CSE 4373/5373 - General Purpose GPU Programming

Multidimensional Grids and Data

Alex Dillhoff

University of Texas at Arlington

Multidimensional Grids and Data

- Grid representations are well suited for parallel computing.
- CUDA provides a **grid** of **blocks** of **threads**.
- These can be organized in 1D, 2D, or 3D.
- This allows us to match the structure of the data.

Grid Organization

Grid Organization

All threads share a block index, `blockIdx`, and a thread index, `threadIdx`.

These indices are three-dimensional vectors of type `dim3`.

```
struct dim3 {
    unsigned int x, y, z;
};
```

Grid Organization

Each grid is a 3D array of blocks, and every block a 3D array of threads.

Consider the kernel execution for `vecAdd` from Lab 0:

```
dim3 blocksPerGrid(32, 1, 1);
dim3 threadsPerBlock(128, 1, 1);
vecAdd<<<blocksPerGrid, threadsPerBlock>>>(a_d, b_d, c_d, n);
```

Grid Organization

This will execute with $32 \times 128 = 4096$ threads, which is well suited for a 1D vector.

If the input is a matrix, the launch dimensions should match its 2D structure.

We seemingly have two options: **grid size** or **block size**.

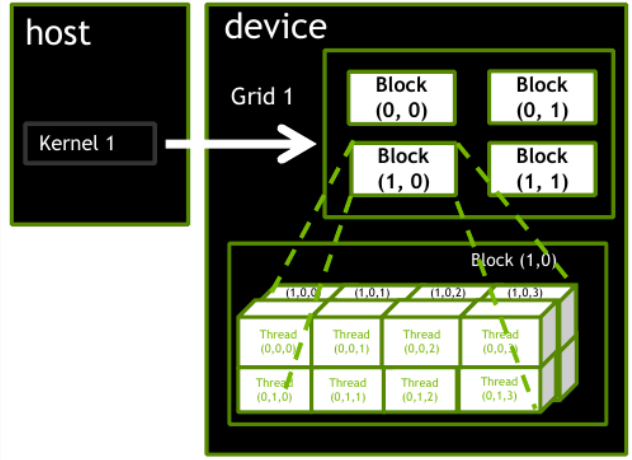
Grid Organization

This will execute with $32 \times 128 = 4096$ threads, which is well suited for a 1D vector.

If the input is a matrix, the launch dimensions should match its 2D structure.

We seemingly have two options: **grid size** or **block size**.

Grid Organization



A 2D grid of blocks, with 16 threads per block in 3D (source: NVIDIA DLI).

Grid Organization

Under such a configuration, we would make use of `gridDim.x`, `gridDim.y`, and `gridDim.z` to access the dimensions of the grid.

The dimensions of the block would be accessed with `blockDim.x`, `blockDim.y`, and `blockDim.z`.

The thread indices would be accessed with `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`.

Grid Organization

Would this be the best way to organize our launch configuration?

Grid Organization

Would this be the best way to organize our launch configuration?

Not exactly. We have no use for the 3D structure if we are only working with matrices.

Grid Organization

Consider an $n \times m$ matrix.

If the matrix is small enough, we could launch a single block with a 2D arrangement of threads.

For larger matrices, we would optimally split the work into multiple blocks.

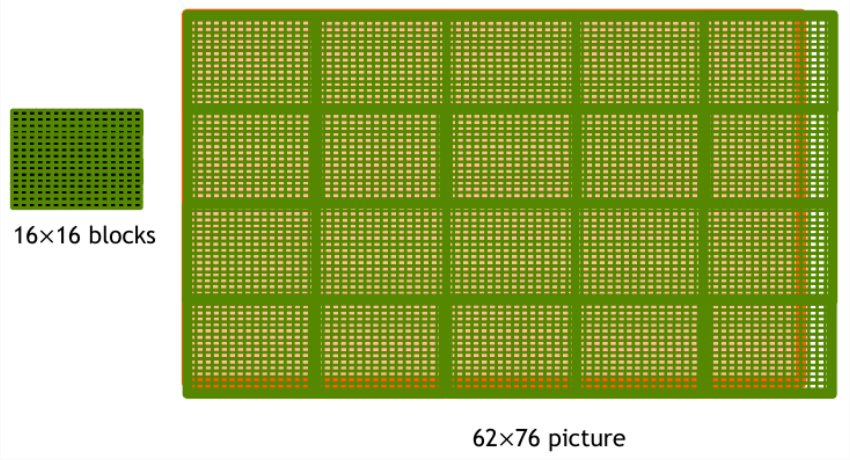
Grid Organization

This would allow us to perform more work in parallel.

Let $n = 62$ and $m = 76$.

If we chose a 16×16 block size, we would need $4 \times 5 = 20$ blocks to cover the entire matrix.

Grid Organization



A 2D grid of blocks with 16 threads each in 2D (source: NVIDIA DLI).

A note on Compute Capability

It is more important to dynamically adjust the grid size so that your program can adapt to varying input sizes.

As of CC 9.0, the maximum number of threads a block can have is 1024, this means that a 32×32 block is the largest we can do for matrix data.

A note on Compute Capability

If the input matrix is smaller than 32×32, then only a single block is needed.

The additional threads allocated to that block will be inactive for indices outside the range of our input.

A note on Compute Capability

If the input matrix is larger than 32×32, additional blocks should be added to the grid to accommodate the increased size.

It is safe to keep the block size fixed, but the grid size **must** be dynamic.

Optimal Launch Parameters

Is it better to have fewer blocks that maximize the amount of threads per block? Or is it better to have more blocks with fewer threads per block?

The current maximum number of threads per block is 1024.

In practice, a maximum block dimension size of 128 or 256 is ideal. This has more to do with the specific problem and the amount of shared memory required. You will explore this question in Lab 1.

Example: Color to Grayscale

Color to Grayscale

Converting a color image to grayscale is an embarrassingly parallel problem.

Each pixel in the output corresponds to a single pixel in the input.

Color to Grayscale

Computing the grayscale value of a pixel is a linear combination of the red, green, and blue values.

$$\text{gray} = 0.299 \times \text{red} + 0.587 \times \text{green} + 0.114 \times \text{blue}$$

Color to Grayscale

A CPU implementation would require a `for` loop over the exact number of pixels.

The CUDA kernel for this is straightforward since it only depends on the current pixel.

The only real challenge is to compute the correct indices for each thread.

Color to Grayscale

```
void colorToGrayscale(unsigned char *rgbImage, unsigned char *grayImage,
                      int numRows, int numCols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= numCols || y >= numRows) return;

    int index = y * numCols + x;
    int rgbOffset = index * 3;
    unsigned char r = rgbImage[rgbOffset];
    unsigned char g = rgbImage[rgbOffset + 1];
    unsigned char b = rgbImage[rgbOffset + 2];
    float channelSum = 0.299f * r + 0.587f * g + 0.114f * b;
    grayImage[index] = channelSum;
}
```

Color to Grayscale

This code assumes an RGB image where each pixel is represented by three unsigned characters.

It is standard convention in C to pass a pointer to the first element of the array.

This implies that we cannot use the `[]` operator to access the elements in a multidimensional way. **We must compute the index ourselves.**

Color to Grayscale

In C, multi-dimensional arrays are stored in row-major order.

To compute the index of row `j` and column `i` in a 2D array, we need to skip over `j` rows and `i` columns.

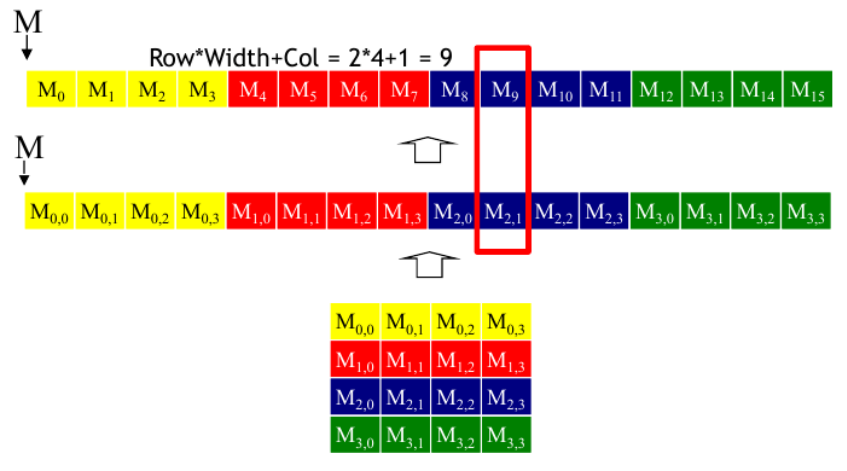
The total number of columns is the width of the array. The total number of rows is the height of the array.

Color to Grayscale

The index is computed as follows:

```
int index = j * width + i;
```

Color to Grayscale



A 2D array stored in row-major order (source: NVIDIA DLI).

Color to Grayscale

Since the image is now represented as a flat 1D array, we can use the index computed previously to access the correct pixel.

The image is typically stored in the same row-major format, **although this is not always the case.**

You should always check the documentation for the image format you are using.

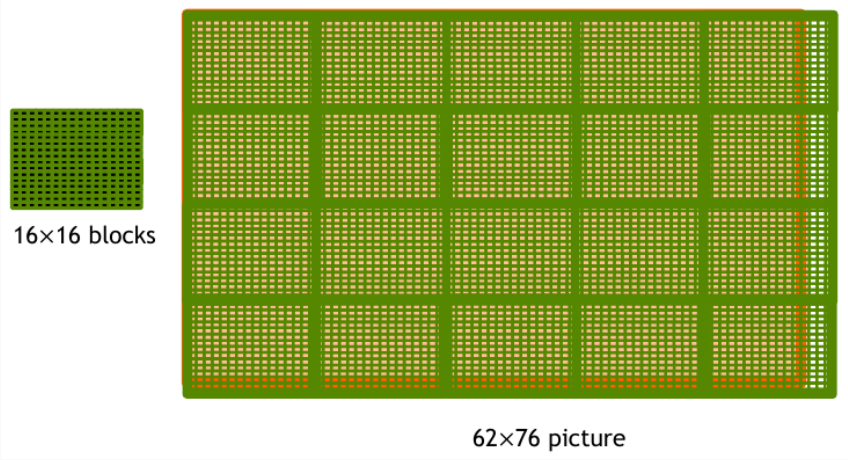
Launch Configurations

As stated above, we are going to launch 20 blocks in a 4×5 grid.

Each block will have 256 threads arranged in a 16×16 2D configuration.

This totals to $20 \times 256 = 5120$ threads.

Launch Configurations



A 2D grid of blocks with 16 threads each in 2D (source: NVIDIA DLI).

Launch Configurations

The previous figure shows this configuration overlaid on a 76×62 image.

That means we have 4712 pixels that need to be converted.

The remaining 408 threads will be idle.

Launch Configurations

Do all 5120 threads launch at the same time?

What if the number of pixels exceeded the number of threads available on the GPU?

Launch Configurations

Do all 5120 threads launch at the same time?

What if the number of pixels exceeded the number of threads available on the GPU?

The short answer is that the GPU will launch as many threads as possible, but the long answer is slightly more complicated and will require a deeper understanding of the CUDA execution model.

Launch Configurations

The previous kernel can be launched with the following call.

```
dim3 blockSize(16, 16, 1);
dim3 gridSize(4, 5, 1);
colorToGrayscale<<<gridSize, blockSize>>>(rgbImage,
                                          grayImage,
                                          numRows,
                                          numCols);
```

Launch Configurations

This produces 20 blocks, each with 256 threads arranged in a 16×16 2D configuration.

This is enough to cover the 62×76 image, but would not be enough to cover a larger image.

No longer embarrassing: overlapping data

Overlapping data

Now comes the next step in shaping your parallel thinking skills.

What if the thread relies on multiple data points that may be used by other threads?

This is further complicated with problems that require some computation to complete before a thread can begin its work.

Overlapping data

Image blurring presents a slightly more complicated problem than color to grayscale.

This is a common technique used in image processing to reduce noise and detail.

The basic idea is to replace each pixel with a weighted average of its neighboring pixels.

Overlapping data

The size of the neighborhood is called the **kernel size**.

The kernel size is typically an odd number so that the pixel of interest is in the center of the neighborhood.

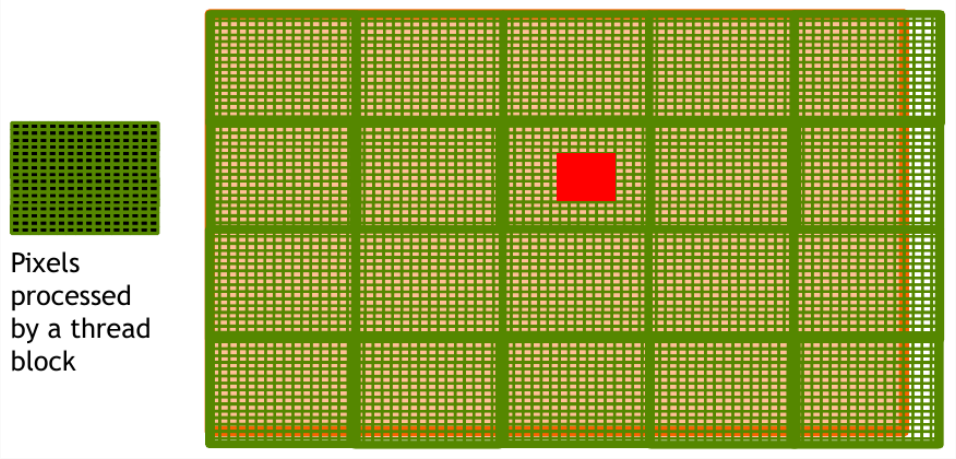
Overlapping data

The core operation behind blurring is called a **convolution**.

Given a kernel size of 5×5 centered on a pixel, we will compute the weighted average of the 25 pixels in the neighborhood.

To keep it simple, the weights will be uniform.

Overlapping Data



A blurring kernel (red) centered on a pixel (source: NVIDIA DLI).

Overlapping data

Given a pixel location (x, y), we can compute the index of the pixel in the neighborhood as follows:

```
int index = (y + ky) * numCols + (x + kx);
```

where *ky* and *kx* are the row and column indices of the kernel.

Overlapping data

The kernel is centered on the pixel of interest, so *ky* and *kx* range from -2 to 2.

The total number of pixels in the neighborhood is $5 \times 5 = 25$.

Overlapping data

```
float sum = 0.0f;
int numPixels = 0;
for (int ky = -2; ky <= 2; ky++) {
    for (int kx = -2; kx <= 2; kx++) {
        if (x + kx < 0 || x + kx >= numCols) continue;
        if (y + ky < 0 || y + ky >= numRows) continue;
        int index = (y + ky) * numCols + (x + kx);
        sum += image[index];
        numPixels++;
    }
}
image[y * numRows + x] = sum / numPixels;
```

Overlapping data

Some extra care will be needed to account for pixels outside the boundaries.

There are several strategies to handle out-of-bounds pixels.

The simplest is to ignore them. We will explore other strategies when discussing convolutions.

Matrix Multiplication

Matrix Multiplication

Matrix multiplication is one of the most important operations in linear algebra.

It is one of the most widely called operations in deep learning, for example.

Parallelizing this and other linear algebra operations has resulted in an explosion of research and applications ranging from computer vision to computational fluid dynamics.

Matrix Multiplication

Exploring the parallelism of matrix multiplication will give us a deeper understanding of the CUDA programming model.

It will also serve as a jumping off point for more advanced topics like shared memory and convolutional neural networks.

Matrix Multiplication

Let $A = \mathbb{R}^{m \times n}$ and $B = \mathbb{R}^{n \times p}$ be two matrices.

The product $C = AB$ is defined as follows:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \quad \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, p$$

Matrix Multiplication

This operation is only defined on compatible matrices.

That is, the number of columns in A must equal the number of rows in B.

The resulting matrix C will have m rows and p columns.

CPU Implementation

There is a double `for` loop to iterate through each element in the *output* matrix.

The inner loop computes the dot product of the *i*th row of *A* and the *j*th column of *B*.

The dot product is computed by summing the element-wise product of the two vectors.

CPU Implementation

```
void matrixMultiplyCPU(float *A, float *B, float *C,
                       int m, int n, int p) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            float sum = 0.0f;
            for (int k = 0; k < n; k++) {
                sum += A[i * n + k] * B[k * p + j];
            }
            C[i * p + j] = sum;
        }
    }
}
```

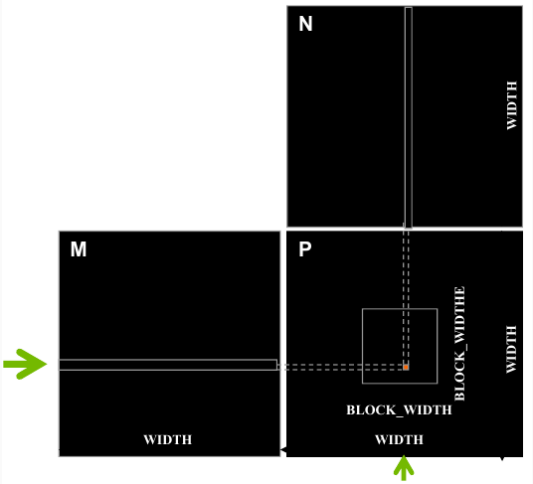
GPU Implementation

For a parallel implementation, we can reason that each thread should compute a single element of the output matrix.

To compute element C_{ij}, the thread needs access to row i from A and column j from B.

Each thread is simply computing the dot product between these two vectors.

GPU Implementation



Matrix multiplication (source: NVIDIA DLI).

GPU Implementation

The output matrix is separated into blocks based on our block size.

When writing the kernel, it is necessary to make sure that the index is not out of bounds.

GPU Implementation

```
void matrixMultiplyGPU(float *A, float *B, float *C,
                       int m, int n, int p) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= m || col >= p) return;

    float sum = 0.0f;
    for (int k = 0; k < n; k++) {
        sum += A[row * n + k] * B[k * p + col];
    }
    C[row * p + col] = sum;
}
```

Launch Configuration

The launch configuration is similar to the previous examples.

We will launch a 2D grid of blocks, each with a 2D arrangement of threads.

The block size will be 16×16 and the grid size will be $m/16 \times p/16$.

GPU Implementation

```
dim3 blockSize(16, 16, 1);
dim3 gridSize((p + blockSize.x - 1) / blockSize.x,
              (m + blockSize.y - 1) / blockSize.y, 1);
matrixMultiplyGPU<<<gridSize, blockSize>>>(A_d, B_d, C_d, m, n, p);
```

GPU Implementation

What happens when the output matrix size exceeds the number of blocks per grid and threads per block?

GPU Implementation

What happens when the output matrix size exceeds the number of blocks per grid and threads per block?

The largest square grid size that can be launched is 65536×65536.

The largest square block that can be launched is 32×32.

This means that your input matrix would need to be larger than 2097152×2097152.

Summary

Summary

- Grids are a natural way to organize parallel work.
- CUDA provides a 3D grid of blocks, each with a 3D arrangement of threads.
- The dimensions of the grid and blocks can be accessed with `gridDim` and `blockDim`.
- The thread indices can be accessed with `threadIdx`.
- The launch configuration should match the structure of the data.

What’s next?

What's Next?

The complexity was slightly increased by considering multidimensional data.

Matrices are a prime example of this.

The algorithms explored required us to consider multiple input values to compute a single output value.

However, the computation did not rely on any thread communication.

What’s Next?

Before diving into more complex operations like thread synchronization, was need a better understanding of the GPU’s architecture and memory hierarchy.

With this knowledge at our disposal, we can begin to optimize our kernels for maximum performance.