

ECE408 / CS483 / CSE408  
Summer 2024

Applied Parallel Programming

Lecture 3: Kernel-Based  
Data Parallel Execution Model

# What Will You Learn Today?

- more about the multi-dimensional logical organization of CUDA threads
- to use control structures, such as loops in a kernel
- concepts
  - thread scheduling
  - latency tolerance
  - hardware occupancy

# Review – Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int N)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(N/256.0) blocks of 256 threads each
    dim3 DimGrid(ceil(N/256.0), 1, 1);
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<DimGrid, DimBlock>>>(A_d, B_d, C_d, N);
}
```

**A** Number of blocks per dimension

**B** Number of threads per dimension in a block

**C** Unique block # in x dimension

**D** Number of threads per block in x dimension

**E** Unique thread # in x dimension in the block

Q: How many threads in total will be executed in this example?

Thread Block 0      Thread Block 1      Thread Block gridDim.x-1

0 1 2 ... 254 255      0 1 2 ... 254 255      0 1 2 ... 254 255

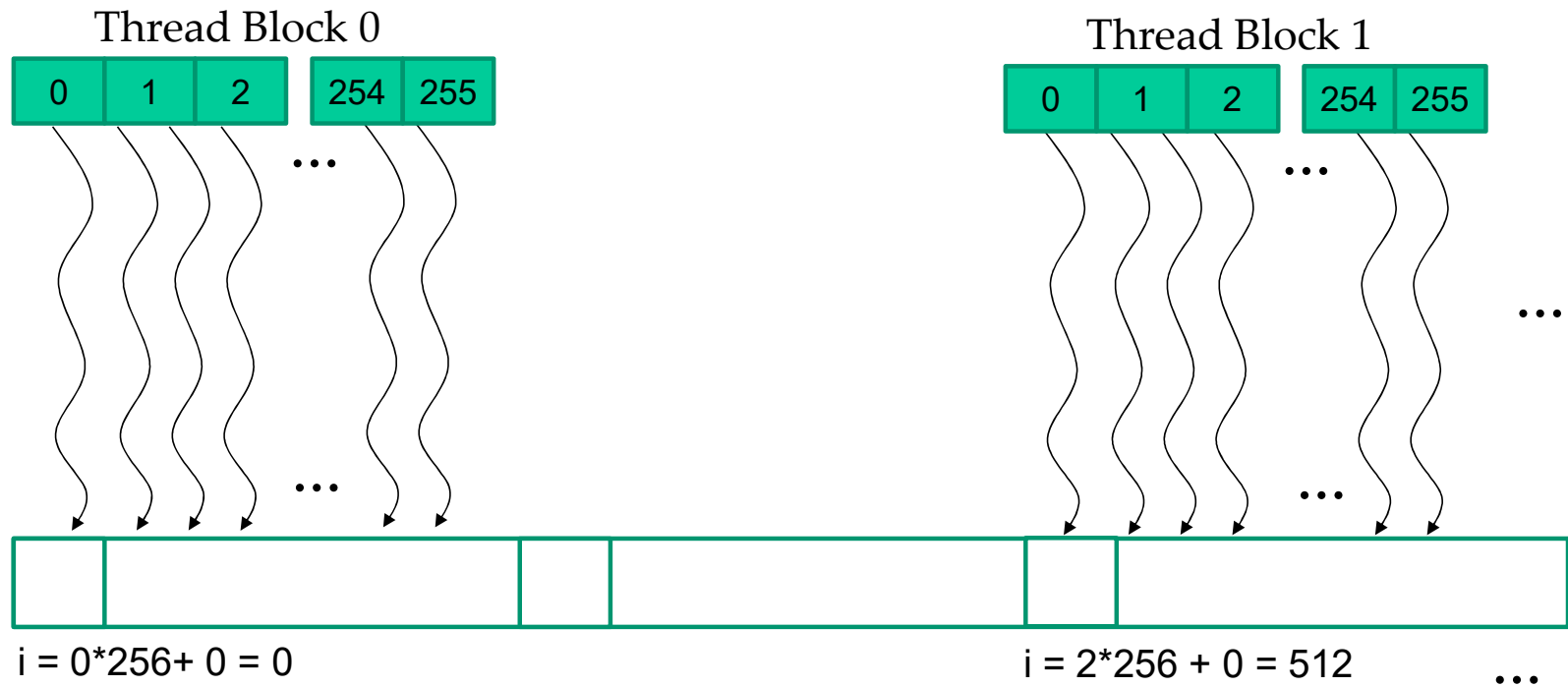
...

$i = 0 \cdot 256 + 0 = 0$        $i = 1 \cdot 256 + 0 = 256$        $i = 3 \cdot 256 + 0 = 768$       ...

# Coarser Grains: Thread Assignment for vecAdd with Two Elements per Thread

```
vecAdd<<<ceil(N/(2*256.0)), 256>>>(...)
```

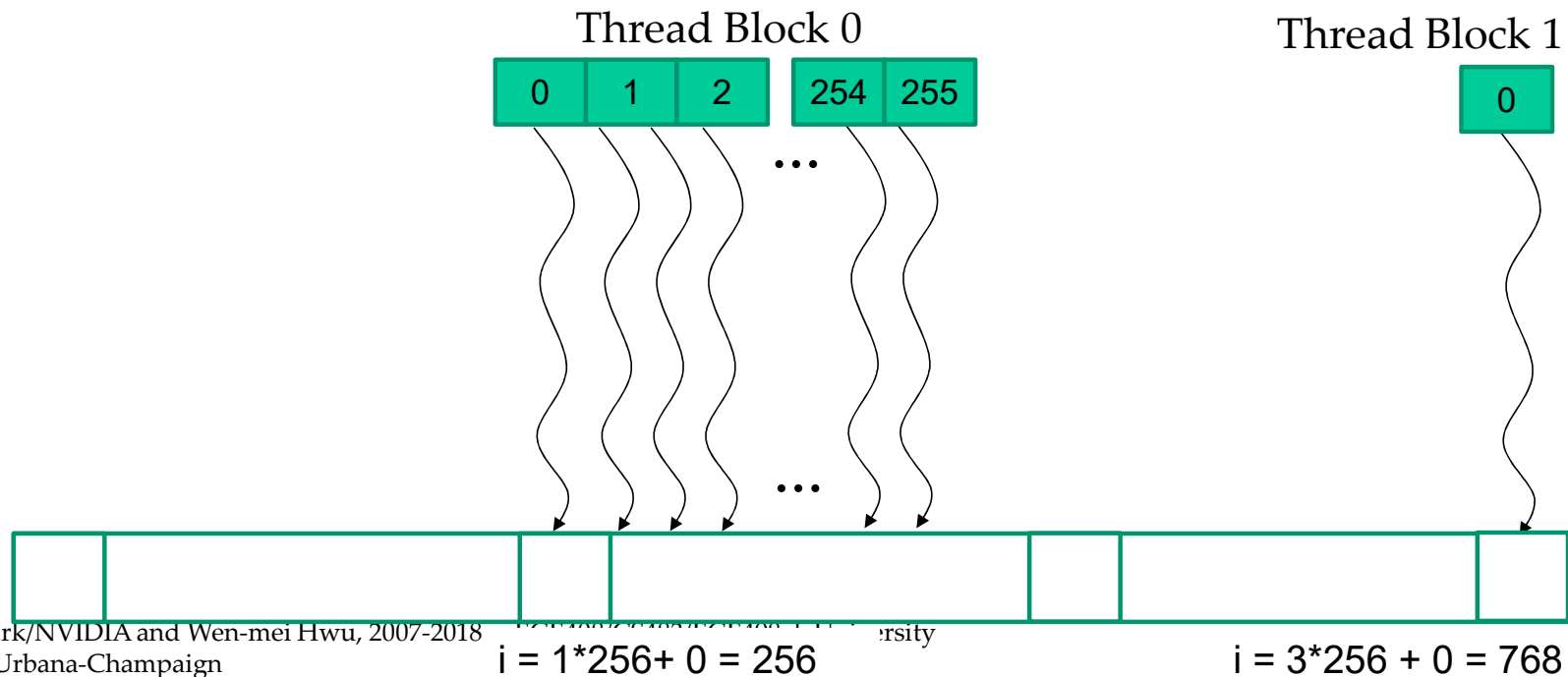
```
i = blockIdx.x * (2*blockDim.x) + threadIdx.x;  
if (i<n) C[i] = A[i] + B[i];
```



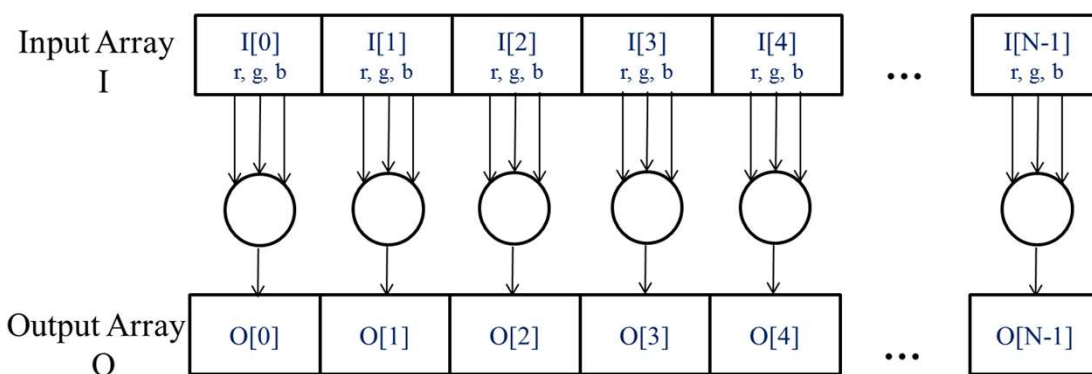
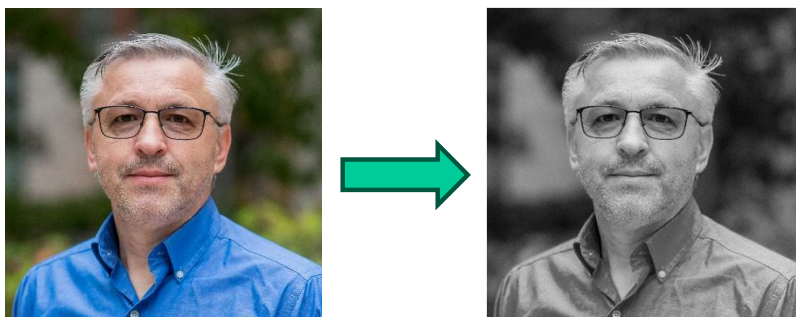
# Coarser Grains: Thread Assignment for vecAdd with Two Elements per Thread

```
vecAdd<<<ceil(N/(2*256.0)), 256>>>(...)
```

```
i = blockIdx.x * (2*blockDim.x) + threadIdx.x;  
if (i<n) C[i] = A[i] + B[i];  
i = i+blockDim.x;  
if (i<n) C[i] = A[i] + B[i];
```

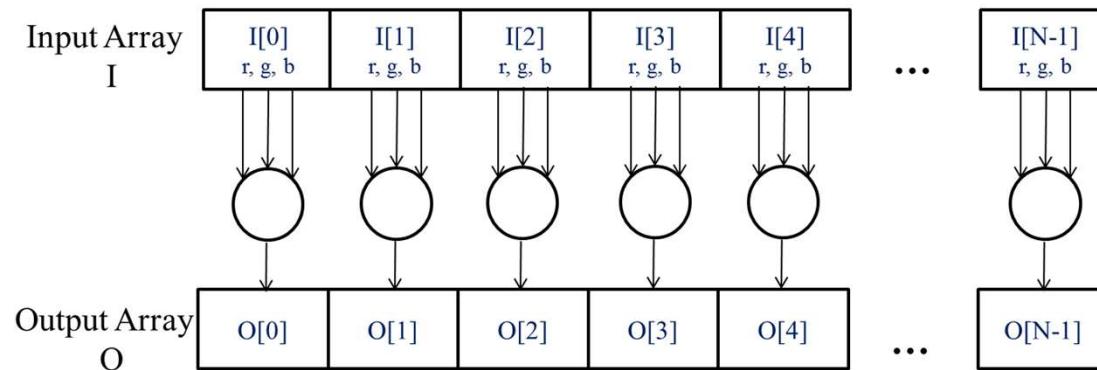


# Example 1: Conversion of a color image to a grey-scale image



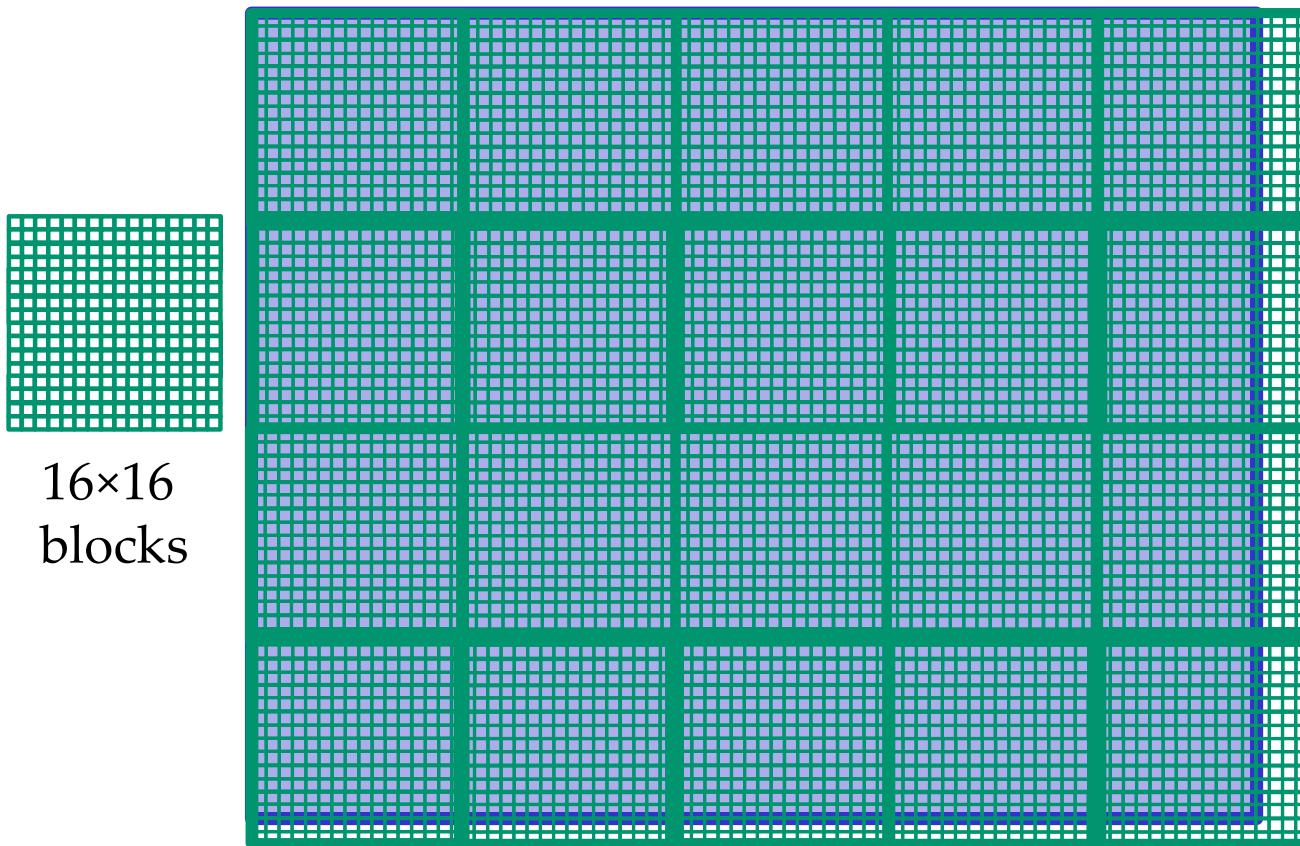
*Pixels can be  
calculated  
independently*

# *Review: Pixels can be calculated independently*

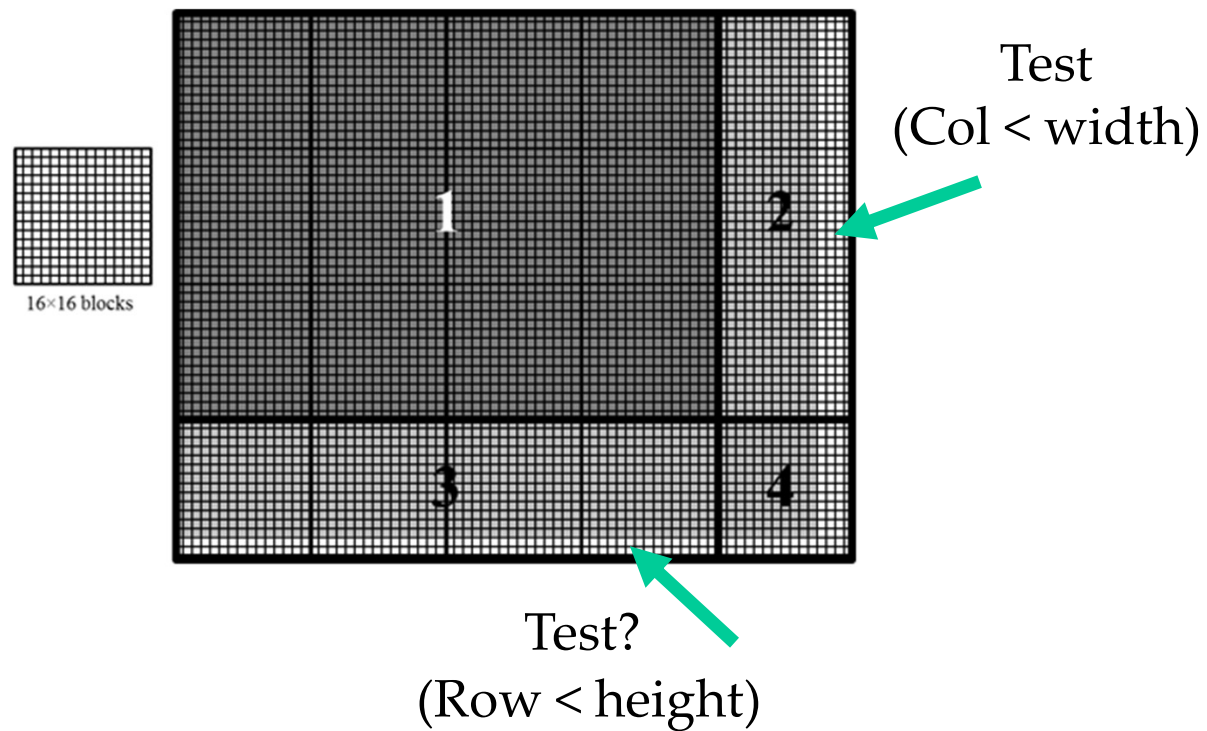




# Processing a Picture with a 2D Grid



# Covering a 76×62 picture with 16×16 blocks



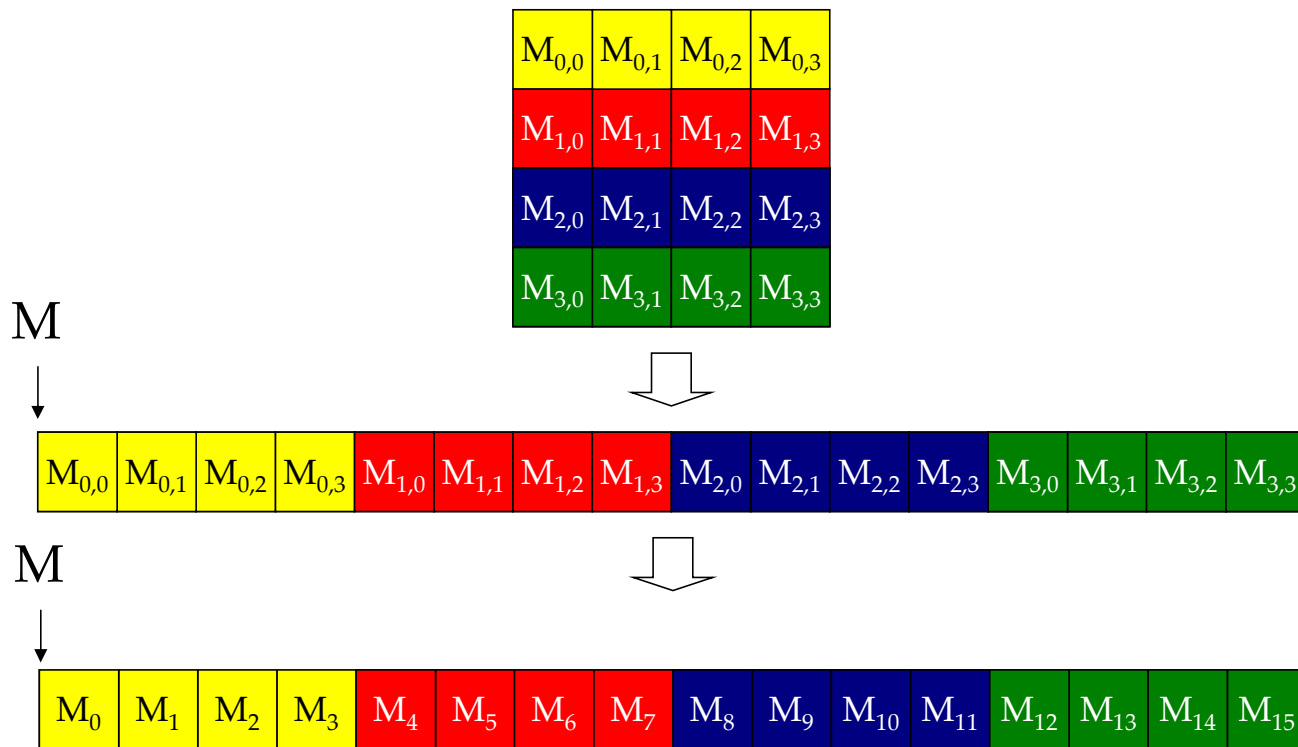
# colorToGreyscaleConversion Kernel with 2D thread mapping to data

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * grayImage, unsigned char * rgbImage,
                                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
        // THREE times as many columns of the gray scale image
        int rgbOffset = 3 * greyOffset;
        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++



© David Kirk/NVIDIA and Wen-mei Hwu, 2011  
 Illinois, Urbana-Champaign

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

# colorToGreyscaleConversion Kernel with 2D thread mapping to data (cont'd)

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

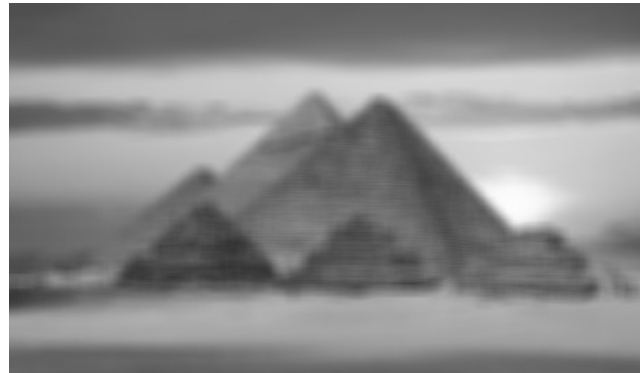
    int Col = threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;

    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
        // THREE times as many columns of the gray scale image
        int rgbOffset = 3 * greyOffset;
        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[greyOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

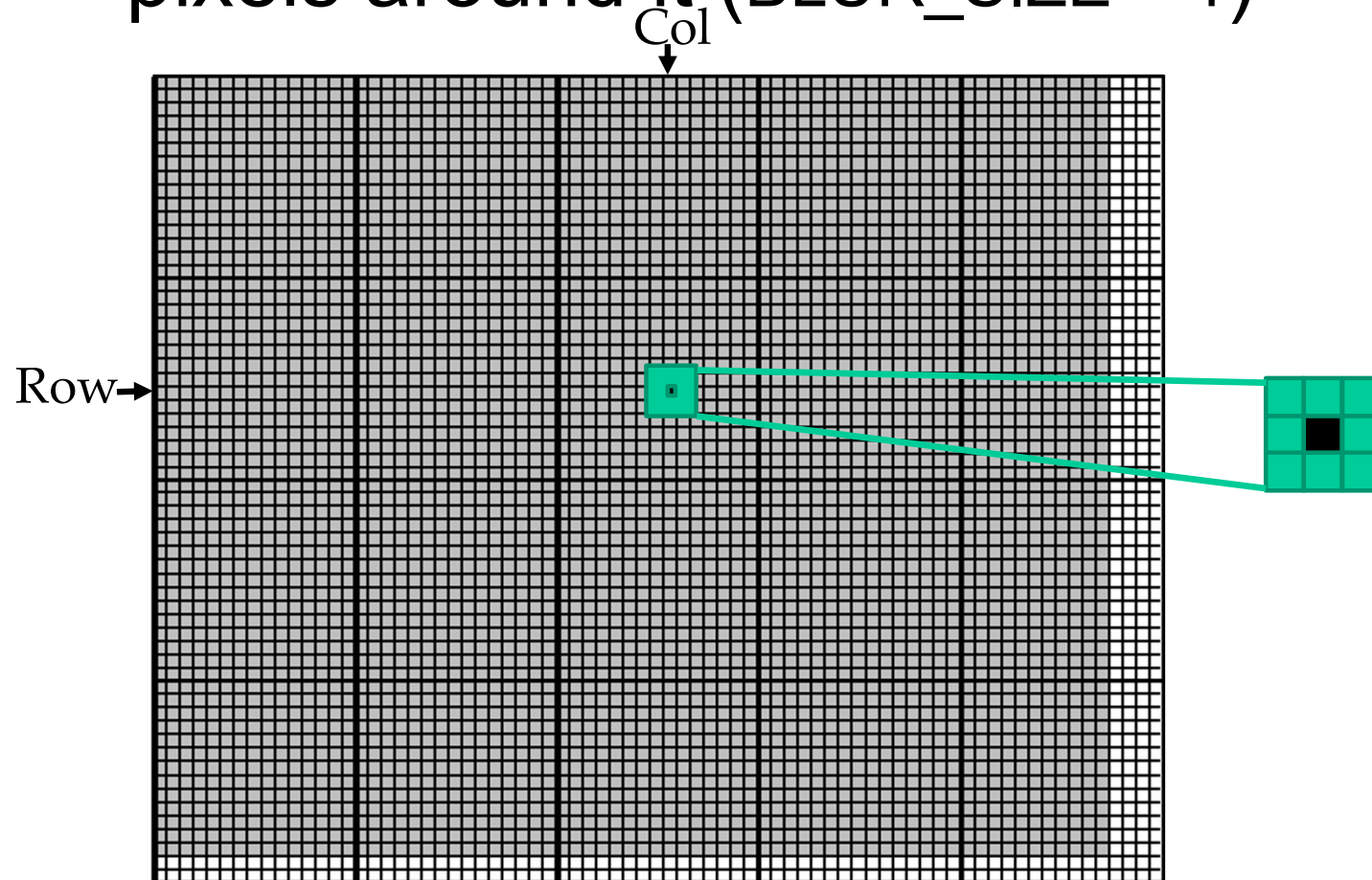
# Image Blurring (Monochrome)



(BLUR\_SIZE is 5)



Each output pixel is the average of  
pixels around it (BLUR\_SIZE = 1)



# An Image Blur Kernel

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
1.        int pixVal = 0;
2.        int pixels = 0;

        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.        for(int blurRow = -BLUR_SIZE; blurRow <= BLUR_SIZE; ++blurRow) {
4.            for(int blurCol = -BLUR_SIZE; blurCol <= BLUR_SIZE; ++blurCol) {

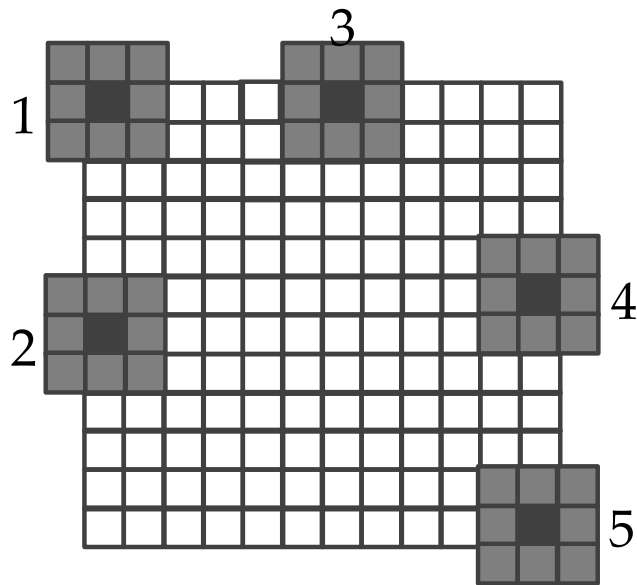
5.                int curRow = Row + blurRow;
6.                int curCol = Col + blurCol;

                // Verify we have a valid image pixel
7.                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.                    pixVal += in[curRow * w + curCol];
9.                    pixels++; // Keep track of number of pixels in the avg
                }
            }
        }

        // Write our new pixel value out
10.    out[Row * w + Col] = (unsigned char) (pixVal / pixels);
    }
}
```



# *Handling boundary conditions for pixels near the edges of the image*



# An Image Blur Kernel

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
1.        int pixVal = 0;
2.        int pixels = 0;

        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.                int curRow = Row + blurRow;
6.                int curCol = Col + blurCol;

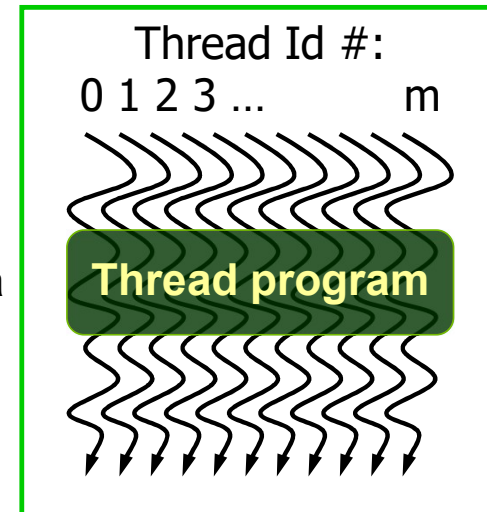
                // Verify we have a valid image pixel
7.                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.                    pixVal += in[curRow * w + curCol];
9.                    pixels++; // Keep track of number of pixels in the avg
                }
            }
        }

        // Write our new pixel value out
10.    out[Row * w + Col] = (unsigned char) (pixVal / pixels);
    }
}
```

# CUDA Thread Blocks

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 1024 concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads within block have **thread index** numbers
- Kernel code uses **thread index and block index** to select work and address shared data
- Threads in the same block **share data** and **synchronize** while doing their share of the work
- Threads in different blocks cannot cooperate
- Blocks **execute in arbitrary order!**

## CUDA Thread Block



Courtesy: John Nickolls,  
NVIDIA

# Compute Capabilities are GPU-Dependent

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOP/s	812.5	1305.6
Compute Capability	3.0	5.0
Shared Memory / SM	16KB / 48 KB	64 KB
Register File Size / SM	256 KB	256 KB
Active Blocks / SM	16	32
Memory Clock	5000 MHz	5400 MHz
Memory Bandwidth	80 GB/s	86.4 GB/s
L2 Cache Size	256 KB	2048 KB
TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm <sup>2</sup>	148 mm <sup>2</sup>
Manufacturing Process	28 nm	28 nm

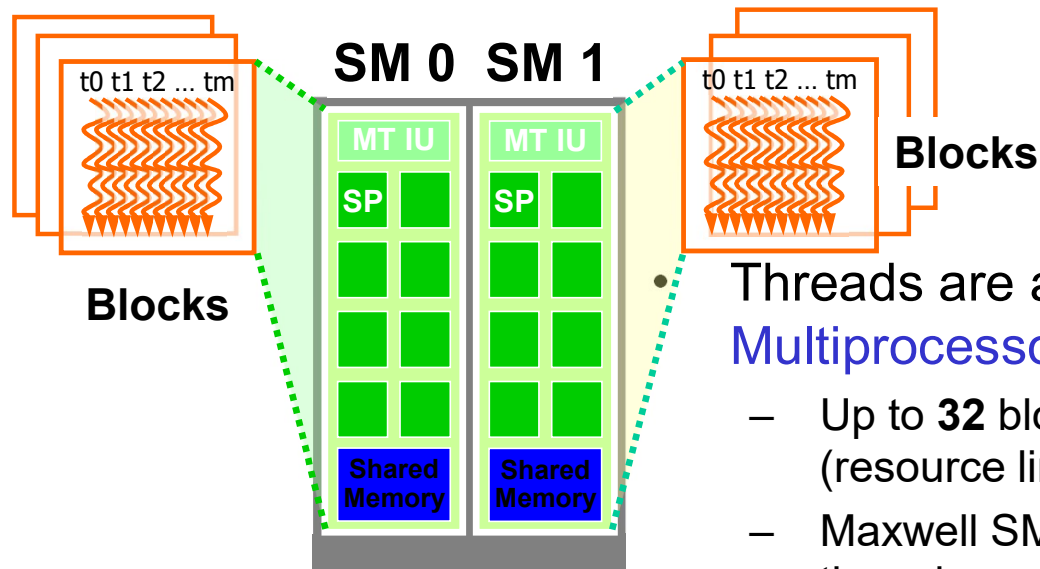
# Compute Capabilities are GPU-Dependent

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

	GK107 (Kepler)	GM107 (Maxwell)
Shared Memory / SM	16 / 48 kB	64 kB
Register File Size / SM	256 kB	256 kB
Active Blocks / SM	16	32

TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm <sup>2</sup>	148 mm <sup>2</sup>
Manufacturing Process	28 nm	28 nm

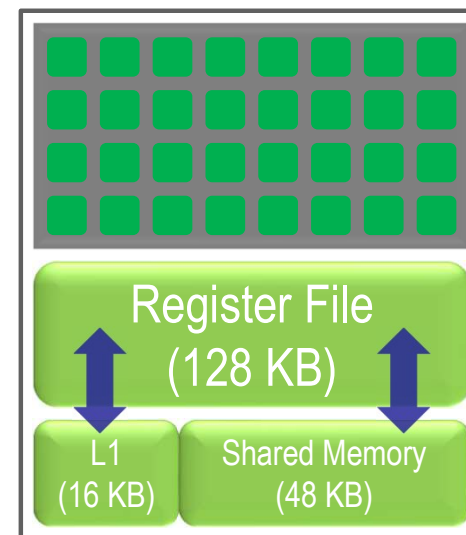
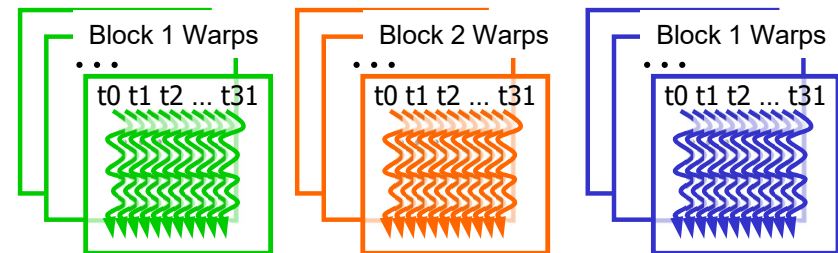
# Executing Thread Blocks



- Threads are assigned to **Streaming Multiprocessors** in block granularity
  - Up to **32** blocks to each SM (resource limit for Maxwell)
  - Maxwell SM can take up to **2048** threads
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

# Thread Scheduling (1/2)

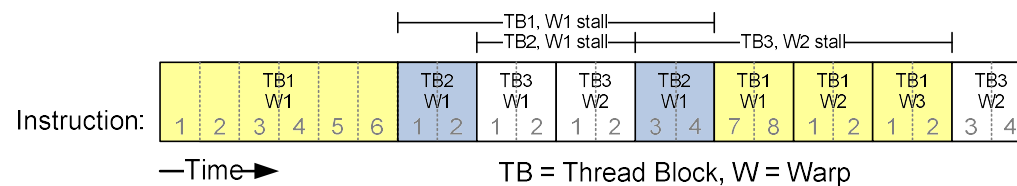
- Each block is executed as 32-thread warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are divided based on their linearized thread index
    - Threads 0-31: warp 0
    - Threads 32-63: warp 1, etc.
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
  - Each block is divided into  $256/32 = 8$  warps
  - $8 \text{ warps/blk} * 3 \text{ blks} = 24 \text{ warps}$



, University

# Thread Scheduling (2/2)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - **All threads in a warp execute the same instruction when selected**



## Example execution timing of an SM



# Pitfall: Control/Branch Divergence

- **branch divergence**
  - threads in a warp take different paths in the program
  - main performance concern with control flow
- GPUs use **predicated execution**
  - Each thread computes a yes/no answer for each path
  - **Multiple paths** taken by threads in a warp are **executed serially!**

# Example of Branch Divergence

- Common case: use of thread ID as a branch condition

```
if (threadIdx.x > 2) {  
    // THEN path (lots of lines)  
} else {  
    // ELSE path (lots more lines)  
}
```

- Two control paths (THEN/ELSE) for threads in warp

**\*\*\* ALL THREADS EXECUTE BOTH PATHS \*\*\***  
(results kept only when predicate is true for thread)

# Avoiding Branch Divergence

- Try to make branch granularity a multiple of warp size (remember, it may not always be 32!)

```
if (threadIdx.x / WARP_SIZE > 2) {  
    // THEN path (lots of lines)  
} else {  
    // ELSE path (lots of lines)  
}
```

- Still has two control paths
- But all threads in any warp follow only one path.

# Block Granularity Considerations

- For colorToGreyscaleConversion, should one use  $8 \times 8$ ,  $16 \times 16$  or  $32 \times 32$  blocks? Assume that in the GPU used, each SM can take up to 1,536 threads and up to 8 blocks.
  - For  $8 \times 8$ , we have 64 threads per block. Each SM can take up to 1536 threads, which is 24 blocks. But each SM can only take up to 8 Blocks, so only 512 threads (16 warps) go into each SM!
  - For  $16 \times 16$ , we have 256 threads per block. Each SM can take up to 1,536 threads (48 warps), which is 6 blocks (within the 8 block limit). Thus, we use the full thread capacity of an SM.
  - For  $32 \times 32$ , we have 1,024 threads per Block. Only one block can fit into an SM, using only 2/3 of the thread capacity of an SM.

A decorative graphic on the left side of the slide consisting of two vertical lines, one blue and one orange, extending from the top to the bottom of the slide.

**QUESTIONS?**

**READ CHAPTER 3!**

# Problem Solving

- Q: A particular CUDA device's streaming multiprocessor (SM) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configurations allows an SM to be fully utilized?
  - 256 threads per block, 384 threads per block, or 576 threads per block
- A:
  - $1536 / 256 = 6$  thread blocks – too many for SM
  - $1536 / 384 = 4$  thread blocks per SM – just the right number
  - $1536 / 576 = 3$  thread blocks per SM – not enough to fully utilize the SM

# Problem Solving

- Q: A 1D array of  $N$  floating point elements is to be processed in a one-element-per-thread fashion by a GPU. The target GPU has 8 SMs, each with 16 SPs. What is the best execution configuration for this kernel?
- A:
  - We do not know the max number of threads the SM can support.

# Problem Solving

- Consider the following CUDA kernel:

```
__global__ void do_work(int q, int *A) {  
    int result = 0;  
    if (q < 5) result = threadIdx.x;  
    A[threadIdx.x] = result;  
}
```

- Q: Is there is control divergence in this code?
- A: No since the value of q is the same for ALL threads in the thread block. For a thread to diverge, its execution path must depend on something unique to the thread, such as threadIdx.