Multidimensional Grid and Data

COSC4397: Sel Topics: Parallel Computations of GPU
Based on: Textbook, Chapter 3

Threads Organization: Blocks

- When a kernel is launched on GPU, a **grid** of **thread blocks** are spawned to execute the kernel function. (two level organization)
- Each thread will be uniquely id'ed by two coordinates: it's block index (blockIdx) within the grid and thread index (threadIdx) within that block.
- In general, the blockIdx is 3-dimensional vector; you can access the 3 coordinates via blockIdx.x, blockIdx.y, blockIdx.z. The same goes for the threadIdx.x, threadIdx.y, threadIdx.z
- The 1-d example below is a special case—just implicitly assuming the y,z dimensions are trivial—dimension 1.
- And in general, the gridDim and blockDim are 3-dimensional as well.

Organization of Threads

```c
#include <stdio.h>

__global__ void vecAdd(float *a, float *b, float *c)
{
    printf("blockIdx (%d,%d,%d), threadIdx (%d,%d,%d)\n",
           blockIdx.x, blockIdx.y, blockIdx.z,
           threadIdx.x, threadIdx.y, threadIdx.z);
    return;
}

int main()
{
    float *a, *b, *c;
    dim3 gridDim(2, 2, 1);
    dim3 blockDim(2,2, 4);
    vecAdd<<<gridDim, blockDim>>>(a, b, c);
    cudaDeviceSynchronize();
}
```

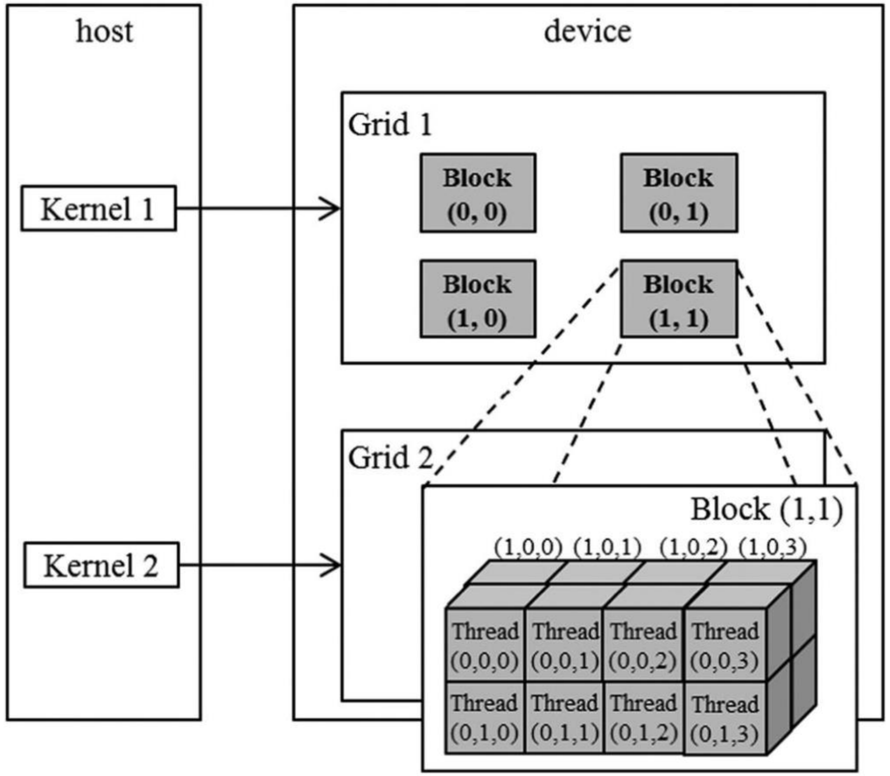


FIGURE 3.1 A multidimensional example of CUDA grid organization.

Organization of Threads

- Each thread block can contain at most 1024 threads (a fixed amount!)
- However, the number of thread blocks is rather unlimited.
- Why? What does this mean?
  - This will make much more sense when we talk about scheduling later
  - At high level, thread blocks are supposed to be independent; meaning they can’t cooperate (synchronize) at granular level.
  - Threads within a block are much more cooperative in terms of synchronization (barrier) and communications (shared memory, warp level, etc)

Mapping threads to multi-dimensional data

- Suppose we are dealing with a picture (2D array of pixels), converting each pixel from color scale to gray scale.
- For simplicity let's say we **map one thread to one pixel**. Let's further say we have a pixel map of size 62x76 pixels.
- It's natural to use a 2D blocks and 2D threads.
- First, we decide on the dim of thread block. 16x16 seems to be good (<1024 limit). So each thread block covers a 16x16 patch of pixels.
- What's the shape(dim) of thread blocks shall we launch?
  (4,5,1) – because we need 4 blocks to cover x-dim and 5 blocks to cover y-dim – we have 64x80 threads to cover 62x76 pixels.

Mapping: thread id -> pixel id

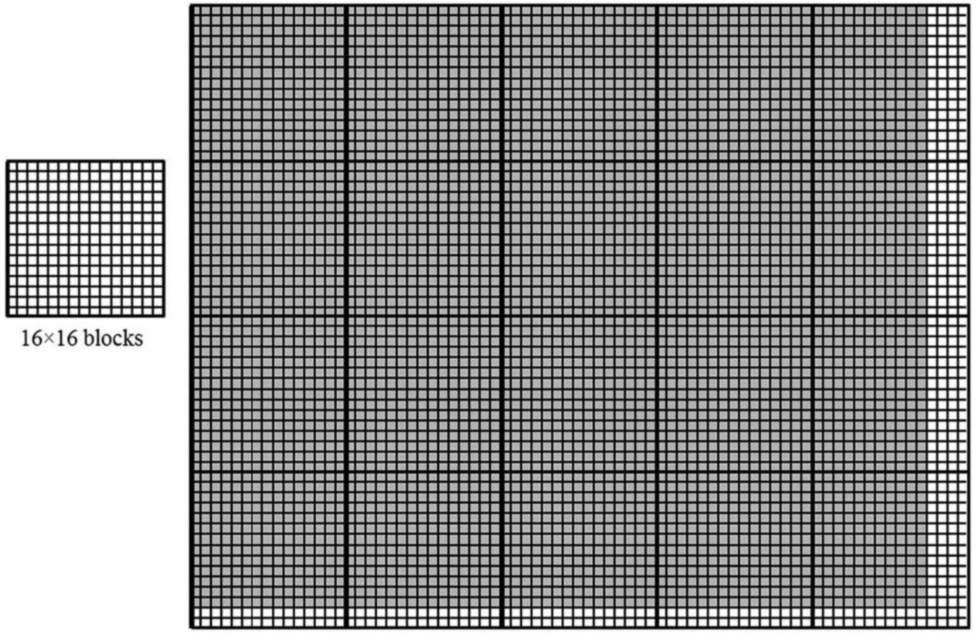


FIGURE 3.2 Using a 2D thread grid to process a 62×76 picture P.

- x dim -> vertical, y dim -> horizontal
- Top left (0,0), bottom right (3,4)

```
__global__ void convert(float *pixel, int rows, int cols)
{
    // each thread converts one pixel[i][j]
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    pixel[i*cols + j] /= 2;
    return;
}

int main()
{
    float *a, *b, *c;
    dim3 gridDim(4, 5, 1);
    dim3 blockDim(16,16, 1);
    convert<<<gridDim, blockDim>>>(a, b, c);
    cudaDeviceSynchronize();
}
```

Aside: Memory layout of multi-dimension array in C

- Machine memory space is flat (one-dimensional); C/C++ high dimension array (with >=2 indices) needs to be able to convert to a single index and back.
- In C/C++/Python(pytorch): Row Major
  Fortran/BLAS/LAPACK libraries: Column Major
- What is Row Major?

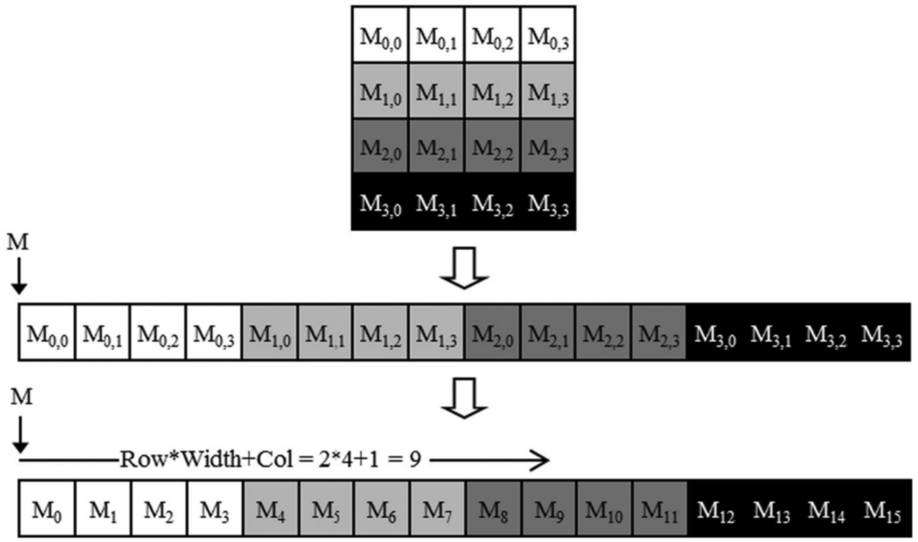
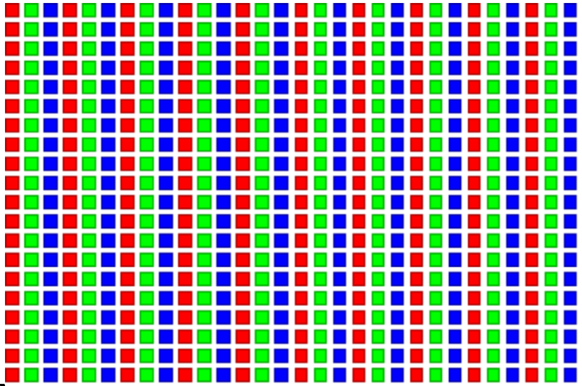


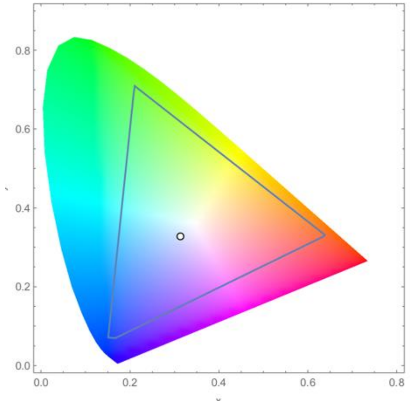
FIGURE 3.3 Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression j*Width+i for an element that is in the jth row and ith column of an array of Width elements in each row.

Example1: ColorToGrayScale

RGB Color Image Representation



- Each pixel in an image is an RGB value
- The format of an image's row is
  (r g b) (r g b) … (r g b)
- RGB ranges are not distributed uniformly
- Many different color spaces, here we show the constants to convert to AdobeRGB color space
  - The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction (1-y–x) of the pixel intensity that should be assigned to R
  - The triangle contains all the representable colors in this color space

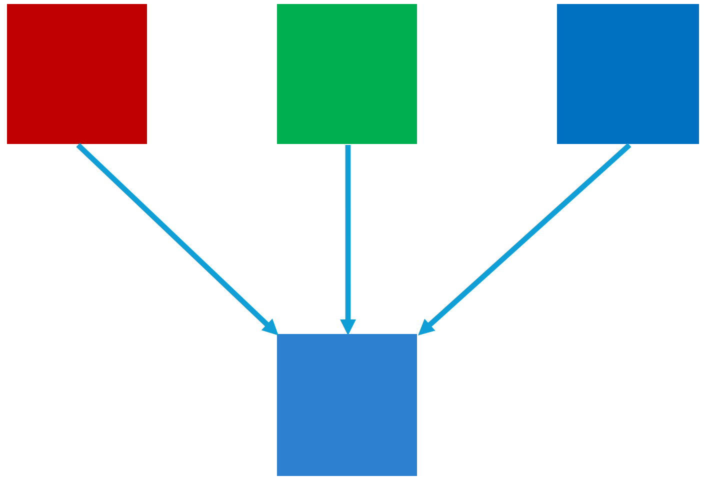


RGB to Grayscale Conversion

A grayscale digital image is an image in which the value of each pixel carries only intensity information.

Color Calculating Formula

- For each pixel (r g b) at (I, J) do:
  grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b
- This is just a dot product <[r,g,b],[0.21,0.71,0.07]> with the constants being specific to input RGB space



RGB to Grayscale Conversion Kernel

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;
    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*CHANNELS;
    unsigned char r =  rgbImage[rgbOffset      ]; // red value for pixel
    unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
    unsigned char b = rgbImage[rgbOffset + 3]; // 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;
 }
}
```

Example2: Image Blurring

- Previous examples are simple: each thread is completely independent, so parallelization is super easy!
- Image blurring: For each pixel, set it to be average of the 3x3 patch centered at it. This softens edges.
- This is a special case called **convolution**

FIGURE 3.6 An original image (*left*) and a blurred version (*right*).

```
__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 2xBLUR_SIZE x 2xBLUR_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 accumulated total
                }
            }
        }

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

Example3: Matrix Multiplication

- Matrix Multiplication (MatMul) is a nice computation kernel and worth a detailed study
  - It’s non-trivial to optimize
  - It’s one the rare kernel that can reach the hardwar FLOPS limit
  - Nowadays it often got its own chip: neural engines, TensorCores, etc
  - It’s the computational workhorse for Deep Neural network training & matrix computations
- Textbook definition: A: m*k, B: k*n, C: m*n
  A*B=C means:
  C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + ... + A[i][k-1]*B[k-1][j]

MatMul: naïve CUDA version

- Decompose by output: map each thread to a C[i][j]; that thread is charged with computing C[i][j]
- Naturally, 2D decomposition (2D blocks, 2D grids)

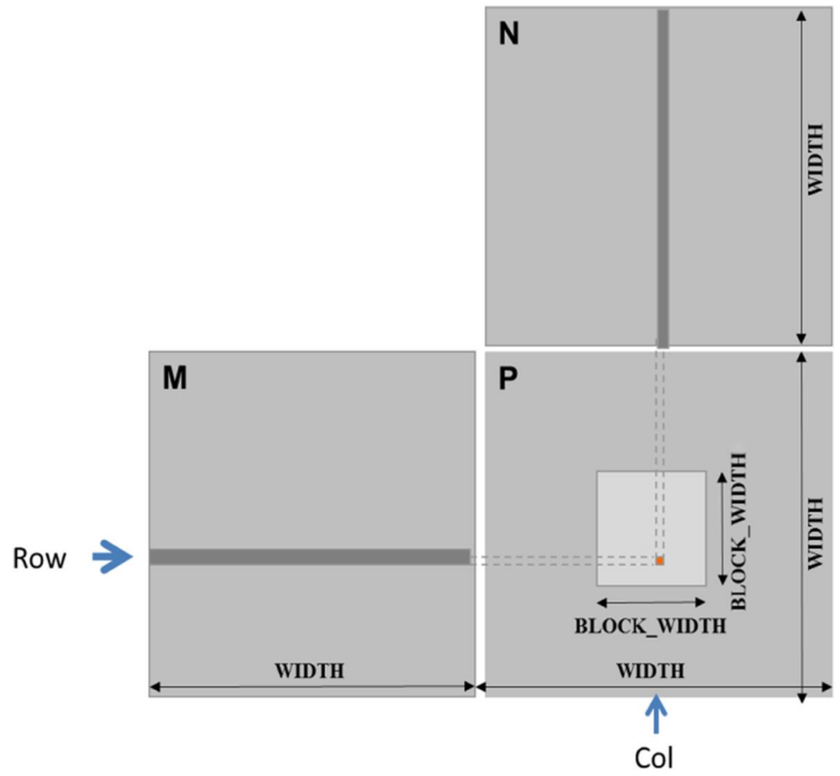


FIGURE 3.10 Matrix multiplication using multiple blocks by tiling P.

```
// compute C=A*B; where A shape is m*k, B shape is k*n, C shape is m*n
__global__ void naiveMatMul(float *A, float *B, float *C, int m, int n, int k)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    if (i < m && j < n) {
        C[i][j] = 0;
        for (int kk=0; k<k; k++) {
            //C[i][j] += A[i][kk]*B[kk][j]
            C[i*n + j] +=  A[i*k + kk] * B[kk*n + j];
        }
    }
}


int main()
{
    float *A, *B, *C;
    int m, n, k;
    // initialize a, b, c, m, n, k...
    dim3 gridDim((m+15)/16, (n+15)/16, 1);
    dim3 blockDim(16,16, 1);
    naiveMatMul<<<gridDim, blockDim>>>(A, B, C, m, n, k);
    cudaDeviceSynchronize();
}
```