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 – Thread Assignment for vecAdd
where N = 1,000, block size = 256

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

\[ i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}; \]

if \( i < N \)
\[ C[i] = A[i] + B[i]; \]

```
…
```

<table>
<thead>
<tr>
<th>Thread Block 0</th>
<th>Thread Block 1</th>
<th>Thread Block gridDim.x-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

```
i = 0*256 + 0 = 0
i = 1*256 + 0 = 256
i = 3*256 + 0 = 768
```

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

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

\[ i = \text{blockIdx.x} \times (2 \times \text{blockDim.x}) + \text{threadIdx.x}; \]

if \( i < N \)
\[ C[i] = A[i] + B[i]; \]

```
…
```

<table>
<thead>
<tr>
<th>Thread Block 0</th>
<th>Thread Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

```
i = 0*256+0 = 0
i = 1*256+0 = 256
i = 2*256 + 0 = 512
```

```
…
```
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];
```

Review: Conversion of a color image to a grey–scale image
```
© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2018
ECE408/CS483/ECE498al, University of Illinois, Urbana-Champaign
```

Review: Pixels can be calculated independently
```
© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2018
ECE408/CS483/ECE498al, University of Illinois, Urbana-Champaign
```

Processing a Picture with a 2D Grid
```
© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2018
ECE408/CS483/ECE498al, University of Illinois, Urbana-Champaign
```
Covering a 76×62 picture with 16×16 blocks

Test (Col < width)

Test? (Row < height)

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

Row-major layout

M

M00 M01 M02 M03
M10 M11 M12 M13
M20 M21 M22 M23
M30 M31 M32 M33

M2,1 \rightarrow \text{Row} \times \text{Width} + \text{Col} = 2 \times 4 + 1 = 9

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];
unsigned char g = rgbImage[rgbOffset + 1];
unsigned char b = rgbImage[rgbOffset + 2];

// perform the rescaling and store it
// We multiply by floating point constants
grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
}
}
A blurred pixel is the average of the surrounding pixels.

```
__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) {
        int pixVal = 0;
        int pixels = 0;
        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
        for (int blurRow = -BLUR_SIZE; blurRow <= BLUR_SIZE; ++blurRow) {
            for (int blurCol = -BLUR_SIZE; blurCol <= BLUR_SIZE; ++blurCol) {
                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
                if (curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++;
                }
            }
        }
        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```
__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) {
        int pixVal = 0;
        int pixels = 0;
        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++;
                }
            }
        }
        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
Covering a 76×62 picture with 16×16 blocks

Test (Col < width)

Test (Row < height)

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 warps/block * 3 blocks = 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

© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2018
ECE408/CS463/ECE498al, University of Illinois, Urbana-Champaign
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 8x8, 16x16 or 32x32 blocks? Assume that in the GPU used, each SM can take up to 1,536 threads and up to 8 blocks.
  - For 8x8, we have 64 threads per block. Each SM can take up to 1,536 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 16x16, 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 32x32, 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.