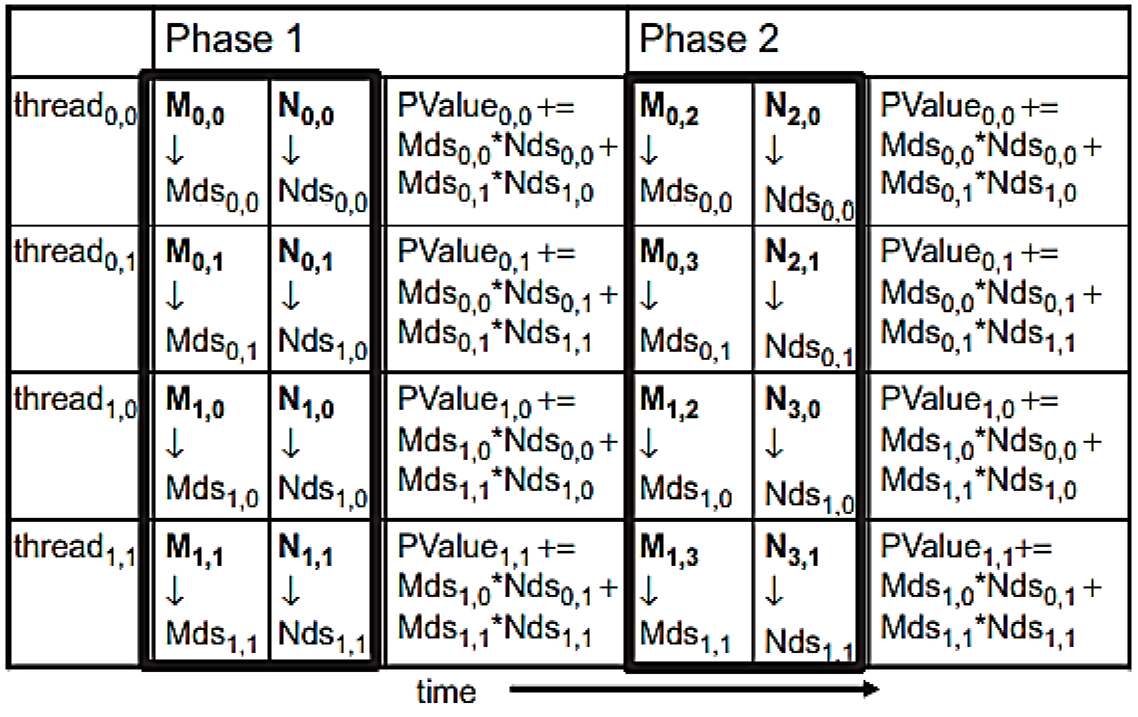
	Phase 1			Phase 2		
thread$_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	**$M_{0,2}$** ↓ $Mds_{0,0}$	**$N_{2,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
thread$_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	**$M_{0,3}$** ↓ $Mds_{0,1}$	**$N_{2,1}$** ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
thread$_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	**$M_{1,2}$** ↓ $Mds_{1,0}$	**$N_{3,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
thread$_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$M_{1,3}$** ↓ $Mds_{1,1}$	**$N_{3,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time $\longrightarrow$

- The table only shows the activities of threads in **block$_{0,0}$**;
- The shared memory array for the **M** and **N** elements is called **Mds** and Nds respectively.
- At the beginning of **phase 1**, the four threads of **block$_{0,0}$** collaboratively load a tile of **M** and a tile of **N** elements into **shared memory.**
- After the two tiles of M and N elements are loaded into the shared memory, these values are used in the calculation of the **dot product**.

A Tiled Matrix-Matrix Multiplication Kernel





	Phase 1			Phase 2		
$\text{thread}_{0,0}$	$M_{0,0}$ ↓ $Mds_{0,0}$	$N_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$M_{0,2}$ ↓ $Mds_{0,0}$	$N_{2,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$\text{thread}_{0,1}$	$M_{0,1}$ ↓ $Mds_{0,1}$	$N_{0,1}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$M_{0,3}$ ↓ $Mds_{0,1}$	$N_{2,1}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$\text{thread}_{1,0}$	$M_{1,0}$ ↓ $Mds_{1,0}$	$N_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$M_{1,2}$ ↓ $Mds_{1,0}$	$N_{3,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$\text{thread}_{1,1}$	$M_{1,1}$ ↓ $Mds_{1,1}$	$N_{1,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$M_{1,3}$ ↓ $Mds_{1,1}$	$N_{3,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time ⟶

- Note that each value in the shared memory is used **twice** which reduces the global memory access.
- Note that the calculation of each dot product is now performed in **two phases.**
- Note that **Pvalue** is an **automatic variable** so a private version is generated for each thread.
- In general, if an input matrix is of dimension **Width** and the tile size is **TILE_WIDTH**, the dot product would be performed in **Width/TILE_WIDTH** phases.
- Note also that **Mds** and **Nds** are **reused** to hold the input values. This allows a much smaller shared memory to serve most of the accesses to global memory. This is due to the fact that each phase focuses on a small subset of the input matrix elements. Such focused access behaviour is called **locality**.

A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

 1.    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 2.    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 3.    int bx = blockIdx.x;  int by = blockIdx.y;
 4.    int tx = threadIdx.x; int ty = threadIdx.y;

       // Identify the row and column of the d_P element to work on
 5.    int Row = by * TILE_WIDTH + ty;
 6.    int Col = bx * TILE_WIDTH + tx;

 7.    float Pvalue = 0;
       // Loop over the d_M and d_N tiles required to compute d_P element
 8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

         // Coolaborative loading of d_M and d_N tiles into shared memory
 9.      Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
10.      Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
11.      __syncthreads();

12.      for (int k = 0; k < TILE_WIDTH; ++k) {
13.        Pvalue += Mds[ty][k] * Nds[k][tx];
         }
14.      __syncthreads();
       }
15.    d_P[Row*Width + Col] = Pvalue;
     }
```

- The barrier **__syncthreads() in line 11** ensures that all threads have finished loading the tiles of d_M and d_N into Mds and Nds before any of them can move forward.

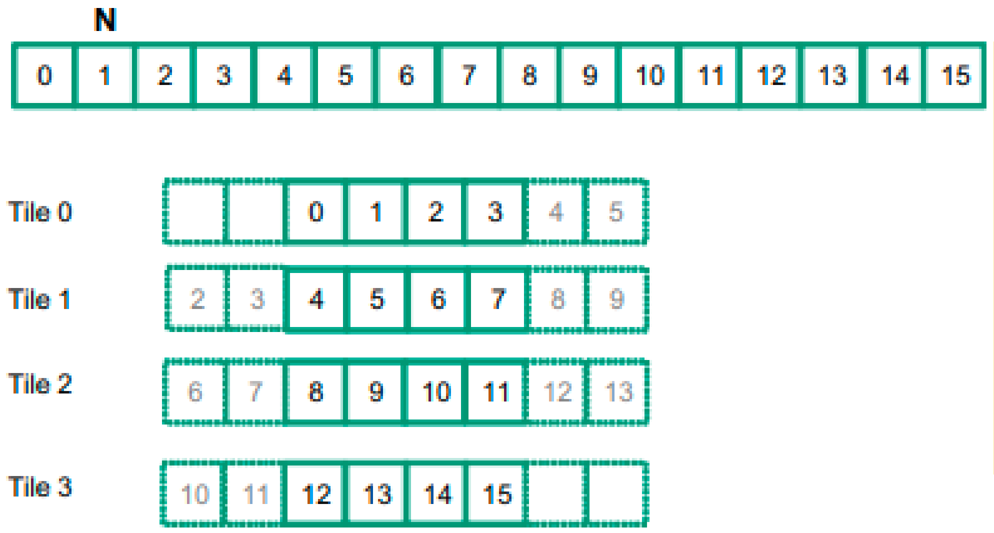
- The **barrier __syncthreads() in line 14** ensures that all threads have finished using the d_M and d_N elements in the shared memory before any of them move on to the next iteration (next phase) and load the elements in the next tiles.

Tiled 1D Convolution with Halo Elements

- 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.

- To understand ***tiled convolution***, we will assume that each thread calculates one output ***P*** element. We will refer to the collection of output elements processed by each block as an ***output tile***.

- The following figure shows a small example of a **16-element**, 1D convolution using **4 thread blocks of 4 threads each**.



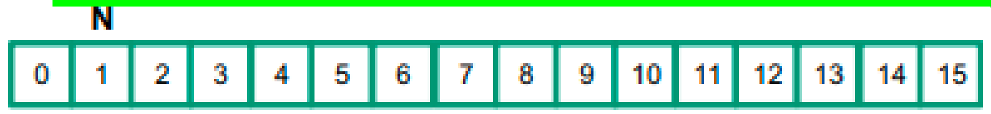
- The **first output tile** covers P[0] through P[3], the **second tile** P[4] through P[7], the **third tile** P[8] through P[11], and the **fourth tile** P[12] through P[15].

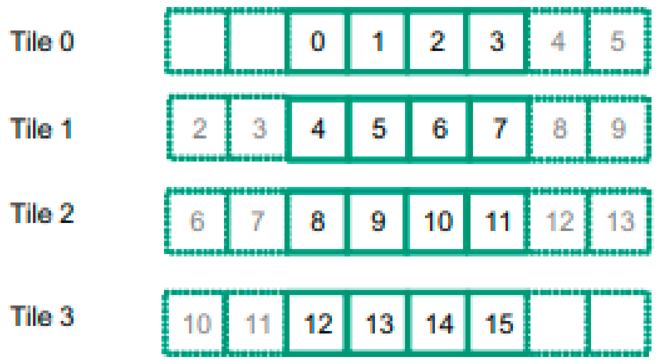
- We will assume that the mask ***M*** elements are in the **constant memory**.

- We will assume that the **mask size is an odd number** equal to **2 × n + 1**. The figure shows an example where **n = 2**

- We will study an intuitive **input data tiling strategy** which involves loading all input data elements needed for calculating all output elements of a thread block into the **shared memory**.

Tiled 1D Convolution with Halo Elements



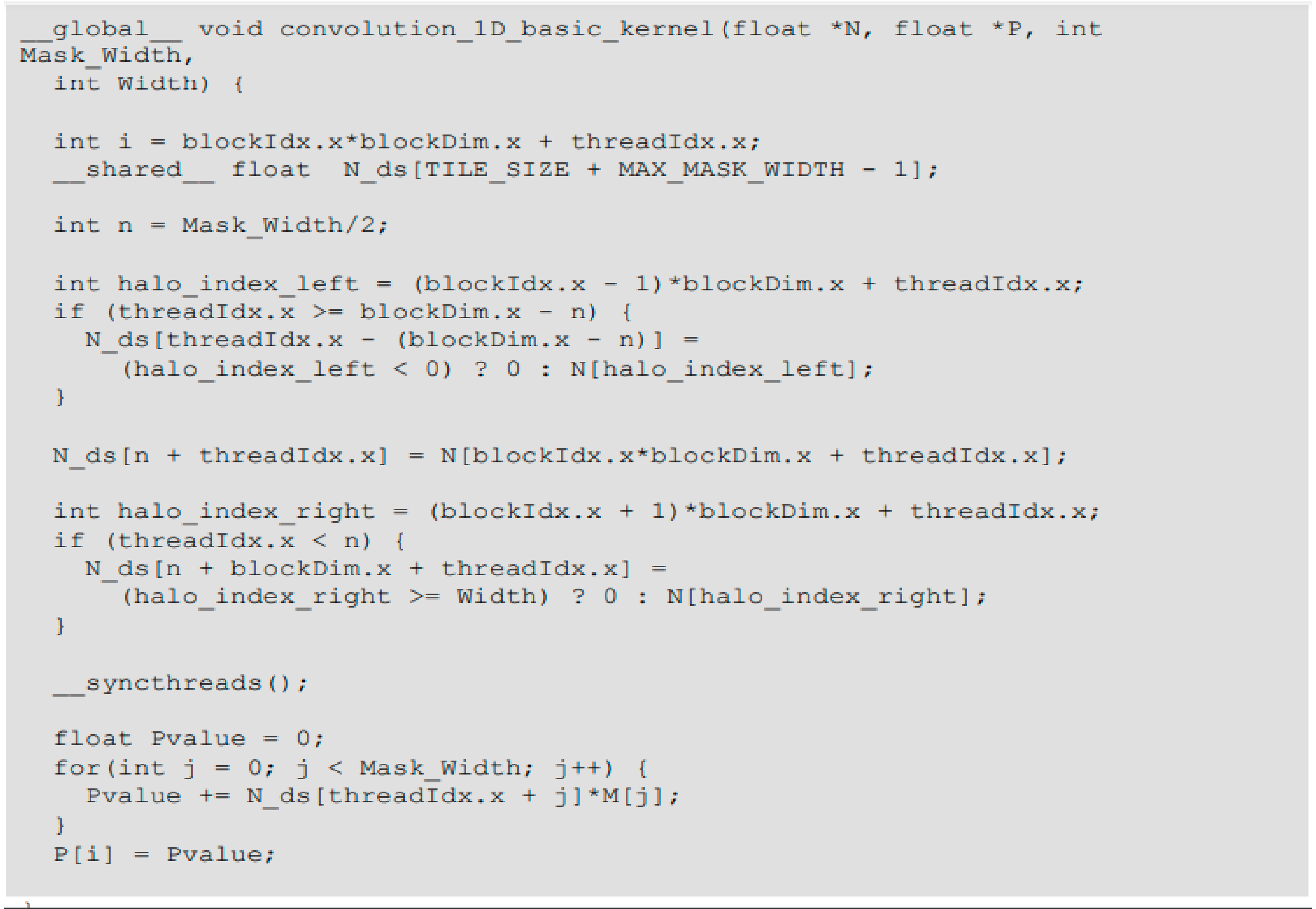


- Threads in **block 0** calculate output elements **P[0]** through **P[3]**. This is the leftmost tile in the output data and is often referred to as the ***left boundary tile***.
- The threads collectively require input elements **N[0]** through **N[5]**.
- Note that the calculation also requires **two ghost elements** to the left of **N [0]**. This is shown as two dashed empty elements on the left end of tile 0. These ghost elements will be assumed have a ***default value of 0***.
- **Tile 3** has a similar situation at the right end of input array N.
- 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].**
- Calculations for tiles 1 and 2 do not involve **ghost elements** and are often referred to as ***internal tiles***.

- The 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 elements***.
- We will refer to the center part of an input tile that is solely used by a single block the ***internal elements*** of that input tile.

Tiled 1D Convolution with Halo Elements



```
__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];
  }
  P[i] = Pvalue;
}
```