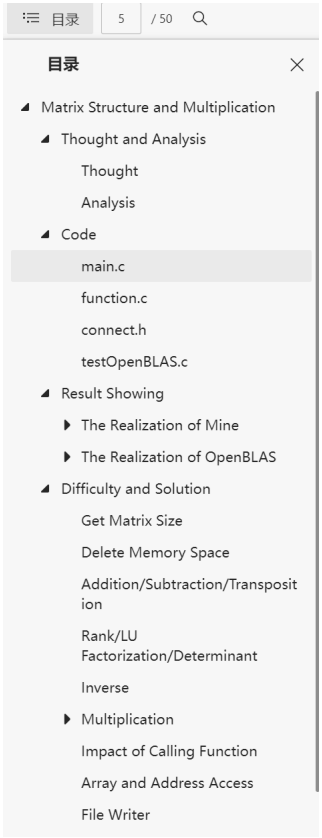
Matrix Structure and Multiplication

NAME： 林洁芳

ID : 12011543

Note: You can connect to each specific section through the directory.



Thought and Analysis

Thought

First of all, you can realize the basic operations between matrices, such as **matrix addition, matrix subtraction, matrix multiplication, inverse, determinant, rank, transpose, stepwise matrix, LU decomposition, adjoint matrix and so on**.

Secondly, on the basis of the last project, f**urther attempts are made to improve the operation efficiency of matrix multiplication**.

Finally, **OpenBLAS, a mature library, is used to compare the efficiency of its operation matrix multiplication, matrix addition, matrix subtraction and transpose with the operating efficiency of its own functions**, and the results are analyzed.

Analysis

In the realization of matrix method, the difficulty lies in how to transform mathematical method into C language.

In terms of matrix multiplication optimization, **compilation optimization, parallel optimization, hardware optimization (IKJ), CUDA optimization, addressing optimization, SIMD optimization and algorithm optimization** should be analyzed, and they should be properly combined to further improve the running speed.

In the comparison with OpenBLAS, objective analysis is made on the comparison result through its function characteristics.

In addition, the following **libraries** are used for this project :

```
#include <stdio.h>
#include "connect.h"
#include <malloc.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <string.h>
#include <immintrin.h>
```

Code

main.c

The file contains the main method body.

In the main method body, the matrix manipulation function and the file processing function are called and the total running time is calculated.

```
#include <stdio.h>
#include "connect.h"
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <immintrin.h>

#pragma GCC optimize(3)

int main(int argc, char **argv) {
    struct timeval a, b, t_strassen, t_strassen_f, tv, f_tv;
    gettimeofday(&a, NULL);
    char *fileA = NULL;
    char *fileB = NULL;
    char *fileC = NULL;
    if (argc > 1) {
        fileA = argv[1];
        fileB = argv[2];
        fileC = argv[3];
    }
    p_Matrix A = createMatrix(fileA);
    p_Matrix B = createMatrix(fileB);
    FILE *fp = fopen(fileC, "w");

    printf("A's transposition cost: \n");
```

```
    p_Matrix tran_A = transpose(A->m_matrix, A->m_row, A->m_column);
    if (tran_A != NULL) {
        fileWriter(fp, tran_A);
        deleteMatrix(tran_A);
    }

    printf("B's transposition cost: \n");
    p_Matrix tran_B = transpose(B->m_matrix, B->m_row, B->m_column);
    if (tran_B != NULL) {
        fileWriter(fp, tran_B);
        deleteMatrix(tran_B);
    }

    p_Matrix sub = subtraction(A->m_matrix, B->m_matrix, A->m_row, A->m_column,
B->m_row, B->m_column);
    if (sub != NULL) {
        fileWriter(fp, sub);
        deleteMatrix(sub);
    }

    p_Matrix pul = plus(A->m_matrix, B->m_matrix, A->m_row, A->m_column, B-
>m_row, B->m_column);
    if (pul != NULL) {
        fileWriter(fp, pul);
        deleteMatrix(pul);
    }

    p_Matrix ladder_A = rank(A->m_matrix, A->m_row, A->m_column);
    A->m_rank = ladder_A->m_rank;
    printf("The A's rank is: %d \n", A->m_rank);
    fileWriter(fp, ladder_A);
    deleteMatrix(ladder_A);

    p_Matrix ladder_B = rank(B->m_matrix, B->m_row, B->m_column);
    B->m_rank = ladder_B->m_rank;
    printf("The B's rank is: %d \n", B->m_rank);
    fileWriter(fp, ladder_B);
    deleteMatrix(ladder_B);

    if(A->m_rank != A->m_row)
        A->hasLU = False;
    if(B->m_rank != B->m_row)
        B->hasLU = False;

    printf("A's determinant is: \n");
    det(A);
    printf("B's determinant is: \n");
    det(B);

    if(A->hasLU == True){
        p_Matrix low_A = (Matrix *) malloc(sizeof(Matrix));
        p_Matrix up_A = (Matrix *) malloc(sizeof(Matrix));
        low_A->m_row = A->m_row;
        up_A->m_row = A->m_row;
        low_A->m_column = A->m_column;
        up_A->m_column = A->m_column;
        low_A->m_matrix = (float *) malloc(sizeof(float) * A->m_row * A-
>m_column);
```

```
        up_A->m_matrix = (float *) malloc(sizeof(float) * A->m_row * A->m_column);
        for(int i = 0 ; i < A->m_row ; ++i){
            for(int j = 0 ; j < A->m_column ; ++j){
                *(low_A->m_matrix + i * A->m_column + j) = 0.f;
                *(up_A->m_matrix + i * A->m_column + j) = 0.f;
            }
        }
        printf("A's LU factorization cost: \n");
        LU(A->m_matrix, low_A->m_matrix, up_A->m_matrix, A->m_row);
        fileWriter(fp, low_A);
        fileWriter(fp, up_A);
        deleteMatrix(low_A);
        deleteMatrix(up_A);
    }
    if(B->hasLU == True){
        p_Matrix low_B = (Matrix *) malloc(sizeof(Matrix));
        p_Matrix up_B = (Matrix *) malloc(sizeof(Matrix));
        low_B->m_row = B->m_row;
        up_B->m_row = B->m_row;
        low_B->m_column = B->m_column;
        up_B->m_column = B->m_column;
        low_B->m_matrix = (float *) malloc(sizeof(float) * B->m_row * B->m_column);
        up_B->m_matrix = (float *) malloc(sizeof(float) * B->m_row * B->m_column);
        for(int i = 0 ; i < B->m_row ; ++i){
                for(int j = 0 ; j < B->m_column ; ++j){
                    *(low_B->m_matrix + i * B->m_column + j) = 0.f;
                    *(up_B->m_matrix + i * B->m_column + j) = 0.f;
            }
        }
        printf("B's LU factorization cost: \n");
        LU(B->m_matrix, low_B->m_matrix, up_B->m_matrix, B->m_row);
        fileWriter(fp, low_B);
        fileWriter(fp, up_B);
        deleteMatrix(low_B);
        deleteMatrix(up_B);
    }

    printf("A's inversion cost: \n");
    p_Matrix inv_A = inverse(A);
    if (inv_A != NULL) {
        fileWriter(fp, inv_A);
        deleteMatrix(inv_A);
    }

    printf("B's inversion cost: \n");
    p_Matrix inv_B = inverse(B);
    if (inv_B != NULL) {
        fileWriter(fp, inv_B);
        deleteMatrix(inv_B);
    }

    p_Matrix mul = multiply(A->m_matrix, B->m_matrix, A->m_row, A->m_column, B->m_row, B->m_column);
    if (mul != NULL) {
        fileWriter(fp, mul);
```

```
        deleteMatrix(mul);
    }

    p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
    C->m_column = B->m_column;
    C->m_row = A->m_row;
    C->m_matrix = (float *) malloc(sizeof(float) * C->m_row * C->m_column);
    for(int i = 0 ; i < A->m_row ; ++i){
           for(int j = 0 ; j < B->m_column ; ++j){
               *(C->m_matrix + i * B->m_column + j) = 0.f;
        }
    }
    if (A->m_column == B->m_row) {
        strassen(A->m_matrix, B->m_matrix, C->m_matrix, A->m_row, A->m_column,
B->m_column, A->m_row);
        fileWriter(fp, C);
    } else printf("MULTIPLY FAILURE!");

    // gettimeofday(&tv, NULL);
    // multiply_func(A->m_matrix, B->m_matrix, C->m_matrix, A->m_row, A-
>m_column, B->m_row, B->m_column);
    // gettimeofday(&f_tv, NULL);
    // printf("%.3f \n", (f_tv.tv_sec - tv.tv_sec) * (double) 1000 +
(f_tv.tv_usec - tv.tv_usec) / (double) 1000);
    for(int i = 0 ; i < A->m_row ; ++i){
        for(int j = 0 ; j < B->m_column ; ++j){
               *(C->m_matrix + i * B->m_column + j) = 0.f;
        }
    }
    mulSIMD(C, A, B);
    fileWriter(fp, C);
    deleteMatrix(C);
    deleteMatrix(A);
    deleteMatrix(B);
    fclose(fp);
    gettimeofday(&b, NULL);
    printf("The running time is %.3f ms\n",
           (b.tv_sec - a.tv_sec) * (double) 1000 + (b.tv_usec - a.tv_usec) /
(double) 1000);
    return 0;
}
```

function.c

Headers, declared global variables, and compiler optimizations enabled are shown below.

```
#include <stdio.h>
#include "connect.h"
#include <malloc.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <string.h>
#include <immintrin.h>

#pragma GCC optimize(3)
struct timeval tv, f_tv;
```

This method realizes the calculation of the row and column of the matrix in the text file, and the initialization of the matrix structure, and returns a pointer to the matrix structure.

```
p_Matrix createMatrix(char *fileName) {
    p_Matrix matrix = (Matrix *) malloc(sizeof(Matrix));
    matrix->m_row = 0;
    matrix->m_column = 0;
    matrix->m_rank = 0;
    matrix->hasInversion = True;
    matrix->hasLU = True;
    int num = 0;
    char c;
    FILE *file = fopen(fileName, "rb");
    while (fscanf(file, "%c", &c) != EOF) {
        if (c == '\n') {
            matrix->m_row++;
            num++;
        }
        if (c == ' ')
            num++;
    }
    num++;
    matrix->m_column = num / matrix->m_row;
    int i = 0;
    matrix->m_matrix = (float *) malloc(sizeof(float) * num);
    FILE *fid = fopen(fileName, "rb");
    while (!feof(fid)) {
        fscanf(fid, "%f", matrix->m_matrix + (i++));
    }
    fclose(file);
    fclose(fid);
    return matrix;
}
```

This method realizes the dynamic application of useless matrix structure variable memory release.

```
void deleteMatrix(p_Matrix matrix) {
    free(matrix->m_matrix);
    free(matrix);
}
```

This method realizes the deep copy of the matrix structure and returns the pointer of the new matrix structure.

```
p_Matrix copyMatrix(p_Matrix matrix) {
    p_Matrix newMatrix = (Matrix *) malloc(sizeof(Matrix));
    newMatrix->m_row = matrix->m_row;
    newMatrix->m_column = matrix->m_column;
    newMatrix->m_matrix = (float *) malloc(sizeof(float) * matrix->m_column *
matrix->m_row);
    const float *a = matrix->m_matrix;
    float *b = newMatrix->m_matrix;
    for (int i = 0; i < matrix->m_row * matrix->m_column; ++i) {
        *(b++) = *(a++);
    }
    return newMatrix;
}
```

This method realizes matrix multiplication, including the test of multiplicability, and improves the efficiency of matrix multiplication by using hardware optimization, multi-parallel optimization, compilation optimization and loop rolling.

```
p_Matrix multiply(const float *A, const float *B, int A_row, int A_column, int
B_row, int B_column) {
    if (A_column != B_row) {
        printf("MULTIPLY FAILURE!");
        return NULL;
    }
    p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
    C->m_row = A_row;
    C->m_column = B_column;
    C->m_matrix = (float *) malloc(sizeof(float) * A_row * B_column);
    float *tem = C->m_matrix;
    for (int i = 0; i < A_row; ++i) {
        for (int j = 0; j < B_column; ++j) {
            *(tem++) = 0.0F;
        }
    }
    float *C_mat = C->m_matrix;
    gettimeofday(&tv, NULL);
    float c;
    int i, j, k;
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            c = *(A + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(C_mat + i * B_column + j) += c * *(B + k * B_column + j);
                *(C_mat + i * B_column + j + 1) += c * *(B + k * B_column + j +
1);
                *(C_mat + i * B_column + j + 2) += c * *(B + k * B_column + j +
2);
                *(C_mat + i * B_column + j + 3) += c * *(B + k * B_column + j +
3);
            }
        }
    }
```

```
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of multiplication costs %.3f ms\n", period);
    return C;
}
```

This method implements matrix multiplication, which is mainly used to observe the time of function call, so as to make a fairer comparison with OpenBLAS.

```
void multiply_func(const float *A, const float *B, float *C, int A_row, int
A_column, int B_row, int B_column) {
    if (A_column != B_row) {
        printf("MULTIPLY FAILURE!");
        return;
    }
    float c;
    int i, j, k;
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            c = *(A + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(C + i * B_column + j) += c * *(B + k * B_column + j);
                *(C + i * B_column + j + 1) += c * *(B + k * B_column + j + 1);
                *(C + i * B_column + j + 2) += c * *(B + k * B_column + j + 2);
                *(C + i * B_column + j + 3) += c * *(B + k * B_column + j + 3);
            }
        }
    }
}
```

In these methods, matrix addition and subtraction are implemented, and the additivity and subtraction of two matrices are tested, and compilation optimization, multi-parallel optimization and loop rolling are used to improve the execution efficiency of calculation.

```
//加法
p_Matrix plus(const float *A, const float *B, int A_row, int A_column, int
B_row, int B_column) {
    if (A_column != B_column || A_row != B_row) {
        printf("PLUS FAILURE!");
        return NULL;
    }
    p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
    C->m_row = A_row;
    C->m_column = A_column;
    C->m_matrix = (float *) malloc(sizeof(float) * C->m_row * C->m_column);
    const float *a = A, *b = B;
    float *c = C->m_matrix;
    gettimeofday(&tv, NULL);
#pragma omp parallel for
    for (int i = 0; i < C->m_row; i++) {
        for (int j = 0; j < C->m_column; j += 4) {
            *(c++) = *(a++) + *(b++);
            *(c++) = *(a++) + *(b++);
            *(c++) = *(a++) + *(b++);
```

```
            *(c++) = *(a++) + *(b++);
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of addition costs %.3f ms\n", period);
    return C;
}
//减法
p_Matrix subtraction(const float *A, const float *B, int A_row, int A_column,
int B_row, int B_column) {
    if (A_column != B_column || A_row != B_row) {
        printf("SUBTRACTION FAILURE!");
        return NULL;
    }
    p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
    C->m_row = A_row;
    C->m_column = A_column;
    C->m_matrix = (float *) malloc(sizeof(float) * C->m_row * C->m_column);
    const float *a = A, *b = B;
    float *c = C->m_matrix;
    gettimeofday(&tv, NULL);
#pragma omp parallel for
    for (int i = 0; i < C->m_row; i++) {
        for (int j = 0; j < C->m_column; j += 4) {
            *(c++) = *(a++) - *(b++);
            *(c++) = *(a++) - *(b++);
            *(c++) = *(a++) - *(b++);
            *(c++) = *(a++) - *(b++);
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of subtraction costs %.3f ms\n", period);
    return C;
}
```

In this method, matrix transpose is implemented, and compilation optimization, multi-parallel optimization and loop rolling are used to improve the execution efficiency of matrix transpose calculation.

```
p_Matrix transpose(const float *A, int A_row, int A_column) {
    p_Matrix t_matrix = (Matrix *) malloc(sizeof(Matrix));
    t_matrix->m_column = A_row;
    t_matrix->m_row = A_column;
    t_matrix->m_matrix = (float *) malloc(sizeof(float) * A_column * A_row);
    const float *a = A;
    float *t = t_matrix->m_matrix;
    gettimeofday(&tv, NULL);
#pragma omp parallel for
    for (int i = 0; i < t_matrix->m_row; ++i) {
        for (int j = 0; j < t_matrix->m_column; j += 4) {
            *(t + i * t_matrix->m_column + j) = *(a + j * A_column + i);
            *(t + i * t_matrix->m_column + j + 1) = *(a + (j + 1) * A_column +
i);
```

```
            *(t + i * t_matrix->m_column + j + 2) = *(a + (j + 2) * A_column +
i);
            *(t + i * t_matrix->m_column + j + 3) = *(a + (j + 3) * A_column +
i);
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of transposition costs %.3f ms\n", period);
    return t_matrix;
}
```

det() realizes the determinant of the matrix, in which compilation optimization and loop rolling are used, and **IntMultiply()** is called to realize the multiplication of large numbers, multi-parallel optimization is realized in **IntMultiply()**, and **swap()** is called to realize the exchange of matrix rows.

```
void swap(p_Matrix matrix, int i, int j) {
    float *temp = (float *) malloc(sizeof(float) * matrix->m_column);
    float *t = temp;
    float *m = matrix->m_matrix;
    for (int k = 0; k < matrix->m_column; ++k)
        *(t + k) = *(m + i * matrix->m_column + k);
    for (int k = 0; k < matrix->m_column; ++k)
        *(m + i * matrix->m_column + k) = *(m + j * matrix->m_column + k);
    for (int k = 0; k < matrix->m_column; ++k)
        *(m + j * matrix->m_column + k) = *(t++);
    free(temp);
}

void IntMultiply(char *a, const char *b, int a_len, int b_len) {
    int c[a_len + b_len];
    memset(c, 0, sizeof(int) * (a_len + b_len));
#pragma omp parallel for
    for (int i = 0; i < a_len; ++i) {
        for (int j = 0; j < b_len; ++j) {
            c[i + j + 1] += (a[i] - 48) * (b[j] - 48);
        }
    }
    for (int i = a_len + b_len - 1; i > 0; --i) {
        if (c[i] > 9) {
            c[i - 1] = c[i - 1] + c[i] / 10;
            c[i] = c[i] % 10;
        }
    }
    int zeroNum = 0;
    int j = 0;
    for (int i = 0; i < a_len + b_len; ++i) {
        if (c[i] == 0)
            zeroNum++;
        if (zeroNum != i + 1) {
            a[j] = (char) (c[i] + 48);
            j++;
        }
    }
}
```

```
void det(p_Matrix matrix) {
    if (matrix->m_column != matrix->m_row) {
        matrix->hasInversion = False;
        printf("DET FAILURE!");
        return;
    }
    p_Matrix mat = copyMatrix(matrix);
    int m_swap = 0;
    float max = 0.0F;
    int max_row_num = 0;
    float temp = 0.0F;
    int det = 1;
    float *m = mat->m_matrix;
    gettimeofday(&tv, NULL);
    for (int i = 0; i < mat->m_row - 1; ++i) {
        max = fabsf(*(m + i * mat->m_column + i));
        max_row_num = i;
        for (int j = i + 1; j < mat->m_row; ++j) {
            if (max < fabsf(*(m + j * mat->m_column + i))) {
                max = fabsf(*(m + j * mat->m_column + i));
                max_row_num = j;
            }
        }
        if (max_row_num != i) {
            swap(mat, i, max_row_num);
            m_swap++;
        }
        for (int j = i + 1; j < mat->m_row; ++j) {
            temp = -*(m + j * mat->m_column + i) / *(m + i * mat->m_column + i);
            for (int k = 0; k < mat->m_column; ++k) {
                *(m + j * mat->m_column + k) += *(m + i * mat->m_column + k) *
temp;
            }
        }
    }
    if (m_swap % 2 == 1)
        m_swap = -1;
    else m_swap = 1;
    char res[mat->m_row * 12];
    memset(res, -1, sizeof(char) * mat->m_row * 12);
    Bool directPrint = True;
    int figureSum = 0;
    Bool sign = True;
    int noNegative = 0;
    for (int i = 0; i < mat->m_column; ++i) {
        float mem = mat->m_matrix[i * mat->m_column + i];
        int thisMem = 0;
        long long calMem = llabs((long long) (mem * pow(10, 7)));
        figureSum += 7;
        if (mem == 0)
            matrix->hasInversion = False;
        int isOver = 0;
        int t_mem = (int) mem;
        while (abs(t_mem) > 0) {
            t_mem /= 10;
            isOver++;
        }
```

```
        figureSum += isOver;
        if (figureSum <= 9) //不会爆
            det *= (int) calMem;
        else {
            if (directPrint == True) {
                if (!((det < 0 && mem < 0) || (det > 0 && mem > 0)))
                    sign = False;
                sprintf((char *) res, "%lld", (long long) abs(det));
            } else {
                if ((sign == False && mem < 0) || (sign == True && mem > 0))
                    sign = True;
                else sign = False;
            }
            directPrint = False;
            char chs[isOver + 7];
            char *memChars = chs;
            sprintf((char *) memChars, "%lld", calMem);
            for (int j = noNegative; j < figureSum; ++j) {
                if ((int) res[j] > 47)
                    noNegative++;
                else break;
            }
            IntMultiply((char *) res, memChars, noNegative, isOver + 7);
        }
    }
    gettimeofday(&f_tv, NULL);
    if (directPrint == True)
        printf("%d", det * m_swap);
    else {
        for (int j = noNegative; j < figureSum; ++j) {
            if ((int) res[j] > 47)
                noNegative++;
            else break;
        }
        int i = 0;
        while (i < noNegative - 7 * mat->m_row - 1) {
            if (i == 0) {
                if ((m_swap == -1 && sign == True) || (m_swap == 1 && sign ==
False))
                    printf("%c", '-');
            }
            printf("%c", res[i]);
            i++;
        }
        if ((m_swap == -1 && sign == True) || (m_swap == 1 && sign == False)) {
            if ((int) res[i] >= 53)
                printf("%c", res[noNegative - 7 * mat->m_row - 1] - 1);
            else printf("%c", res[noNegative - 7 * mat->m_row - 1]);
        } else {
            if ((int) res[i] >= 53)
                printf("%c", res[noNegative - 7 * mat->m_row - 1] + 1);
            else printf("%c", res[noNegative - 7 * mat->m_row - 1]);
        }
    }
    printf("\n");
    deleteMatrix(mat);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
```

```
    printf("The calculation of determinant costs %.3f ms\n", period);
}
```

inverse() realizes the inverse of the matrix, in which compilation optimization and multi-parallelism are used, **scaleRow()** is called to realize the multiplication or subtraction of a row, and **swap()** is called to realize the row exchange of the matrix, and the result returns the pointer of the structure of the inverse matrix.

```
void swap(p_Matrix matrix, int i, int j) {
    float *temp = (float *) malloc(sizeof(float) * matrix->m_column);
    float *t = temp;
    float *m = matrix->m_matrix;
    for (int k = 0; k < matrix->m_column; ++k)
        *(t + k) = *(m + i * matrix->m_column + k);
    for (int k = 0; k < matrix->m_column; ++k)
        *(m + i * matrix->m_column + k) = *(m + j * matrix->m_column + k);
    for (int k = 0; k < matrix->m_column; ++k)
        *(m + j * matrix->m_column + k) = *(t++);
    free(temp);
}

void scaleRow(p_Matrix matrix, int i, float mul) {
    float *m = matrix->m_matrix;
    for (int k = 0; k < matrix->m_column; ++k) {
        *(m + i * matrix->m_column + k) *= mul;
    }
}

p_Matrix inverse(p_Matrix matrix) {
    if (matrix->hasInversion == False) {
        printf("INVERSE FAILURE!\n");
        return NULL;
    }
    p_Matrix mat = copyMatrix(matrix);
    p_Matrix inv = (Matrix *) malloc(sizeof(Matrix));
    inv->m_row = mat->m_row;
    inv->m_column = mat->m_column;
    inv->m_matrix = (float *) malloc(sizeof(float) * inv->m_column * inv-
>m_row);
    float *e = inv->m_matrix;
    float *m = mat->m_matrix;
    for (int i = 0; i < inv->m_row; ++i) {
        for (int j = 0; j < inv->m_column; ++j) {
            if (i == j)
                *(e + i * inv->m_column + j) = 1.0f;
            else *(e + i * inv->m_column + j) = 0.0f;
        }
    }
    float max = 0.0f;
    int max_row_num;
    int m_swap = 0;
    float temp;
    gettimeofday(&tv, NULL);
    for (int i = 0; i < mat->m_row - 1; i++) {
        max = fabsf(*(m + i * mat->m_column + i));
        max_row_num = i;
        for (int j = i + 1; j < mat->m_row; j++) {
```

```
            if (max < fabsf(*(m + j * mat->m_column + i))) {
                max = fabsf(*(m + j * mat->m_column + i));
                max_row_num = j;
            }
        }
        if (i != max_row_num) {
            swap(mat, i, max_row_num);
            swap(inv, i, max_row_num);
            m_swap++;
        }
        for (int j = i + 1; j < mat->m_row; j++) {
            temp = -*(m + j * mat->m_column + i) / *(m + i * mat->m_column + i);
            for (int k = 0; k < mat->m_column; k++) {
                *(m + j * mat->m_column + k) += *(m + i * mat->m_column + k) *
temp;
                *(e + j * inv->m_column + k) += *(e + i * inv->m_column + k) *
temp;
            }
        }
    }
    for (int i = 0; i < mat->m_row; i++) {
        temp = 1 / *(m + i * mat->m_column + i);
        scaleRow(mat, i, temp);
        scaleRow(inv, i, temp);
    }
#pragma omp parallel for
    for (int i = mat->m_row - 1; i > 0; i--) {
        for (int j = i - 1; j >= 0; j--) {
            temp = -*(m + j * mat->m_column + i) / *(m + i * mat->m_column + i);
            for (int k = 0; k < mat->m_column; k++) {
                *(m + j * mat->m_column + k) += temp * *(m + i * mat->m_column +
k);
                *(e + j * inv->m_column + k) += temp * *(e + i * mat->m_column +
k);
            }
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of inversion costs %.3f ms\n", period);
    deleteMatrix(mat);
    return inv;
}
```

This method uses algorithm optimization, namely the idea of divide and conquer to realize matrix multiplication, which can be used to compare with the time-consuming matrix multiplication based on hardware optimization, fully embodies the advantages of hardware optimization.

```
void strassenMul(const float *a, const float *b, float *c, int A_row, int
A_column, int B_column) {
    for (int i = 0; i < A_row; ++i) {
        for (int j = 0; j < B_column; ++j) {
            *(c + i * B_column + j) = 0.0F;
        }
    }
    float t;
```

```
#pragma omp parallel for
    for (int i = 0; i < A_row; i++) {
        for (int k = 0; k < A_column; k++) {
            t = *(a + i * A_column + k);
            for (int j = 0; j < B_column; j += 4) {
                *(c + i * B_column + j) += t * *(b + k * B_column + j);
                *(c + i * B_column + j + 1) += t * *(b + k * B_column + j + 1);
                *(c + i * B_column + j + 2) += t * *(b + k * B_column + j + 2);
                *(c + i * B_column + j + 3) += t * *(b + k * B_column + j + 3);
            }
        }
    }
}

void strassen(const float *a, const float *b, float *c, int a_row, int a_column,
int b_column, int out_decided) {
    struct timeval t_strassen, t_strassen_f;
    if ((a_column % 2 != 0 || b_column % 2 != 0 || a_row % 2 != 0) || a_row <
64) {
        gettimeofday(&t_strassen, NULL);
        strassenMul(a, b, c, a_row, a_column, b_column);
        gettimeofday(&t_strassen_f, NULL);
        if (out_decided == 32)
            printf("The calculation of multiplication using strassen costs %.3f
ms (%d)\n",
                   (t_strassen_f.tv_sec - t_strassen.tv_sec) * (double) 1000 +
                   (t_strassen_f.tv_usec - t_strassen.tv_usec) / (double) 1000,
out_decided);
        return;
    }
    int A_r = a_row;
    int A_c = a_column;
    int B_c = b_column;
    a_row /= 2;
    b_column /= 2;
    a_column /= 2;

    float *M1 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *M2 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *M3 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *M4 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *M5 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *M6 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *M7 = (float *) malloc(sizeof(float) * a_row * b_column);
    float *A11 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A12 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A22 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A21 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A11022 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A21022 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A11012 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A21111 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *A12122 = (float *) malloc(sizeof(float) * a_row * a_column);
    float *B22 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B11 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B21 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B12 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B11022 = (float *) malloc(sizeof(float) * a_column * b_column);
```

```
    float *B12122 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B21111 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B11012 = (float *) malloc(sizeof(float) * a_column * b_column);
    float *B21022 = (float *) malloc(sizeof(float) * a_column * b_column);

    // clock_t start, stop;
    // start = clock();
    gettimeofday(&t_strassen, NULL);

    for (int i = 0; i < a_row; ++i) {
        for (int j = 0; j < a_column; ++j) {
            *(A11 + i * a_column + j) = *(a + i * A_c + j);
            *(A12 + i * a_column + j) = *(a + i * A_c + j + a_column);
            *(A22 + i * a_column + j) = *(a + (i + a_row) * A_c + j + a_column);
            *(A21 + i * a_column + j) = *(a + (i + a_row) * A_c + j);
            *(A11022 + i * a_column + j) = *(A11 + i * a_column + j) + *(A22 + i
* a_column + j);
            *(A21022 + i * a_column + j) = *(A21 + i * a_column + j) + *(A22 + i
* a_column + j);
            *(A11012 + i * a_column + j) = *(A11 + i * a_column + j) + *(A12 + i
* a_column + j);
            *(A21111 + i * a_column + j) = *(A21 + i * a_column + j) - *(A11 + i
* a_column + j);
            *(A12122 + i * a_column + j) = *(A12 + i * a_column + j) - *(A22 + i
* a_column + j);
        }
    }
    for (int i = 0; i < a_column; ++i) {
        for (int j = 0; j < b_column; ++j) {
            *(B11 + i * b_column + j) = *(b + i * B_c + j);
            *(B21 + i * b_column + j) = *(b + (i + a_column) * B_c + j);
            *(B22 + i * b_column + j) = *(b + (i + a_column) * B_c + j +
b_column);
            *(B12 + i * b_column + j) = *(b + i * B_c + j + b_column);
            *(B11022 + i * b_column + j) = *(B11 + i * b_column + j) + *(B22 + i
* b_column + j);
            *(B12122 + i * b_column + j) = *(B12 + i * b_column + j) - *(B22 + i
* b_column + j);
            *(B21111 + i * b_column + j) = *(B21 + i * b_column + j) - *(B11 + i
* b_column + j);
            *(B11012 + i * b_column + j) = *(B11 + i * b_column + j) + *(B12 + i
* b_column + j);
            *(B21022 + i * b_column + j) = *(B21 + i * b_column + j) + *(B22 + i
* b_column + j);
        }
    }
    for (int i = 0; i < a_row; ++i) {
        for (int j = 0; j < b_column; ++j) {
            *(M1 + i * b_column + j) = 0.0f;
            *(M2 + i * b_column + j) = 0.0f;
            *(M3 + i * b_column + j) = 0.0f;
            *(M4 + i * b_column + j) = 0.0f;
            *(M5 + i * b_column + j) = 0.0f;
            *(M6 + i * b_column + j) = 0.0f;
            *(M7 + i * b_column + j) = 0.0f;
        }
    }
    strassen(A11022, B11022, M1, a_row, a_column, b_column, out_decided);
```

```
    strassen(A21022, B11, M2, a_row, a_column, b_column, out_decided);
    strassen(A11, B12122, M3, a_row, a_column, b_column, out_decided);
    strassen(A22, B21111, M4, a_row, a_column, b_column, out_decided);
    strassen(A11012, B22, M5, a_row, a_column, b_column, out_decided);
    strassen(A21111, B11012, M6, a_row, a_column, b_column, out_decided);
    strassen(A12122, B21022, M7, a_row, a_column, b_column, out_decided);
    for (int i = 0; i < a_row; i++) {
        for (int j = 0; j < b_column; j++) {
            *(c + i * B_c + j) =
                    *(M1 + i * b_column + j) + *(M4 + i * b_column + j) - *(M5 +
i * b_column + j) +
                    *(M7 + i * b_column + j);
            *(c + i * B_c + j + b_column) = *(M3 + i * b_column + j) + *(M5 + i
* b_column + j);
            *(c + (i + a_row) * B_c + j + b_column) =
                    *(M1 + i * b_column + j) + *(M3 + i * b_column + j) - *(M2 +
i * b_column + j) +
                    *(M6 + i * b_column + j);
            *(c + (i + a_row) * B_c + j) = *(M2 + i * b_column + j) + *(M4 + i *
b_column + j);
        }
    }
    gettimeofday(&t_strassen_f, NULL);
    if (out_decided == A_r)
        printf("The calculation of multiplication using strassen costs %.3f ms
(%d)\n",
               (t_strassen_f.tv_sec - t_strassen.tv_sec) * (double) 1000 +
               (t_strassen_f.tv_usec - t_strassen.tv_usec) / (double) 1000,
out_decided);

    free(M1);
    free(M2);
    free(M3);
    free(M4);
    free(M5);
    free(M6);
    free(M7);
    free(A11);
    free(A12);
    free(A22);
    free(A11012);
    free(A11022);
    free(A12122);
    free(A21022);
    free(A21111);
    free(B11);
    free(B12);
    free(B21);
    free(B22);
    free(B11012);
    free(B11022);
    free(B12122);
    free(B21022);
    free(B21111);
}
```

This method is used to calculate the rank of the matrix, and can get the ladder matrix of the matrix, and return the pointer to the structure of the ladder matrix.

```
void exchangeRow(float *a, int i, int j, int col, int c_mark) {
    int k;
    float t;
#pragma omp parallel for
    for (k = 0; k < col - c_mark; ++k) {
        t = *(a + i * col + c_mark + k);
        *(a + i * col + c_mark + k) = *(a + j * col + c_mark + k);
        *(a + j * col + c_mark + k) = t;
    }
}

void mulRow(float *a, int r, float k, int col, int c_mark) {
    for (int i = 0; i < col - c_mark; i++)
        *(a + r * col + c_mark + i) *= k;
}

void unitization(float *a, int i, int col, int c) {
    float fir = *(a + i * col + c);
    for (int j = c; j < col; ++j) {
        *(a + i * col + j) /= fir;
    }
}

void addRow(float *a, int r1, int r2, float k, int col, int c_mark) {
    for (int i = 0; i < col - c_mark; i++)
        *(a + col * r1 + c_mark + i) += *(a + r2 * col + c_mark + i) * k;
}

p_Matrix rank(const float *mat, int m, int n) {
    int r_mark = 0;
    int c_mark = 0;  //行标记与列标记
    int sign_c;  //某列是否全为0的标志，为1表示全为0
    p_Matrix ladder = (Matrix *) malloc(sizeof(Matrix));
    ladder->m_row = m;
    ladder->m_column = n;
    ladder->m_matrix = (float *) malloc(sizeof(float) * m * n);
    float *c = ladder->m_matrix;
    for (int i = 0; i < m * n; ++i) {
        *(c + i) = *(mat + i);
    }
    gettimeofday(&tv, NULL);
    while (c_mark < n) {
        sign_c = 1;
        for (int i = r_mark; i < m; ++i) {
            if (*(c + i * n + c_mark) != 0) {
                if (i != r_mark) {
                    if (sign_c)
                        exchangeRow(c, r_mark, i, n, c_mark);
                    else {
                        float t = *(c + i * n + c_mark);
                        mulRow(c, i, *(c + r_mark * n + c_mark), n, c_mark);
                        addRow(c, i, r_mark, -t, n, c_mark);
                    }
                } else
```

```
                    unitization(c, i, n, c_mark);
                sign_c = 0;
            }
        }
        if (!sign_c) r_mark++;
        c_mark++;
    }
    gettimeofday(&f_tv, NULL);
    printf("The calculation of rank costs %.3f ms \n",
           (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000);
    ladder->m_rank = r_mark;
    return ladder;
}
```

This method implements the output matrix to a file.

```
void filewriter(FILE *fp, p_Matrix matrix) {
    const float *m = matrix->m_matrix;
    for (int i = 0; i < matrix->m_row; ++i) {
        for (int j = 0; j < matrix->m_column; ++j) {
            fprintf(fp, "%.4f ", *(m + i * matrix->m_column + j));
        }
        fprintf(fp, "\n");
    }
    fprintf(fp, "\n");
}
```

This method implements LU decomposition of matrices, mainly for square matrices with full rank.

```
void LU(const float *A, float *Low, float *Up, int side) {
    float *r1 = (float *) malloc(sizeof(float) * side);
    float *r2 = (float *) malloc(sizeof(float) * side);
    for (int i = 0; i < side; ++i) {
        *(r1 + i) = 0.f;
        *(r2 + i) = 0.f;
    }
    float temp = 0.f;
    for (int i = 0; i < side; i++) {
        for (int j = 0; j < side; j++) {
            if (i == j)
                *(Low + i * side + i) = 1.f;
        }
    }
    gettimeofday(&tv ,NULL);
    for (int k = 0; k < side; k++) {
        for (int j = k; j < side; j++) {
            if (k == 0) {
                temp = 0.f;
            } else {
                for (int i = 0; i < k; i++) {
                    *(r1 + i) = *(Low + k * side + i);
                }
                for (int i = 0; i < k; i++) {
                    *(r2 + i) = *(Up + i * side + j);
                }
```

```
                temp = 0.f;
                for (int i = 0; i < k; i++) {
                    temp += *(r1 + i) * *(r2 + i);
                }
            }
            *(Up + k * side + j) = *(A + k * side + j) - temp;
        }

        for (int i = 0; i < side; ++i) {
            *(r1 + i) = 0.f;
            *(r2 + i) = 0.f;
        }
        temp = 0.f;
        for (int t = k + 1; t < side; t++) {
            if (k == 0) {
                temp = 0.f;
            } else {
                for (int i = 0; i < k; i++) {
                    *(r1 + i) = *(Low + t * side + i);
                }
                for (int i = 0; i < k; i++) {
                    *(r2 + i) = *(Up + i * side + k);
                }
                temp = 0.f;
                for (int i = 0; i < k; i++) {
                    temp += *(r1 + i) * *(r2 + i);
                }
            }
            *(Low + t * side + k) = (*(A + t * side + k) - temp) / *(Up + k *
side + k);
        }
    }
    gettimeofday(&f_tv , NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of LU factorization costs %.3f ms\n", period);
}
```

This method uses SIMD to optimize matrix multiplication and plays a comparative role.

```
void mulSIMD(Matrix * c, const Matrix  * a, const Matrix * b ) {
    if (a->m_column != b->m_row){
        printf("MULTIPLY FAILURE!");
        return;
    }
    gettimeofday(&tv, NULL);
    __m256 num0, s;
    int i, j, k, n;
#pragma omp parallel for
    for (i = 0; i < a->m_row; ++i) {
        for (k = 0; k < a->m_column ; ++k){
            s = _mm256_broadcast_ss(a->m_matrix + i * a->m_column + k);
            for (j = 0; j < b->m_column; j += 8) {
                n = i * c->m_column;
                num0 = _mm256_loadu_ps(c->m_matrix + n + j);
                num0 = _mm256_add_ps(num0, _mm256_mul_ps(s,_mm256_loadu_ps(b-
>m_matrix + k * b->m_column + j)));
```

```
                _mm256_storeu_ps(c->m_matrix + i * c->m_column + j, num0);
            }
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of multiplication using SIMD costs %.3f ms\n",
period);
}
```

connect.h

```
#pragma once
#pragma GCC optimize(3)

typedef enum boolean {
    True, False
} Bool;

typedef struct {
    int m_rank;
    int m_row;
    int m_column;
    float *m_matrix;
    Bool hasInversion;
    Bool hasLU;
} Matrix, *p_Matrix;

// struct timeval
// {
//     long tv_sec; /*秒*/
//     long tv_usec; /*微秒*/
// };

p_Matrix createMatrix(char *fileName);

void deleteMatrix(p_Matrix matrix);

p_Matrix copyMatrix(p_Matrix matrix);

p_Matrix multiply(const float *A, const float *B, int A_row, int A_column, int
B_row, int B_column);

p_Matrix plus(const float *A, const float *B, int A_row, int A_column, int
B_row, int B_column);

p_Matrix subtraction(const float *A, const float *B, int A_row, int A_column,
int B_row, int B_column);

p_Matrix transpose(const float *A, int A_row, int A_column);

void swap(p_Matrix matrix, int i, int j);

void det(p_Matrix matrix);

void fileWriter(FILE *fp, p_Matrix matrix);
```

```
void scaleRow(p_Matrix matrix, int i, float mul);

p_Matrix inverse(p_Matrix matrix);

void IntMultiply(char *a, const char *b, int a_len, int b_len);

void strassenMul(const float *a, const float *b, float *c, int A_row, int
A_column, int B_column);

void strassen(const float *a, const float *b, float *c, int a_row, int a_column,
int b_column,int out_decided);

void exchangeRow(float *a, int i, int j, int col, int c_mark);

void mulRow(float *a, int r, float k, int col, int c_mark);

void addRow(float *a, int r1, int r2, float k, int col, int c_mark);

p_Matrix rank(const float *mat, int m, int n);

void LU(const float *A, float *Low, float *Up, int side);

void mulSIMD(Matrix * c, const Matrix  * a, const Matrix * b);

void multiply_func(const float *A, const float *B, float *C, int A_row, int
A_column, int B_row, int B_column);
```

testOpenBLAS.c

```
#include <stdio.h>
#include <string.h>
#include <cblas.h>
#include <time.h>
#include <stdlib.h>

typedef struct {
//    int m_rank;
    int m_row;
    int m_column;
    float *m_matrix;
//    Bool hasInversion;
} Matrix, *p_Matrix;

void fileWriter(FILE *fp, p_Matrix matrix) {
    for (int i = 0; i < matrix->m_row; ++i) {
        for (int j = 0; j < matrix->m_column; ++j) {
            fprintf(fp, "%.4f ", matrix->m_matrix[i * matrix->m_column + j]);
        }
        fprintf(fp, "\n");
    }
    fprintf(fp, "\n");
}

//using namespace std;
p_Matrix createMatrix(char *fileName) {
    p_Matrix matrix = (Matrix *) malloc(sizeof(Matrix));
```

```
    matrix->m_row = 0;
    matrix->m_column = 0;
    int num = 0;
    char c;
    FILE *file = fopen(fileName, "rb");
    while (fscanf(file, "%c", &c) != EOF) {
        if (c == '\n') {
            matrix->m_row++;
            num++;
        }
        if (c == ' ')
            num++;
    }
    num++;
    matrix->m_column = num / matrix->m_row;
    int i = 0;
    matrix->m_matrix = (float *) malloc(sizeof(float) * num);
    FILE *fid = fopen(fileName, "rb");
    while (!feof(fid)) {
        fscanf(fid, "%f", matrix->m_matrix + (i++));
    }
    fclose(file);
    fclose(fid);
    return matrix;
}

void deleteMatrix(p_Matrix matrix) {
    free(matrix->m_matrix);
    free(matrix);
}

int main(int argc, char **argv) {
    struct timeval start, stop, up, down;
    gettimeofday(&start,NULL);
    char *fileA = NULL;
    char *fileB = NULL;
    char *fileC = NULL;
    if (argc > 1) {
        fileA = argv[1];
        fileB = argv[2];
        fileC = argv[3];
    }
    FILE *fp = fopen(fileC, "w");
    p_Matrix A = createMatrix(fileA);
    p_Matrix B = createMatrix(fileB);

    p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
    C->m_row = A->m_row;
    C->m_column = B->m_column;
    C->m_matrix = (float *)malloc(sizeof(float) * C->m_column * C->m_row);

    const float *a;
    const float *b;
    float *c;
    a = A->m_matrix;
    b = B->m_matrix;
    c = C->m_matrix;
    const int M = A->m_row;
```

```
    const int N = A->m_column;
    const int K = B->m_column;
    const float alpha = 1;
    const float beta_positive = 1;
    const float beta_negative = -1.0f;
    const float beta = 0;
    const int lda = M;
    const int ldb = K;
    const int ldc = N;

    float * e = (float *)malloc(sizeof(float) * A->m_row * A->m_column);
    for(int i = 0; i < A->m_row; ++i){
        *(e + i * A->m_column + i) = 1.0f;
    }
    const float * E = e;
    double period;

    // gettimeofday(&up, NULL);
    // cblas_cgemm3m(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, &alpha,
(float *) a, lda,(float *) b, ldb, &beta,(float *) c, ldc);
    // gettimeofday(&down,NULL);
    // double period = (down.tv_sec - up.tv_sec)*(double)1000 + (down.tv_usec -
up.tv_usec)/(double)1000;
    // printf("OpenBLAS calculates the multiplication (less matrix mul) costs
%.3f ms\n",period);
    // fileWriter(fp, C);

    gettimeofday(&up, NULL);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha,(float
*) a, lda,(float *) b, ldb, beta,(float *) c, ldc);
    gettimeofday(&down,NULL);
    period = (down.tv_sec - up.tv_sec)*(double)1000 + (down.tv_usec -
up.tv_usec)/(double)1000;
    printf("OpenBLAS calculates the multiplication costs %.3f ms\n",period);
    fileWriter(fp, C);

    gettimeofday(&up, NULL);
    cblas_sgemm(CblasRowMajor, CblasTrans, CblasNoTrans, M, N, K, alpha,(float
*) a, lda,(float *) E, ldb, beta,(float *) c, ldc);
    gettimeofday(&down,NULL);
    period = (down.tv_sec - up.tv_sec)*(double)1000 + (down.tv_usec -
up.tv_usec)/(double)1000;
    printf("OpenBLAS calculates the transposition costs %.3f ms\n",period);
    fileWriter(fp, C);

    for(int i = 0; i < B->m_row; ++i){
        for(int j = 0; j < B->m_column; ++j)
        *(c + i * B->m_column + j) = *(b + i * B->m_column + j);
    }
    // const float * E = e;

    gettimeofday(&up, NULL);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha,(float
*) a, lda,(float *) E, ldb, beta_positive,(float *) c, ldc);
    gettimeofday(&down,NULL);
    period = (down.tv_sec - up.tv_sec)*(double)1000 + (down.tv_usec -
up.tv_usec)/(double)1000;
    printf("OpenBLAS calculates the addition costs %.3f ms\n",period);
```

```
    fileWriter(fp, C);

    for(int i = 0; i < B->m_row; ++i){
        for(int j = 0; j < B->m_column; ++j)
        *(c + i * B->m_column + j) = *(b + i * B->m_column + j);
    }

    // const float alph = -1;
    gettimeofday(&up, NULL);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha,(float
*) a, lda,(float *) E, ldb, beta_negative,(float *) c, ldc);
    gettimeofday(&down,NULL);
    period = (down.tv_sec - up.tv_sec)*(double)1000 + (down.tv_usec -
up.tv_usec)/(double)1000;
    printf("OpenBLAS calculates the subtraction costs %.3f ms\n",period);
    fileWriter(fp, C);

    deleteMatrix(C);
    deleteMatrix(A);
    deleteMatrix(B);
    fclose(fp);

    gettimeofday(&stop, NULL);
    period = (stop.tv_sec - start.tv_sec)*(double)1000 + (stop.tv_usec -
start.tv_usec)/(double)1000;
    printf("The running time is %.3f ms\n",period);
    return 0 ;
}
```

Result Showing

The Realization of Mine

32 by 32

```
linjiefang@LAPTOP-99UCO9DE:~/project03$ ./pro01 mat-A-32.txt mat-B-32.txt result.txt
A's transposition cost:
The calculation of transposition costs 2.402 ms
B's transposition cost:                                                    transport
The calculation of transposition costs 0.005 ms
The calculation of addition costs 0.004 ms                                 plus
The calculation of subtraction costs 0.003 ms                              subtract
The calculation of rank costs 0.018 ms
The A's rank is: 32
The calculation of rank costs 0.015 ms                                     rank
The B's rank is: 32
A's determinant is:
357806434713518189423994696240809152624764953947582566479232078 7
The calculation of determinant costs 0.195 ms
B's determinant is:                                                        determinant
-446199865053443499800564550655623775396019132185095398947681554 3
The calculation of determinant costs 0.140 ms
A's LU factorization cost:
The calculation of LU factorization costs 0.030 ms
B's LU factorization cost:                                                 LU
The calculation of LU factorization costs 0.028 ms
A's inversion cost:
The calculation of inversion costs 0.035 ms
B's inversion cost:                                                        inversion
The calculation of inversion costs 0.034 ms
The calculation of multiplication costs 0.005 ms
The calculation of multiplication using strassen costs 0.008 ms (32)       multiply
The calculation of multiplication using SIMD costs 0.031 ms
The running time is 24.553 ms                                              Running time
```

The ladder matrix:

A:

```
1.0000 2.3632 4.1651 2.2689 1.0660 2.6981 1.2264 1.4387 0.7972 2.5519 2.2123 1.0047 1.6368 2.3302 1.2406 1.64
0.0000 1.0000 1.6470 0.5492 0.0504 0.6944 0.0624 0.4272 0.2082 0.9620 0.7824 -0.0051 0.4697 0.6904 0.4070 0.5
0.0000 0.0000 1.0000 -0.3133 -0.2232 0.3054 -0.4120 -0.1482 -0.6584 0.3179 0.2995 -0.0409 0.0957 0.4219 -0.35
0.0000 0.0000 0.0000 1.0000 -0.1021 -1.0985 -1.4783 -0.3162 -0.5157 0.5997 -1.8686 -0.5183 0.0347 -2.4248 -1.
0.0000 0.0000 0.0000 0.0000 1.0000 3.7651 5.0171 1.3749 1.9252 -1.2170 5.2156 1.7159 0.9090 5.1393 3.5690 3.2
0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.9705 0.4487 -0.0377 -0.6996 1.4342 0.6773 0.7893 0.6952 1.3818 1.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.1423 0.4242 -0.2564 0.0199 -0.2379 -0.6656 1.3225 -0.4173
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -1.0300 -1.3906 1.2220 0.1892 0.9958 0.1361 1.5422 0.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.8903 -0.9329 -0.0769 0.7651 -0.3771 -0.7473
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -3.0774 4.6678 4.6846 -5.5889 -1.7624 4
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -2.4970 -1.3968 1.9958 0.4263 -1
```

B:

```
1.0000 0.3918 1.2765 1.2765 1.1406 1.1706 1.4092 1.0190 1.0174 1.1485 0.2243 0.0964 1.3934 0.9795 0.3539 0.33
0.0000 1.0000 6.1983 11.3490 5.2434 10.7023 4.2118 7.1548 7.4422 6.1551 -0.7037 -6.2272 4.8332 -2.0463 -5.302
0.0000 0.0000 1.0000 1.5624 0.8586 1.4401 0.7299 0.9746 1.1595 0.9059 -0.1843 -1.0509 0.7311 -0.3299 -0.8347
0.0000 0.0000 0.0000 1.0000 0.1632 0.2558 -0.0046 0.5582 0.4985 0.2942 -0.0854 -0.3027 0.1423 -0.2647 -0.3926
0.0000 0.0000 0.0000 0.0000 1.0000 -0.8602 -2.2693 -0.6040 2.0046 0.5846 -0.0978 -1.5352 -0.5764 -2.2260 -0.9
0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 6.6580 0.9329 -4.8300 -0.7553 -1.1463 6.3679 3.0368 5.5811 2.0364 5
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.1530 -0.6756 -0.1030 -0.2286 0.9735 0.4399 0.8437 0.2522 1
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.8208 1.0223 0.7759 0.0128 -0.4503 -0.4089 1.4275 -0
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.3136 0.5553 -1.9061 0.0589 -0.2063 -1.2114 -
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.8948 1.5620 -0.6373 -1.2618 0.9012 0
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -1.1546 -0.0739 0.4816 -1.2732 -
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.3670 -0.9677 2.5113 -0
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.3480 2.1046 -1.1
```

The result of A’ s LU:

Lower triangular matrix

```
1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0
3.7358 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0
0.1179 -0.3084 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.
0.5236 -0.1085 0.6748 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.
4.2170 0.7773 0.4431 3.1545 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0
2.8868 0.3403 0.7376 2.6061 1.3410 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0
1.8821 0.4220 0.3120 1.0331 -0.8499 -1.5948 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
1.2028 0.1101 0.7089 -0.2268 -0.8538 -1.4846 0.2343 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
2.7264 0.7081 -0.3801 1.1538 0.1985 -0.3486 0.1215 0.7115 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
1.5377 -0.0571 0.7437 1.4652 -0.0754 -1.2138 -0.2202 0.7987 -0.0408 1.0000 0.0000 0.0000 0.0000 0.0000 0.000
```

Upper triangular matrix

```
21.2000 50.1000 88.3000 48.1000 22.6000 57.2000 26.0000 30.5000 16.9000 54.1000 46.9000 21.3000 34.7000 49.40
0.0000 -183.1660 -301.6754 -100.5943 -9.2302 -127.1906 -11.4321 -78.2434 -38.1358 -176.2094 -143.3113 0.9264
0.0000 0.0000 -92.2553 28.9026 20.5881 -28.1733 38.0081 13.6715 60.7452 -29.3261 -27.6307 3.7739 -8.8265 -38.
0.0000 0.0000 0.0000 -30.1002 3.0724 33.0663 44.4981 9.5176 15.5225 -18.0498 56.2444 15.6014 -1.0443 72.9862
0.0000 0.0000 0.0000 0.0000 -43.8432 -165.0760 -219.9665 -60.2821 -84.4049 53.3570 -228.6676 -75.2292 -39.852
0.0000 0.0000 0.0000 0.0000 0.0000 92.3312 89.6084 41.4282 -3.4821 -64.5957 132.4232 62.5365 72.8779 64.1860
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 -132.4810 18.8515 -56.1938 33.9652 -2.6344 31.5217 88.1854 -175.209
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 61.1084 -62.9392 -84.9796 74.6774 11.5616 60.8517 8.3147 94.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 53.5475 47.6752 -49.9532 -4.1175 40.9706 -20.1908 -40
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 15.3186 -47.1418 71.5040 71.7615 -85.6137 -26.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 112.4269 -280.7242 -157.0415 224.3862 4
```

The result of B’ s LU:

Lower triangular matrix

```
1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.
1.0758 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.
0.7694 -9.5899 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
0.9668 -4.0001 0.3572 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
1.4408 -7.8386 0.7840 1.0739 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
0.5276 -7.7349 0.6691 1.1848 1.9054 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
0.4897 -11.7099 1.0187 1.5646 -0.4212 -5.8518 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.000
0.4139 -4.4138 0.3918 0.4112 1.4630 2.2545 -0.2547 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.3886 -11.9719 1.1213 1.0450 1.2754 -0.1994 0.1625 0.3726 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.2622 -3.5707 0.3355 0.2134 0.5543 -1.3095 0.3092 1.4455 -0.4970 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.2338 -9.6286 0.8630 0.8594 1.9330 0.2230 0.1379 0.9698 0.5276 0.9604 1.0000 0.0000 0.0000 0.0000 0.0000 0
```

Upper triangular matrix

```
63.3000 24.8000 80.8000 80.8000 72.2000 74.1000 89.2000 64.5000 64.4000 72.7000 14.2000 6.1000 88.2000 62.000
0.0000 -7.2806 -45.1270 -82.6270 -38.1749 -77.9190 -30.6640 -52.0910 -54.1834 -44.8128 5.1232 45.3374 -35.188
0.0000 0.0000 -490.6281 -766.5500 -421.2413 -706.5455 -358.0913 -478.1717 -568.8609 -444.4829 90.4065 515.589
0.0000 0.0000 0.0000 -130.8555 -21.3582 -33.4761 0.5955 -73.0460 -65.2290 -38.4928 11.1752 39.6096 -18.6142 3
0.0000 0.0000 0.0000 0.0000 37.6241 -32.3625 -85.3792 -22.7235 75.4230 21.9942 -3.6784 -57.7591 -21.6879 -83.
0.0000 0.0000 0.0000 0.0000 0.0000 24.5918 163.7310 22.9411 -118.7789 -18.5742 -28.1885 156.5986 74.6799 137.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 935.7646 143.2115 -632.1700 -96.3643 -213.9473 910.9644 411.6756 78
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 43.7669 35.9249 44.7448 33.9596 0.5598 -19.7099 -17.8955 62.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 51.4185 16.1235 28.5506 -98.0073 3.0263 -10.6053 -62.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 -60.5031 54.1396 -94.5062 38.5597 76.3418 -54.
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 -65.1109 75.1761 4.8090 -31.3567 82.899
```

32 by 32:

Transposition

```
21.2000 79.2000 2.5000 11.1000 89.4000 61.2000 39.9000 
50.1000 4.0000 62.4000 46.1000 68.9000 82.3000 17.0000 
88.3000 28.2000 11.2000 16.7000 97.0000 84.2000 10.1000
48.1000 79.1000 65.6000 25.5000 42.5000 47.5000 26.0000
22.6000 75.2000 26.1000 29.8000 63.1000 26.5000 85.5000
```

```
63.3000 68.1000 48.7000 61.2000 91.2000 33.4000 31.000
24.8000 19.4000 88.9000 53.1000 92.8000 69.4000 97.400
80.8000 41.8000 4.3000 83.4000 85.5000 63.4000 68.2000
80.8000 4.3000 88.0000 4.0000 22.6000 13.8000 21.5000
72.2000 39.5000 0.4000 50.7000 87.7000 97.9000 4.0000
```

Subtraction

```
-42.1000 25.3000 7.5000 -32.7000 -49.6000 -16.9000 -63.2000 -34.0000 -47.5000 -18.6000 32.7000 15.2000 -53.5000
11.1000 -15.4000 -13.6000 74.8000 35.7000 84.7000 20.4000 18.4000 9.9000 -7.5000 11.5000 28.6000 -16.1000 -23.5
-46.2000 -26.5000 6.9000 -22.4000 25.7000 -79.9000 40.0000 -29.6000 74.2000 -9.8000 -30.1000 -79.5000 -24.8000
-50.1000 -7.0000 -66.7000 21.5000 -20.9000 -39.7000 3.4000 16.3000 58.9000 -42.7000 41.0000 -19.1000 -58.8000 -
-1.8000 -23.9000 11.5000 19.9000 -24.6000 -26.2000 34.6000 18.4000 -43.8000 -13.6000 -36.6000 23.8000 -47.6000
```

Addition

```
84.5000 74.9000 169.1000 128.9000 94.8000 131.3000 115.2000 95.0000 81.3000 126.8000 61.1000 27.4000 122.9000 11
147.3000 23.4000 70.0000 83.4000 114.7000 88.3000 151.0000 53.0000 40.1000 59.3000 52.3000 132.4000 103.3000 139
51.2000 151.3000 15.5000 153.6000 26.5000 115.5000 49.2000 112.4000 74.8000 72.6000 74.3000 91.5000 68.4000 72.6
72.3000 99.2000 100.1000 29.5000 80.5000 155.3000 166.6000 70.1000 80.1000 61.9000 114.4000 77.5000 99.8000 179.
180.6000 161.7000 182.5000 65.1000 150.8000 164.4000 41.4000 68.8000 110.0000 162.8000 82.4000 108.6000 112.4000
```

Inversion

A

```
-0.0019 0.0189 0.0082 0.0101 -0.0087 0.0081 -0.0012 0.0160
0.0018 0.1016 0.0238 0.0422 -0.0541 0.0468 0.0055 0.0537 -
-0.0081 -0.1066 -0.0260 -0.0607 0.0580 -0.0528 -0.0146 -0.
0.0047 0.0315 0.0080 0.0144 -0.0178 0.0172 0.0006 0.0101 -
0.0064 0.0285 0.0052 0.0151 -0.0154 0.0148 0.0084 0.0194 -
0.0079 -0.1504 -0.0420 -0.0595 0.0709 -0.0584 0.0002 -0.08
```

B

```
-0.0019 0.0189 0.0082 0.0101 -0.0087 0.0081 -0.0012 0.0160
0.0018 0.1016 0.0238 0.0422 -0.0541 0.0468 0.0055 0.0537 -
-0.0081 -0.1066 -0.0260 -0.0607 0.0580 -0.0528 -0.0146 -0.
0.0047 0.0315 0.0080 0.0144 -0.0178 0.0172 0.0006 0.0101 -
0.0064 0.0285 0.0052 0.0151 -0.0154 0.0148 0.0084 0.0194 -
0.0079 -0.1504 -0.0420 -0.0595 0.0709 -0.0584 0.0002 -0.08
```

multiplication

```
60279.1562 67331.4141 56633.3320 60811.2656 73374.4922 66704.7109 71979.1328 71427.7031 57368.1953 77109.0078
62787.8008 77804.6641 76427.6484 73717.7891 90963.1719 74125.7891 96060.6562 73474.1562 66720.4141 92853.0703
58343.7734 67662.4922 57325.6602 57463.6328 78322.0312 57996.0547 76328.6406 62307.8125 63345.8867 79102.3984
65116.0625 73441.5391 70333.0859 61170.4180 81065.4688 72297.5469 77137.3281 67358.1641 69491.1094 91903.0703
73297.4922 88327.7422 76795.1172 81119.1094 95371.8516 85730.0547 96954.8281 83247.3281 76364.2109 96245.8672
```

256 by 256

```
linjiefang@LAPTOP-99UCO9DE:~/project03$ ./pro01 mat-A-256.txt mat-B-256.txt result.txt
A's transposition cost:
The calculation of transposition costs 1.072 ms
B's transposition cost:
The calculation of transposition costs 0.444 ms
```

transport

```
The calculation of addition costs 0.453 ms
```

plus

```
The calculation of subtraction costs 0.875 ms
```

subtract

```
The calculation of rank costs 2.721 ms
The A's rank is: 256
The calculation of rank costs 2.546 ms
The B's rank is: 256
```

rank

```
A's determinant is:
14039818958505953922937622524449399997802704423836940666928337496559090862389698117236505820773779096865558668415931727625963191709347459912783228416581835356660755976920305290936088897502253562662318166261071633488367980015110703971463907796772187055508434509300670830383642210122453153747288080127695313177506365651036230726554000982477805905886982639312291460113293657582408893830281973480059241177269430462476909483389090987212762163871028731244466582391828809800178701679213198574498307952043941761689460052248887065274637930201150124657582089563541432686668362298560391598586096370589138070415195018747019749794485577668700
The calculation of determinant costs 4.998 ms
B's determinant is:
-898791130063554471736108072546088728452368805325259880008605273707285771033011082269519962655813102613473717875534467662271803177801757385301080271115582578284077927231437925277871307481893975028008824308247502747138250888393419380717329004908887594167082891350408837517981684117162196713185122135035727105478853263824123057338619711085085800817843221318212905515669399073439014966670711074510932486971536332209395328358674638540952593265943239914890246548262209403477135577458870102511203403574976717017804817900505486786833767455812522022236757284309807336411279468988192806143813032787090095979960299924835407582581144688565 6
The calculation of determinant costs 11.520 ms
```

determinant

```
A's LU factorization cost:
The calculation of LU factorization costs 17.111 ms
B's LU factorization cost:
The calculation of LU factorization costs 11.709 ms
```

LU

```
A's inversion cost:
The calculation of inversion costs 5.528 ms
B's inversion cost:
The calculation of inversion costs 12.671 ms
```

inversion

```
The calculation of multiplication costs 0.773 ms
The calculation of multiplication using strassen costs 4.005 ms (256)
The calculation of multiplication using SIMD costs 2.108 ms
```

multiply

```
The running time is 552.960 ms
```

Running time

256 by 256:

Transposition

A

```
1    93.1000 26.5000 51.1000 98.7000 87.6000 1.7000 7
2    44.0000 37.9000 82.4000 44.8000 72.7000 49.2000
3    71.3000 70.8000 7.2000 6.1000 39.1000 6.8000 75.
4    49.1000 32.2000 15.8000 48.0000 91.2000 0.0000 3
5    21.3000 78.0000 42.2000 93.6000 8.2000 39.8000 3
6    60.8000 96.8000 85.8000 26.3000 70.2000 56.2000
7    50.8000 59.3000 67.0000 73.4000 79.3000 59.7000
```

B

```
258  35.2000 58.0000 9.8000 79.7000 92.9000 8.3000 93
259  18.7000 63.1000 58.0000 25.1000 2.1000 73.6000 4
260  63.8000 65.4000 91.1000 54.5000 45.7000 30.4000
261  81.8000 42.3000 84.7000 0.9000 2.4000 95.3000 14
262  70.9000 74.3000 91.0000 79.7000 58.5000 51.1000
263  42.4000 99.9000 16.3000 77.6000 44.2000 84.6000
264  14.0000 97.3000 20.4000 63.1000 57.8000 14.1000
```

Addition

```
772  128.3000 62.7000 135.1000 130.9000 92.2000
773  84.5000 101.0000 136.2000 74.5000 152.3000
774  60.9000 140.4000 98.3000 100.5000 133.2000
775  178.4000 69.9000 60.6000 48.9000 173.3000
776  180.5000 74.8000 84.8000 93.6000 66.7000 1
777  10.0000 122.8000 37.2000 95.3000 90.9000 1
```

Subtraction

```
515  57.9000 25.3000 7.5000 -32.7000 -49.6000 18.4000 3
516  -31.5000 -25.2000 5.4000 -10.1000 3.7000 -3.1000 -
517  41.3000 24.4000 -83.9000 -68.9000 -48.8000 69.5000
518  19.0000 19.7000 -48.4000 47.1000 13.9000 -51.3000
519  -5.3000 70.6000 -6.6000 88.8000 -50.3000 26.0000 2
```

Inversion

A

```
1029  0.0062 -0.0002 -0.0063 0.0023 -0.0101 0.0038 -0.0011 -
1030  0.0013 0.0004 0.0000 0.0017 -0.0046 0.0010 -0.0000 -0.
1031  -0.0012 -0.0013 0.0018 0.0002 0.0013 -0.0002 -0.0001 -
1032  0.0009 -0.0021 -0.0004 0.0007 -0.0050 0.0015 -0.0006 -
1033  0.0037 -0.0012 -0.0065 -0.0005 -0.0013 0.0012 -0.0003
1034  -0.0016 -0.0009 0.0021 0.0002 0.0000 0.0002 0.0010 -0.
```

B

```
1286  0.0062 -0.0002 -0.0063 0.0023 -0.0101 0.0038 -0.0011 -
1287  0.0013 0.0004 0.0000 0.0017 -0.0046 0.0010 -0.0000 -0.
1288  -0.0012 -0.0013 0.0018 0.0002 0.0013 -0.0002 -0.0001 -
1289  0.0009 -0.0021 -0.0004 0.0007 -0.0050 0.0015 -0.0006 -
1290  0.0037 -0.0012 -0.0065 -0.0005 -0.0013 0.0012 -0.0003
1291  -0.0016 -0.0009 0.0021 0.0002 0.0000 0.0002 0.0010 -0.
```

Multiplication

```
1543  609701.3125 668480.6250 621181.8125 619568.8750 649068.9375 626989.0000 628409.8750 606410.8125 589301.5625
1544  622537.8750 694945.5625 630179.5625 614834.8750 663380.1875 640656.1875 621345.5625 633641.8750 642800.1875
1545  621054.7500 693947.3750 642419.2500 652366.4375 664910.3125 644445.5625 630180.4375 661696.6250 628899.1250
1546  671541.8125 685620.4375 621037.6875 663779.6875 706507.4375 666723.3125 630142.3125 660631.9375 639101.6875
1547  580221.9375 626748.1875 566113.5625 598267.3750 625390.5625 614839.0000 588099.1250 623297.6250 601829.8125
```

Ladder : A

```
1    1.0000 0.4726 0.7658 0.5274 0.2288 0.6531 0.5456 0
2    0.0000 1.0000 1.9903 0.7182 2.8349 3.1327 1.7670 0
3    0.0000 0.0000 1.0000 0.3583 0.9104 0.8795 0.4316 -
4    0.0000 0.0000 0.0000 1.0000 6.5296 1.2232 2.4555 -
5    0.0000 0.0000 0.0000 0.0000 1.0000 0.1932 0.3175 -
6    0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -1.3315
```

B

```
258  1.0000 0.5312 1.8125 2.3239 2.0142 1.2045
259  0.0000 1.0000 -1.2304 -2.8644 -1.3170 0.93
260  0.0000 0.0000 1.0000 1.5413 1.0181 -0.3226
261  0.0000 0.0000 0.0000 1.0000 -0.1546 0.6131
262  0.0000 0.0000 0.0000 0.0000 1.0000 2.2190
263  0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
264  0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
```

The result of A' s LU:

Lower triangular matrix

```
1029  1.0000 0.4726 0.7658 0.5274 0.2288 0.6531 0.5456 0
1030  0.0000 1.0000 1.9903 0.7182 2.8349 3.1327 1.7670 0
1031  0.0000 0.0000 1.0000 0.3583 0.9104 0.8795 0.4316 -
1032  0.0000 0.0000 0.0000 1.0000 6.5296 1.2232 2.4555 -
1033  0.0000 0.0000 0.0000 0.0000 1.0000 0.1932 0.3175 -
1034  0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -1.3315
1035  0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -
```

Upper triangular matrix

```
1286  1.0000 0.5312 1.8125 2.3239 2.0142 1.2045 0.3977
1287  0.0000 1.0000 -1.2304 -2.8644 -1.3170 0.9303 2.29
1288  0.0000 0.0000 1.0000 1.5413 1.0181 -0.3226 -0.758
1289  0.0000 0.0000 0.0000 1.0000 -0.1546 0.6131 0.2128
1290  0.0000 0.0000 0.0000 0.0000 1.0000 2.2190 -0.4139
1291  0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.6526
1292  0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
```

The result of B' s LU:

Lower triangular matrix

```
1543  1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
1544  0.2846 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0
1545  0.5489 2.2955 1.0000 0.0000 0.0000 0.0000 0.0000 0
1546  1.0602 -0.0728 0.4451 1.0000 0.0000 0.0000 0.0000
1547  0.9409 1.2334 0.6106 2.6312 1.0000 0.0000 0.0000 0
1548  0.0183 1.9072 0.6142 -0.1491 -0.0138 1.0000 0.0000
1549  0.7669 2.0869 0.5713 -0.5235 -0.0325 5.3336 1.0000
```

Upper triangular matrix

```
1800  93.1000 44.0000 71.3000 49.1000 21.3000 60.8000 50
1801  0.0000 25.3758 50.5052 18.2242 71.9372 79.4939 44.
1802  0.0000 0.0000 -147.8680 -52.9828 -134.6210 -130.04
1803  0.0000 0.0000 0.0000 20.8545 136.1712 25.5096 51.2
1804  0.0000 0.0000 0.0000 0.0000 -376.6656 -72.7763 -11
1805  0.0000 0.0000 0.0000 0.0000 0.0000 -13.8431 18.432
1806  0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 -81.8646
```

2048 by 2048

```
linjiefang@LAPTOP-99UCO9DE:~/project03$ ./pro01 mat-A-2048.txt mat-B-2048.txt result.txt
A's transposition cost:
The calculation of transposition costs 41.481 ms
B's transposition cost:
The calculation of transposition costs 57.720 ms
The calculation of subtraction costs 15.853 ms
The calculation of addition costs 0.969 ms
The calculation of rank costs 2728.569 ms
The A's rank is: 2048
The calculation of rank costs 2743.048 ms
The B's rank is: 2048
```

transport

plus

subtract

rank

A' s determinant

```
A's determinant is:
1246375738153290277522920975512414634318785582604023004167056541153370601524177721092561938085243167165606322057484083599011245213452659742996102149552
1341789605108990473798091709824420511856599997272244980494998035468410570792958225664722583551601330687872903078578387835704841247440812007230939298862
6304074492814250072243899361959242980626933499816956762468687109116744133139129687287175653589833553073615382265304923710951313012061669866007344941187
1232586949438214149846022189651661007795976121683242881924311542458264448170025253050545887494211472371044569635195035603956765375327179799663759115425
0707875161305110885476414219474271461872755538893863379707626075805875442230888230038470239417620790085462941192574680319485497947069114355232235439930
3413759456590296519291063848137436370272305525184324728845128395259509738146729014360253976009207585524782999222205775316044360624236984588323517596308
0950620255132597937582836688729466064119824318801700255820473635816362697956573097015683879122832125000812456125163321236741195490450154068010055320144
2177327160229386696094630908266479545142717573194984726011635670832099422063689553900441116038273783387801234885798116792079727570251199492996678282398
4741031470960798379930747201057642372263844418365521765166933536800151102115745206824911657384666597626972928084316869150298194239336176740952015676770
1087637022222762270043678049211619504056743908702233770966501230382901317160114365261258503846706054798655927392785764606354992941233371134174849120935
6919459742450088896022693214860685982787868699860017415695680933318865435814524794209494917801906756859449150587665386876699574295246791082758723233587
7328771741826286363053971137370189527011042200148985011984616959263016112826388896124506244656178420090530240994733723185398771778096598770068383256132
0879542711835636251561950592863033206060148233645201906807502615345489116393957096694484301118910390577353043657635885774722466312358473285032451169303
1601037543320143410960324011577918904729647845402830183933410406086903356970006774612818712954768652357039723500297374460184015842133413690507602690373
4755402325838800771929723375558442688480454626348129141874135917628053232225219629734817326857456849378756451169580423434673684697321177905830880041646
4648395632371514329877234001083029563816800394716474721119855717687463834422198824771726247502769555984827593575497416643060510199451319172088293575573
2034081144302386179278852534473400251578086811815635938010976471463535365685155478215369947821525690955027771961811087695636795014452383704005955137854
7639699090893934023239280966692885017502220016001135587638465413093588791754540824243952226066296226660914099312720292288619200913963219814207145655292
7646298080629282692903199404124777585008952890598407664641926416713727064962722619516054120370534923569391732712209206610124826884601202070502189370405
0964367509273360138772288418110378354929213134075208718103922246830826247595280658251509447489823822095864584508162328151926661920810754532793296032391
8161634172943746358750637480737082451721538348365694425758766261975202797694916740252668503928000719588944628364293027716569753232831499472621775710518
1189900407128525059778986660595476917411943663783839481210324539791805271163330717863729257838095960503301945246491053598295269849395375858516758146326
2592444796851160769833910915608282114968547074819437903456193130866288976430500245496173556339020603781545798082180955763520228012651208972288566617194
1020920146140166101542571900079709819859530046839540414141731349515507244734080212072576672538146135135264191868478797282453631370091930753671690886725
0199893196182800617308000405377993849022957150527095755226443126990687091384180859832275443109324053128205716393707991904755319255941832221150979528349
6414489939517321972130227892364923570781977891564561126606206333302776792799661186092637269773902222291657318295971828677434354074649596533067367779984
5017876773892654726354592483984462007077408873443039157259606365611579358870446992774333995684077802254333454074950198125434878834620399107544477600546
8767694811809025952467244953262735603932931148729399367726959250413962150434244474657719968967517143191097660752852618793211726308485859992348453483457
6261025374994891680934144493966869863442025753889347750150479347734026005700185864610450263478598580152448501960098720763329218584529812975598130275027
8310790850674195247307133583252267730105787682155395140133167435264528774968627550187625276298747544048641329940678085313734407951258317371156286668737
9328763246841760199612212941336931704508699397377904228276486195318481267550433634135120943907679995917407413859128220798579778165561677418522234862371
8180165758952492447704198469244280022473913639213319780066752055478995836959600275417264034285820571400633837312006258140081741605755454532010950873674
5746485379452094756682872286437065016861588122491952884544317514547920316925795211510997079942940596038795844385733420515942259633777759756653013131036
4812936568624966513576134134828145955036049426472347352613944343923086199075019466320020910974911516501648170797926887248853490694983337413102541041030
2021342054057202751374090824150731000146141489715862953944820006353358912654087351204920939921145388705175896736153703190310657611357770175606394685551
1956298671862710585175847425076651194038139215479103094259546232282626526473032278438186305312027419596977940320212598756703446853494384666632924417102
9636031996566341406951193629245162507788632973960287707281640617604708261156096973718018838189513766599876486231017507100540031428726765674115577908199
1486783627493205443959104739895366704968020833248156475472413273207529245419014132071859428235011351548620262208317559262028530210088978309855674618491
3159158733742009466597867184993653225190215412782212749282798437172283236389437355071515538200611309414387497342773736155567604360961249808554523394492
525219161302547570054253503494771166297059872572339
The calculation of determinant costs 3149.055 ms
```

B's determinant is:

B’ s determinant

```
-441271205780663327059894213783772837558894021469472988182660898887159279347120514163488852228875716998261578231991848039208661266582466783869558674540
097668560334008656243856931614155393065155950952793173890586584796287149111724302726505614709133100314860132190953574076290531646910559762612133561017 9
928594829956659821241709069793000310198539233146617049832567370748867737058825663005053421873460662065840919769901910397324208383028116474977052997159 0
464317055622019254095150631522635149148449414136090498740856031028651662514493007118931890142623897488097292376783875130109575415408646982522627320 2
088339980551637696075530613869445340733788296668314206800396399855468804266668475145585856785801549093551835977520215092549933127358957307629781162136 4
761230206271024567129263353364908798586054073982001030271992821086447890382621719339482527487643271674486369031571185987982322557702296474692250629503 0
507682392222312556772110231361669693013097677944903231183775194780686432010268001179321431540082716739298532492102635057231622693833203402297291686817 3
804052034255012022044709566976343251820394900840067759039468264625058096588337074554557906761928048486805189560295564788948873215877537942570630620928 4
485391998177492094673475589814806572609570317002814195672765834560818466317095292626765414214300323555753627067465697244079798188531235550202940025434 3
921649932523913754618552102975349542777131645375825419747229779773978968614013080591242534480990250728265228283700331753358657103338738262252518515473 94
409191248009249022899789857251933742466464350206754877393454714161129828592868076795567286959193083245418143965826625475493686939171801842517551369869 46
031184632664958107948188988662563194414265524458735021567849510155645246403626897144700099766734137800191301308560935076116441891214955222991575165719 9
259005762106129984125167723540882362507808228020111457097133715806885488387599085666324455079172502524239634357423449291130597800713615481362489172590 0
700901264198006006749810133120108689298394277061981903979301181007185828715635401910324726936203160095334135904912883382436815072131478469094785555036 6
023973923126741389328739218928221968115982369261992680175933642149615385218962477561363911638968680826551803242182476366961740098644538324848347421675 6
901639786449722752942999865180843618944975842555678156896091973998025364485678389231488330204089925019848221322765672446318597283615363841294822287319 9
303905517824150226698284110769017640347017473890882006164949887458239021081198745686348739567365845043515790211151186084109730951085136885203381525591 8
678206830972447717310688132048608518001328329178968848030076229954142470420205715770368743173393361278885911085437166605786854739956261072975887446248 9
989044051653588231295863311840721636374438035381850295484616911817345481692666411369774989554849243516381457959123669199262475855879916571153655770414 2
253848040417643019919146478241365016674156305317775720652397353215486966142923347587686000423786324324444715830484710995674097595844195635344109255104 64
641259946326971044789381254777646424423732418659063256063727476325153188519720633057020551512209762065410694763226028110211507852811335160164935165444 2
364996519359910589848351816397118574733745778546508693670934124903162946486012991876607505765294491717600973108618185373836593350426935961863067488604 7
714026018374399358635421054619094213161340182365231027054889235110335301890839682901792228746607769250304284648625790295169780210016915093863961543023 8
007373808270180425577891545155126520207499366697496737719469414513128998589385674857782885167979729208561759185480817858980804639899478155906025027076 5
936348201581390292163070955879389759718295169029293233132608565349386794777524737128372786108430727462251070255231796982768109049957154409649513706362 7
997159040453010203662222757997512556814265150536281134811659001651021334242885572007058859453722297973179204623615912060786559345754708342154742971053 8
890502712560792003181985755630051382292585620860935776160259751186588914689490830501274610019494075147622030506239524100247992090295482148916881930822 4
137585140578666062348592801171116496398182243063310517715299573050748888363006212771465627666669160366007015762718739434968019739842266149550534181209 2
491149720360629151562677899861343949216200873611340787848232273794823749656478792349207779235097130964831293140785555905571995026003052591792345824546 5
432125922044792156460512427051307291980332313434108010189100847293975632007481587954198316950325758964652304113660176246347379597548311031106109060912 3
389019476395661790450047209923618987555263084817830580597823896347107881968132682511546412869322964053910671782419863271442465982667024499796297702188 89
301719534288970634519466905959729345946211217669882546132465014521790750619539655622898953363974640783080220840617111679692160653220369364873633811506 8
116991508158970520889290697573565797616723576885244055344374310210855750180912500313227492968341576192888321151901867531270797067609073722621515704940 4
483091245723251849800006043122036575148306745813020104653842068645333314400953331042027645624745629824209516548518878025188102973198643923894458316326 3
120135326775844374615463723415588296941747767355383391001451123158247999647860085078540859525861457005060649447487976078592019704913606179831640588 00
591514986396475547395856557602152201820069763162836639536788669417159813075235358228287493277992627175606695590192679273467219181899148773497045431570 6
624852236243614272145428117696574252118313885439188242867463518588512894252445991114451750752006190216679921869970341724071677894392100648692763700374 6
093723811755912038686273911818204679436083591593569256007897536301217838359890051029794537350925188541661845969893380850693215359421998088239599286343 93
527487072011679111679969821946210687588219870046192651769835537669291582503389928171469133903896859384818669761721743159146651334684266680392646524170
823866469419477086238781539671101702654222715 4
The calculation of determinant costs 3248.404 ms
```

```
A's inversion cost:
The calculation of inversion costs 11320.888 ms
B's inversion cost:
The calculation of inversion costs 10504.639 ms
```

inversion

```
A's LU factorization cost:
The calculation of LU factorization costs 28489.778 ms
B's LU factorization cost:
The calculation of LU factorization costs 29145.200 ms
```

LU

```
The calculation of multiplication costs 438.130 ms
The calculation of multiplication using strassen costs 2675.709 ms (2048)
The calculation of multiplication using SIMD costs 791.091 ms
```

multiply

```
The running time is 150765.169 ms
```

Running time

2048 by 2048:

Transposition

A

1	63.2000	47.5000	46.7000	24.6000	8.6000	51.0000
2	26.5000	33.1000	47.1000	59.1000	6.1000	52.7000
3	14.8000	86.2000	53.7000	26.6000	96.7000	57.700
4	49.2000	82.0000	7.1000	60.3000	19.9000	18.0000
5	32.2000	6.6000	11.6000	7.8000	14.2000	15.1000

B

2050	86.0000	10.2000	25.7000	12.2000	31.0000	
2051	26.1000	68.4000	29.3000	74.5000	77.2000	
2052	49.9000	70.5000	71.9000	51.3000	37.1000	
2053	83.0000	42.4000	77.5000	70.5000	84.3000	
2054	74.8000	87.9000	70.7000	78.4000	15.6000	

Addition

1	149.2000	52.6000	64.7000	132.2000	107.0000	
2	57.7000	101.5000	156.7000	124.4000	94.5000	
3	72.4000	76.4000	125.6000	84.6000	82.3000	87
4	36.8000	133.6000	77.9000	130.8000	86.2000	1
5	39.6000	83.3000	133.8000	104.2000	29.8000	1

Subtraction

2050	-22.8000	0.4000	-35.1000	-33.8000	-42.6000	
2051	37.3000	-35.3000	15.7000	39.6000	-81.3000	-
2052	21.0000	17.8000	-18.2000	-70.4000	-59.1000	
2053	12.4000	-15.4000	-24.7000	-10.2000	-70.600	
2054	-22.4000	-71.1000	59.6000	-64.4000	-1.4000	

Inversion

A

4099	-22.8000	0.4000	-35.1000	-33.8000	-42.6000	
4100	37.3000	-35.3000	15.7000	39.6000	-81.3000	-
4101	21.0000	17.8000	-18.2000	-70.4000	-59.1000	
4102	12.4000	-15.4000	-24.7000	-10.2000	-70.600	
4103	-22.4000	-71.1000	59.6000	-64.4000	-1.4000	

B

6148	149.2000	52.6000	64.7000	132.2000	107.00	
6149	57.7000	101.5000	156.7000	124.4000	94.50	
6150	72.4000	76.4000	125.6000	84.6000	82.3000	
6151	36.8000	133.6000	77.9000	130.8000	86.20	
6152	39.6000	83.3000	133.8000	104.2000	29.80	

Multiplication

12295	5252342.0000	5053434.0000	5087805.0000	5006593.5000	5185730.5000	5085111.0000
12296	5087200.5000	4995318.0000	5043295.5000	4986767.5000	5186423.0000	5009396.0000
12297	5232800.0000	5175324.5000	5066050.0000	5082806.0000	5192962.0000	5140405.0000
12298	5280116.0000	5207315.0000	5143454.0000	5107790.5000	5388452.0000	5210511.0000
12299	5247541.0000	5108683.5000	5107082.5000	5049346.5000	5321387.5000	5184131.5000

Ladder : A

1	1.0000	0.4193	0.2342	0.7785	0.5095	0.7611
2	0.0000	1.0000	5.6949	3.4151	-1.3351	-1.892
3	0.0000	0.0000	1.0000	1.0815	-0.2154	-0.871
4	0.0000	0.0000	0.0000	1.0000	0.0331	-0.4127
5	0.0000	0.0000	0.0000	0.0000	1.0000	2.3322

B

2050	1.0000	0.3035	0.5802	0.9651	0.8698	
2051	0.0000	1.0000	0.9889	0.4985	1.2102	
2052	0.0000	0.0000	1.0000	1.1750	0.6250	
2053	0.0000	0.0000	0.0000	1.0000	-0.0329	
2054	0.0000	0.0000	0.0000	0.0000	1.0000	

The result of A’ s LU:

Lower triangular matrix

4099	1.0000	0.0000	0.0000	0.0000	0.0000	0.0
4100	0.7516	1.0000	0.0000	0.0000	0.0000	0.0
4101	0.7389	2.0874	1.0000	0.0000	0.0000	0.0
4102	0.3892	3.7006	2.2552	1.0000	0.0000	0.0
4103	0.1361	0.1892	-0.7063	-0.5401	1.0000	0

Upper triangular matrix

6148	63.2000	26.5000	14.8000	49.2000	32.20	
6149	0.0000	13.1831	75.0766	45.0222	-17.60	
6150	0.0000	0.0000	-113.9519	-123.2349	24.	
6151	0.0000	0.0000	0.0000	152.4644	5.0409	
6152	0.0000	0.0000	0.0000	0.0000	33.2082	7

The result of B’ s LU:

Lower triangular matrix

8197	1.0000	0.0000	0.0000	0.0000	0.0000	0
8198	0.1186	1.0000	0.0000	0.0000	0.0000	0
8199	0.2988	0.3292	1.0000	0.0000	0.0000	0
8200	0.1419	1.0841	-0.7220	1.0000	0.0000	
8201	0.3605	1.0381	-1.3416	1.4311	1.0000	

Upper triangular matrix

10246	86.0000	26.1000	49.9000	83.0000	74.8000	92.2
10247	0.0000	65.3044	64.5816	32.5558	79.0284	39.06
10248	0.0000	0.0000	35.7256	41.9781	22.3283	-36.21
10249	0.0000	0.0000	0.0000	53.7381	-1.7667	-35.575
10250	0.0000	0.0000	0.0000	0.0000	-60.9180	7.1390

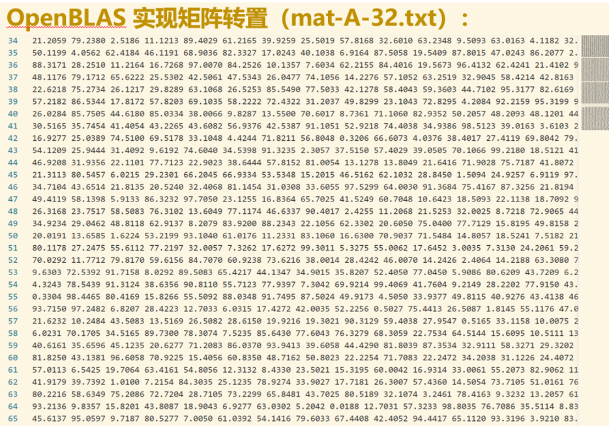
CMakeLists

```cmake
cmake_minimum_required(VERSION 3.16)

project (pro01)
MATH(EXPR stack_size "256 * 1024 * 1024")

FIND_PACKAGE(OpenMP REQUIRED)
if (OPENMP_FOUND)
    message("OPENMP FOUND")
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif ()

add_compile_options(-mavx2)
add_compile_options(-fopenmp)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -msse4.1")

add_executable(pro01 pro01.c func.c connect.h)
```

The Realization of OpenBLAS

32 by 32

```
linjiefang@LAPTOP-99UCO9DE:~/testOpenBLAS$ ./main mat-A-32.txt mat-B-32.txt result.txt
OpenBLAS calculates the multiplication costs 0.117 ms
OpenBLAS calculates the transposition costs 0.016 ms
OpenBLAS calculates the addition costs 0.007 ms
OpenBLAS calculates the subtraction costs 0.009 ms
The running time is 14.209 ms
```



256 by 256

```
linjiefang@LAPTOP-99UCO9DE:~/testOpenBLAS$ ./main mat-A-256.txt mat-B-256.txt result.txt
OpenBLAS calculates the multiplication costs 1.329 ms
OpenBLAS calculates the transposition costs 0.251 ms
OpenBLAS calculates the addition costs 0.155 ms
OpenBLAS calculates the subtraction costs 0.169 ms
The running time is 262.471 ms
```

OpenBLAS 实现矩阵乘法：

```
609701.1250 668480.1875 621181.6250 619568.8750 649069.1250 626989.1250 628410.1250 606410.8750 589301.5625
622537.7500 694945.7500 630179.6250 614834.7500 663380.1250 640656.3750 621345.6250 633641.6250 642800.1250
621054.2500 693947.3750 642418.8750 652366.6250 664910.2500 644445.3750 630180.1875 661696.8750 628899.1250
671541.9375 685620.5625 621037.9375 663779.6250 706507.5000 666723.1875 630142.5000 660631.8750 639101.6875
580221.8125 626748.5000 566113.6250 598267.0000 625390.3750 614839.0000 588099.3125 623297.7500 601829.8125
575252.4375 649868.2500 581206.1250 588119.5000 598075.2500 606192.1250 581955.6250 598612.6250 572940.3125
601826.3125 677101.6250 616942.3125 630811.7500 652230.0625 609822.3125 634520.8750 626716.5000 618361.9375
607880.4375 634730.8125 598152.5625 604251.6875 639393.6875 616323.0625 572012.3750 605252.0625 579111.2500
604122.3750 636984.1875 595389.8750 631224.5000 665548.0625 611791.8125 596632.5625 636197.2500 598902.6875
616666.4375 676415.3125 619935.8750 632277.0625 649560.3125 624673.1250 604028.3750 641494.2500 617165.0625
```

OpenBLAS 实现矩阵转置（mat-A-256.txt）：

```
 93.1000 26.5000 51.1000 98.7000 87.6000 1.7000 71.4000 27.5000 2.9000 91.3000 39.8000 44.1000 78.1000 16.1000
 44.0000 37.9000 82.4000 44.8000 72.7000 49.2000 86.7000 79.8000 47.8000 77.6000 27.2000 74.1000 44.3000 72.100
 71.3000 70.8000 7.2000 6.1000 39.1000 6.8000 75.6000 57.5000 59.4000 75.4000 78.0000 36.3000 22.6000 79.2000 6
 49.1000 32.2000 15.8000 48.0000 91.2000 0.0000 34.5000 67.8000 75.3000 90.0000 89.1000 97.7000 74.0000 19.5000
 21.3000 78.0000 42.2000 93.6000 8.2000 39.8000 30.5000 60.7000 84.1000 6.4000 70.8000 60.5000 90.3000 93.6000
 60.8000 96.8000 85.8000 26.3000 70.2000 56.2000 53.4000 19.2000 64.7000 74.8000 21.2000 98.4000 65.2000 28.000
 50.8000 59.3000 67.0000 73.4000 79.3000 59.7000 89.6000 30.9000 61.5000 90.1000 1.4000 89.8000 5.3000 49.9000
 46.7000 29.5000 64.7000 4.3000 67.8000 9.3000 56.0000 87.8000 66.8000 27.0000 67.0000 62.3000 52.2000 19.9000
 9.6000 38.5000 41.6000 42.5000 15.3000 95.3000 15.9000 18.8000 6.9000 58.2000 93.3000 97.6000 7.0000 18.0000 1
 24.2000 4.9000 26.0000 79.5000 66.7000 27.3000 6.6000 48.7000 85.0000 36.7000 31.3000 35.3000 82.4000 69.0000
 6.7000 46.1000 52.6000 97.8000 38.0000 9.9000 81.4000 80.2000 24.7000 16.2000 35.5000 57.4000 31.6000 27.3000
```

OpenBLAS 实现矩阵加法：

```
 128.3000 62.7000 135.1000 130.9000 92.2000 103.2000 64.8000 127.4000 66.7000 67.0000 80.7000 23.8000 84.1000 12
 84.5000 101.0000 136.2000 74.5000 152.3000 196.7000 156.6000 113.0000 126.3000 12.0000 143.7000 102.1000 63.100
 60.9000 140.4000 98.3000 100.5000 133.2000 102.1000 87.4000 77.5000 57.0000 46.6000 143.7000 72.4000 88.7000 10
 178.4000 69.9000 60.6000 48.9000 173.3000 103.9000 136.5000 48.1000 83.3000 81.1000 147.9000 80.3000 53.2000 15
 180.5000 74.8000 84.8000 93.6000 66.7000 114.4000 137.1000 88.1000 26.9000 164.1000 61.6000 140.2000 45.8000 11
 10.0000 122.8000 37.2000 95.3000 90.9000 140.8000 73.8000 52.2000 104.6000 93.6000 74.0000 76.9000 77.2000 109.
 164.6000 133.3000 166.3000 49.2000 125.5000 100.6000 148.2000 133.8000 94.2000 97.1000 162.4000 97.1000 37.5000
 109.1000 157.3000 97.4000 159.6000 83.8000 89.1000 101.9000 148.6000 49.2000 104.8000 80.8000 90.6000 101.5000
 94.0000 145.6000 73.2000 106.2000 177.7000 158.5000 120.4000 81.7000 24.2000 137.8000 28.5000 156.3000 116.8000
 125.0000 97.7000 172.6000 121.9000 96.8000 100.5000 111.0000 104.8000 61.6000 65.8000 36.6000 115.6000 148.3000
 99.5000 94.8000 137.4000 173.5000 152.4000 110.3000 55.0000 121.9000 156.1000 101.1000 132.5000 109.0000 83.800
```

OpenBLAS 实现矩阵减法：

```
 57.9000 25.3000 7.5000 -32.7000 -49.6000 18.4000 36.8000 -34.0000 -47.5000 -18.6000 -67.3000 15.2000 46.5000 -1
 -31.5000 -25.2000 5.4000 -10.1000 3.7000 -3.1000 -38.0000 -54.0000 -49.3000 -2.2000 -51.5000 86.7000 -21.1000 6
 41.3000 24.4000 -83.9000 -68.9000 -48.8000 69.5000 46.6000 51.9000 26.2000 5.4000 -38.5000 45.2000 -37.1000 39.
 19.0000 19.7000 -48.4000 47.1000 13.9000 -51.3000 10.3000 -39.5000 1.7000 77.9000 47.7000 -51.5000 -14.4000 -39
 -5.3000 70.6000 -6.6000 88.8000 -50.3000 26.0000 21.5000 47.5000 3.7000 -30.7000 14.4000 5.4000 14.6000 -57.000
 -6.6000 -24.4000 -23.6000 -95.3000 -11.3000 -28.4000 45.6000 -33.6000 86.0000 -39.0000 -54.2000 -41.1000 -22.40
 -21.8000 40.1000 -15.1000 19.8000 -64.5000 6.2000 31.0000 -21.8000 -62.4000 -83.9000 0.4000 -88.9000 -6.5000 -0
 -54.1000 2.3000 17.6000 -24.0000 37.6000 -50.7000 -40.1000 27.0000 -11.6000 -7.4000 79.6000 -83.8000 -97.1000 6
 -88.2000 -50.0000 45.6000 44.4000 -9.5000 -29.1000 2.6000 51.9000 -10.4000 32.2000 20.9000 34.3000 -29.8000 17.
 57.6000 57.5000 -21.8000 58.1000 -84.0000 49.1000 69.2000 -50.8000 54.8000 7.6000 -4.2000 -47.6000 8.9000 51.30
 -19.9000 -40.4000 18.6000 4.7000 -10.8000 -67.9000 -52.2000 12.1000 30.5000 -38.5000 -61.5000 -30.0000 -14.8000
```

2048 by 2048

```
linjiefang@LAPTOP-99UCO9DE:~/testOpenBLAS$ ./main mat-A-2048.txt mat-B-2048.txt result.txt
OpenBLAS calculates the multiplication costs 166.291 ms
OpenBLAS calculates the transposition costs 137.426 ms
OpenBLAS calculates the addition costs 112.967 ms
OpenBLAS calculates the subtraction costs 120.107 ms
The running time is 26815.441 ms
```

OpenBLAS 实现矩阵乘法：

```
1   5252342.0000 5053433.0000 5087801.5000 5006596.0000 5185735.5000 5085109.0000 5122404.0000 5107456.5000
2   5087196.0000 4995318.0000 5043294.0000 4986767.0000 5186429.0000 5009397.0000 5017119.5000 4946203.0000
3   5232809.0000 5175334.0000 5066052.0000 5082805.0000 5192954.5000 5140409.0000 5073379.0000 5165177.5000
4   5280119.0000 5207319.0000 5143448.5000 5107788.0000 5388450.5000 5210514.0000 5195746.0000 5205974.5000
5   5247539.0000 5108688.0000 5107083.5000 5049345.5000 5321386.0000 5184136.0000 5222848.0000 5215534.5000
6   5176192.5000 5081500.5000 5031236.5000 5011528.5000 5218720.0000 5137236.0000 5121147.0000 5156782.0000
7   5154794.0000 5012489.0000 5055651.5000 5006135.5000 5200883.0000 5117641.0000 5100247.0000 5074682.0000
8   5101008.0000 4962109.0000 4966409.0000 4921508.0000 5129892.0000 4938237.5000 5016163.0000 4980568.0000
9   5044815.5000 5027464.0000 5035615.0000 5016410.5000 5220034.0000 4991954.0000 5047684.5000 5042572.0000
10  5126887.0000 4983509.0000 5060329.5000 5039839.5000 5185980.0000 5082103.0000 4988288.0000 5047732.0000
```

OpenBLAS 实现矩阵转置（mat-A-2048.txt）：

```
2050   63.2000 47.5000 46.7000 24.6000 8.6000 51.0000 72.0000 96.9000 48.2000 21.8000 33.7000 28.2000 67.7000 1.0000
2051   26.5000 33.1000 47.1000 59.1000 6.1000 52.7000 13.9000 29.9000 37.8000 93.8000 6.7000 8.5000 71.6000 81.1000
2052   14.8000 86.2000 53.7000 26.6000 96.7000 57.7000 28.2000 2.4000 53.4000 31.3000 41.0000 39.0000 39.5000 60.400
2053   49.2000 82.0000 7.1000 60.3000 19.9000 18.0000 1.9000 11.6000 62.3000 86.9000 56.5000 78.9000 24.0000 47.7000
2054   32.2000 6.6000 11.6000 7.8000 14.2000 15.1000 22.2000 63.9000 92.4000 59.4000 31.1000 25.5000 56.4000 59.3000
2055   48.1000 11.2000 82.8000 87.5000 43.1000 43.8000 92.8000 98.2000 51.0000 24.3000 9.2000 39.2000 24.7000 75.300
2056   60.6000 41.9000 6.0000 19.9000 58.6000 62.6000 99.6000 15.3000 21.8000 84.7000 17.0000 1.1000 74.5000 56.3000
2057   45.8000 40.5000 31.6000 64.3000 84.7000 12.9000 63.5000 84.7000 61.7000 87.7000 97.3000 81.0000 24.9000 81.00
2058   2.2000 8.9000 54.8000 35.1000 36.5000 47.5000 6.3000 68.5000 81.4000 32.8000 94.8000 13.7000 52.8000 9.2000 2
2059   84.9000 47.0000 31.0000 71.0000 27.6000 74.2000 51.9000 23.8000 17.3000 25.9000 27.7000 88.7000 5.9000 25.300
2060   98.8000 36.9000 20.4000 96.8000 73.3000 58.9000 98.3000 35.4000 37.6000 24.0000 63.9000 90.3000 31.2000 64.00
2061   5.6000 89.9000 27.2000 19.0000 14.5000 98.5000 58.3000 78.0000 45.0000 74.4000 7.3000 19.1000 52.2000 86.4000
```

OpenBLAS 实现矩阵加法：

```
4099   149.2000 52.6000 64.7000 132.2000 107.0000 140.3000 74.1000 48.7000 79.5000 94.6000 108.9000 47.8000 106.1000 9
4100   57.7000 101.5000 156.7000 124.4000 94.5000 61.2000 92.1000 111.8000 19.9000 139.3000 122.4000 132.1000 39.7000
4101   72.4000 76.4000 125.6000 84.6000 82.3000 87.0000 39.2000 93.7000 114.0000 124.6000 96.3000 72.5000 123.7000 157
4102   36.8000 133.6000 77.9000 130.8000 86.2000 133.5000 96.1000 96.9000 115.9000 120.4000 131.6000 33.9000 141.9000
4103   39.6000 83.3000 133.8000 104.2000 29.8000 121.7000 129.7000 152.4000 68.7000 39.1000 156.4000 53.7000 12.0000 9
4104   149.6000 89.3000 101.0000 107.9000 81.1000 45.6000 123.3000 90.5000 61.9000 145.6000 138.2000 128.3000 167.6000
4105   111.5000 34.6000 108.0000 14.1000 39.9000 121.2000 132.5000 106.6000 76.4000 54.6000 186.0000 135.1000 142.1000
4106   195.6000 88.6000 59.4000 37.7000 65.5000 136.5000 83.9000 169.0000 140.9000 106.9000 89.9000 84.2000 132.1000 3
4107   137.3000 96.6000 114.1000 93.9000 93.3000 147.3000 35.6000 61.8000 114.9000 99.2000 75.7000 68.2000 104.0000 98
4108   95.5000 106.8000 121.4000 140.4000 130.8000 79.9000 158.7000 139.5000 43.2000 120.1000 57.3000 131.4000 39.1000
4109   48.1000 64.5000 109.3000 80.2000 86.4000 14.9000 36.7000 160.0000 157.7000 107.2000 86.9000 47.5000 93.7000 23.
```

OpenBLAS 实现矩阵减法：

```
6148   -22.8000 0.4000 -35.1000 -33.8000 -42.6000 -44.1000 47.1000 42.9000 -75.1000 75.2000 88.7000 -36.6000 -2.7000 4
6149   37.3000 -35.3000 15.7000 39.6000 -81.3000 -38.8000 -8.3000 -30.8000 -2.1000 -45.3000 -48.6000 47.7000 19.1000 3
6150   21.0000 17.8000 -18.2000 -70.4000 -59.1000 78.6000 -27.2000 -30.5000 -4.4000 -62.6000 -55.5000 -18.1000 -61.500
6151   12.4000 -15.4000 -24.7000 -10.2000 -70.6000 41.5000 -56.3000 31.7000 -45.7000 21.6000 62.0000 4.1000 35.5000 8.
6152   -22.4000 -71.1000 59.6000 -64.4000 -1.4000 -35.5000 -12.5000 17.0000 4.3000 16.1000 -9.8000 -24.7000 -9.8000 -1
6153   -47.6000 16.1000 14.4000 -71.9000 -50.9000 42.0000 1.9000 -64.7000 33.1000 2.8000 -20.4000 68.7000 0.6000 -44.0
6154   32.5000 -6.8000 -51.6000 -10.3000 4.5000 64.4000 66.7000 20.4000 -63.8000 49.2000 10.6000 -18.5000 -17.3000 64.
6155   -1.8000 -28.8000 -54.6000 -14.5000 62.3000 59.9000 -53.3000 0.4000 -3.9000 -59.3000 -19.1000 71.8000 20.7000 3.
6156   -40.9000 -21.0000 -7.3000 30.7000 91.5000 -45.3000 8.0000 61.6000 47.9000 -64.6000 -0.5000 21.8000 16.8000 -69.
6157   -51.9000 80.8000 -58.8000 33.4000 -12.0000 -31.3000 10.7000 35.9000 22.4000 -68.3000 -9.3000 17.4000 10.7000 -2
6158   19.3000 -51.1000 -27.3000 32.8000 -24.2000 3.5000 -2.7000 34.6000 31.9000 -51.8000 40.9000 -32.9000 -23.9000 13
```

CMakeLists

```
cmake_minimum_required (VERSION 3.16)
project (main)

set (TEST_VERSION 0.1)

set(CMAKE_BUILD_TYPE "Debug")
set(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb -DDEBUG")
set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")

set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/../testOpenBLAS)

# aux_source_directory(. DIR_SRC)

include_directories(${PROJECT_SOURCE_DIR}/../OpenBLAS)

# include_directories(${PROJECT_SOURCE_DIR}/include)

link_directories(${PROJECT_SOURCE_DIR}/../OpenBLAS)
add_executable(main main.c)

target_link_libraries(main libopenblas.a)
target_link_libraries(main -lm)
target_link_libraries(main -lpthread)
```

Difficulty and Solution

Get Matrix Size

Construct the matrix structure.

```
typedef struct {
    int m_rank; // the rank of matrix
    int m_row; //the row of matrix
    int m_column; //the column of matrix
    float *m_matrix; //the matirx
    Bool hasInversion; // does it have inversion
    Bool hasLU; // does it have LU factorization
} Matrix, *p_Matrix; // p_Matrix is the point of Matrix
```

Confirm the number of matrix columns by iterating the number of Spaces, and confirm the number of matrix rows by the number of carriage returns.

```
FILE *file = fopen(fileName, "rb");
while (fscanf(file, "%c", &c) != EOF) {
    if (c == '\n') {
        matrix->m_row++;
        num++;
    }
    if (c == ' ')
        num++;
}
num++;
```

Each float is read using the **fscanf()** method and stored in dynamically requested memory.

```
FILE *fid = fopen(fileName, "rb");
while (!feof(fid)) {
    fscanf(fid, "%f", matrix->m_matrix + (i++));
}
```

But this has the disadvantage that if you assume that you don't know the size of the matrix, it will take longer to iterate through the characters in the text to get the size for dynamically applying single-precision floating-point arrays. **fscanf ()** can only read float one by one, making the program running time increased.

Therefore, if you can determine the size of the array, you can save a character traversal, which can improve the speed of the program.

Delete Memory Space

In order to delete the dynamically allocated memory space, the free() method should be used properly. Because there is a float array inside the matrix structure, and the size of the matrix is 32 by 32, if you do not manually free, after the program terminates, the operating system will also free the memory. But if the size of the matrix is 2048 by 2048, because in the execution of a program to create a lot of array, applied for a lot of memory space, dynamic memory leak harm will be more obvious, such as abnormal program termination, etc., so need to release the memory matrix structure of the array, then release the memory structure itself, Ensure that unused memory is released correctly.

```
free(matrix->m_matrix);
free(matrix);
```

Addition/Subtraction/Transposition

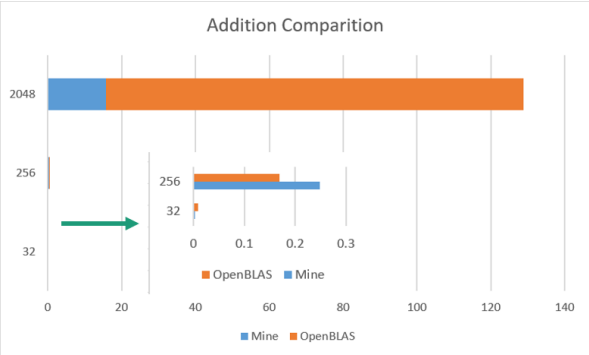
First additivity and subtract sexual judgment, by matrix operation rule, can pass judgment two interaction matrix is the same size to judge whether the two matrices can be added or reduced, if can undertake matrix operations, the program will continue, if not add or subtract the returns **NULL**, which can guarantee the integrity of the program.

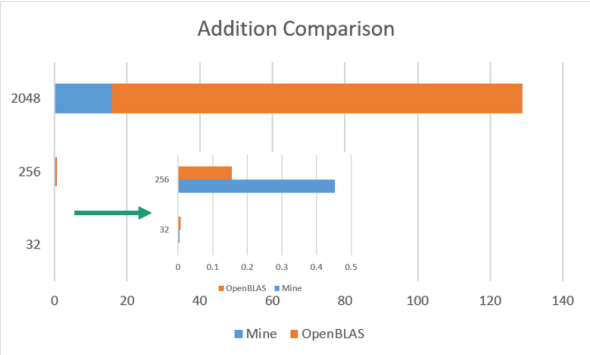
Matrix transpose is row and column swapping, **using compiler optimization, multi-parallelism, and loop rolling as well**.

The techniques used for addition and subtraction include **compiler optimization, multi-parallelism, loop rolling, and direct manipulation of address memory (more on these later)**. Thus the efficiency of matrix addition and subtraction is improved.

To sum up, two questions are raised:

The ***first*** problem is that compared with the running time of OpenBLAS's addition and subtraction, I found that the matrix calculation method written by myself was as fast as OpenBLAS, and even showed great speed advantage in matrix calculation at 2048 by 2048. As shown in the figure below.





The possible **reasons** are analyzed. Since the matrix and matrix operations in OpenBLAS library are mainly realized in matrix multiplication operation and special matrix multiplication operation commonly used in industry (**detailed later**), and the implementation of OpenBLAS is relatively universal, The **cblas_sgemm ()** method used in this project is implemented by passing in the parameters required in the following equation specified:

```
void cblas_sgemm(OPENBLAS_CONST enum CBLAS_ORDER Order,
                 OPENBLAS_CONST enum CBLAS_TRANSPOSE TransA,
                 OPENBLAS_CONST enum CBLAS_TRANSPOSE TransB,
                 OPENBLAS_CONST blasint M,
                 OPENBLAS_CONST blasint N,
                 OPENBLAS_CONST blasint K,
                 OPENBLAS_CONST float alpha,
                 OPENBLAS_CONST float *A,
                 OPENBLAS_CONST blasint lda,
                 OPENBLAS_CONST float *B,
                 OPENBLAS_CONST blasint ldb,
                 OPENBLAS_CONST float beta,
                 float *C,
                 OPENBLAS_CONST blasint ldc);
```

$$\alpha AB + \beta C = C$$

Then pass the above three matrices and two parameters into the method body and adjust the parameters:

$$A + B(C) = C(\alpha = 1, \beta = 1, B = E, C = B)$$

$$A - B(C) = C(\alpha = 1, \beta = -1, B = E, C = B)$$

Addition and subtraction can be realized, but perhaps because of the universality of this method, its ability for addition and subtraction is not very strong, that is, the pertinence is poor, and the speed is not so extreme.

Transpose is implemented with the parameter "**CblasTrans**" and is represented by an enumeration class defined internally.

The ***second*** problem is that the subtraction and addition of operations have subtle differences in the aspect of hardware, subtraction should step to take the first operating more than additive and therefore should be subtraction is slow in addition to, but after starting addition subtraction calculation sequence of time is slower than subtraction of addition, and in some cases large difference.

"Mathematically, the ALU in the CPU does only two things arithmetically, addition, shift, at most plus inversion, and logically, only and or neither xor."

ADDITION - ADDITION
SUBTRACTION - NEGATION -> ADDITION

However, when addition and subtraction are swapped, the appearance of addition is better than subtraction, which is just the opposite of the previous situation. Therefore, it can be considered as the function of the **cache mechanism**. Since addition and subtraction are similar, the cache mechanism is conducive to the use of cache when the latter method is executed, thus improving the operation speed. The analysis chart is as follows:

CONDITION\ TIME(2048)	ADDITION	SUBTRACTION
ONLY ADDITION	5.351 ms	-
ONLY SUBTRACTION	-	5.556 ms
ADDITION -> SUBTRACTION	8.268 ms	5.563 ms
ADDITION -> ADDITION	5.611 ms	9.734 ms

Rank/LU Factorization/Determinant

To find the **rank** of the matrix, **gauss-Jordan method** is adopted to carry out Gaussian elimination. In the process of elimination row by row, it is determined whether the row is 0 row. If 0 row appears, the matrix is not rank, otherwise, the matrix is full rank.

According to the definition of matrix properties, only a matrix with full rank can be decomposed into LU, so whether the rank of a square matrix is equal to its row can be determined by **LU decomposition**. Then the upper triangular matrix is calculated first, then the lower triangular matrix is calculated, this process uses **gauss-Jordan elimination method**, and finally the upper triangular matrix and the lower triangular matrix are returned.

The gauss-Jordan elimination method is also used to find the **determinant** of the matrix. The matrix is transformed into a diagonal matrix, and the determinant is multiplied by the number of diagonal lines in the diagonal matrix. Finally, the positive and negative of the determinant is determined by the number of row exchanges. In the process, however, it was found that a matrix of only 32 by 32 had a determinant that exceeded the maximum value of any basic data type, so large numbers had to be multiplied to output the result as characters. The code for multiplying large numbers is as follows :

```
void IntMultiply(char *a, const char *b, int a_len, int b_len) {
    int c[a_len + b_len];
    memset(c, 0, sizeof(int) * (a_len + b_len));
    int i, j;
#pragma omp parallel for
    for (i = 0; i < a_len; ++i) {
        for (j = 0; j < b_len; ++j) {
            c[i + j + 1] += (a[i] - 48) * (b[j] - 48);
        }
    }
    for (i = a_len + b_len - 1; i > 0; --i) {
        if (c[i] > 9) {
            c[i - 1] = c[i - 1] + c[i] / 10;
            c[i] = c[i] % 10;
        }
    }
    int zeroNum = 0;
```

```
    j = 0;
    for (i = 0; i < a_len + b_len; ++i) {
        if (c[i] == 0)
            zeroNum++;
        if (zeroNum != i + 1) {
            a[j] = (char) (c[i] + 48);
            j++;
        }
    }
}
```

Multiple parallelism is also used in this method.

Inverse

The inverse of the matrix first satisfies the condition that the determinant is not zero, which is used as a judgment condition to improve the robustness of the function. Methods Gauss-Jordan elimination method was adopted:

$$A|I = I|A^{-}$$

And returns the pointer to the inverse matrix structure.

Multiplication

Here, three different optimization methods are used to accelerate matrix multiplication, which are based on **hardware, algorithm and SIMD** respectively.

compilation optimization

First, analyze compilation optimization. Because compilers are constantly updated and iterated, compilers today are capable of providing advanced code optimization. This project adopts O3 optimization. After opening O3 optimization, the following help can be provided to accelerate the efficiency of this program :

.-floop-optimize : **Optimizes how to generate loops in assembly language.** Typically, programs are made up of many large and complex loops, and by removing variable assignments that do not change their value within the loop, you can reduce the number of instructions executed within the loop and greatly improve performance. Also optimize conditional branches that determine when to leave the loop to reduce the impact of the branches.

.-fcprop-registers :Because registers are assigned to variables in the function, the compiler performs a second check to reduce scheduling dependencies (both segments require the same register) and to **remove unnecessary register copy operations**.

.-fforce-mem : **This optimization forces all variables stored in memory locations to be copied to registers before any instruction uses variables.** For variables that are involved in many instructions (requiring mathematical manipulation), this is a significant optimization because the processor can access a value in a register much faster than it can access a value in memory.

.-fstreng-reduce : This optimization technique **optimizes the loop and removes iterated variables**. Iterative variables are variables that are bundled to a loop counter, such as a for-next loop that uses a variable and then uses a loop counter variable to perform mathematical operations.

.-frerun-CSE-after-loop : This technique **reruns the generic subexpression elimination routine** after any loop has been optimized. This ensures that the code is further optimized after the loop is unwrapped.

.-fschedule-insns : The compiler will **try to rearrange instructions so as to eliminate processors waiting for data.** For processors that have a delay when performing floating-point operations, this allows the processor to load other instructions while waiting for floating-point results.

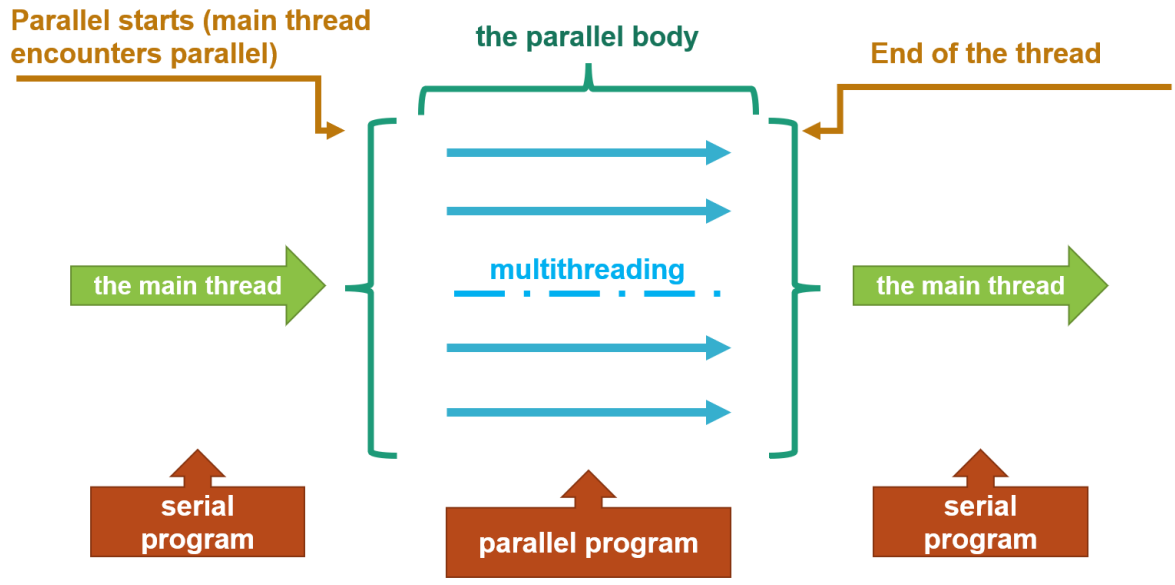
.-fcaller-Saves : This option instructs the compiler to **save and restore registers on function calls** so that the function can **access register values without having to save and restore them.** If multiple functions are called, this saves time because registers are saved and restored only once, rather than on every function call.

.-fweb : **builds a pseudo register network for saving variables.** Pseudo registers contain data as if they were registers, but can be optimized using various other optimization techniques, such as CSE and LOOP optimization techniques.

Through the compiler optimization program execution efficiency has been significantly improved.

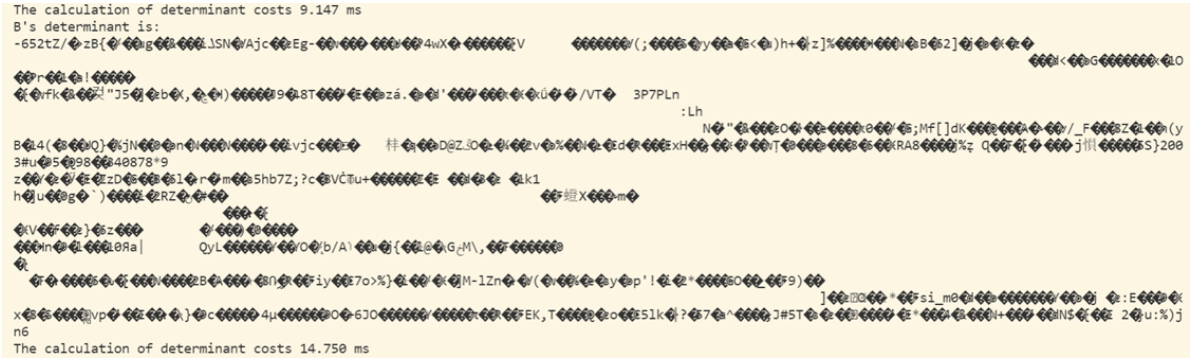
OpenMP

The preceding functions can be implemented only when the OpenMP execution mode is enabled. OpenMP uses the **fork-join execution** mode. You start with **a single main thread**, and when parallel computation is required, **branch threads are spawned to perform parallel tasks**. When the parallel code execution is complete, the branch threads merge and hand control over to a single main thread **(as shown below)**.



In hardware optimization, **multi-thread optimization** is used, that is, the main thread is put into parallel state by "#pragma omp parallel for" statement. Multi-core parallelism can be utilized due to the simple logic in the for loop. In this compilation instruction, parallel and for instruction are combined. Also used before the for loop statement, the code that represents the body of the for loop will be executed in parallel by multiple threads. It has two functions of parallel domain generation and task sharing at the same time, making the program execution efficiency improved.

However, if the OpenMP execution mode is not used correctly, **such as opening OpenMP mode before the while loop**, data will be corrupted and garbled (as shown in the figure below).



And because of multi-parallel processing, easy to **cause data confusion due to crowded CPU memory, the accuracy of the results of the decline**, but this situation rarely happens (in the use of this program rarely appear data errors), **error types include negative numbers in the positive addition operation results and so on**.

Loop Rolling

Now when programming in a high-level language, you don't have to do loop unwrapping because the compiler does it for you. However, loop unrolling is important in CUDA programming because it gives the thread bundle scheduler more threads to use to effectively hide latency. So we can use the method of loop unrolling to further optimize the parallel reduction program.

By reducing instruction consumption and adding more independent scheduling instructions, more concurrent operations are added to the pipeline, resulting in higher instruction and memory bandwidth. At the macro level, the overall time of program execution is reduced.

This project will be the inner loop step length is set to 4 (32, 256, 2048, can be divided exactly by 4, the space in time the compromise between the optimal position need to do concrete analysis for concrete problem, so the selection as step 4), the processing, speed and no obvious change, so you can think that the compiler automatically optimize has to achieve the effect of manual optimization.

```
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            c = *(A + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(C_mat + i * B_column + j) += c * *(B + k * B_column + j);
                *(C_mat + i * B_column + j + 1) += c * *(B + k * B_column + j +
1);
                *(C_mat + i * B_column + j + 2) += c * *(B + k * B_column + j +
2);
                *(C_mat + i * B_column + j + 3) += c * *(B + k * B_column + j +
3);
            }
        }
    }
```

IKJ

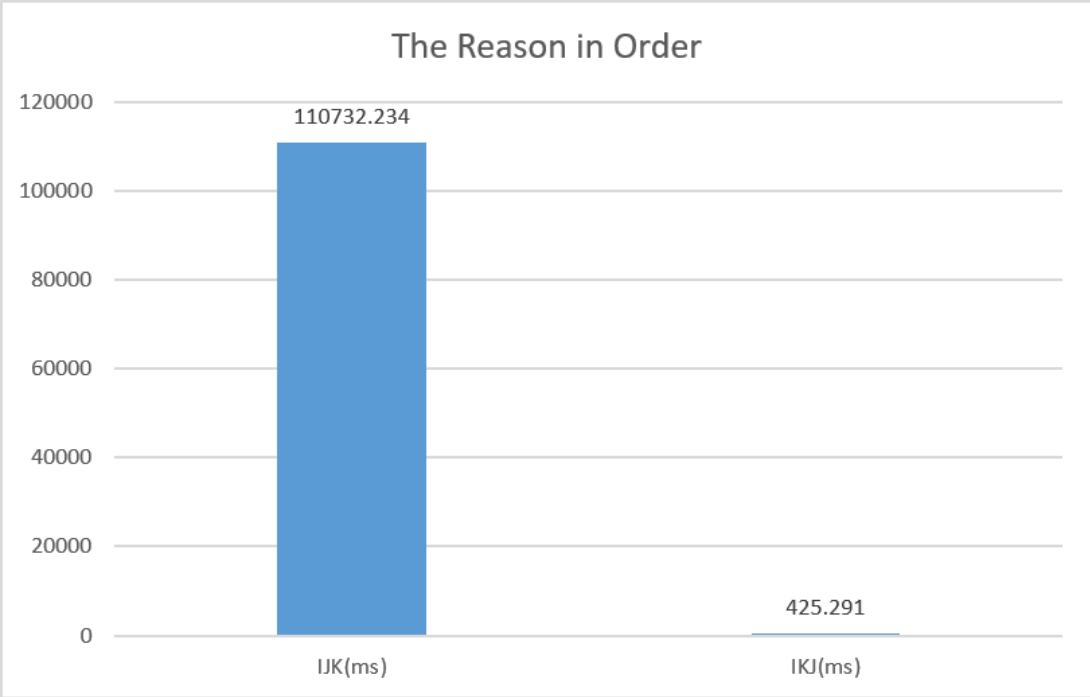
Changing the order in which IJK is run can significantly improve the running time of the program. The reason for this is that intermediate results can be extracted much faster from a level 1 Cache than from a level 2 Cache, and **some loop sequences exploit the computer's Cache structure**. In addition, **the calculation of some results depends on the results of previous calculations. When the previous results are not available, some calculations start to queue up and wait**.

Therefore, the **unreasonable arrangement of calculation order** leads to the unnecessary increase of calculation time.

The order of substitution is also worth thinking about.

ORDER	SPEED
IKJ	N^2
IJK	$N^3 + N^2 - N$
KIJ	$2N^2$
KJI	$2N^3$
JIK	$N^3 + N^2 + N$
JKI	$2N^3 + N^2$

As shown in the figure above, IKJ has the highest running rate, so the sequence of IKJ is adopted in this project.



As shown in the figure above, the speed difference is very large (**tens of times**), and **IKJ improves performance significantly**.

*VS Previous Optimization

In project 2, **many attempts have been made to optimize the running efficiency of the program, and compilation optimization, IKJ, loop Rolling and multi-parallel optimization have also been adopted**. In this project, IKJ, Loop Rolling and multi-parallel are simultaneously **applied to other matrix functions** with a more detailed understanding of compilation optimization.

Compared with the previous project, attempts to improve operation efficiency in this project include the implementation of **SIMD and Strassen algorithms**, and the final results of **these two optimization approaches are not more efficient than** the matrix multiplication program shown below (the previous implementation).

```
//Show core code
#pragma GCC optimize(3)
```

```
//The core implementation in the method
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            c = *(A + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(C_mat + i * B_column + j) += c * *(B + k * B_column + j);
                *(C_mat + i * B_column + j + 1) += c * *(B + k * B_column + j +
1);
                *(C_mat + i * B_column + j + 2) += c * *(B + k * B_column + j +
2);
                *(C_mat + i * B_column + j + 3) += c * *(B + k * B_column + j +
3);
            }
        }
    }
```

SIMD

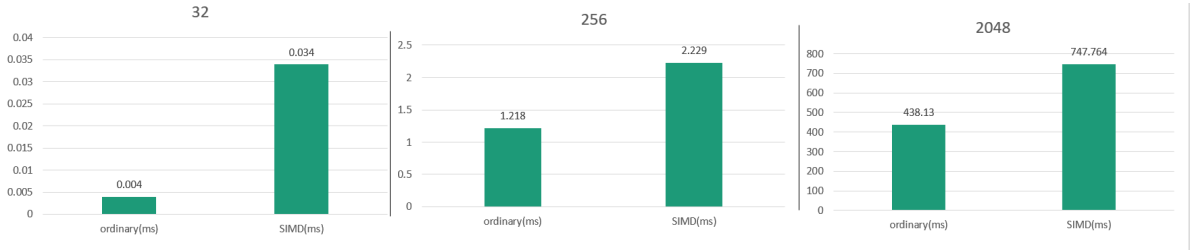
SIMD (Single Instruction Multiple Data) is a Single Instruction Multiple Data stream that can read Multiple operands and pack them in a set of instructions in a large register. After obtaining multiple operands at a time, they are stored in a large register and then calculated, so as to achieve the effect of calculating multiple objects with one instruction and realize acceleration. This test equipment has good support for SIMD calculation, so it tries to optimize the calculation efficiency by SIMD.

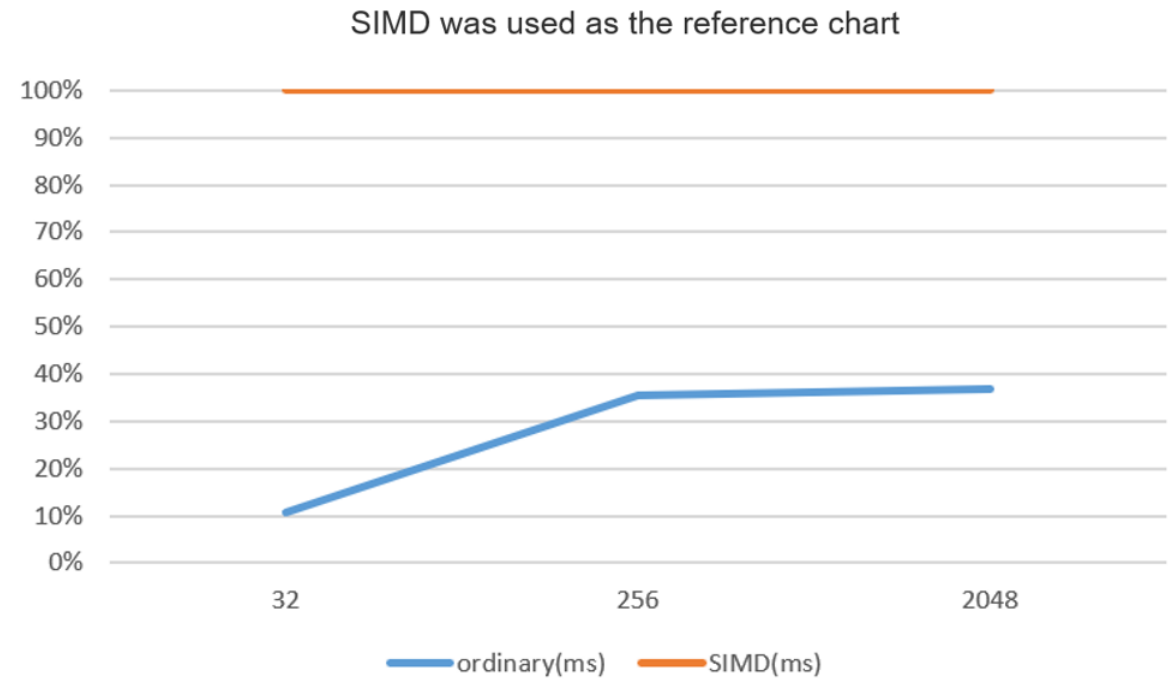
The implementation code is as follows:

```
#include <immintrin.h>
#pragma GCC optimize(3)

void mulSIMD(Matrix * c, const Matrix  * a, const Matrix * b ) {
    if (a->m_column != b->m_row){
        printf("MULTIPLY FAILURE!");
        return;
    }
    gettimeofday(&tv, NULL);
    __m256 num0, s;
    int i, j, k, n;
#pragma omp parallel for
    for (i = 0; i < a->m_row; ++i) {
        for (k = 0; k < a->m_column ; ++k){
            s = _mm256_broadcast_ss(a->m_matrix + i * a->m_column + k);
            for (j = 0; j < b->m_column; j += 8) {
                n = i * c->m_column;
                num0 = _mm256_loadu_ps(c->m_matrix + n + j);
                num0 = _mm256_add_ps(num0, _mm256_mul_ps(s,_mm256_loadu_ps(b-
>m_matrix + k * b->m_column + j)));
                _mm256_storeu_ps(c->m_matrix + i * c->m_column + j, num0);
            }
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of multiplication using SIMD costs %.3f ms\n",
period);
```

```
}
```

The following figure shows the influence of the use of SIMD and the use of SIMD on the operation efficiency of matrix multiplication. It is not difficult to find that the use of **SIMD has reduced the efficiency**, which is still a disadvantage although the decline is not large. Therefore, the following guesses can be made:





First, in the case of some simple operations, the compiler will automatically convert SIMD code into SIMD code through some optimization techniques in the case that you do not actively use SIMD, so **the fastest SIMD code implemented by yourself will not exceed the results of automatic optimization by the compiler**.

For this simple calculation, **the efficiency improvement brought by SIMD cannot cover the extra performance loss brought by SIMD**. Perhaps SIMD can only accelerate the calculation when the calculation is more complicated.

Strassen

TIME COMPLEXITY	ALTHORITHM
N^3	
$N^{2,81}$	Strassen (1969)
$N^{2.376}$	Coppersmith-Winograd (1990)
$N^{2.374}$	Stothers (2010)
$N^{2.3729}$	Williams (2011)
$N^{2.37287}$	Le Gall (2014)

In order to reduce the complexity of matrix multiplication, many mathematicians in the world put forward many algorithms, but there are still many algorithms have not been universally recognized, and **the Strassen algorithm used most in industry**, its time complexity is 2 to the power of 2.81.

This approach takes a divide-and-conquer approach to recursion and simplifies multiplication as much as possible into more computer-friendly calculations. **The algorithm can be realized by converting the following figure into C language.**

2.2 Strassen's Algorithm

Strassen's 1969 algorithm, which gives $\omega < 2.81$ follows similarly. (For reference see [Str69], [Wik09], [BCS97] pages 10-14 or almost any book on algebraic algorithms). Let $A, B, C \in \mathbb{R}^{2\times 2}$.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix} \tag{2}$$

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\ \mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \end{aligned} \tag{3}$$

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\ \mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\ \mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{aligned}$$

Instead of using the 8 multiplications of the trivial approach, Strassen's algorithm only uses 7. Applying a divide and conquer strategy recursively (view $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$ as matrices instead of scalars) allows matrix multiplication over $n = 2^N$ size matrices to be performed using only $7^N = 7^{\log_2 n} = n^{\log_2 7} = O(n^{2.81})$ multiplications.

In the specific implementation of this algorithm in the last project, **the key ending recursion condition and the wrong operation of memory space were ignored**, which led to its very low efficiency in matrix multiplication operation, and even directly failed to run normally when calculating the matrix of 2048 times 2048. Therefore, **three improvements of this project** are summarized, which enables the method to run normally. **Although the efficiency is lower than IKJ and SIMD, it is much faster than the efficiency of brute force calculation matrix multiplication.**

Firstly, **the setting of recursive termination conditions should meet the requirement that, in the case of small matrix scale, recursion should be withdrawn and IKJ should be used to complete subsequent calculations.** This does not make recursion meaningless, consuming not only memory but also execution efficiency. In this method, the termination condition is that IKJ is

used to complete the subsequent calculation **if the matrix size is less than 64**. **Further reduction or increase of this value will lead to the decline of operating efficiency.**

```
void strassenMul(const float *a, const float *b, float *c, int A_row, int
A_column, int B_column) {
    for (int i = 0; i < A_row; ++i) {
        for (int j = 0; j < B_column; ++j) {
            *(c + i * B_column + j) = 0.0F;
        }
    }
    float t;
    int i, j, k;
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            t = *(a + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(c + i * B_column + j) += t * *(b + k * B_column + j);
                *(c + i * B_column + j + 1) += t * *(b + k * B_column + j + 1);
                *(c + i * B_column + j + 2) += t * *(b + k * B_column + j + 2);
                *(c + i * B_column + j + 3) += t * *(b + k * B_column + j + 3);
            }
        }
    }
}

//In the method body - Strassen
if ((a_column % 2 != 0 || b_column % 2 != 0 || a_row % 2 != 0) || a_row < 64) {
        strassenMul(a, b, c, a_row, a_column, b_column);
        return;
}
```

The **second** point is to **apply dynamically in the memory application process**, otherwise it will lead to stack overflow, resulting in abnormal termination of the program.

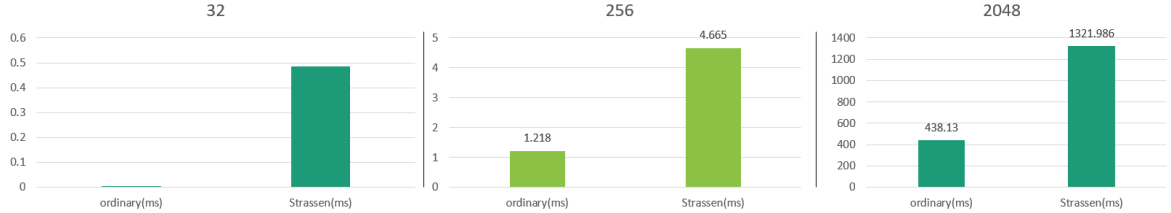
```
//In the method body - Strassen
float *M1 = (float *) malloc(sizeof(float) * a_row * b_column);
float *M2 = (float *) malloc(sizeof(float) * a_row * b_column);
float *M3 = (float *) malloc(sizeof(float) * a_row * b_column);
float *M4 = (float *) malloc(sizeof(float) * a_row * b_column);
float *M5 = (float *) malloc(sizeof(float) * a_row * b_column);
float *M6 = (float *) malloc(sizeof(float) * a_row * b_column);
float *M7 = (float *) malloc(sizeof(float) * a_row * b_column);
float *A11 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A12 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A22 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A21 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A11022 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A21022 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A11012 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A21111 = (float *) malloc(sizeof(float) * a_row * a_column);
float *A12122 = (float *) malloc(sizeof(float) * a_row * a_column);
float *B22 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B11 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B21 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B12 = (float *) malloc(sizeof(float) * a_column * b_column);
```

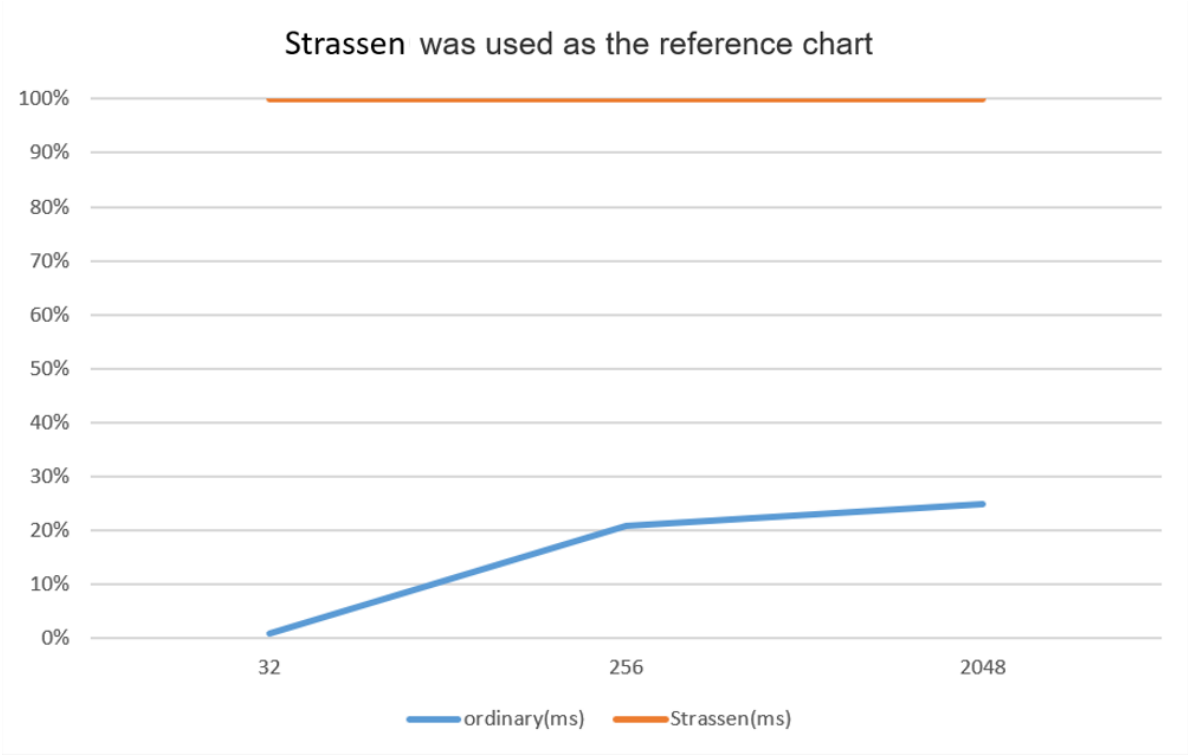
```
float *B11022 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B12122 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B21111 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B11012 = (float *) malloc(sizeof(float) * a_column * b_column);
float *B21022 = (float *) malloc(sizeof(float) * a_column * b_column);
```

Third, **manually release the memory after it is used**, otherwise it will cause memory leaks and, in the case of large matrix size, will cause the method to terminate abnormally.

```
//In the method body - Strassen
free(M1);
free(M2);
free(M3);
free(M4);
free(M5);
free(M6);
free(M7);
free(A11);
free(A12);
free(A22);
free(A11012);
free(A11022);
free(A12122);
free(A21022);
free(A21111);
free(B11);
free(B12);
free(B21);
free(B22);
free(B11012);
free(B11022);
free(B12122);
free(B21022);
free(B21111);
```

In conclusion, although the algorithm reduces the time complexity of matrix multiplication, the method call itself takes time because **it is a recursive call to the method. In addition, the method body applies for a large array for many times and uses the original array to assign values to a small array in the method body for many times**. In the above process, **not only the complexity of the program is increased, which is not conducive to compiler optimization, but also the multi-parallelism and other optimization approaches are not able to be used. In addition, the demand for memory space and stack space is large, which lead to the running efficiency of this algorithm is lower than the first two methods (IKJ and SIMD)**. The efficiency comparison chart is shown below.





Impact of Calling Function

The reason why function call is time-consuming in C language is that before calling a function, it is necessary to retain the running state of the function before calling it, including the value of variables and so on, and then execute the contents in the function body, so the function call itself needs a certain amount of time consumption.

In this project, many functions build a new matrix structure inside the function and return the pointer to that structure, so the initialization of the structure variable is ignored when calculating the running time, so the calculation results are accurate. However, during the use of OpenBLAS method, the matrix structure is initialized outside the function body, **so the calculation of time differs from that of function call**. Therefore, in order to eliminate the interference of function call time, the following method body is designed to simulate OpenBLAS, and the matrix structure is initialized outside the function body and then passed in as a parameter.

The following code is the OpenBLAS runtime calculation.

```
gettimeofday(&up, NULL);
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha,(float *)
a, lda,(float *) E, ldb, beta_positive,(float *) c, ldc);
gettimeofday(&down,NULL);
period = (down.tv_sec - up.tv_sec)*(double)1000 + (down.tv_usec -
up.tv_usec)/(double)1000;
printf("OpenBLAS calculates the addition costs %.3f ms\n",period);
```

Here is how time is calculated for this project.

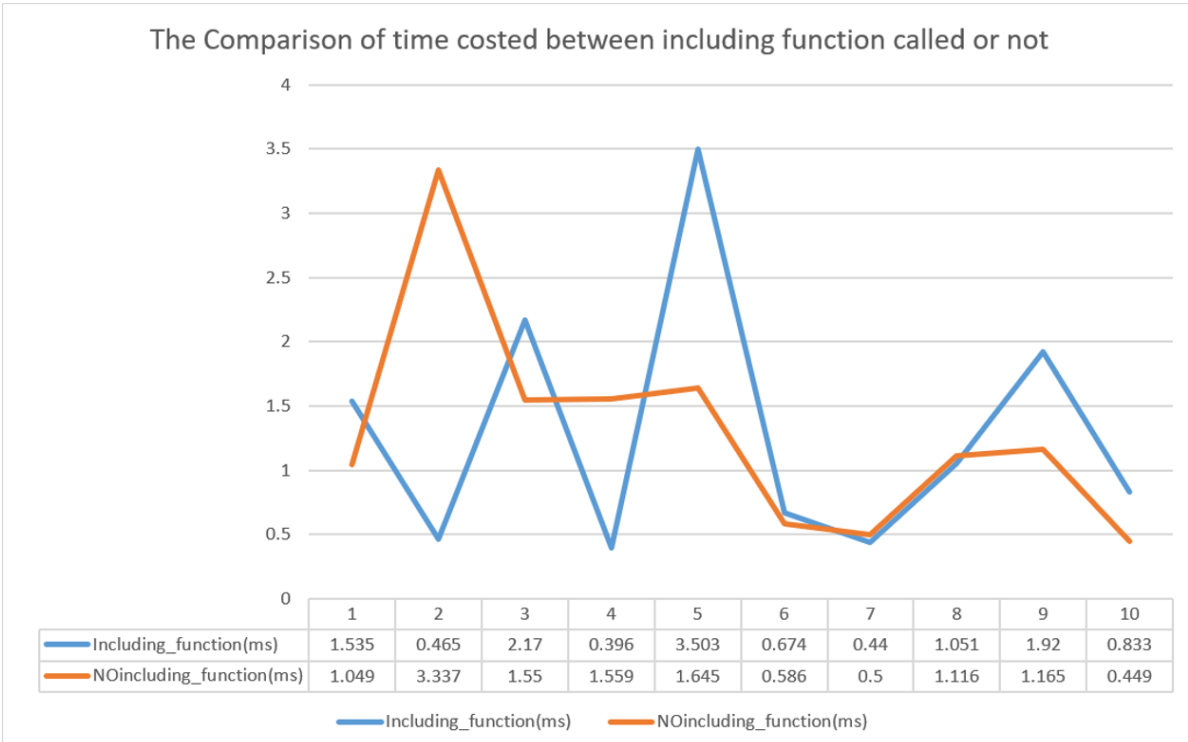
```
p_Matrix mul = multiply(A->m_matrix, B->m_matrix, A->m_row, A->m_column, B-
>m_row, B->m_column);

p_Matrix multiply(const float *A, const float *B, int A_row, int A_column, int
B_row, int B_column) {
    if (A_column != B_row) {
        printf("MULTIPLY FAILURE!");
```

```
        return NULL;
    }
    p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
    C->m_row = A_row;
    C->m_column = B_column;
    C->m_matrix = (float *) malloc(sizeof(float) * A_row * B_column);
    float *tem = C->m_matrix;
    for (int i = 0; i < A_row; ++i) {
        for (int j = 0; j < B_column; ++j) {
            *(tem++) = 0.0F;
        }
    }
    float *C_mat = C->m_matrix;
    gettimeofday(&tv, NULL);
    float c;
    int i, j, k;
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            c = *(A + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(C_mat + i * B_column + j) += c * *(B + k * B_column + j);
                *(C_mat + i * B_column + j + 1) += c * *(B + k * B_column + j +
1);
                *(C_mat + i * B_column + j + 2) += c * *(B + k * B_column + j +
2);
                *(C_mat + i * B_column + j + 3) += c * *(B + k * B_column + j +
3);
            }
        }
    }
    gettimeofday(&f_tv, NULL);
    double period = (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000;
    printf("The calculation of multiplication costs %.3f ms\n", period);
    return C;
}
```

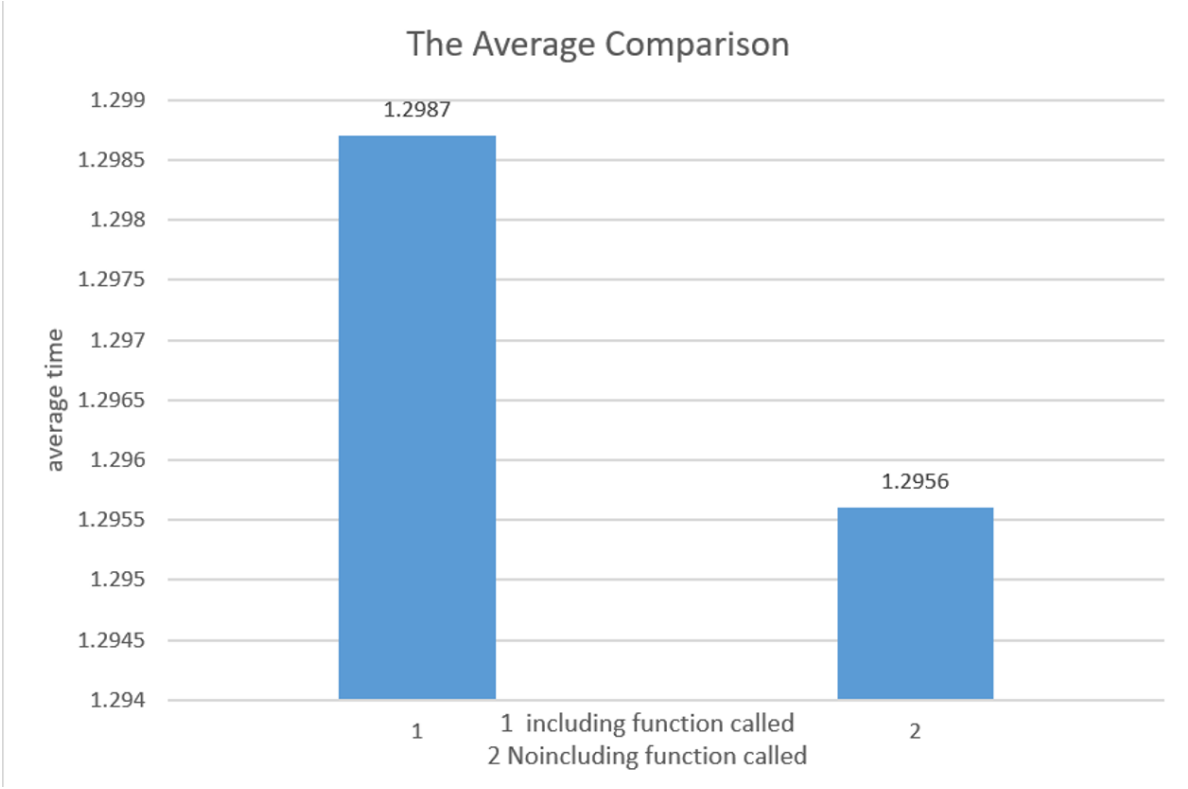
To test the effect of a function call on the computation time, the function and its call and run time are calculated as follows.

```
struct timeval tv, f_tv;

p_Matrix C = (Matrix *) malloc(sizeof(Matrix));
C->m_column = B->m_column;
C->m_row = A->m_row;
C->m_matrix = (float *) malloc(sizeof(float) * C->m_row * C->m_column);
for(int i = 0 ; i < A->m_row ; ++i){
    for(int j = 0 ; j < B->m_column ; ++j){
           *(C->m_matrix + i * B->m_column + j) = 0.f;
    }
}
gettimeofday(&tv, NULL);
multiply_func(A->m_matrix, B->m_matrix, C->m_matrix, A->m_row, A->m_column, B-
>m_row, B->m_column);
gettimeofday(&f_tv, NULL);
```

```
printf("%.3f \n", (f_tv.tv_sec - tv.tv_sec) * (double) 1000 + (f_tv.tv_usec -
tv.tv_usec) / (double) 1000);

void multiply_func(const float *A, const float *B, float *C, int A_row, int
A_column, int B_row, int B_column) {
    if (A_column != B_row) {
        printf("MULTIPLY FAILURE!");
        return;
    }
    float c;
    int i, j, k;
#pragma omp parallel for
    for (i = 0; i < A_row; i++) {
        for (k = 0; k < A_column; k++) {
            c = *(A + i * A_column + k);
            for (j = 0; j < B_column; j += 4) {
                *(C + i * B_column + j) += c * *(B + k * B_column + j);
                *(C + i * B_column + j + 1) += c * *(B + k * B_column + j + 1);
                *(C + i * B_column + j + 2) += c * *(B + k * B_column + j + 2);
                *(C + i * B_column + j + 3) += c * *(B + k * B_column + j + 3);
            }
        }
    }
}
```



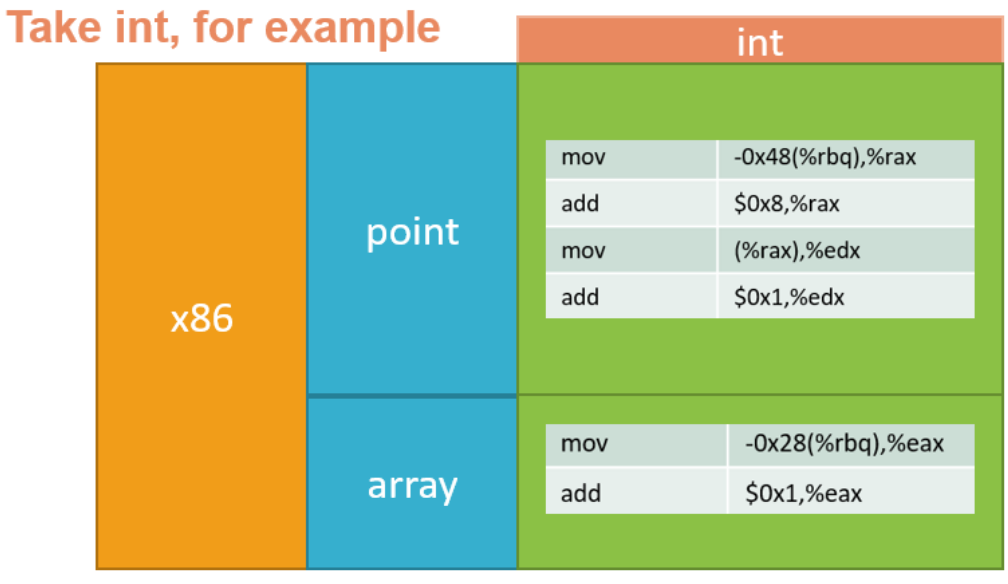
	1	2	3	4	5	6	7	8	9	10
Including_function(ms)	1.535	0.465	2.17	0.396	3.503	0.674	0.44	1.051	1.92	0.833
NOincluding_function(ms)	1.049	3.337	1.55	1.559	1.645	0.586	0.5	1.116	1.165	0.449



It can be concluded from the above two figures that in this project, due to the short code of the main method and the small number of variables, the time consumed by the function call is negligible.

Array and Address Access

Whether access through an array index or through an address is more efficient depends on the situation, and there is little difference between the two when optimized at compile time.



Above integers, for example, you can know through **"(* (a + I)) + +"** needs to be two addition operation and mobile operations, and through **"arr [I] + +"** only need a addition and a mobile operation, thus addressing an array faster, but if the index is expressed as additive form, so the gap between the two is more little, Example: **arr[a + I]++**.

File Writer

```
void fileWriter(FILE *fp, p_Matrix matrix) {
    const float *m = matrix->m_matrix;
    for (int i = 0; i < matrix->m_row; ++i) {
        for (int j = 0; j < matrix->m_column; ++j) {
            fprintf(fp, "%.4f ", *(m + i * matrix->m_column + j));
        }
        fprintf(fp, "\n");
    }
    fprintf(fp, "\n");
}
```

Simply print the results to a txt file with **fprintf()**.

*Adjoint Matrix

The adjoint matrix can be calculated in the following ways :

$$A* = det * A^{-}$$

Where the determinant and inverse have been solved, so the solution of the adjoint matrix is ignored.

Reference

https://blog.csdn.net/weixin_43800762/article/details/87811697

https://blog.csdn.net/weixin_43800762/article/details/87811697

https://www.csdn.net/tags/OtDakg4sMzQ2MTMtYmxvZwO0O0OO0O0O.html

And a lot of blogs on CSDN.