COP6616 Parallel Computing
Instructor: Scott Piersall
Fall 2024
Assignment 3

README.MD

```
echo "PATH=/usr/local/cuda/bin/:$PATH" >> ~/.bash_profile
source ~/.bash_profile

sudo apt install libpng-dev
OR (NON-SUDO)
wget http://prdownloads.sourceforge.net/libpng/libpng-1.6.44.tar.xz
tar xf libpng-1.6.44.tar.xz
cd libpng-1.6.44
./configure --prefix=$HOME/lib/libpng
make
make install

*q2*
nvcc -lm q2.cu -o q2
./q2 1000000 100

*q3*
nvcc -lm -lpng q3.cu -o q3
OR (NON-SUDO)
nvcc -lm q3.cu -o q3 -L$HOME/lib/libpng/lib -I$HOME/lib/libpng/include -lpng
./q3

*q4*
nvcc -lm q4.cu -o q4
```

./q4 1000 1000 1000 10

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

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

   In spatial locality, items being accessed are close to one another in memory. It's why we retrieve memory in blocks, because items next to each other in memory are likely to be accessed in the same time frame. In temporal locality, we're reusing items in cache multiple times. This leads to paging paradigms like FIFO (first in, first out) or LRU (least recently used) for caching, where we fill the cache by replacing the oldest or least recently used element.

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

   Yes, the root contributes data in MPI_Gather.
   Source: **https://rookiehpc.org/mpi/docs/mpi_gather/index.html**

   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;
}
```

```
/**
 * if (rank = 0) { should be if (rank == 0) {
 * MPI_Barrier(...) should be called by all processes, not just rank 0
 * The code as written will cause a deadlock because rank 0 will be waiting
 * for all other processes to reach the barrier, but the other processes will
 * not reach the barrier because they are not calling MPI_Barrier(...).
 *
 * SOURCE:  https://rookiehpc.org/mpi/docs/mpi_barrier/index.html
```

```
 */

MPI_Barrier(...);
if (rank == 0) {
    // do a thing with rank 0 if you would like
} else {
    // do something with all other ranks
}
```

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

For an example structure:

```
struct Particle {
    double posx, posy, posz;
    double velx, vely, velz;
    double mass;
};
```

Structure of Arrays memory layout would look like:

```
posx = {0.1, 0.2, 0.3, ...}
posy = {0.4, 0.5, 0.6, ...}
posz = {0.7, 0.8, 0.9, ...}
velx = {1.0, 1.1, 1.2, ...}
vely = {1.3, 1.4, 1.5, ...}
velz = {1.6, 1.7, 1.8, ...}
mass = {2.0, 2.1, 2.2, ...}
```

Array of Structures memory layout would look like:

```
particles = {
    {0.1, 0.4, 0.7, 1.0, 1.3, 1.6, 2.0},
    {0.2, 0.5, 0.8, 1.1, 1.4, 1.7, 2.1},
    {0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.2},
    ...
}
```

The Structure of Arrays provides spatial locality. In this example, in order to perform operations on a given variable in the structures, we can either access contiguous memory with SoA or access a double, skip the memory space for six doubles, access the next double, etc. with AoS. SoA also allows for "coalesced memory access" where consecutive threads access consecutive memory addresses.

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

Memory Type	Qualities
Global	Accessible by all GPU threads, largest, high latency (slow)
Local	Private to each GPU thread, same latency as global, temporary storage of register spillover variables
Shared	Accessible by all GPU threads WITHIN THE SAME BLOCK, low latency, **FASTEST MEMORY FOR WITHIN BLOCK ACCESS UNLESS THERE IS A BANK CONFLICT**
Texture	Cached memory optimized for 2D spatial locality
Constant	Cached memory accessible by all threads, optimized for broadcasting to multiple threads within one warp, slower for unique access per thread

Sources:

https://www.arccompute.io/arc-blog/gpu-101-memory-hierarchy#:~:text=Global%20Memory:%20Size:%20Global%20memory%20is%20the,input%20data%2C%20output%20data%2C%20and%20global%20constants.

https://carpentries-incubator.github.io/lesson-gpu-programming/global_local_memory.html

https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticslocal.htm

https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/#:~:text=Access%20to%20shared%20memory%20is,mechanism%20for%20threads%20to%20cooperate.

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];
```

A bank conflict occurs when multiple threads in a warp try to access different addresses that map to the same memory bank of shared memory.

The shared memory is divided into 32 banks. We can guarantee no bank conflicts if we use a stride of 32 (s = 32).

If we set stride to 2 (s = 2), we would have a one-way bank conflict from threads 0 and 16 (if we had that many threads):

```
shared[baseIndex + s * threadIdx.x];
shared[baseIndex + (2 * 0) % 32] = shared[baseIndex + 0]
shared[baseIndex + (2 * 16) % 32] = shared[baseIndex + 0]
```

If we set the stride to 16, which feels intuitive, we actually end up with two-way conflict between banks 0 and 16.

```
shared[baseIndex + (16 * 0) % 32] = shared[baseIndex + 0]
shared[baseIndex + (16 * 1) % 32] = shared[baseIndex + 16]
shared[baseIndex + (16 * 2) % 32] = shared[baseIndex + 0]
shared[baseIndex + (16 * 3) % 32] = shared[baseIndex + 16]
shared[baseIndex + (16 * 4) % 32] = shared[baseIndex + 0]
…
```

The array bounds don’t matter, the bank size and stride alignment do.

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];
```

Each char maps to a bank, so unless we have a number of threads that is <=32, the number of banks, we're going to have bank conflicts. If baseIndex isn't 0, we can also wrap around and have bank conflicts. Knowing the size of shared would provide a clearer answer, but given a typical access where baseIndex == 0, there is a high potential to have each thread have conflicts if our number of threads exceeds 32.

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:

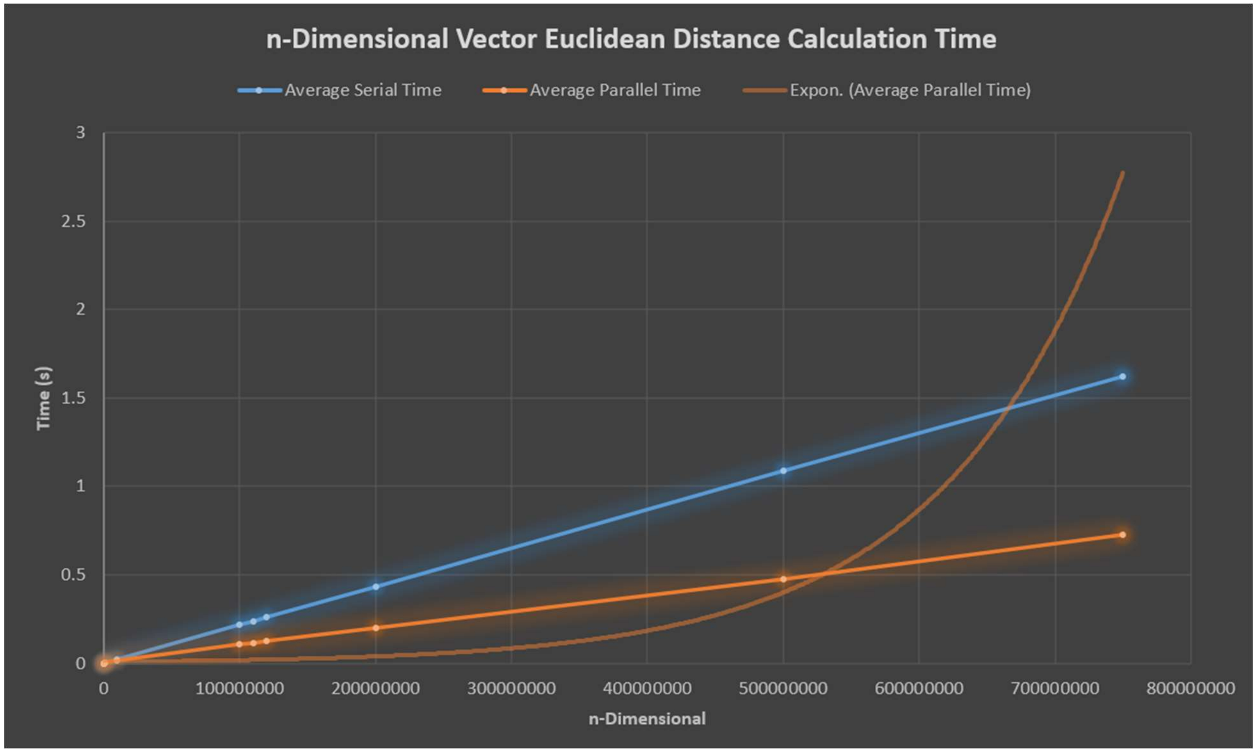
- Declare the arrays (host and device). All arrays should be dynamically allocated; the host arrays can be allocated either with `malloc` or `new`, while the device arrays should be allocated with `cudaMalloc`.
- Print the number of CUDA-enabled hardware devices attached to the system by calling `cudaGetDeviceCount`.
- 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?

```
Number of CUDA-enabled devices: 1
Device Name: NVIDIA GeForce GTX TITAN X
Compute Capability: 5.2
```

```
Total Global Memory: 12792627200 bytes
Shared Memory Per Block: 49152 bytes
Registers Per Block: 65536
```

Vector Dimension	Number of Tests	Average Serial Time (s)	Average Parallel Time (s)	Theoretical Speedup (Amdahl's Law)	Actual Speedup	Speedup Ratio (% of Theoretical)
1000	100	0.000002	0.000477	1.998002	0.004609	0.230667
100000	100	0.000218	0.000668	1.999980	0.326267	16.313530
1000000	100	0.002178	0.002671	1.999998	0.815509	40.775493
10000000	100	0.021802	0.018909	2.000000	1.152992	57.649604
100000000	100	0.217731	0.107885	2.000000	2.018177	100.908847
110000000	100	0.239486	0.116823	2.000000	2.049986	102.499319
120000000	100	0.261313	0.126060	2.000000	2.072920	103.646001
200000000	100	0.435498	0.199160	2.000000	2.186681	109.334049
500000000	100	1.088922	0.477112	2.000000	2.282322	114.116090
750000000	100	1.622030	0.725269	2.000000	2.236453	111.822659



Amdahl's requires me to guess at the parallelizable fraction of the work, which is why the percentage ultimately exceeds 100%. That data wasn't asked for in this assignment, so I did not bother adjusting it. The speedup ratio did eventually drop off, and an exponential trend line does show that while the rate of time increase is slower than the same calculations performed serially (that trend line distorts the graph), eventually we're going to slow down. As with other parallel calculations, the time required to push data to the GPU does have overhead, so it's not until we're calculating the distance between larger vectors (n>1000000) that we begin to take advantage of the GPU calculations.

```
/**
 * Author: Jason Gardner
 * Date: 11/6/2024
 * Class: COP6616 Parallel Computing
 * Instructor: Scott Piersall
```

```
 * Assignment: Homework 3
 * Filename: q2.cu
 * Description: This program computes the Euclidean distance between two points in n-D space using CUDA.
 */

/**
 * (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?
 */

// nvcc -lm q2.cu -o q2
// ./q2 1000000 100
```

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
// Interestingly, _Bool does not work in nvcc
#include <stdbool.h>
#include <math.h>

#define MIN 0
#define MAX 99
#define TOLERANCE 0.0001
#define PARALLELIZABLE_FRACTION 0.50
#define BLOCK_SIZE 512

/** Struct to store start and stop times */
typedef struct {
    struct timespec start;
    struct timespec stop;
} Stopwatch;

/** Seed the random number generator with entropy from /dev/urandom */
void seed_random() {
    int fd = open("/dev/urandom", O_RDONLY);
    unsigned int seed;
    read(fd, &seed, sizeof(seed));
    close(fd);
    srandom(seed);
}
```

```c
/** Calculate time in seconds
 * @param timer: The stopwatch struct
 * @return: The time in seconds
 */
double calculate_time(Stopwatch timer) {
    return (timer.stop.tv_sec - timer.start.tv_sec) + (timer.stop.tv_nsec - timer.start.tv_nsec) / 1e9;
}

/** Calculate the theoretical speedup using Amdahl's law */
double amdahl_speedup(int p) {
    return 1.0 / ((1.0 - PARALLELIZABLE_FRACTION) + (PARALLELIZABLE_FRACTION / p));
}

/**
 * Calculate the Euclidean distance between two vectors in serial
 * @param p: The first vector
 * @param q: The second vector
 * @param n: The number of dimensions
 * @return: The partial Euclidean distance
 */
double euclideanDistanceSerial(int* p, int* q, int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += (p[i] - q[i]) * (p[i] - q[i]);
    }
    return sqrt(sum);
}

/**
 * Calculate a random vector value between MIN and MAX
 * @return: A random vector value between MIN and MAX
```

```
 */
int random_vector_value() {
    return MIN + rand() % (MAX - MIN + 1);
}

/** Function to compare serial and parallel results with a tolerance */
bool compare_result(double result_s, double result_m) {
    return fabs(result_s - result_m) < TOLERANCE;
}

/** Display CUDA Information */
void display_cuda_info() {
    int device_count;
    cudaGetDeviceCount(&device_count);
    printf("Number of CUDA-enabled devices: %d\n", device_count);

    cudaDeviceProp device_properties;
    cudaGetDeviceProperties(&device_properties, 0);
    printf("Device Name: %s\n", device_properties.name);
    printf("Compute Capability: %d.%d\n", device_properties.major, device_properties.minor);
    printf("Total Global Memory: %lu bytes\n", device_properties.totalGlobalMem);
    printf("Shared Memory Per Block: %lu bytes\n", device_properties.sharedMemPerBlock);
    printf("Registers Per Block: %d\n", device_properties.regsPerBlock);
}

// CUDA kernel to compute squared differences
__global__ void squaredDifferenceKernel(const int *a, const int *b, double *result, const int m) {
    // Calculate starting index for this thread
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // Compute squared difference if within bounds
    if (i < m) {
```

```
        int diff = a[i] - b[i];
        result[i] = (double)(diff * diff);
    }
}

// Custom atomicAdd function for double because apparently I'm using some antiquated version of CUDA
__device__ double atomicAddDouble(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed, __double_as_longlong(val + __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}

// CUDA kernel for parallel reduction within a block
__global__ void reduceKernel(double *input, double *output, int size) {
    __shared__ double sharedData[BLOCK_SIZE];

    int threadId = threadIdx.x;
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // Load elements into shared memory
    sharedData[threadId] = (i < size) ? input[i] : 0.0;
    __syncthreads();

    // Perform reduction in shared memory
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (threadId < stride) {
            sharedData[threadId] += sharedData[threadId + stride];
        }
```

```
        __syncthreads();
    }

    // Write result of this block's reduction to global memory
    if (threadId == 0) {
        output[blockIdx.x] = sharedData[0];
    }
}

// CUDA kernel to perform final reduction across blocks
__global__ void finalReductionKernel(double *input, double *output, int size) {
    __shared__ double sharedData[BLOCK_SIZE];

    int threadId = threadIdx.x;
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // Load elements into shared memory
    sharedData[threadId] = (i < size) ? input[i] : 0.0;
    __syncthreads();

    // Perform reduction in shared memory
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (threadId < stride) {
            sharedData[threadId] += sharedData[threadId + stride];
        }
        __syncthreads();
    }

    // Write result of this block's reduction to global memory
    if (threadId == 0) {
        atomicAddDouble(output, sharedData[0]);
    }
```

```
}

// Host function to compute Euclidean distance
double euclideanDistanceCUDA(const int *a, const int *b, int m) {
    int *d_a, *d_b;
    double *d_temp, *d_block_sums, *d_result;

    // Allocate memory on the device for vectors and the temporary result array
    cudaMalloc((void**)&d_a, m * sizeof(int));
    cudaMalloc((void**)&d_b, m * sizeof(int));
    cudaMalloc((void**)&d_temp, m * sizeof(double));
    cudaMalloc((void**)&d_block_sums, ((m + BLOCK_SIZE - 1) / BLOCK_SIZE) * sizeof(double));
    cudaMalloc((void**)&d_result, sizeof(double));

    // Copy data from host to device
    cudaMemcpy(d_a, a, m * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, m * sizeof(int), cudaMemcpyHostToDevice);

    // Initialize result to 0
    double initial_result = 0.0;
    cudaMemcpy(d_result, &initial_result, sizeof(double), cudaMemcpyHostToDevice);

    // Launch kernel to compute squared differences
    int numBlocks = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
    squaredDifferenceKernel<<<numBlocks, BLOCK_SIZE>>>(d_a, d_b, d_temp, m);

    // Perform reduction to sum up squared differences within each block
    reduceKernel<<<numBlocks, BLOCK_SIZE>>>(d_temp, d_block_sums, m);

    // Launch final reduction kernel to sum up all block results
    int finalNumBlocks = (numBlocks + BLOCK_SIZE - 1) / BLOCK_SIZE;
    finalReductionKernel<<<finalNumBlocks, BLOCK_SIZE>>>(d_block_sums, d_result, numBlocks);
```

```
    // Copy the final result to host
    double sum;
    cudaMemcpy(&sum, d_result, sizeof(double), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_temp);
    cudaFree(d_block_sums);
    cudaFree(d_result);

    // Return the square root of the sum to compute the Euclidean distance
    return sqrt(sum);
}

/**
 * Main function
 * @param argc: Number of arguments
 * @param argv: Array of arguments
 * @return: 0 if successful
 */
int main(int argc, char** argv) {
    // Command line argument variables
    unsigned int num_runs;
    unsigned int m;

    // Parse command line arguments or use default values
    if (argc < 3) {
        printf("Usage:\t\t\t%s <vector dimension> <number of runs to average>\n", argv[0]);
        if (argc < 3) {
            printf("Using default values:\tnum_runs = 100, m = 1000000\n\n");
```

```
            m = 1000000;
            num_runs = 100;
        }
        else if (argc < 2) {
            printf("Using default value:\tnum_runs = 100\n\n");
            num_runs = 100;
            sscanf(argv[1], "%u", &m);
        }

    }
    else {
        sscanf(argv[1], "%u", &m);
        sscanf(argv[2], "%u", &num_runs);
    }

    // Validate command line arguments
    if (m < 1) {
        fprintf(stderr, "Vector dimension must be greater than 0!\n");
        exit(EXIT_FAILURE);
    }
    if (num_runs < 1) {
        fprintf(stderr, "Number of runs must be greater than 0!\n");
        exit(EXIT_FAILURE);
    }

    // Allocate vector memory
    int* vector1 = (int*) malloc(m * sizeof(int));
    int* vector2 = (int*) malloc(m * sizeof(int));

    if (!vector1 || !vector2) {
        fprintf(stderr, "Vector memory allocation failed!\n");
        exit(EXIT_FAILURE);
```

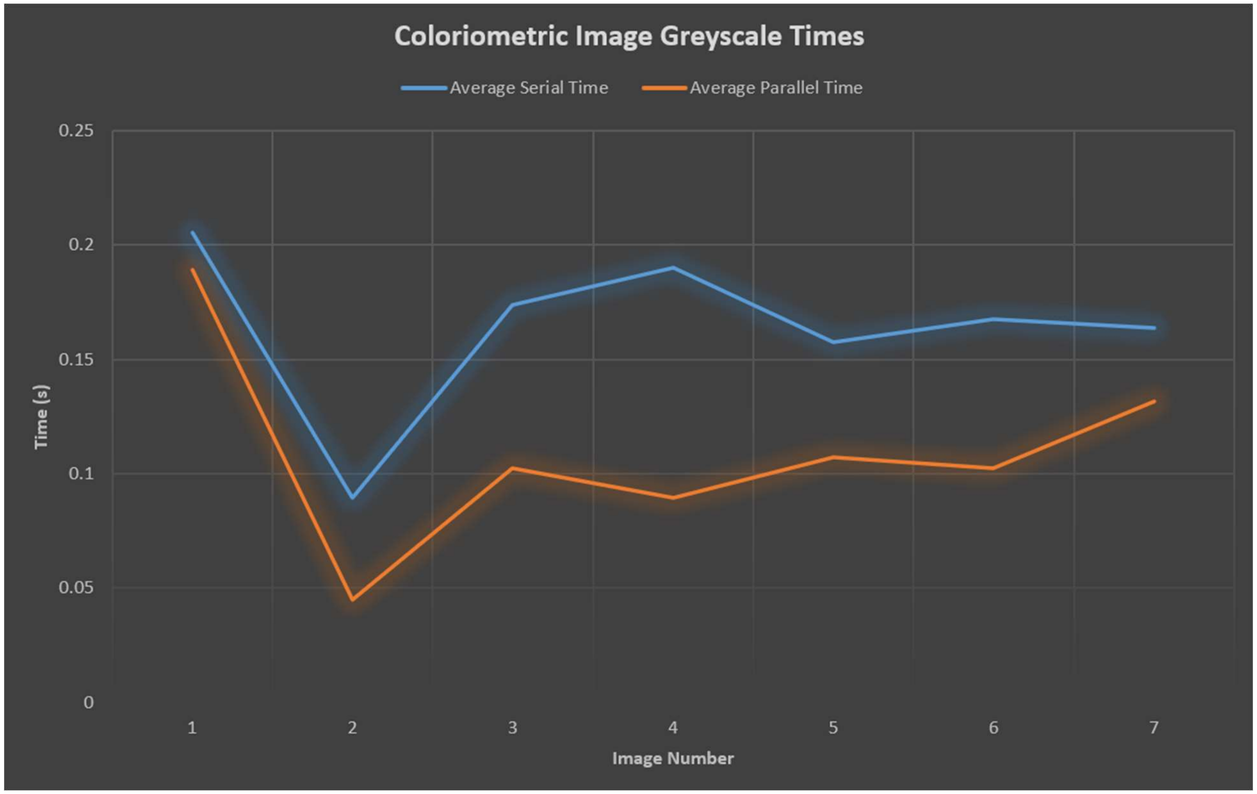
```
    }

    // Fill vectors with random values
    seed_random();
    for (int i = 0; i < m; i++) {
        vector1[i] = random_vector_value();
        vector2[i] = random_vector_value();
    }

    // Variables for tracking time
    double total_parallel_time = 0;
    double total_serial_time = 0;

    display_cuda_info();

    // Perform the Euclidean distance calculation for num_runs
    for (int run = 0; run < num_runs; run++) {
        Stopwatch parallel_timer, serial_timer;

        // Start parallel timing
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.start);

        // Perform parallel Euclidean distance calculation
        double result_c = euclideanDistanceCUDA(vector1, vector2, m);

        // Stop parallel timing
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.stop);
        total_parallel_time += calculate_time(parallel_timer);

        // Start serial timing
        clock_gettime(CLOCK_MONOTONIC, &serial_timer.start);
```

```
        // Perform serial Euclidean distance calculation
        double result_s = euclideanDistanceSerial(vector1, vector2, m);

        clock_gettime(CLOCK_MONOTONIC, &serial_timer.stop);
        total_serial_time += calculate_time(serial_timer);

        // Compare results after each run
        if (!compare_result(result_s, result_c)) {
            printf("Results do not match in run %d! Serial: %lf, Parallel: %lf\n", run + 1, result_s, result_c);
        }
    }

    // Calculate average times
    double average_parallel_time = total_parallel_time / num_runs;
    double average_serial_time = total_serial_time / num_runs;

    // Output results and speedup calculations
    double actual_speedup = average_serial_time / average_parallel_time;
    double theoretical_speedup = amdahl_speedup(m);
    double speedup_ratio = (actual_speedup / theoretical_speedup) * 100;

    printf("Average Serial Time:\t\t\t%lfs\n", average_serial_time);
    printf("Average Parallel Time:\t\t\t%lfs\n", average_parallel_time);
    printf("Theoretical Speedup [Amdahl's Law]:\t%lf\n", theoretical_speedup);
    printf("Actual Speedup:\t\t\t\t%lf\n", actual_speedup);
    printf("Speedup Efficiency:\t\t\t%lf%%\n", speedup_ratio);

    // Free allocated memory
    free(vector1);
    free(vector2);

    return 0;
```

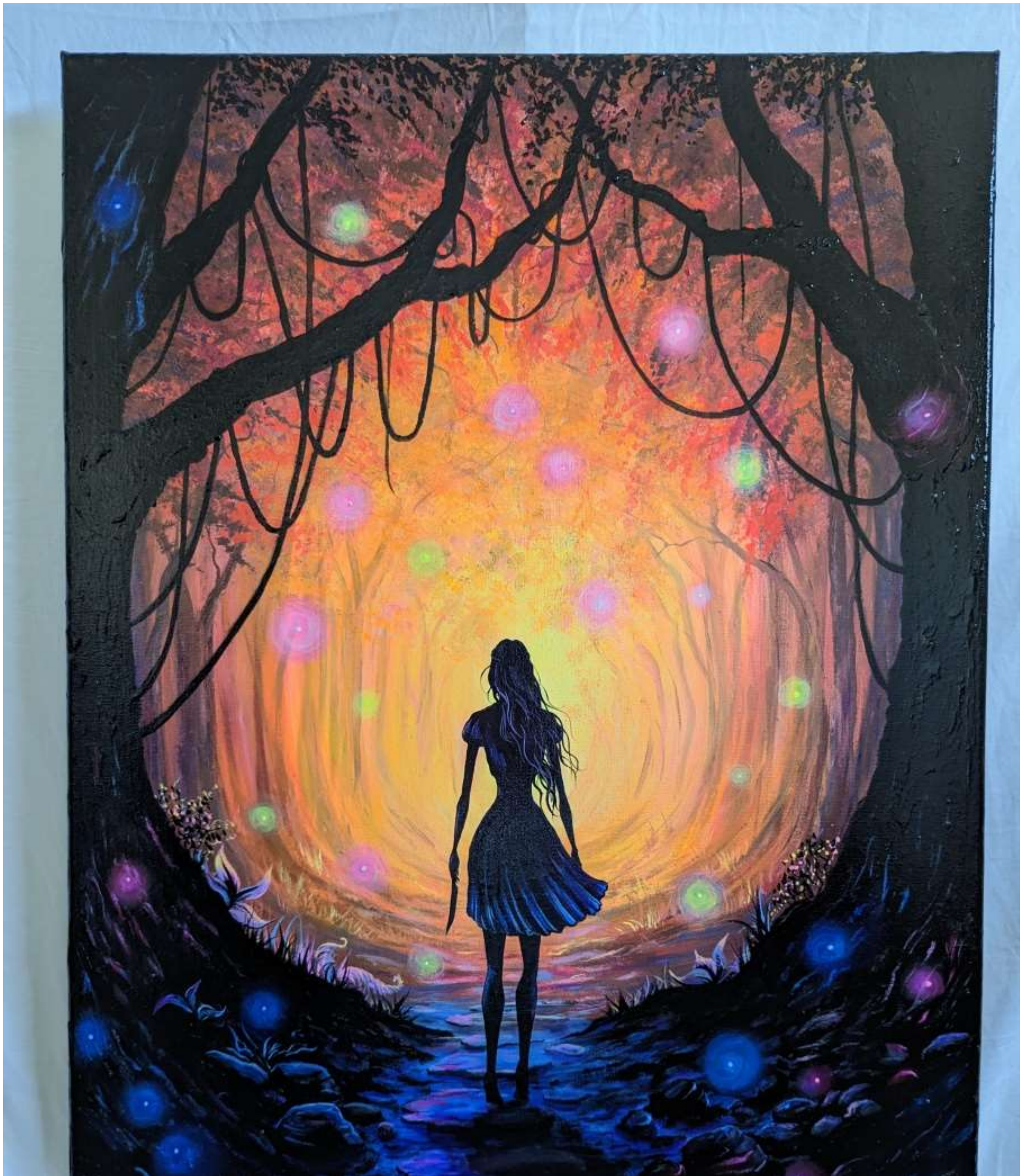
```
}
```

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?

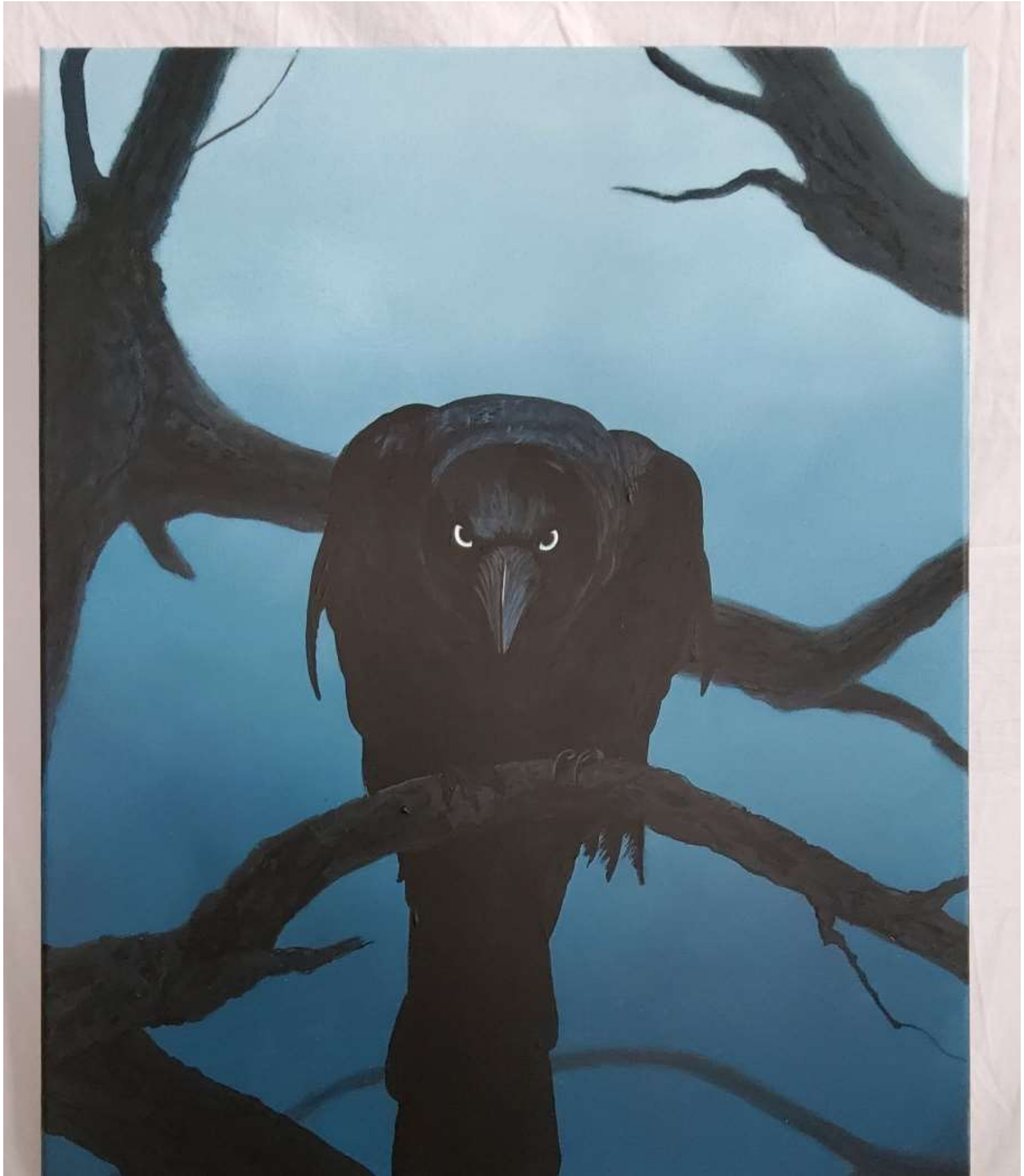
Image Number	**Average Serial Time (s)**	**Average Parallel Time (s)**	**Theoretical Speedup (Amdahl's Law)**	**Actual Speedup**	**Speedup Ratio (% of Theoretical)**
1	**0.205432**	**0.188922**	**1.665582**	**1.087390**	**65.285871**
2	**0.089472**	**0.044951**	**1.665582**	**1.990425**	**119.503240**
3	**0.173901**	**0.102234**	**1.665582**	**1.701008**	**102.126896**
4	**0.189884**	**0.089226**	**1.665582**	**1.701008**	**102.126896**
5	**0.157423**	**0.107307**	**1.665582**	**1.467029**	**88.079039**
6	**0.167432**	**0.102241**	**1.665582**	**1.637612**	**98.320699**
7	**0.163922**	**0.131684**	**1.665582**	**1.244815**	**74.737519**



There was a speedup using the GPU to parallelize this problem, yes. The GPU is suited to this problem both because we have a large number of pixels for each image that can be calculated in parallel (SIMD), and because the GPU has hardware that is optimized for floating point operations. Additionally, the GPU makes use of high memory bandwidth and coalesced memory access (optimization for accessing adjacent memory locations). Images are small enough to load into memory for both problems, but it's far faster on the GPU as the number of pixels increases. If I had thought to process all of the test images at once, instead of in series, the difference would have been more pronounced.




CatherineK. G.




CatherineK.G.

```
/**
 * Author: Jason Gardner
 * Date: 11/6/2024
 * Class: COP6616 Parallel Computing
 * Instructor: Scott Piersall
 * Assignment: Homework 3
 * Filename: q3.cu
 * Description: This program converts a color image to grayscale using the Colorimetric method. It compares the
 * performance of a serial CPU-based implementation to a parallel CUDA-based implementation. The program reads
 * PNG images from the "in" directory and writes the grayscale images to the "out" directory. The program
outputs
 * the average time taken for the serial and parallel implementations, the theoretical speedup using Amdahl's
law,
 * the actual speedup, and the speedup efficiency.
 *
 * INPUT IMAGES ARE OWNED BY CATHERINE ANDERSON OF THE PAINTED ME, LLC AND ARE USED FOR EDUCATIONAL PURPOSES
ONLY
 * SHE IS MY PARTNER AND I HAVE PERMISSION TO USE THEM (I AM ALSO A PART OWNER OF THE PAINTED ME, LLC)
 * IMAGE SOURCE: https://www.thepaintedme.com/
 */

/**
 * (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?
 */

// nvcc -lm -lpng q3.cu -o q3
// OR
// nvcc -lm q3.cu -o q3 -L$HOME/libpng/lib -I$HOME/libpng/include -lpng
// ./q3

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <png.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
// Interestingly, _Bool does not work in nvcc
#include <stdbool.h>
#include <math.h>

#define PARALLELIZABLE_FRACTION 0.40
#define BLOCK_SIZE 1024
#define NUM_IMAGES 7
#define INPUT_LOCATION "./in/"
#define OUTPUT_LOCATION "./out/"
#define R 0.2126
#define G 0.7152
#define B 0.0722
```

```c
#define FILENAME_SIZE 256

/** Struct to store start and stop times */
typedef struct {
    struct timespec start;
    struct timespec stop;
} Stopwatch;

/** Calculate time in seconds
 * @param timer: The stopwatch struct
 * @return: The time in seconds
 */
double calculate_time(Stopwatch timer) {
    return (timer.stop.tv_sec - timer.start.tv_sec) + (timer.stop.tv_nsec - timer.start.tv_nsec) / 1e9;
}

/** Calculate the theoretical speedup using Amdahl's law */
double amdahl_speedup(int p) {
    return 1.0 / ((1.0 - PARALLELIZABLE_FRACTION) + (PARALLELIZABLE_FRACTION / p));
}

// Convert PNG image to greyscale and save the output (Serial version)
void convertPNGToGreyScaleSerial(char* filename, char* output_filename) {
    // Open the input file
    FILE* file = fopen(filename, "rb");
    if (!file) {
        fprintf(stderr, "Error opening file %s\n", filename);
        exit(EXIT_FAILURE);
    }

    // Create PNG read struct
    // png_create_read_struct initializes the PNG structure used for reading data.
```

```
    png_structp png = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png) {
        fprintf(stderr, "Error creating read struct\n");
        exit(EXIT_FAILURE);
    }

    // Create PNG info struct
    // png_create_info_struct creates a PNG info structure that stores image information.
    png_infop info = png_create_info_struct(png);
    if (!info) {
        fprintf(stderr, "Error creating info struct\n");
        exit(EXIT_FAILURE);
    }

    // Set up error handling using setjmp/longjmp mechanism
    // If any error occurs within libpng, control will jump to this point using longjmp.
    if (setjmp(png_jmpbuf(png))) {
        fprintf(stderr, "Error during init_io\n");
        exit(EXIT_FAILURE);
    }

    // Initialize PNG IO
    // png_init_io sets the input file stream for reading PNG data.
    png_init_io(png, file);
    png_read_info(png, info);

    // Get image information (width, height, color type, bit depth)
    int width = png_get_image_width(png, info);
    int height = png_get_image_height(png, info);
    png_byte color_type = png_get_color_type(png, info);
    png_byte bit_depth = png_get_bit_depth(png, info);
```

```
    // Transformations to ensure correct format for processing
    if (bit_depth == 16) png_set_strip_16(png); // Strip 16-bit depth to 8-bit
    if (color_type == PNG_COLOR_TYPE_PALETTE) png_set_palette_to_rgb(png); // Convert palette images to RGB
    if (color_type == PNG_COLOR_TYPE_GRAY && bit_depth < 8) png_set_expand_gray_1_2_4_to_8(png); // Expand
grayscale images to 8-bit
    if (png_get_valid(png, info, PNG_INFO_tRNS)) png_set_tRNS_to_alpha(png); // Convert transparency to alpha
channel

    // Update the info struct after applying transformations
    png_read_update_info(png, info);

    // Allocate memory for row pointers (each row of the image)
    // png_get_rowbytes returns the number of bytes required to hold one row of image data.
    png_bytep* row_pointers = (png_bytep*)malloc(sizeof(png_bytep) * height);
    for (int y = 0; y < height; y++) {
        row_pointers[y] = (png_byte*)malloc(png_get_rowbytes(png, info));
    }

    // Read the image data into row pointers
    // png_read_image reads the entire image into memory (row_pointers).
    png_read_image(png, row_pointers);
    fclose(file);

    // Manually convert RGB to grayscale using the given formula
    for (int y = 0; y < height; y++) {
        png_bytep row = row_pointers[y];
        for (int x = 0; x < width; x++) {
            int r = row[x * 3];
            int g = row[x * 3 + 1];
            int b = row[x * 3 + 2];
            unsigned char gray = (unsigned char)(R * r + G * g + B * b);
            row[x * 3] = gray;
```

```
            row[x * 3 + 1] = gray;
            row[x * 3 + 2] = gray;
        }
    }

    // Write the greyscale image to the output file
    FILE* output_file = fopen(output_filename, "wb");
    if (!output_file) {
        fprintf(stderr, "Error opening output file %s\n", output_filename);
        exit(EXIT_FAILURE);
    }

    // Create PNG write struct
    // png_create_write_struct initializes the PNG structure used for writing data.
    png_structp png_out = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png_out) {
        fprintf(stderr, "Error creating write struct\n");
        exit(EXIT_FAILURE);
    }

    // Create PNG info struct for output
    // png_create_info_struct creates a PNG info structure for writing image information.
    png_infop info_out = png_create_info_struct(png_out);
    if (!info_out) {
        fprintf(stderr, "Error creating info struct\n");
        exit(EXIT_FAILURE);
    }

    // Set up error handling for writing using setjmp/longjmp
    if (setjmp(png_jmpbuf(png_out))) {
        fprintf(stderr, "Error during writing header\n");
        exit(EXIT_FAILURE);
```

```
    }

    // Initialize PNG IO for output file
    // png_init_io sets the output file stream for writing PNG data.
    png_init_io(png_out, output_file);
    png_set_IHDR(
        png_out,
        info_out,
        width, height,
        8, // Bit depth
        PNG_COLOR_TYPE_RGB, // Color type
        PNG_INTERLACE_NONE,
        PNG_COMPRESSION_TYPE_DEFAULT,
        PNG_FILTER_TYPE_DEFAULT
    );
    png_write_info(png_out, info_out);

    // Write the image data row by row
    if (setjmp(png_jmpbuf(png_out))) {
        fprintf(stderr, "Error during writing bytes\n");
        exit(EXIT_FAILURE);
    }

    for (int y = 0; y < height; y++) {
        png_bytep row = row_pointers[y];
        png_write_row(png_out, row);
    }

    // Finalize writing the PNG file
    if (setjmp(png_jmpbuf(png_out))) {
        fprintf(stderr, "Error during end of write\n");
        exit(EXIT_FAILURE);
```

```
    }

    png_write_end(png_out, NULL);

    // Cleanup allocated memory
    for (int y = 0; y < height; y++) {
        free(row_pointers[y]);
    }
    free(row_pointers);
    fclose(output_file);
}

// Convert PNG image to greyscale and save the output (CUDA version)
__global__ void greyscale_kernel(unsigned char* d_input, unsigned char* d_output, int width, int height) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < width * height) {
        int pixel_idx = idx * 3;
        int r = d_input[pixel_idx];
        int g = d_input[pixel_idx + 1];
        int b = d_input[pixel_idx + 2];
        d_output[idx] = (unsigned char)(R * r + G * g + B * b);
    }
}

void convertPNGToGreyScaleCUDA(char* filename, char* output_filename) {
    // Open the input file
    FILE* file = fopen(filename, "rb");
    if (!file) {
        fprintf(stderr, "Error opening file %s\n", filename);
        exit(EXIT_FAILURE);
    }
```

```
    // Create PNG read struct
    // png_create_read_struct initializes the PNG structure used for reading data.
    png_structp png = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png) {
        fprintf(stderr, "Error creating read struct\n");
        exit(EXIT_FAILURE);
    }

    // Create PNG info struct
    // png_create_info_struct creates a PNG info structure that stores image information.
    png_infop info = png_create_info_struct(png);
    if (!info) {
        fprintf(stderr, "Error creating info struct\n");
        exit(EXIT_FAILURE);
    }

    // Set up error handling using setjmp/longjmp mechanism
    if (setjmp(png_jmpbuf(png))) {
        fprintf(stderr, "Error during init_io\n");
        exit(EXIT_FAILURE);
    }

    // Initialize PNG IO
    // png_init_io sets the input file stream for reading PNG data.
    png_init_io(png, file);
    png_read_info(png, info);

    // Get image information (width, height, color type, bit depth)
    int width = png_get_image_width(png, info);
    int height = png_get_image_height(png, info);
    png_byte color_type = png_get_color_type(png, info);
    png_byte bit_depth = png_get_bit_depth(png, info);
```

```
    // Transformations to ensure correct format for processing
    if (bit_depth == 16) png_set_strip_16(png); // Strip 16-bit depth to 8-bit
    if (color_type == PNG_COLOR_TYPE_PALETTE) png_set_palette_to_rgb(png); // Convert palette images to RGB
    if (color_type == PNG_COLOR_TYPE_GRAY && bit_depth < 8) png_set_expand_gray_1_2_4_to_8(png); // Expand
grayscale images to 8-bit
    if (png_get_valid(png, info, PNG_INFO_tRNS)) png_set_tRNS_to_alpha(png); // Convert transparency to alpha
channel

    // Update the info struct after applying transformations
    png_read_update_info(png, info);

    // Allocate memory for row pointers (each row of the image)
    // png_get_rowbytes returns the number of bytes required to hold one row of image data.
    png_bytep* row_pointers = (png_bytep*)malloc(sizeof(png_bytep) * height);
    for (int y = 0; y < height; y++) {
        row_pointers[y] = (png_byte*)malloc(png_get_rowbytes(png, info));
    }

    // Read the image data into row pointers
    // png_read_image reads the entire image into memory (row_pointers).
    png_read_image(png, row_pointers);
    fclose(file);

    // Copy image data to linear buffer (h_input)
    unsigned char* h_input = (unsigned char*)malloc(3 * width * height);
    unsigned char* h_output = (unsigned char*)malloc(width * height);
    for (int y = 0; y < height; y++) {
        memcpy(h_input + y * width * 3, row_pointers[y], png_get_rowbytes(png, info));
    }

    // Allocate device memory and copy data to GPU
```

```
    unsigned char *d_input, *d_output;
    cudaMalloc(&d_input, 3 * width * height);
    cudaMalloc(&d_output, width * height);
    cudaMemcpy(d_input, h_input, 3 * width * height, cudaMemcpyHostToDevice);

    // Launch kernel to convert to greyscale
    int num_threads = BLOCK_SIZE;
    int num_blocks = (width * height + num_threads - 1) / num_threads;
    greyscale_kernel<<<num_blocks, num_threads>>>(d_input, d_output, width, height);

    // Copy result back to host
    cudaMemcpy(h_output, d_output, width * height, cudaMemcpyDeviceToHost);

    // Write the greyscale image to the output file
    FILE* output_file = fopen(output_filename, "wb");
    if (!output_file) {
        fprintf(stderr, "Error opening output file %s\n", output_filename);
        exit(EXIT_FAILURE);
    }

    // Create PNG write struct
    // png_create_write_struct initializes the PNG structure used for writing data.
    png_structp png_out = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png_out) {
        fprintf(stderr, "Error creating write struct\n");
        exit(EXIT_FAILURE);
    }

    // Create PNG info struct for output
    // png_create_info_struct creates a PNG info structure for writing image information.
    png_infop info_out = png_create_info_struct(png_out);
    if (!info_out) {
```

```
        fprintf(stderr, "Error creating info struct\n");
        exit(EXIT_FAILURE);
    }

    // Set up error handling for writing using setjmp/longjmp
    if (setjmp(png_jmpbuf(png_out))) {
        fprintf(stderr, "Error during writing header\n");
        exit(EXIT_FAILURE);
    }

    // Initialize PNG IO for output file
    // png_init_io sets the output file stream for writing PNG data.
    png_init_io(png_out, output_file);
    png_set_IHDR(
        png_out,
        info_out,
        width, height,
        8, // Bit depth
        PNG_COLOR_TYPE_GRAY, // Color type
        PNG_INTERLACE_NONE,
        PNG_COMPRESSION_TYPE_DEFAULT,
        PNG_FILTER_TYPE_DEFAULT
    );
    png_write_info(png_out, info_out);

    // Write the image data row by row
    if (setjmp(png_jmpbuf(png_out))) {
        fprintf(stderr, "Error during writing bytes\n");
        exit(EXIT_FAILURE);
    }

    for (int y = 0; y < height; y++) {
```

```
        png_bytep row = (png_bytep)malloc(width * sizeof(png_byte));
        for (int x = 0; x < width; x++) {
            row[x] = h_output[y * width + x];
        }
        png_write_row(png_out, row);
        free(row);
    }

    // Finalize writing the PNG file
    if (setjmp(png_jmpbuf(png_out))) {
        fprintf(stderr, "Error during end of write\n");
        exit(EXIT_FAILURE);
    }

    png_write_end(png_out, NULL);

    // Cleanup allocated memory
    for (int y = 0; y < height; y++) {
        free(row_pointers[y]);
    }
    free(row_pointers);
    free(h_input);
    free(h_output);
    cudaFree(d_input);
    cudaFree(d_output);

    fclose(output_file);
}

/**
 * Main function
 * @param argc: Number of arguments
```

```
 * @param argv: Array of arguments
 * @return: 0 if successful
 */
int main(int argc, char** argv) {
    char input_filename[FILENAME_SIZE];
    char output_filename_s[FILENAME_SIZE];
    char output_filename_c[FILENAME_SIZE];

    for (int image_num = 1; image_num < NUM_IMAGES + 1; image_num++) {
        // Create output filenames
        sprintf(input_filename, "%s%d.png", INPUT_LOCATION, image_num);
        sprintf(output_filename_s, "%simage_%d_s.png", OUTPUT_LOCATION, image_num);
        sprintf(output_filename_c, "%simage_%d_c.png", OUTPUT_LOCATION, image_num);

        Stopwatch parallel_timer, serial_timer;

        // Start serial timing
        clock_gettime(CLOCK_MONOTONIC, &serial_timer.start);
        convertPNGToGreyScaleSerial(input_filename, output_filename_s);
        clock_gettime(CLOCK_MONOTONIC, &serial_timer.stop);
        double total_serial_time = calculate_time(serial_timer);

        // Start parallel timing
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.start);
        convertPNGToGreyScaleCUDA(input_filename, output_filename_c);
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.stop);
        double total_parallel_time = calculate_time(parallel_timer);

        // Output results and speedup calculations
        double actual_speedup = total_serial_time / total_parallel_time;
        double theoretical_speedup = amdahl_speedup(BLOCK_SIZE);
        double speedup_ratio = (actual_speedup / theoretical_speedup) * 100;
```

```
        printf("Image %d:\n", image_num);
        printf("Average Serial Time:\t\t\t%lfs\n", total_serial_time);
        printf("Average Parallel Time:\t\t\t%lfs\n", total_parallel_time);
        printf("Theoretical Speedup [Amdahl's Law]:\t%lf\n", theoretical_speedup);
        printf("Actual Speedup:\t\t\t%lf\n", actual_speedup);
        printf("Speedup Efficiency:\t\t\t%lf%%\n\n", speedup_ratio);
    }

    return 0;
}
```

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.

Full testing of this code was ultimately limited by having to run the serial code. I also tested with integers, because I had read a study where somebody used integers and "it's much easier math" for AI. But, at the limits of what I could test on the GPU at the school, I didn't see an improvement with floating point numbers. Given I wrote about it for question 3, I'm a bit confused. It's supposed to be a benefit, but the GPU handled integers like a champ. The speedups are the right two fields of the table on page 58. Once the matrix size becomes large, the speedup starts increasing, by a substantial amount.

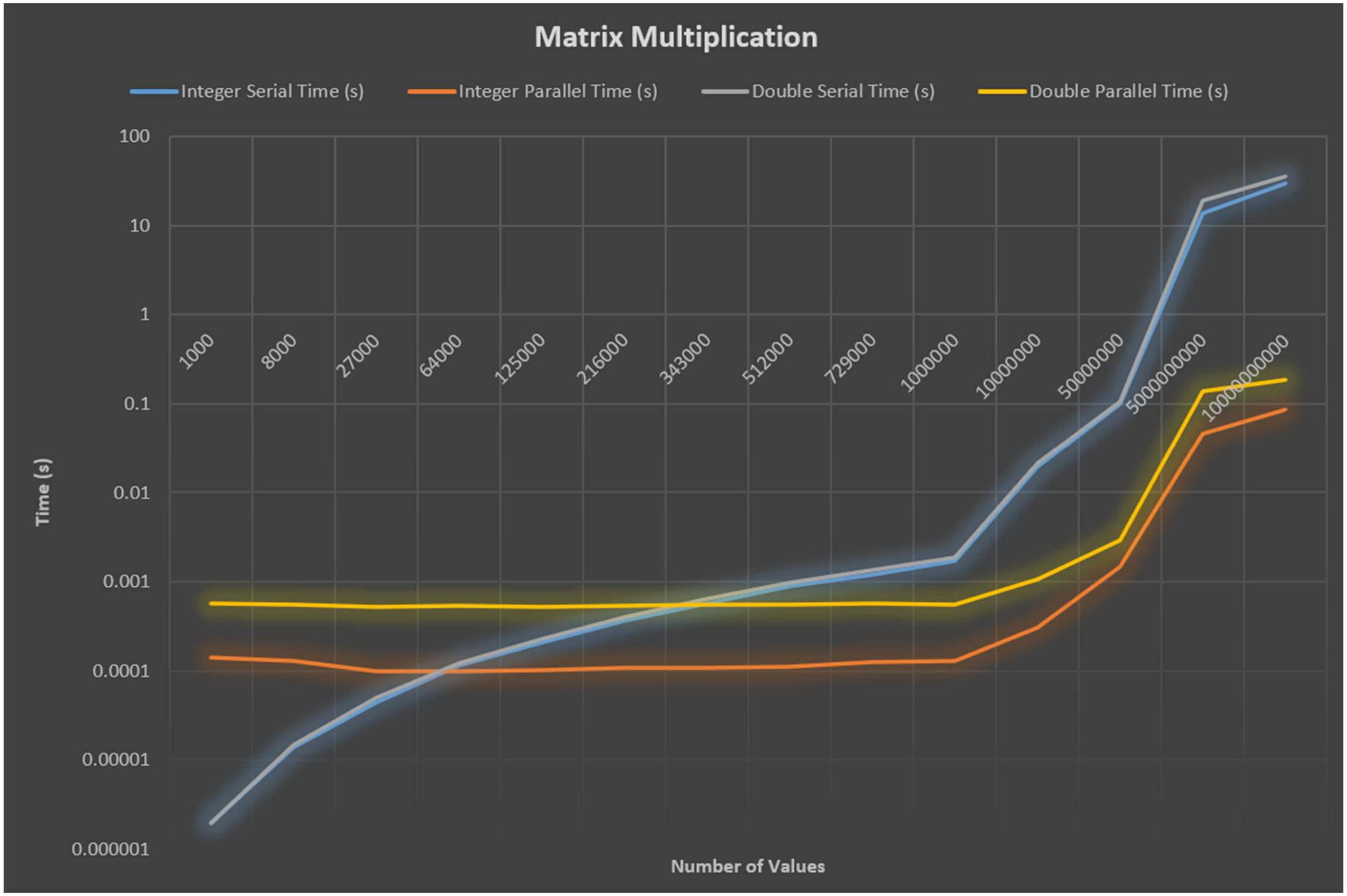
<table>
<tr><th></th><th>CPU</th><th>GPU</th></tr>
<tr><td>Parallelism</td><td>Sequential, one matrix element at a time</td><td>Thousands of parallel threads calculating matrix elements concurrently</td></tr>
<tr><td>Memory Access Patterns</td><td>Main memory or cache</td><td>Hierarchal memory model, shared memory reducing global accesses</td></tr>
<tr><td>Computation Distribution</td><td>Single thread handles all computation, limited by clock speed/cores.</td><td>Distributed across many threads. Exploits data-level parallelism inherent in matrix multiplication.</td></tr>
<tr><td>Synchronization</td><td>N/A</td><td>Utilizes __syncthreads() to synchronize threads within a block. Ensures all threads have loaded data into shared memory.</td></tr>
<tr><td>Code Structure</td><td><pre>void matrixMultiplySerial(double* A,
double* B, double* C, int m, int n,
int k) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            double sum = 0;
            for (int l = 0; l < n;
l++) {
                sum += A[i * n + l] *
B[l * k + j];
            }
            C[i * k + j] = sum;
        }
    }
}</pre></td><td><pre>__global__ void
matrixMultiplyKernelShared(double*
A, double* B, double* C, int m, int
n, int k) {
    __shared__ double
As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double
Bs[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y *
blockDim.y + threadIdx.y;
    int col = blockIdx.x *
blockDim.x + threadIdx.x;
    double value = 0.0;

    for (int t = 0; t < (n +
BLOCK_SIZE - 1) / BLOCK_SIZE; t++) {
        // Data -> Shared memory
        // Synchronize threads
        // Compute partial sums</pre></td></tr>
</table>

		`` } if (row < m && col < k) { C[row * k + col] = value; } } ```
Performance	**Limited by CPU speed and memory bandwidth**	**Benefits from massive parallelism. Scales well into larger matrices due to parallel computation.**

m	n	k	Integer		Double		Δt_s (s)	Δt_p (s)	S_i	S_d
			Serial Time (s)	Parallel Time (s)	Serial Time (s)	Parallel Time (s)				
10	10	10	0.000002	0.000140	0.000002	0.000570	0.000000	-0.000429	0.01428571	0.00350877
20	20	20	0.000014	0.000129	0.000015	0.000559	-0.000001	-0.000430	0.10852713	0.02683363
30	30	30	0.000045	0.000100	0.000050	0.000524	-0.000005	-0.000425	0.45	0.09541985
40	40	40	0.000115	0.000099	0.000124	0.000535	-0.000010	-0.000436	1.16161616	0.2317757
50	50	50	0.000207	0.000101	0.000229	0.000530	-0.000022	-0.000428	2.04950495	0.43207547
60	60	60	0.000365	0.000107	0.000397	0.000533	-0.000032	-0.000426	3.41121495	0.74484053
70	70	70	0.000567	0.000110	0.000634	0.000554	-0.000067	-0.000444	5.15454545	1.14440433
80	80	80	0.000889	0.000112	0.000966	0.000553	-0.000077	-0.000441	7.9375	1.74683544
90	90	90	0.001210	0.000125	0.001342	0.000574	-0.000131	-0.000125	9.68	2.33797909
100	100	100	0.001727	0.000129	0.001879	0.000558	-0.000152	-0.000429	13.3875969	3.36738351
100	100	1000	0.019716	0.000308	0.021104	0.001074	-0.001388	-0.000766	64.012987	19.6499069
100	100	5000	0.099439	0.001489	0.105567	0.002920	-0.006127	-0.001431	66.7824043	36.1530822
1000	1000	5000	13.646605	0.046152	19.005602	0.137432	-5.358997	-0.091280	295.688269	138.290951
1000	1000	10000	29.542459	0.085631	35.720104	0.185100	-6.177645	-0.099470	344.997244	192.977331


Matrix Multiplication
Integer Serial Time (s)
Integer Parallel Time (s)
Double Serial Time (s)
Double Parallel Time (s)
Time (s)
100
10
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
1000
8000
27000
64000
125000
216000
343000
512000
729000
1000000
10000000
50000000
5000000000
10000000000
Number of Values

```
/**
 * Author: Jason Gardner
 * Date: 11/6/2024
 * Class: COP6616 Parallel Computing
 * Instructor: Scott Piersall
 * Assignment: Homework 3
 * Filename: q4.cu
 * Description: This program computes the multiplication of two matrices using
CUDA. The matrices are filled with random
 * values and the results are compared to a serial implementation. The program
calculates the average time for both the
 * serial and parallel implementations, as well as the theoretical and actual
speedup. The program also outputs the
 * difference in average serial time, average parallel time, theoretical speedup,
actual speedup, and speedup efficiency
 * between the double and integer implementations.
 */

/**
 * (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.
 */

// nvcc -lm q4.cu -o q4
// ./q4 1000000 1000000 100

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
```

```
#include <stdbool.h>
#include <math.h>

#define MIN 0
#define MAX 99
#define TOLERANCE 0.000001
#define PARALLELIZABLE_FRACTION_DOUBLE 0.40
#define PARALLELIZABLE_FRACTION_INTEGER 0.99
#define BLOCK_SIZE 16
#define MAX_ELEMENTS 1000000000 // Threshold for warning about potential
segmentation faults

/** Struct to store start and stop times */
typedef struct {
    struct timespec start;
    struct timespec stop;
} Stopwatch;

/** Seed the random number generator with entropy from /dev/urandom */
void seed_random() {
    int fd = open("/dev/urandom", O_RDONLY);
    unsigned int seed;
    read(fd, &seed, sizeof(seed));
    close(fd);
    srandom(seed);
}

/** Calculate time in seconds
 * @param timer: The stopwatch struct
 * @return: The time in seconds
 */
double calculate_time(Stopwatch timer) {
    return (timer.stop.tv_sec - timer.start.tv_sec) + (timer.stop.tv_nsec -
timer.start.tv_nsec) / 1e9;
}

/** Calculate the theoretical speedup using Amdahl's law */
double amdahl_speedup(int p, double PARALLELIZABLE_FRACTION) {
    return 1.0 / ((1.0 - PARALLELIZABLE_FRACTION) + (PARALLELIZABLE_FRACTION /
p));
}

/**
 * Generate a random double value between MIN and MAX
 * @return: A random double value
```

```
 */
double random_double() {
    return MIN + ((double)rand() / RAND_MAX) * (MAX - MIN);
}

/**
 * Generate a random integer value between MIN and MAX
 * @return: A random integer value
 */
int random_int() {
    return MIN + rand() % (MAX - MIN + 1);
}

/** Function to compare two matrices with a tolerance */
bool compare_matrices(double* C_serial, double* C_parallel, int m, int k) {
    for (int i = 0; i < m * k; i++) {
        if (fabs(C_serial[i] - C_parallel[i]) > TOLERANCE) {
            return false;
        }
    }
    return true;
}

/** Matrix multiplication in serial (double) */
void matrixMultiplySerial(double* A, double* B, double* C, int m, int n, int k) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            double sum = 0;
            for (int l = 0; l < n; l++) {
                sum += A[i * n + l] * B[l * k + j];
            }
            C[i * k + j] = sum;
        }
    }
}

/** Matrix multiplication in serial (int) */
void matrixMultiplySerialInt(int* A, int* B, int* C, int m, int n, int k) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            int sum = 0;
            for (int l = 0; l < n; l++) {
                sum += A[i * n + l] * B[l * k + j];
            }
            C[i * k + j] = sum;
```

```
        }
    }
}

/** CUDA kernel for matrix multiplication using shared memory (double) */
__global__ void matrixMultiplyKernelShared(double* A, double* B, double* C, int
m, int n, int k) {
    __shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    double value = 0.0;

    for (int t = 0; t < (n + BLOCK_SIZE - 1) / BLOCK_SIZE; t++) {
        if (row < m && (t * BLOCK_SIZE + threadIdx.x) < n) {
            As[threadIdx.y][threadIdx.x] = A[row * n + t * BLOCK_SIZE +
threadIdx.x];
        } else {
            As[threadIdx.y][threadIdx.x] = 0.0;
        }

        if ((t * BLOCK_SIZE + threadIdx.y) < n && col < k) {
            Bs[threadIdx.y][threadIdx.x] = B[(t * BLOCK_SIZE + threadIdx.y) * k +
col];
        } else {
            Bs[threadIdx.y][threadIdx.x] = 0.0;
        }

        __syncthreads();

        for (int i = 0; i < BLOCK_SIZE; i++) {
            value += As[threadIdx.y][i] * Bs[i][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < m && col < k) {
        C[row * k + col] = value;
    }
}

/** CUDA kernel for matrix multiplication (int) */
```

```
__global__ void matrixMultiplyKernelInt(int* A, int* B, int* C, int m, int n, int
k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < k) {
        int value = 0;
        for (int i = 0; i < n; i++) {
            value += A[row * n + i] * B[i * k + col];
        }
        C[row * k + col] = value;
    }
}

/** Host function for matrix multiplication using CUDA (double) */
void matrixMultiplyCUDA(double* A, double* B, double* C, int m, int n, int k) {
    double *d_A, *d_B, *d_C;

    // Allocate device memory
    cudaError_t err;
    err = cudaMalloc((void**)&d_A, m * n * sizeof(double));
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed for A: %s\n",
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaMalloc((void**)&d_B, n * k * sizeof(double));
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed for B: %s\n",
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaMalloc((void**)&d_C, m * k * sizeof(double));
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed for C: %s\n",
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy matrices from host to device
    cudaMemcpy(d_A, A, m * n * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, n * k * sizeof(double), cudaMemcpyHostToDevice);

    // Define block and grid sizes
    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
```

```
    dim3 gridSize((k + blockSize.x - 1) / blockSize.x, (m + blockSize.y - 1) /
blockSize.y);

    // Launch the kernel
    matrixMultiplyKernelShared<<<gridSize, blockSize>>>(d_A, d_B, d_C, m, n, k);
    cudaDeviceSynchronize();

    // Copy result back to host
    cudaMemcpy(C, d_C, m * k * sizeof(double), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

/** Host function for matrix multiplication using CUDA (int) */
void matrixMultiplyCUDAInt(int* A, int* B, int* C, int m, int n, int k) {
    int *d_A, *d_B, *d_C;

    // Allocate device memory
    cudaError_t err;
    err = cudaMalloc((void**)&d_A, m * n * sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed for A: %s\n",
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaMalloc((void**)&d_B, n * k * sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed for B: %s\n",
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaMalloc((void**)&d_C, m * k * sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed for C: %s\n",
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy matrices from host to device
    cudaMemcpy(d_A, A, m * n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, n * k * sizeof(int), cudaMemcpyHostToDevice);
```

```
    // Define block and grid sizes
    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridSize((k + blockSize.x - 1) / blockSize.x, (m + blockSize.y - 1) /
blockSize.y);

    // Launch the kernel
    matrixMultiplyKernelInt<<<gridSize, blockSize>>>(d_A, d_B, d_C, m, n, k);
    cudaDeviceSynchronize();

    // Copy result back to host
    cudaMemcpy(C, d_C, m * k * sizeof(int), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

/** Main function */
int main(int argc, char** argv) {
    // Command line argument variables
    unsigned int m = 512, n = 512, k = 512, num_runs = 10;

    // Parse command line arguments or use default values
    if (argc > 1) {
        if (sscanf(argv[1], "%u", &m) != 1) {
            printf("Invalid value for m. Using default: m = 512\n");
            m = 512;
        }
    } else {
        printf("Missing value for m. Using default: m = 512\n");
    }

    if (argc > 2) {
        if (sscanf(argv[2], "%u", &n) != 1) {
            printf("Invalid value for n. Using default: n = 512\n");
            n = 512;
        }
    } else {
        printf("Missing value for n. Using default: n = 512\n");
    }

    if (argc > 3) {
        if (sscanf(argv[3], "%u", &k) != 1) {
            printf("Invalid value for k. Using default: k = 512\n");
```

```
            k = 512;
        }
    } else {
        printf("Missing value for k. Using default: k = 512\n");
    }

    if (argc > 4) {
        if (sscanf(argv[4], "%u", &num_runs) != 1) {
            printf("Invalid value for num_runs. Using default: num_runs = 10\n");
            num_runs = 10;
        }
    } else {
        printf("Missing value for num_runs. Using default: num_runs = 10\n");
    }

    // Validate command line arguments
    if (m < 1 || n < 1 || k < 1 || num_runs < 1) {
        fprintf(stderr, "Matrix dimensions and number of runs must be greater
than 0!\n");
        exit(EXIT_FAILURE);
    }

    // Check if the total number of elements is too large
    unsigned long long total_elements = (unsigned long long)m * n * k;
    if (total_elements > MAX_ELEMENTS) {
        unsigned long long estimated_memory = (unsigned long long)(m * n *
sizeof(double) + n * k * sizeof(double) + m * k * sizeof(double));
        double estimated_memory_mb = estimated_memory / (1024.0 * 1024.0);
        double estimated_memory_gb = estimated_memory / (1024.0 * 1024.0 *
1024.0);
        fprintf(stderr, "Warning: The total number of elements (%llu) may be too
large and could lead to a segmentation fault due to", total_elements);
        fprintf(stderr, "\ninsufficient memory! Estimated memory usage: %.2lf MB
(%.2lf GB).\n", estimated_memory_mb, estimated_memory_gb);
    }

    // Allocate memory for matrices (double)
    double* A_d = (double*)malloc(m * n * sizeof(double));
    double* B_d = (double*)malloc(n * k * sizeof(double));
    double* C_serial_d = (double*)malloc(m * k * sizeof(double));
    double* C_parallel_d = (double*)malloc(m * k * sizeof(double));

    if (!A_d || !B_d || !C_serial_d || !C_parallel_d) {
        fprintf(stderr, "Matrix memory allocation failed for double
matrices!\n");
```

```
        exit(EXIT_FAILURE);
    }

    // Allocate memory for matrices (int)
    int* A_i = (int*)malloc(m * n * sizeof(int));
    int* B_i = (int*)malloc(n * k * sizeof(int));
    int* C_serial_i = (int*)malloc(m * k * sizeof(int));
    int* C_parallel_i = (int*)malloc(m * k * sizeof(int));

    if (!A_i || !B_i || !C_serial_i || !C_parallel_i) {
        fprintf(stderr, "Matrix memory allocation failed for int matrices!\n");
        exit(EXIT_FAILURE);
    }

    // Fill matrices with random values
    seed_random();
    for (int i = 0; i < m * n; i++) {
        A_d[i] = random_double();
        A_i[i] = random_int();
    }
    for (int i = 0; i < n * k; i++) {
        B_d[i] = random_double();
        B_i[i] = random_int();
    }

    // Variables for tracking time
    double total_parallel_time_d = 0;
    double total_serial_time_d = 0;
    double total_parallel_time_i = 0;
    double total_serial_time_i = 0;

    // Perform the matrix multiplication for num_runs (double)
    for (unsigned int run = 0; run < num_runs; run++) {
        Stopwatch parallel_timer, serial_timer;

        // Start parallel timing (double)
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.start);

        // Perform parallel matrix multiplication (double)
        matrixMultiplyCUDA(A_d, B_d, C_parallel_d, m, n, k);

        // Stop parallel timing (double)
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.stop);
        total_parallel_time_d += calculate_time(parallel_timer);
```

```
        // Start serial timing (double)
        clock_gettime(CLOCK_MONOTONIC, &serial_timer.start);

        // Perform serial matrix multiplication (double)
        matrixMultiplySerial(A_d, B_d, C_serial_d, m, n, k);

        clock_gettime(CLOCK_MONOTONIC, &serial_timer.stop);
        total_serial_time_d += calculate_time(serial_timer);

        // Compare results (double)
        if (!compare_matrices(C_serial_d, C_parallel_d, m, k)) {
            fprintf(stderr, "Error: Double matrices do not match in run %u!\n",
run + 1);
        }
    }

    // Perform the matrix multiplication for num_runs (int)
    for (unsigned int run = 0; run < num_runs; run++) {
        Stopwatch parallel_timer, serial_timer;

        // Start parallel timing (int)
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.start);

        // Perform parallel matrix multiplication (int)
        matrixMultiplyCUDAInt(A_i, B_i, C_parallel_i, m, n, k);

        // Stop parallel timing (int)
        clock_gettime(CLOCK_MONOTONIC, &parallel_timer.stop);
        total_parallel_time_i += calculate_time(parallel_timer);

        // Start serial timing (int)
        clock_gettime(CLOCK_MONOTONIC, &serial_timer.start);

        // Perform serial matrix multiplication (int)
        matrixMultiplySerialInt(A_i, B_i, C_serial_i, m, n, k);

        clock_gettime(CLOCK_MONOTONIC, &serial_timer.stop);
        total_serial_time_i += calculate_time(serial_timer);

        // Compare results (int)
        if (memcmp(C_serial_i, C_parallel_i, m * k * sizeof(int)) != 0) {
            fprintf(stderr, "Error: Integer matrices do not match in run %u!\n",
run + 1);
        }
    }
```

```
    // Calculate average times
    double average_parallel_time_d = total_parallel_time_d / num_runs;
    double average_serial_time_d = total_serial_time_d / num_runs;
    double average_parallel_time_i = total_parallel_time_i / num_runs;
    double average_serial_time_i = total_serial_time_i / num_runs;

    // Output results and speedup calculations (double)
    double actual_speedup_d = average_serial_time_d / average_parallel_time_d;
    double theoretical_speedup_d = amdahl_speedup(m,
PARALLELIZABLE_FRACTION_DOUBLE);
    double speedup_ratio_d = (actual_speedup_d / theoretical_speedup_d) * 100;

    printf("Double Matrix Multiplication:\n");
    printf("Average Serial Time:\t\t\t%lfs\n", average_serial_time_d);
    printf("Average Parallel Time:\t\t\t%lfs\n", average_parallel_time_d);
    printf("Theoretical Speedup [Amdahl's Law]:\t%lf\n", theoretical_speedup_d);
    printf("Actual Speedup:\t\t\t\t%lf\n", actual_speedup_d);
    printf("Speedup Efficiency:\t\t\t%lf%%\n", speedup_ratio_d);

    // Output results and speedup calculations (int)
    double actual_speedup_i = average_serial_time_i / average_parallel_time_i;
    double theoretical_speedup_i = amdahl_speedup(m,
PARALLELIZABLE_FRACTION_INTEGER);
    double speedup_ratio_i = (actual_speedup_i / theoretical_speedup_i) * 100;

    printf("\nInteger Matrix Multiplication:\n");
    printf("Average Serial Time:\t\t\t%lfs\n", average_serial_time_i);
    printf("Average Parallel Time:\t\t\t%lfs\n", average_parallel_time_i);
    printf("Theoretical Speedup [Amdahl's Law]:\t%lf\n", theoretical_speedup_i);
    printf("Actual Speedup:\t\t\t\t%lf\n", actual_speedup_i);
    printf("Speedup Efficiency:\t\t\t%lf%%\n", speedup_ratio_i);

    printf("\nDifference in Average Serial Time (Double vs Integer):\t\t%lfs\n",
average_serial_time_d - average_serial_time_i);
    printf("Difference in Average Parallel Time (Double vs Integer):\t%lfs\n",
average_parallel_time_d - average_parallel_time_i);
    printf("Difference in Theoretical Speedup (Double vs Integer):\t\t%lf\n",
theoretical_speedup_d - theoretical_speedup_i);
    printf("Difference in Actual Speedup (Double vs Integer):\t\t%lf\n",
actual_speedup_d - actual_speedup_i);
    printf("Difference in Speedup Efficiency (Double vs Integer):\t\t%lf%%\n",
speedup_ratio_d - speedup_ratio_i);
```

```
    // Free allocated memory
    free(A_d);
    free(B_d);
    free(C_serial_d);
    free(C_parallel_d);
    free(A_i);
    free(B_i);
    free(C_serial_i);
    free(C_parallel_i);

    return 0;
}
```

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.