EASTERN WASHINGTON UNIVERSITY
start something big

Mapping Threads to 2D Data

Computer Science Department
Eastern Washington University
Yun Tian (Tony) Ph.D.

Outline Today

- Multi-dimensional grid and blocks
- Thread Organization
- Mapping Threads to 2D Data
- Case Study
- Linearize 2D matrix and store it as a 1D array.

Thread Organization

- Fine-grained, data parallel threads are the fundamental means of parallel execution in CUDA.
  - Thread Hierarchical Organization, like US telephone sys.
    - Grid consists of thread blocks
    - Block consists of up to 1024 threads, (comp. capability 2.0 and above)
      - Blocks are independent.
      - Threads in blocks can coordinate each other(more later)
  - Unlike parallel code on CPU using pthread.

Thread Organization

- gridDim, blockDim, blockIdx and threadIdx built in variables.
  - Preinitialized by CUDA runtime system.
  - Used to identify each thread in grid.
  - We learned one dimensional grid.

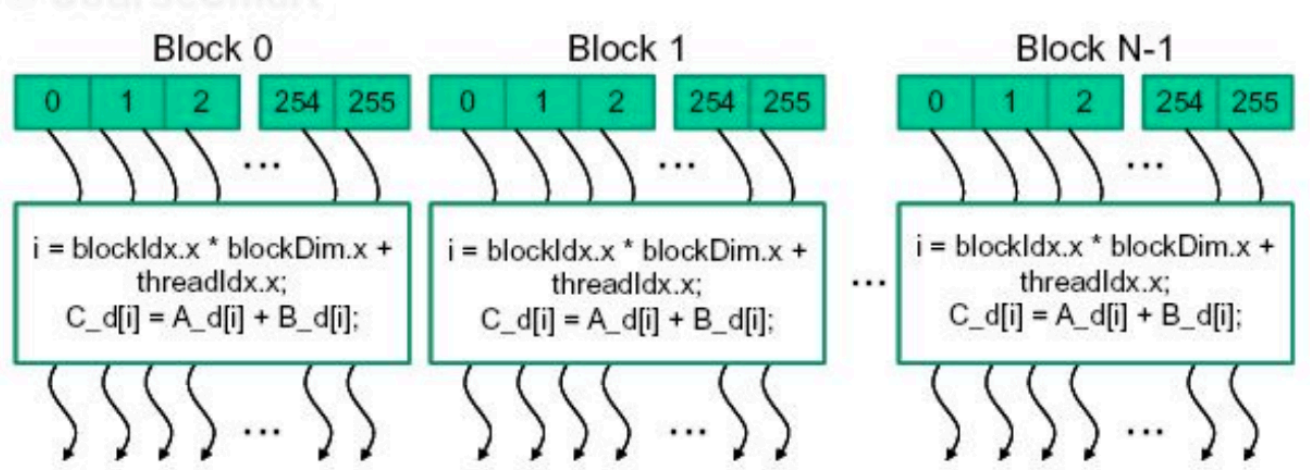


FIGURE 3.10

All threads in a grid execute the same kernel code.

Thread Organization

- In general, a grid is a 3D of blocks. Each block is a 3D array of threads.(comp. capability 2.0 and above)
  - Use fewer dimensions by setting the unused dimension to 1.
  - Organization of a grid is determined by the execution configuration parameters.
    - During kernel launch, shown as <<< p1, p2 >>>
    - p1 specifies the dimensions of the grid in # of blocks.
    - p2 specifies the dimensions of each block in # of threads.
    - p1 and p2 are of dim3 type, a C struct that has x, y, z three integer fields.

Thread Organization

- New way to launch the vecAddkernel() and generate a 1D grid that consists of 128 blocks, each of which consists of 32 threads.
  - Total # of threads in the grid is 128 * 32 = 4096.

```
……
dim3 dimGrid(128, 1, 1);
dim3 dimBlock(32, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(……);
……
//in the textbook P65, the description about this has error.
//dimGrid and dimBlock could be replaced by other variable names.( chose your
// own names.)
```

Thread Organization

- Typically, the number of blocks and the dimension of the grid depends on the property of data.
  - Data dimension and data size.

```
……
dim3 dimGrid(ceil( n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(……);
……
//# of blocks vary with the size of the vectors so that the grid has enough
//threads to cover all vector elements.
// if n = 1000, # of blocks is 4.      if n = 1001, # of block is 5.
```

Thread Organization

- CUDA provides shortcut for launching a kernel with 1D grids and blocks. Instead using dim3 variables.
  - We used this in Lab1 and Lab2.

```
……
vecAddKernel<<<ceil(n/256.0), 256>>>(……);
……
//Cuda C compiler takes the parameters as the x dimensions, and
// assumes that y and z dimensions are 1.
```

Thread Organization

- In CUDA C, the allowed values of gridDim.x, gridDim.y and gridDim.z range from 1 to 65536.
  - All threads in a block share the same blockIdx.x, blockIdx.y and blockIdx.z
  - 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.

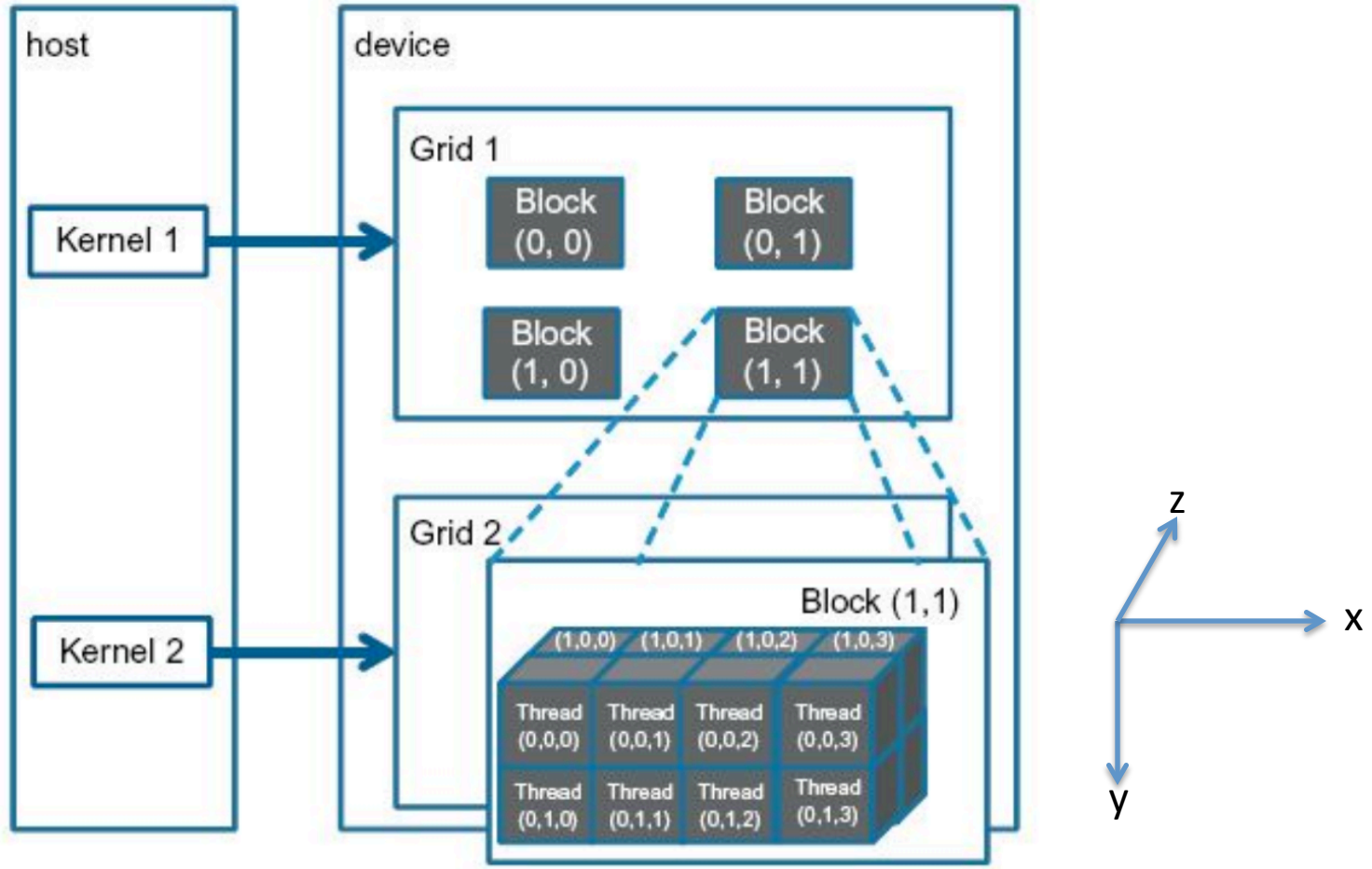
Thread Organization

- For the rest of this book, we will use the notation(x, y, z) for a 3D grid with x blocks in the x direction, y blocks in the y direction, and z blocks in the z direction.
- The number of threads **in each dimension** of a block is specified by the second execution configuration parameter at the kernel launch.

Thread Organization

- Total size of a block is limited to 1024 threads, distributed into the three dimensions. (comp. capability 2.0 and above)
  - As long as the total # of threads <= 1024.
  - E.g. blockDim(512, 1, 1), blockDim(8, 16, 4) and (32, 16, 2) are allowed values.
  - But blockDim(32,32,2) is not allowable.

Example of Multidimensional Grid



Example of Multidimensional Grid

- Four blocks organized into a 2 by 2 array.
  - Block(1, 0), x coordinate(horizontal axis) is 0, y coordinate (vertical axis) is 1.
    - blockIdx.y = 1 and blockIdx.x = 0
    - The **notation** Bock(1, 0) with y coordinate going first. The reversed order of the execution configurations.

```
……
dim3 dimGrid( 2, 2, 1);
dim3 dimBlock(4, 2, 2);
kernelFunction<<< dimGrid, dimBlock >>>(……);
……
```

Example of Multidimensional Grid

- Each threadIdx has three fields
  - threadIdx.x, the x coordinate, threadIdx.y, the y coordinate, threadIdx.z, the z coordinate
  - Each block is organized into 4 by 2 by 2 arrays of threads.
  - We use this small grid to for simplicity. Typically CUDA grid contains thousands to millions of threads.

Mapping Threads to 2D Data

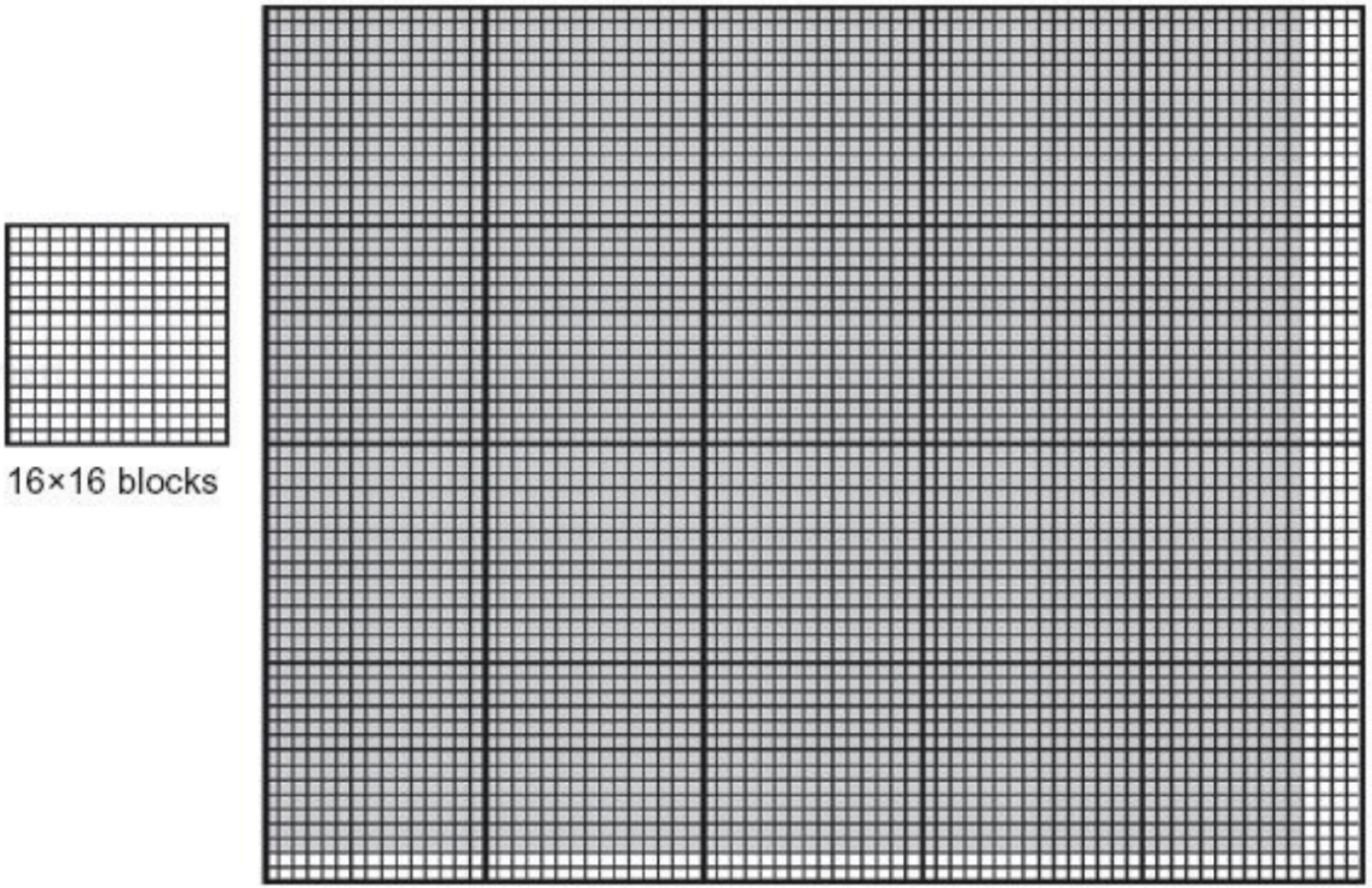


FIGURE 4.2

Using a 2D grid to process a picture.

Mapping Threads to 2D Data

- Pictures are 2D arrays.
- Convenient to use a 2D grid that consists of 2D blocks to process the pixels of a picture.
- Example in the previous slide shows such an arrangement.
  - We like to scale each pixel value by 2 in this example.

Mapping Threads to 2D Data

- 76 by 62 picture,(76 in x direction, 62 pixels in y direction).
- We use 16 * 16 block.
- How many block we need in x direction and y direction in order for the grid to cover the whole picture?

Five blocks in x direction, ceil(76 / 16) = 5
Four block in the y direction. Ceil(62 / 16) = 4
We result in total of 20 blocks in diagram.

Mapping Threads to 2D Data

- 76 by 62 picture,(76 in x direction, 62 pixels in y direction). We use 16 * 16-block.
- Four extra threads in x direction, and two extra in the y direction.
  - We actually generate 80 * 64 threads to process 76 * 62 pixels.
  - Similarly, we expect an **if** statements in kernel to test whether the thread indices threadIdx.x and threadIdx.y fall within the valid range of pixels.

Mapping Threads to 2D Data

- 76 by 62 picture,(76 in x direction, 62 pixels in y direction). We use 16 * 16 block.

```
dim3 dimGrid( ceil( n/ 16.0), ceil( m/ 16.0), 1);
 dim3 dimBlock( 16, 16, 1);
pictureKernel<<<dimGrid, dimBlock>>>( d_ Pin, d_ Pout, n, m);
//n is number of columns in picture, here is 76
//m is the number of rows in picture, here is 62
```

2D array stored in 1D linear mem

- Ideally we like to access 2D array d_Pin as a 2D array, using d_Pin[i][j], for pixel at row i and column j.
- But it is very hard to setup double pointers on GPU.
- Reasons for not using 2D array or pointer to pointers in Kernel.
  - We need to know the number columns of 2D array in advance at compile time. (Sometimes this is not possible)

2D array stored in 1D linear mem

- Reasons for not using 2D array or pointer to pointers in Kernel.
  - We need to know the number columns of 2D array in advance at compile time. (Sometimes this is not possible)
  - cudaMalloc() and cudaMemCopy() supports pointer directly.
    - Feasible but very **painful** to setup double pointer on device. (hard and slow in performance)

2D array stored in 1D linear mem

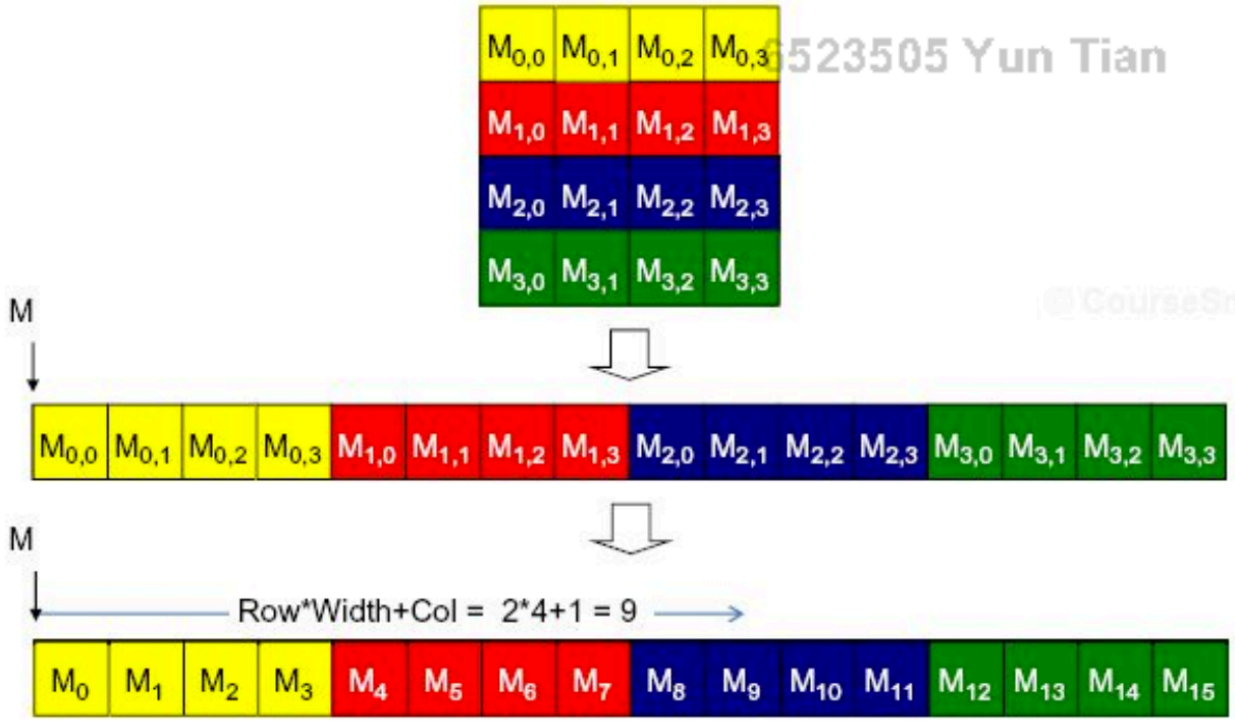


FIGURE 4.3

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.

2D array stored in 1D linear mem

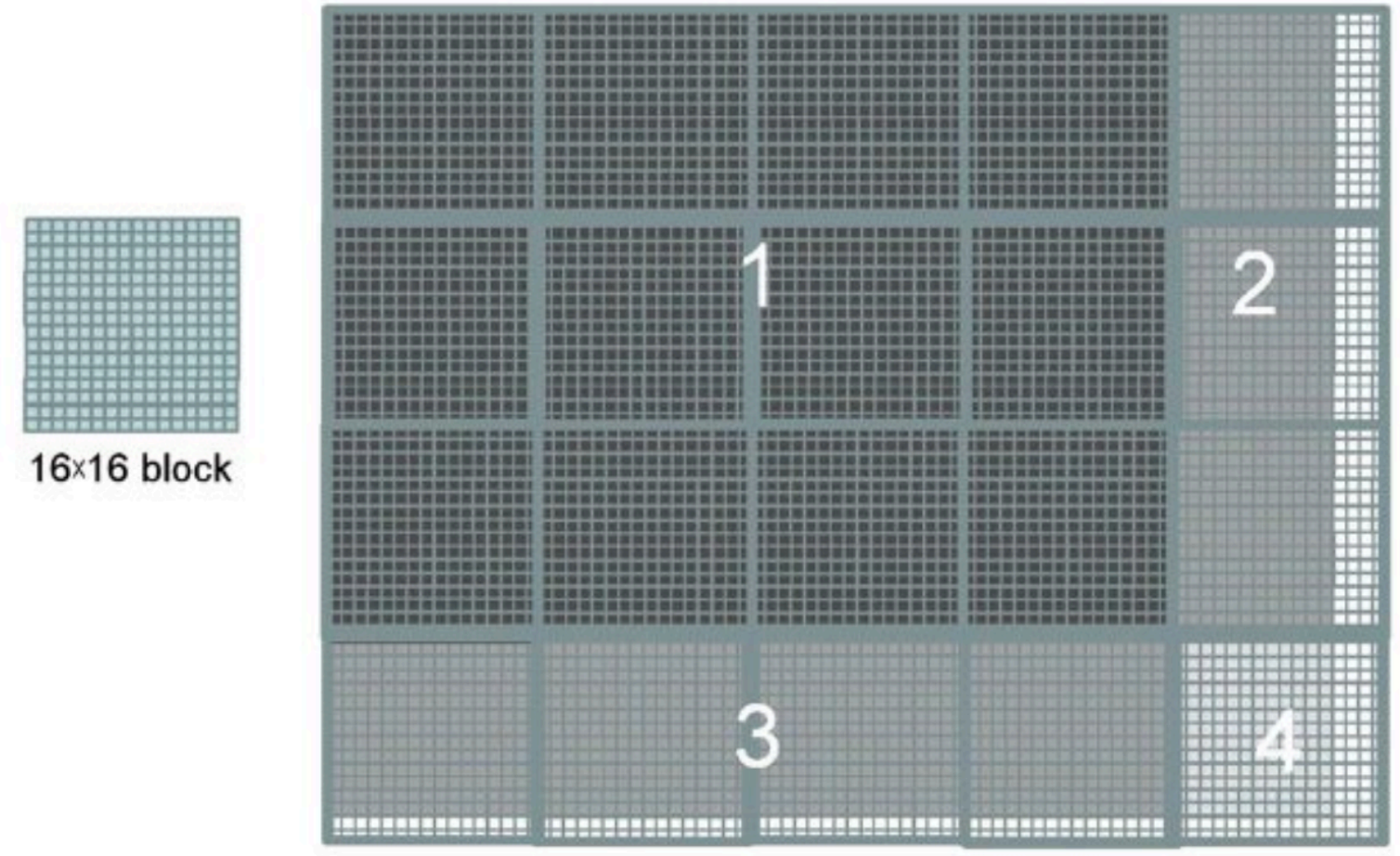
- Instead, we linearize our 2D array, so that each row of 2D array is stored one after another in a 1D array(or pointer).
  - Instead using M[i][j] to access element in 2D matrix M,
  - We use M_1D[i * columnWidth + j] to access an element in the original M.
  - This is called row-major layout of storage of 2D array.
  - By default, C and Java use row-major storage.

Mapping Threads to 2D Data

```
__global__ void PictureKernel(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];
  }

}
```

// n is column width, m is # of rows.
// Row is the global thread id in y direction.
// Col is the global thread id in x direction.

Mapping Threads to 2D Data



Summary

- Multi-dimensional grid and blocks
- Thread Organization
- Mapping Threads to 2D Data
- Case Study
- Linearize 2D matrix and store it as a 1D array.

Next Class

- Set up double pointers on Device
- Matrix multiplication case study.