COP6616 Parallel Computing
Instructor: Scott Piersall
Student: Aaron Goldstein
Fall 2024
Assignment 3

1. **Short Answer Questions (14 points).**

   1). What is the difference between data spatial locality and temporal locality?

   Answer:

   Data spatial locality refers to the tendency of a program to access data locations that are close to each other in memory within a short time period. This means that if a particular data item is accessed, nearby data items (e.g., the next elements in an array) are likely to be accessed soon. Temporal locality, on the other hand, refers to the tendency of a program to access the same data location multiple times within a relatively short time frame. This is common in scenarios like loops where the same variables are accessed repeatedly. Both types of locality are leveraged to optimize cache performance: spatial locality through fetching contiguous blocks of data (cache lines) and temporal locality through keeping recently accessed data in the cache for quick reuse.

   ---

   2). In the MPI Gather routine, does the root contribute data?

   Answer:

   Yes, each process, including the root sends the contents of its send buffer to the root process. Afterwards the root process receives the messages and stores them in rank order.

   ----

   3). For the following MPI code, decide whether there is/are any potential problem(s) with it. If there are problems, please fix them.

```
if (rank =0){
   MPI_Barrier(…);
} else {
   do something;
}
```

Answer:

```
if (rank == 0) {
  // do something
}
else {
  // do something else
}

MPI_Barrier(MPI_COMM_WORLD); // Move barrier here so all processes call it
```

Explanation: MPI_Barrier() is a collective operation that will only complete once invoked by all the MPI processes. No processes can exit the barrier until all processes have entered it, if some processes don't reach it, it will hang forever.

---

4). In CUDA programs, if we compare "structure of arrays" and "array of structures", which one is offers better memory access performance? WHY?

Answer:

Although a Structure of Arrays (SoA) generally offers better memory access performance than an Array of Structures (AoS), the optimal solution depends on the access pattern of the algorithm. For example, if you are processing the R,G, and B components together of an array of RGB pixels, then AoS would be preferable as the elements of each structure would be contiguous in memory allowing for better cache utilization. This would also facilitate contiguous access by threads where for example, thread 0 accesses memory address N, thread 1 accesses memory address N, + 1 thread 2 should access N+2, and so on. Contiguous access is required by coalescing which refers to combining multiply memory accesses by threads into a single memory transaction, which greatly improves bandwidth. If you still wanted to operate on a series of RGB pixels, but you only wanted to process all of the R components separately, structure of arrays would suit you better as all the r components will be contiguous in memory, allowing for coalescing.

---

5). Discuss the difference between the following GPU memory: global, local, shared, texture, and constant. Which one is the fastest?

Answer:
Here is a brief description of each of the above mentioned above, starting from the slowest, all the way to the fastest.

Slow Memory:
Global Memory :
- Largest memory pool in the GPU and can be accessed by all threads across all blocks.
- Is relatively slow and resides in off-chip DRAM memory.
- Typically used for things like input / output data and global constants.
- Requires optimization to avoid performance degradation (careful access patterns such as coalescing)

Local memory:

- Is about as slow as global memory
- Memory that is private to each thread and lasts the lifetime of that thread
- Like global memory, it resides off-chip in device RAM
- Local memory is cached
- Used for storing temporary or automatic variables
- Used when register space is insufficient (register spilling)

Moderate Speed Memory:

Texture Memory:

- Resides off-chip and is cached
- Read only memory on the device that can be faster than global memory when all reads in a warp are physically adjacent.
- All threads across all blocks can access these memory types
- It is a special type of memory optimized for textures.
- Optimized for 2D / 3D spatial locality, images being an example.

Constant Memory:

- Resides off-chip and is cached
- It is a special type of memory optimized for constant memory
- Used for read-only data shared across all threads

- Optimized for broadcast access where all threads in a warp read the same memory location.

Fast Memory:

Shared Memory:

- Smaller but fast memory pool
- Resides on chip
- Data is visible to all threads within that block and lasts for the duration of the block.
- Allows for threads to communicate and share data between one another playing a key role in inter-thread communication
- Divided into banks and susceptible to conflicts when threads try to access the same bank at the same time.

----

6). If shared memory in a GPU is defined and used in the following way, under what scenario about the variable "s", is there a bank conflict? Under what scenario is there no bank conflict? Why?

```
__shared__ float shared[16];
float foo =   shared[baseIndex + s * threadIdx.x];
```

Answer:

Shared memory is divided into equally sized memory banks that can be accessed simultaneously. Consecutive 32-bit words are mapped to consecutive banks, where each bank can service one memory request per clock cycle. Modern GPUs typically have 32 banks, so bank 0 will store word 0, bank 1 will store word 1, bank 31 will store word 31, etc. Bank 0's next word will be word 32. If s is 1 and assuming a base index of 0, the consecutive threads will map directly to the banks, thread 0 to bank 0, thread 1 to bank 1, etc. However, if s is 32, a multiple of the number of banks, you will have terrible bank conflicts because thread 1 will try to access word 32, thread 2 will try to access word 64. These are all words in bank 0, and the bank can only service one of these requests at a time, resulting in serialized memory access and reduced performance! To avoid this issue, we should take care to use access patterns that result in consecutive threads accessing consecutive memory locations or banks.

---

7). If shared memory in GPU is defined and used in the following way, what kind of bank conflict (such as 2-way, 4-way, 8-way) does it have? Why?

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```

Answer:

For modern GPUs, each bank handles 32-bit or 4 byte words and each character is 1 byte. This means that 4 consecutive chars fit into one 32-bit word which will be stored in a bank. Now consider a scenario where shared[] is an array with 4 characters and you have four threads. These four threads will access four consecutive characters or bytes of the same word, hence they all are accessing the same word, and thus the same bank! This means that we have a 4-way bank conflict. Luckily this doesn't cause a performance penalty for GPUs devices with a Compute Capability of 2.0+. This is because when multiple threads of the same warp access any address within the same 32-bit word, even though the addresses fall in the same bank, in the case of read access as in the code excerpt above, modern GPUs broadcast the word to all requesting threads simultaneously.

---

2. **(15 points)** Please write **a CUDA program** to compute the Euclidean distance, similar to the problem in assignment 2.

You may design your code using the following steps:

- Declare the arrays (host and device). All arrays should be dynamically allocated; the host arrays can be allocated either with `malloc` or `new`, while the device arrays should be allocated with `cudaMalloc`.
- Print the number of CUDA-enabled hardware devices attached to the system by calling `cudaGetDeviceCount`.
- Print at least 3 interesting properties of Device 0, including the device name, by calling `cudaGetDeviceProperties`. The first argument to this function is a pointer to a struct of type `cudaDeviceProp`.
- Calls `InitArray` to initialize the host arrays. `InitArray` initializes an integer array with random numbers within a fairly small range (0 to 99).
- Calls `cudaMemcpy` to copy the host input arrays to the device.
- Calls the CUDA kernel, which computes the square of the difference of the components for each dimension, reduce all the elements of the output array in parallel (you may need to investigate how to implement an efficient parallel reduce in CUDA), and takes the square root of the sum.

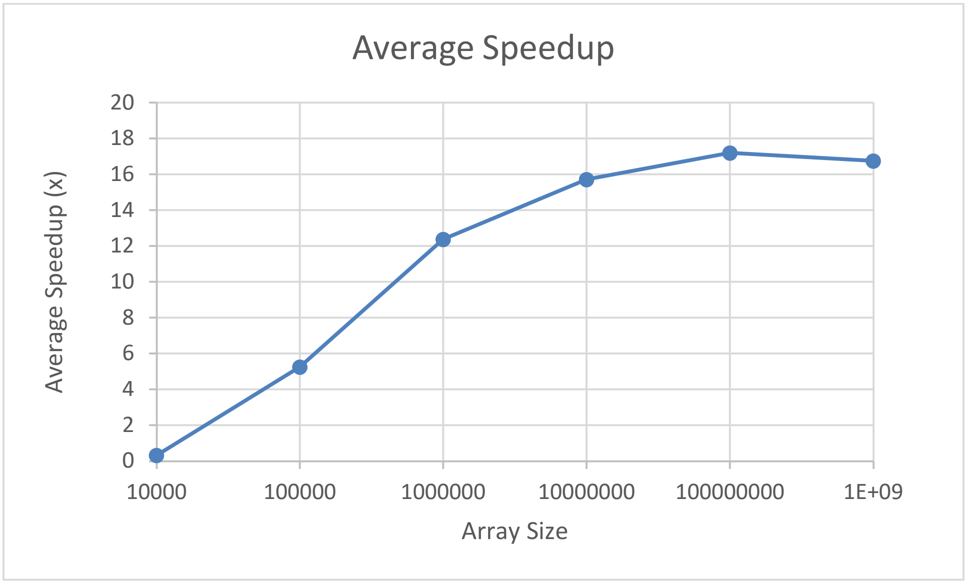
  Run experiments using varying size of inputs. Graph and discuss the speedup provided by the GPU/CUDA implementation over varying input sizes. Is there an input size where the speedup stops?

<u>Source Code:</u>

```
/*
 * This program compares the performance of serial (CPU) and parallel (GPU/CUDA)
 * implementations of Euclidean distance calculation between two N-dimensional points.
 * It runs multiple tests with different array sizes and averages the results.
 */

// Include standard C libraries and the CUDA runtime library
#include <stdio.h>        // For input/output operations
#include <stdlib.h>       // For memory allocation and random number generation
#include <math.h>         // For mathematical functions like pow() and sqrt()
#include <time.h>         // For time functions and clock measurements
#include <cuda_runtime.h> // For CUDA functions and types

#define NUM_RUNS 10 // Number of times to run each test for averaging

// Define global variables
const int sizes[] = {10000, 100000, 1000000, 10000000, 100000000, 1000000000}; // Array of different sizes
to test
int numSizes = sizeof(sizes) / sizeof(sizes[0]);                               // Length of sizes array

double *hostA; // First point coordinates
double *hostB; // Second point coordinates

/**
 * @brief Initialize arrays with random numbers between 0 and 99
 *
 * @param size The size of the arrays to initialize.
 */
void initArrays(int size)
{
    for (int i = 0; i < size; ++i)
    {
        hostA[i] = (double)(rand() % 100);
        hostB[i] = (double)(rand() % 100);
    }
}

/**
 * @brief Calculates the Euclidean distance using serial computation.
 *
 * This function computes the Euclidean distance between two arrays A and B using
 * serial computation.
 *
 * @param size The size of the input arrays.
 * @return The Euclidean distance between the input arrays.
 */
double serialEuclideanDistance(int size)
{
    double sum = 0.0;
    for (int i = 0; i < size; ++i)
    {
        sum += pow(hostA[i] - hostB[i], 2); // Calculate squared difference
    }
    return sqrt(sum); // Return square root of sum
}

/**
 * @brief CUDA kernel for partial sum calculation.
 *
 * This kernel calculates the squared differences between corresponding elements of
 * input arrays A and B and stores them in the partialSums array. Each
 * element in the partialSums array is the sum of all computations performed by
 * threads in a single block, where the element index corresponds to the block index.
 * Adding all these partial sum elements after will provide the total sum of squared differences.
 *
 * @param A Pointer to the first input array on the device.
 * @param B Pointer to the second input array on the device.
 * @param partialSums Pointer to the output array on the device.
 * @param size The size of the input arrays.
 */
__global__ void partialSumKernel(double *A, double *B, double *partialSums, int size)
{

    // Store intermediate results within each block
    extern __shared__ double sharedData[];

    // Thread index within the block
    unsigned int tid = threadIdx.x;

    // Global index of the thread
    unsigned int i = tid + blockDim.x * blockIdx.x;

    // Each thread calculates the squared difference for its assigned element and stores it in shared
memory.
```

```
    // If i exceeds the array size, we store 0 to avoid going out of bounds.
    sharedData[tid] = (i < size) ? pow(A[i] - B[i], 2) : 0;

    // Ensure all threads in a block have completed their writes to shared memory before proceeding
    __syncthreads();

    // This loop performs the reduction, it halves the number of threads with each iteration
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {

        // Threads with indices less than s add the value from the corresponding partner thread at tid + s
        if (tid < s)
        {
            sharedData[tid] += sharedData[tid + s];
        }
        // Makes sure all threads complete this iteration before moving to the next, needed to prevent
element sums from being skipped or race conditions
        __syncthreads();
    }

    // Once the reduction is complete, the first thread in each block writes the result of the reduction
to the partialSums array.
    // This is the sum of the squared differences for this block
    if (tid == 0)
    {
        partialSums[blockIdx.x] = sharedData[0];
    }
}

/**
 * @brief Cleans up allocated memory.
 *
 * This function frees the memory allocated for hostA and hostB arrays.
 */
void cleanup()
{
    free(hostA);
    free(hostB);
}

int main()
{

    // Print the number of CUDA-enabled hardware devices
    int numDevices;
    cudaGetDeviceCount(&numDevices);
    printf("Number of CUDA-enabled devices: %d\n", numDevices);

    // Print properties of Device 0
    cudaDeviceProp props;
    cudaGetDeviceProperties(&props, 0);
    printf("Device Name: %s\n", props.name);
    printf("Max Threads per Block: %d\n", props.maxThreadsPerBlock);
    printf("Max Grid Size: %d x %d x %d\n\n", props.maxGridSize[0], props.maxGridSize[1],
props.maxGridSize[2]);

    // Seed the random number generator once at the beginning
    srand(time(NULL));

    // Find the maximum size in the sizes array
    int maxSize = 0;
    for (int i = 0; i < numSizes; i++)
    {
        if (sizes[i] > maxSize)
        {
            maxSize = sizes[i];
        }
    }

    // Allocate memory for the arrays
    hostA = (double *)malloc(sizeof(double) * maxSize);
    hostB = (double *)malloc(sizeof(double) * maxSize);

    // Loop through different array sizes
    for (int i = 0; i < numSizes; i++)
    {
        int size = sizes[i];

        // Arrays to store timing and distance results for each run
        double serialTimes[NUM_RUNS];
        double cudaTimes[NUM_RUNS];
        double serialDistances[NUM_RUNS];
        double cudaDistances[NUM_RUNS];

        // Perform multiple runs for each size for averaging
        for (int run = 0; run < NUM_RUNS; run++)
        {
            // Initialize arrays with random numbers for run
            initArrays(size);
            // Serial (CPU) version
            clock_t start = clock();
            double serialDistance = serialEuclideanDistance(size);
            clock_t end = clock();
            serialTimes[run] = (double)(end - start) / CLOCKS_PER_SEC * 1000; // Convert to milliseconds
            serialDistances[run] = serialDistance;

            // CUDA (GPU) version
            double *d_A, *d_B, *d_partialSums; // Device arrays

            // Allocate memory on GPU
            cudaMalloc(&d_A, sizeof(double) * size);
            cudaMalloc(&d_B, sizeof(double) * size);
            cudaMalloc(&d_partialSums, sizeof(double) * size);

            // Copy data from host to device
            cudaMemcpy(d_A, hostA, sizeof(double) * size, cudaMemcpyHostToDevice);
            cudaMemcpy(d_B, hostB, sizeof(double) * size, cudaMemcpyHostToDevice);

            // Set block to max threads per block, a multiple of the fixed size of warps
            int blockSize = props.maxThreadsPerBlock;

            // Calculate to ensure all elements are processed even if size is not a multiple of blockSize
            int numBlocks = (size + blockSize - 1) / blockSize;

            // Specify the amount of shared memory to allocate per block
            size_t sharedMemSize = blockSize * sizeof(double);

            // Launch kernel and measure time, calculating partial sums of squared differences for each
block
            start = clock();
            partialSumKernel<<<numBlocks, blockSize, sharedMemSize>>>(d_A, d_B, d_partialSums, size);

            // Make sure device computation has finished for all blocks
            cudaDeviceSynchronize();

            // Copy results back to host and calculate final distance
            double *partialSums = (double *)malloc(sizeof(double) * numBlocks);
            cudaMemcpy(partialSums, d_partialSums, sizeof(double) * numBlocks, cudaMemcpyDeviceToHost);

            // Sum up partial results of each block
            double cudaSum = 0.0;
            for (int j = 0; j < numBlocks; ++j)
            {
                cudaSum += partialSums[j];
            }

            // Calculate the final Euclidean distance by taking the square root of the sum of the squared
differences
            double cudaDistance = sqrt(cudaSum);

            // Record timing and distance
            end = clock();
            cudaTimes[run] = (double)(end - start) / CLOCKS_PER_SEC * 1000; // Convert to milliseconds
            cudaDistances[run] = cudaDistance;

            // Free GPU memory
            cudaFree(d_A);
            cudaFree(d_B);
            cudaFree(d_partialSums);
            free(partialSums);
        }

        // Calculate averages for array size
        double avgSerialTime = 0.0;
        double avgCudaTime = 0.0;
        double avgSerialDistance = 0.0;
        double avgCudaDistance = 0.0;

        for (int run = 0; run < NUM_RUNS; run++)
        {
            avgSerialTime += serialTimes[run];
            avgCudaTime += cudaTimes[run];
            avgSerialDistance += serialDistances[run];
            avgCudaDistance += cudaDistances[run];
        }

        avgSerialTime /= NUM_RUNS;
        avgCudaTime /= NUM_RUNS;
        avgSerialDistance /= NUM_RUNS;
        avgCudaDistance /= NUM_RUNS;

        // Calculate average speedup
        double averageSpeedup = avgSerialTime / avgCudaTime;

        // Print average results for array size
        printf("Size: %d\n", size);
        printf("Average Serial Time: %.2f ms\n", avgSerialTime);
        printf("Average CUDA Time: %.2f ms\n", avgCudaTime);
        printf("Average Speedup: %.2fx\n", averageSpeedup);
        printf("Average Serial Distance: %.4f\n", avgSerialDistance);
        printf("Average CUDA Distance: %.4f\n", avgCudaDistance);

        // Print individual run results Note: Only used for testing distance results
        /**printf("\nIndividual Run Results:\n");
        for (int run = 0; run < NUM_RUNS; run++)
        {
            printf("Run %d:\n", run + 1);
            printf("Serial Time: %.2f ms\n", serialTimes[run]);
            printf("CUDA Time: %.2f ms\n", cudaTimes[run]);
            printf("Serial Distance: %.4f\n", serialDistances[run]);
            printf("CUDA Distance: %.4f\n", cudaDistances[run]);
            printf("\n");
        }**/
        printf("\n");
    }

    // Clean up and exit
    cleanup();
    return 0;
}
```

Analysis:



Starting at an array size of 10,000, CUDA is slower than serial execution (about 0.33 times as fast). This tells us the problem size of 10^4 does not justify the overhead of parallelization (launching kernels, moving data between devices, and synchronizing threads, etc). However, if we move up to 100,000, we see CUDA jump up to about 5.25 times as fast as serial execution. We see significant gains all the way up to 10^7 at 15.71 times speedup where it then starts to level off and the speedup starts to become consistent, only growing slightly to 17.19 times speedup at 10^8, then dropping down to 16.74 times speedup at 10^9. This is likely because eventually the maximum number of simultaneous threads that can be executed at once are being executed around the 10^7 mark, and the GPU is being fully utilized, thus increasing the problem size will not provide any more significant speedup aside from random fluctuations and chance. It appears for this GPU, NVIDIA RTX 4090, 10^7 is an optimal minimum problem size to take advantage of the maximum speedup provided by CUDA parallelization.

3. **(10 points)** You are working on team that is building an edge detection module. The first step in edge detection is to remove color information and work directly in black-and-white. You do not need to do edge detection, just the first step: removing the color information from images. So, please write **a CUDA program** to convert a color image to grayscale using the Colorimetric method. I suggest that you use PNG or BMP images as input. I want to see the original images that you tested with and the output/resulting image. You may do this assignment in C or Python, and are free to create a Jupyter notebook. If you do a Jupyter notebook, please include the notebook in your assignment report. How does using a GPU for this vs. a serial CPU-based implementation perform? Is there a speedup? How (be specific) is the GPU architecture well-suited to this task?

Source Code:

```
from pycuda.compiler import SourceModule # Provide source module class for compiling C code
import pycuda.autoinit # Automatically initialize the CUDA driver and context
import pycuda.driver as cuda # Provide CUDA driver functions for memory management and kernel launches
import numpy as np # Used for array manipulation and image processing
import cv2 # OpenCV library for image I/O and processing
import time # Used for measuring execution time


mod = SourceModule("""
    /**
    * @brief Converts an RGB image to grayscale using the standard luminance formula.
    *
    * This kernel calculates the intensity of each pixel in the new grayscale image
    * based on the RGB values of that pixel in the original image. It uses the ITU-R BT.601 standard
    * coefficients for luma calculation: 0.299 for red, 0.587 for green, and 0.114 for blue.
    *
    * Unsigned char an 8-bit unsigned integer, ranging 0 from to 255, making it suitable for storing
    * pixel values.
    *
    * @param rgb Pointer to the input RGB image data (flattened 3D array)
    * @param gray Pointer to the output grayscale image data (flattened 2D array)
    * @param width Width of the input image
    * @param height Height of the input image
    */
    __global__ void rgb_to_gray(unsigned char *rgb, unsigned char *gray, int width, int height)
    {
        // Get the column of the element the thread will be working on
        int col = threadIdx.x + blockIdx.x * blockDim.x;

        // Get the row of the element the thread will be working on
        int row = threadIdx.y + blockIdx.y * blockDim.y;

        // If the element x and y indices are valid, calculate the gray pixel from the original rgb values
        if (col < width && row < height) {

            // Get the pixel this thread is calculating for the grayscale output
            // gray is a flattened 2D array with width * height indexes
            int grayOffset = row * width + col;

            // Get the first RGB channel for this pixel in the original image by multiplying by 3 since
each pixel has 3 channels
            // rgb is a flattened 3D array with width * height * 3 indexes
            int rgbOffset = grayOffset * 3;

            // Get the red intensity value the new gray pixel had in the original image
            unsigned char r = rgb[rgbOffset + 0];

            // Get the green intensity value the new gray pixel had in the original image
            unsigned char g = rgb[rgbOffset + 1];

            // Get the blue intensity value the new gray pixel had in the original image
            unsigned char b = rgb[rgbOffset + 2];

            // We calculate the grayscale intensity using the luminosity  method, as it is the most
accurate.
            // It uses a weighted average of the red, green, and blue intensities to calculate the new
gray value.
            // We add 0.5f to convert the value to the nearest intensity before converting the float to an
integer.
            gray[grayOffset] = (unsigned char)(0.299f*r + 0.587f*g + 0.114f*b + 0.5f);
        }
    }
""")
```

```
# Read input image
img = cv2.imread('input.png')

# CPU Code:
# Start measuring CPU time
start_cpu = time.time()

# Convert image to grayscale serially using cv2 library
gray_img_cpu = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Stop measuring CPU time
end_cpu = time.time()

# Get CPU time taken in milliseconds and print it
cpu_time = (end_cpu - start_cpu) * 1000
print(f"CPU execution time: {cpu_time:.2f} ms")

# GPU Code:
# Extract width and height of the image, ignoring the 3 RGB channels
height, width, _ = img.shape

# Create empty Numpy array to store output grayscale image
gray_img = np.empty((height, width), dtype=np.uint8)

# Allocate GPU memory for the input RGB image
rgb_img_gpu = cuda.mem_alloc(img.nbytes)

# Copy the image data from host to device memory
cuda.memcpy_htod(rgb_img_gpu, img)

# AAllocate GPU memory for the output grayscale image
gray_img_gpu = cuda.mem_alloc(gray_img.nbytes)

# Retrieve a reference to the rgb_to_gray kernel function from the compiled CUDA module
rgb_to_gray = mod.get_function("rgb_to_gray")

# The block size is set to (32, 32, 1) meaning each block has 32 x 32 threads
block_dim = (32, 32, 1)

# The grid size is calculated based on the image dimensions and block size
grid_dim_x = (width + block_dim[0] - 1) // block_dim[0] # Choose number of blocks in the x dimension to
ensure there are enough threads in the x dimension (blocks.x * blockdim.x) to cover the full width for one
row.
grid_dim_y = (height + block_dim[1] - 1) // block_dim[1] # Choose number of blocks in the y dimension to
```

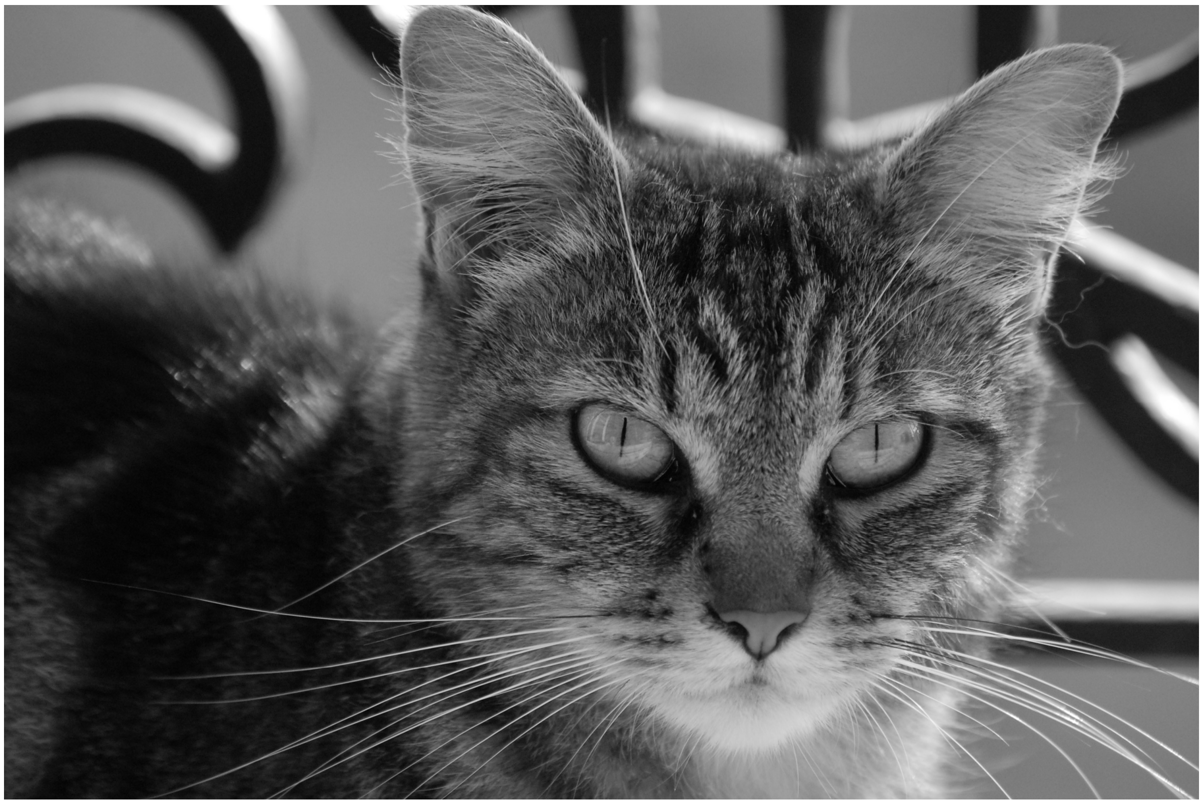
```
ensure there are enough threads in the y dimension (blocks.y * blockdim.y) to cover the full height for
one column.
grid_dim = (grid_dim_x, grid_dim_y, 1) # Multiplying the grid_dim_x times grid_dim_y for the number of
blocks ensures each row and column is fully covered.

# Start measuring GPU time
start_gpu = time.time()

# Call the GPU kernel to get the grayscale image
rgb_to_gray(rgb_img_gpu, gray_img_gpu, np.int32(width), np.int32(height), block=block_dim, grid=grid_dim)

# Copy result from device memory to host memory
cuda.memcpy_dtoh(gray_img, gray_img_gpu)

# Stop measuring GPU time
end_gpu = time.time()

# Get time taken for GPU in ms
gpu_time = (end_gpu - start_gpu) * 1000

# Print GPU time and speedup over CPU
print(f"GPU execution time: {gpu_time:.2f} ms")
speedup = cpu_time / gpu_time
print(f"Speedup: {speedup:.2f}x")

# Save output
cv2.imwrite('output_cpu.png', gray_img_cpu)
cv2.imwrite('output_gpu.png', gray_img)
```

Analysis:

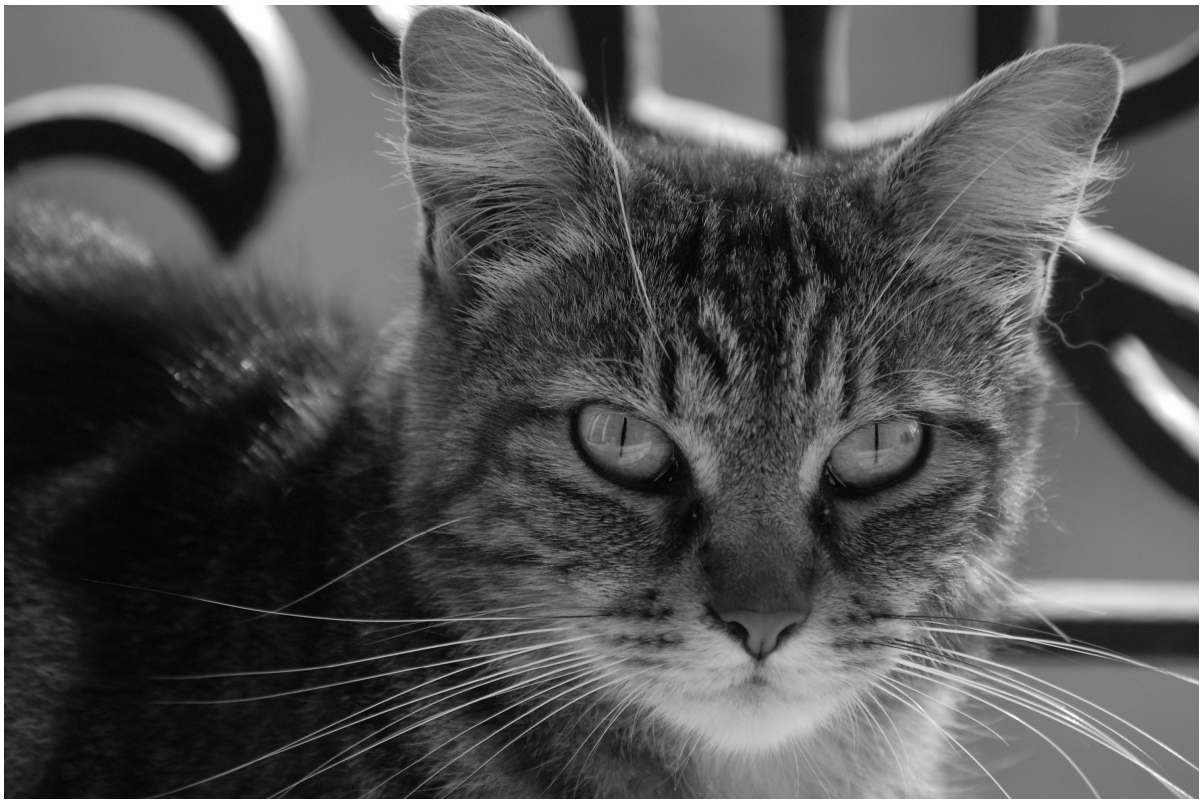
The GPU implementation shows speedup over the CPU implementation ranging from 1.33x to 2.50x for the image I tested, although the speedup may not be too impressive for small pictures due to them not fully utilizing GPU parallelism and overhead. Overhead in this case would be launching the kernel and memory transfer between devices. This problem is well suited for parallelism because each pixel conversion is independent, and the conversion formula is simple arithmetic. Each thread processes one pixel, while blocks of 32 * 32 threads work in parallel. The total number of blocks is equal to your grid x dimension * grid y dimension however, the actual number of threads that can be run in parallel is limited to your GPU hardware. The RGB values are contiguous in memory and accessed consecutively by warps, allowing for coalescence, combining thread memory access into a single transaction, providing improved performance. The above-mentioned factors make the GPU architecture well suited to this task. I have attached an example input to my program, as well as two example outputs for the serial and parallel implementations.

Before Conversion:

After Conversion (CPU - OpenCV library)



After Conversion (GPU):



4. **(11 points)** Write a matrix multiplication program that uses GPU/CUDA. Your matrices should consist of floating-point values (NOT INTEGERS). You may do this assignment in C, Python, and are free to create a Jupyter notebook. If you do a Jupyter notebook, please include the notebook in your assignment report. You should run MANY experiments with varying sized matrices. What is the speedup? Is the speedup affected by matrix size? Explain (diagrams are useful here) how the GPU implementation of this code differs from a serial (CPU-based) implementation.

Source Code:

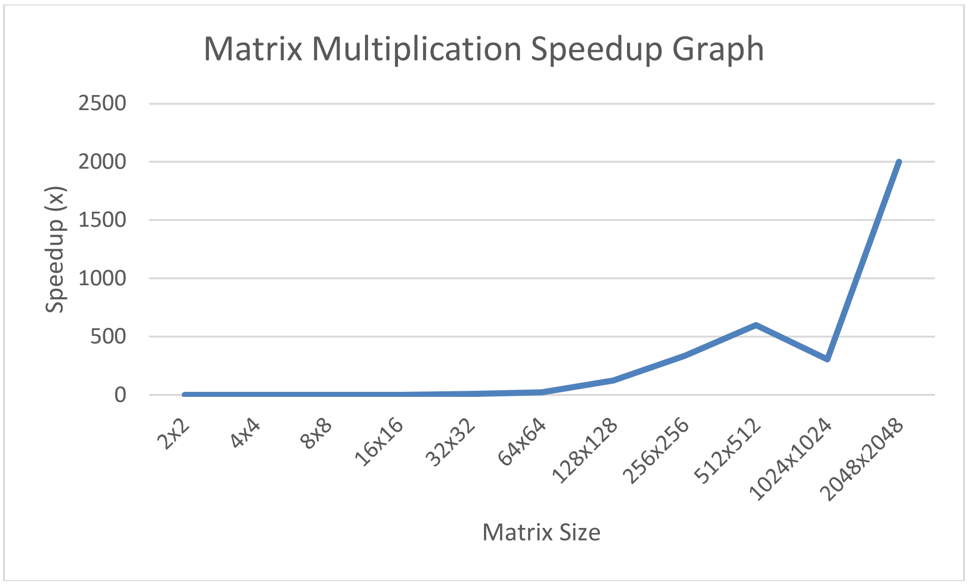
```c
#include <stdio.h>        // Standard Input / Output functions (printf)
#include <stdlib.h>       // General-purpose standard library functions (random number generation, malloc(), etc)
#include <cuda_runtime.h> // Cuda Runtime API needed for CUDA functionality
#include <time.h>         // Time-related functions, used for seeding random number generator

#define NUM_RUNS 10 // Define a constant for the number of times each matrix size will be multiplied to get an average time.

// Define an array of integers containing the sizes of matrices to be tested. These represent square matrices 256x256, 512x512, etc.
const int matrixSizes[] = {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048};

// Calculate the number of elements in the matrixSizes array dynamically.
const int NUM_SIZES = sizeof(matrixSizes) / sizeof(matrixSizes[0]);

/**
 * @brief Initializes a size * size matrix with random float values between 0 and 1.
 *
 * @param matrix Pointer to the matrix to be initialized.
 * @param size Row and column length of the square matrix.
 */
void initializeMatrix(float *matrix, int size)
{
    // The loop iterates size * size times, covering all elements of the matrix.
    for (int i = 0; i < size * size; i++)
    {
        // For each element, a random float value is generated using rand().
        // We scale it between 0 and 1 by dividing by RAND_MAX.
        matrix[i] = (float)rand() / RAND_MAX;
    }
}
```

```
/**
 * @brief Performs matrix multiplication on the CPU.
 *
 * @param A Pointer to the first input matrix.
 * @param B Pointer to the second input matrix.
 * @param C Pointer to the output matrix.
 * @param size Row and column length of the square matrices.
 */
void cpuMatrixMultiply(float *A, float *B, float *C, int size)
{
    // Iterate through each row of the result matrix
    for (int i = 0; i < size; i++)
    {
        // For each row in result matrix, iterate through each column
        for (int j = 0; j < size; j++)
        {

            // Dot product for current element will be accumulated here
            float sum = 0.0f;
            // For each element (i, j) in the result matrix calculate it's value as the dot product
            // of the ith row in matrix A and the jth column in matrix B.
            // This is done by multiplying each corresponding kth element in the ith row and jth column
            // and adding their product to a sum.
            for (int k = 0; k < size; k++)
            {
                // i * size gets us to the ith row, then adding k gets us to the current element in matrix
A being multiplied in the dot product.
                // Size * k gets us to the current row of the element being multiplied in the dot product
in matrix B, then adding j ensures we are moving through the jth column's rows.
                sum += A[i * size + k] * B[k * size + j];
            }
            // Once each corresponding element in the ith row and jth column has been multiplied and added
to the sum, store it at (i, j) in the result matrix.
            // i * size gets the first column in the ith row, then adding j gets us to the jth column in
that row.
            C[i * size + j] = sum;
        }
    }
}

/**
 * @brief CUDA kernel function to perform matrix multiplication on the GPU.
 *
 * Each thread calculates one element of the result matrix that has a value of the dot product
 * of it's corresponding row and column in the A and B matrices respectively.
 *
 * @param A Pointer to the first input matrix on the device.
 * @param B Pointer to the second input matrix on the device.
 * @param C Pointer to the output matrix on the device.
 * @param size Row and column length of the square matrices.
 */
__global__ void gpuMatrixMultiply(float *A, float *B, float *C, int size)
{

    // Get the row of the result element this thread will be working on
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // Get the column of the result element this thread will be working on
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure this thread's global index corresponding to a valid element index in the result matrix
    // This is because the final block may have extra indexes if the problem size isn't evenly divisible
by the block size
    if (row < size && col < size)
    {
        // Keep track of the sum for the dot product
        float sum = 0.0f;

        // Go through each corresponding kth element in the current row in the A matrix and current column
in the B matrix, multiply them, and accumulate their products in a sum.
        for (int k = 0; k < size; k++)
        {
            sum += A[row * size + k] * B[k * size + col];
        }

        // Store the sum in the appropriate location in the result matrix.
        // The element's row and column matches the row in matrix A, and column in matrix B respectively
whose dot product was calculated.
        C[row * size + col] = sum;
    }
}

/**
 * @brief Prints a square matrix to the console.
 *
 * This function takes a pointer to a 1D array representing a square matrix
 * and its size, and prints it in a 2D format to the console.
 *
 * @param matrix Pointer to the 1D array representing the square matrix.
 * @param size The number of rows (and columns) in the square matrix.
```

```c
 */
void printMatrix(float *matrix, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%.2f ", matrix[i * size + j]);
        }
        printf("\n");
    }
    printf("\n");
}

/**
 * @brief Compares two matrices element-wise.
 *
 * This function compares two matrices of the same size element-wise and returns
 * true if all elements are within a specified tolerance, false otherwise.
 *
 * @param A Pointer to the first matrix.
 * @param B Pointer to the second matrix.
 * @param size The size of the square matrices.
 * @param tolerance The maximum allowed difference between elements.
 * @return true if the matrices match within the tolerance, false otherwise.
 */
bool compareMatrices(float *A, float *B, int size, float tolerance)
{
    for (int i = 0; i < size * size; i++)
    {
        if (fabs(A[i] - B[i]) > tolerance)
        {
            return false;
        }
    }
    return true;
}

/**
 * @brief Main function that orchestrates the matrix multiplication benchmark.
 *
 * @param argc Number of command-line arguments (not used in this program).
 * @param argv Array of command-line argument strings (not used in this program).
 * @return 0 on successful execution.
 */
```

```
int main(int argc, char **argv)
{
    // Called to seed the random number generator with the current time, ensuring different random
sequences each time the program is run.
    srand(time(NULL));

    // Declare the prop variable that will hold our CUDA device properties
    cudaDeviceProp prop;

    // Retrieve the properties of the CUDA device with index 0, and store them in the prop variable called
prop
    cudaGetDeviceProperties(&prop, 0);

    // Get the maximum number of threads per block based on device
    int maxThreadsPerBlock = prop.maxThreadsPerBlock;

    // The maximum block size for the x and y dimensions is calculated as the square root of the maximum
threads per block.
    int maxBlockSizeX = sqrt(maxThreadsPerBlock);
    int maxBlockSizeY = maxBlockSizeX;

    // Print maximum threads per block and the chosen block size for the GPU kernel
    printf("Max threads per block: %d\n", maxThreadsPerBlock);
    printf("Using block size: %dx%d\n", maxBlockSizeX, maxBlockSizeY);

    // Declare variables for the block size and grid size using the CUDA dim3 type.
    // The block sizes for the x and y dimensions are set to the values determined earlier.
    dim3 blockSize(maxBlockSizeX, maxBlockSizeY);
    dim3 gridSize;

    // Declare pointers for the matrices on the host.
    float *h_A, *h_B, *h_C;

    // Declare pointer for CPU result, h_C will store GPU result on host
    float *h_C_cpu;

    // Declare pointers for matrices on the device.
    float *d_A, *d_B, *d_C;

    // Declare event variables and timing variables for measuring GPU execution time
    cudaEvent_t start, stop;
    float gpuTime, cpuTime;

    // Start of a loop that iterates over each matrix size in the matrixSizes array.
    for (int i = 0; i < NUM_SIZES; i++)
    {
        // Current matrix size is stored in the size variable
        int size = matrixSizes[i];

        // The grid size in each dimension is calculated by dividing
        // the matrix size (width and height are the same) by the
        // block size of that dimension and rounding up to the nearest integer.
        gridSize.x = (size + blockSize.x - 1) / blockSize.x;
        gridSize.y = (size + blockSize.y - 1) / blockSize.y;

        // Allocate memory on the host for input matrices h_A and h_B.
        h_A = (float *)malloc(size * size * sizeof(float));
        h_B = (float *)malloc(size * size * sizeof(float));

        // Allocate memory on the host for the device execution output matrix h_C. (Device output will be
moved here)
        h_C = (float *)malloc(size * size * sizeof(float));

        // Allocate memory on the host for the host execution output matrix h_C_cpu
        h_C_cpu = (float *)malloc(size * size * sizeof(float));

        // Allocate memory on the CUDA device for the input matrices d_A and D_B using cudaMalloc.
        cudaMalloc(&d_A, size * size * sizeof(float));
        cudaMalloc(&d_B, size * size * sizeof(float));

        // Allocate memory on the CUDA device for the output matrix d_C.
        cudaMalloc(&d_C, size * size * sizeof(float));

        // Initialize the input matrices h_A and h_B with random values using the initializeMatrix
function.
        initializeMatrix(h_A, size);
        initializeMatrix(h_B, size);

        // Copy the input matrices from the host memory to the device memory using cudaMemcpy with the
cudaMemcpyHostToDevice flag.
        cudaMemcpy(d_A, h_A, size * size * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size * size * sizeof(float), cudaMemcpyHostToDevice);

        // CPU computation
        cpuTime = 0.0f;

        // Matrix multiplication for this matrix size is performed 10 times on CPU.
        for (int j = 0; j < NUM_RUNS; j++)
        {
            // Record current clock time before calling matrix multiply function.
            clock_t begin = clock();

            // Multiply matrices h_A and h_B serially and store the product in h_C_cpu
            cpuMatrixMultiply(h_A, h_B, h_C_cpu, size);

            // Record the current clock time after function completes
            clock_t end = clock();

            // Add elapsed time to the cpuTime variable to get total time taken so far across all runs.
            cpuTime += (float)(end - begin) / CLOCKS_PER_SEC; // Calculate difference between the end and
start times in clock ticks and divide by clock ticks per second to convert to seconds.
        }

        // Get the average CPU execution time for this matrix size by dividing the total time taken by the
number of runs.
        cpuTime /= NUM_RUNS;
        cpuTime = cpuTime * 1000; // Convert to miliseconds

        // Print matrices for small size to verify results
        if (size == 2)
        {
            printf("Matrix A:\n");
            printMatrix(h_A, size);
            printf("Matrix B:\n");
            printMatrix(h_B, size);
            printf("CPU Result:\n");
            printMatrix(h_C_cpu, size);
        }

        // GPU computation
        gpuTime = 0.0f;
        // Matrix multiplication for this matrix size is performed 10 times on GPU.
        for (int j = 0; j < NUM_RUNS; j++)
        {
            // Create CUDA start and stop event objects using cudaEventCreate and have the start and stop
variables declared earlier store the references to them.
            cudaEventCreate(&start);
            cudaEventCreate(&stop);

            // Record the start event using cudaEventRecord to mark the beginning of the GPU kernel
execution.
            // The second argument specifies the CUDA stream to associate with the event, 0 represents the
default stream.
            cudaEventRecord(start, 0);
            // Launch the gpuMatrixMultiply kernel with the specified grid and block size, with the device
matrices and matrix size as arguments.
            // The kernel performs the matrix multiplication on the GPU
            gpuMatrixMultiply<<<gridSize, blockSize>>>(d_A, d_B, d_C, size);

            // The stop event is recorded using cudaEventRecord to mark the end of the GPU kernel
execution
            // 0 specifies to associate the default CUDA stream with the event.
            cudaEventRecord(stop, 0);

            // Call to ensure the stop event has completed.
            // Synchronize the host (CPU) with the stop event using the cudaEventSynchronize function.
            // Ensures the host has waited until the stop event has been recorded, and that all GPU
operations have been completed.
            cudaEventSynchronize(stop);

            // Declare variable of type float to store elapsed time for this run
            float tempTime;

            // Calculate the elapsed time between the start and stop events using cudaEventElapsedTime and
store it in tempTime.
            cudaEventElapsedTime(&tempTime, start, stop); // Time is returned in milliseconds

            // Add this run time to the total gpuTime across all iterations
            gpuTime += tempTime;

            // Destroy the CUDA event objects using cudaEventDestroy and free up resources
            cudaEventDestroy(start);
            cudaEventDestroy(stop);
        }

        // Get average GPU execution time for this matrix size
        gpuTime /= NUM_RUNS; // No need to convert to milliseconds, cudaEventElapsed time gives ms
already.

        // Move GPU product matrix to host
        cudaMemcpy(h_C, d_C, size * size * sizeof(float), cudaMemcpyDeviceToHost);

        // Print matrices for small size to verify results
        if (size == 2)
        {
            printf("GPU Result:\n");
            printMatrix(h_C, size);
        }
        // Compare CPU and GPU results for sizes <= 1024
        if (size <= 1024)
        {
            bool match = compareMatrices(h_C, h_C_cpu, size, 1e-4);
            printf("CPU and GPU results for %dx%d matrix %s\n", size, size, match ? "match" : "do not match");
        }
        else
        {
            printf("CPU and GPU results for %dx%d matrix not compared (size > 1024) due to performance reasons and potential floating-point precision limitations\n", size, size);
        }

        // Print the matrix size, average CPU time, average GPU time, and the speedup achieved by the GPU over the CPU for the current matrix size.
        printf("Matrix size: %dx%d\n", size, size);
        printf("Average CPU time: %.2f ms\n", cpuTime);
        printf("Average GPU time: %.2f ms\n", gpuTime);
        printf("Average Speedup: %.2fx\n", cpuTime / gpuTime);
        printf("\n");

        // Free the allocated memory on the host using free
        free(h_A);
        free(h_B);
        free(h_C);
        free(h_C_cpu);

        // Free the allocated memory on the device using cudaFree
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
    }

    // Return 0 to indicate successful execution
    return 0;
}
```

Analysis:



Looking at the above graph showcasing my speedup results, we can see that I tested the speed of my CPU and GPU implementations for matrix multiplication with several powers of 2 from 2^1 to 2^11. 2^12 was chosen as the cutoff point because it simply took long to execute to be feasible for this experiment. Up until but not including matrix sizes 2^5, GPU implementation is slower due to kernel launching overhead. For a 32x32 matrix sizes, the speedup jumps to 6.73x. Next we see a relatively moderate increase to 21.59x speedup in a 64x64 matrix. After, the speedup skyrockets to 123.41x for a 256x256 matrix. The pattern increases up to and including 512x12 matrices where it reaches 598.91x speedup, then it drops at 1024x1024 to 305.23x speedup. When we move up to 2048x2048 matrices, the speedup increases again to a staggering 2002.61x speedup! It appears that this pattern will only continue as we move to larger and larger matrix sizes. This shows that the speedup is absolutely affected by matrix size. Both the GPU and CPU implementation are similar in that they calculate the value of each element in the result matrix as the dot product of that element's corresponding row and column in the A and B input matrices respectively, the key difference is that the CPU implementation calculates each element one by one in row major order while the GPU implementation calculates each element in parallel where each thread is responsible for a specific element. Let's look at a simple example of how the CPU execution works with a simple 2x2 matrix then move to a more complicated example with a 4x4 matrix using the GPU implementation to show how we use global thread indexing with multiple grid dimensions and multiple block dimensions to have each thread calculate an element in the resulting 4x4 matrix. Take a look at the 2x2 matrix below.

Matrix A:
| A[0,0] A[0,1] |
| A[1,0] A[1,1] |
Matrix B:
| B[0,0] B[0,1] |
| B[1,0] B[1,1] |
Result Matrix C (calculated serially):
| C[0,0] C[0,1] |
| C[1,0] C[1,1] |

CPU Algorithm Steps:

The CPU algorithm will calculate result matrix C in the following order, one by one.

CPU Algorithm Steps:

1. Calculate C[0,0] = A[0,0] * B[0,0] + A[0,1] * B[1,0]
2. Calculate C[0,1] = A[0,0] * B[0,1] + A[0,1] * B[1,1]
3. Calculate C[1,0] = A[1,0] * B[0,0] + A[1,1] * B[1,0]
4. Calculate C[1,1] = A[1,0] * B[0,1] + A[1,1] * B[1,1]

This is done using a triple nested for loop, the outer two, ensure we traverse through element in the result matrix, and the inner ensures we multiply each corresponding element in that element's corresponding row and column in the A and B matrices respectively, add their products, and store the result in the correct location in the result matrix. This has a time complexity of O(N^3) and can be very time consuming on the CPU when the matrices get large, although it not too bad when we are dealing with small matrices like this.

Now, let's look at the GPU algorithm for a 4x4 matrix

```
| A[0,0] A[0,1] A[0,2] A[0,3] |
| A[1,0] A[1,1] A[1,2] A[1,3] |
| A[2,0] A[2,1] A[2,2] A[2,3] |
| A[3,0] A[3,1] A[3,2] A[3,3] |
Matrix B:
| B[0,0] B[0,1] B[0,2] B[0,3] |
| B[1,0] B[1,1] B[1,2] B[1,3] |
| B[2,0] B[2,1] B[2,2] B[2,3] |
| B[3,0] B[3,1] B[3,2] B[3,3] |
Result Matrix C:
| C[0,0] C[0,1] C[0,2] C[0,3] |
| C[1,0] C[1,1] C[1,2] C[1,3] |
| C[2,0] C[2,1] C[2,2] C[2,3] |
| C[3,0] C[3,1] C[3,2] C[3,3] |
```

Thread Indexing in CUDA

We'll use 2x2 blocks for this example, which means we'll have a 2x2 grid of blocks. Each block will contain 2x2 threads. Note: This block size is just for simplicity to split a small matrix across multiple blocks to see how indexing the elements with threads works. Our actual block sizes will be much bigger and a matrix of this size (4x4) would be handled by one block. For my actual implementation I used block sizes of 32x32 for a total 1024 threads per block.

We determined the number of blocks in the x dimension by dividing the matrix width (4), and then dividing by the number of threads in the block x dimension (2). 4 / 2 tells us we need 2 blocks in the grid's x dimension.

We determined the number of blocks in the grid y dimension a similar way, by dividing the matrix height (4) by the number of threads in the block y dimension (2). 4 / 2 tells us we need 2 blocks in the grid's y dimension. This tells us we need a 2x2 grid of blocks as you can see below in the table showing how blocks and their threads' corresponding global indices are organized. The global 2D index for each element is calculated with the below 2 formulas.

Here's how the blocks and threads are organized:

```
Block (0,0)    Block (0,1)
[0,0] [0,1]   [0,2] [0,3]
[1,0] [1,1]   [1,2] [1,3]

Block (1,0)    Block (1,1)
[2,0] [2,1]   [2,2] [2,3]
[3,0] [3,1]   [3,2] [3,3]
```

The global 2D index for each element is calculated with the below 2 formulas.

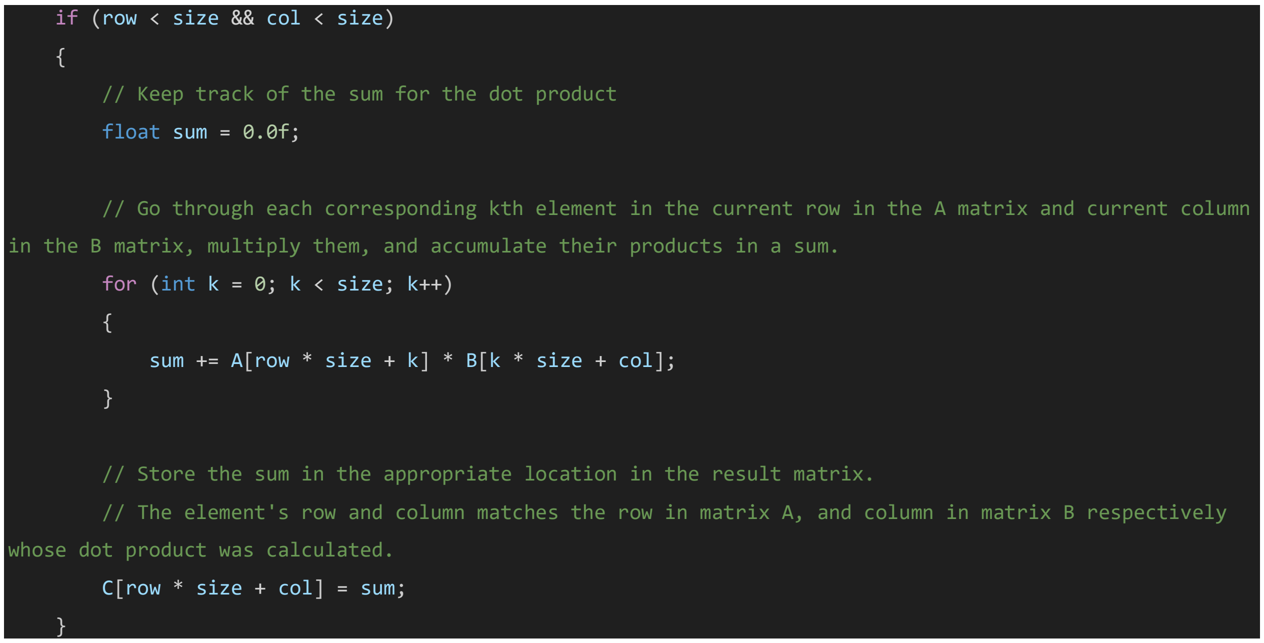
Row = blockIdx.y * blockDim.y + threadIdx.y

Column = blockIdx.x * blockDim.x + threadIdx.x

Now, let's look at how thread indexing works for a few threads to confirm the table is correct:

1. Thread in top-left corner (Block (0,0), Thread (0,0)):
   - blockIdx.x = 0, blockIdx.y = 0
   - threadIdx.x = 0, threadIdx.y = 0
   - row = blockIdx.y * blockDim.y + threadIdx.y = 0 * 2 + 0 = 0
   - col = blockIdx.x * blockDim.x + threadIdx.x = 0 * 2 + 0 = 0 This thread calculates C[0,0]
2. Thread in bottom-right corner of first block (Block (0,0), Thread (1,1)):
   - blockIdx.x = 0, blockIdx.y = 0
   - threadIdx.x = 1, threadIdx.y = 1
   - row = blockIdx.y * blockDim.y + threadIdx.y = 0 * 2 + 1 = 1
   - col = blockIdx.x * blockDim.x + threadIdx.x = 0 * 2 + 1 = 1 This thread calculates C[1,1]
3. Thread in the top-left of the second block (Block (0,1), Thread (0,0)):
   - blockIdx.x = 1, blockIdx.y = 0
   - threadIdx.x = 0, threadIdx.y = 0
   - row = blockIdx.y * blockDim.y + threadIdx.y = 0 * 2 + 0 = 0
   - col = blockIdx.x * blockDim.x + threadIdx.x = 1 * 2 + 0 = 2 This thread calculates C[0,2]
4. Thread in the bottom-right corner of the last block (Block (1,1), Thread (1,1)):
   - blockIdx.x = 1, blockIdx.y = 1
   - threadIdx.x = 1, threadIdx.y = 1
   - row = blockIdx.y * blockDim.y + threadIdx.y = 1 * 2 + 1 = 3
   - col = blockIdx.x * blockDim.x + threadIdx.x = 1 * 2 + 1 = 3 This thread calculates C[3,3]

Sometimes the matrix width will not be evenly divisible by the block x dimension, the same goes for the matrix height and the block y dimension. In this case we round up the quotient to the nearest integer which will increase the x or y dimension of the grid by 1. In my program, the width and height are the same, as well as the x and y dimensions of my blocks, so both would need to be rounded up. This also means that some global thread indexes won't be valid if this happens which is why we have a check in the GPU kernel to make sure the element at resultMatrix[row][column] is a valid element and avoid out of bounds errors before calculating that element's value as the dot product of the corresponding row and column of the A and B input matrices respectively.



```
    if (row < size && col < size)
    {
        // Keep track of the sum for the dot product
        float sum = 0.0f;

        // Go through each corresponding kth element in the current row in the A matrix and current column
in the B matrix, multiply them, and accumulate their products in a sum.
        for (int k = 0; k < size; k++)
        {
            sum += A[row * size + k] * B[k * size + col];
        }

        // Store the sum in the appropriate location in the result matrix.
        // The element's row and column matches the row in matrix A, and column in matrix B respectively
whose dot product was calculated.
        C[row * size + col] = sum;
    }
```

The global thread indexing essentially replaces the two outer loops of the sequential implementation while the inner loop remains the same, calculating a dot product of a row and column in the input matrices. The combination of the formula to determine the number of blocks based on the block dimensions and problem size, and the formulas to assign a thread's row and column based on the block index, block dimension, and thread index ensures that every element in the result matrix has a corresponding thread assigned to calculate it. You can see that when I show how the blocks and threads were organized in the previous table, each thread's two-dimensional global index corresponded to an element in the result matrix. These threads execute in parallel, meaning multiple result elements are being calculated at once where only one was being computed at a time in the sequential execution.

What to submit:

1. Answer all questions and provide any code inline. Do not paste screenshots of code, post your source code, I want to see graphs/visualizations of all of your experimental results results. Crete a PDF and upload the PDF to the assignment specification in canvas.