GPU Series – VII

CUDA THREAD ORGANIZATION

Processing Multidimensional Data

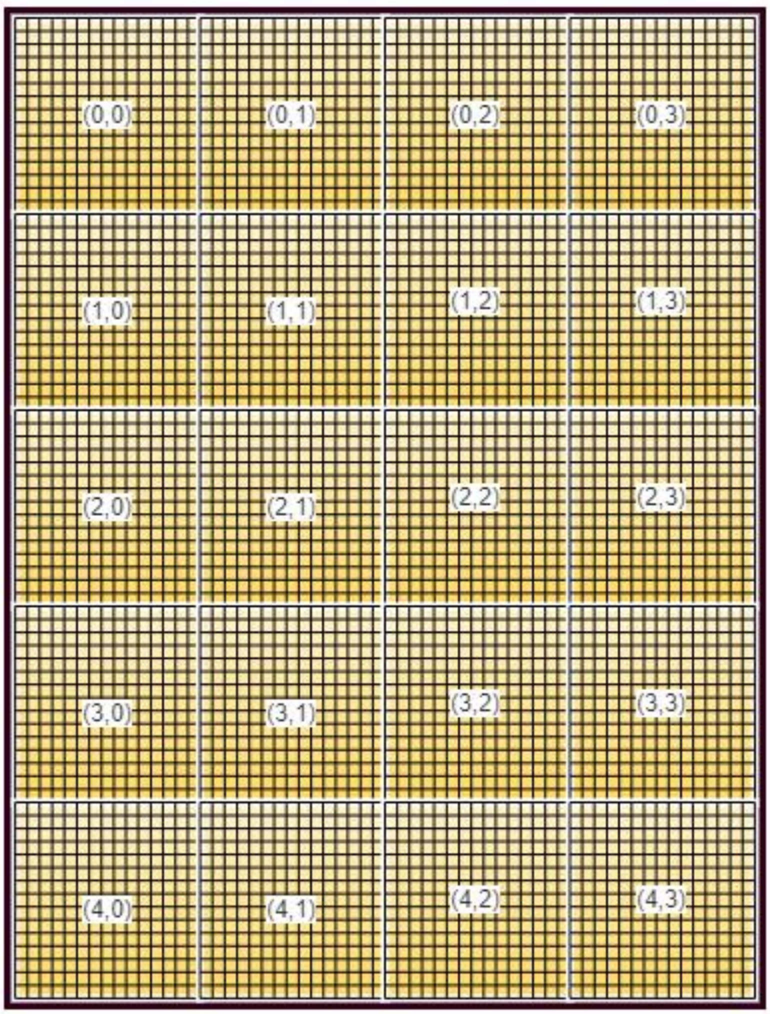
1. Organizing CUDA Threads

We can organize CUDA threads in three dimensionalities:

- **1D Thread Organization**: Typically used for linear data structures like vectors.
- **2D Thread Organization**: Ideal for processing images, which are naturally two-dimensional arrays of pixels.
- **3D Thread Organization**: Used for volumetric data, such as 3D models or simulations involving 3D space.

Let's explore an example using a 2D thread grid for image processing.

2. Example: 2D Thread Grid for Image Processing



Grid of CUDA Blocks with Thread Blocks

This image shows a 5x4 grid of blocks, where each block is labeled with its block index, such as (0,0), (0,1), etc. Each block contains a 16x16 grid of threads.

Imagine you have a 2D image that is 76 pixels wide and 62 pixels tall (a 76 × 62 matrix). If you decide to use a 16 × 16 block of threads, meaning each block will have 256 threads arranged in a 16 by 16 grid, you need to figure out how many such blocks are required to cover the entire image.

Calculating Blocks Needed:

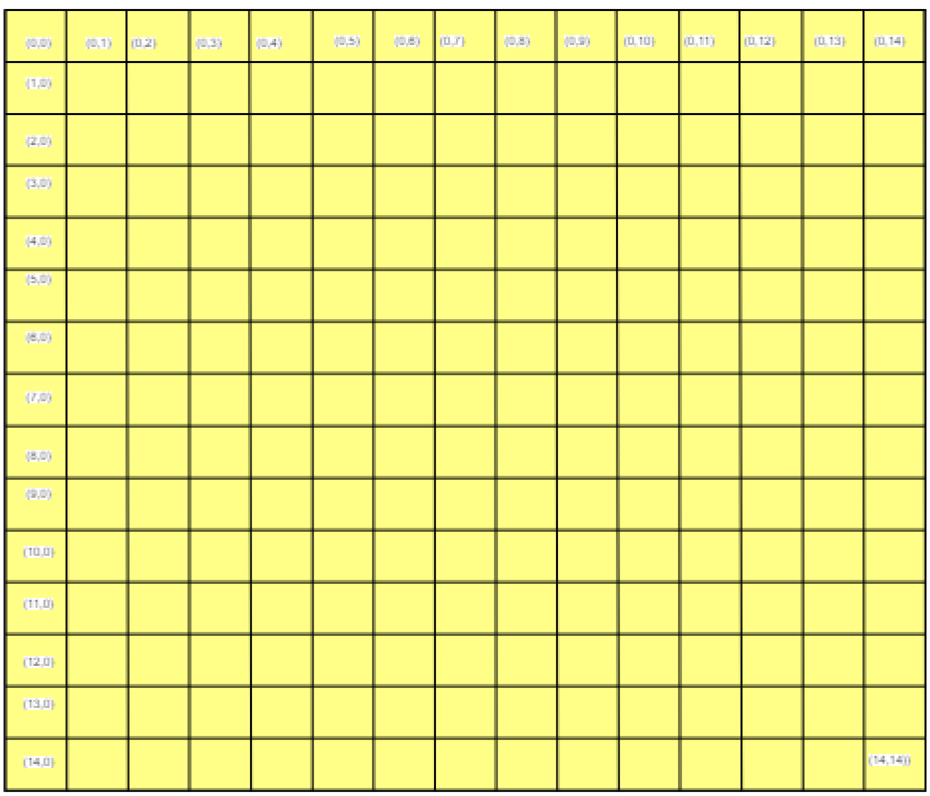
- **In the x direction (width)**: 76 pixels / 16 threads per block = 4.75, rounded up to 5 blocks.
- **In the y direction (height)**: 62 pixels / 16 threads per block = 3.875, rounded up to 4 blocks.
- Therefore, you need 5 blocks horizontally and 4 blocks vertically, resulting in a total of 20 blocks.

However, this configuration generates more threads than there are pixels:

- 5 blocks * 16 threads/block = 80 threads in the x direction.
- 4 blocks * 16 threads/block = 64 threads in the y direction.

You end up with 80 × 64 = 5120 threads to process 76 × 62 = 4712 pixels. This discrepancy means that some threads won't correspond to any pixels and need to be handled properly.

3. Understanding Threads and Blocks in More Detail



Thread Indexing within a Block

This image depicts a 16x16 grid of threads within a single CUDA block, where each thread is labeled with its thread index, such as (0,0), (0,1), etc.

- **Picture as a 2D Array**:
    - A picture can be thought of as a 2D array of pixels.
    - In this example, the picture has dimensions of 76 pixels in the x direction (horizontal) and 62 pixels in the y direction (vertical).
    - Each pixel in this array has an (x, y) coordinate, where x is the horizontal position and y is the vertical position.

Threads and Blocks

- **Threads**: In CUDA, a thread is the smallest unit of execution. Each thread can perform operations on a single element of data, such as a single pixel in an image.
- **Blocks**: Threads are grouped into blocks. Each block contains a certain number of threads arranged in a grid. In this example, we use 16 threads in the x direction and 16 threads in the y direction, forming a 16 × 16 block.
- Thus, each block has 256 threads (16 × 16 = 256).

4. Mapping Threads to the Picture

- **Grid of Blocks**:
    - To process the entire picture, which is larger than a single block, multiple blocks are needed.
    - The picture is 76 pixels wide and 62 pixels tall. Since each block is 16 × 16, we need to figure out how many blocks are required to cover the entire image.

Calculating Number of Blocks:

- **In the x direction (width)**:
    - The picture is 76 pixels wide. Each block covers 16 pixels.
    - Number of blocks needed = 76 / 16 = 4.75, which we round up to 5 blocks.
- **In the y direction (height)**:
    - The picture is 62 pixels tall. Each block covers 16 pixels.
    - Number of blocks needed = 62 / 16 = 3.875, which we round up to 4 blocks.
- Therefore, we need a 5 × 4 grid of blocks to cover the picture. This totals 20 blocks.

Since the number of threads exceeds the number of pixels, you need to include a condition in your kernel to ensure that threads only process valid pixels:

```
if (Col < width && Row < height) {
   // Only process if the thread corresponds to a valid pixel
}
```

This condition ensures that any threads that fall outside the bounds of the image do not attempt to process pixels that don't exist, which would cause errors or undefined behavior.

5. Addressing 2D Arrays in C

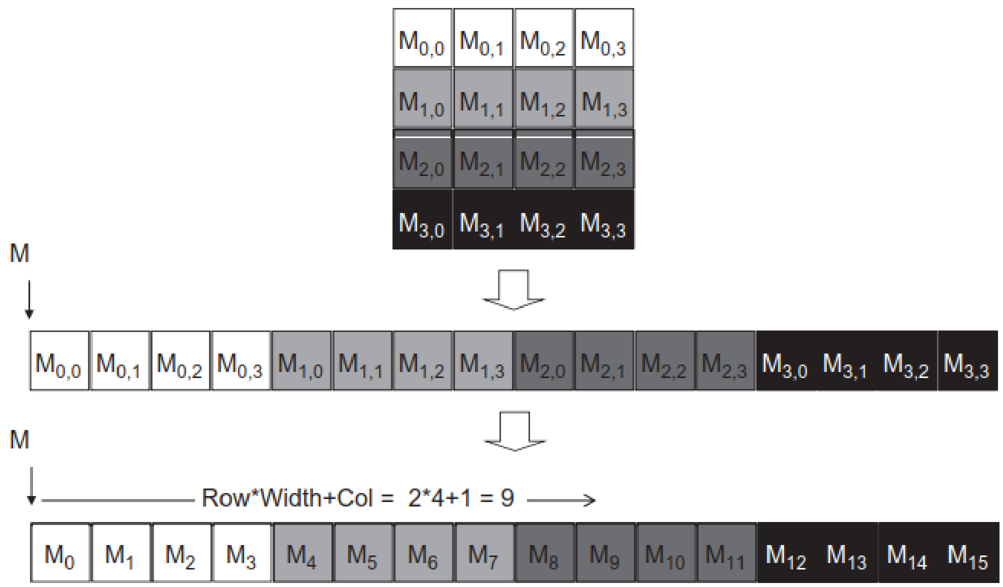


FIGURE 3.3

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 *j* th row and *i* th column of an array of Width elements in each row.

Img Src: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu

Before writing the program, let's revise the basics of how to address 2D arrays in C. Arrays are stored in a linear memory space. Thus, 2D arrays must be converted to a 1D array to work with them in programming.

Row-Major Order:

- **Row-Major Layout**: This is a method used to linearize 2D arrays.
- In this layout, all the elements of the first row of the array are stored consecutively in memory, followed by all the elements of the second row, and so on.

Example:

Suppose you have a 2D array M with dimensions 4 × 4 (4 rows and 4 columns). The array might look like this:

To access an element in this 2D array using a 1D index, you need to compute the index based on its row and column in the original 2D layout.

1D index = row × width of the row (number of columns) + column

In this formula, row is the row index, and column is the column index.

6. Example: Color to Greyscale Conversion in CUDA

Now, let’s see how to write a CUDA code for color to greyscale conversion.

Conversion Formula:

The goal of the *colorToGreyscaleConversion* kernel is to convert each color pixel in an image to a corresponding greyscale pixel. This involves combining the red, green, and blue (RGB) values of a pixel into a single intensity value that represents its greyscale equivalent.

The conversion formula used is:

$$L = 0.21 \times r + 0.72 \times g + 0.07 \times b$$

- **Here**: r, g, and b are the red, green, and blue components of the color pixel, respectively.
- These components are combined using specific weights to produce a single greyscale value L.

Thread Organization:

- **Threads in CUDA**: Each pixel in the image is processed by a separate CUDA thread. The threads are organized into blocks, and the blocks are organized into a grid.

Grid and Block Dimensions:

- The grid dimensions (*gridDim.x* and *gridDim.y*) determine how many blocks are used in the horizontal and vertical directions.
- The block dimensions (*blockDim.x* and *blockDim.y*) determine how many threads are in each block in the horizontal and vertical directions.

Calculating Pixel Positions:

- **Horizontal (x) Position**:
    - The horizontal position (Col) of the thread within the image is calculated using:

$$Col = blockIdx.x \times blockDim.x + threadIdx.x$$

- This formula gives a unique value for Col for each thread, ranging from **0** *to* $gridDim.x \times blockDim.x - 1$, ensuring each pixel in the horizontal direction is covered.

Vertical (y) Position:

- Similarly, the vertical position (Row) is calculated as:

$$Row = blockIdx.y \times blockDim.y + threadIdx.y$$

Indexing the Pixels:

- **Greyscale Pixel**:
    - To find the position of the greyscale pixel in the 1D output array Pout, the index is calculated as:

$$greyOffset = Row \times width + Col$$

    - This *greyOffset* is where the calculated greyscale value will be stored.
- **Color Pixel**:
    - For the color pixel in the input image, which has three components (R, G, B), the index (*rgbOffset*) is calculated as:

rgbOffset = greyOffset × 3

- This index points to the start of the color data for the pixel (the red component), followed by the green and blue components.

Here's the CUDA kernel code for converting a color image to greyscale:

```
__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) {
        int greyOffset = Row * width + Col;
        int rgbOffset = greyOffset * 3;

        unsigned char r = Pin[rgbOffset];      // Red
        unsigned char g = Pin[rgbOffset + 1];  // Green
        unsigned char b = Pin[rgbOffset + 2];  // Blue

        Pout[greyOffset] = 0.21f * r + 0.72f * g + 0.07f * b;  // Convert to greyscale
    }
}
```

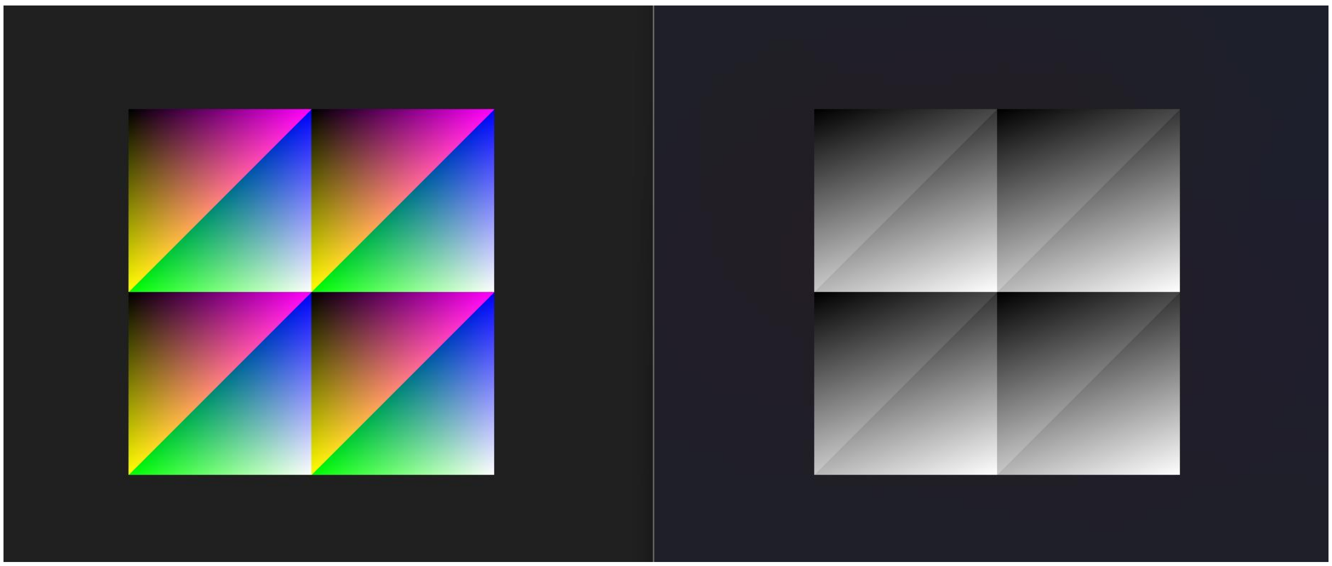
Complete Code:

```
#include <iostream>
#include <cuda_runtime.h>
#include <opencv2/opencv.hpp>

// Kernel function to convert color image to greyscale
__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) {
        int greyOffset = Row * width + Col;
        int rgbOffset = greyOffset * 3;

        unsigned char r = Pin[rgbOffset];      // Red
        unsigned char g = Pin[rgbOffset + 1];  // Green
        unsigned char b = Pin[rgbOffset + 2];  // Blue

        Pout[greyOffset] = 0.21f * r + 0.72f * g + 0.07f * b;  // Convert to greyscale
    }
}
```

```
// Host code
int main() {
    // Image dimensions
    int width = 512;  // Example width
    int height = 512; // Example height
    int numPixels = width * height;

    // Generate a simple synthetic color image using OpenCV (just for visualization)
    cv::Mat h_colorImg(height, width, CV_8UC3);
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            h_colorImg.at<cv::Vec3b>(i, j) = cv::Vec3b(j % 256, i % 256, (i + j) % 256); //
Simple gradient
        }
    }

    // Save the original color image for comparison
    cv::imwrite("color_image.png", h_colorImg);

    // Allocate host memory for the output image
    unsigned char *h_Pout = new unsigned char[numPixels];  // Greyscale image

    // Allocate device memory
    unsigned char *d_Pin, *d_Pout;
    cudaMalloc(&d_Pin, numPixels * 3 * sizeof(unsigned char));
    cudaMalloc(&d_Pout, numPixels * sizeof(unsigned char));

    // Copy input data from host to device
    cudaMemcpy(d_Pin, h_colorImg.data, numPixels * 3 * sizeof(unsigned char),
cudaMemcpyHostToDevice);

    // Define grid and block dimensions
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x, (height + dimBlock.y - 1) /
dimBlock.y, 1);

    // Launch the kernel
    colorToGreyscaleConversion<<<dimGrid, dimBlock>>>(d_Pout, d_Pin, width, height);

    // Copy the result back to the host
    cudaMemcpy(h_Pout, d_Pout, numPixels * sizeof(unsigned char), cudaMemcpyDeviceToHost);

    // Convert the output array to an OpenCV matrix and save the result
    cv::Mat h_greyscaleImg(height, width, CV_8UC1, h_Pout);
    cv::imwrite("greyscale_image.png", h_greyscaleImg);

    // Free device memory
    cudaFree(d_Pin);
    cudaFree(d_Pout);
```

```
    // Free host memory
    delete[] h_Pout;

    std::cout << "Color to greyscale conversion completed successfully!" << std::endl;
    std::cout << "Images saved as color_image.png and greyscale_image.png" << std::endl;

    return 0;
}
```

Output:



~~ To be Continued ~~

Reference: Programming Massively Parallel Processors by David B.Kirk, Wen-mei W.Hwu