EC-444 Parallel & Distributed Computing

BS (CE)-2020

Semester Project Report

Submitted By:

Reg. No.	Name
20-CE-035	Abdullah Javed

Department of Computer Engineering

HITEC University Taxila

Submitted To:

Engr. Fasih Ahmad

Contents

Table of Figures

Implementation of the 2D-Convolution in CUDA C

Problem Statement:

Convolution is the most important and fundamental concept in signal processing and analysis. Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. Mathematically, it describes a rule of how to combine two functions or pieces of information to form a third function. The feature map (or input data) and the kernel are combined to form a transformed feature map. The convolution algorithm is often interpreted as a filter, where the kernel filters the feature map for certain information.

Tools Used:

- Linux Ubuntu
- GCC/G++ Compiler
- OpenMP
- CUDA
- “Timer.h” (For Profiling)

Algorithm:

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

x

1	0	1
0	1	0
0	0	1

h

Figure 1 image matrix x and kernel h

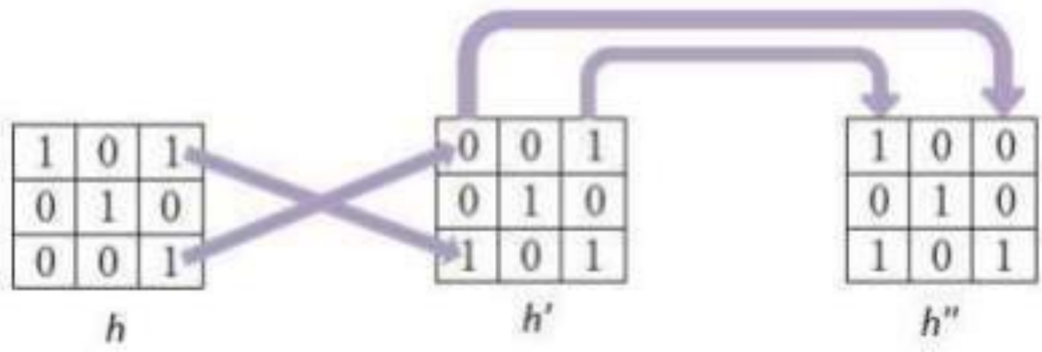


Figure 2 Inversion of h matrix

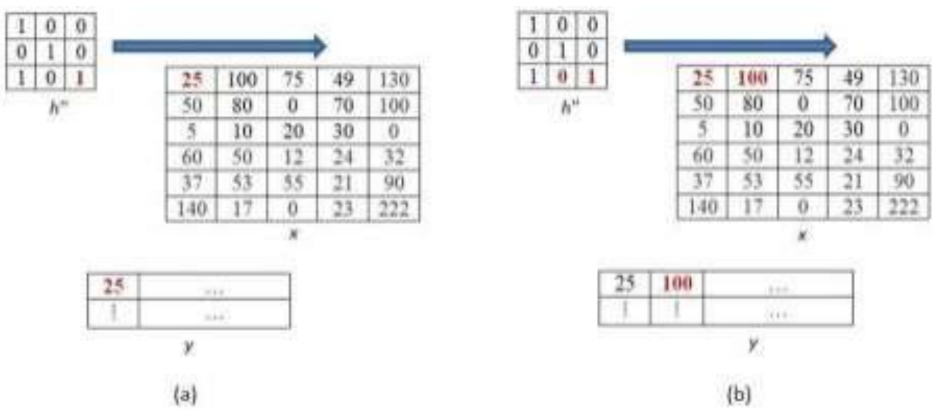


Figure 3 Convolution of pixels Row by Row

1	0	0
0	1	0
1	0	1

h''

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

x

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	...			
⋮	⋮					

y

$$y(4,3) = 50 \times 1 + 80 \times 0 + 0 \times 0 + 5 \times 0 + 10 \times 1 + 20 \times 0 + 60 \times 1 + 50 \times 0 + 12 \times 1$$
$$= 50 + 0 + 0 + 0 + 10 + 0 + 60 + 0 + 12 = 132$$

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

x

1	0	0
0	1	0
1	0	1

h''

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	255	...	
⋮	⋮					

y

$$y(6,5) = 12 \times 1 + 24 \times 0 + 32 \times 0 + 55 \times 0 + 21 \times 1 + 90 \times 0 + 0 \times 1 + 23 \times 0 + 222 \times 1$$
$$= 12 + 0 + 0 + 0 + 21 + 0 + 0 + 0 + 222 = 255$$

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	255	137	254
0	140	54	53	78	243	90
0	0	140	17	0	23	255

y

Figure 4 Resultant Matrix

Parallelization Strategy:

Version 1 (Serial):

We need to initialize the **convolution kernel** with the desire or some random values. Also, if we want to print the output matrix, we can do it through a **nested for loop**.

We will include all the necessary **libraries** for this operation. For larger images or more computationally intensive tasks, **parallel execution** using techniques like **multithreading** or **GPU acceleration** may provide significant performance improvements.

Version 2 (Open MP):

To parallelize the convolution operation, we will **parallelize the loop**. I added the **#pragma omp parallel for** directive just before the **outer loop**. This directive instructs OpenMP to **distribute the iterations of the loop** across multiple threads for parallel execution. We just have to make sure that **OpenMP** is installed and the library in included in the source file.

Version 3 (CUDA):

In this version, we have a **convolutionKernel** function that performs the 2D convolution operation on the **GPU**. Each thread in the **CUDA thread block** loads a portion of the input image and the convolution kernel into **shared memory**. It then performs the convolution operation and stores the result in the **output image**.

In the **main** function, we allocate memory on the **host** and the **device**, initialize the input image with **random values**, and **copy** it from the **host to device**. We define the **grid and block dimensions** based on the image size and the specified **block size**. Finally, we launch the **convolutionKernel** function on the GPU, copy the output image from the device to the host, and print the first few elements of the output for verification.

We need to make sure that we have an **Nvidia GPU** in the system and install the **CUDA Toolkit** before running this program.

Source Code:

Version 1 (Serial):

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "Timer.h"

// Constants
const int MATRIX_SIZE = 512;
const int KERNEL_SIZE = 15;

// Function to perform convolution
void convolution(const unsigned char* inputMatrix, const float* kernel, unsigned char* outputMatrix) {
    // Loop over the input matrix
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            float sum = 0.0;

            // Apply the convolution kernel
            for (int k = 0; k < KERNEL_SIZE; k++) {
                for (int l = 0; l < KERNEL_SIZE; l++) {
                    int x = i - KERNEL_SIZE / 2 + k;
                    int y = j - KERNEL_SIZE / 2 + l;

                    // Handle boundary conditions by clamping
                    if (x < 0) x = 0;
                    if (x >= MATRIX_SIZE) x = MATRIX_SIZE - 1;
                    if (y < 0) y = 0;
                    if (y >= MATRIX_SIZE) y = MATRIX_SIZE - 1;

                    sum += inputMatrix[x * MATRIX_SIZE + y] * kernel[k * KERNEL_SIZE + l];
                }
            }

            // Store the result in the output matrix
            outputMatrix[i * MATRIX_SIZE + j] = sum;
        }
    }
}

int main()
{
    //Declare Timer Variable
    Timer t1("Serial Execution Time: ");

    // Seed the random number generator
    std::srand(std::time(nullptr));

    // Create the input grayscale image matrix
    unsigned char inputMatrix[MATRIX_SIZE * MATRIX_SIZE];
    for (int i = 0; i < MATRIX_SIZE * MATRIX_SIZE; i++) {
        inputMatrix[i] = std::rand() % 256;  // Random pixel value between 0 and 255
```

```cpp
    }

    // Create the convolution kernel
    float kernel[KERNEL_SIZE * KERNEL_SIZE];
    // Initialize the kernel with your desired values here

    // Create the output grayscale image matrix
    unsigned char outputMatrix[MATRIX_SIZE * MATRIX_SIZE];

    //Start Timer
    t1.Start();

    // Perform convolution
    convolution(inputMatrix, kernel, outputMatrix);

    //Stop Timer
    t1.Stop();

    //Print Execution Time
    t1.Print();

    // Print the output matrix if desired
    /*
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            std::cout << static_cast<int>(outputMatrix[i * MATRIX_SIZE + j]) << " ";
        }
        std::cout << std::endl;
    }
    */

    return 0;
}
```

Figure 5 Version 1 Source Code (Serial)

Version 2 (OpenMP):

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>
#include "Timer.h"

// Constants
const int MATRIX_SIZE = 512;
const int KERNEL_SIZE = 15;

// Function to perform convolution
void convolution(const unsigned char* inputMatrix, const float* kernel, unsigned char* outputMatrix) {
    // Loop over the input matrix in parallel
    #pragma omp parallel for
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            float sum = 0.0;

            // Apply the convolution kernel
            for (int k = 0; k < KERNEL_SIZE; k++) {
                for (int l = 0; l < KERNEL_SIZE; l++) {
                    int x = i - KERNEL_SIZE / 2 + k;
                    int y = j - KERNEL_SIZE / 2 + l;

                    // Handle boundary conditions by clamping
                    if (x < 0) x = 0;
                    if (x >= MATRIX_SIZE) x = MATRIX_SIZE - 1;
                    if (y < 0) y = 0;
                    if (y >= MATRIX_SIZE) y = MATRIX_SIZE - 1;

                    sum += inputMatrix[x * MATRIX_SIZE + y] * kernel[k * KERNEL_SIZE + l];
                }
            }

            // Store the result in the output matrix
            outputMatrix[i * MATRIX_SIZE + j] = sum;
        }
    }
}

int main()
{
    //Declare Timer Variable
    Timer t1("Parallel Execution Time: ");

    // Seed the random number generator
    std::srand(std::time(nullptr));

    // Create the input grayscale image matrix
    unsigned char inputMatrix[MATRIX_SIZE * MATRIX_SIZE];
```

```cpp
    unsigned char inputMatrix[MATRIX_SIZE * MATRIX_SIZE];
    for (int i = 0; i < MATRIX_SIZE * MATRIX_SIZE; i++) {
        inputMatrix[i] = std::rand() % 256;  // Random pixel value between 0 and 255
    }

    // Create the convolution kernel
    float kernel[KERNEL_SIZE * KERNEL_SIZE];
    // Initialize the kernel with your desired values here

    // Create the output grayscale image matrix
    unsigned char outputMatrix[MATRIX_SIZE * MATRIX_SIZE];

    //Start Timer
    t1.Start();

    // Perform convolution
    convolution(inputMatrix, kernel, outputMatrix);

    //Stop Timer
    t1.Stop();

    //Print Execution Time
    t1.Print();

    // Print the output matrix if desired
    /*
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            std::cout << static_cast<int>(outputMatrix[i * MATRIX_SIZE + j]) << " ";
        }
        std::cout << std::endl;
    }
    */

    return 0;
}
```

Figure 6 Version 2 Source Code (OMP)

Version 3 (CUDA):

```
#include <stdio.h>
#include "Timer.h"

#define WIDTH 512
#define HEIGHT 512
#define KERNEL_SIZE 15
#define BLOCK_SIZE 16

__global__ void convolutionKernel(const unsigned char* input, unsigned char* output) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Compute the global position of the thread
    int row = by * blockDim.y + ty;
    int col = bx * blockDim.x + tx;

    // Shared memory for the convolution kernel
    __shared__ unsigned char sharedKernel[KERNEL_SIZE][KERNEL_SIZE];

    // Load the convolution kernel into shared memory
    if (ty < KERNEL_SIZE && tx < KERNEL_SIZE) {
        sharedKernel[ty][tx] = input[row * WIDTH + col];
    }
    __syncthreads();

    // Convolution operation
    int sum = 0;
    if (row < HEIGHT && col < WIDTH) {
        for (int i = 0; i < KERNEL_SIZE; i++) {
            for (int j = 0; j < KERNEL_SIZE; j++) {
                sum += sharedKernel[i][j] * input[(row + i) * WIDTH + (col + j)];
            }
        }
        output[row * WIDTH + col] = sum / (KERNEL_SIZE * KERNEL_SIZE);
    }
}

int main()
{
    //Declare Timer Variables
    Timer gputime;
    initTimer(&gputime, "GPU Execution Time: ");

    unsigned char *h_input, *h_output;
    unsigned char *d_input, *d_output;

    size_t size = WIDTH * HEIGHT * sizeof(unsigned char);

    // Allocate host memory
```

```
    h_input = (unsigned char*)malloc(size);
    h_output = (unsigned char*)malloc(size);

    // Allocate device memory
    cudaMalloc((void**)&d_input, size);
    cudaMalloc((void**)&d_output, size);

    // Initialize input data with random values
    for (int i = 0; i < WIDTH * HEIGHT; i++) {
        h_input[i] = rand() % 256;
    }

    // Copy input data from host to device
    cudaMemcpy(d_input, h_input, size, cudaMemcpyHostToDevice);

    // Define grid and block dimensions
    dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridDim((WIDTH + BLOCK_SIZE - 1) / BLOCK_SIZE, (HEIGHT + BLOCK_SIZE - 1) / BLOCK_SIZE);

    //Start Timer
    startTimer(&gputime);

    // Launch the convolution kernel
    convolutionKernel<<<gridDim, blockDim>>>(d_input, d_output);

    //Stop Timer
    stopTimer(&gputime);

    //Print Execution Time
    printTimer(gputime);

    // Copy output data from device to host
    cudaMemcpy(h_output, d_output, size, cudaMemcpyDeviceToHost);

    // Print the first few elements of the output for verification
    //for (int i = 0; i < 10; i++) {
    //    printf("%u ", h_output[i]);
    //}
    //printf("\n");

    // Free device memory
    cudaFree(d_input);
    cudaFree(d_output);

    // Free host memory
    free(h_input);
    free(h_output);

    return 0;
}
```

Figure 7 Version 3 Source Code (CUDA)

Detailed Profile & Analysis:

Version 1:

We have used the **Timer.h** header file for profiling. We simply initialize a timer variable which will store the execution time. Start the timer before calling the convolution function and end the timer after the execution. Finally, we just print the execution time.

```
abdullah@abdullah-PE70-6QE:~/pdc/project/serial$ g++ code.cpp -o output
abdullah@abdullah-PE70-6QE:~/pdc/project/serial$ ./output
Serial Execution Time:  : 304.931 msec
abdullah@abdullah-PE70-6QE:~/pdc/project/serial$
```

Figure 8 Serial Execution Time

Version 2:

We have followed the same procedure for profiling as in version 1. Include **Timer.h** library, start and end the timer before and after the function call for the convolution.

I have tested parallel execution with 2, 4, 6 and 8 threads to see which one gives the best performance.

```
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ export OMP_NUM_THREADS=2
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ g++ -o output -fopenmp code.cpp
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ ./output
Parallel Execution Time:  : 156.366 msec
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ export OMP_NUM_THREADS=4
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ g++ -o output -fopenmp code.cpp
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ ./output
Parallel Execution Time:  : 82.697 msec
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ export OMP_NUM_THREADS=6
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ g++ -o output -fopenmp code.cpp
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ ./output
Parallel Execution Time:  : 98.531 msec
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ export OMP_NUM_THREADS=8
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ g++ -o output -fopenmp code.cpp
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$ ./output
Parallel Execution Time:  : 78.384 msec
abdullah@abdullah-PE70-6QE:~/pdc/project/omp$
```

Figure 9 Parallel Execution Time Using OMP

Version 3:

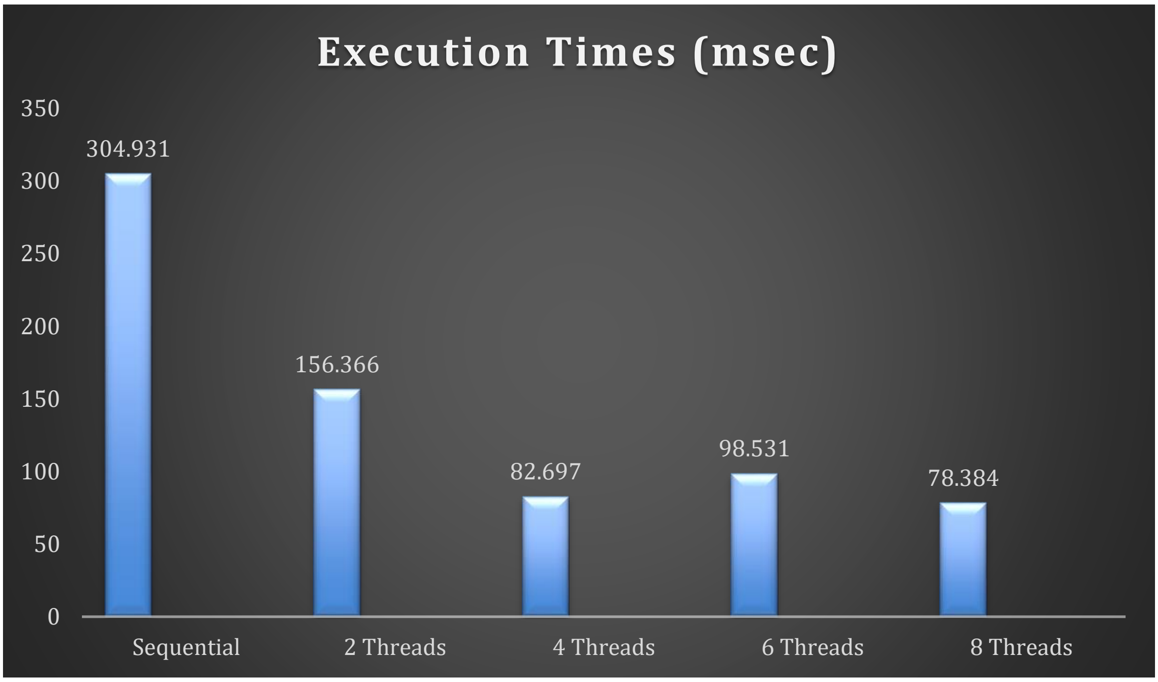
Simple include the **Timer.h** library and start and end the timer before and after the convolutionKernel call.

```
abdullah@abdullah-PE70-6QE:~/pdc/project/cuda$ nvcc -o output code.cu
code.cu(44): warning #2464-D: conversion from a string literal to "char *" is deprecated

code.cu(44): warning #2464-D: conversion from a string literal to "char *" is deprecated

abdullah@abdullah-PE70-6QE:~/pdc/project/cuda$ ./output
GPU Execution Time:  = 0.001 msec
abdullah@abdullah-PE70-6QE:~/pdc/project/cuda$
```

Figure 10 GPU Execution Time

Comparison Between Version 1 & Version 2:

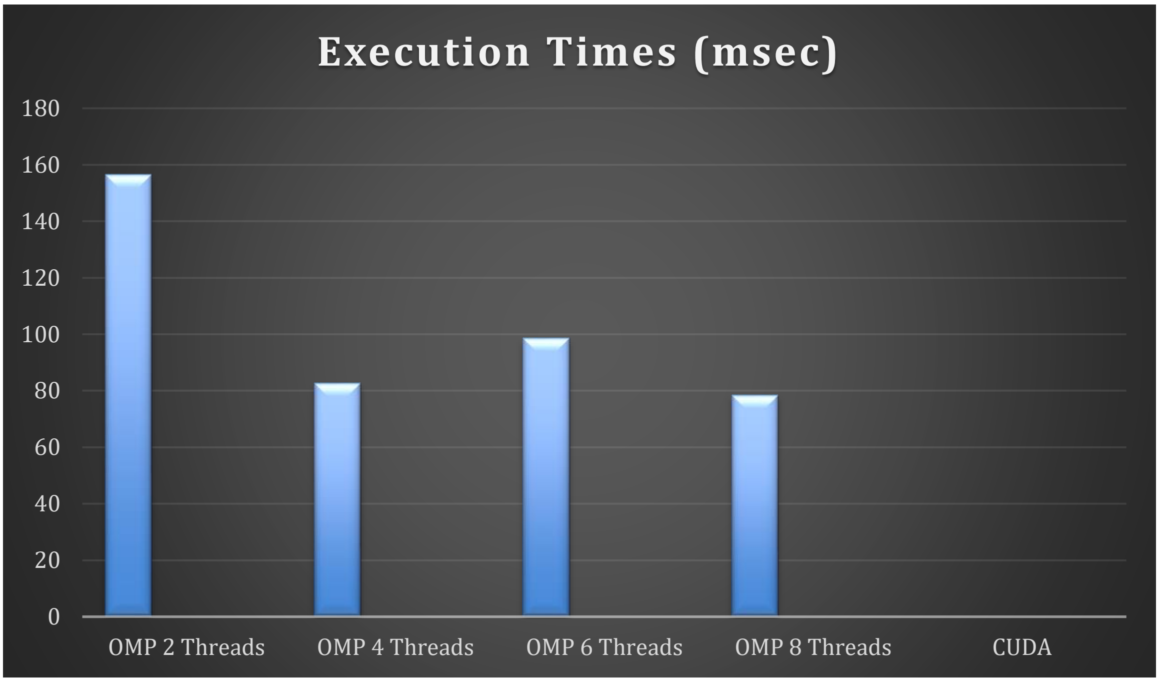
Version 1 was the **Sequential** version of the convolution program, and **Version 2** was the **Parallel** version of the same program. Both have been executed on **CPU**. The Following figure will give us a comparison between the performances of the two versions:



The best results have been achieved using **8 Threads**

Comparison Between Version 2 & Version 3:

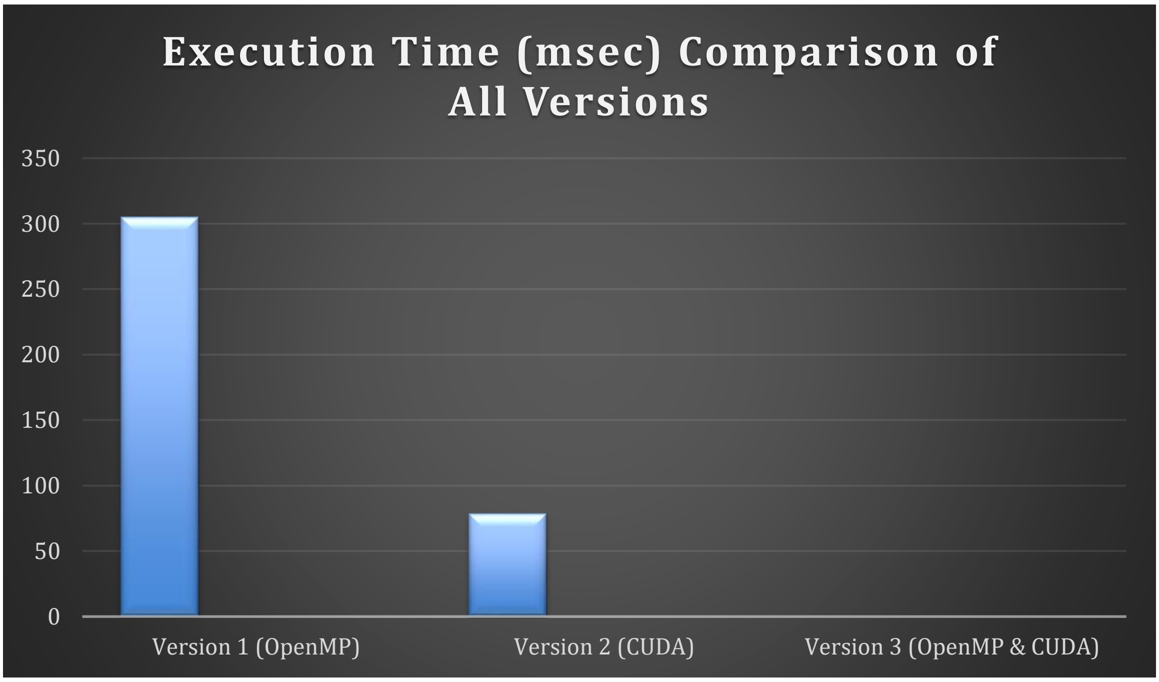
Version 2 was the parallel version for convolution using OpenMP and Version 3 was the GPU or CUDA version for the same program. The following figure will give us a comparison between the performances of the two versions:



CUDA gives such a huge performance improvement

Execution Time & Speed up Results:

Versions	Execution Times
Version 1	304.931 msec
Version 2	78.384 msec (best)
Version 3	0.001 msec



Speedup From Version 1 to Version 2:

Speed up = Execution Time (old) / Execution Time (new)

Speed up = 304.931 / 78.384

Speed up = 3.8902

Speedup From Version 2 to Version 3:

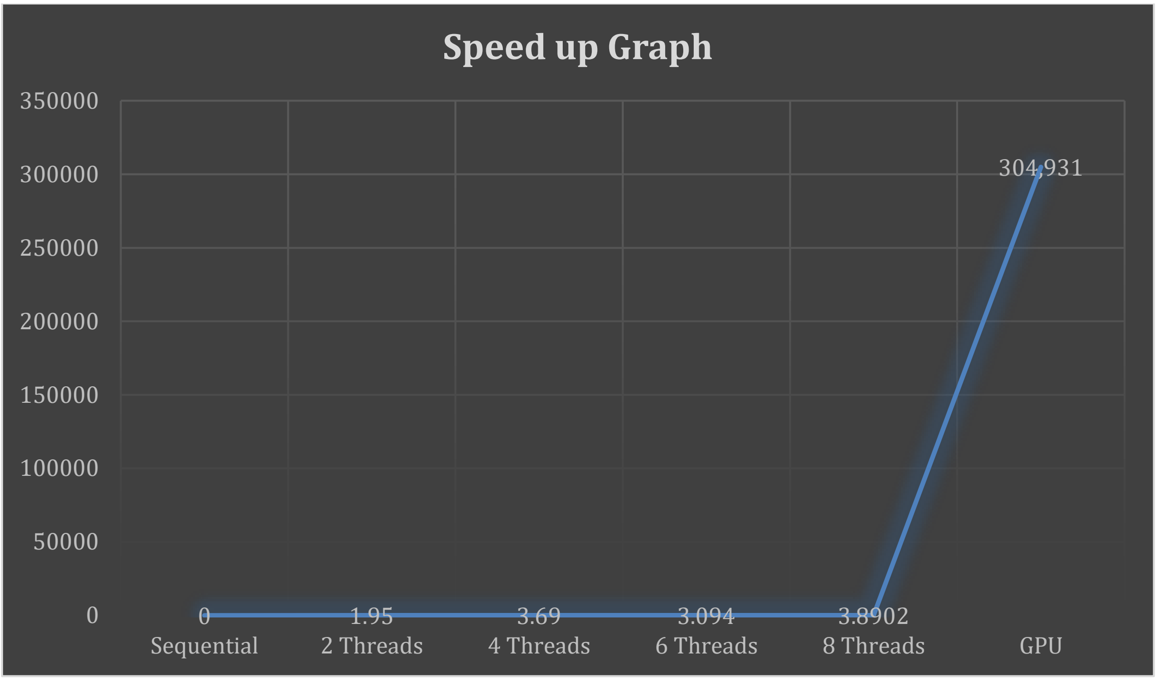
Speed up = 78.384 / 0.001

Speed up = 78,384

Speedup from Serial to CUDA Version:

Speed up = 304.931 / 0.001

Speed up = 304,931



Conclusion:

We have implemented many possible execution strategies like **sequential, parallel with 2, 4, 6 and 8 Threads** on **CPU** and execution on **GPU.** After testing all of the methods, **repetitive profiling** and testing, calculating **Speedup,** we have come to the conclusion that **GPU** adds exponential increase in a machine's performance. The graphs show that **multithreading on CPU** is good, but it is no way near as fast as **GPU,** because the GPU has a much greater number of **cores & threads** as compared to a **CPU.**