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
Review – Thread Assignment for vecAdd
Each thread processes 2 elements (Q4)

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

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

CUDA Thread Grids are Multi-Dimensional

Processing a Picture with a 2D Grid

Conversion of a color image to grey-scale image (review)
The pixels can be calculated independently of each other (review)

```
// 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
        // CHANNEL times columns of the grayscale image
        int rgbOffset = greyOffset*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;
    }
}
```

Covering a 76×62 picture with 16×16 blocks

```
// Row-Major Layout of 2D arrays in C/C++
```

```
M

M_0
M_1
M_2
M_3

M_4
M_5
M_6
M_7

M_8
M_9
M_10
M_11

M_12
M_13
M_14
M_15

M_2,1 \rightarrow \text{Row}^*\text{Width} + \text{Col} = 2*4+1 = 9
```
colorToGreyscaleConversion Kernel with 2D thread mapping to data (cont.)

```c
__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) {
    int greyOffset = Row*width + Col;
    int rgbOffset = greyOffset*CHANNELS;
    unsigned char r = rgbImage[rgbOffset];
    unsigned char g = rgbImage[rgbOffset + 1];
    unsigned char b = rgbImage[rgbOffset + 2];
    grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
  }
}
```

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

```c
__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;
    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;
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol];
          pixels++;
        }
      }
    }
    out[Row * w + Col] = (unsigned char)(pixVal / pixels);
  }
}
```

Image Blurring

An Image Blur Kernel

```c
__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;
    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;
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol];
          pixels++;
        }
      }
    }
    out[Row * w + Col] = (unsigned char)(pixVal / pixels);
  }
}
```
Handling boundary conditions for pixels near the edges of the image

An Image Blur Kernel

```c
__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);
    }
}
```

CUDA Thread Block (review)

- 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 have thread index numbers within block
  - 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
  - Each block can execute in any order relative to other blocks!

CUDA Thread Block

Compute Capabilities are GPU Dependent
Covering a 76×62 picture with 16×16 blocks

Test (Col < width)

Executing Thread Blocks

- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to 32 blocks to each SM as resource allows in 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
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; threads 0, 1 and 2 follow different path than the rest of the threads in warp 0
  - 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 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 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 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 would 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.