Experiment design In this section, we describe our specific implementation of our matrix multiplication algorithm and the efforts to accelerate it as much as possible.

In order to reduce unnecessary memory usage and save the running time, we represent matrices by one-dimensional arrays, other than two-dimensional. The relationship of the 1-D to 2-D conversion is: for M[i] [j] representing the item on row i+1, column j+1 of a matrix by a 2-D array, M[n*i+j] represents the same item in a 1-D array, where n represents the total number of the columns. That is, the matrix is stores by rows.

We implement the algorithms using Java and C++.

The specific coding implementation of our experiment includes 11 parts:

1. **the standard matrix multiplication (naive method/ brute-force method) for arbitrary size**

   As the pseudocode shows before, the standard matrix multiplication is computed by three nested for-loops.

2. **extend brute-force method by partitioning matrices based on minimum alignment factor for arbitrary size**

   The minimum alignment factor means: factorizing the matrix order N into two factors such that the difference of the two factors is minimum.

3. **normal Strassen's method (for matrix size of $2^n \times 2^n$)**

   As the pseudocode shows before.

4. **extend Strassen's method to arbitrary size, by padding zeros to the outermost so that the matrix size is $2^n \times 2^n$**

5. **optimize Strassen's method by partitioning matrices based on factor of power-of-2 for even order matrices**

6. **extend Strassen's method of <5> to arbitrary size, by padding zeros to the outermost so that the matrix size is $2^n \times 2^n$**

7. **optimize Strassen's method of <4> by adding threshold judgment**

   According to our experimental test, we set the threshold as N=64. Once the size of submatrices is under 64 by 64, stop the Strassen's recursions and switch to the brute-force algorithm.

8. **optimize Strassen's method of <6> by adding threshold judgment**

9. **optimize Strassen's method of <8> by completing the matrix to make one factor be power-of-2**

Other than padding zeros to the outermost to get $2^n \times 2^n$ matrices, we apply a cleverer way of matrix completion: padding appropriate zeros to make its size suitable for a good factorization which enables one factor to be a high-power-of-2.

10. **optimize algorithm by exchanging the sequences of loops: change ijk into ikj**

    The discontinuous memory access is one of the main reasons for the slow computing speed of matrix multiplication. The normal three nested for-loops follows the sequence of ijk:

    ```
    for i <- 1 to n
        for j <- 1 to n
            for k = 1 to n
                C[i,j] += A[i,k]*B[k,j]
    ```

    For each element in C, the corresponding row of A and column of B are multiplied and added in order. Since the matrix is stored by rows, the total discontinuous "jumps" occur $n^3 + n^2 - n$ times.

    In order to accelerate algorithm, we change the sequence into ikj:

    ```
    for i <- 1 to n
        for k <- 1 to n
            s=A[i,k]
            for j = 1 to n
                C[i,j] += s*B[k,j]
    ```

    In this case, the total discontinuous "jumps" occur n^2 times, which is much less than the former case. Thus, the sequence of ijk shall be a faster method. This inversion of the order in computing the matrix multiplication has the maximized accuracy when accessing the CPU cache, avoiding the interruption in the address sequence of the processor.

11. **implementation of MKL (Intel Math Kernel Library)**

    The MKL is an optimized implementation of many math functions exclusively on x86 architecture and processors supports the Intel SIMD instructions, especially in Intel(R) processors. This library is implemented by assembly codes and C++ codes that are extremely optimized by Intel SIMD instructions (SSE,AVX), in order to maximize performance on matrix operations.

The 11 parts above is a process of gradual optimization step by step for the matrix multiplication algorithm. As a result of full investigation and careful analysis, we eventually established our algorithm for accelerating matrix multiplication.

In the experiment we use randomly generated square matrices of increasing size with values in the range [-1,1] for test problems.

The following section presents our experimental results.

Empirical Analysis We split this experimental section. We present experimental results in terms of the common method, algorithm optimization and hardware optimization separately. Our experimental environment is:

CPU: Intel(R) Core(TM) i7-9750HQ@3.85~4.05GHz

RAM: 32GB 2666MHz

1. Naive method and Strassen's algorithm

By implementing naive matrix multiplications and Strassen algorithm on Java and fit the data, the standard data of the two algorithms are

$$\begin{aligned} naive\ method: &\quad T(N) = O(n^{3.371}) \\ Strassen's\ (with\ threshold\ 2): &\quad T(S;2) = O(N^{3.546}) \end{aligned} \tag{1}$$

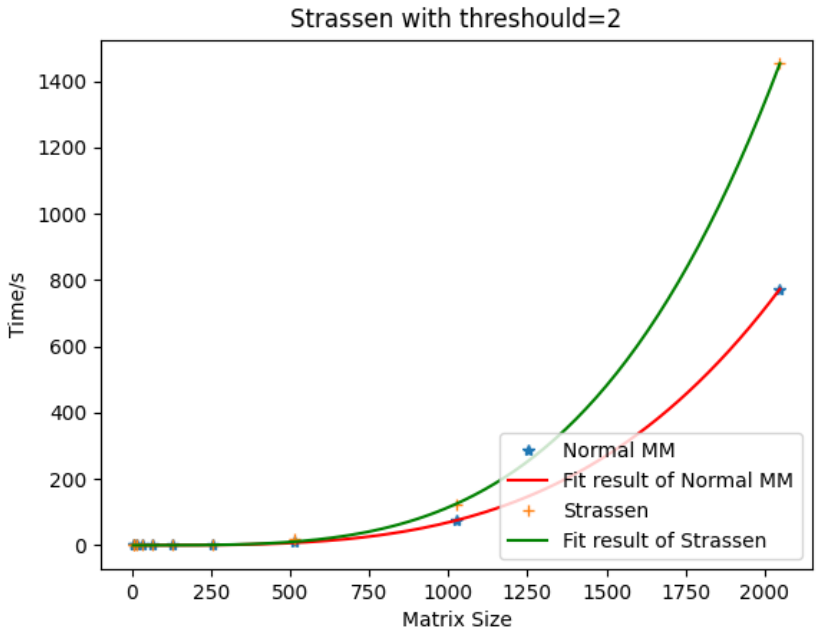

Figure 1:

This result shows that the Strassen's Algorithm is seemingly faster than the brute-force matrix multiplication, but neither of them even reaches the time complexity of $O(n^3)$.

2. Strassen's algorithm with minimum size threshold

The main reason causing the Strassen's algorithm slower than the brute-forcing algorithm is the time costs in trace-backing the recursion. Thus, the time cost of the Strassen's algorithm can be significantly reduced by calibrating the cross-point and setting the minimum size of recursion.

$$\begin{aligned} \textit{naive method}: &\quad T(N) = O(n^{3.384}) \\ \textit{Strassen's (with threshold 32)}: &\quad T(S;32) = O(N^{3.337}) \end{aligned} \tag{2}$$

And the constant of the Strassen's algorithm is about 1/3 of that of the brute-force algorithm.

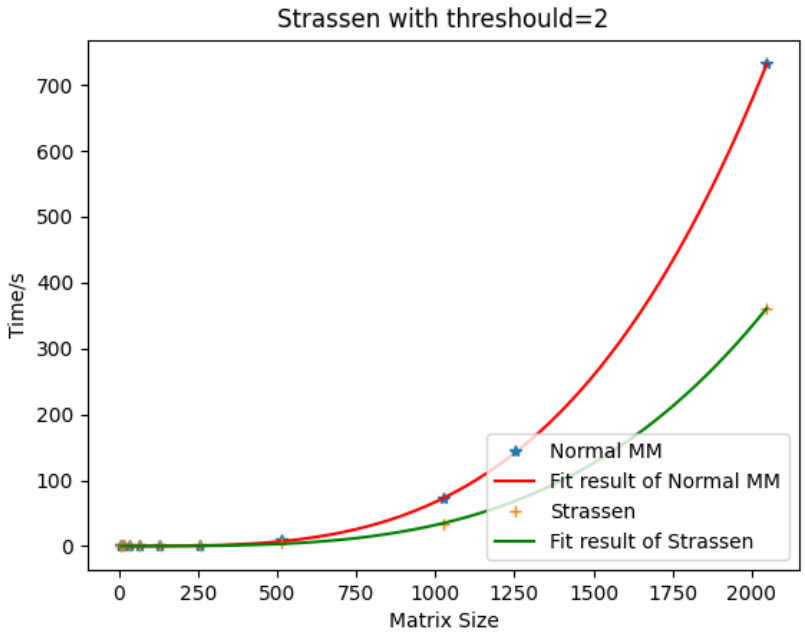

Figure 2:

3. Hardware optimization

(1) Cache alignment

By implementing the ikj sequence of for-loops with improved cache accuracy, as explained before, the Strassen's algorithm can meet the theoretical time complexity:

$$\begin{aligned} T(N) &= O(n^{2.991}) \\ T(S;32) &= O(N^{2.816}) \end{aligned} \tag{3}$$

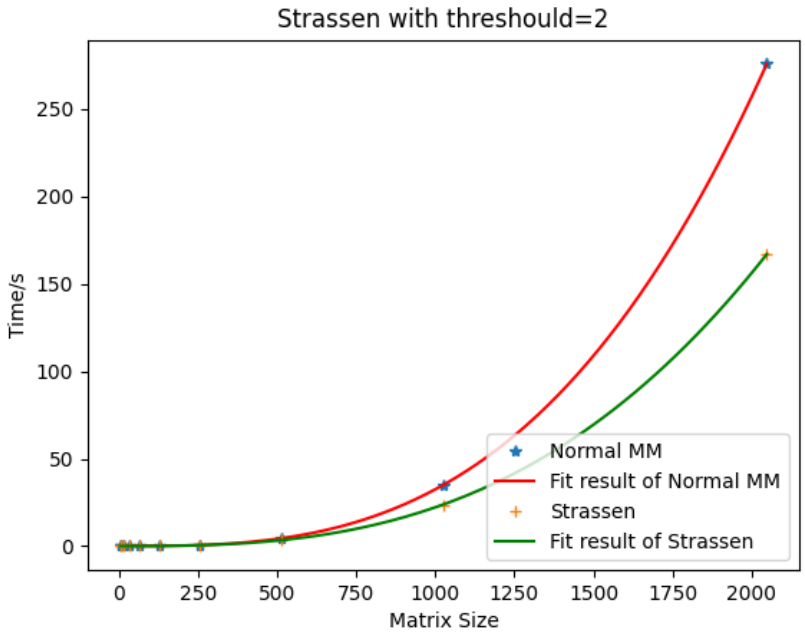
Strassen with threshould=2
Time/s
Matrix Size
0
50
100
150
200
250
0
250
500
750
1000
1250
1500
1750
2000
Normal MM
Fit result of Normal MM
Strassen
Fit result of Strassen

Figure 3:

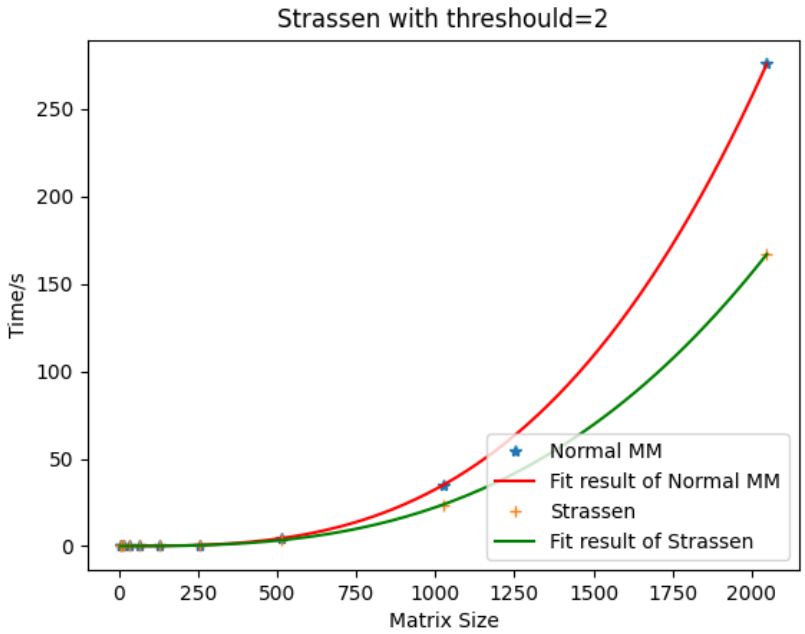
Strassen with threshould=2
Time/s
Matrix Size
0
50
100
150
200
250
0
250
500
750
1000
1250
1500
1750
2000
Normal MM
Fit result of Normal MM
Strassen
Fit result of Strassen

Figure 4:

When the order of the matrix is very large, the length of the row-major array may exceed the maximum capacity of the L1 cache, the speed of the matrix multiplication will be limited. Thus, we implemented sectioned matrix multiplication algorithm. Let s be the sectioning number which the original matrix is divided by, and k be the polynomial degree of the time complexity, divide the matrix into small chunks so that the length of the chunks are below the maximum capacity of the L1 cache.

$$
\begin{aligned}
T(n;s) &= s^k t(\frac{n}{s})^k \\
\frac{n}{k} < BW(L1\ Cache) \Rightarrow t(n/k) &= T(PUSH) + T(MUL) + T(TOP) \\
&\ll 2T(MOV) + T(PUSH) + T(MUL) + T(TOP)
\end{aligned} \tag{4}
$$

And we can get the time complexity by data fitting:

$$
\begin{aligned}
T(N) &= O(n^{3.013}) \\
T(N;s) &= O(n^{2.822})
\end{aligned} \tag{5}
$$

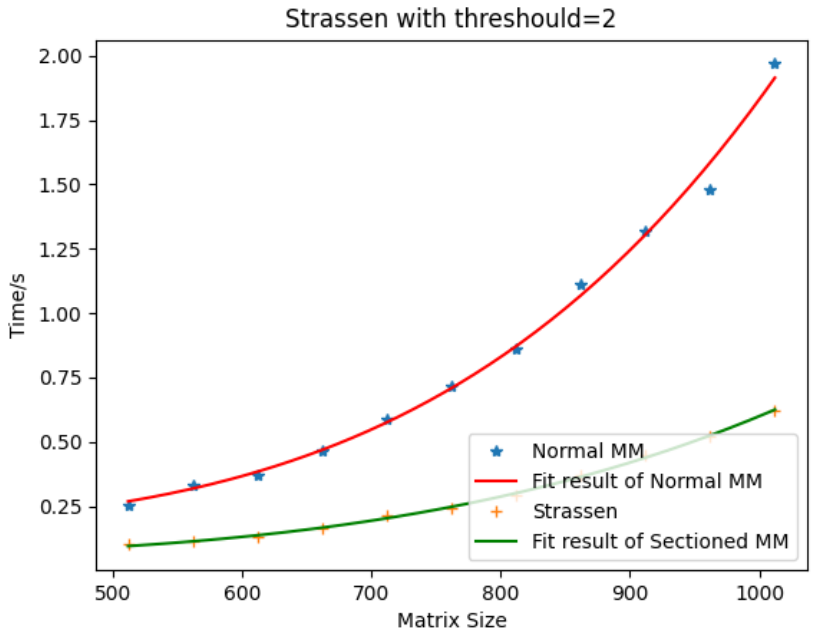

Figure 5:

(2) Implementation with Intel MKL

Under the C++ implementation, the Intel MKL computes the 2500*2500 matrix multiplication within 6000 milliseconds, including filling the arrays.

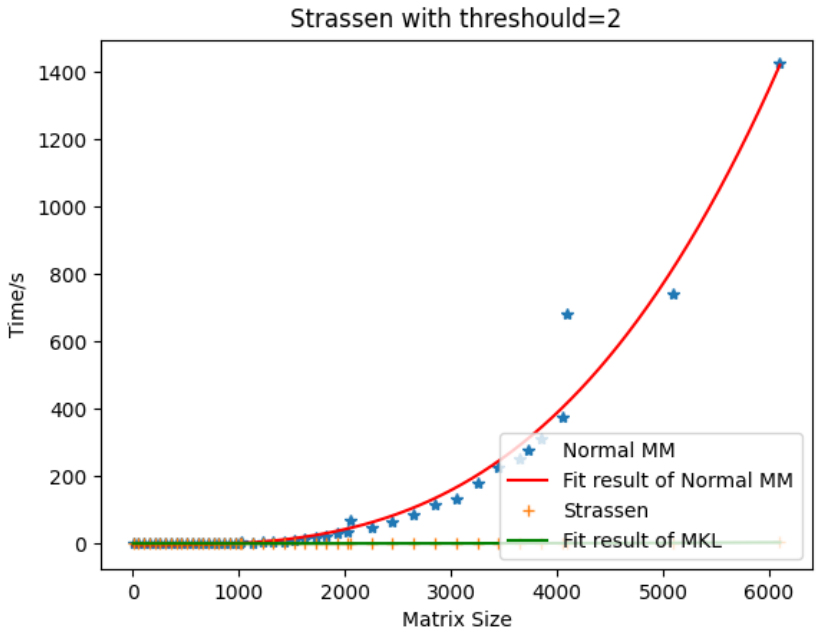

Figure 6:

Under the Java implementation, we adopted to accelerate the Strassen's algorithm, the deletion of the unnecessary array filling progress exploits the full capabilities of the MKL:

$$\begin{aligned} T(N) &= 4.008 \times 10^{-9} n^{3.052} \approx \frac{1}{f_{CPU}} n^{3.052} \\ T(MKL) &= 1.817 \times 10^{-11} n^{2.982} \approx \frac{1}{250 f_{CPU}} n^{2.982} \end{aligned} \tag{6}$$

The extremely optimization based on the SIMD instructions of the Intel CPU made the MKL run at 250x faster than the naive approach using merely sequential operations.

Now we have accelerated our matrix multiplication operation to a quite optimal result.

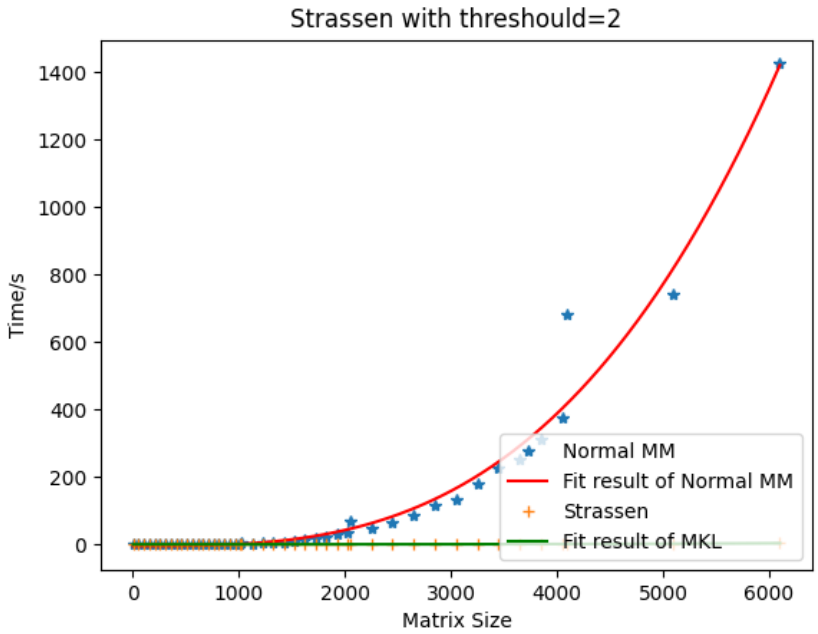
Strassen with threshould=2
Time/s
Matrix Size
0
200
400
600
800
1000
1200
1400
0
1000
2000
3000
4000
5000
6000
Normal MM
Fit result of Normal MM
Strassen
Fit result of MKL

Figure 7: