Introduction to CUDA and Computational Patterns

6 Hours

Topics covered:

Topics covered:

- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
- Matrix-Matrix Multiplication—A More Complex Kernel
- Calculations of global threadID
- Synchronization and Transparent Scalability
- Assigning Resources to Blocks
- Querying Device Properties

Topics covered:

- 1D Sequential Convolution
- 1D Parallel Convolution – A Basic Algorithm
- Atomic and Arithmetic Functions
- A Simple Parallel Scan Algorithm
- Sequential Sparse-Matrix Vector Multiplication (SpVM)
- Parallel SpVM using CSR

Topics covered:

- Importance of Memory Access Efficiency
- GPU Device Memory Types
- A Strategy or Reducing Global Memory Traffic
- A Tiled Matrix-Matrix Multiplication Kernel
- Constant Memory and Caching
- Tiled 1D Convolution with Halo Elements

Introduction

- CUDA stands for **Compute Unified Device Architecture**

- CUDA C is an extension to the popular C programming language used for writing massively parallel programs in a heterogeneous computing system.

- To a CUDA programmer, the computing system consists of a ***host*** that is a traditional CPU, and one or more ***devices*** (GPUs) that are processors with a massive number of arithmetic units.

- Software applications often have sections that exhibit a rich amount of ***data parallelism***, a phenomenon that *allows arithmetic operations to be safely performed on different parts of the data structures in parallel.*

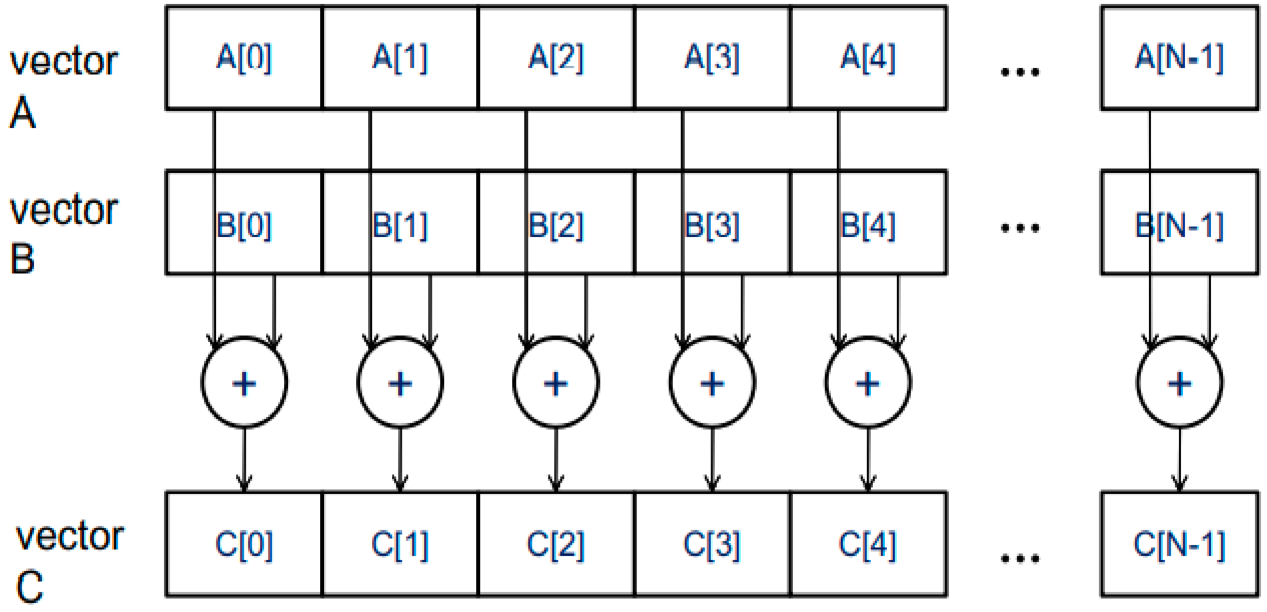
- CUDA devices accelerate the execution of software applications by *applying their massive number of arithmetic units to the data-parallel program sections.*

Data Parallelism

- Modern software applications often process a large amount of data and *incur long execution time on sequential computers*.
- Parallel programming uses both *Task parallelism* and *Data parallelism*.

Task Parallelism Vs Data parallelism

- **Task parallelism** exists if the two tasks can be done independently. For example, a simple application may need to do a vector addition and a matrix-vector multiplication. Each of these would be a task.
- **Data parallelism** refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner. For example, in vector addition, we use data parallelism.



CUDA Program Structure

- The structure of a CUDA program reflects the coexistence of a **host (CPU)** and one or more **devices (GPUs)** in the computer.

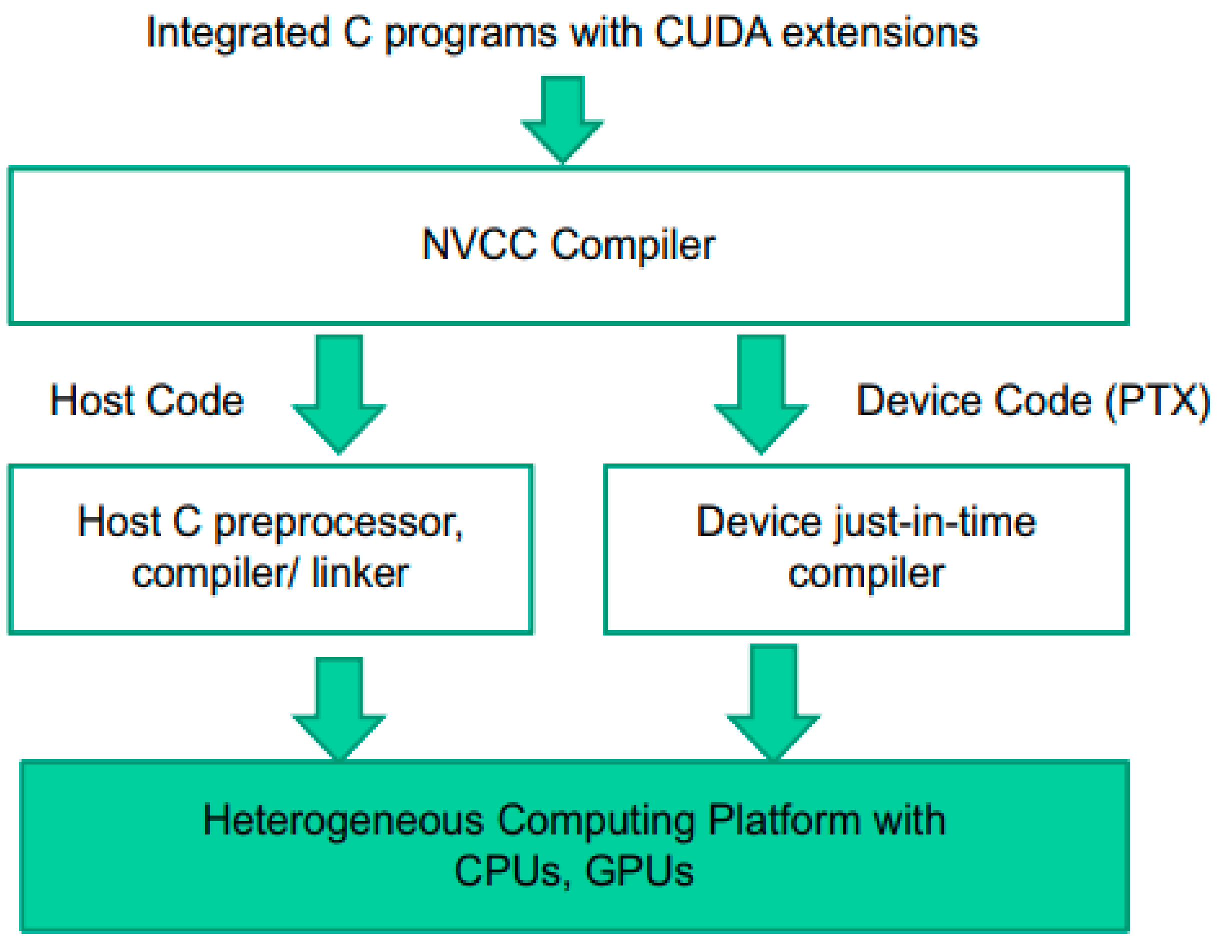
- Each CUDA source file can have a mixture of **both host and device code**.

- By default, any traditional C program is a CUDA program that contains only host code. One can add device functions and data declarations into any C source file by marking them with **special CUDA keywords**.

- The **NVIDIA C Compiler (NVCC)** separates the host code and the device code during compilation process.

  - ❑ The host code is compiled with the host's standard C compilers and runs as an ordinary CPU process.
  - ❑ The device code(kernels) is compiled by the NVCC and executed on a GPU device.

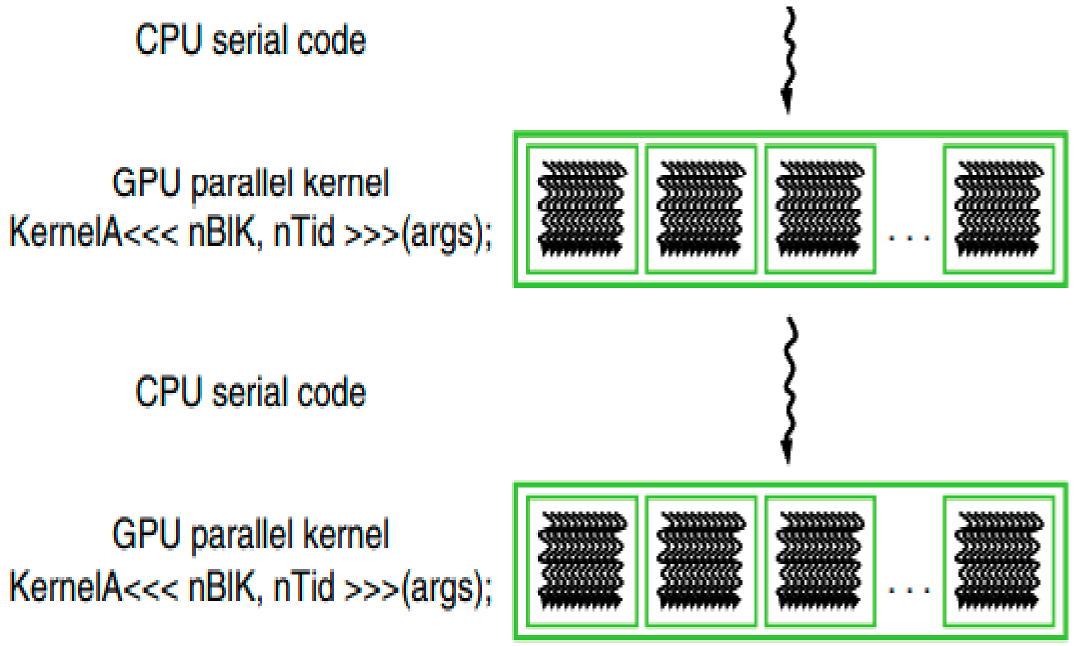
CUDA Program Structure


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/ linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

An Overview of the compilation process of a CUDA program

CUDA Program Structure

- The execution of a CUDA program starts with ***host (CPU) execution***.
- When a **kernel function** is called (launched), *it is executed by a large number of threads* on a device.
- All the threads that are generated by a kernel launch are collectively called a ***grid***.
- When all threads of a kernel complete their execution, the corresponding **grid terminates**, and the execution continues on the host until another kernel is launched.



CUDA threads take very few cycles to generate and schedule due to efficient hardware support.

CPU threads typically require **thousands of clock cycles to generate and schedule**

A Vector Addition Kernel

- A traditional vector addition C code example:.

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
        for (i = 0; i < n; i++)
                h_C[i] = h_A[i] + h_B[i];
}

int main()
{
        // Memory allocation for h_A, h_B, and h_C
        // I/O to read h_A and h_B, N elements each …
        vecAdd(h_A, h_B, h_C, N);
 }
```

A Vector Addition Kernel

- A modified vecAdd() function for execution on a CUDA device:.

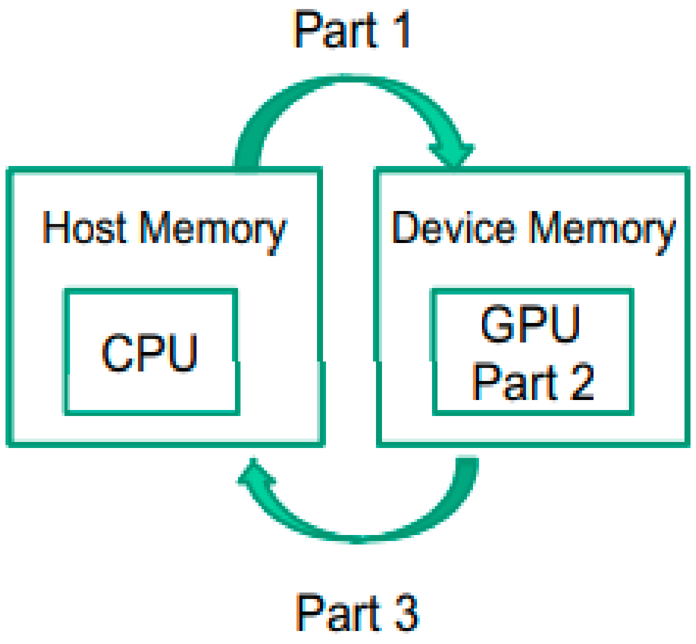
```
#include <cuda.h>

void vecAdd(float* A, float*B, float* C, int n)
{
          float *d_A, *d_B, *d_C;
          int size = n* sizeof(float);

          Part-1. // Allocate device memory for A, B, and C
                  // Copy A and B to device memory

          Part-2. // Kernel launch code – to have the device
                  // to perform the actual vector addition

          Part-3. // Copy C from the device memory
                  // Free device vectors
 }
```



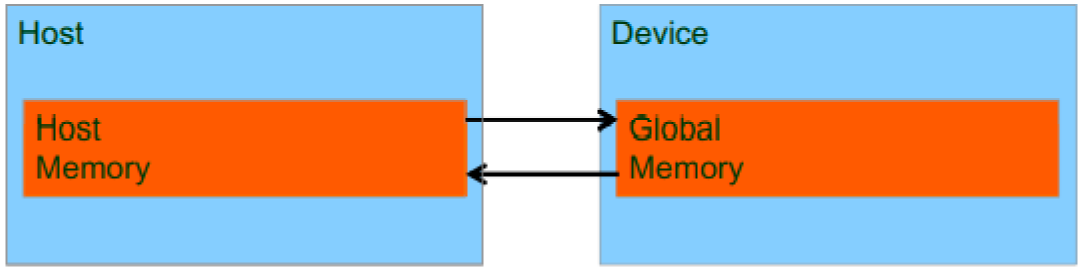
Device Global Memory and Data transfer

- In CUDA, the host and devices have separate memory spaces. Devices are typically hardware cards that come with their own **Dynamic Random Access Memory (DRAM**) which is also called as ***Global memory***.

- The CUDA runtime system provides **Application Programming Interface (API)** functions to perform the following activities on behalf of the programmer:

  - ❑ To execute a kernel on a device, the programmer needs to *allocate global memory on the device* and transfer pertinent data from the host memory to the allocated device memory (**Part-1**).

  - ❑ Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed (**Part-3**).



CUDA host memory and device memory model for programmers

Device Global Memory and Data transfer

- The CUDA runtime system provides API functions for managing data in the device memory.

- Function **cudaMalloc()** can be called from the host code to allocate a piece of device global memory for an object. It takes two parameters:

1. The ***first parameter*** to the cudaMalloc() function is the address of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to (void) because the function expects a generic pointer.

2. The ***second parameter*** to the cudaMalloc() function gives the size of the data to be allocated, in terms of bytes.

- Function **cudaFree()** is called to free the storage space allocated for an object from the device global memory.

Device Global Memory and Data transfer

```
#include <cuda.h>

void vecAdd(float* A, float*B, float* C, int n)
{
                float *d_A, *d_B, *d_C;
                int size = n* sizeof(float);

                Part-1. // Allocate device memory for A, B, and C
                        // Copy A and B to device memory

                Part-2. // Kernel launch code – to have the device
                        // to perform the actual vector addition

                Part-3. // Copy C from the device memory
                        // Free device vectors
 }
```

The addresses in d_A, d_B, and d_C are addresses in the device memory. **These addresses should not be dereferenced in the host code.** They should be mostly used in calling API functions and kernel functions. Dereferencing a device memory pointer in the host code can cause exceptions or other types of runtime errors during runtime

```
#include <cuda.h>

void vecAdd(float* A, float*B, float* C, int n)
{
                float *d_A, *d_B, *d_C;
                int size = n* sizeof(float);


                Part-1. cudaMalloc((void**)&d_A, size);
                        cudaMalloc((void**)&d_B, size);
                        cudaMalloc((void**)&d_C, size);
                        // Copy A and B to device memory

                Part-2. // Kernel launch code – to have the device
                        // to perform the actual vector addition

                Part-3. // Copy C from the device memory
                       cudaFree(d_A);
                       cudaFree(d_B);
                       cudaFree(d_C);
 }
```

Device Global Memory and Data transfer

- Once the host code has allocated device memory for the data objects, it calls **cudaMemcpy()** to transfer the data from host to device.

- The **cudaMemcpy()** function takes four parameters:

1. The *first parameter* is a pointer to the destination location for the data object to be copied.

2. The *second parameter* points to the source location.

3. The *third parameter* specifies the number of bytes to be copied.

4. The *fourth parameter* indicates the types of memory involved in the copy:
    - ❑ from host memory to host memory
    - ❑ from host memory to device memory
    - ❑ from device memory to host memory
    - ❑ from device memory to device memory

- **cudaMemcpy() cannot be used to copy between different GPUs in multi-GPU systems**

Device Global Memory and Data transfer

```
#include <cuda.h>

void vecAdd(float* A, float*B, float* C, int n)
{
        float *d_A, *d_B, *d_C;
        int size = n* sizeof(float);

        Part-1. // Allocate device memory for A, B, and C
                // Copy A and B to device memory

        Part-2. // Kernel launch code – to have the device
                // to perform the actual vector addition

        Part-3. // Copy C from the device memory
                // Free device vectors
 }
```

```
#include <cuda.h>

void vecAdd(float* A, float*B, float* C, int n)
{
  float *d_A, *d_B, *d_C;
  int size = n* sizeof(float);

  // Part-1
  cudaMalloc((void**)&d_A, size);
  cudaMalloc((void**)&d_B, size);
  cudaMalloc((void**)&d_C, size);

  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  // Part-2
  // Kernel launch code – to have the device to perform the
  //actual vector addition

  // Part-3
  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
  cudaFree(d_A);  cudaFree(d_B); cudaFree(d_C);
 }
```

Error Handling in CUDA

- CUDA API functions return flags that indicate whether an error has occurred when they served the request.

- Most errors are due to inappropriate argument values used in the API call.

- In practice, we should surround the API call with code that tests for error conditions and prints out error messages so that the user can be aware of the fact that an error has occurred.

```
// if the system is out of device memory, the user will be informed about the situation.

cudaError_t err = cudaMalloc((void** ) &d_A, size);

if (err ! = cudaSuccess)
{
   printf("%s in %s at line %d\n", cudaGetErrorString (err), __FILE__, __LINE__);
   exit(EXIT_FAILURE);
}
```

Kernel Functions and Threading

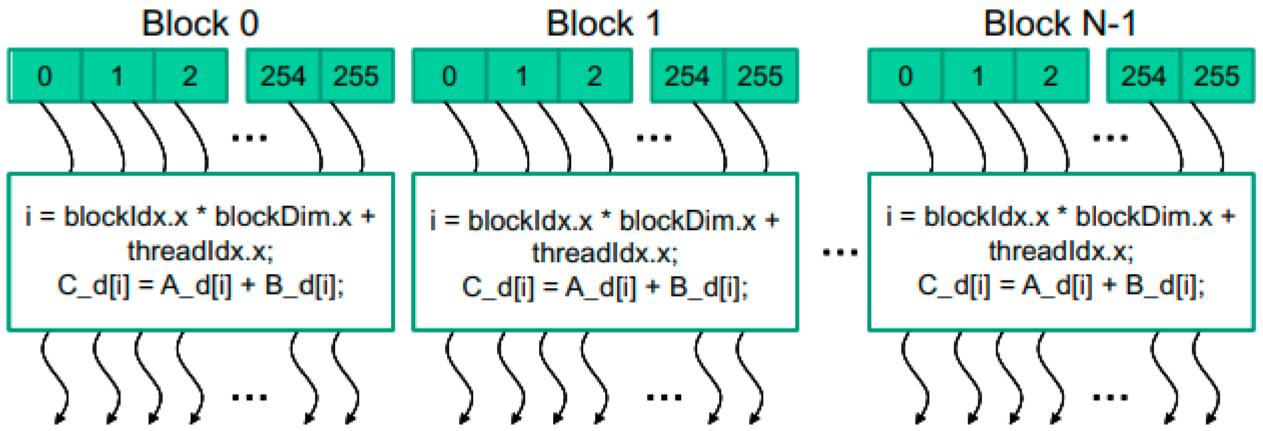
- In CUDA, a **kernel function** specifies the code to be executed by all threads during a parallel phase.
- Since all the threads execute the same code, CUDA programming is an instance of the well-known SPMD (single program, multiple data) parallel programming style.

<u>SPMD Vs SIMD</u>

- ➢ SPMD is not the same as SIMD (single instruction, multiple data).
- ➢ In an SPMD system, the parallel processing units execute the same program on multiple parts of the data.
- ➢ In a SIMD system, all processing units are executing the same instruction at any instant.

Kernel Functions and Threading

- When a host code **launches a kernel**, the CUDA runtime system generates a **grid of threads** that are organized in a two-level hierarchy.
- Each grid is organized into an **array of thread blocks**.
- All blocks of a grid are of the **same size**; each block can contain up to **1,024 threads**.
- The **number of threads** in each thread block is specified by the host code when a kernel is launched.
- The same kernel can be launched with **different numbers of threads** at different parts of the host code.
- For a given grid of threads, the number of threads in a block is available in the **blockDim** variable.

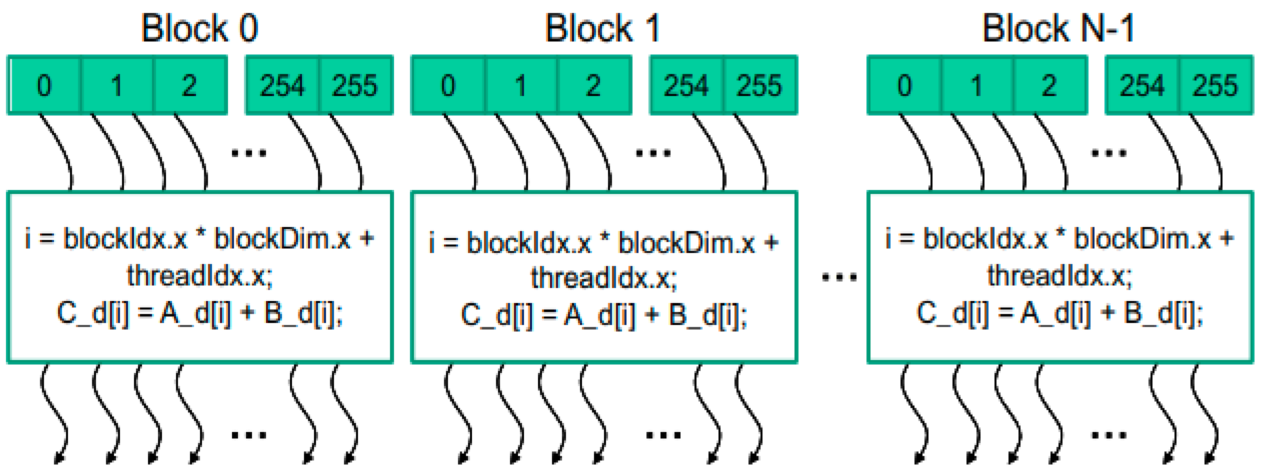

Block 0
0 1 2 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
Block 1
Block N-1

The value of **blockDim.x** variable is **256**. In general, the dimensions of thread blocks should be **multiples of 32** due to hardware efficiency reasons.

Kernel Functions and Threading

- Each thread in a block has a **unique threadIdx value (0, 1, . . 255)**.
- This allows each thread to combine its **threadIdx** and **blockIdx** values to create a **unique global index for itself with the entire grid**.
- A data index **i** is calculated as:

  i = blockIdx.x * blockDim.x + threadIdx.x.
- By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with **n** or more threads, one can process vectors of length **n**.


Block 0
0
1
2
254
255
...
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
Block 1
Block N-1

Since **blockDim** is **256** in our example, the **i** values of threads in **block 0** ranges from **0 to 255**. The **i** values of threads in **block 1** range from **256 to 511**. The **i** values of threads in **block 2** range **from 512 to 767**.

Since each thread uses **i** to access **d_A, d_B**, and **d_C**, these threads cover the **first 768 vector elements for the addition**.

Kernel Functions and Threading

Kernel function for a vector addition

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
_ _global_ _ void vecAddKernel(float* A, float* B, float* C, int n)
{
        int i = threadIdx.x + blockDim.x * blockIdx.x;

        if(i < n)
            C[i] = A[i] + B[i];
}
```

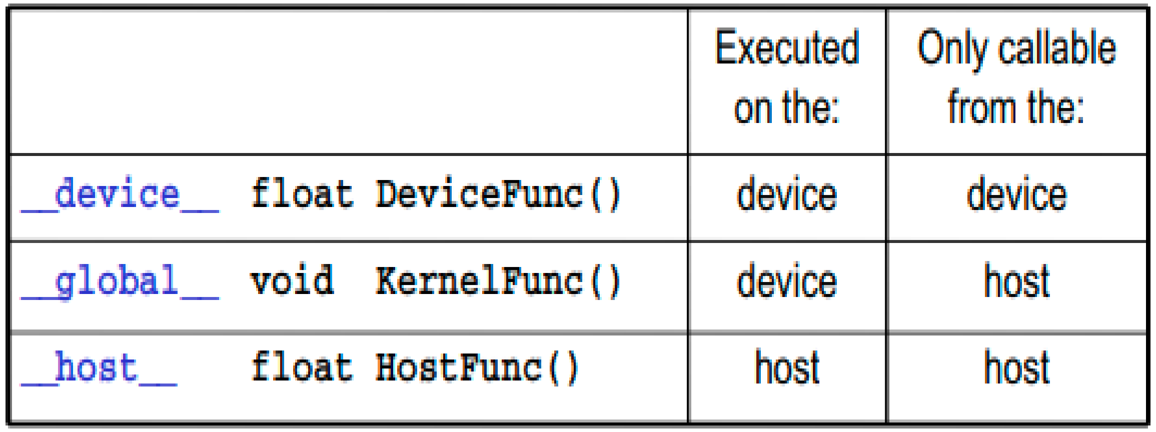
- The keyword __**global**__ indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device.
- The condition **if(i < n)** will only execute the required threads and prevents the execution of unnecessary threads.

For example, if the **vector length is 100**, the smallest efficient **thread block dimension is 32**. we need to launch **four thread blocks** to process all the **100** vector elements which will create **128** threads.

By setting condition **if(i < n) ,** we disable the last **28** threads in thread block 3 from doing work.

Kernel Functions and Threading

- CUDA extends C language with three qualifier keywords in function declarations:



	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

➢ The __**global**__ keyword indicates that the function being declared is a CUDA kernel function. A __global__ function is to be executed on the device and can only be called from the host code.

➢ The __**device**__ keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can only be called from a kernel function or another device function. **Device functions can have neither recursive function calls nor indirect function calls through pointers in them.**

➢ The __**host**__ keyword indicates that the function being declared is a CUDA host function. A host function is simply a traditional C function that executes on the host and can only be called from another host function.

Kernel Functions and Threading

- **By default**, all functions in a CUDA program **are host functions** if they do not have any of the CUDA keywords in their declaration.

- One can use **both** __**host**__ and __**device**__ in a function declaration. This combination tells the compilation system to generate two versions of object files for the same function:
  - ➢ One is executed on the host and can only be called from a host function.
  - ➢ The other is executed on the device and can only be called from a device or kernel function.

- When the host code launches a kernel, it sets the **grid and thread block dimensions** via **execution configuration parameters:**
  - o The configuration parameters are given between the **<<<** and **>>>** before the traditional C function arguments.
  - o The *first configuration parameter* gives the **number of thread blocks in the grid**.
  - o The *second configuration parameter* specifies the **number of threads in each thread block**.

```
vecAddKernel<<< ceil(n/256.0), 256 >>> (d_A, d_B, d_C, n);
```

Complete CUDA program for vecADD()

```
#include <cuda.h>

void vecAdd(float* A, float*B, float* C, int n)
{
    float *d_A, *d_B, *d_C;
    int size = n* sizeof(float);

    // Part-1
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    // Part-2
    vecAddKernel<<< ceil(n/256.0), 256 >>>(d_A, d_B, d_C, n);

    // Part-3
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);  cudaFree(d_B); cudaFree(d_C);
 }
```

```
// Kernel for computing vector sum C = A+B
_ _global_ _ void vecAddKernel(float* A, float* B, float* C, int n)
{
       int i = threadIdx.x + blockDim.x * blockIdx.x;

       if(i < n)
          C[i] = A[i] + B[i];
}
```

CUDA Thread Organization

- All CUDA threads in a grid execute the **same kernel function** and they rely on **coordinates** to distinguish themselves from each other and to identify the appropriate portion of the data to process.

- CUDA threads are organized into a two-level hierarchy:
  1) A grid consists of one or more blocks
  2) Each block in turn consists of one or more threads.

- All threads in a block share the same block index, which can be accessed as the **blockIdx** variable in a kernel.

- Each thread also has a thread index, which can be accessed as the **threadIdx** variable in a kernel.

- When a thread executes a kernel function, references to the **blockIdx** and **threadIdx** variables return the **coordinates of the thread**.

- The **execution configuration parameters** in a kernel launch statement specify the dimensions of the grid and the dimensions of each block:

```
vecAddKernel<<< ceil(n/256.0), 256 >>>(d_A, d_B, d_C, n);
```

CUDA Thread Organization

- In general, a grid is a **3D array of blocks,** and each block is a **3D array of threads**.
- The programmer can choose to use fewer dimensions by setting the unused dimensions to 1.
- The exact organization of a grid is determined by the execution configuration parameters (within <<< and >>>) of the kernel launch statement. The first execution configuration parameter specifies the **dimensions of the grid in number of blocks**. The second specifies the **dimensions of each block in number of threads**.
- Each such parameter is of **dim3** type, which is a **C structure** with three unsigned integer fields, x, y, and z. These three fields correspond to the three dimensions:

```
dim3 dimGrid(128, 1, 1);
dim3 dimBlock(32, 1, 1);
vecAddKernel <<< dimGrid, dimBlock >>> (...);
```

CUDA Thread Organization

- The grid and block dimensions can also be calculated from other variables. In the following example, the value of variable **n** at kernel launch time will determine the dimension of the grid:

```
dim3 dimGrid (ceil(n/256.0), 1, 1);
dim3 dimBlock(32, 1, 1);
vecAddKernel <<< dimGrid, dimBlock >>> (...);     // n is the number of data elements
```

- For convenience, CUDA C provides a **special shortcut** for launching a kernel with 1D grids and blocks. Instead of using **dim3** variables, one can use arithmetic expressions to specify the configuration of 1D grids and blocks. In this case, the CUDA C compiler simply takes the arithmetic expression as the **x** dimensions and assumes that the **y** and **z** dimensions are 1.

```
vecAddKernel <<< ceil(n/256.0), 32 >>> (...);     // n is the number of data elements
```

- For a given grid of threads, the **dimension of grid** is available in **gridDim** variable, and the **dimension of each block** is available in the **blockDim** variable.

CUDA Thread Organization

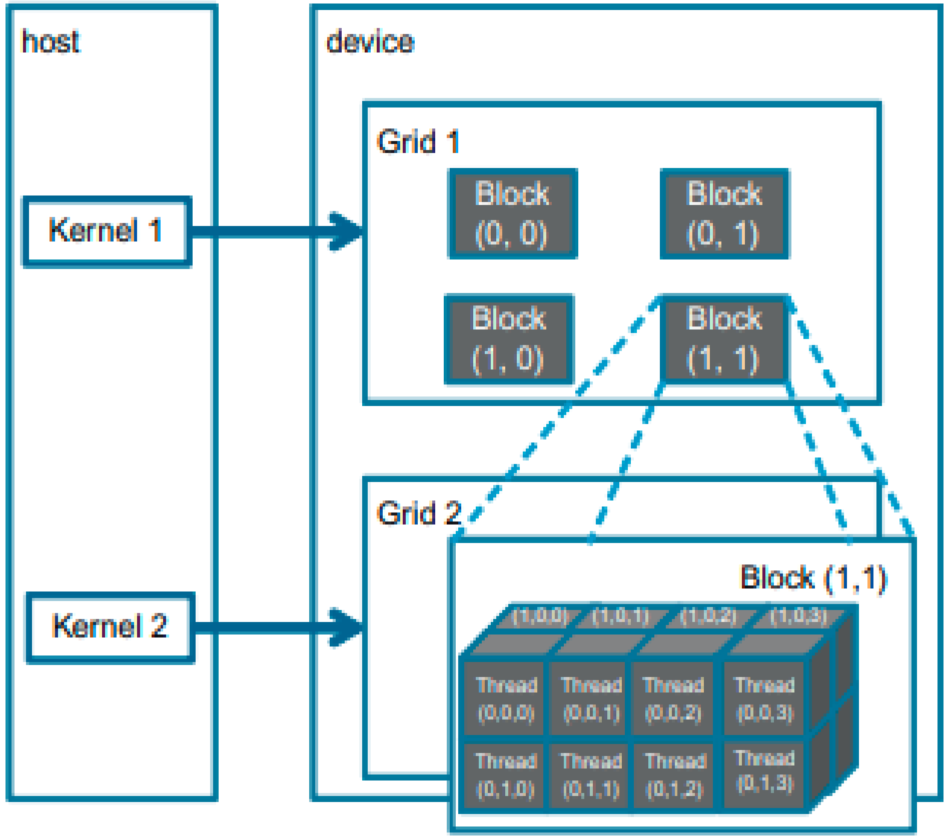
- In CUDA C, the allowed values of gridDim.x, gridDim.y, and gridDim.z range from **1** to **65,536**.
- *All threads in a block share the same* **blockIdx.x**, **blockIdx.y**, and **blockIdx.z** values.
- Among all blocks, the **blockIdx.x** value ranges between **0** and **gridDim.x-1**, the **blockIdx.y** value between **0** and **gridDim.y-1**, and the **blockIdx.z** value between **0** and **gridDim.z-1**.
- **All blocks in a grid have the same dimensions.**
- The total size of a block is limited to 1,024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1,024.
  - ➢ For example, (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowable blockDim values, but (32, 32, 2) is not allowable since the total number of threads would exceed 1,024.

```
dim3 dimGrid(128, 1, 1);  // gridDim.x = 128, gridDim.y = 1, gridDim.z = 1
dim3 dimBlock(32, 1, 1);  // blockDim.x = 32, blockDim.y = 1, blockDim.z = 1
vecAddKernel <<< dimGrid, dimBlock >>> (...);
```

CUDA Thread Organization

- Example of a 2D (2, 2, 1) grid that consists of 3D (4, 2, 2) blocks. The grid can be generated with the following host code:

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
vecAddKernel <<< dimGrid, dimBlock >>> (...);
```

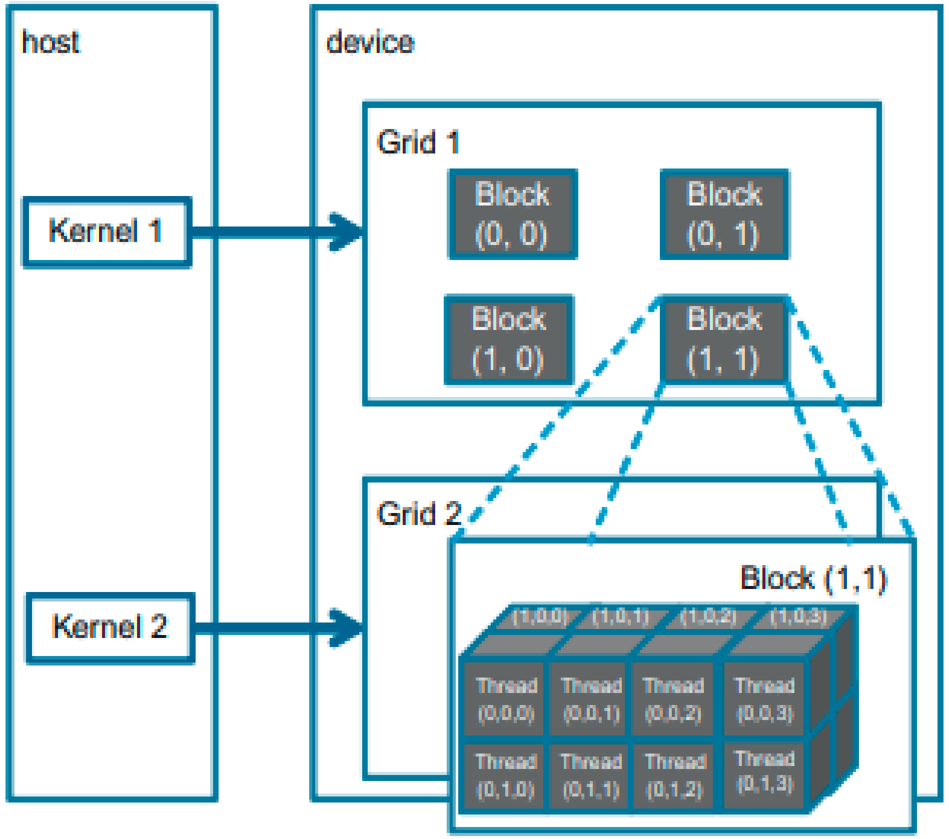


- Each block in Figure is labeled with **(blockIdx.y, blockIdx.x)**.
- For example, block(1,0) has blockIdx.y=1 and blockIdx.x=0.
- **Note that the ordering of the labels is such that the highest dimension comes first. This is reverse of the ordering used in the configuration parameters where the lowest dimension comes first.**
- This reversed ordering for labeling threads works better for mapping of thread coordinates into data indexes in accessing multidimensional arrays.

CUDA Thread Organization

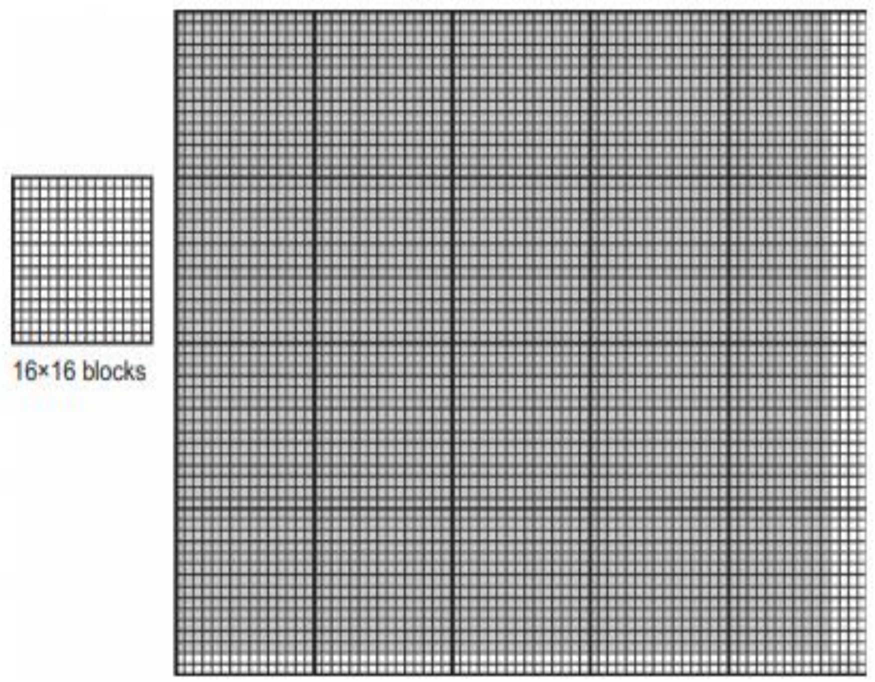
- Each **threadIdx** also consists of three fields: the **x** coordinate **threadId.x**, the **y** coordinate **threadIdx.y**, and the **z** coordinate **threadIdx.z**.

- In the following example, each block is organized into **4 x 2 x 2** arrays of threads. The Figure expands block(1,1) to show its 16 threads. For example, thread(1,0,2) has threadIdx.z=1, threadIdx.y=0, and threadIdx.x=2. In this example, we have four blocks of 16 threads each, with a grand total of 64 threads in the grid.



Mapping Threads to Multidimensional Data

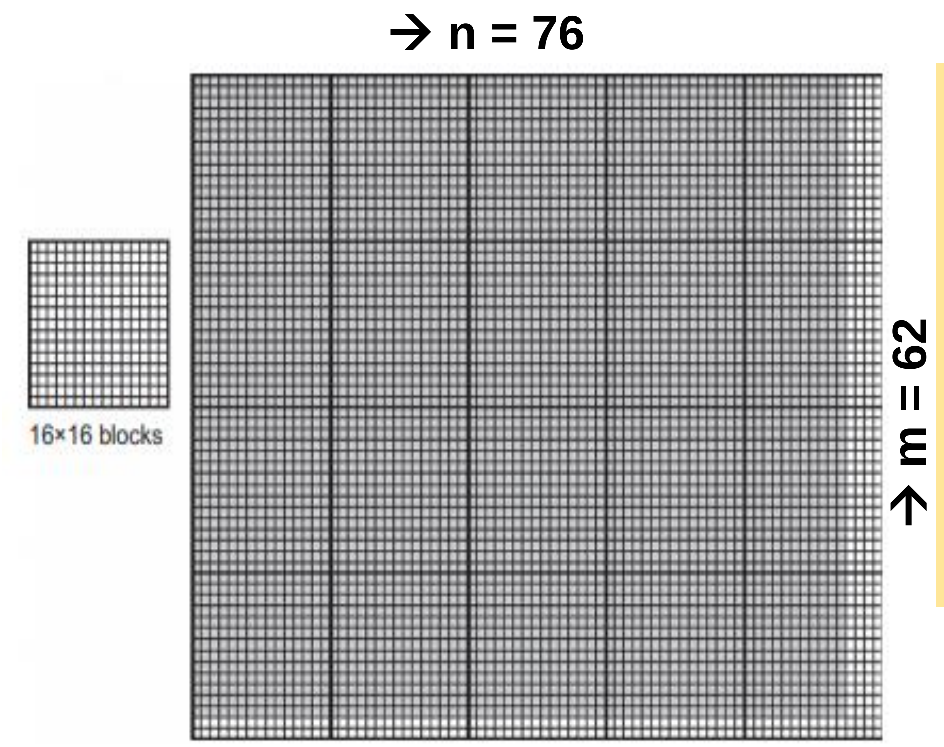
- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data.
- For example, pictures are a 2D array of pixels. It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture.



- 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);
```

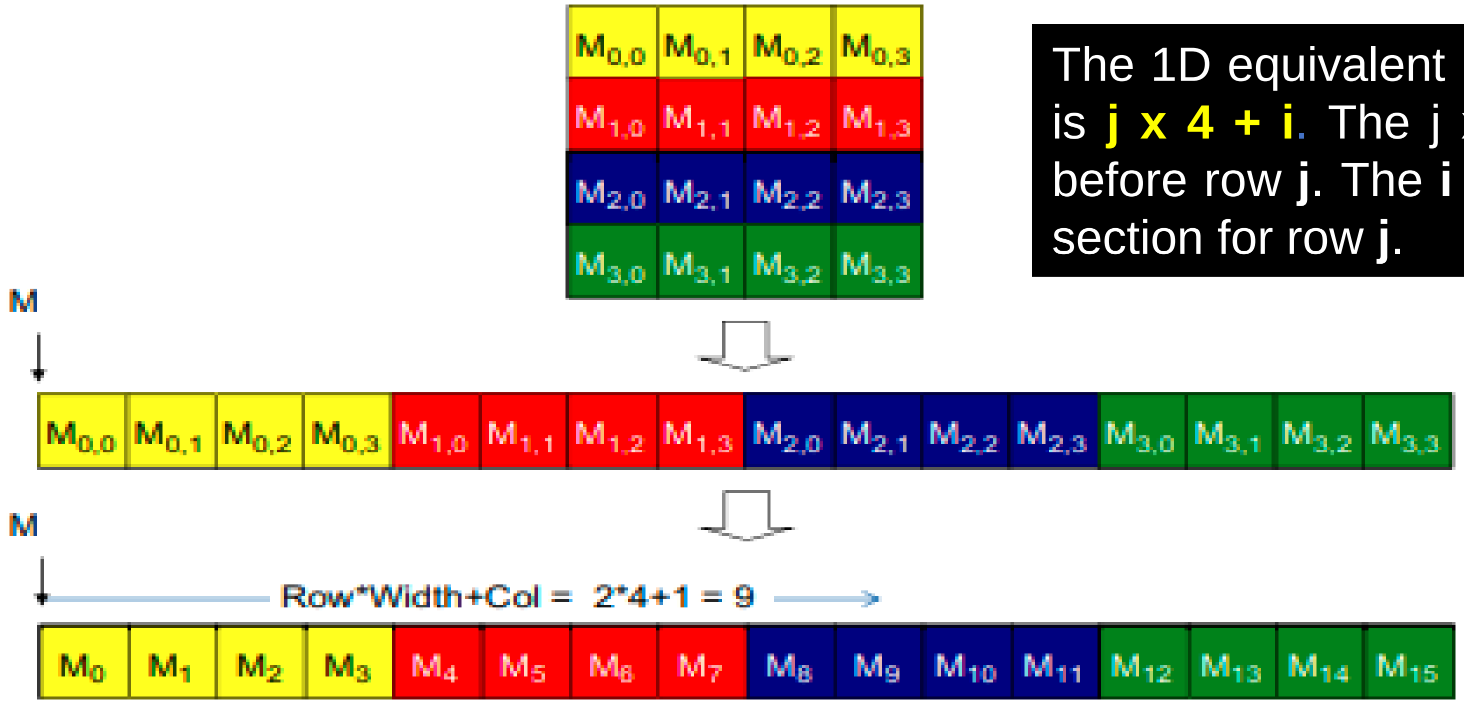
- Within the kernel function, references to built-in variables gridDim.x, gridDim.y, blockDim.x, and blockDim.y will result in 5, 4, 16, and 16, respectively.

Mapping Threads to Multidimensional Data - 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.



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

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**.

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. **column-major layout**

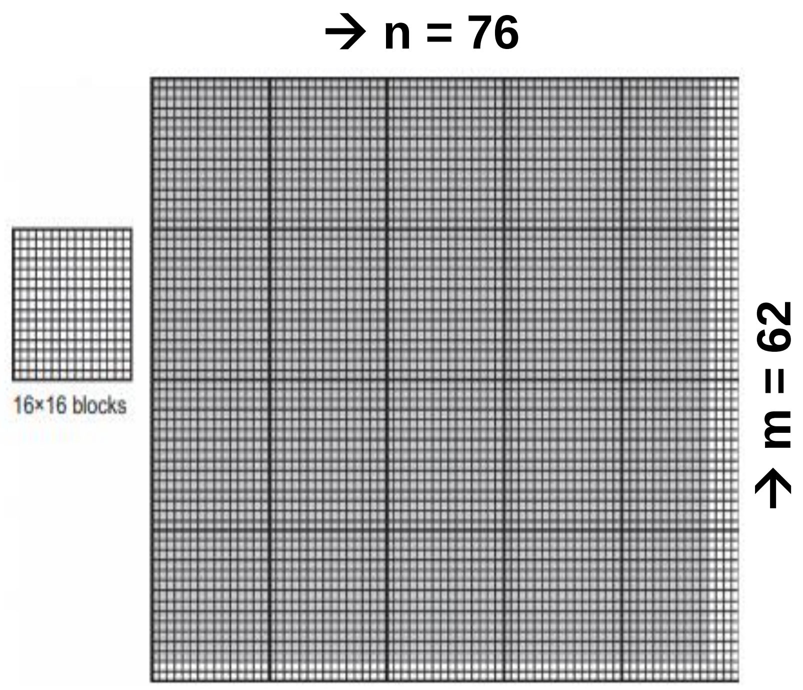
- Here we place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space.

- 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.

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.

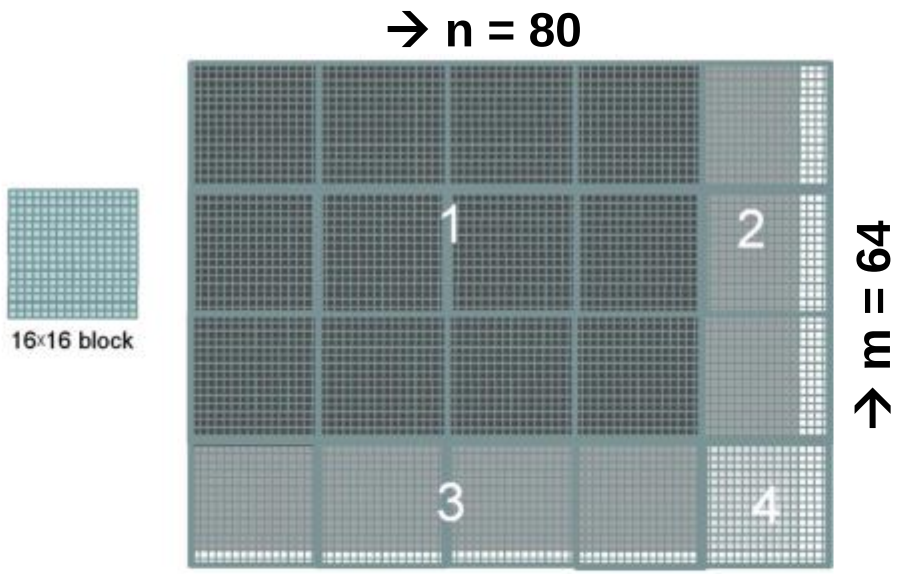
- There are a total of **blockDim.x * gridDim.x** threads in the horizontal direction
- and **blockDim.y * gridDim.y** threads in the vertical direction.

- 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:

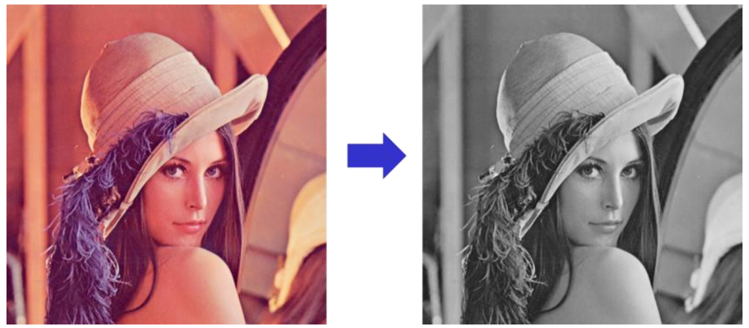
1. The **first area**, marked as *1* consists of the threads that belong to the *12* blocks covering the majority of pixels in the picture. Both Col and Row values of these threads are within range (76).(16 *16)

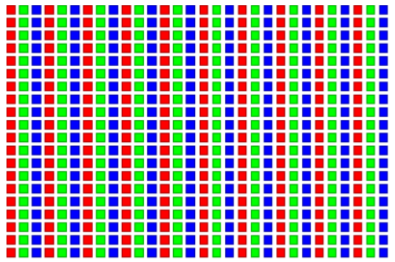
2. The **second area**, marked as *2* contains the threads that belong to the *3* blocks covering the upper-right pixels of the picture. Although the Row values of these threads are always within range, the Col values of some of them exceed the n value.(12*16)

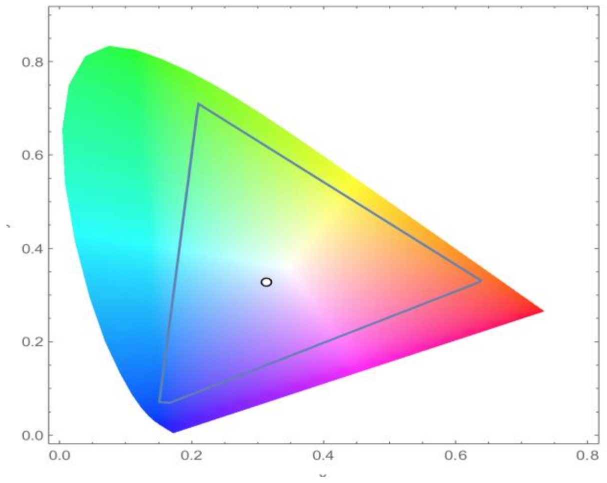
3. The **third area**, marked as *3* contains the threads that belong to the *4* blocks covering the lower-left pixels of the picture. Although the Col values of these threads are always within range, the Row values of some of them exceed the m value (62).(16*14)

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.(14*12)

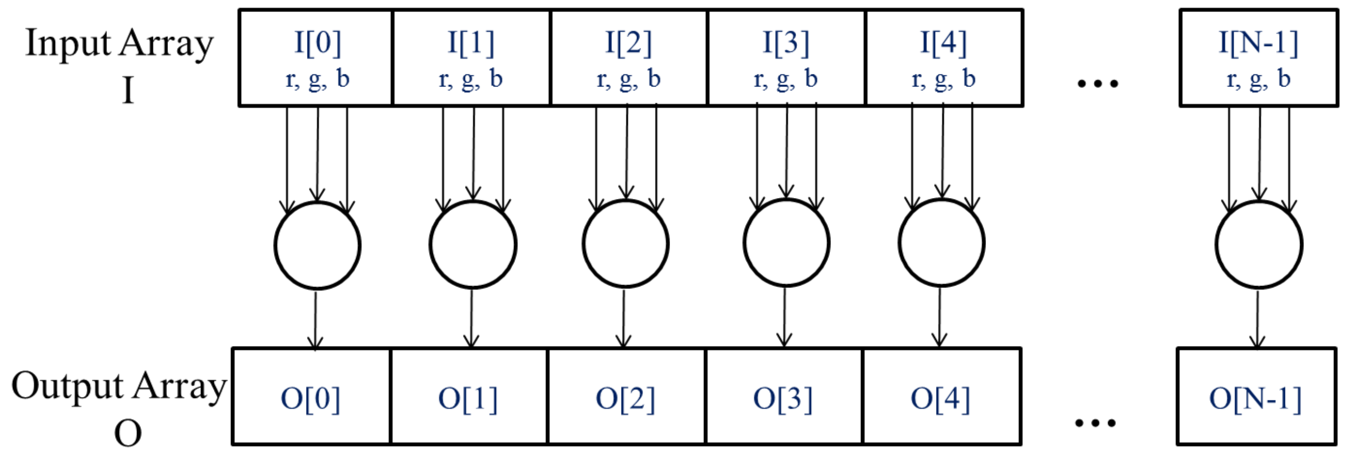
Conversion of a color image to grey–scale image (review)



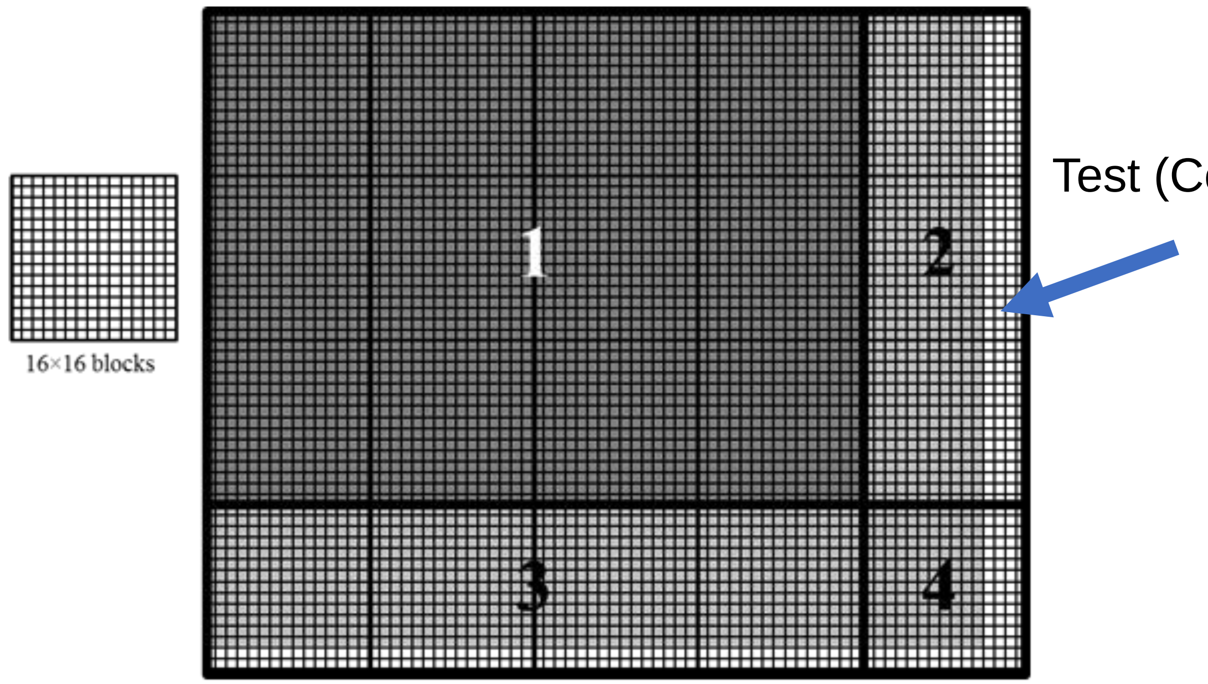




The pixels can be calculated independently of each other (review)



Covering a 76×62 picture with 16×16 blocks



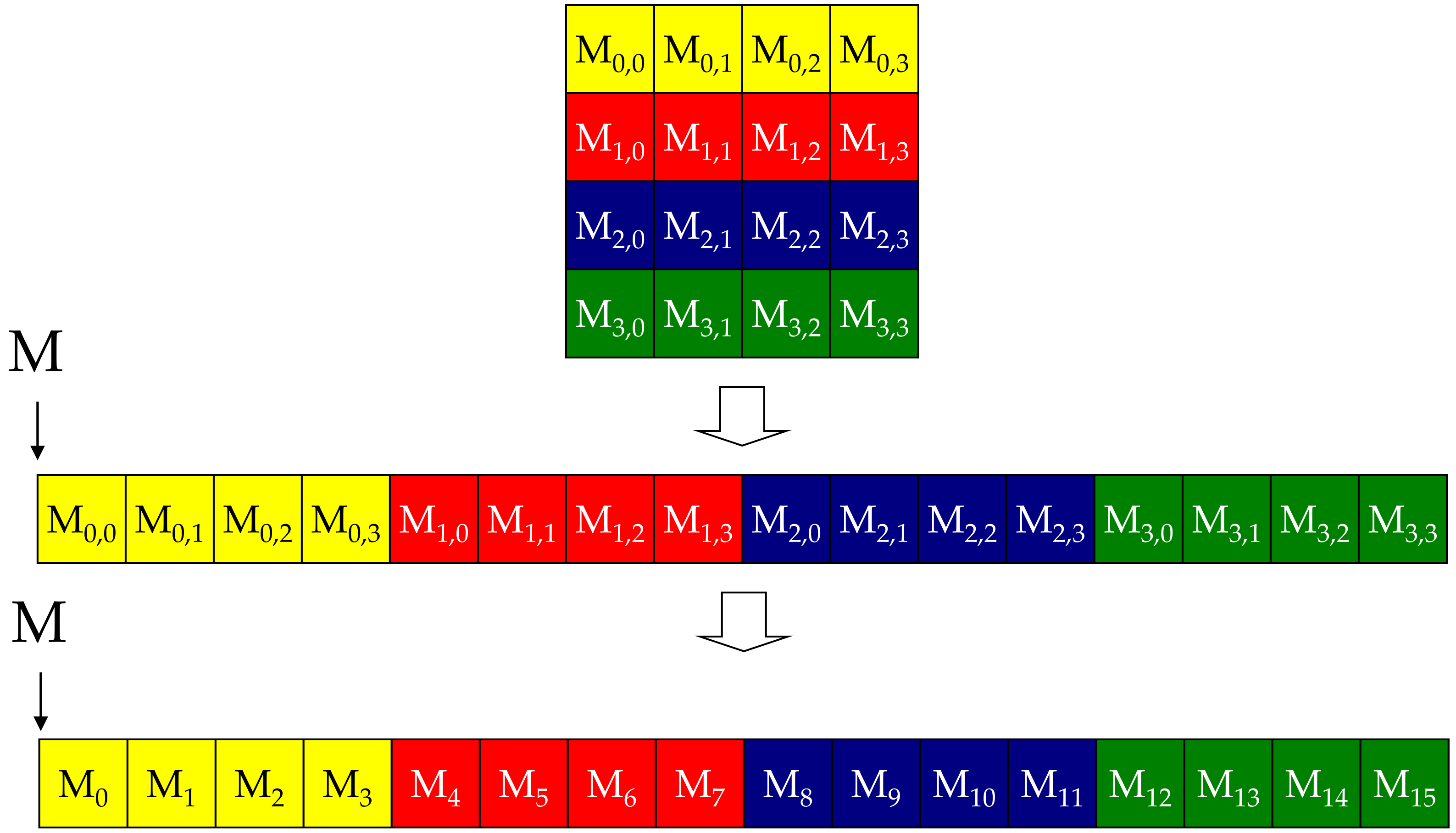
colorToGreyscaleConversion Kernel with 2D thread mapping to data

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConvertion(unsigned char * Pout,  unsigned char * Pin,
                        int width, int height) {

 int Col =   threadIdx.x + blockIdx.x * blockDim.x;
 int Row = threadIdx.y + blockIdx.y * blockDim.y;

 if (Col < width && Row < height) {
    // get 1D coordinate for the grayscale image
    int greyOffset = Row*width + Col;
    // one can think of the RGB image having
    // CHANNEL times columns of the gray scale image
    int rgbOffset = greyOffset*CHANNELS;
    unsigned char r =  rgbImage[rgbOffset      ]; // red value for pixel
    unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
    unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
    // perform the rescaling and store it
    // We multiply by floating point constants
    grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

Row-Major Layout of 2D arrays in C/C++



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
M_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}

$M_{2,1} \rightarrow$ Row*Width+Col = 2*4+1 = 9

colorToGreyscaleConversion Kernel with 2D thread mapping to data (cont.)

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConvertion(unsigned char * Pout,  unsigned char * Pin,
                 int width, int height) {

 int Col =   threadIdx.x + blockIdx.x * blockDim.x;
 int Row = threadIdx.y + blockIdx.y * blockDim.y;

 if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns of the gray scale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset      ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

1D Sequential Convolution

- **Convolution** is a popular *array operation* that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision.

- Convolution is often performed as a **filter** that transforms signals and pixels into more desirable values. For example, **Gaussian filters** are convolution filters that can be used to **sharpen boundaries** and edges of objects in images.

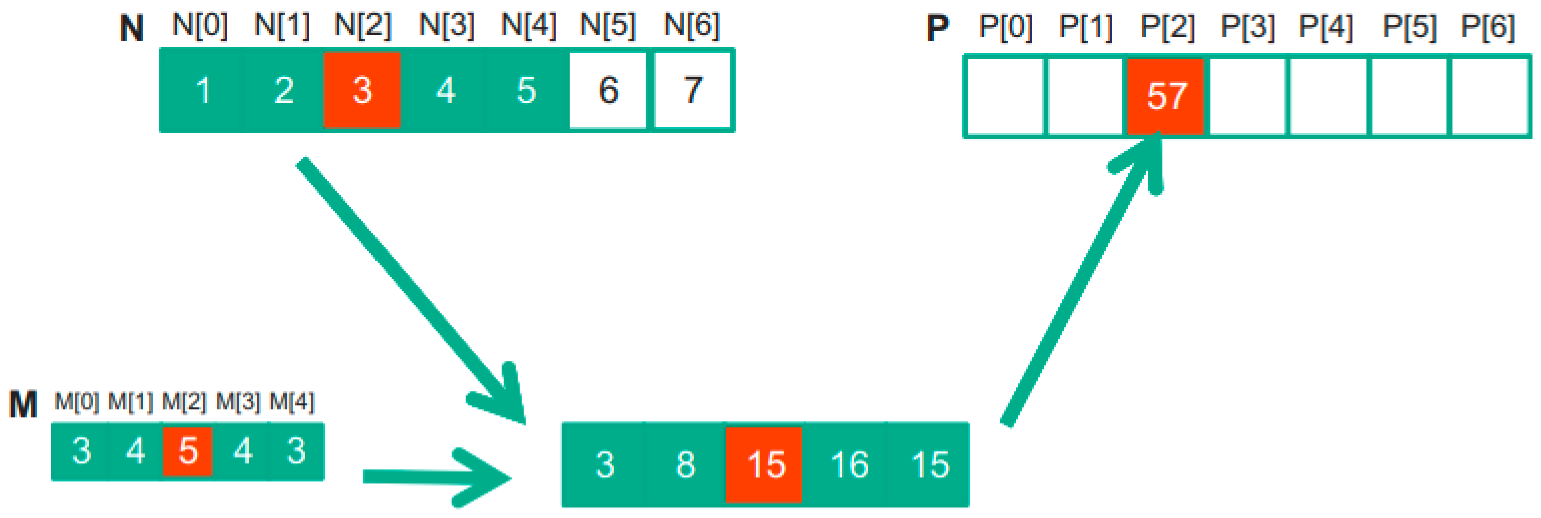
- Mathematically, convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements.

- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the **convolution mask** **OR** **convolution kernel**.

- The same convolution mask is typically used for all elements of the array.

1D Sequential Convolution

- The following example shows a convolution example for 1D data where a ***five-element convolution mask array M*** is applied to a ***seven-element input array N***.


N
N[0] N[1] N[2] N[3] N[4] N[5] N[6]
1 2 3 4 5 6 7
P
P[0] P[1] P[2] P[3] P[4] P[5] P[6]
57
M
M[0] M[1] M[2] M[3] M[4]
3 4 5 4 3
3 8 15 16 15

- The fact that we use a five-element mask *M* means that each *P* element is generated by a weighted sum of the corresponding *N* element, up to two elements to the left and up to two elements to the right.

- Each weight value is multiplied to the corresponding *N* element values before the products are summed together.

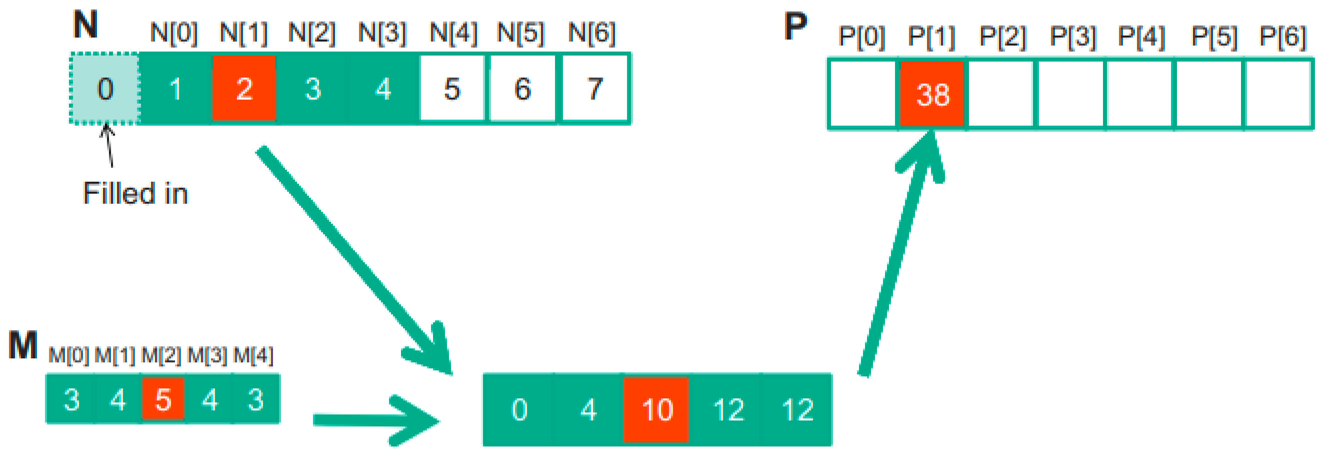
$$
\begin{aligned}
P[2] &= N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4] \\
&= 1*3 + 2*4 + 3*5 + 4*4 + 5*3 \\
&= 57
\end{aligned}
$$

- In general, the size of the mask tends to be an odd number, which makes the weighted sum calculation symmetric around the element being calculated.

1D Sequential Convolution

- Because convolution is defined in terms of neighboring elements, ***boundary conditions*** naturally exist for output elements that are close to the ends of an array.

- For example, when we calculate P[1], there is only one N element to the left of N[1]. That is, there are not enough N elements to calculate P[1] according to our definition of convolution. A typical approach to handling such a boundary condition is to define a default value to these missing N elements. For most applications, the default value is 0.



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

- These missing elements are typically referred to as ***ghost elements*** in literature.

1D Parallel Convolution – A Basic Algorithm

- The calculation of all output *(P)* elements can be done in parallel in a 1D convolution.

- **The first step is to define the major input parameters for the kernel**. We assume that the 1D convolution kernel receives five arguments: *pointer to input array N*, *pointer to input mask M*, *pointer to output array P*, *size of the mask Mask_Width*, and *size of the input and output arrays Width*. Thus, we have the following set up:

```
__global__ void convolution_1D_basic_kernel(float *N, float
 *M, float *P,
int Mask_Width, int Width) {
// kernel body
}
```

- **The second step is to determine and implement the mapping of threads to output elements**. Since the output array is one dimensional, a simple and good approach is to organize the threads into a 1D grid and have each thread in the grid calculate one output element.

```
__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;

}
```

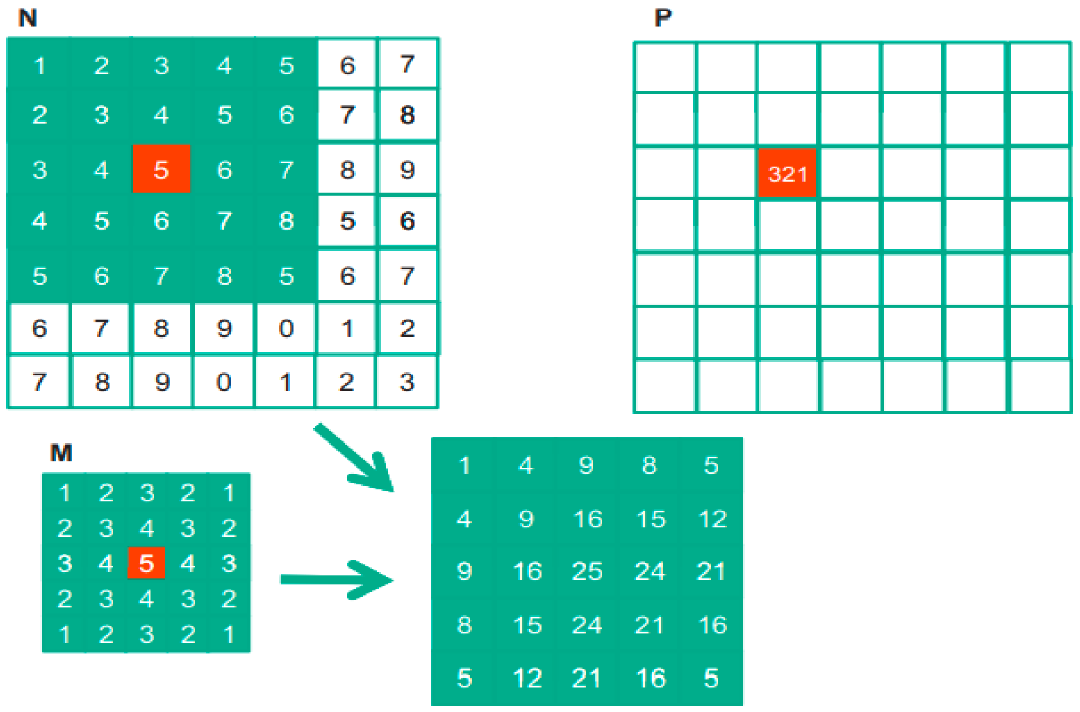
- We assume that *Mask_Width* is an odd number and the convolution is symmetric

- The *for* loop accumulates all the contributions from the neighboring elements to the output *P* element.

- The *if* statement in the loop tests if any of the input *N* elements used are ghost elements, either on the left side or the right side of the *N* array.

2D Sequential Convolution

- For image processing and computer vision, input data is usually in ***2D form***, with pixels in an x-y space. Image convolutions are performed using a ***2D convolution mask M***.
- The *x* and *y* dimensions of mask *M* determine the **range of neighbors to be included** in the weighted sum calculation.
- In general, the mask does not have to be a square array. To generate an output element, we take the subarray of which the center is at the corresponding location in the input array N. We then perform pairwise multiplication between elements of the input array and those of the mask array.



N

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

P

		321				

M

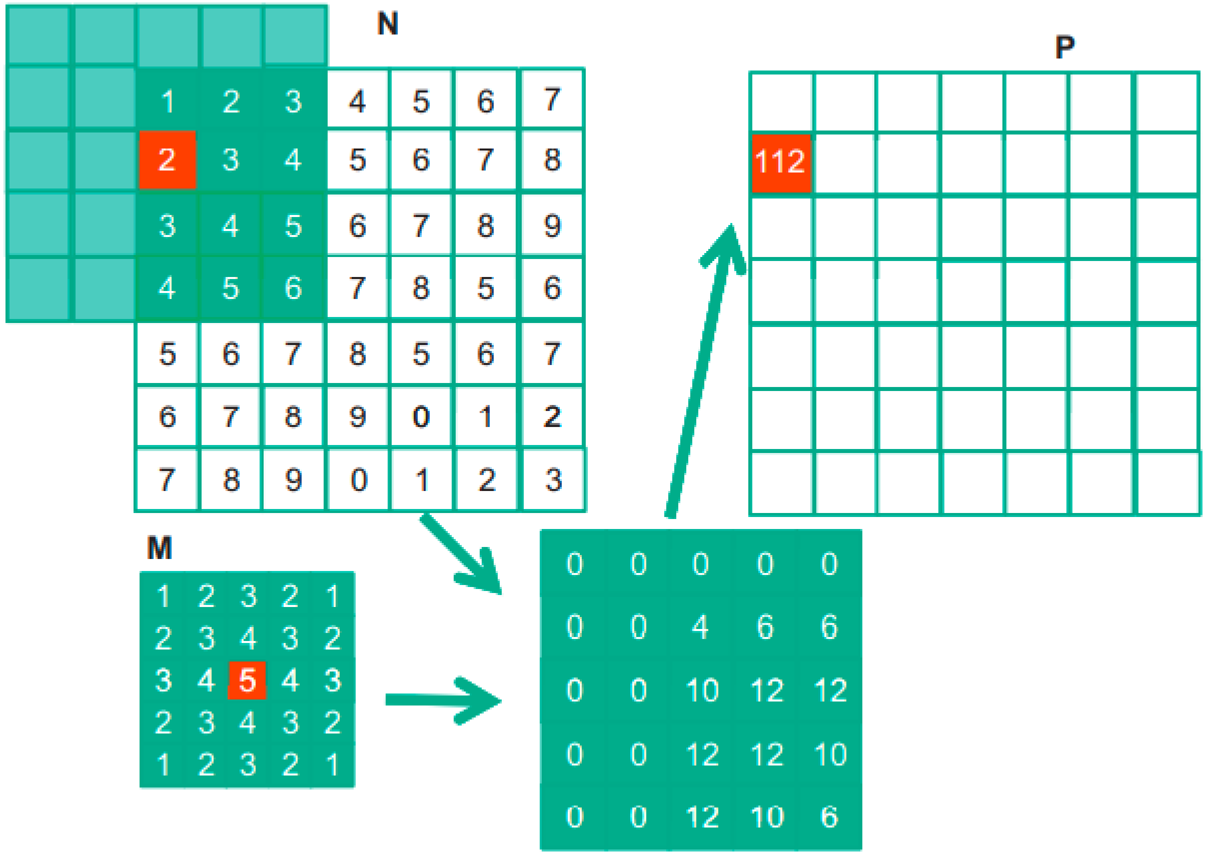
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

$$\begin{aligned}
P_{2,2} = {} & N_{0,0}*M_{0,0} + N_{0,1}*M_{0,1} + N_{0,2}*M_{0,2} + N_{0,3}*M_{0,3} + N_{0,4}*M_{0,4} \\
& + N_{1,0}*M_{1,0} + N_{1,1}*M_{1,1} + N_{1,2}*M_{1,2} + N_{1,3}*M_{1,3} + N_{1,4}*M_{1,4} \\
& + N_{2,0}*M_{2,0} + N_{2,1}*M_{2,1} + N_{2,2}*M_{2,2} + N_{2,3}*M_{2,3} + N_{2,4}*M_{2,4} \\
& + N_{3,0}*M_{3,0} + N_{3,1}*M_{3,1} + N_{3,2}*M_{3,2} + N_{3,3}*M_{3,3} + N_{3,4}*M_{3,4} \\
& + N_{4,0}*M_{4,0} + N_{4,1}*M_{4,1} + N_{4,2}*M_{4,2} + N_{4,3}*M_{4,3} + N_{4,4}*M_{4,4} \\
= {} & 1*1 + 2*2 + 3*3 + 4*2 + 5*1 \\
& + 2*2 + 3*3 + 4*4 + 5*3 + 6*2 \\
& + 3*3 + 4*4 + 5*5 + 6*4 + 7*3 \\
& + 4*2 + 5*3 + 6*4 + 7*3 + 8*2
\end{aligned}$$

2D Sequential Convolution

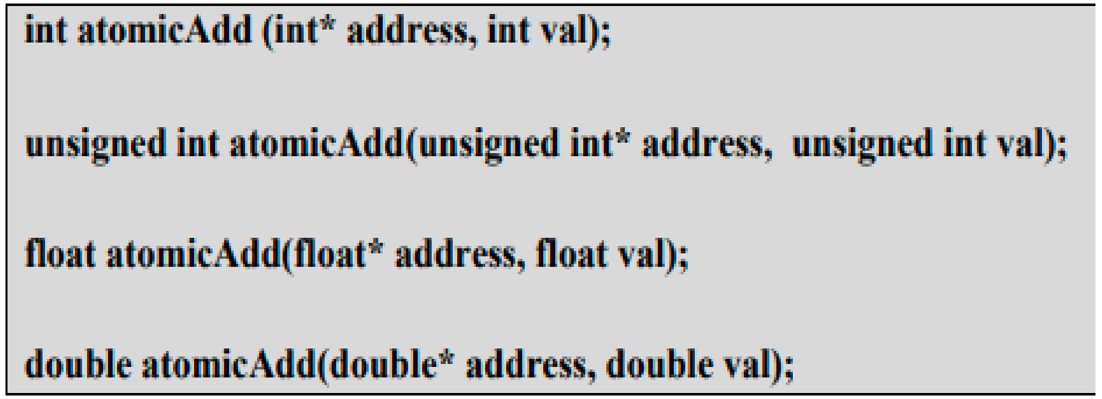
- Like 1D convolution, 2D convolution must also deal with boundary conditions.
- With boundaries in both the *x* and *y* dimensions, there are more complex boundary conditions: the calculation of an output element may involve boundary conditions along a horizontal boundary, a vertical boundary, or both.
- In the following example, the calculation of $\boldsymbol{P_{1,0}}$ involves two missing columns and one missing horizontal row in the subarray of *N*.



Atomic and Arithmetic functions

In a multithreaded scenario, if multiple threads try to modify a single **shared memory variable**, the issue of data inconsistency will arise. To overcome this atomic functions need to be used:

atomicAdd()

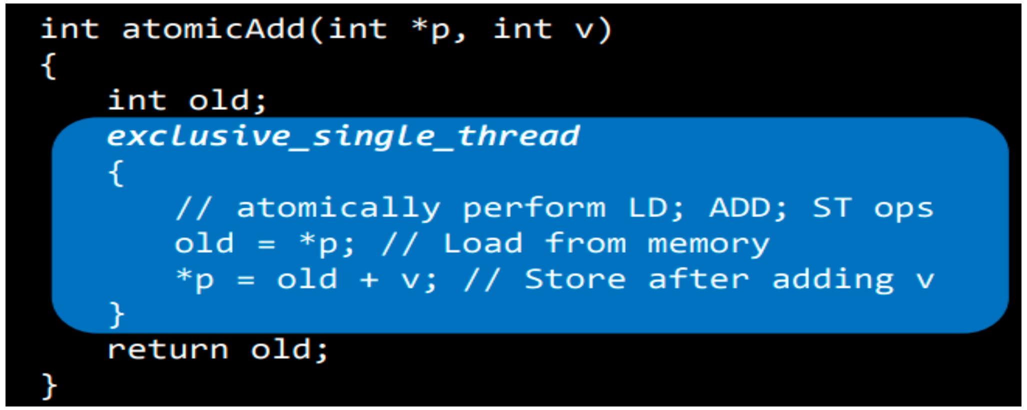


```
int atomicAdd (int* address, int val);

unsigned int atomicAdd(unsigned int* address,  unsigned int val);

float atomicAdd(float* address, float val);

double atomicAdd(double* address, double val);
```



```
int atomicAdd(int *p, int v)
{
    int old;
    exclusive_single_thread
    {
        // atomically perform LD; ADD; ST ops
        old = *p; // Load from memory
        *p = old + v; // Store after adding v
    }
    return old;
}
```

- **Reads** the word ***old*** from the ***address*** located in global or shared memory, computes **(old + val)**, and **stores** the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns ***old***.

Atomic and Arithmetic functions

atomicSub()

```
int atomicSub(int* address, int val);

unsigned int atomicSub(unsigned int* address,  unsigned int val);
```

- **Reads** the word ***old*** located at the ***address*** in global or shared memory, computes **(old - val)**, and **stores** the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns ***old***.

Atomic and Arithmetic functions

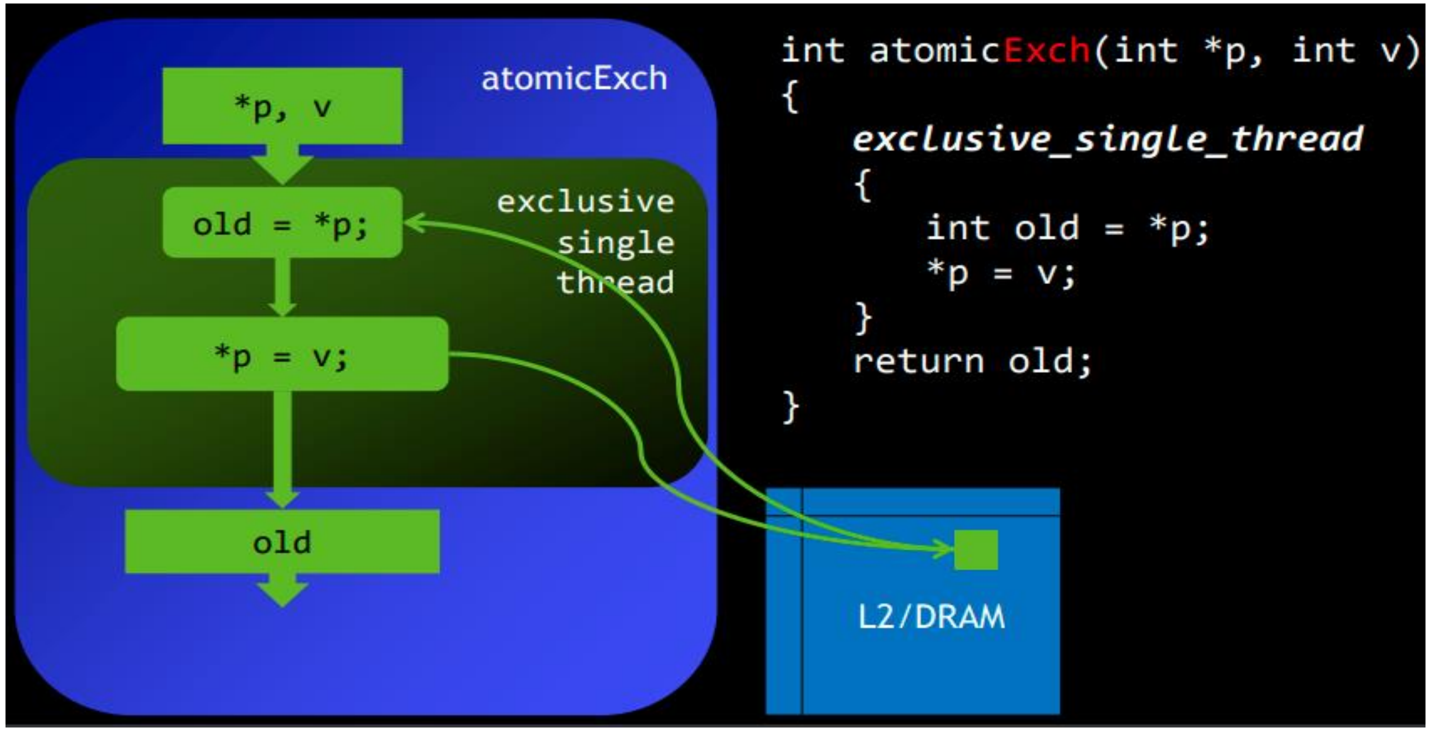
atomicExch()

```
int atomicExch(int* address, int val);

unsigned int atomicExch(unsigned int* address,  unsigned int val);

float atomicExch(float* address, float val);
```

- **Reads** the word ***old*** located at the ***address*** in global or shared memory and **stores** ***val*** in memory at the same address. These two operations are performed in one atomic transaction. The function returns ***old***.



```
int atomicExch(int *p, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        *p = v;
    }
    return old;
}
```

Atomic and Arithmetic functions

atomicMin()

```
int atomicMin(int* address, int val);

unsigned int atomicMin(unsigned int* address,  unsigned int val);
```

- **Reads** the word ***old*** located at the ***address*** in global or shared memory, computes the **minimum** of ***old*** and ***val***, and **stores** the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns ***old***.

atomicMax()

```
int atomicMax(int* address, int val);

unsigned int atomicMax(unsigned int* address,  unsigned int val);
```

- **Reads** the word ***old*** located at the ***address*** in global or shared memory, computes the **maximum** of ***old*** and ***val***, and **stores** the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns ***old***.

Atomic and Arithmetic functions

atomicInc()

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
```

- **Reads** the word ***old*** located at the ***address*** in global or shared memory, computes **((old >= val) ? 0 : (old+1))**, and **stores** the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns ***old***.

atomicDec()

```
unsigned int atomicDec(unsigned int* address, unsigned int val);
```

- **Reads** the word ***old*** located at the ***address*** in global or shared memory, computes **((old == 0) || (old > val)) ? Val : (old-1)** , and **stores** the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns ***old***.

A CUDA program to read a string and determines the number of occurrences of a character ‘a’ in the string using **atomicAdd()** function.

```
#include "cuda_runtime.h“
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#define N 1024

__global__ void CUDACount(char* A, unsigned int *d_count)
{
		int i = threadIdx.x;
			if(A[i]==’a’)
				atomicAdd(d_count,1);
 }
```

A CUDA program to read a string and determines the number of occurrences of a character ‘a’ in the string using **atomicAdd()** function.

```
int main() {
char A[N]; char *d_A; unsigned int *count=0,*d_count,*result;
printf("Enter a string");
gets(A);
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaMalloc((void**)&d_A, strlen(A)*sizeof(char));
cudaMalloc((void **)&d_count,sizeof(unsigned int));
cudaMemcpy(d_A, A, strlen(A)*sizeof(char), cudaMemcpyHostToDevice);
cudaMemcpy(d_count,count,sizeof(unsigned int),cudaMemcpyHostToDevice);

cudaError_t error =cudaGetLastError();
if (error != cudaSuccess) {
		printf("CUDA Error1: %s\n", cudaGetErrorString(error));
}
CUDACount<<1, strlen(A)>>(d_A,d_count);
error =cudaGetLastError();
if (error != cudaSuccess) {
		printf("CUDA Error2: %s\n", cudaGetErrorString(error));
}
cudaEventRecord(stop, 0);
```

```
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start,
stop);

cudaMemcpy(result, d_count, sizeof(unsigned
int), cudaMemcpyDeviceToHost);

printf("Total occurences of a=%u",result);
printf("Time Taken=%f",elapsedTime);
cudaFree(d_A);
cudaFree(d_count);
printf("\n");
getch();
return 0;
}
```