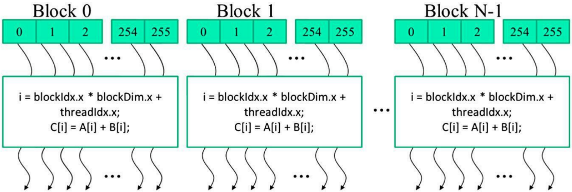
CSc 8530
Parallel Algorithms

Spring 2019

March 7th, 2019

Grid organization



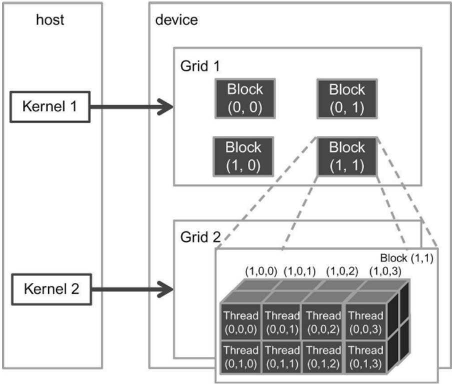
- **blockIdx**, **blockDim**, and **threadIdx** are *global, built-in* variables that allow you to index any thread
  - CUDA initializes the necessary values when you call a kernel function
  - Global variables are accessible to all functions in a program
  - You should never modify built-in variables yourself

Final vector addition code

```
void vecAdd(float* A, float* B, float* C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc((void **) &d_A, size);
  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMalloc((void **) &d_B, size);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **) &d_C, size);

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        // Free device memory for A, B, C
  cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

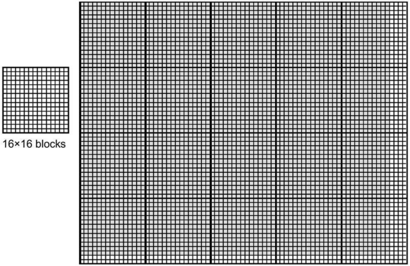
- The number of threads-per-block is fixed
- The number of blocks is a function of the vector length n
- Blocks can be executed *in any order*
  - The GPU's specs define how many blocks can run in parallel

Grid and block organization



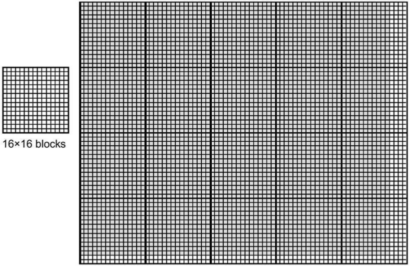
- Grid and blocks need not have the same dimensionality
- Above, we have 2D grids with 3D blocks
- The code is `dim3 gridDim(2,2,1)` and `blockDim(4,2,2)`

Mapping grids to data



- The choice of dimensionality is driven by the data
- e.g., images are naturally 2D
- Above, we are using 5×4, 16×16 blocks for 76×62 pixels
- The total number of threads is 80×64
- Some of the threads will have NULL data

Mapping grids to data

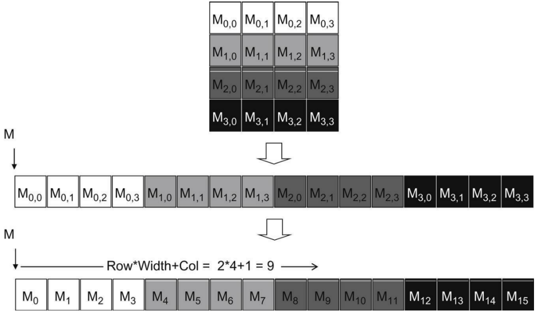


- The choice of dimensionality is driven by the data
- e.g., images are naturally 2D
- Above, we are using 5×4, 16×16 blocks for 76×62 pixels
- The total number of threads is 80×64
- Some of the threads will have NULL data
- `dim3 gridDim(5,4,1)` and `blockDim(16,16,1)`

Example: color-to-grayscale conversion

- Recall our initial example of converting a color image to grayscale
  - A color image is $n \times m \times 3$
  - Colors are stored as [red,green,blue] vectors
- If variables n and m are known, we can launch a kernel function as:
  - `dim3 dimGrid(ceil(m/16.0),ceil(n/16.0),1);`
  - `dim3 dimBlock(16,16,1);`
  - `colorToGrayscaleConversion<<<dimGrid, dimBlock>>>(d_Pin,d_Pout,m,n)`
- `d_Pin` and `d_Pout` are the pointers to the input and output arrays in the GPU

Memory organization

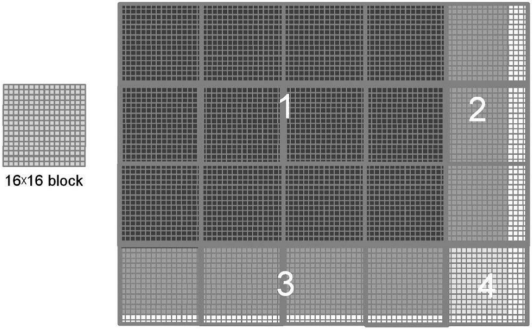


- The number of columns of dynamically allocated arrays is not known at compile time
- Thus, in CUDA C we cannot access elements directly, i.e., d_Pin[j][i]
- We have to treat 2D (and 3D) arrays as "flat", 1D arrays

Example: CUDA code

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * Pout,  unsigned
             char * Pin, 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 than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

Example: Block utilization

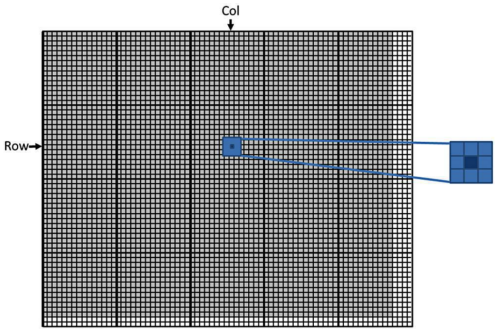


- The threads in region 1 are fully utilized
- Some of the ones in the other regions are not
  - Hence the need to check for row and column sizes

A more complex example: image blurring

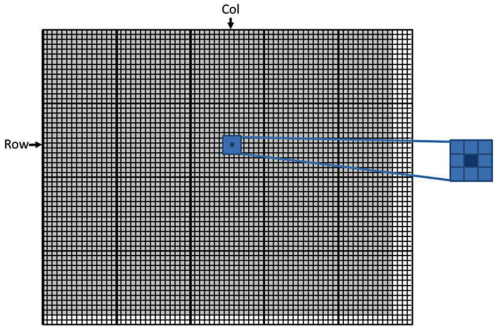
- Both vector addition and grayscale conversion are very simple examples
- We will now analyze a slightly more complicated operation: image blurring
- Here, we "smear" the value of a pixel by averaging it with its neighbors
- The effect is similar to myopia
- It is a special case of 2D convolution
    - We will revisit convolution later in the course

Image blurring



- Here, the neighbors of a pixel are the six pixels that share either an edge or a corner
  1. More generally, we can consider larger neighborhoods (e.g., those that are three pixels away or less)
- In image blurring, we replace the original value by a (potentially weighted) sum of the values of its neighborhood

Image blurring

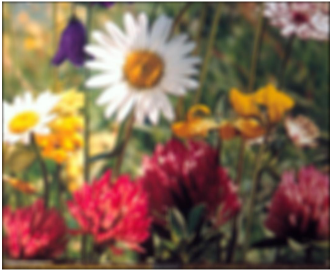


- In the simplest case, we take the mean of the seven values:

$$I(p) = \frac{1}{|N(p)|} \sum_{q \in N(p)} I(q)$$

For simplicity, here we assume that $p \in N(p)$

Image blurring



- Image blurring removes high-frequency details from an image
- It can be useful for reducing certain types of noise
  - e.g., the uneven illumination from flash in dark images

Image blurring: code

```
__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++; // Keep track of number of pixels in the avg
          }
        }
      }

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

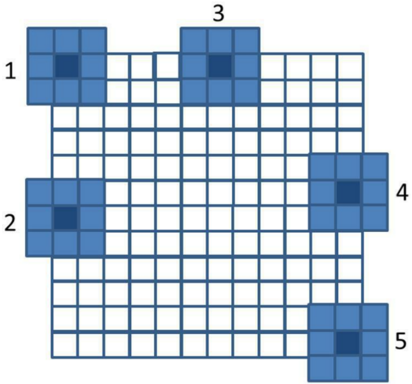
- Note how different processors may access the same pixel values (CREW model)
- What is the complexity of this code?

Image blurring: code

```
__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++; // Keep track of number of pixels in the avg
          }
        }
      }

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

- The two nested **for**-loops iterate $k =$ BLUR_SIZE steps
- So, the complexity is $O(k^2) = O(1)$ (if k is independent of n and m)

Border handling



- We need special checks to handle border and corner pixels
- Only valid pixels are used to compute the average value