# ECE408/CS483/CSE408 Fall 2022

# Applied Parallel Programming

# Lecture 3: Kernel-Based Data Parallel Execution Model

# Course Reminders

- Lab 0 was due yesterday
  - You should have submitted it by that deadline
  - But if you signed up for the course just recently, submit the lab soon
- Lab 1 is out, it is due this Friday
  - **Unlike Lab 0, this lab counts and must be submitted on time!**
- Email Andy Schuh (aschuh@illinois.edu) to get GitHub and rai if you just signed up for the course
- Check your grades in Canvas

# Objective

- To learn more about the multi-dimensional logical organization of CUDA threads
- To learn to use control structures, such as loops in a kernel
- To learn the concepts of thread scheduling, latency tolerance, and 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) blocks of 256 threads each
  dim3 DimGrid(ceil(n/256), 1, 1);
  dim3 DimBlock(256, 1, 1);

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

C D E

A B

**A** Number of blocks per dimension

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

**C** Unique block # in x dimension

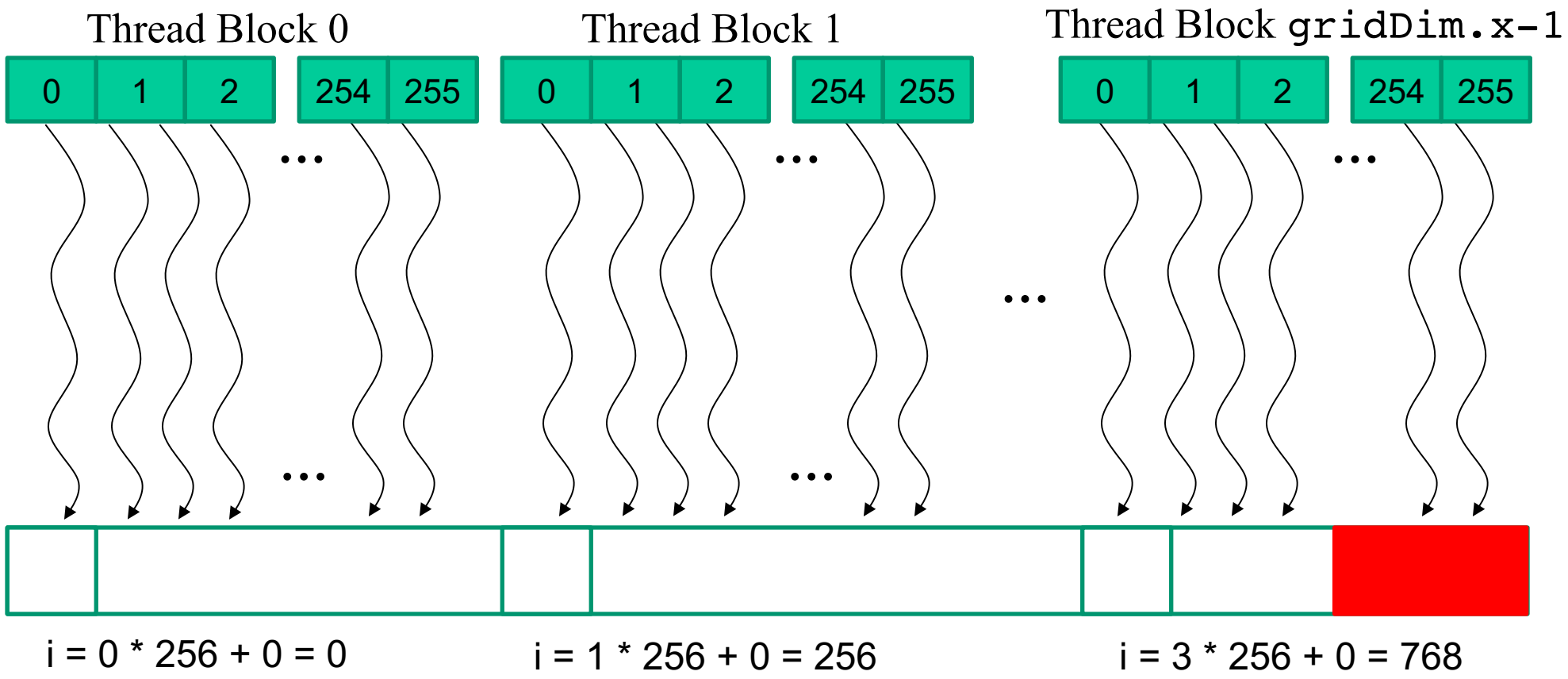
**D** Number of threads per block in x dimension

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

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

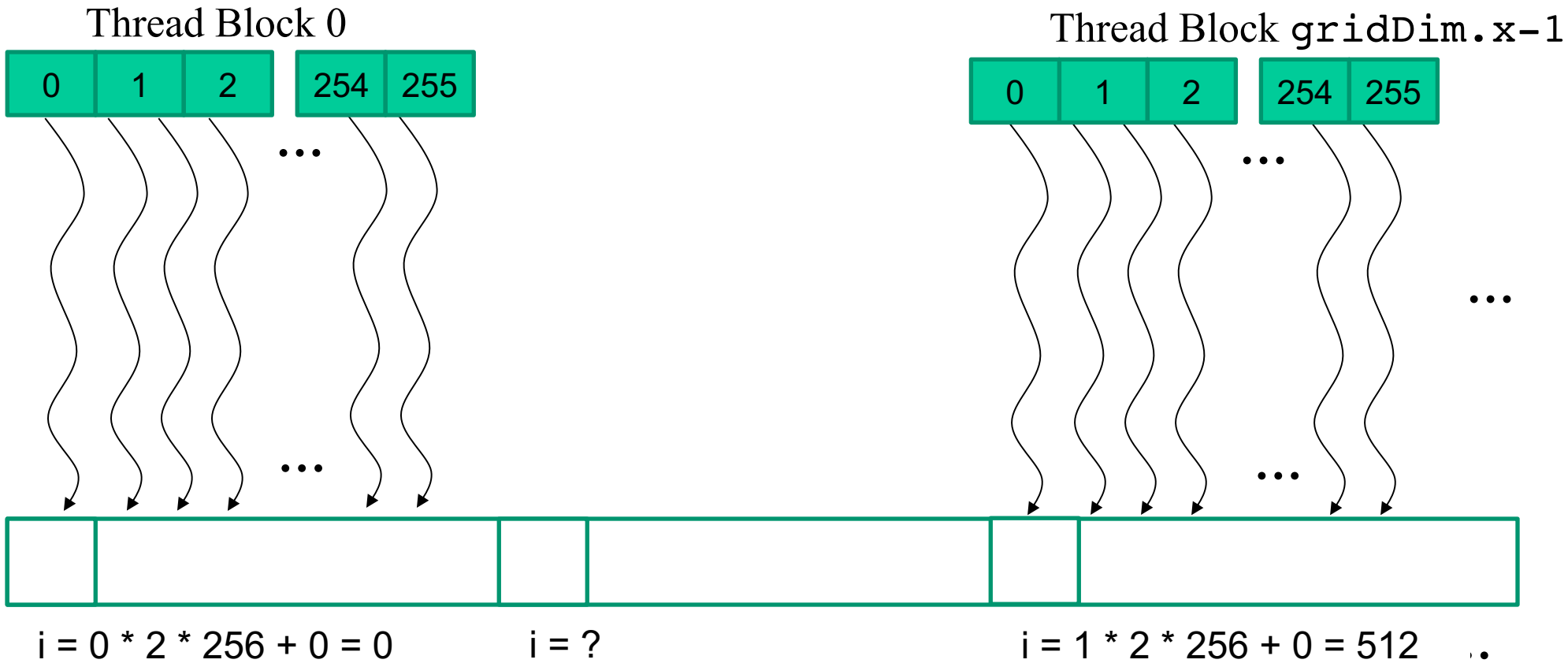
# Review – Thread Assignment for vecAdd
# n = 1000, block size = 256

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

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

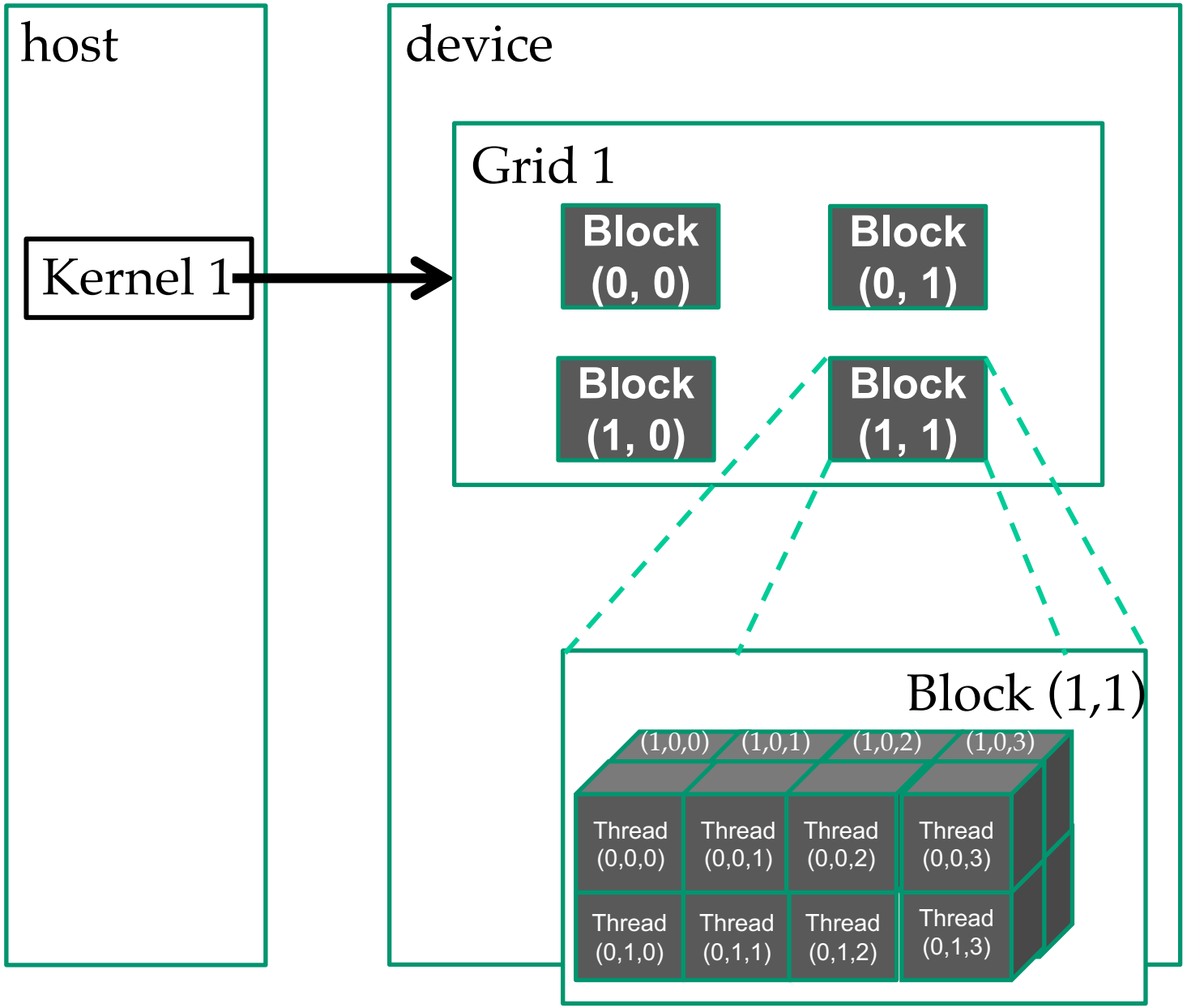


# Each thread processes 2 elements

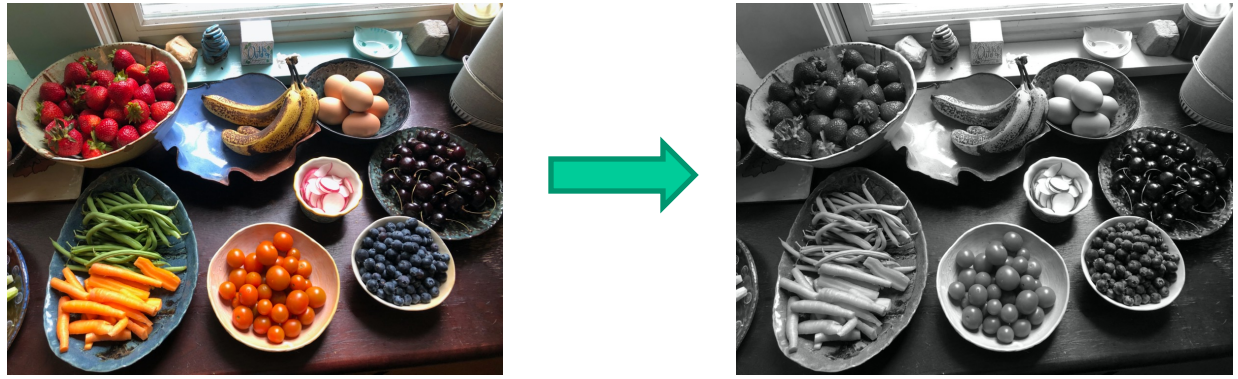
```
vecAdd<<<ceil(N/(2*256.0)), 256>>>(...)

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];
```

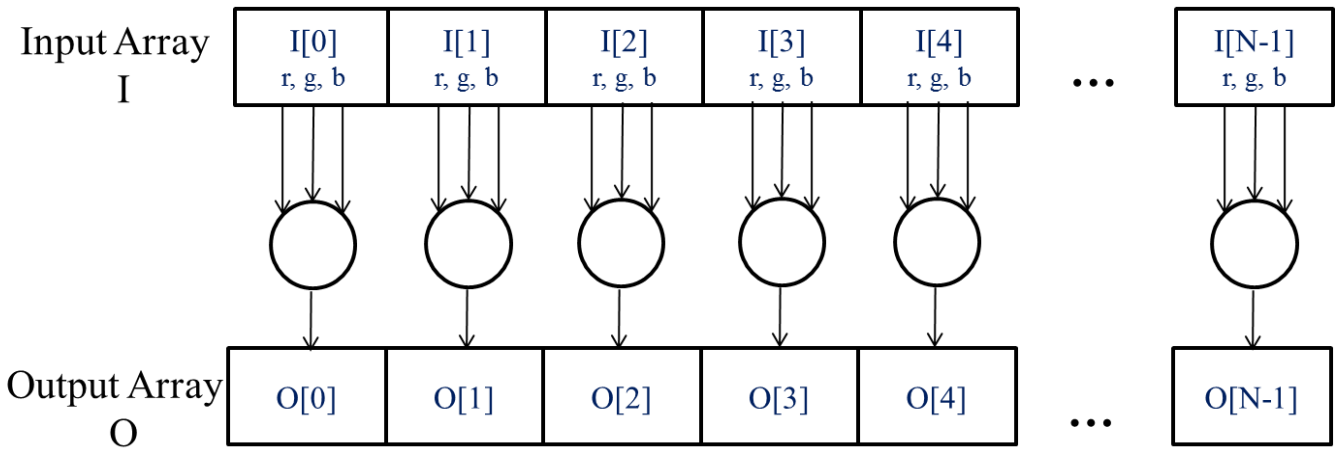


# CUDA Thread Grids are Multi-Dimensional



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

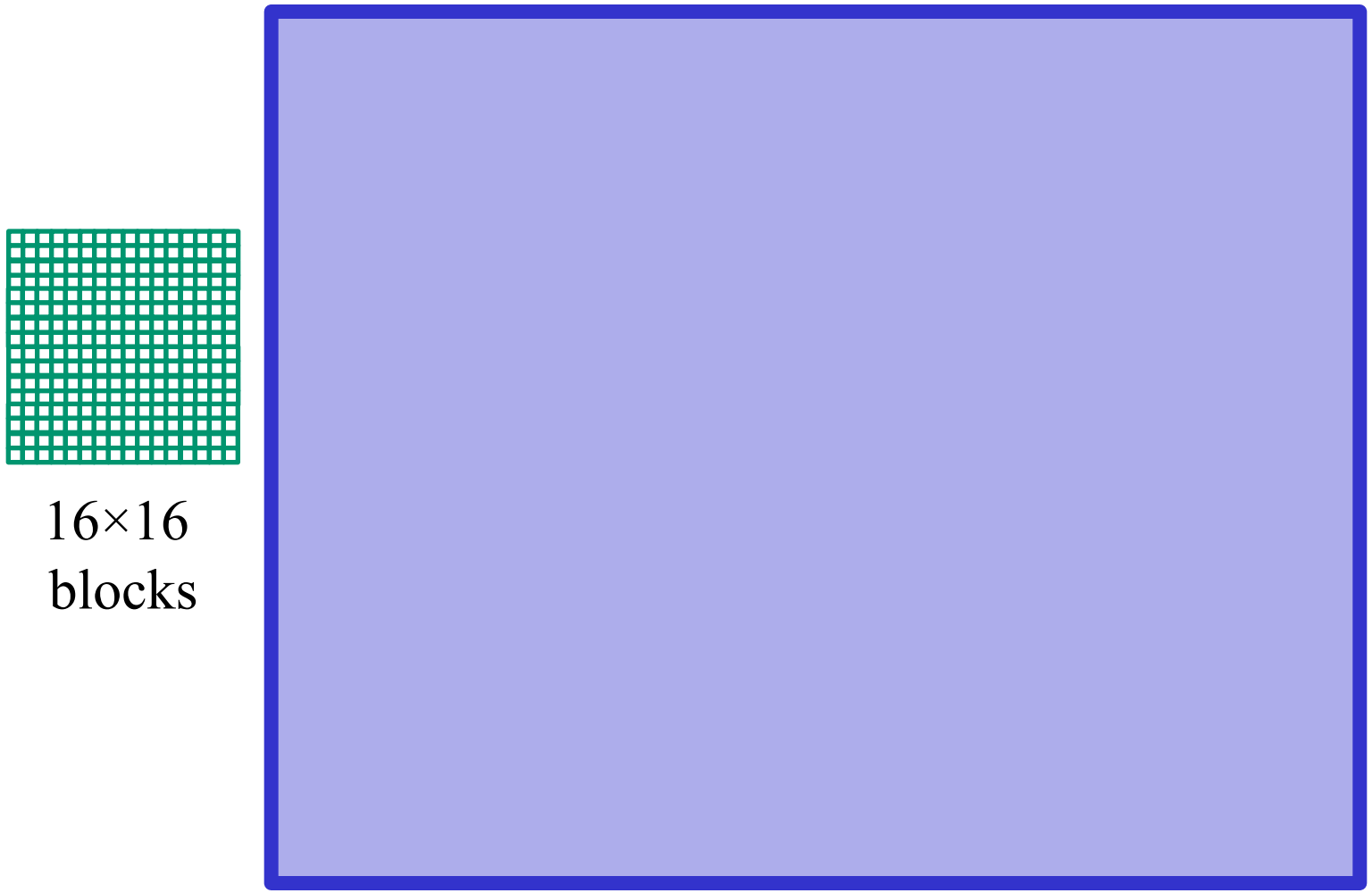





*Pixels can be calculated independently*

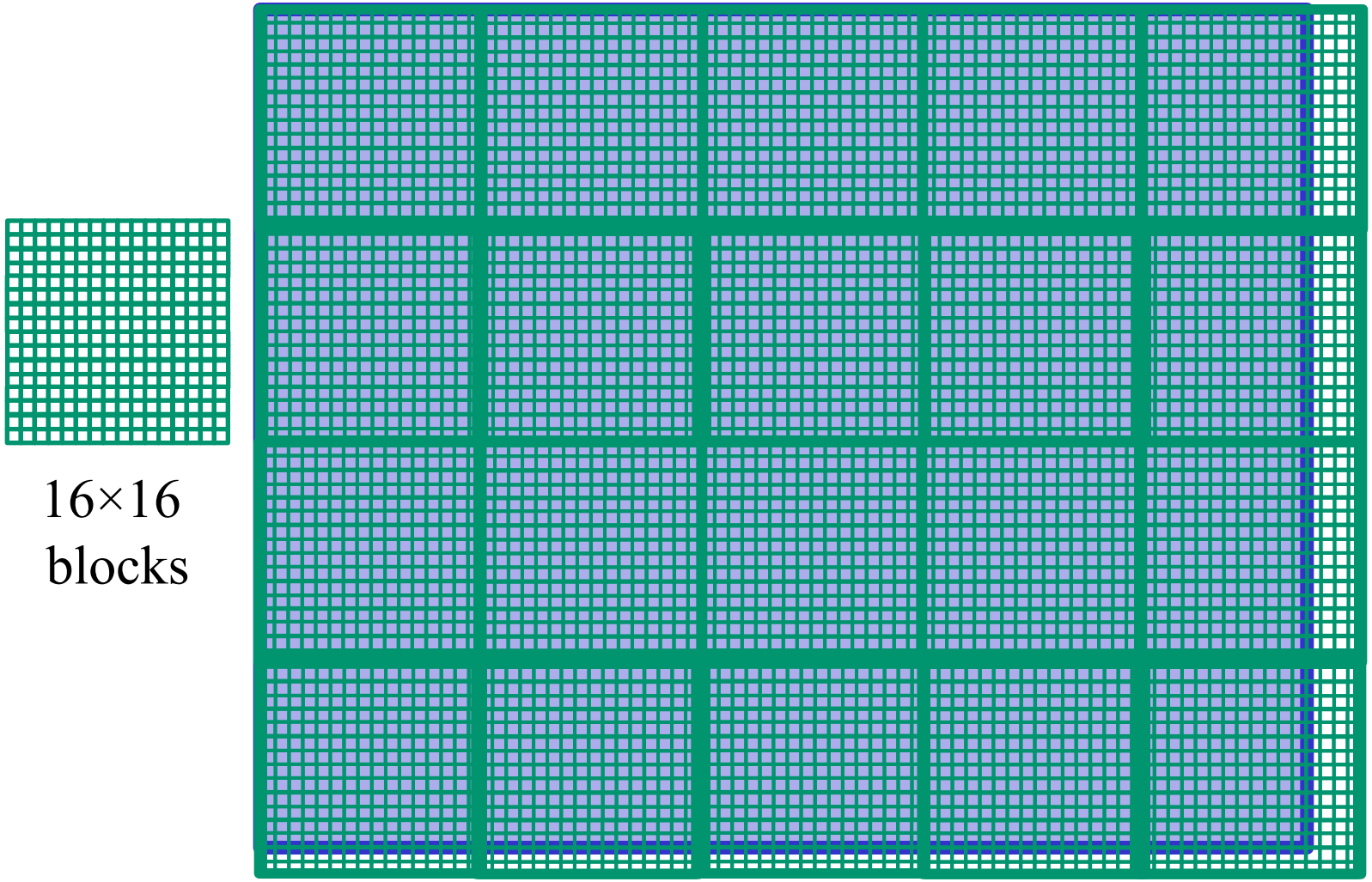
# Processing a Picture with a 2D Grid

# Processing a Picture with a 2D Grid



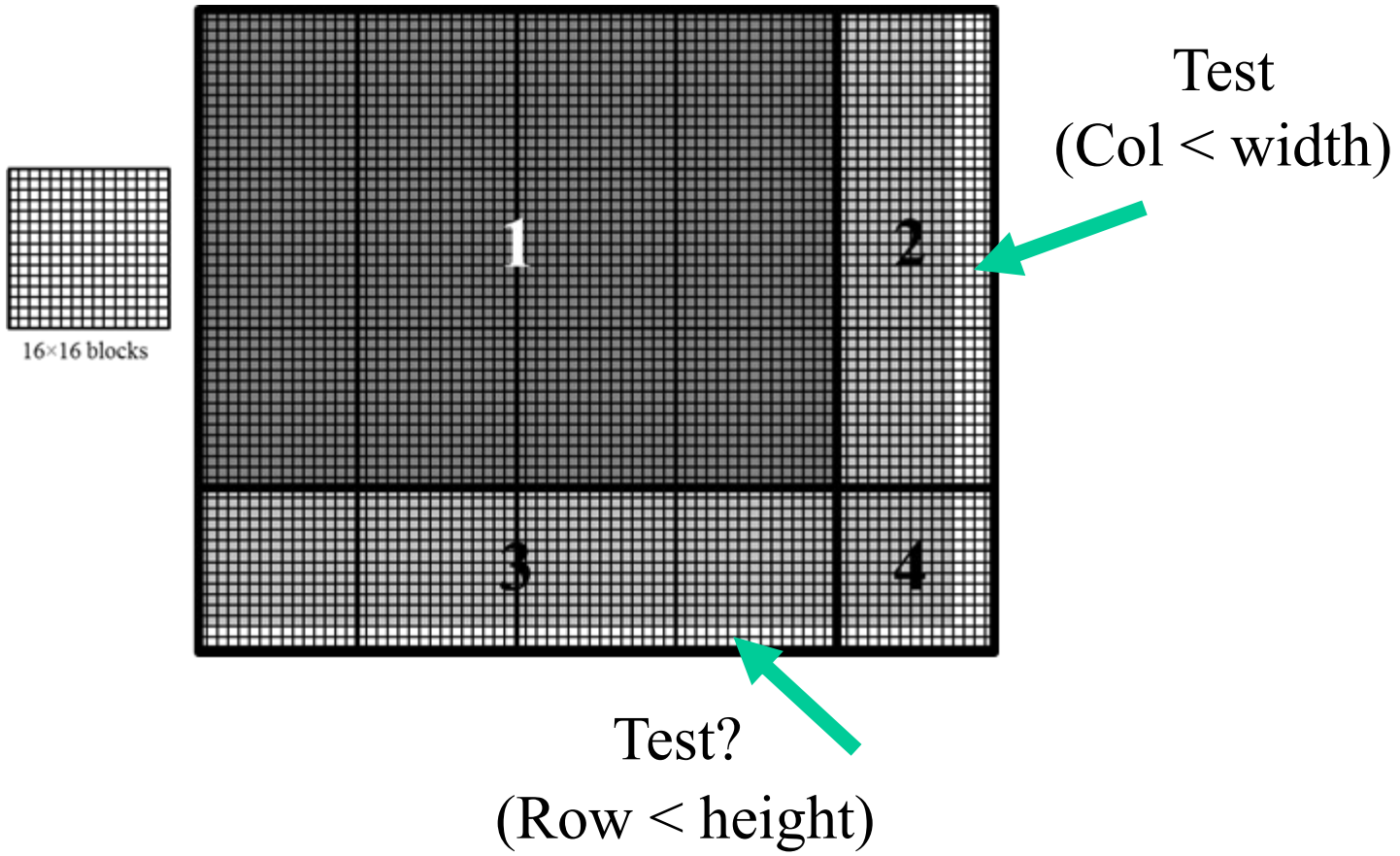
16×16
blocks

# Processing a Picture with a 2D Grid

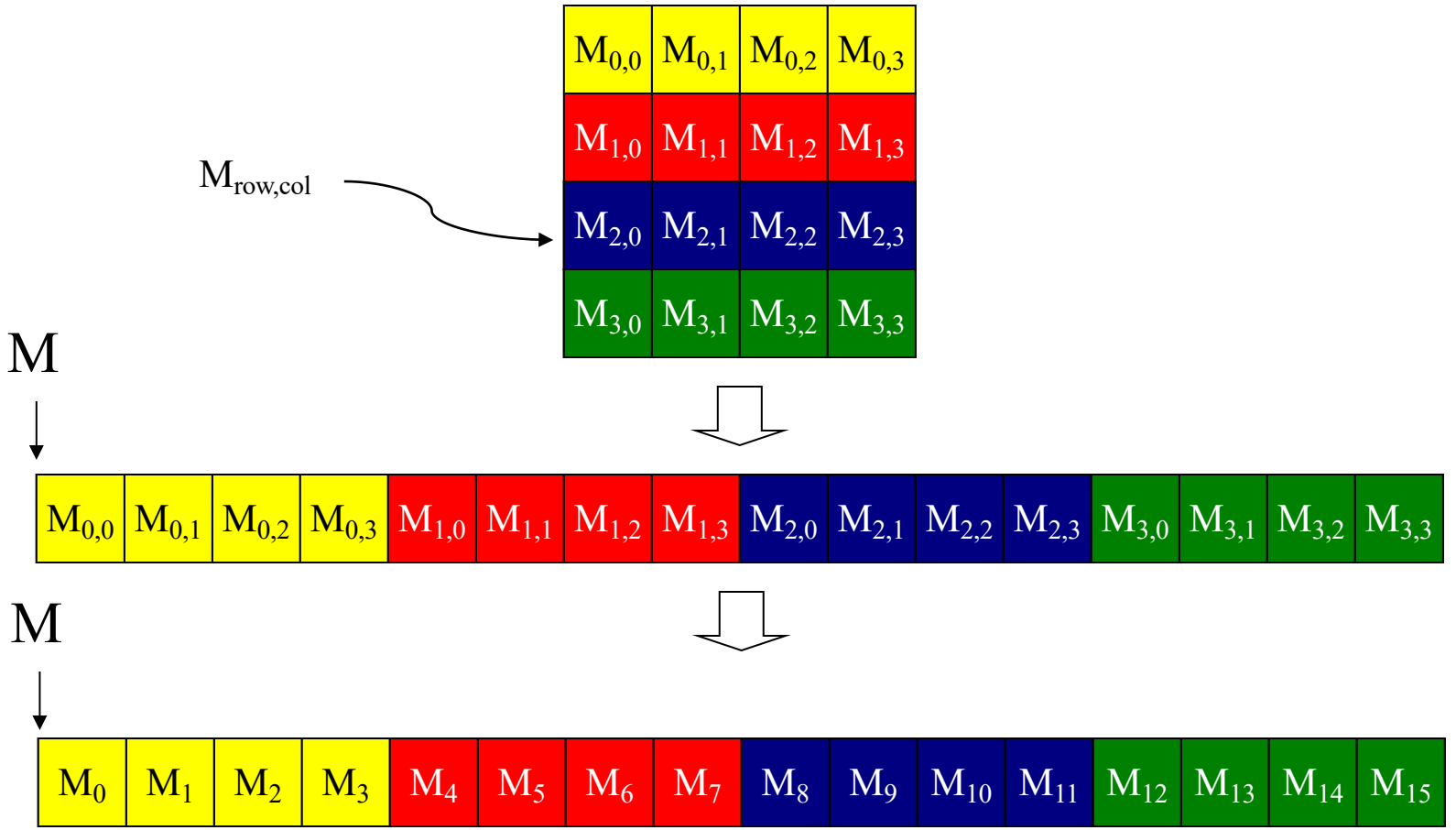


16×16
blocks

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



# Row-Major Layout of 2D Arrays in C/C++




$M_{2,1}$ → Row*Width+Col = 2*4+1 = 9

# RGB to Grayscale Kernel with 2D thread mapping

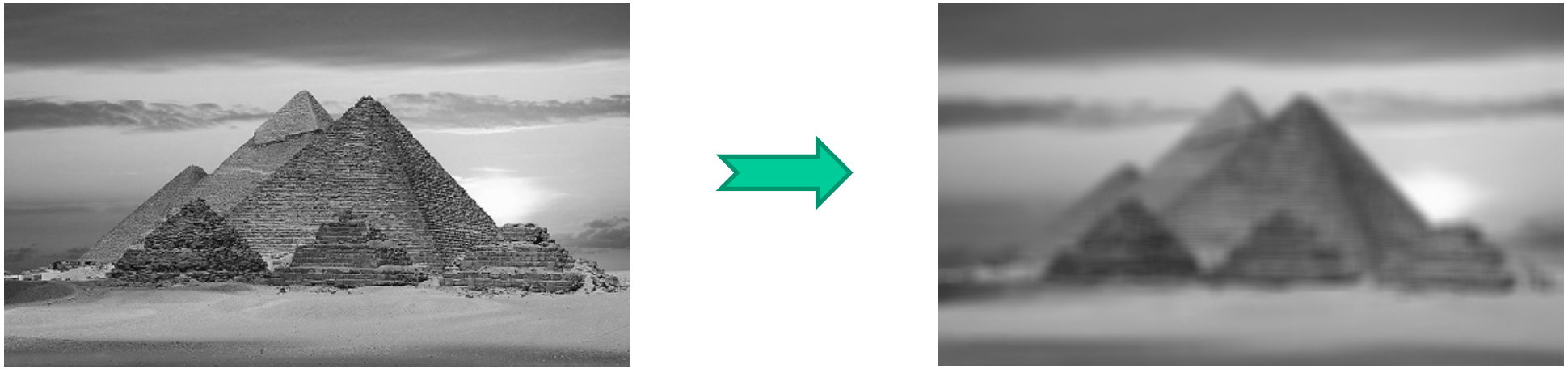
```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void RGBToGrayscale(unsigned char * grayImage,  unsigned char * rgbImage, int width, int height)
{

 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 grayOffset = Row*width + Col;
    // one can think of the RGB image having
    // CHANNEL times columns of the gray scale image
    int rgbOffset = grayOffset*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;
 }
}
```

# RGB to Grayscale Kernel with 2D thread mapping

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void RGBToGrayscale(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 grayOffset = Row*width + Col;
    // one can think of the RGB image having
    // CHANNEL times columns of the gray scale image
    int rgbOffset = grayOffset*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;
 }
}
```

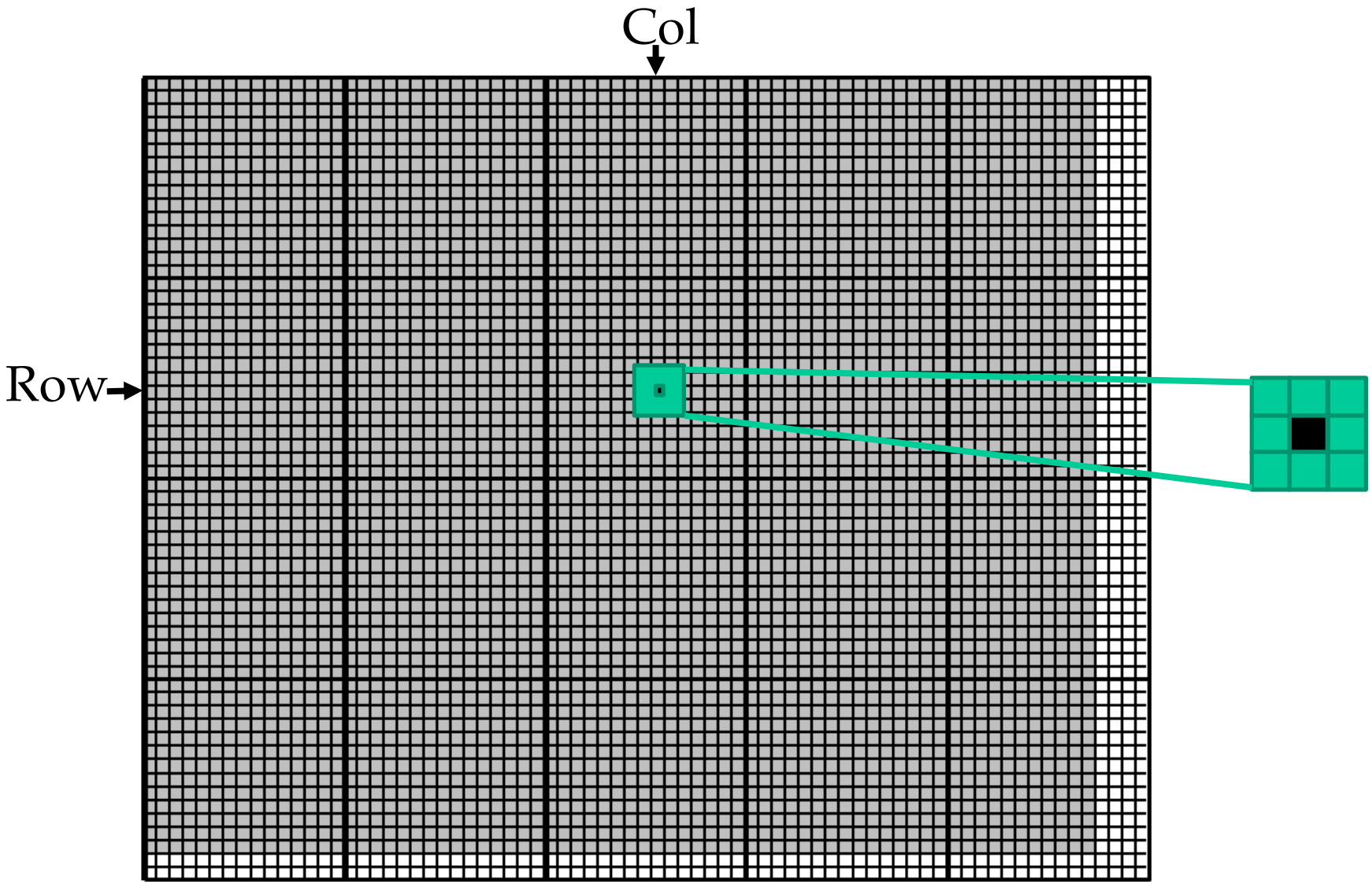
# Example 2: Image Blur

(BLUR_SIZE is 5)



# Each output pixel is the average of pixels around it

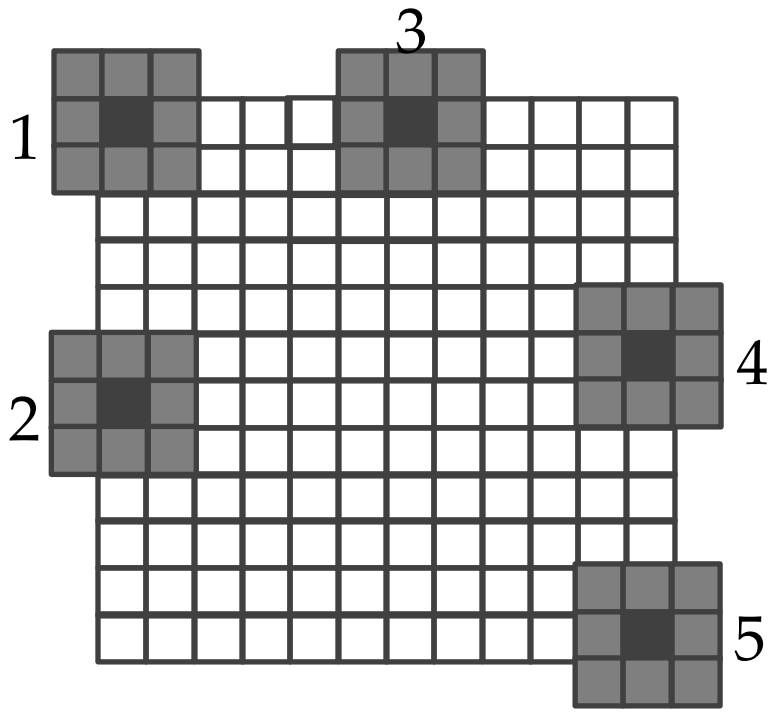
(BLUR_SIZE = 1)



# 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*



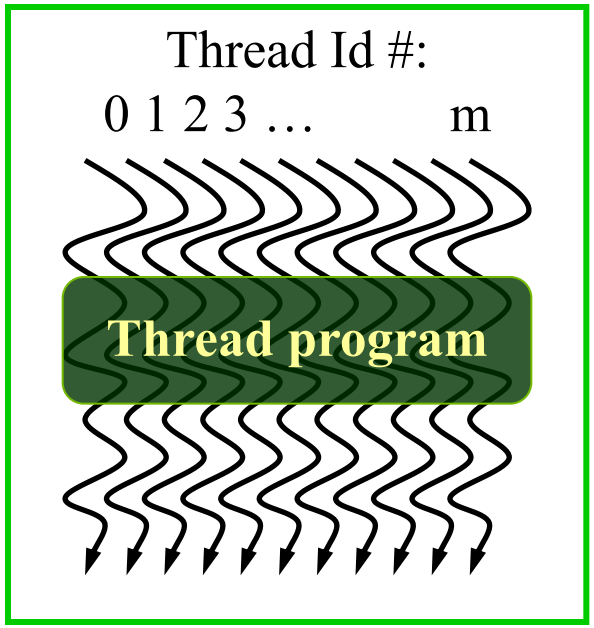
# An Image Blur Kernel

```
__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 Execution Model: Thread Blocks

- All threads in a block execute the same kernel program (SPMD)
- 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!
- Threads within the same block execute in warp order…

**CUDA Thread Block**




Courtesy: John Nickolls, NVIDIA

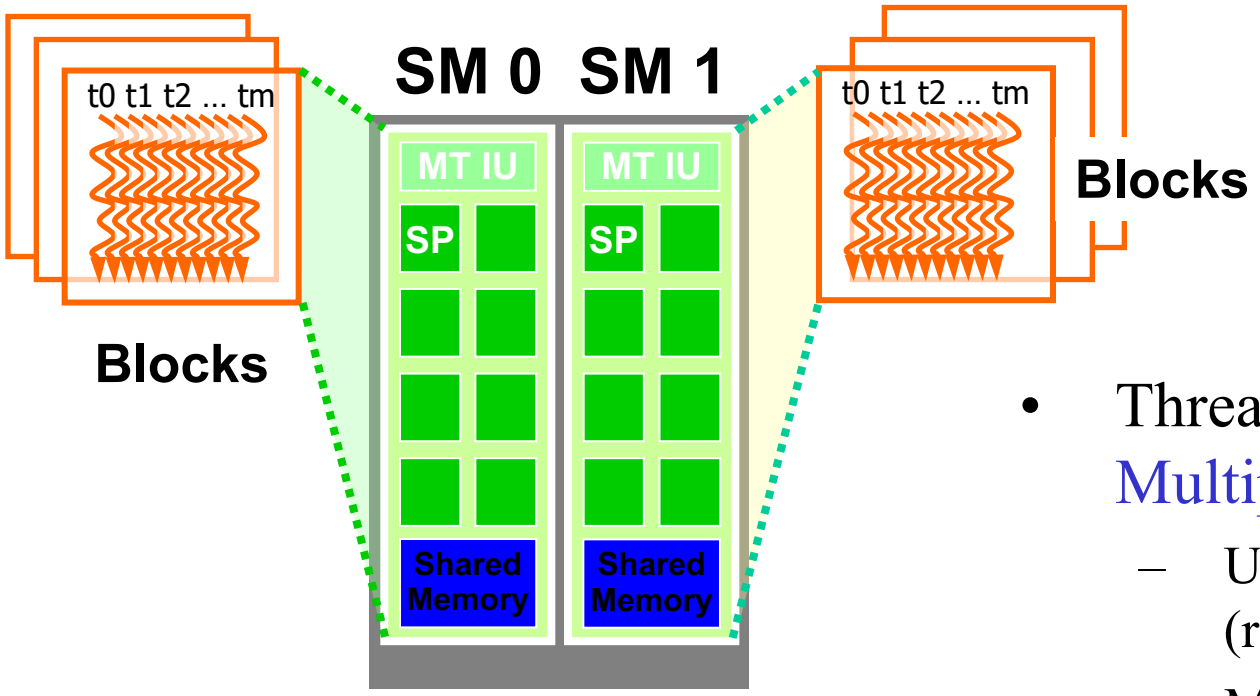
# Compute Capabilities are GPU-Dependent

**Table 15. Technical Specifications per Compute Capability**

| | Compute Capability | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Technical Specifications** | **3.5** | **3.7** | **5.0** | **5.2** | **5.3** | **6.0** | **6.1** | **6.2** | **7.0** | **7.2** | **7.5** | **8.0** | **8.6** |
| Maximum number of resident grids per device (Concurrent Kernel Execution) | 32 | | | | 16 | 128 | 32 | 16 | 128 | 16 | 128 | | |
| Maximum dimensionality of grid of thread blocks | 3 | | | | | | | | | | | | |
| Maximum x-dimension of a grid of thread blocks | $2^{31}-1$ | | | | | | | | | | | | |
| Maximum y- or z-dimension of a grid of thread blocks | 65535 | | | | | | | | | | | | |
| Maximum dimensionality of a thread block | 3 | | | | | | | | | | | | |
| Maximum x- or y-dimension of a block | 1024 | | | | | | | | | | | | |
| Maximum z-dimension of a block | 64 | | | | | | | | | | | | |
| Maximum number of threads per block | 1024 | | | | | | | | | | | | |
| Warp size | 32 | | | | | | | | | | | | |
| Maximum number of resident blocks per SM | 16 | | 32 | | | | | | | | 16 | 32 | 16 |
| Maximum number of resident warps per SM | 64 | | | | | | | | | | 32 | 64 | 48 |
| Maximum number of resident threads per SM | 2048 | | | | | | | | | | 1024 | 2048 | 1536 |
| Number of 32-bit registers per SM | 64 K | 128 K | 64 K | | | | | | | | | | |
| Maximum number of 32-bit registers per thread block | 64 K | | | | 32 K | 64 K | | 32 K | 64 K | | | | |
| Maximum number of 32-bit registers per thread | 255 | | | | | | | | | | | | |
| Maximum amount of shared memory per SM | 48 KB | 112 KB | 64 KB | 96 KB | 64 KB | | 96 KB | 64 KB | 96 KB | | 64 KB | 164 KB | 100 KB |
| Maximum amount of shared memory per thread block [33] | 48 KB | | | | | | | | 96 KB | 96 KB | 64 KB | 163 KB | 99 KB |
| Number of shared memory banks | 32 | | | | | | | | | | | | |
| Maximum amount of local memory per thread | 512 KB | | | | | | | | | | | | |

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

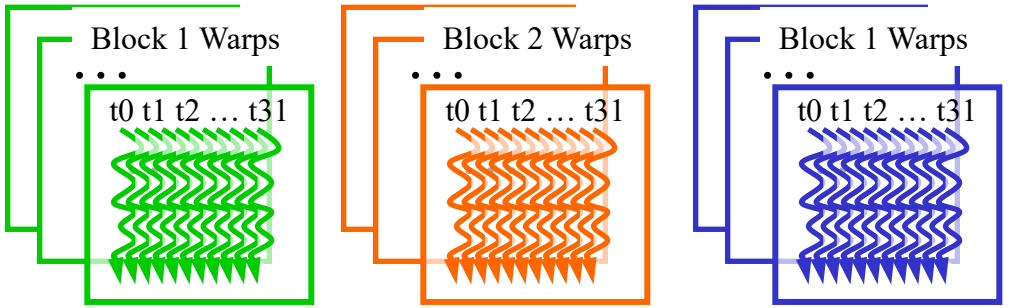
# Executing Thread Blocks

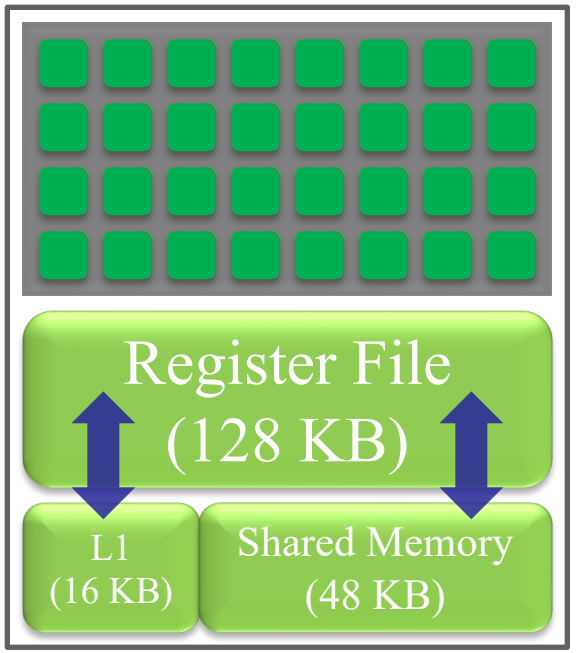



- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to **32** blocks to each SM (resource limit for Maxwell)
  - Maxwell/Pascal/Turing SM can take up to **2048** threads
- Threads run concurrently, as warps
  - SM maintains thread/block id #s
  - SM manages/schedules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

# Thread Scheduling (1/2)

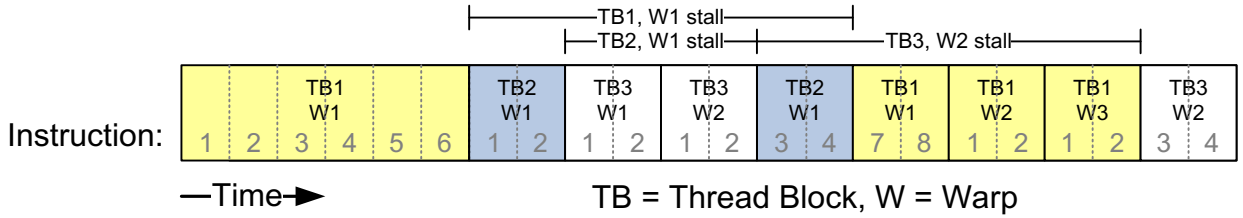
- Each block is executed as 32-thread warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are divided based on their linearized thread index
    - Threads 0-31: warp 0
    - Threads 32-63: warp 1, etc.
    - X dim, Y dim, then Z dim
  - Warps are scheduling units in 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 warps/blk * 3 blks = 24 warps



# 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 `RGBToGrayscaleConversion`, should we use 8×8, 16×16 or 32×32 blocks? Assume the GPU can have 1,536 threads and up to 8 blocks per SM.
  - For 8×8, we have 64 threads per block. Each SM can take up to 1,536 threads, which is 1,536/64=24 blocks. But each SM can only take up to 8 Blocks, so only 512 threads (16 warps) go into each SM!
  - For 16×16, 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 32×32, 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.

# ANY MORE QUESTIONS?
# READ CHAPTER 3