Andrew Porter

933007811

CSCE 735 Parallel Computing

Final Project Report

Overview

For this project, we had to parallelize Strassen's algorithm. Two options were presented, an OpenMP-based code, or CUDA GPU-based code.

Initially, this project was attempted using CUDA. However, difficulty in project compilation and execution of the CUDA kernels resulted in this attempt being abandoned. It will be resumed at a later time. The issue was most likely CUDA compilation using proper flags that correlated to the architecture.

This coding project was broken up into the following segments: Implementation of a serial Strassen's Algorithm, Memory Management, and Parallelization. The longest sprint of this project was the initial implementation of the algorithm. Extra care was taken to write test code for the helper functions that would ultimately comprise Strassen's algorithm. These tests proved crucial in pointing out pointers and memory allocation bugs. Setting the DEBUG macro to 1 will run each test. Tests are activated by the corresponding TEST__XX macros being set to 1.

Once testing of the helper functions was completed, the functions had to be tested in a recursive manner that applied to Strassen's algorithm. These resulted in more bugs being discovered with many matrix operations occurring.

Memory issues had to be removed once the serial algorithm was correctly working. This problem caused the program to crash occasionally. Valgrind was a very helpful tool in pointing out memory leaks in the code. This was a very straightforward section.

Parallelization of the code was also not very problematic. The approach I initially took was to parallelize each helper function of code (Add_Matrix, Sub_Matrix, etc) and also parallelize the main algorithm. This parallelization could be done using OpenMP sections to calculate the submatrices.

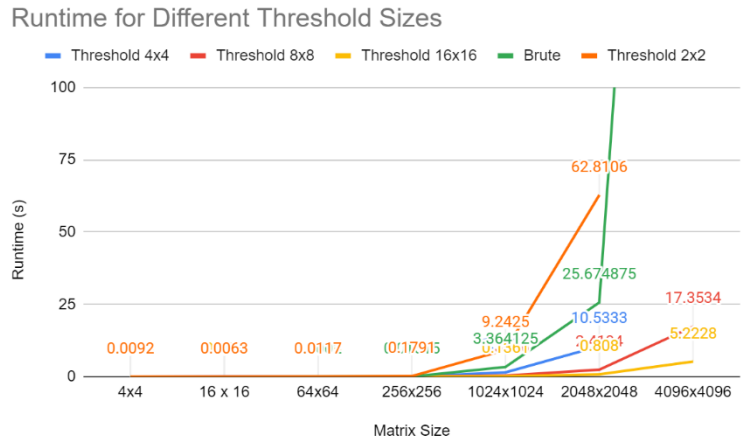
Lessons Learned

One lesson learned was the importance of writing tests as the project progresses. Previously, I had not done this as much with C code. Outside of gdb, the most used debugging tool had been print statements. However, writing activatable code segments with testing-focused helper functions, made the process much smoother and helped see issues before they got to be too large.

Another lesson learned was the ability to modify pointer values in functions. I had to use this for the Split Matrix function. This sped up the process as I could divide a whole matrix at once without having to have several case statements.

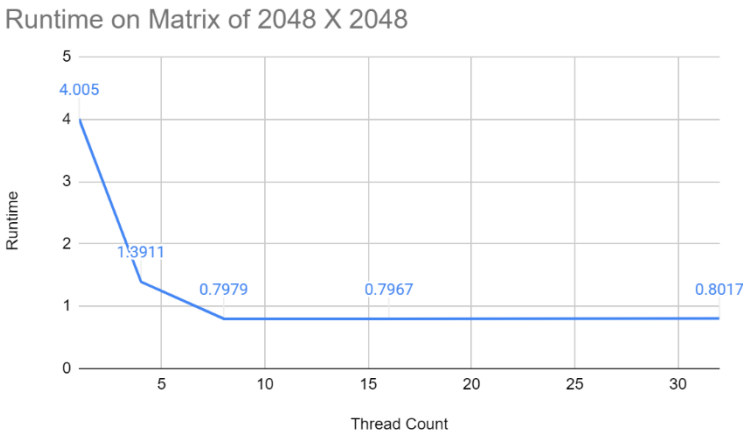
Finally, I figured out that recording issues with my code is the best way to overcome them. I encountered several memory crashes while testing my program but would forget the cause after I fixed them. I began to write the error and the fix for each bug in the top of my code file. Although this practice started too late to be of any use, it helps me remember and more quickly identify the problem with the source code.

Results

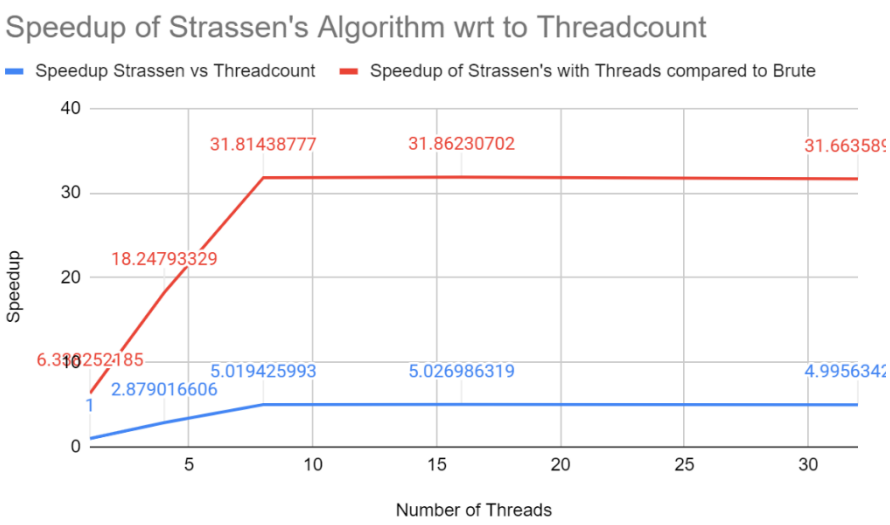


Execution time for the program can be seen in this Threshold test which compares the execution runtime of the algorithm with different thresholds to the brute force algorithm. Due to the scale of the brute force algorithm runtime for the 4096 x 4096 matrix (runtime-486.3909 seconds) it has been omitted from the scale. It can be seen in the figure that a threshold size for Strassen's Algorithm is best when the threshold matrix size is 16 x16 (yellow). Strassen's algorithm does the worst when the threshold is very small, although all threshold sizes showed improvement over the brute force method for large matrices.

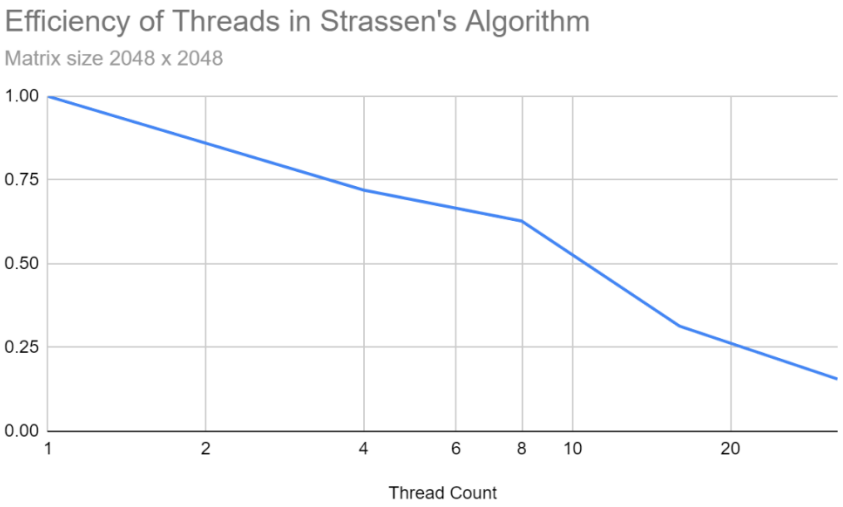
For small matrices, Strassen's algorithm performed worse because of the overhead required for creating threads.



The figure on the left shows Run-Time of the program with respect to number of threads. Introducing more threads did not seem to improve performance of the algorithm after 10 threads. However, even with a single thread, the runtime performance of the algorithm was 84% faster than that of the brute force (runtime = 25.4 seconds).



This chart shows the overall speedup of Strassen's Algorithm compared to the brute force method as more threads are introduced. The blue line indicates the speedup of Strassen's algorithm with respect to itself and the number of threads.



As with all algorithms, mine did not make good use of threads as more were added. I believe the issue to be just too many threads and not enough jobs. Efficiency decreased almost linearly compared to the execution of the program.

Compilation and Execution Instructions

To run the program, one must load the **intel** module. Compilation and execution can be done with the following commands:

```
module load intel
icx -qopenmp strassen_open_mp.c -o strassen.exe
./strassen.exe <k> <m> <t>
```

where

- *k* represents the size of a matrix, where the total number of elements will be *k* X *k* elements: $size\ of\ matrix = 2^{2k}$

- m represents the size of the threshold matrix for which Strassen's algorithm will use brute force multiply where the number of elements in an m X m threshold matrix will be $threshold = 2^{2m}$ elements.
- t represents the number of threads running in the parallel sections of the code

Test outputs below

```
Welcome To Strassen Algorith OpenMP
Growing Matrix Sizes, Fixed Thread Count = 10, Threshold = 2 x 2 Matrix
Matrix Size = 4^2, Threads = 10, time (sec) =   0.0092, brute_time =   0.0000
Matrix Size = 16^2, Threads = 10, time (sec) =   0.0063, brute_time =   0.0000
Matrix Size = 64^2, Threads = 10, time (sec) =   0.0117, brute_time =   0.0003
Matrix Size = 256^2, Threads = 10, time (sec) =   0.1791, brute_time =   0.0143
Matrix Size = 1024^2, Threads = 10, time (sec) =   9.2425, brute_time =   3.6071
Matrix Size = 2048^2, Threads = 10, time (sec) =  62.8106, brute_time =  25.4518

Growing Matrix Sizes, Fixed Thread Count = 10, Threshold = 4 x 4 Matrix
Matrix Size = 4^2, Threads = 10, time (sec) =   0.0013, brute_time =   0.0000
Matrix Size = 16^2, Threads = 10, time (sec) =   0.0058, brute_time =   0.0000
Matrix Size = 64^2, Threads = 10, time (sec) =   0.0085, brute_time =   0.0002
Matrix Size = 256^2, Threads = 10, time (sec) =   0.0409, brute_time =   0.0144
Matrix Size = 1024^2, Threads = 10, time (sec) =   1.5032, brute_time =   3.2849
Matrix Size = 2048^2, Threads = 10, time (sec) =  10.5333, brute_time =  25.4467

Growing Matrix Sizes, Fixed Thread Count = 10, Threshold = 8 x 8 Matrix
Matrix Size = 4^2, Threads = 10, time (sec) =   0.0023, brute_time =   0.0000
Matrix Size = 16^2, Threads = 10, time (sec) =   0.0071, brute_time =   0.0000
Matrix Size = 64^2, Threads = 10, time (sec) =   0.0083, brute_time =   0.0002
Matrix Size = 256^2, Threads = 10, time (sec) =   0.0195, brute_time =   0.0151
Matrix Size = 1024^2, Threads = 10, time (sec) =   0.3616, brute_time =   3.2826
Matrix Size = 2048^2, Threads = 10, time (sec) =   2.4194, brute_time =  25.4342
Matrix Size = 4096^2, Threads = 10, time (sec) =  17.3534, brute_time = 486.0071

Growing Matrix Sizes, Fixed Thread Count = 10, Threshold = 16 x 16 Matrix
Matrix Size = 4^2, Threads = 10, time (sec) =   0.0022, brute_time =   0.0000
Matrix Size = 16^2, Threads = 10, time (sec) =   0.0010, brute_time =   0.0000
Matrix Size = 64^2, Threads = 10, time (sec) =   0.0076, brute_time =   0.0003
Matrix Size = 256^2, Threads = 10, time (sec) =   0.0149, brute_time =   0.0168
Matrix Size = 1024^2, Threads = 10, time (sec) =   0.1361, brute_time =   3.2819
Matrix Size = 2048^2, Threads = 10, time (sec) =   0.8080, brute_time =  26.3668
Matrix Size = 4096^2, Threads = 10, time (sec) =   5.2228, brute_time = 486.7747

Fixed Matrix Size = 2048 x 2048, Growing Thread Count, Threshold = 16 x 16 Matrix
Matrix Size = 2048^2, Threads = 1, time (sec) =   4.0050, brute_time =  25.3847
Matrix Size = 2048^2, Threads = 4, time (sec) =   1.3911, brute_time =  25.4290
Matrix Size = 2048^2, Threads = 8, time (sec) =   0.7979, brute_time =  25.4007
Matrix Size = 2048^2, Threads = 16, time (sec) =   0.7967, brute_time =  25.3845
Matrix Size = 2048^2, Threads = 32, time (sec) =   0.8017, brute_time =  25.5897
```