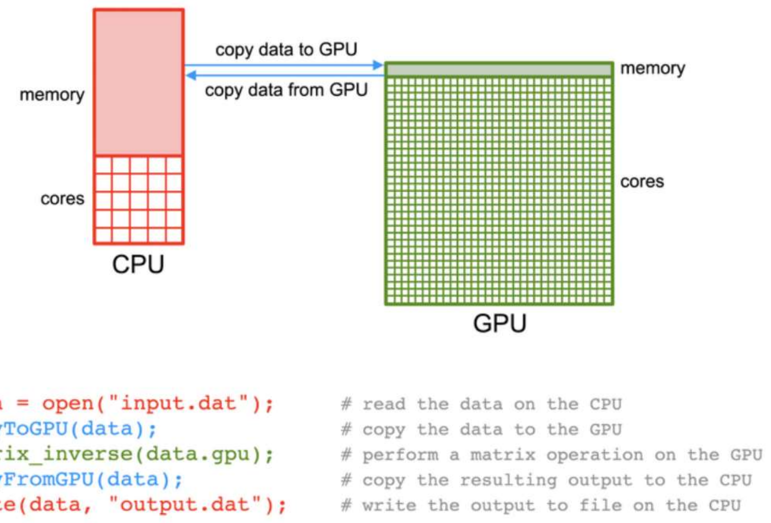
Coursework 4 – Evaluation of an Integral using Midpoint Rule

Damir Kadyrzhan 20494312

Introduction

Graphics Processing Unit (GPU) computing is vital in accelerating large datasets within hardware. While Central Processing Units (CPUs) comprise of one or tens of processing cores, GPUs have thousands of cores for large scalable computing. These processing cores can be used in parallel for specific loads, such as training artificial neural networks, image processing, and solving differential equations. GPUs are particularly useful in computing large-scale operations that can happen simultaneously [1].

One example is the computation of a large 2-dimensional inverse of the matrix. This kind of process can be executed using a CPU; however, the computation speed would be limited by its capacity. A GPU can execute this process in parallel, hence making it faster.



```
data = open("input.dat");        # read the data on the CPU
copyToGPU(data);                 # copy the data to the GPU
matrix_inverse(data.gpu);        # perform a matrix operation on the GPU
copyFromGPU(data);               # copy the resulting output to the CPU
write(data, "output.dat");       # write the output to file on the CPU
```

Figure 1. Numerical Computation of Inverse Matrix [1]

There are usually three steps to execute a kernel on a GPU in code:

1. Send input data from CPU memory to the GPU memory
2. Load and execute the GPU kernel on the GPU
3. Send the result back to CPU memory

Figure 1 shows highlights of this function, whereby the CPU reads the matrix before transferring the necessary data to the GPU for every iteration of the inverse matrix. The computational result from the GPU is sent back to the CPU, where it performs further operations. The GPU is an accelerator that can be used with a CPU to quickly carry large numerical-intensive tasks.

Task 1. Numerical Integration and Problem Analysis

Numerical integration defines the use of numerical techniques to calculate the approximate value of an integral. It is also a method of calculating or obtaining a definite integral $\int_a^b f(x)dx$ from a set of numerical values of the integrand $f(x)$; finding the centre of mass of an object, calculating how much work is required to move an object to a certain speed, or determining how much fluid is flowing through a specific area are all applications of numerical integration. Some applications are difficult to solve by hand, which is where numerical computing becomes invaluable [2].

The numerical integration method demonstrated in this paper is the midpoint rule. The midpoint rule estimates the integral of a function or area under a curve by dividing the area into rectangles or segments of equal width. This is shown in Figure 2.

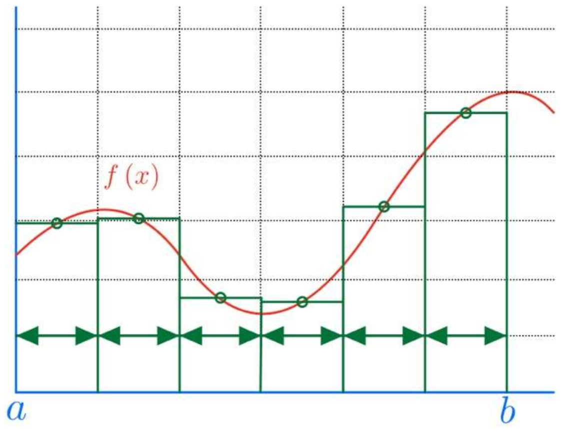


Figure 2. Midpoint Rule Visual Representation

The midpoint formula is generally given as follows:

$$\int_a^b f(x)dx = \sum_{i=1}^{n} f(a + (i - \frac{1}{2})\Delta)\Delta$$

Where i is the rectangle index, n is the number of rectangles under the curve, f is the function of the curve evaluated at the midpoint, and Δ is the rectangle width, defined as:

$$\Delta = \frac{b-a}{n}$$

Where a is lower bound, b is upper bound and n is the number of rectangles.

Equation to evaluate using midpoint rule is given as:

$$f(x) = e^{-x^2}$$

Where e^{-x^2} is the Gaussian function and it is equal to $\sqrt{\pi}$.

Given upper and lower bounds as -2 to 2, the function can be solved as following:

$$\Delta = \frac{b-a}{n} = \frac{2-(-2)}{4} = 1$$

$$\int_{-2}^{2} e^{-x^2}dx \approx f\left(-2 + \left(1 - \frac{1}{2}\right) \times 1\right) + f\left(-2 + \left(2 - \frac{1}{2}\right) \times 1\right) + f\left(-2 + \left(3 - \frac{1}{2}\right) \times 1\right) + f\left(-2 + \left(4 - \frac{1}{2}\right) \times 1\right)$$

$$\int_{-2}^{2} e^{-x^2}dx \approx f(-1.5) + f(-0.5) + f(0.5) + f(1.5)$$

$$\int_{-2}^{2} e^{-x^2}dx \approx e^{-2.25} + e^{-0.25} + e^{-0.25} + e^{-2.25}$$

$$\int_{-2}^{2} e^{-x^2}dx \approx 0.105 + 0.78 + 0.78 + 0.105 = 1.77$$

The example above was evaluated by hand using four rectangles. The number of rectangles determine how precise the resultant area is, i.e., the larger the sample size, the greater the resolution. In this paper, the number of rectangles will be varied to highlight the computational efficiency of CPU and GPU with large datasets.

Sets up one time, can be done in CPU

$$\boxed{\Delta = \frac{b-a}{n} = \frac{2-(-2)}{4} = 1}$$

Repeated large dataset that can be parallelised

$$\int_{-2}^{2} e^{-x^2}dx \approx f\left(-2+\left(1-\frac{1}{2}\right)\times 1\right) + f\left(-2+\left(2-\frac{1}{2}\right)\times 1\right) + f\left(-2+\left(3-\frac{1}{2}\right)\times 1\right) + f\left(-2+\left(4-\frac{1}{2}\right)\times 1\right)$$

$$\int_{-2}^{2} e^{-x^2}dx \approx f(-1.5) + f(-0.5) + f(0.5) + f(1.5)$$

$$\int_{-2}^{2} e^{-x^2}dx \approx e^{-2.25} + e^{-0.25} + e^{-0.25} + e^{-2.25}$$

Summation of a vector to itself can only be done in series

$$\boxed{\int_{-2}^{2} e^{-x^2}dx \approx 0.105 + 0.78 + 0.78 + 0.105 = 1.77}$$

Figure 3. Data that can be parallelised in GPU

Task 2. CPU Code

Various tests have been conducted to validate the performance of the implemented evaluation of an integral using midpoint rule on CPUs against GPUs with different settings of blocks and threads. The rest of the sections will present the findings from these tests, along with overall final CPU/GPU performance evaluation.

The first tests conducted highlight CPU performance of the realised integral evaluation using midpoint rule. Figures 4 and 5 show the results.

Table 1. CPU Benchmark Results

Computation Time (ms)	Number of Intervals (n)	Value of Integration
0.2	4	1.768
0.2	8	1.765
0.2	16	1.764
0.2	32	1.764
0.2	64	1.764
0.2	128	1.764
0.3	256	1.764
0.3	512	1.764
0.3	1024	1.764
0.4	2048	1.764
0.5	4096	1.764
0.9	8192	1.764
1.7	16384	1.764
3.5	32768	1.764
6.6	65500	1.764

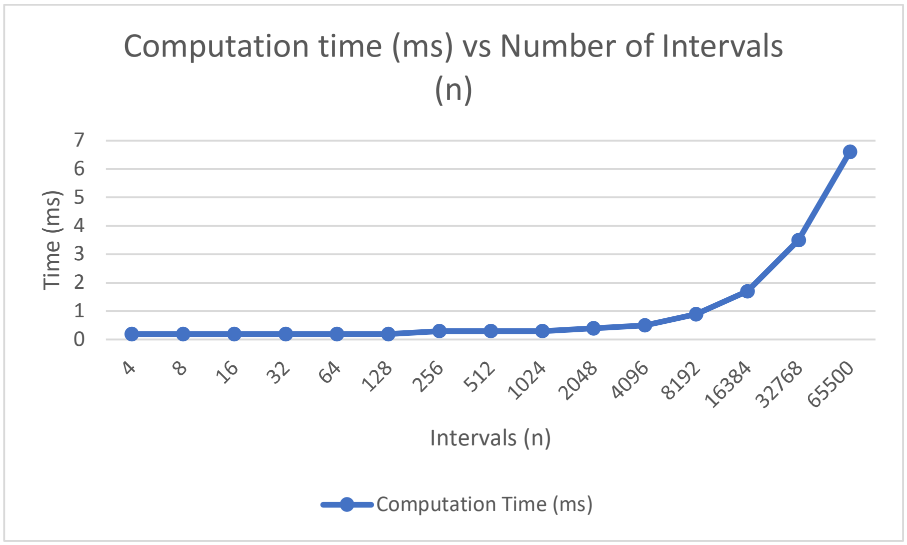


Figure 4. Computation Time against Number of Rectangles Graph

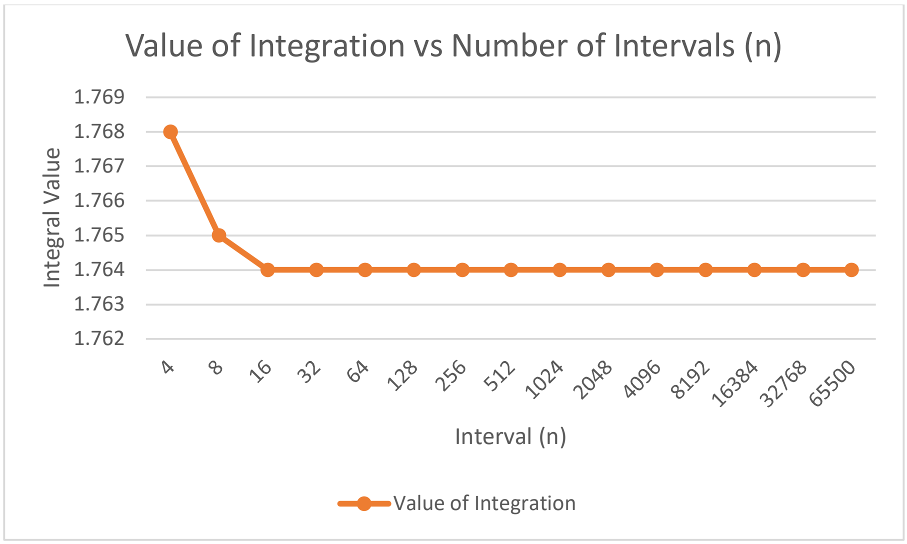
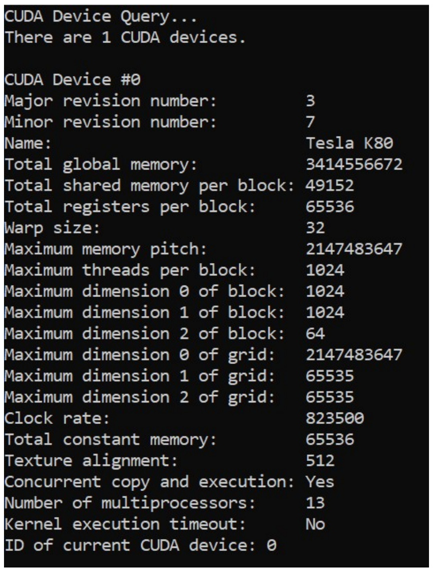


Figure 5. Value of Integral Against Number of Rectangles

Task 3. GPU Properties


```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         3
Minor revision number:         7
Name:                          Tesla K80
Total global memory:           3414556672
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    823500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
ID of current CUDA device: 0
```

Figure 6. GPU Properties

Table 2. Important Properties for this Coursework

Device Name	Tesla K80
Total Registers Per Block	65536
Maximum Threads Per Block	1024
Total Constant Memory	65536

Task 4. GPU Code

Task 4.1. Distribution amongst blocks only

Following the same integration method, the GPU performance was evaluated using only blocks. Table 3 highlights the comparison between CPU and GPU performance in a range of numeric intervals, from 4 to 65,500.

Table 3. GPU Benchmark with Block Only

CPU Computation Time (ms)	GPU Computation Time (ms)	Number of Intervals (n)	Value of Integration	Speed up Achieved CPU Time/GPU Time
0.2	0.5	4	1.768	0.4
0.2	0.6	8	1.764	0.3
0.2	0.5	16	1.764	0.4
0.2	0.5	32	1.764	0.4
0.2	0.5	64	1.764	0.4
0.2	0.5	128	1.764	0.4
0.3	0.5	256	1.764	0.6
0.3	0.5	512	1.764	0.6
0.3	0.6	1024	1.764	0.5

0.4	0.6	2048	1.764	0.6
0.5	0.7	4096	1.764	0.7
0.9	0.9	8192	1.764	1
1.7	2	16384	1.764	0.85
3.5	2.6	32768	1.764	1.35
6.6	3.7	65500	1.764	1.78

Analysis of the acquired results shows the GPU algorithm outperforms the CPU algorithm as the number of intervals increase. This result supports the benefits of a GPUs larger core count comparative to the CPU, thus allowing for the distribution of the workload. This distribution has resulted in a gradual improvement in computational timing, as highlighted in Figures 7 to 9.

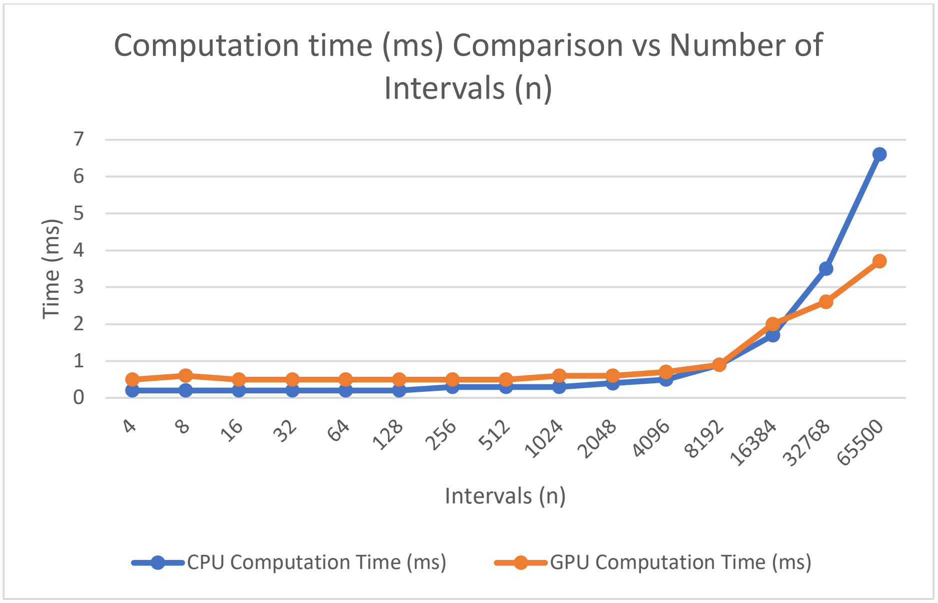


Figure 7. GPU/CPU Computation Time Comparison

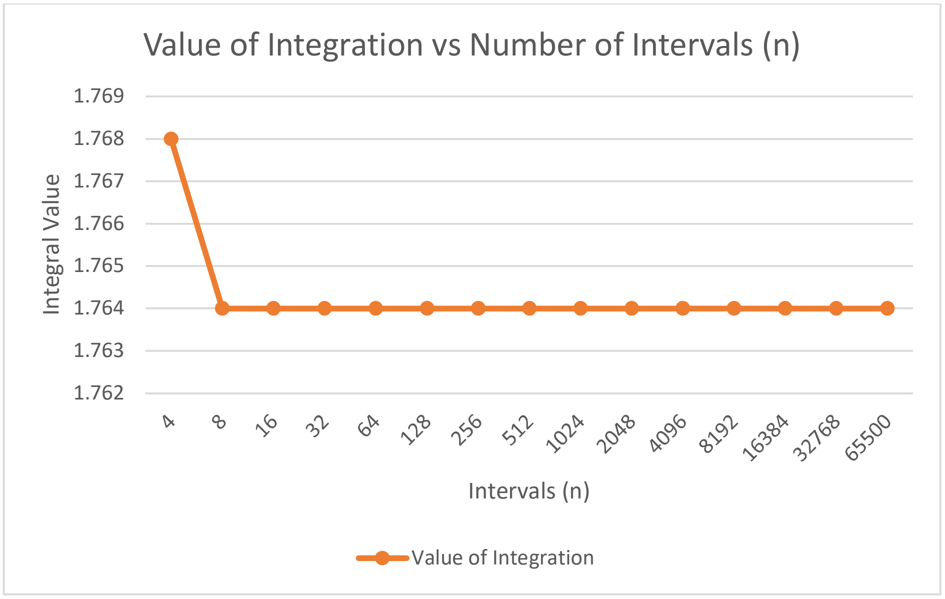


Figure 8. Integration Value against Number of Intervals

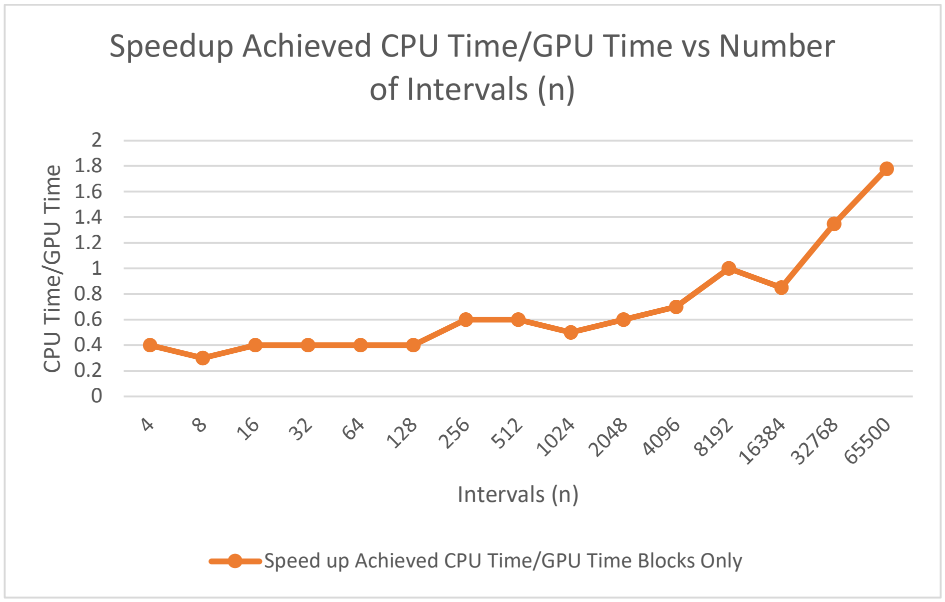


Figure 9. Speedup Achieved using GPU Blocks Only

Task 4.2. Distribution amongst threads only

A timing performance comparison was conducted between the CPU algorithm and the GPU CUDA algorithm using only threads. Various numeric intervals were used to conduct the timing. Table 4 shows these results. For each of the 1024 threads, 1 block has been used to perform larger operations, due to the limitation of the GPU having only 1024 threads per block.

Table 4. GPU Benchmark with Threads Only

CPU Computation Time (ms)	GPU Computation Time (ms)	Number of Intervals (n)	Value of Integration	Speed up Achieved CPU Time/GPU Time
0.2	0.6	4	1.768	0.3
0.2	0.6	8	1.764	0.3
0.2	0.5	16	1.764	0.4
0.2	0.5	32	1.764	0.4
0.2	0.5	64	1.764	0.4
0.2	0.5	128	1.764	0.4
0.3	0.5	256	1.764	0.6
0.3	0.5	512	1.764	0.6
0.3	0.5	1024	1.764	0.6
0.4	0.5	2048	1.764	0.8
0.5	0.5	4096	1.764	1
0.9	0.5	8192	1.764	1.8
1.7	0.7	16384	1.764	2.4
3.5	0.7	32768	1.764	5
6.6	1	65500	1.764	6.6

The timing performance is significantly better than the GPU algorithm using only blocks as shown in Figures 10-12. Each block runs on a streaming multiprocessor (SM), each of which running multiple threads. These threads are comparable to logical cores that can perform the tasks faster.

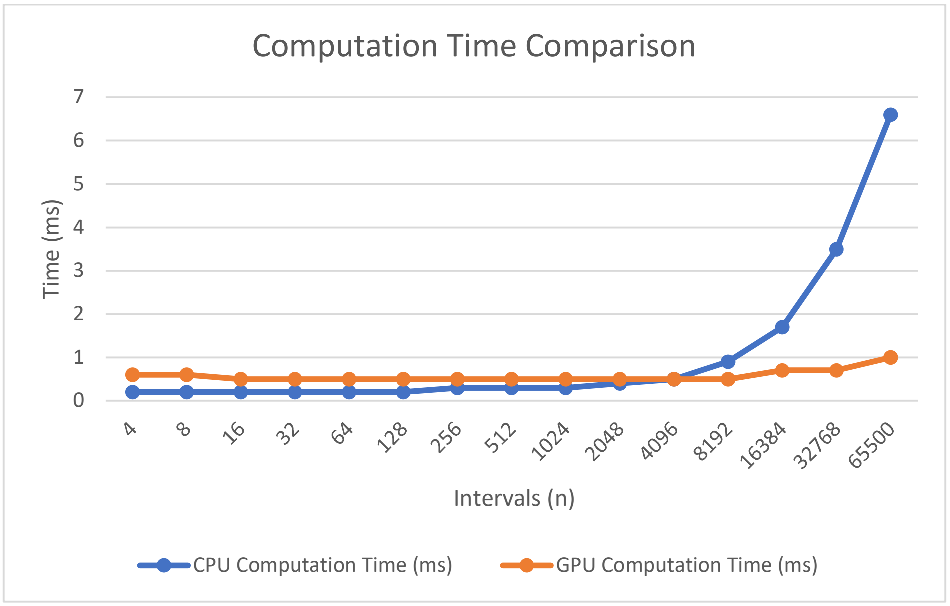


Figure 10. CPU/GPU Computation Time Comparison

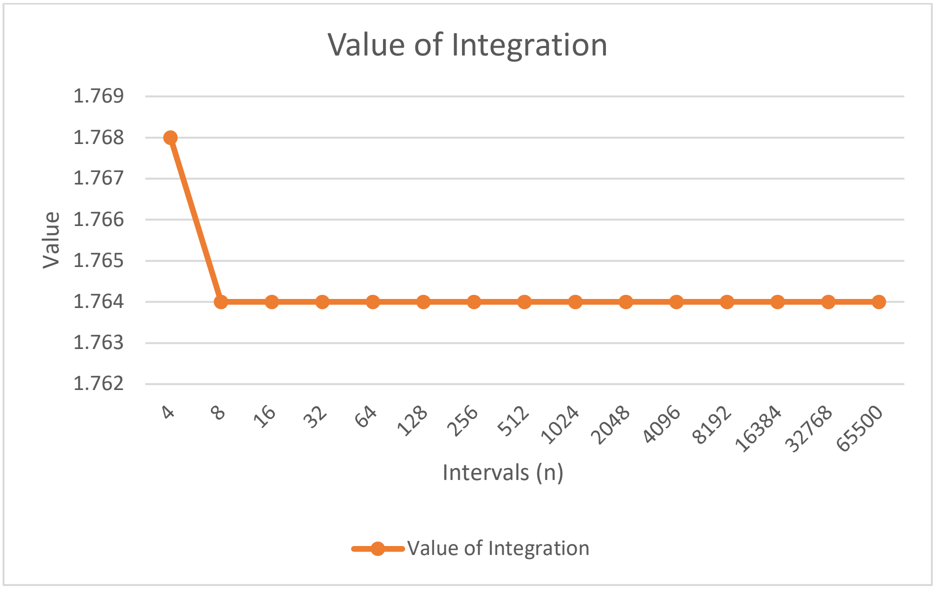


Figure 11. Value of Integration against Number of Intervals

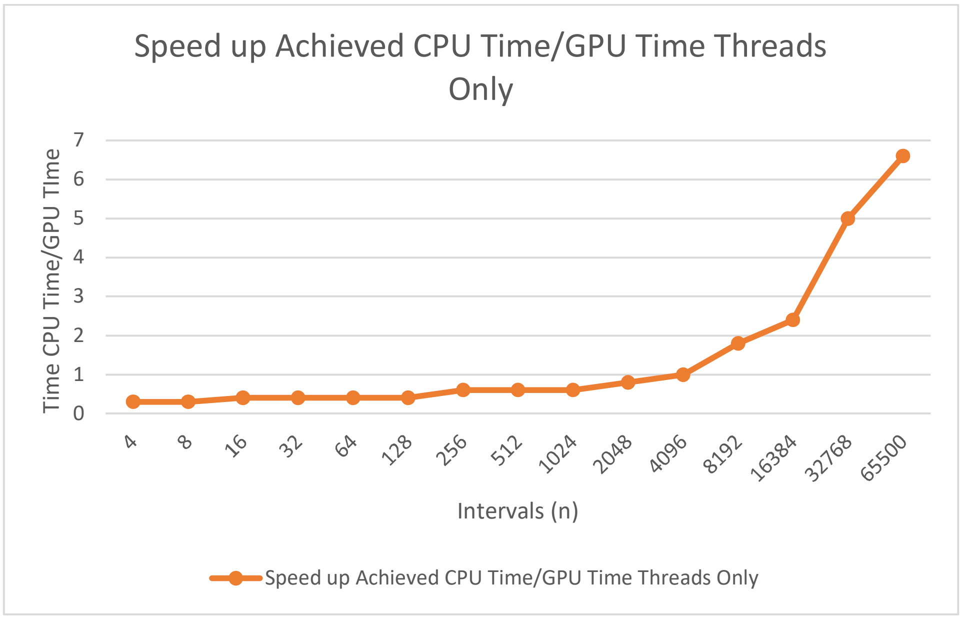


Figure 12. Speedup Achieved using GPU Threads Only

<u>Task 4.3. Distribution amongst blocks and threads</u>

Following the same GPU CUDA algorithm, the timing performance of blocks and threads together was evaluated against CPU timings, highlighted in Table 5.

Table 5. GPU Benchmark with Blocks and Threads

CPU Computation Time (ms)	GPU Computation Time (ms)	Number of Intervals (n)	Value of Integration	Speed up Achieved CPU Time/GPU Time
0.2	0.4	4	1.768	0.5
0.2	0.4	8	1.764	0.5
0.2	0.5	16	1.764	0.4
0.2	0.5	32	1.764	0.4
0.2	0.5	64	1.764	0.4
0.2	0.5	128	1.764	0.4
0.3	0.5	256	1.764	0.6
0.3	0.5	512	1.764	0.6
0.3	0.5	1024	1.764	0.6
0.4	0.5	2048	1.764	0.8
0.5	0.5	4096	1.764	1
0.9	0.5	8192	1.764	1.8
1.7	0.6	16384	1.764	2.83
3.5	0.6	32768	1.764	5.83
6.6	0.9	65500	1.764	7.3

From the acquired results, the timing of the GPU with blocks and threads outperforms every algorithm method across the end range of the data.

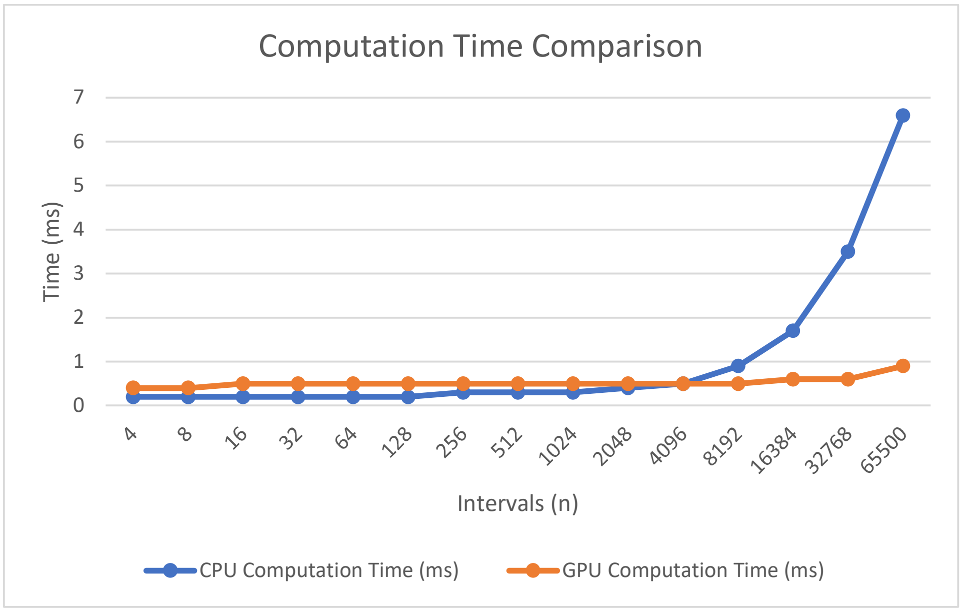


Figure 13. Computation Time Comparison GPU with Blocks and Threads against CPU

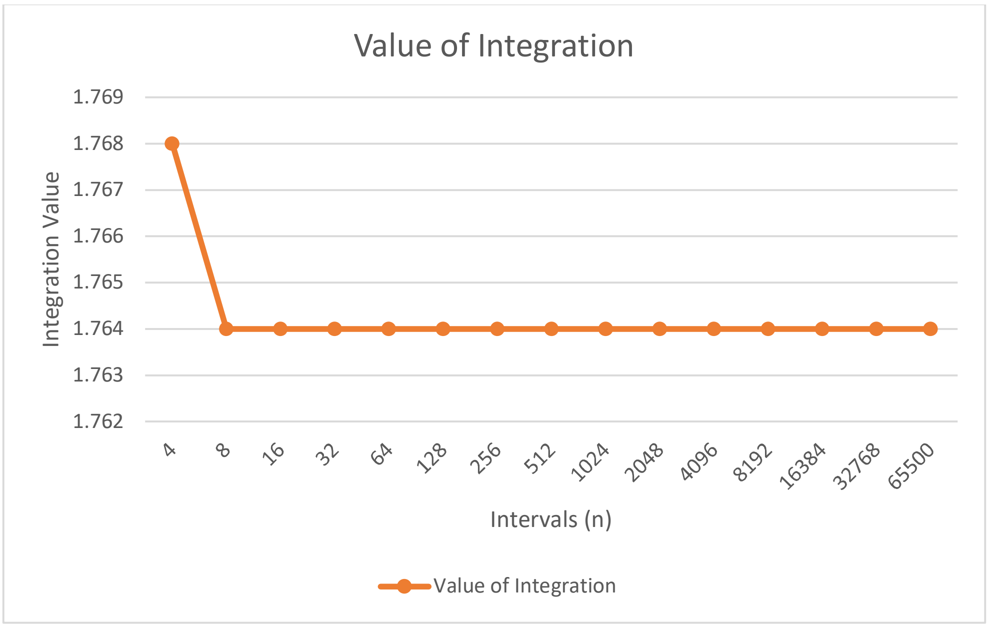


Figure 14. Value of Integration against Number of Intervals

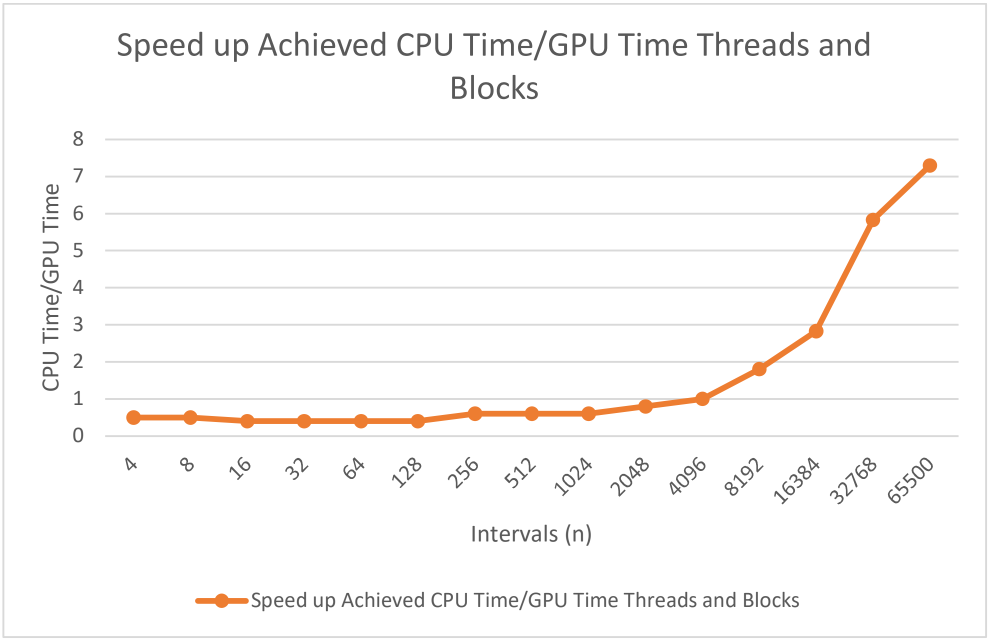


Figure 15. Speedup Achieved using GPU Blocks and Threads

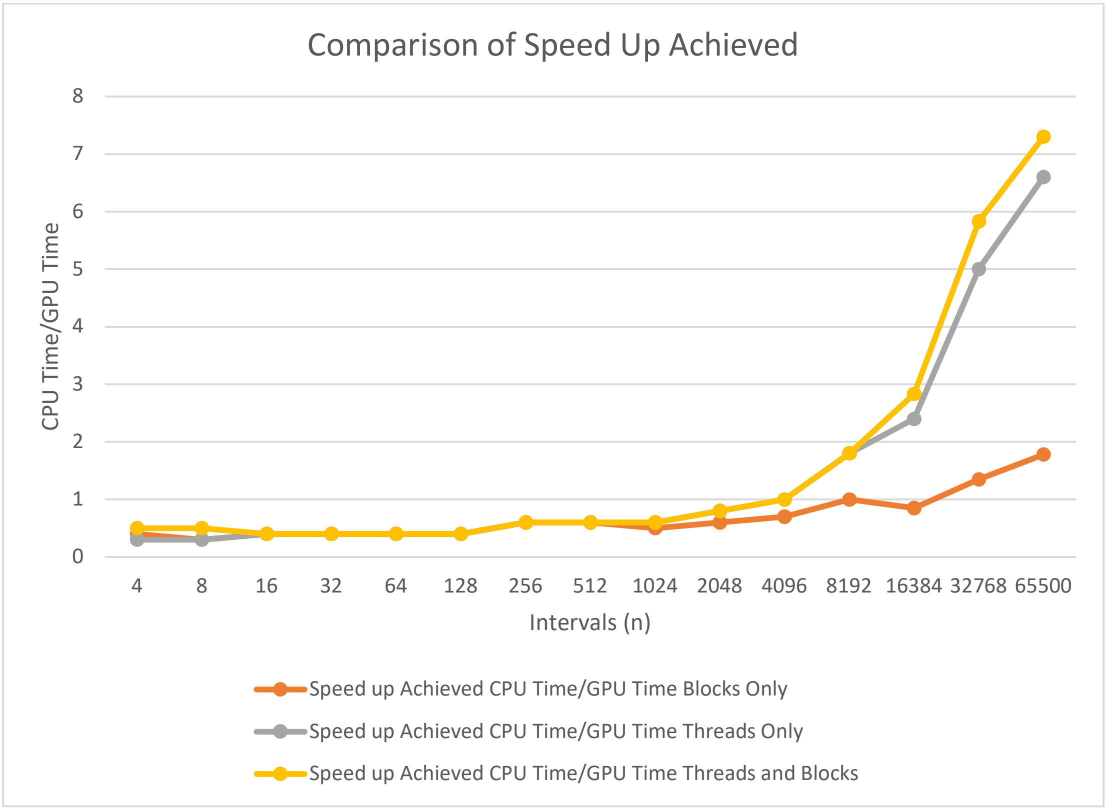


Figure 16: Speedup Achieved Comparison

Conclusion

This paper aimed to investigate the efficiency of GPUs and CPUs in evaluating an integral using the midpoint rule. A comprehensive GPU algorithm has been presented alongside its implementation within CUDA C. In accordance with the coursework aims, the GPU has shown a significant boost in performance with larger datasets. This was further investigated by incorporating benchmark results that have shown a computational efficiency element. The results acquired highlight CPU and GPU performances on midpoint rule integration. The GPU algorithms presented have shown enhanced speed and efficiency over the CPU counterpart with a higher number of calculations.

References

[1] "GPU Computing," *Princeton University*, 2022. https://researchcomputing.princeton.edu/support/knowledge-base/gpu-computing

[2] A. Barta, "Midpoint Rule - Formula & Examples," 2022. https://study.com/learn/lesson/midpoint-rule-formula-examples.html

Appendix

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main() {

    int a = -2; // Lower bound
    int b = 2; // Upper bound
    double n = 256; // Number of subintervals

    double delta = (b - a) / n; // Width of each subinterval
    double sum = 0;
    double x; // Function variable

    // Start Measure Timing
    double start = omp_get_wtime();

    int i;
    for (i = 1; i <= n; i++) {
        x = a + (i - 0.5) * delta; // midpoint of the i-th subinterval

        x = exp(-(pow(x, 2))); // given exponential function

        //printf("%f \n", x); // print the value of subintervals
        sum += (x * delta);  // multiply the sum by the width of the subintervals to get the 
approximation of integral

    }
    printf("Integral Value: %f \n", sum);

    // Measure Timing
    double end = omp_get_wtime();
    printf("start = %.16g\nend = %.16g\ndiff = %.16g\n", start, end, end - start); //in secs

    return 0;
}
```

Code 1. CPU Code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

// Print device properties
void printDevProp(cudaDeviceProp devProp)
{
    printf("Major revision number:         %d\n", devProp.major);
    printf("Minor revision number:         %d\n", devProp.minor);
    printf("Name:                          %s\n", devProp.name);
    printf("Total global memory:           %u\n", devProp.totalGlobalMem);
    printf("Total shared memory per block: %u\n", devProp.sharedMemPerBlock);
    printf("Total registers per block:     %d\n", devProp.regsPerBlock);
    printf("Warp size:                     %d\n", devProp.warpSize);
    printf("Maximum memory pitch:          %u\n", devProp.memPitch);
    printf("Maximum threads per block:     %d\n", devProp.maxThreadsPerBlock);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of block:  %d\n", i, devProp.maxThreadsDim[i]);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
    printf("Clock rate:                    %d\n", devProp.clockRate);
    printf("Total constant memory:         %u\n", devProp.totalConstMem);
    printf("Texture alignment:             %u\n", devProp.textureAlignment);
    printf("Concurrent copy and execution: %s\n", (devProp.deviceOverlap ? "Yes" : "No"));
    printf("Number of multiprocessors:     %d\n", devProp.multiProcessorCount);
    printf("Kernel execution timeout:      %s\n", (devProp.kernelExecTimeoutEnabled ? "Yes" : "No"));
    return;
}

int main()
{
    // Number of CUDA devices
    int devCount;
    cudaGetDeviceCount(&devCount);
    printf("CUDA Device Query...\n");
    printf("There are %d CUDA devices.\n", devCount);

    // Iterate through devices
    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printDevProp(devProp);
    }

    cudaDeviceProp prop;
    int device;
    cudaGetDevice(&device);
    printf("ID of current CUDA device: %d \n", device);

    return 0;
}
```

Code 2. GPU Properties Code

```
#include <stdio.h>
#include <math.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <omp.h>


#define BLOCK_SIZE 256  // define the block size


__global__ void midpoint_rule_kernel(double a, double b, int n, double* result)
{
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x; // index of the current thread
    double delta = (b - a) / n; // width of each subinterval

    if (tid < n) { // check if the index is within the number of subintervals

        double x = exp(-(pow(a + delta * (tid + 0.5), 2))); // midpoint of tid-th subinterval
        result[tid] = x; // store the value of the function in result array

    }
}

int main()
{
    // Host Input
    double a = -2; // lower bound
    double b = 2;  // upper bound
    int n = 65500; // number of subintervals

    double* d_result; // device array to store the result of the kernel function

    // Number of blocks needed to process all subintervals
    int num_threads = (n + BLOCK_SIZE - 1) / BLOCK_SIZE; // number of threads needed to process all
subintervals
    size_t size = n * sizeof(double); // size of result array

    // Device Output
    double* h_result; // host array to store the result of the kernel function

    cudaDeviceSynchronize();

    h_result = (double*)malloc(size); // allocate memory for each vector on GPU

    // Allocate memory on GPU and copy result array from the host
    cudaMalloc((void**)&d_result, size);
    cudaMemcpy(d_result, h_result, size, cudaMemcpyHostToDevice);


    double start = omp_get_wtime(); // Start Timing


    // Launch kernel function
    midpoint_rule_kernel <<< BLOCK_SIZE, 1 >>> (a, b, n, d_result);
    cudaDeviceSynchronize();

    // Copy the result array from the device to the host
    cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);

    // Free the device memory
    cudaFree(d_result);


    // Compute the sum of the values in the result array
    double sum = 0;
    for (int i = 0; i < n; i++) {
        //printf("Operation %d: %f \n", i, h_result[i]);
        sum += h_result[i];
    }

    // Approximation of the integral using midpoint rule
    double integral = sum * (b - a) / n;
    printf("The approximate value of the integral is: %f \n", integral);

    free(h_result);

    double end = omp_get_wtime(); // End Timing
    printf("start = %.16g\nend = %.16g\ndiff = %.16g\n", start, end, end - start); //in secs

    return 0;
}
```

Code 3. GPU Code (Blocks Only)

```
#include <stdio.h>
#include <math.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <omp.h>


#define BLOCK_SIZE 256  // define the block size

__global__ void midpoint_rule_kernel(double a, double b, int n, double* result)
{
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x; // index of the current thread
    double delta = (b - a) / n; // width of each subinterval

    if (tid < n) { // check if the index is within the number of subintervals
```

```
        double x = exp(-(pow(a + delta * (tid + 0.5), 2))); // midpoint of tid-th subinterval
        result[tid] = x; // store the value of the function in result array

    }
}

int main()
{
    // Host Input
    double a = -2; // lower bound
    double b = 2;  // upper bound
    int n = 65500; // number of subintervals

    double* d_result; // device array to store the result of the kernel function

    // Number of blocks needed to process all subintervals
    int num_threads = (n + BLOCK_SIZE - 1) / BLOCK_SIZE; // number of threads needed to process all
subintervals
    size_t size = n * sizeof(double); // size of result array

    // Device Output
    double* h_result; // host array to store the result of the kernel function

    cudaDeviceSynchronize();

    h_result = (double*)malloc(size); // allocate memory for each vector on GPU

    // Allocate memory on GPU and copy result array from the host
    cudaMalloc((void**)&d_result, size);
    cudaMemcpy(d_result, h_result, size, cudaMemcpyHostToDevice);

    double start = omp_get_wtime(); // Start Timing

    // Launch kernel function
    midpoint_rule_kernel <<< 1, num_threads >>> (a, b, n, d_result);
    cudaDeviceSynchronize();

    // Copy the result array from the device to the host
    cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);

    // Free the device memory
    cudaFree(d_result);


    // Compute the sum of the values in the result array
    double sum = 0;
    for (int i = 0; i < n; i++) {
        //printf("Operation %d: %f \n", i, h_result[i]);
        sum += h_result[i];
    }


    // Approximation of the integral using midpoint rule
    double integral = sum * (b - a) / n;
    printf("The approximate value of the integral is: %f \n", integral);

    free(h_result);

    double end = omp_get_wtime(); // End Timing
    printf("start = %.16g\nend = %.16g\ndiff = %.16g\n", start, end, end - start); //in secs

    return 0;
}
```

Code 4. GPU Code (Threads Only)

```
#include <stdio.h>
#include <math.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <omp.h>
```

```
#define BLOCK_SIZE 256  // define the block size


__global__ void midpoint_rule_kernel(double a, double b, int n, double* result)
{
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x; // index of the current thread
    double delta = (b - a) / n; // width of each subinterval

    if (tid < n) { // check if the index is within the number of subintervals

        double x = exp(-(pow(a + delta * (tid + 0.5), 2))); // midpoint of tid-th subinterval
        result[tid] = x; // store the value of the function in result array

    }
}


int main()
{
    // Host Input
    double a = -2; // lower bound
    double b = 2;  // upper bound
    int n = 65500; // number of subintervals

    double* d_result; // device array to store the result of the kernel function

    // Number of blocks needed to process all subintervals
    int num_threads = (n + BLOCK_SIZE - 1) / BLOCK_SIZE; // number of threads needed to process all
subintervals
    size_t size = n * sizeof(double); // size of result array

    // Device Output
    double* h_result; // host array to store the result of the kernel function

    cudaDeviceSynchronize();

    h_result = (double*)malloc(size); // allocate memory for each vector on GPU

    // Allocate memory on GPU and copy result array from the host
    cudaMalloc((void**)&d_result, size);
    cudaMemcpy(d_result, h_result, size, cudaMemcpyHostToDevice);


    double start = omp_get_wtime(); // Start Timing


    // Launch kernel function
    midpoint_rule_kernel <<< BLOCK_SIZE, num_threads >>> (a, b, n, d_result);
    cudaDeviceSynchronize();

    // Copy the result array from the device to the host
    cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost);

    // Free the device memory
    cudaFree(d_result);


    // Compute the sum of the values in the result array
    double sum = 0;
    for (int i = 0; i < n; i++) {
        //printf("Operation %d: %f \n", i, h_result[i]);
        sum += h_result[i];
    }


    // Approximation of the integral using midpoint rule
    double integral = sum * (b - a) / n;
    printf("The approximate value of the integral is: %f \n", integral);

    free(h_result);

    double end = omp_get_wtime(); // End Timing
    printf("start = %.16g\nend = %.16g\ndiff = %.16g\n", start, end, end - start); //in secs


    return 0;
}
```

Code 5. GPU Code (Blocks and Threads)