EC-444 Parallel & Distributed Computing

BS (CE)-2020

Complex Engineering Problem

Submitted By:

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

Department of Computer Engineering

HITEC University Taxila

Submitted To:

Dr. Imran Ashraf

Contents

Table of Figures

(Click on any figure to jump directly)

Problem Statement:

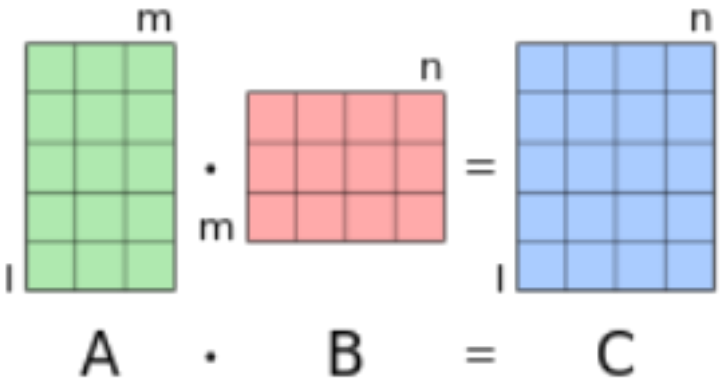


Figure 1 : Matrix Multiplication

We have to implement **matrix multiplication** with different versions, i.e **serial** and **parallel,** and also use **CPU** and **GPU** for both sequential and parallel execution. We can implement **GPU** execution of the program with the help of **CUDA**, and we can achieve **CPU** parallelism using **OpenMP.**

Tools Used:

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

Parallelization Strategy:

Version 1 (OpenMP):

For **Version 1,** parallelization can be achieved using **OpenMp.** OpenMP is a standard library with the **C/C++ Compilers**. First, we have to analyse the dependencies. If there are any data dependencies in the functions which we want to execute in parallel, we have to first get rid of the dependencies. Once there are no dependencies, we can use OpenMP. For this version, I have parallelized the **multiplication function.** I have used different number of threads to see which one works the best and gives the least execution time. I have used 2, 4, 6 and 8 threads.

```
44  //Multiply Matrix A and B
45  void mat_mul(int mat1[r][c], int mat2[c][r], int prod[r][r])
46  {
47      int i, j, k;
48
49      #pragma omp parallel for private(i, j, k) shared(mat1,mat2, prod)
50          for (i = 0; i < r; i++)
51          {
52              for (j = 0; j < r; ++j)
53              {
54                  for (k = 0; k < c; ++k)
55                  {
56                      prod[i][j] += mat1[i][k] * mat2[k][j];
57                  }
58              }
59          }
60  }
```

Figure 2: Version 1 Using OpenMP

The number of threads have been set each time through the **terminal** using **Export** command. We will see that in the analysis section of the report.

Version 2 (CUDA):

Version 2 has been executed on **GPU** using **CUDA Toolkit.** I have simply parallelized the program's multiplication function by creating and executing a kernel. First, the data is initialized on the CPU (Host), then it is transferred to the GPU (Device) where the multiplication is performed, then the results are sent back to the Host.

```
46
47      // Define block and grid dimensions
48      dim3 blockSize(16, 16);
49      dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) / blockSize.y);
50
51      // Launch kernel
52      startTimer(&gpuMulTime);
53      matrixMultiplication<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
54      stopTimer(&gpuMulTime);
55
56      // Copy result back to host
57      cudaMemcpy(h_C, d_C, N * N * sizeof(int), cudaMemcpyDeviceToHost);
58
```

Figure 3: Version 2 Using GPU and CUDA Toolkit

Version 3 (CUDA & OpenMP):

Version 3 is simply a hybrid version of both **OpenMP & CUDA**. The strategy I have used for this version is to simply execute the **Host** part of the program in parallel using **OpenMP.** The **Device** part of the program (matrix multiplication) has been done using **CUDA.**

```
29      //OMP Section for Host Memory Allocation and Data Initialization
30      startTimer(&cpuTime);
31      #pragma omp parallel sections
32      {
33          #pragma omp section
34          {
35              // Allocate host memory
36              h_A = (int*)malloc(N * N * sizeof(int));
37              h_B = (int*)malloc(N * N * sizeof(int));
38              h_C = (int*)malloc(N * N * sizeof(int));
39          }
40
41          #pragma omp section
42          {
43              // Initialize host matrices with some values
44              for (int i = 0; i < N * N; i++)
45              {
46                  h_A[i] = i;
47                  h_B[i] = i;
48              }
49          }
50      }
51      stopTimer(&cpuTime);
```

Figure 4: Version 3 OpenMP Section

```
62      // Define block and grid dimensions
63      dim3 blockSize(16, 16);
64      dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (N + blockSize.y - 1) / blockSize.y);
65
66      // Launch kernel
67      startTimer(&gpuMulTime);
68      matrixMultiplication<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
69      stopTimer(&gpuMulTime);
70
71      // Copy result back to host
72      cudaMemcpy(h_C, d_C, N * N * sizeof(int), cudaMemcpyDeviceToHost);
```

Figure 5: Version 3 CUDA Kernel

Detailed Profile & Analysis:

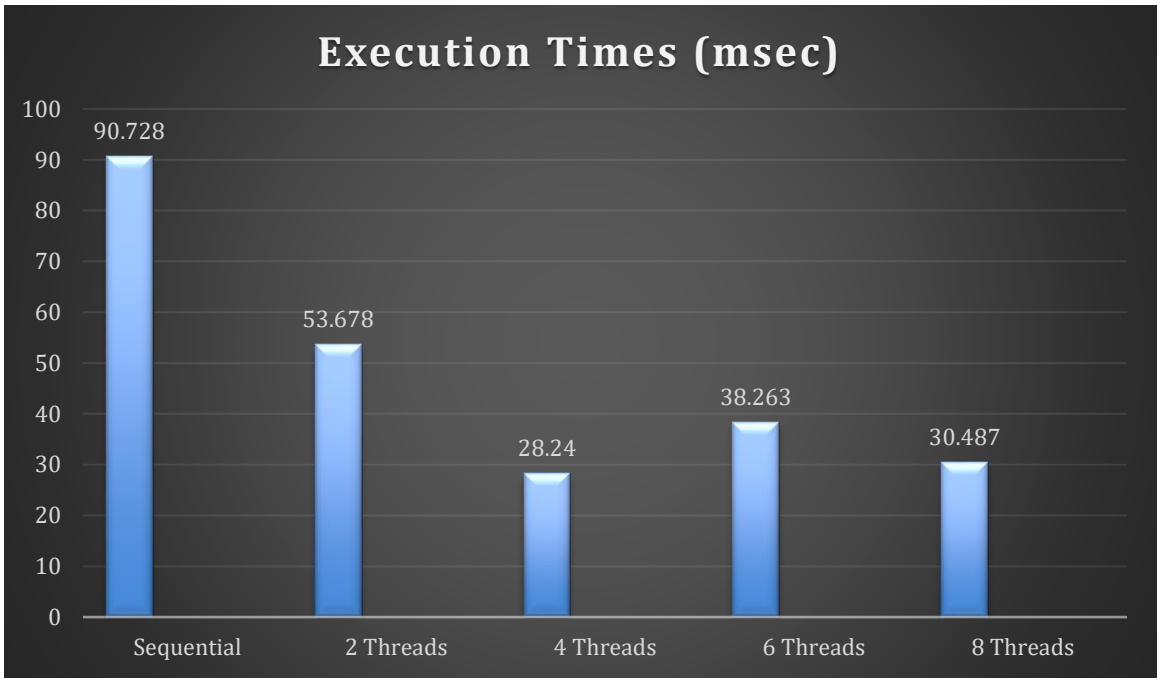
Comparison Between Version 0 & Version 1:

Version 0 was the **Sequential** version of the matrix multiplication program, and **Version 1** was the **Parallel** version of the same program. Both have been executed on **CPU**. The Following figures will give us a comparison between the performances of the two versions:

```
abdullah@abdullah-PE70-6QE:~/pdc/cep$ ./version0.out
CPU EXecution Time:  : 90.728 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep$ export OMP_NUM_THREADS=2
abdullah@abdullah-PE70-6QE:~/pdc/cep$ g++ -o version1.out -fopenmp version1.cpp
abdullah@abdullah-PE70-6QE:~/pdc/cep$ ./version1.out
CPU Parallel Execution Time With 2 Threads:  : 53.678 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep$ export OMP_NUM_THREADS=4
abdullah@abdullah-PE70-6QE:~/pdc/cep$ g++ -o version1.out -fopenmp version1.cpp
abdullah@abdullah-PE70-6QE:~/pdc/cep$ ./version1.out
CPU Parallel Execution Time With 4 Threads:  : 28.24 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep$ export OMP_NUM_THREADS=6
abdullah@abdullah-PE70-6QE:~/pdc/cep$ g++ -o version1.out -fopenmp version1.cpp
abdullah@abdullah-PE70-6QE:~/pdc/cep$ ./version1.out
CPU Parallel Execution Time With 6 Threads:  : 38.263 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep$ export OMP_NUM_THREADS=8
abdullah@abdullah-PE70-6QE:~/pdc/cep$ g++ -o version1.out -fopenmp version1.cpp
abdullah@abdullah-PE70-6QE:~/pdc/cep$ ./version1.out
CPU Parallel Execution Time With 8 Threads:  : 30.487 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep$
```

Figure 6: Sequential & Parallel Execution Times

The serial version has been done on a single thread and the parallel OpenMP version has been tested using 2, 4, 6 and 8 threads one by one.



The best results have been achieved using **4 Threads**

Version 2 (CUDA):

The execution time in this version has greatly reduced. The **multiplication** function has taken way less time to execute than what it took in both the **previous versions.**

```
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version2$ nvcc -o version2.out version2.cu
version2.cu(21): warning #2464-D: conversion from a string literal to "char *" is deprecated

version2.cu(21): warning #2464-D: conversion from a string literal to "char *" is deprecated

abdullah@abdullah-PE70-6QE:~/pdc/cep/Version2$ ./version2.out
GPU Multiplication Time = 0.005 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version2$
```

Figure 7: Version 2 Execution Time

Version 3 (Hybrid):

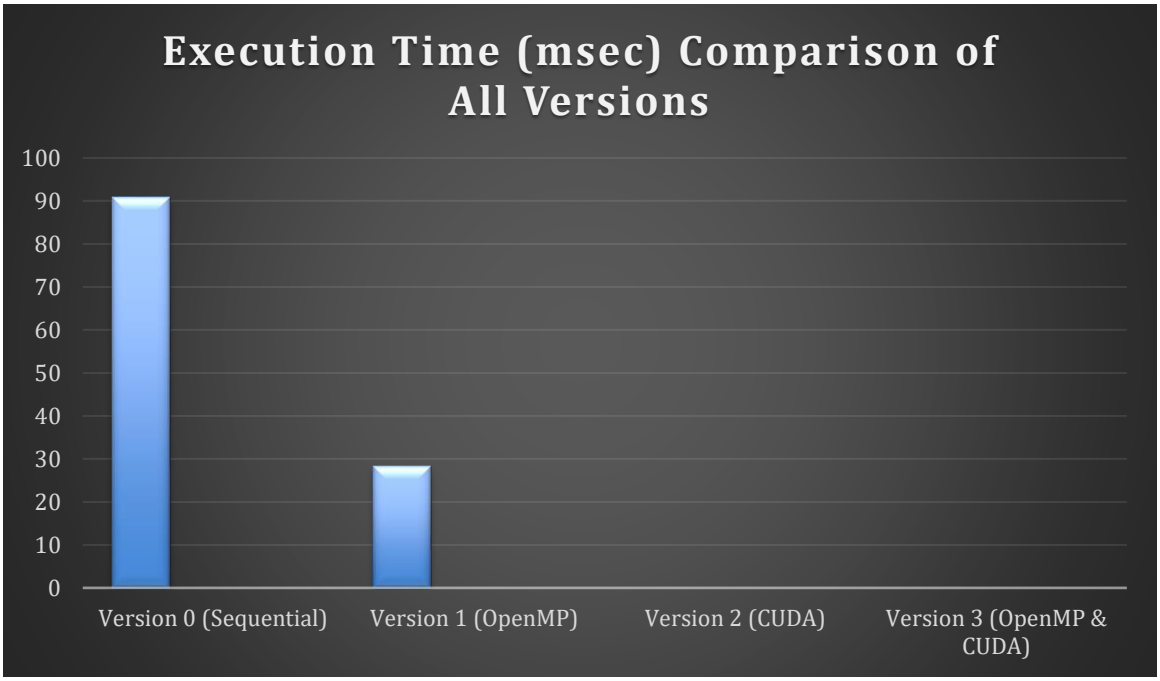
The version 3 had a bit of an advantage over **version 2,** because the CPU execution section was executed in parallel. Otherwise the matrix multiplication function had no improvement in performance.

```
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version2$ cd
abdullah@abdullah-PE70-6QE:~$ cd pdc
abdullah@abdullah-PE70-6QE:~/pdc$ cd cep
abdullah@abdullah-PE70-6QE:~/pdc/cep$ cd Version3
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version3$ ls
Timer.h  version3.cu
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version3$ subl version3.cu
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version3$ nvcc -o version3.out version3.cu
version3.cu(22): warning #2464-D: conversion from a string literal to "char *" is deprecated

version3.cu(23): warning #2464-D: conversion from a string literal to "char *" is deprecated

version3.cu(22): warning #2464-D: conversion from a string literal to "char *" is deprecated

version3.cu(23): warning #2464-D: conversion from a string literal to "char *" is deprecated

abdullah@abdullah-PE70-6QE:~/pdc/cep/Version3$ ./version3.out
GPU Multiplication Time = 0.005 msec
CPU Time for Initialization and Allocation:  = 6.614 msec
abdullah@abdullah-PE70-6QE:~/pdc/cep/Version3$
```

Figure 8: Version 3 Execution Time

Execution Time & Speed up Results:

Versions	Execution Times
Version 0	90.728 msec
Version 1	28.24 msec (best result after trial & test)
Version 2	0.005 msec
Version 3	0.005 msec



Speedup From Version 0 to Version 1:

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

Speed up = 90.728 / 28.24

Speed up = 3.212

Speedup From Version 1 to Version 3:

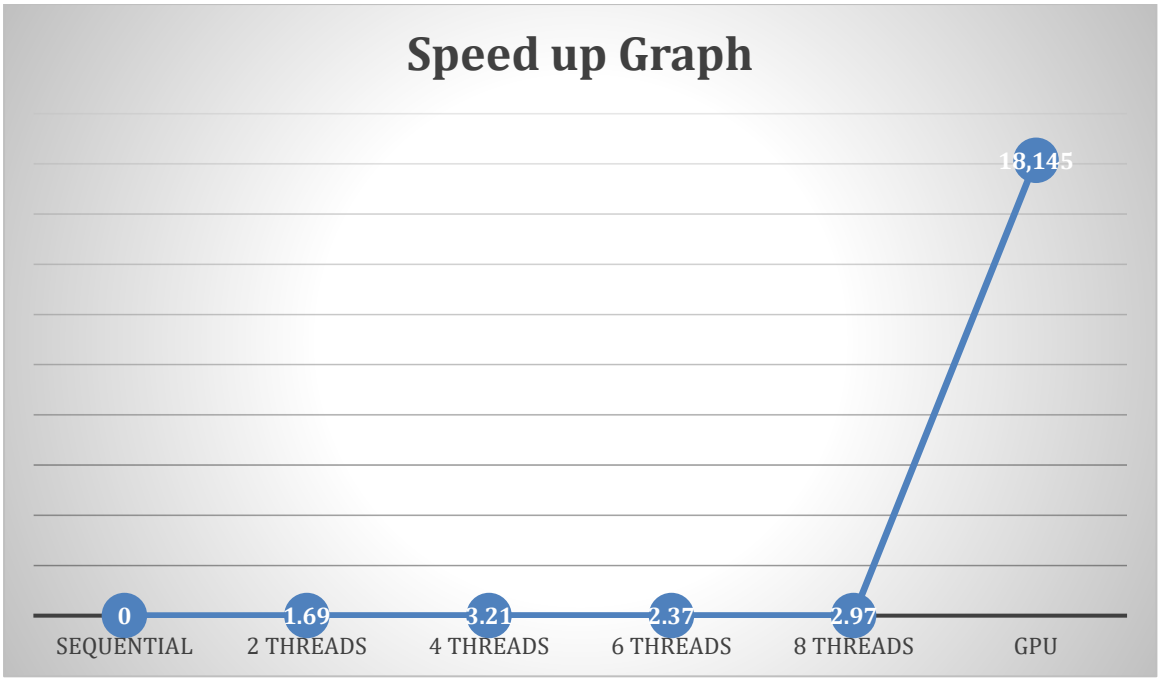
Speed up = 28.24 / 0.005

Speed up = 5,648

Speedup from Sequential to CUDA Version:

Speed up = 90.728 / 0.005

Speed up = 18,145



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.**