EC527 High Performance Programming with Multicore and GPUs - Final Project

Advanced Matrix Multiplication:

An In-Depth Exploration and Optimization of the Strassen Algorithm

Yuhang Song

yuhangs@bu.edu

1 – Introduction

Matrix multiplication is a fundamental operation in various fields of science and engineering. It is especially important as a basic operation in today's machine learning. Optimizing and improving matrix multiplication operations is meaningful and necessary. In this project, we start with the triple-for loop matrix multiplication as a baseline, implement and optimize the Strassen algorithm, and analyze its performance.

2 – Descrition of the algorithm

2.1 Baseline: Triple-for loop matrix multiplication with OpenMP and Blocking enabled

The most typical calculation method for matrix multiplication is the triple-for loop, as shown below. It is simple and straightforward.

```
for (i = 0; i < length; i++) {
  for (j = 0; j < length; j++) {
    sum = IDENT;
    for (k = 0; k < length; k++) {
      sum += a0[i*length+k] * b0[k*length+j];
    }
    c0[i*length+j] += sum;
  }
}
```

As we did in the lab, a minor adjustment to improve performance is to swap the loop order from ijk to kij, as shown below.

```
for (k = 0; k < length; k++) {
  for (i = 0; i < length; i++) {
    r = a0[i*length+k];
    for (j = 0; j < length; j++) {
      c0[i*length+j] += r*b0[k*length+j];
    }
  }
}
```

To further improve performance, we can implement blocking. As demonstrated in the lab, blocking optimizes the memory access pattern by utilizing cache locality.

```
{
  for (k = 0; k < row_length; k++) {
    for (i = 0; i < row_length; i++) {
      r = a0[i*row_length+k];
      #pragma omp for
      for (j = 0; j < row_length; j++)
        c0[i*row_length+j] += r*b0[k*row_length+j];
```

```
      }
    }
  }
```

In Lab 6, we implemented OpenMP for the blocking triple-for loop matrix multiplication to further enhance performance.

```
#pragma omp parallel shared(a0,b0,c0,row_length) private(i,j,k,r)
  {
    for (k = 0; k < row_length; k++) {
      for (i = 0; i < row_length; i++) {
        r = a0[i*row_length+k];
        #pragma omp for
        for (j = 0; j < row_length; j++)
          c0[i*row_length+j] += r*b0[k*row_length+j];
      }
    }
  }
```

We added pragma to utilize OpenMP and run it in parallel within the outer loop.

In this project, we use the above optimal version as our baseline. Our primary goal is to compare the performance of our Strassen algorithm with this baseline. During the computation of the Strassen algorithm, we need to recursively split the matrix into smaller ones. However, at some point, when the matrix is small enough, the Strassen algorithm is no longer efficient, and we need to switch to triple-for loop matrix multiplication. We will use this baseline matrix multiplication for that purpose.

2.2 The Strassen Algorithm

Compared to the $O(n^3)$ time complexity of the triple-for loop algorithm, the Strassen algorithm reduces the time complexity to $O(n^{2.81})$. It reduces the matrix multiplication to 7 multiplications and 18 additions. And the algorithm works recursively.

When can calculate the matrix multiplication

$$C = AB$$

we will split both A and B into four smaller equal-sized matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Then, we calculate the following seven matrices using the smaller matrices above:

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22})B_{11}$$

$$P_3 = A_{11}(B_{12} - B_{22})$$

$$P_4 = A_{22}(B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12})B_{22}$$

$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

After obtaining the seven intermediate matrices above, we can calculate the components of matrix C as follows:

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{21} = P_2 + P_4$$

$$C_{12} = P_3 + P_5$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

Then, we can combine the four matrices to yield the final result:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

The Strassen algorithm works recursively. However, the Strassen algorithm is not efficient for smaller matrices. Therefore, there is a breakpoint at which the recursion is stopped, and a well-optimized triple-for loop matrix multiplication is used instead.

3 – Implementation and Design

3.1 Code structure

The base file is taken from Lab 6, specifically the OpenMP matrix multiplication section. It contains a nice structure for timing as well as testing on different matrix sizes. However, as this is a large project, it is unwise to include everything in a single file. To address this, I implemented a CMake build framework and added unit tests. The project file structure looks like this:

```
$ tree
.
├── CMakeLists.txt
├── LICENSE
├── README.md
├── src
│   ├── main.c
│   ├── matrix.c
│   ├── matrix.h
│   ├── mmm.c
│   ├── mmm.h
│   ├── strassen.c
│   ├── strassen.h
│   ├── strassen_simd.c
│   ├── strassen_simd.h
│   ├── timer.c
│   └── timer.h
└── tests
    ├── compare_time.c
    ├── test_matrix.c
    ├── test_mmm.c
    ├── test_strassen.c
    ├── test_strassen_simd.c
    └── test_timer.c
```

To build this project, you can simply use CMake and Make commands without worrying about which files to compile or which compile flags to use. Simply navigate to the directory of this project and use the following commands to compile:

```
mkdir -p build && cd build
```

```
cmake ..
make -j
```

All the binaries will be compiled into the current build folder.

I used unit tests to verify the correctness of the project. The unit tests use the standard CTest and should be recognized by any IDE. To run all the tests in the command line, use "`make test`".

Although the timer-related code is provided by the lab, I modified the code structure and moved it to a separate file, `timer.h`. I also added a small test, `test_timer.c,` to verify its functionality.

For basic matrix operations, like creating/deleting matrices, initializing matrices with zeros, sequences or random numbers, and matrix addition and subtraction, I took the base code from the lab and completed the missing functions in `matrix.h`. I also added matrix comparison functions and matrix comparisons with tolerance to help determine whether two matrices are equal. I added the file `test_matrix.c` to test every function and ensure the correctness of these basic matrix operations.

Then, I implemented the triple-for loop matrix multiplication in `mmm.h`. It contains the kij for loop, as well as the blocking and OpenMP versions. I used `test_matrix.c` to measure correctness. To verify the correctness of matrix multiplication, I created a gold matrix multiplication using the triple-for loop without any optimization. I compared all the matrix multiplication results with this golden reference. For most of the triple-for loop optimizations, the precision remains unchanged, passing the unit test check with a tolerance of 5%. However, it is somewhat surprising that the kij for loop with OpenMP enabled has significant precision loss. It has about a 15% precision loss at some elements. I manually checked the results on a small matrix, and there is no error in the computation itself. This is mainly caused by the change in the order of floating-point number computations, which leads to precision loss.

Then, I followed the definition of the Strassen algorithm to write `strassen.h` and also measured its correctness in `test_strassen.c`. After verifying correctness, I used the unit test `compare_time.c` to measure performance.

Lastly, I optimized the Strassen algorithm by applying the AVX256 intrinsics to matrix multiplication in `strassen_simd.h` and verified the correctness in `test_strassen_simd.c`. I will discuss the details later in this report.

All the updated code can be found in my GitHub repository https://github.com/Souukou/bu_ec527_strassen_matrix_multiplication.

4 – Strassen Algorithm and Experiment

In this section, I will describe how I implemented the Strassen algorithm and what I did to improve its performance. I measured the performance using an AMD Ryzen 9 5950X 16-Core Processor. It has a 32KB L1 cache for each core, and a 64MB L3 cache in total for all cores. I will measure the performance with the OpenMP thread setting at 32. Additionally, I am measuring the size of matrices that can fit into the L3 cache as well as those that do not fit into it.

It is worth mentioning that all the tables and figures are in seconds. I am not calculating or plotting out the cycles per element (CPE), as in a multithreaded program, multiple cores are running at different clock speeds and accumulating cycles at different rates. Thus, it would be meaningless to measure the cycles.

4.1 Plain Strassen Algorithm

Initially, I strictly followed the definition of the Strassen algorithm to implement the Strassen algorithm. I created a set of small matrices A_{11}, A_{12} ... B_{11} ... C_{22}, as well as a list of intermediate matrices P_1, P_2 ... P_7, and two temporary matrices for addition. Then, I copied the data from the original matrices to the smaller new matrices A_{11}, A_{12} ... B_{11} ... C_{22}, computed the intermediate results recursively, and combined them together before writing them back to matrix C.

Here is the performance of the plain Strassen algorithm.

rowlen	kij	kij_omp	kij_block_omp	strassen
160	0.007604	0.002555	0.006893	0.008827
208	0.01664	0.003695	0.001907	0.002354
320	0.05969	0.01133	0.008655	0.01286
496	0.218	0.0377	0.02132	0.03104
736	0.7067	0.1056	0.06943	0.144
1040	1.99	0.2712	0.2214	0.5214
1408	4.934	0.6517	0.5246	0.9386
1840	11	1.341	1.134	1.68
2336	22.5	2.668	2.286	4.866
2896	42.85	4.673	4.262	7.306

Table 1. Comparison of Strassen Algorithm with Other MM Algorithms from Lab 6

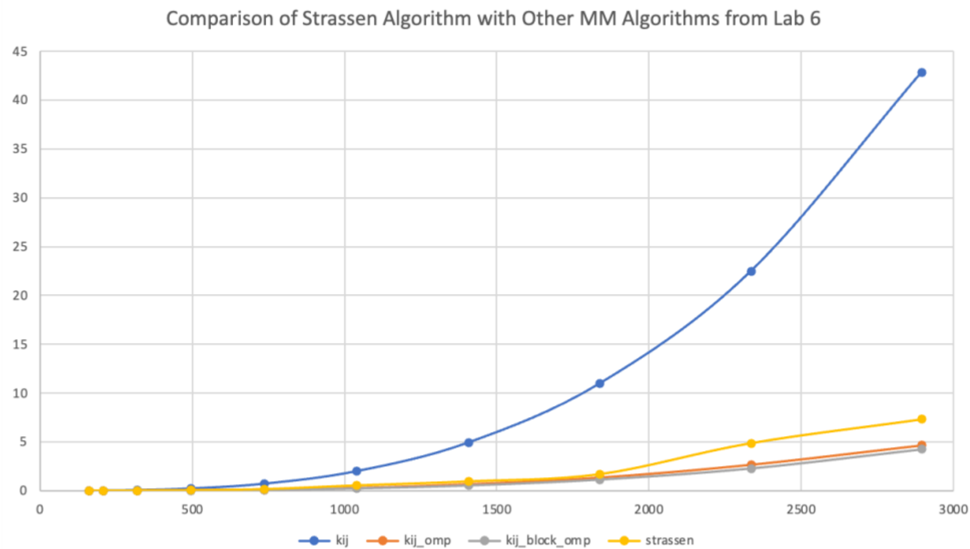

Figure 1. Comparison of Strassen Algorithm with Other MM Algorithms from Lab 6

The Strassen algorithm performance results are shown in the last column. We can see that, compared to the most basic ijk or kji approaches, it has a significant improvement, with about a 5.8 times speedup. However, when compared to the fastest result we obtained from the previous lab, which used the kij triple loop with blocking and OpenMP, the Strassen algorithm is noticeably slower.

4.2 Strassen Algorithm with SIMD Optimization

We learned to use SIMD intrinsic to optimize matrix transposition this semester. Similarly, I implemented AVX256 intrinsic to optimize matrix computation in this algorithm, including matrix multiplication, addition, and subtraction. This is implemented in `strassen_simd.h`.

The performance data is shown below.

rowlen	kij	kij_omp	kij_block_omp	strassen	strassen_simd
160	0.008146	0.002176	0.01093	0.0088	0.0004861
208	0.01715	0.002869	0.002971	0.002497	0.0007681
320	0.07402	0.009379	0.008117	0.01393	0.006229
496	0.2466	0.03332	0.02592	0.04052	0.01497
736	0.7111	0.1031	0.07205	0.1528	0.06869
1040	2.004	0.2746	0.2216	0.5071	0.2545
1408	4.972	0.6335	0.5267	0.924	0.4725
1840	11.09	1.302	1.189	1.617	0.8054
2336	22.69	2.652	2.402	4.671	2.465
2896	43.24	4.682	4.421	7.098	3.645

Table 2. Comparison of SIMD Strassen Algorithm with Other MM Algorithms

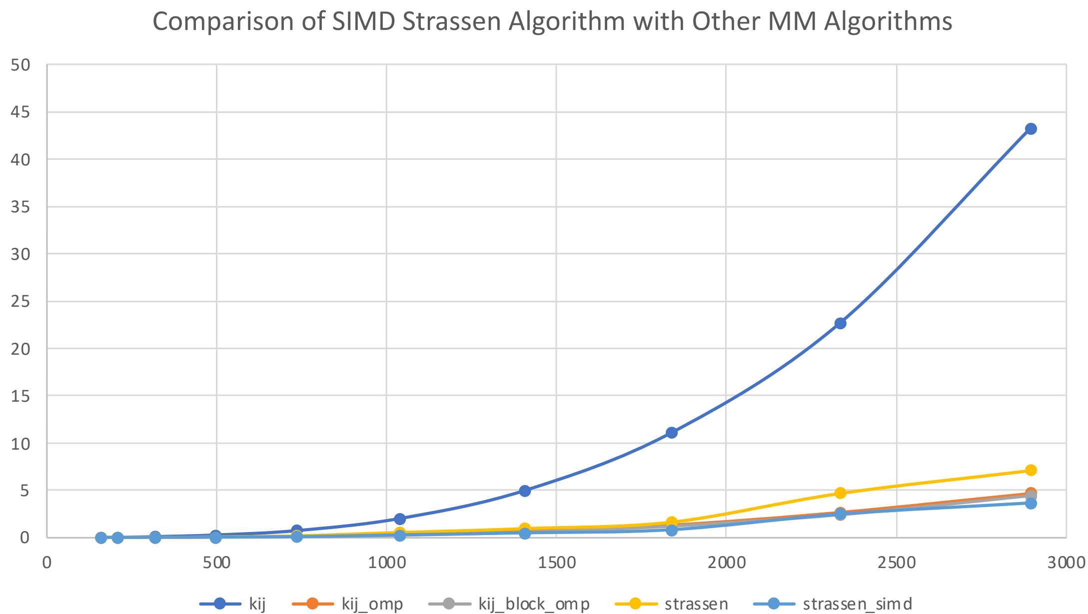

Figure 2. Comparison of SIMD Strassen Algorithm with Other MM Algorithms

The Strassen algorithm with AVX256 intrinsics enabled, as shown in the last column, demonstrates that after this optimization, the Strassen algorithm can outperform the fastest algorithm we have.

4.3 A Further Exploration on Strassen Algorithm

4.3.1 An Attempt to Implement Matrix View to Eliminate Memory Copies

One reason that the plain Strassen algorithm is not as efficient as the well-optimized version from the lab is that it involves too much memory copying. It needs to split the large matrix into several smaller, equal-sized matrices, and after the computation of the intermediate matrices, it needs to combine them back into a larger matrix. All these operations involve memory copying. A good idea to eliminate these memory accesses is to implement an extra layer of matrix views. When creating new small matrices, we would not actually create the matrix and copy it in memory. Instead, we would create a matrix view, which records the exact area of the large matrix to which the small matrix belongs.

This seems like a very convincing optimization. However, when I tried to implement this optimization, I found that it is not only very difficult to apply but also may not offer any optimization. Even with the above matrix view mechanism, we would still need to create P_1…P_7 matrices to store the intermediate results in every round of computation. Therefore, the majority of the memory copies would still exist. Furthermore, the matrix view breaks cache locality. In the original Strassen algorithm, as we copied the current working part into a small matrix, it would utilize cache spatial locality, as all the data in the matrix is sequential in memory. However, when we add the matrix view, the data in the same matrix is no longer continuous.

After checking online sources, I found that no one has implemented this optimization. Thus, I concluded that it is not a good optimization and decided to abandon it.

4.3.2 A Comparison of Strassen Algorithm Breakpoints

As mentioned earlier, the Strassen algorithm will not be efficient for small matrices, and there should be a breakpoint during the recursion to switch to the triple-for loop computation. I tested different breakpoints to find the best breakpoint for the triple-for loop to take effect. The results are as follows:

rowlen	strassen_32	strassen_64	strassen_128	strassen_256	strassen_512
160	0.008206	0.004003	0.001769	0.00075	0.0007989
208	0.01368	0.00622	0.003003	0.001199	0.001206
320	0.05784	0.03013	0.01462	0.008218	0.005052
496	0.09142	0.0584	0.03356	0.01926	0.01374
736	0.3892	0.2037	0.1127	0.07627	0.05959
1040	0.7983	0.8168	0.4066	0.212	0.1151
1408	2.618	1.371	0.7272	0.4121	0.27
1840	4.187	2.319	1.203	0.6907	0.4811
2336	14.61	6.755	3.657	2.067	1.368
2896	16.01	10.01	5.64	3.106	2.146

Table 3. Comparison of Different Strassen Algorithm Breakpoints

We can see that when the breakpoint is too small, the algorithm runs extremely slowly, as the overhead of Strassen is larger than the time it saves. Fewer levels of Strassen recursion are preferred to achieve better performance.

4.3.3 Comparison of the Strassen Algorithm on Large Matrices that Do Not Fit into the L3 Cache

Since we have a 64MB L3 cache, when the row length exceeds 2000, the matrix will not be able to fit into the L3 cache. We want to measure the performance on matrix sizes that do not fit into the L3 cache.

rowlen	kij_block_omp	strassen_simd
1024	0.2496	0.08552
2176	2.093	0.8592
5376	31.62	9.962
10624	258.2	69.58

Table 4. Comparison of the Strassen Algorithm on Large Matrices that Do Not Fit into the L3 Cache

In the measurements above, we can see that when the matrix cannot fit into the L3 cache, the Strassen algorithm is still efficient and performs better than the optimal kij triple-for loop.

4.3.4 Comparison of Different Numbers of Threads on the SIMD Strassen Algorithm

In this section, I use the previously mentioned SIMD-optimized Strassen algorithm code for testing. As mentioned earlier, I am using a 16-core, 32-thread processor for the experiment. Here, we want to find the most efficient number of threads when running this algorithm. To make the results more accurate, we have increased the breakpoint mentioned above to 512 for this test. This is because only after the breakpoint will we perform OpenMP parallelization. The results are shown below.

rowlen	strassen_omp16	strassen_omp20	strassen_omp24	strassen_omp28	strassen_omp32	strassen_omp36	strassen_omp48
160	0.000995	0.0007386	0.000737	0.0007679	0.0007704	0.0008049	0.0008013
208	0.001575	0.001998	0.002036	0.00121	0.001212	0.001273	0.001233
320	0.005604	0.005401	0.004864	0.004891	0.005072	0.004602	0.00345
496	0.01802	0.02195	0.01824	0.01828	0.01375	0.01371	0.01553
736	0.05448	0.05622	0.05352	0.05277	0.04334	0.04332	0.03429
1040	0.0988	0.1285	0.1098	0.1085	0.09255	0.1153	0.1278
1408	0.2271	0.2643	0.2248	0.2272	0.1918	0.2447	0.2416
1840	0.5302	0.5415	0.4745	0.4579	0.4121	0.4672	0.4194
2336	1.015	1.2	0.982	0.979	0.9559	1.152	1.117
2896	1.806	2.153	1.86	1.779	1.545	1.925	1.765

Table 5. Comparison of Different Numbers of Threads on the SIMD Strassen Algorithm

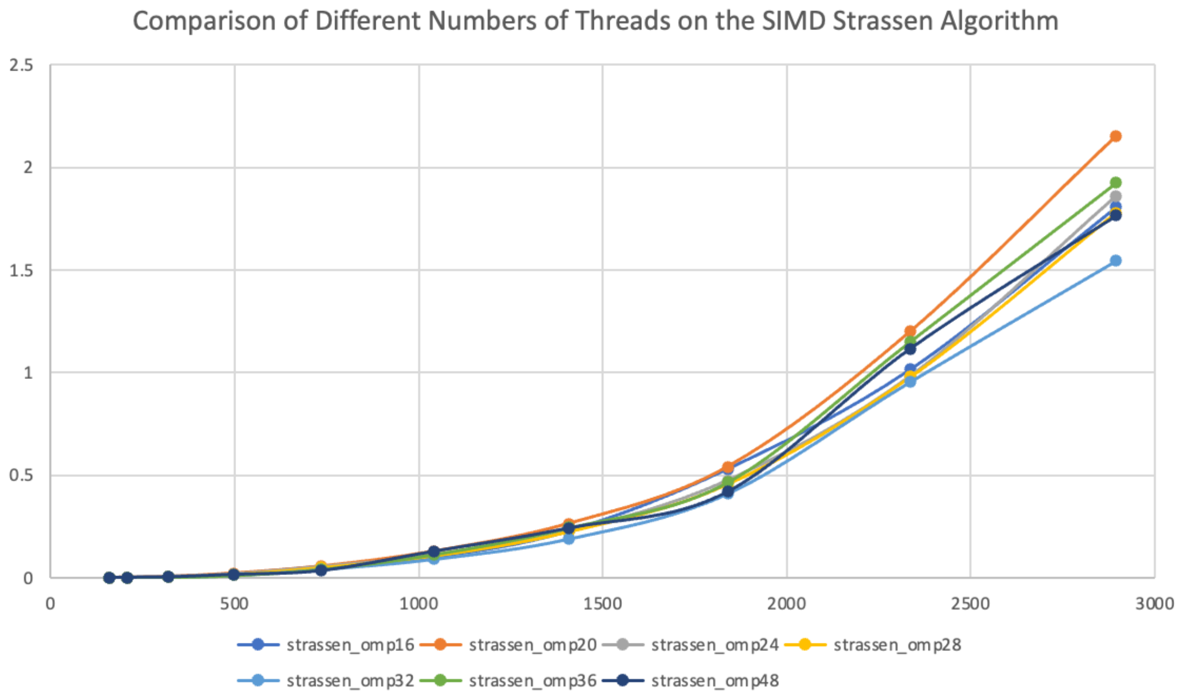

Figure 3. Comparison of Different Numbers of Threads on the SIMD Strassen Algorithm

As shown in the table, 32 threads run the fastest, followed by 16 threads. This is likely because 32 threads utilize the entire CPU resources without too much thread-switching overhead. It is somewhat interesting that the number of threads between 16 and 32 do not perform very well. This might be related to OpenMP itself or task splitting.

5 – Conclusion

In this final project, we first compared the Strassen algorithm with the triple-for loop matrix multiplication algorithm. Then, we listed the optimizations we have already implemented this semester, using the fastest version – the kij triple-for loop with blocking and OpenMP enabled – as our baseline. We took the file from Lab 6 as our base file but created a fantastic building and testing framework using CMake. We added unit tests for all the functions and verified their correctness. We also included performance-based unit tests to measure and compare the performance.

With this file base, we initially implemented the plain Strassen algorithm, strictly following its definition. Although its speed was much faster than the plain kij for loop, the performance still could not surpass the optimized version we completed in the lab. Subsequently, we applied SIMD intrinsics to matrix addition, subtraction, and multiplication. After these modifications, the performance significantly improved and outperformed the optimal version from the lab.