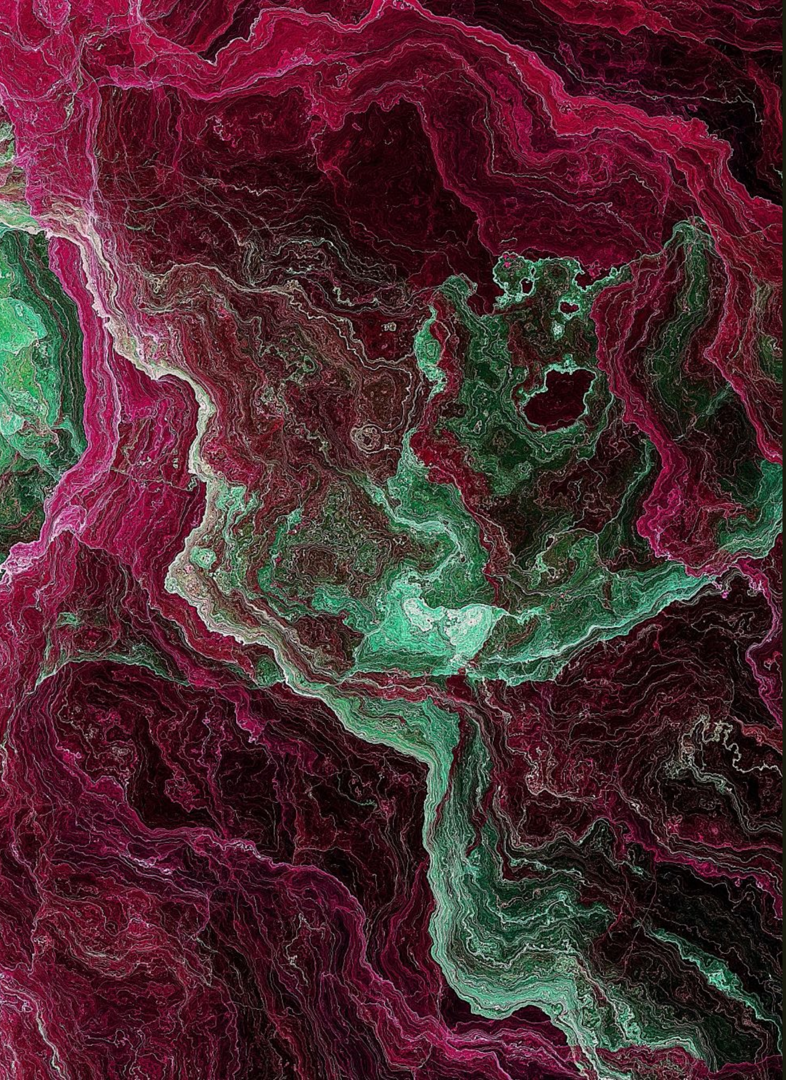


GPUS: DATA-PARALLEL EXECUTION MODEL

Dr. Steve Petruzza

UtahStateUniversity

Objective

- To learn more on the logical multi-dimensional organization of CUDA threads
- To learn to use control structures like loops in a kernel
- To learn the concept of thread scheduling, latency tolerance, and hardware occupancy

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

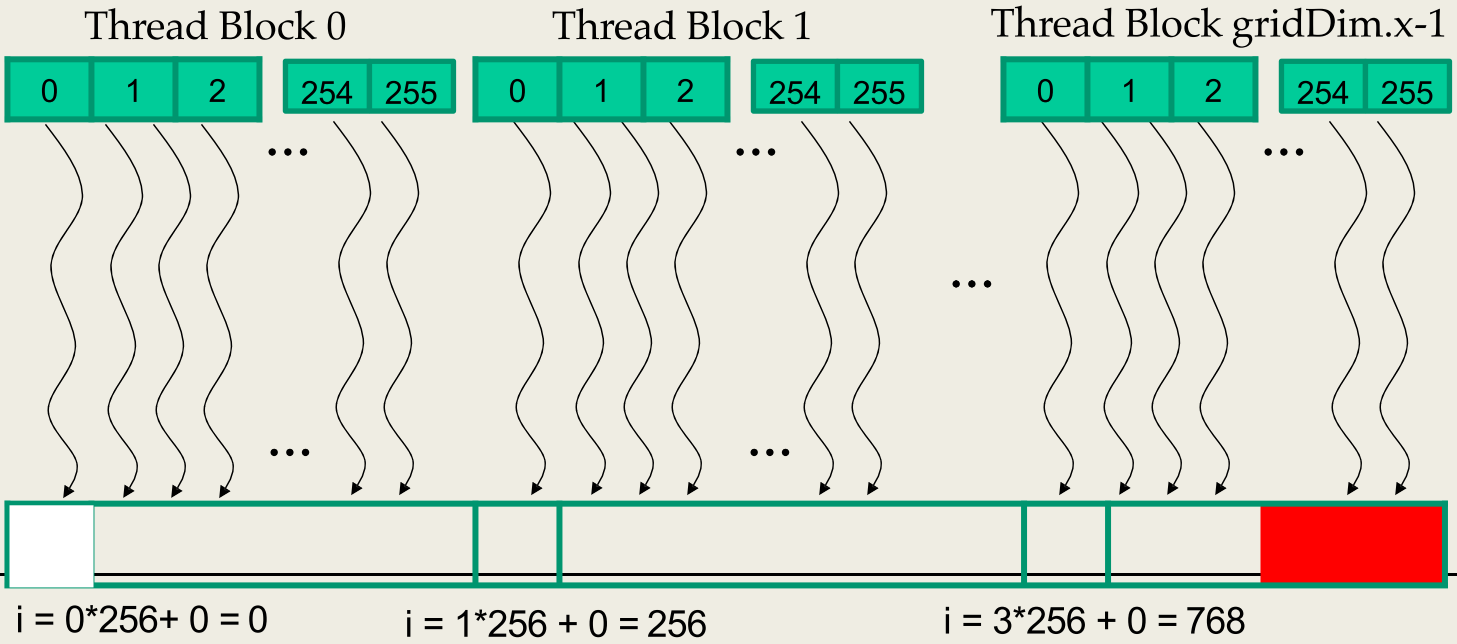
C Unique block # in x dimension

D Number of threads per block in x dimension

E Unique thread # in x dimension in the block

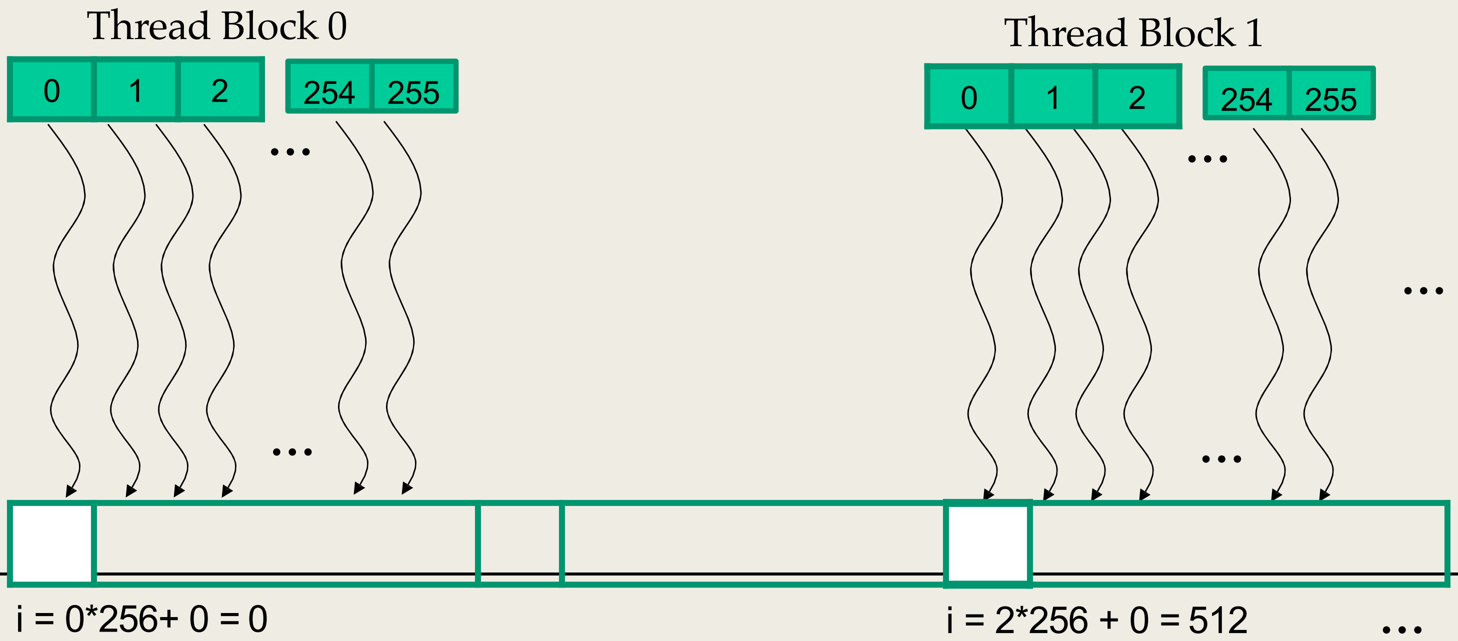
Review – Thread Assignment for vecAdd N = 1000, block size = 256

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

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

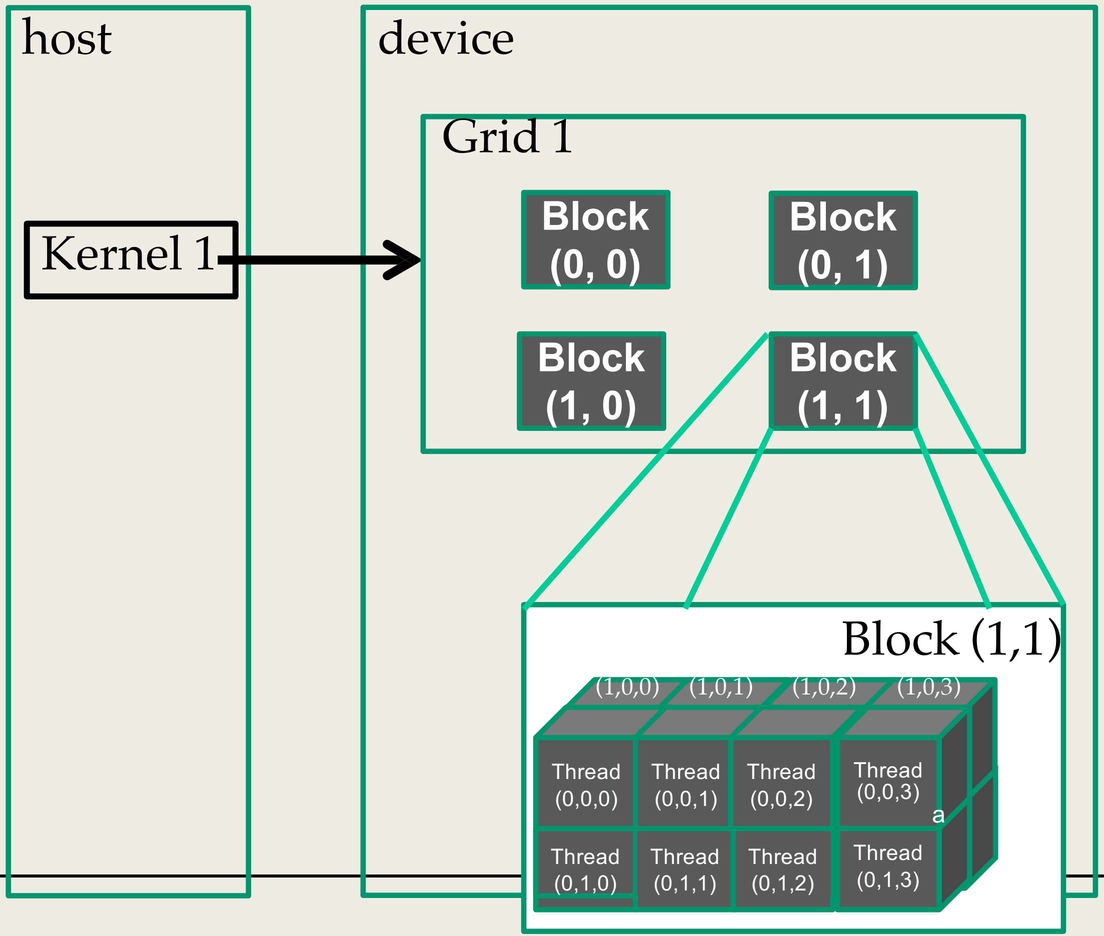


Each thread processes 2 elements

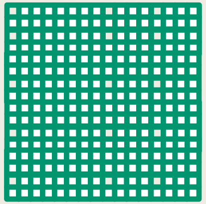
```
vecAdd<<<ceil(N/(2*256.0)), 256>>>(...)

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];
```



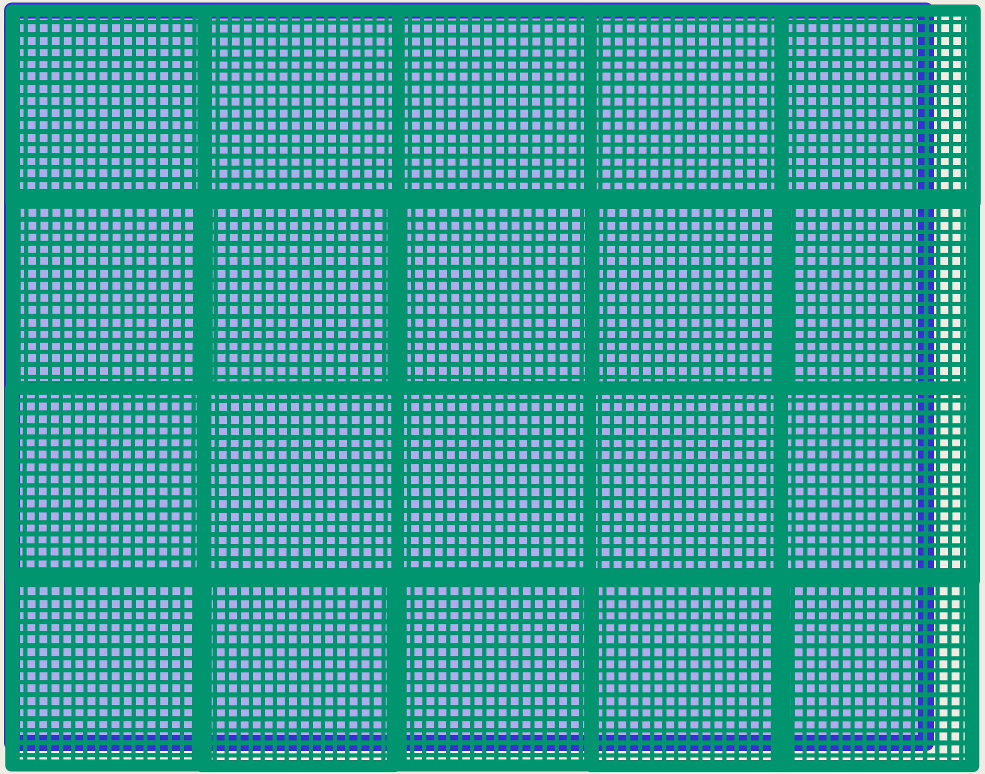
CUDA Thread Grids are Multi-Dimensional



Processing a Picture with a 2D Grid

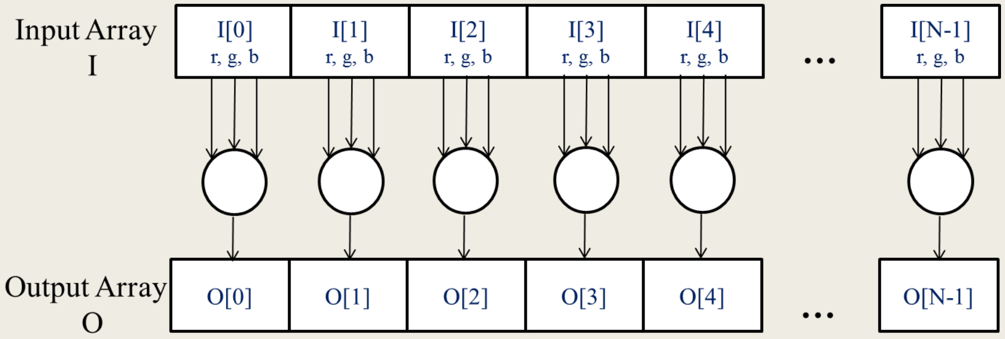


16×16 blocks

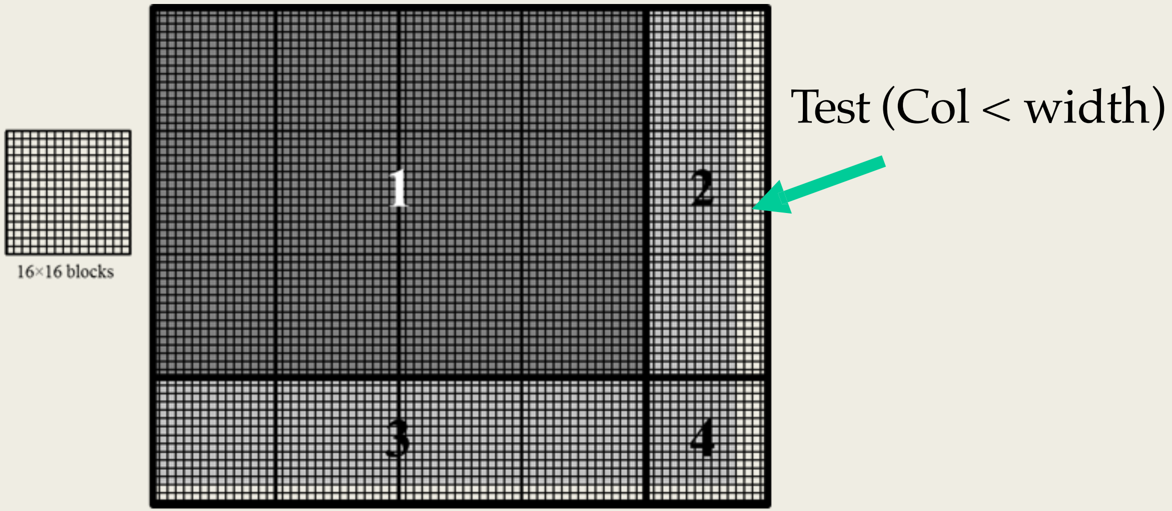


Conversion of a color image to gray–scale image (review)

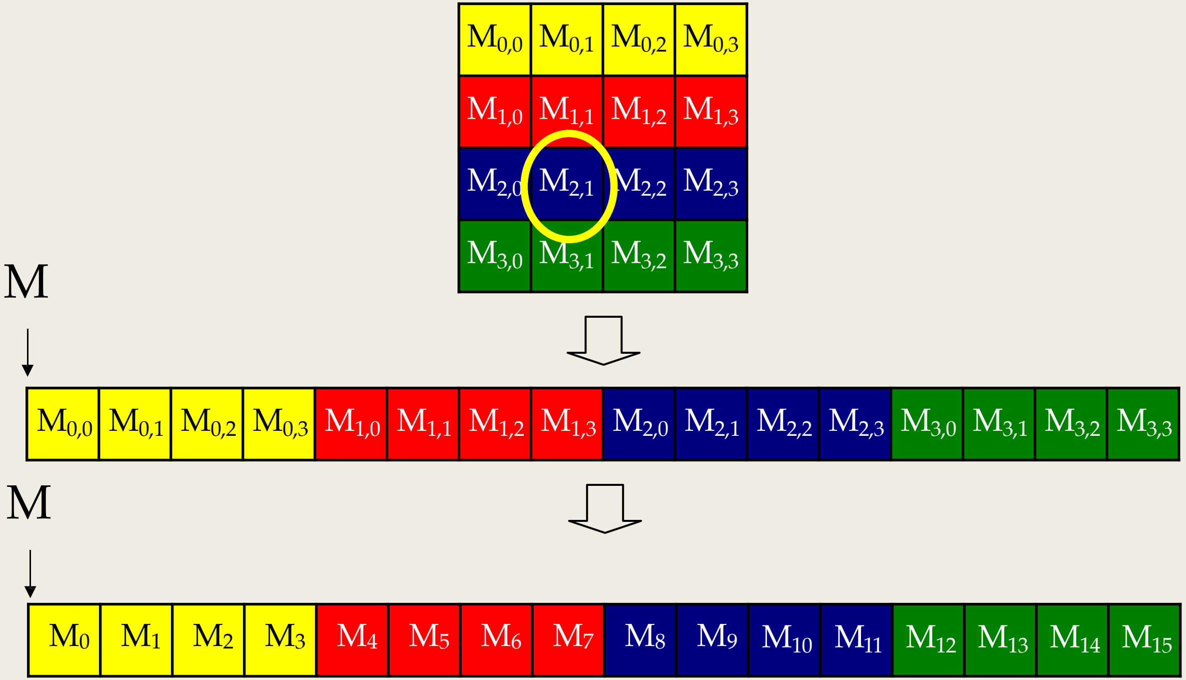
The pixels can be calculated independently of each other



Covering a 76×62 picture with 16×16 blocks



Row-Major Layout of 2Darrays in C/C++



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

RGB to Grayscale Kernel with 2D thread mapping

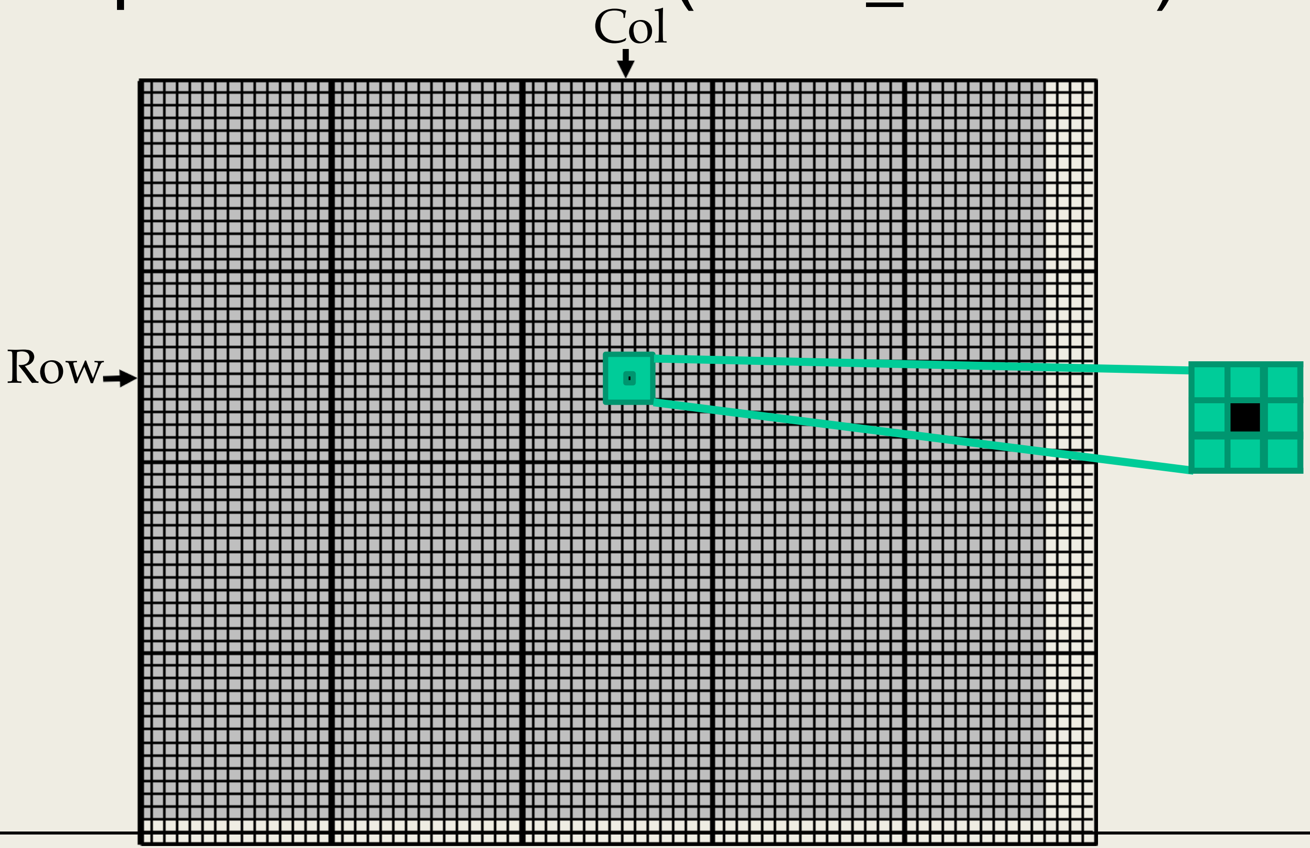
```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void RGBToGrayscale(unsigned char * grayImage,  unsigned char * rgbImage, int width, int height)
{

 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 grayscale 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;
 }
}
```

Image Blurring

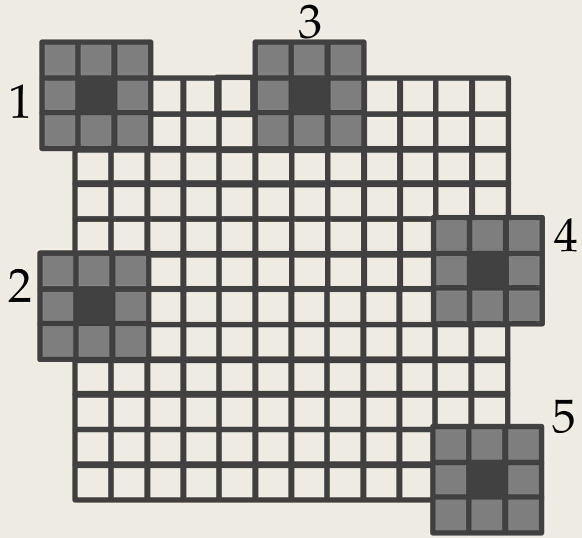
Each output pixel is the average of pixels around it (BLUR_SIZE = 1)



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+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);
     }
   }
```

Handling boundary conditions for pixels near the edges of the image



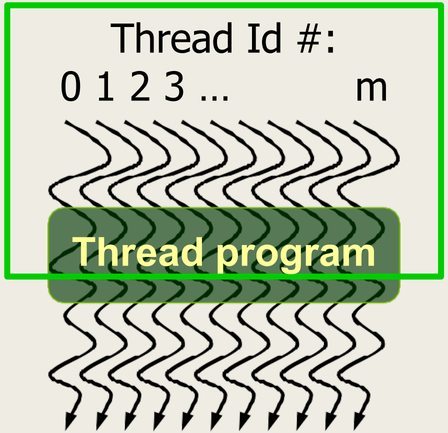
An Image Blur Kernel

```
__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);
   }
 }
```

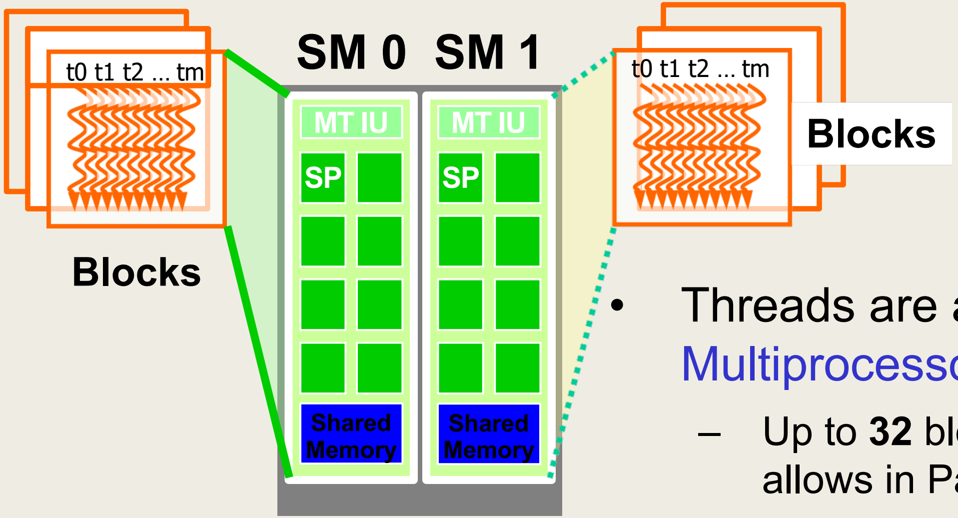
CUDA Thread Block (review)

- All threads in a block execute the same kernel
- Programmer declares block geometry
  - Block size from 1 to 2048 threads
  - Block shape 1D, 2D, or 3D
- Threads / Blocks have index numbers
  - Kernel code uses these index numbers to select work and calculate addresses
- Threads within a block can cooperate using atomic operation and shared memory
- Threads in different blocks can cooperate too, but much more expensive, and therefore less common.

CUDA Thread Block



Executing Thread Blocks

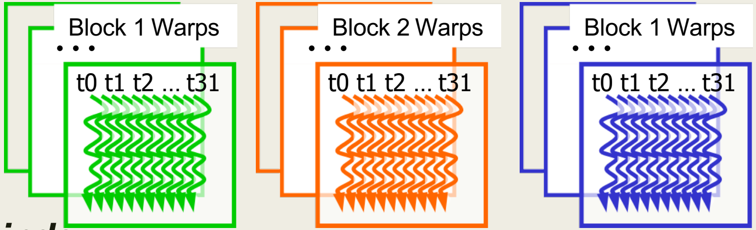


- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **32** blocks to each SM as resource allows in Pascal (& Turing)
  - Pascal/Volta/Turing SM can take up to **2048** threads
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread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

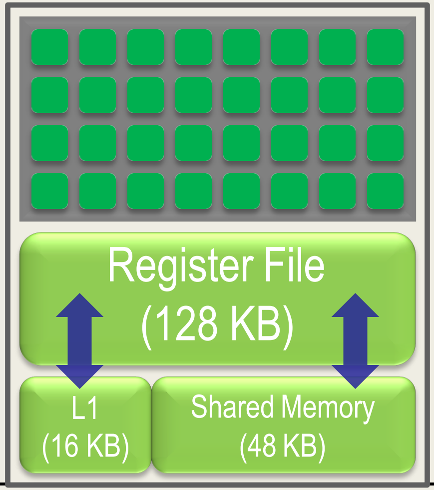
Compute Capabilities are GPUDependent

Feature	Kepler GK210	Maxwell GM200	Maxwell GM204	Pascal GP100	Pascal GP102
Compute Capability	3.7	5.2		6.0	6.1
Threads per Warp	32				
Max Warps per SM	64				
Max Threads per SM	2048				
Max Thread Blocks per SM	16	32			
Max Concurrent Kernels	32			128	32
32-bit Registers per SM	128 K	64 K			
Max Registers per Thread Block	64 K				
Max Registers per Thread	255				
Max Threads per Thread Block	1024				
L1 Cache Configuration	split with shared memory			24KB dedicated L1 cache	
Shared Memory Configurations	16KB + 112KB L1 Cache 32KB + 96KB L1 Cache 48KB + 80KB L1 Cache *(128KB total)*	96KB dedicated		64KB dedicated	96KB dedicated
Max Shared Memory per Thread Block	48KB				

Thread Scheduling (1/2)

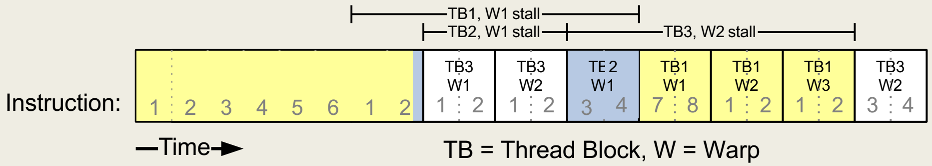


- Each block is executed as 32-thread warps
  - *An implementation decision, not part of the CUDA programming model*
  - *Warps are divided based on their* ***linearized thread index***
    - Threads 0-31: warp 0
    - Threads 32-63: warp 1, etc.
    - X-dimension first, then Y, then Z
  - Warps are 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

Control (branch) Divergence

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in current GPUs
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: divergence could occur when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x % 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; even threads follow different path than odd threads in warp
  - Example without divergence:
    - `If ((threadIdx.x / WARP_SIZE) % 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Block Granularity Considerations

- For RGBToGrayscale, should one use 8X8, 16X16 or 32X32 blocks? Assume that in the GPU used, each SM can take up to 1536 threads and up to 8 blocks.
  - For 8X8, 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, only 512 threads (16 warps) will go into each SM!
  - For 16X16, we have 256 threads per block. Since each SM can take up to 1536 threads (48 warps), which is 6 blocks (within the 8 block limit). Thus we use the full thread capacity of an SM.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM, using only 2/3 of the thread capacity of an SM.