南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Matrix Multiplication Acceleration
Implemented By Strassen Algorithm and Intel(R) Math Kernel Library

组员: 仇琨元 徐嘉睿 肖锐卓 吴雨飞

Insitute of EEE
SUSTech

December 21, 2020

Contents

Contents

Necessity of Matrix Multiplication Acceleration

The multiplication of two matrices is one of the most basic operations of linear algebra and scientific computing.

Modern signal processing, artificial intelligence and computer vision are all based on the fast and accurate algorithm of matrix multiplication, LU/QR/SVD decomposition and many other operations.

Comparison of the time costs of computing the FFT

CPU	Clock Frequency	DFT	FFT
1941	60 Hz	152.3 y	271.4 d
1971 (4004)	108KHz	30.8 d	3.6 h
1978 (8086)	10MHz	8.0 h	2.3 min
1982 (80286)	20MHz	4.0 h	1.2min
1985 (80386)	33MHz	2.4h	42.6s
1989 (80486)	100MHz	48.0min	14.1s
1995 (Pentium)	200MHz	24.0min	7.0s
1999 (Pentium Ⅲ)	450MHz	10.7min	3.1s
2000 (Pentium 4)	1.4GHz	3.4min	1.0s
2001 (Pentium 4)	2GHz	2.4min	0.7s

Contents

Brute-Force Algorithm

The Strassen Multiplication uses divide-conquer to reduce the time complexity of MM operations. Normal MM uses 3 nested loops to perform the vector dotting and traversal of the rows and columns of the 2 operands:

```
1  STANDARD-MATRIX-MULTIPLY (A,B):
2  let C be a new m*n matrix
3    for i <- 1 to m
4      for j <- 1 to n
5        C[i,j] = 0
6          for k = 1 to p
7              C[i,j] += A[i,k]*B[k,j]
8    return C
```

Brute-Force Algorithm

From the pseudocode above, let MUL, ADD, READ and WRITE refers the corresponding $assembly$ commands of the computer. Each nested loop multiplies the time complexity by its iteration number:

$$\begin{aligned} T(\text{Naive}) &= m \cdot n \cdot p \cdot (T(\text{MUL} + \text{ADD} + \text{READ} + \text{WRITE})) \\ &= \Theta(mnp) \\ m = n = p \Rightarrow T(\text{Naive}) &= \Theta(n^3) \end{aligned} \tag{1}$$

Strassen Algorithm

Pseudocode of the Strassen algorithm:

```
1   STRASSEN (MatrixA,MatrixB)
2       N=MatrixA.rows
3     Let MatrixResult be a new N N matrix
4     if N==1
5       MatrixResult=MatrixA*MatrixB
6     else
7       // DIVIDE: partitioning input Matrices into 4 submatrices each
8         for i <- 0 to N/2
9             for j <- 0 to N/2
10              A11[i][j] <- MatrixA[i][j]
11              A12[i][j] <- MatrixA[i][j + N/2]
12              A21[i][j] <- MatrixA[i + N/2][j]
13              A22[i][j] <- MatrixA[i + N/2][j + N/2]
14
15              B11[i][j] <- MatrixB[i][j]
16              B12[i][j] <- MatrixB[i][j + N/2]
17              B21[i][j] <- MatrixB[i + N/2][j]
18              B22[i][j] <- MatrixB[i + N/2][j + N/2]
```

Strassen Algorithm

```
// CONQUER: here we calculate P1...P7 matrices
    P1 <- STRASSEN(A11, B12-B22) //P1=A11(B12-B22)
    P2 <- STRASSEN(A11+A12, B22) //P2=(A11+A12)B22
    P3 <- STRASSEN(A21+A22, B11) //P3=(A21+A22)B11
    P4 <- STRASSEN(A22, B21-B11) //P4=A22(B21-B11)
    P5 <- STRASSEN(A11+A22, B11+B22) //P5=(A11+A22)(B11+B22)
    P6 <- STRASSEN(A12-A22, B21+B22) //P6=(A12-A22)(B21+B22)
    P7 <- STRASSEN(A11-A21, B11+B12) //P7=(A11-A21)(B11+B12)

     // calculate the result submatrices
    C11 <- P5 + P4 - P2 + P6
    C12 <- P1 + P2
    C21 <- P3 + P4
    C22 <- P5 + P1 - P3 - P7

     // MERGE: put them together and make our resulting Matrix
    for i <- 0 to N/2
       for j <- 0 to N/2
         MatrixResult[i][j] <- C11[i][j]
         MatrixResult[i][j + N/2] <- C12[i][j]
         MatrixResult[i + N/2][j] <- C21[i][j]
         MatrixResult[i + N/2][j + N/2] <- C22[i][j]
    return MatrixResult
```

Strassen Algorithm

Use the recursion function to evaluate the time complexity of the algorithm. For $n \geqslant 2$,

$$\begin{aligned} & T(2n) = 7\,T(n) + \Theta(n^2) \\ \log_2 n = k \Rightarrow\ & T(k+1) = 7\,T(k) + \Theta(2^{2k}) \\ \Rightarrow\ & T(n) = O(n^{\log_2 7}) \\ & \approx O(n^{2.81}) < \Theta(n^3) \end{aligned} \tag{2}$$

The n in this result is the number of the $Operation$, which is in propotition of the order of the matrix:

$$t(n) = \alpha \mathsf{Rows}, \alpha \in (1, \infty) \tag{3}$$

Contents

Experimental Enviornments and Test Code

Experimental Enviormnent

CPU:Intel(R) Core(TM) i7-9750HQ@3.85~4.05GHz
RAM:32GB 2666MHz

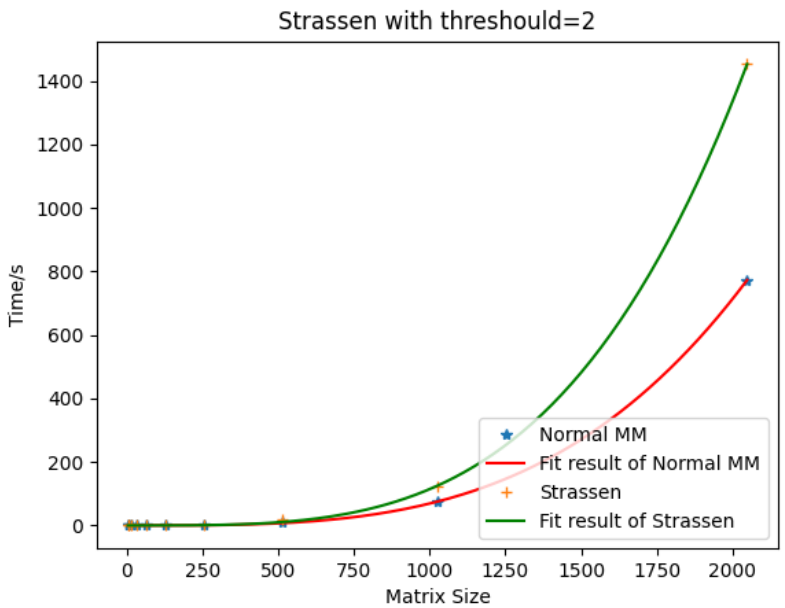
```
1   /**TEST CODE**/
2   #include <...>
3   int main(int argc, char *argv[]) {
4     int turns = 10, rows = 2, cols = 2, interm = 0;
5     randinit();
6     for (int i = 0; i <= turns; i++) {
7       LARGE_INTEGER t1, t2, tc;
8       QueryPerformanceFrequency(&tc);
9       QueryPerformanceCounter(&t1);
10      /**Generate Rows and Elements**/
11      printf_s("%3d   ", rows);
12      /**Function To Be Timed**/
13      QueryPerformanceCounter(&t2);
14      time = (double) (t2.QuadPart - t1.QuadPart) / (double) tc.QuadPart;
15      printf_s("%.3f\n", time);
16      }
17    getchar();
18    return 0;
19  }
```

Contents

Naive Matrix Multiplication implemented using C++

By implementing naive MM and Strassn algorithm on C++ and fit the data, the standard data of the two algorithms are

$$\begin{aligned} T(N) &= O(n^{3.371}) \\ T(S;2) &= O(n^{3.546}) \end{aligned} \tag{4}$$



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

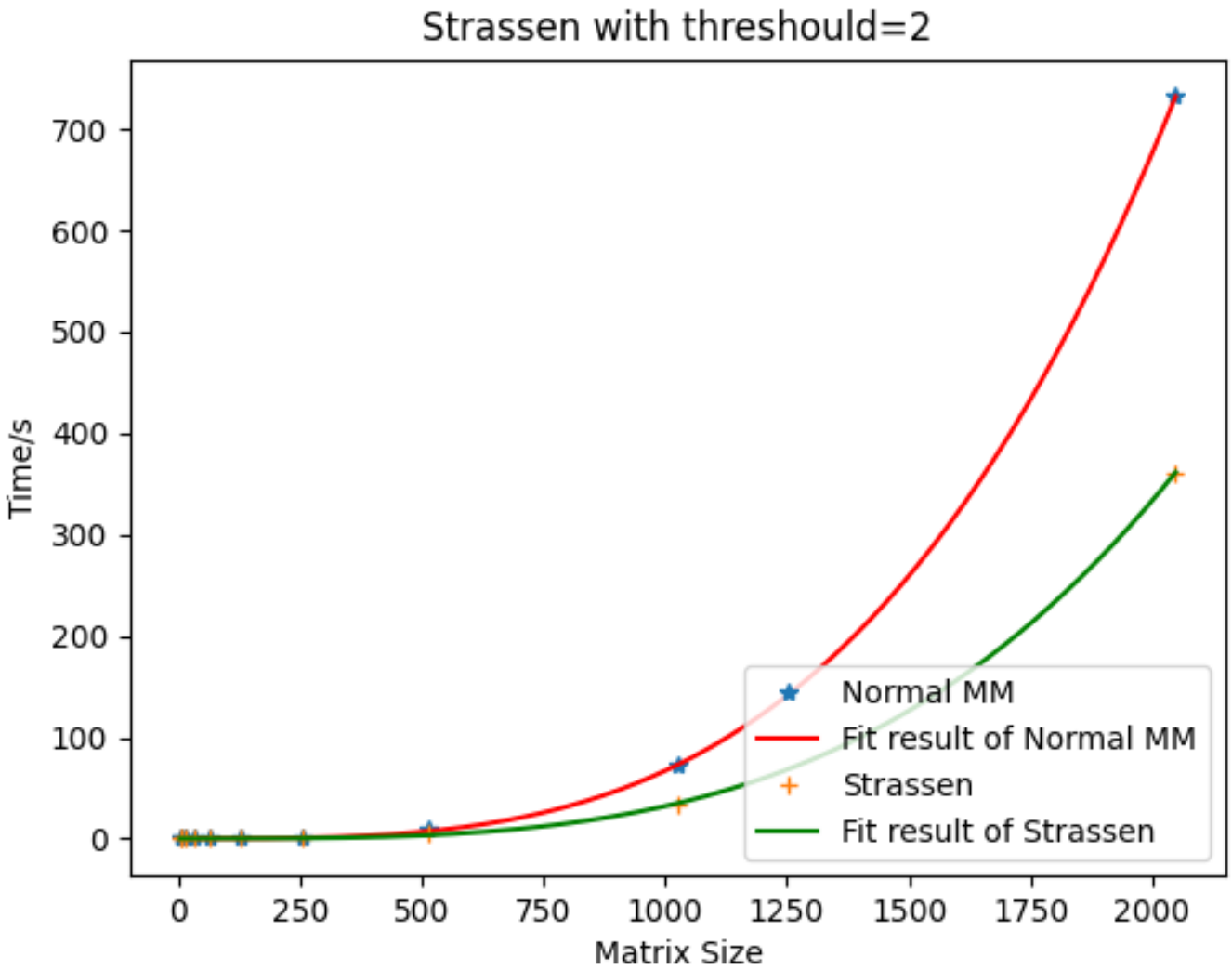
Strassen Algorithm with Minimun Size Threshold

The main reason causing the Strassen Algorithm slower than the brute-forcing algorithm is the time costs in tracebacking the recursion. Thus, the time cost of the Strassen Algorithm can be significantly reduced by calibrating the crosspoint and setting the minimum size of recursion.

$$\begin{aligned} T(N) &= O(n^{3.384}) \\ T(S;32) &= O(n^{3.337}) \end{aligned} \tag{5}$$

And the constant of the Strassen Algorithm is about $1/3$ of that of the brute-forcing algorithm.

Strassen Algorithm with Minimun Size Threshold



Contents

Acceleration of Brute-Force MM

To improve the performance of the brute-forcing MM, use a alternative order of multiplication:

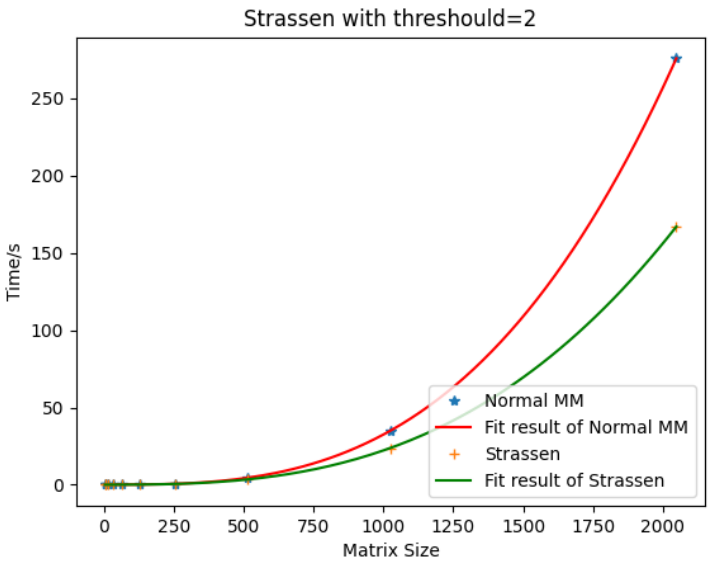
```
1    Alternative-Order-MM(Matrix A,Matrix B):
2      for i in [0:length(A)]:
3        for k in [0:length(A[0])]:
4          s=A[i][k]
5          for j in [0:length(B[0])]:
6            Result[i][k]+=s*B[k][j]
7          end
8        end
9      end
```

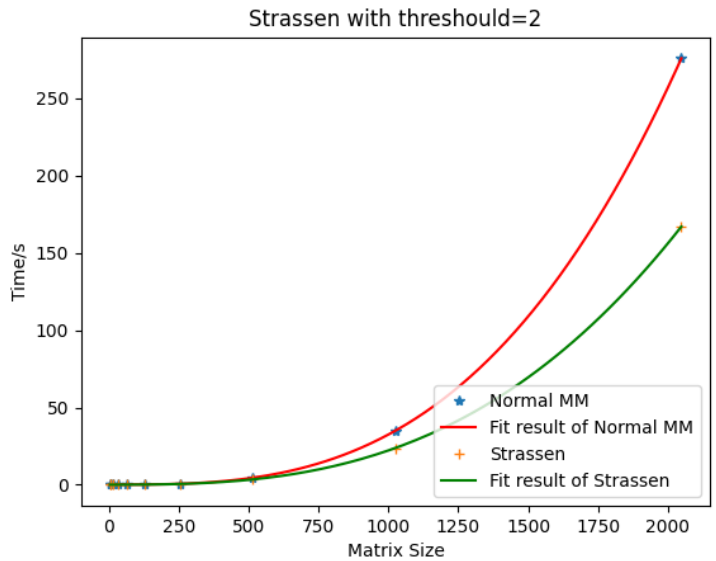
This inversion of the order in computing the matrix multiplication has the maximized accuracy when accessing the CPU cache, avoiding the interruption in the address sequence of the processor.

Strassen Algorithm with Improved Cache Accuracy

With the patch above, the Strassen algorithm can meet the therotical time complexity:

$$T(N) = O(n^{2.991})$$
$$T(S;32) = O(n^{2.816}) \tag{6}$$





Sectioned Matrix Multiplication Algorithm

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. 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.

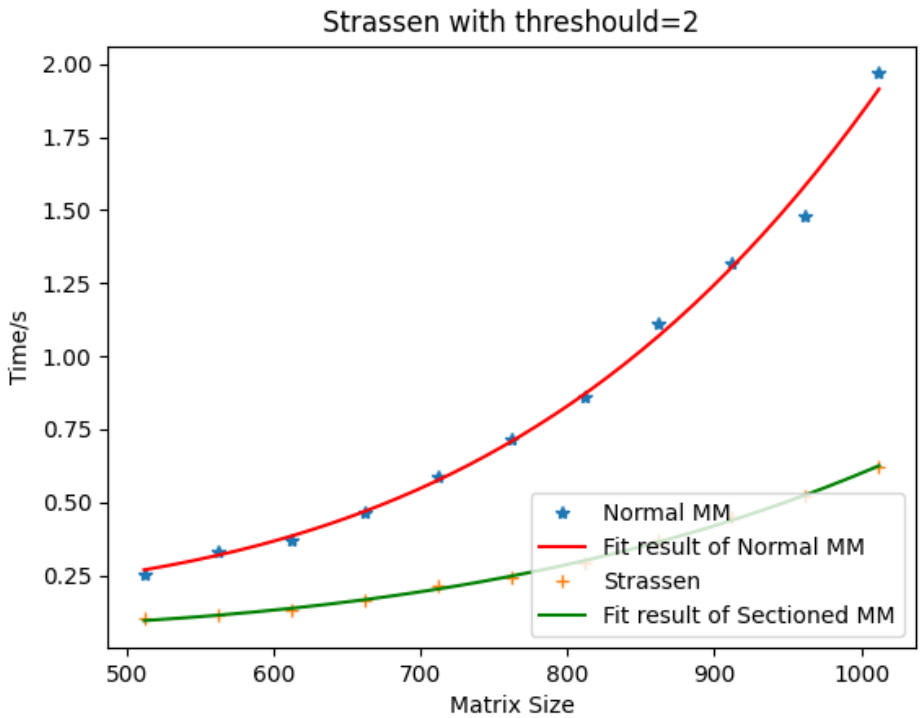
$$T(n;s) = s^k t\left(\frac{n}{s}\right)^k$$

$$\frac{n}{k} < BW(\text{L1 Cache}) \Rightarrow t(n/k) = T(\text{PUSH}) + T(\text{MUL}) + T(\text{POP})$$

$$<< 2\,T(\text{MOV}) + T(\text{PUSH}) + T(\text{MUL}) + T(\text{POP}) \tag{7}$$

Sectioned Matrix Multiplication Algorithm

$$T(N) = O(n^{3.013})$$
$$T(N; s) = O(2.822) \tag{8}$$



Intel Math Kernel Library

Intel Math Kernel Library

The **MKL** is a 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.

C++ Test of double MM

```
1  int mkldgemm(int rowa, int cola, int colb) {
2  using namespace std;
3  double *A, *B, *C;
4  int a = 1, b = 1;
5  A = (double *) MKL_malloc(rowa * cola * sizeof(double), 64);
6  B = (double *) MKL_malloc(cola * colb * sizeof(double), 64);
7  C = (double *) MKL_malloc(rowa * colb * sizeof(double), 64);
8  for (int i = 0; i < rowa * cola; ++i) {
9      A[i] = randgen(-500, 500);
10     }
11 for (int i = 0; i < cola * colb; ++i) {
12     B[i] = randgen(-500, 500);
13     }
14 for (int i = 0; i < rowa * colb; ++i) {
15     C[i] = randgen(-650, 390) * randgen(-500, 500);
16     }
17 cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
18   rowa, colb, cola, 1, A, cola,
19   B, colb, b, C, colb);
20 MKL_free(A);
21 MKL_free(B);
22 MKL_free(C);
23 return 0;
24 }
```

Performance of Intel MKL

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

```
Rows of A:500, Cols of A:2432
Rows of B:2432, Cols of B:500 time=308.777800 ms
Rows of A:700, Cols of A:2789
Rows of B:2789, Cols of B:700 time=548.609200 ms
Rows of A:900, Cols of A:2168
Rows of B:2168, Cols of B:900 time=553.480600 ms
Rows of A:1100, Cols of A:2793
Rows of B:2793, Cols of B:1100 time=835.885200 ms
Rows of A:1300, Cols of A:2073
Rows of B:2073, Cols of B:1300 time=815.472500 ms
Rows of A:1500, Cols of A:2293
Rows of B:2293, Cols of B:1500 time=1045.739900 ms
Rows of A:1700, Cols of A:2738
Rows of B:2738, Cols of B:1700 time=1399.515300 ms
Rows of A:1900, Cols of A:2189
Rows of B:2189, Cols of B:1900 time=1349.106500 ms
Rows of A:2100, Cols of A:2462
Rows of B:2462, Cols of B:2100 time=1649.516100 ms
Rows of A:2300, Cols of A:2453
Rows of B:2453, Cols of B:2300 time=1809.381100 ms
Rows of A:2500, Cols of A:2916
Rows of B:2916, Cols of B:2500 time=2278.926100 ms
```

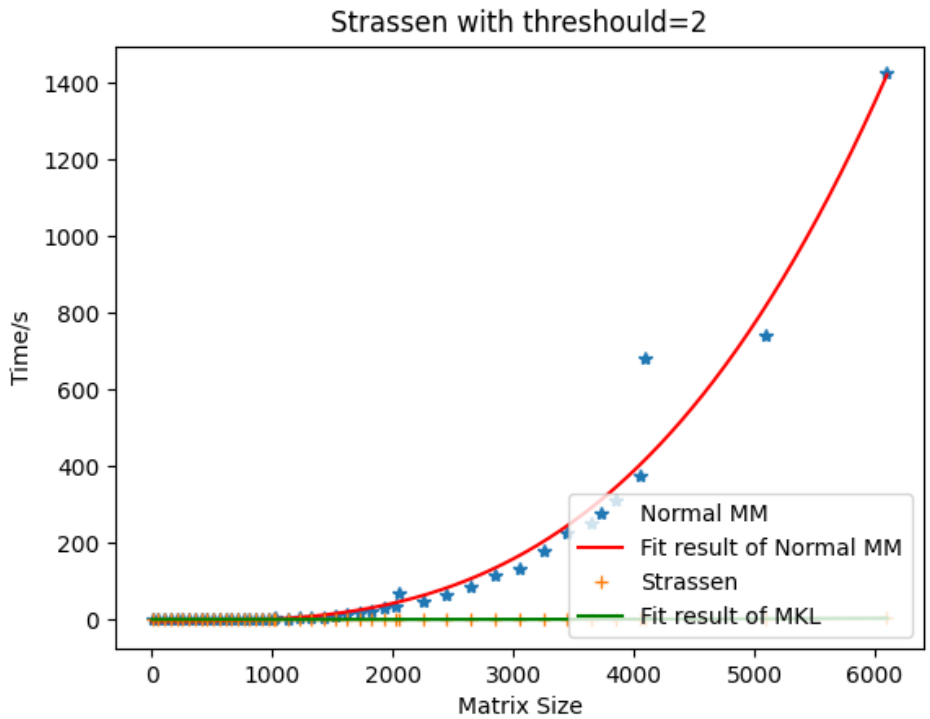
Performance of Intel MKL

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

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

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

Performance of Intel MKL



Contents

Project Management via GitHub

Thank you for your attention!