CS205 C/C++ Program Design - Project 4

Name: 秦禹洲(Qin Yuzhou)
SID: 12110601
Code: https://github.com/QinYuzhou/C-Program-Design-Project-4

Part 1 - Analysis

这次Project的主要任务为使用C语言优化矩阵乘法，并且比较普通的矩阵乘法，优化后的矩阵乘法和OpenBLAS矩阵乘法效率之间的差距。

矩阵的存储和读取

本次作业仍然是使用了一个struct来存储矩阵的信息，具体包括矩阵的行数，列数和矩阵中的元素，其中使用一个一维指针来存储矩阵中的元素，这样在调用时会更加快速。为了便于比较速度，我先写了一个数据生成程序，分别生成 16x16，128x128，1Kx1K，8Kx8K和64Kx64K的矩阵并存储在文件中，文件中的第一行是两个整数，分别代表矩阵的行数和列数，之后则是矩阵中的元素，最后通过读取文件中的信息来进行矩阵乘法运算并计算运算时间和准确率。

矩阵部分函数的实现

主要实现了创建矩阵，删除矩阵与输出矩阵三个函数，这三个函数主要是从Project3中直接复制而来，但是由于本次Project中会使用SIMD优化，所以在创建矩阵分配空间时，我没有使用malloc函数，而是使用aligned_alloc函数使为矩阵分配的空间对齐。

由于本次Project中需要比较不同矩阵乘法的效率和准确度，所以我写了一个similar函数来比较两个矩阵之间的元素差异最后返回两个矩阵中相同位置的差的绝对值最大值。

```
double similar(Matrix *mat_1, Matrix *mat_2) //比较两个矩阵是否相似
{
    if (mat_1 == NULL || mat_2 == NULL)
        return -1;
    if (mat_1->col != mat_2->col || mat_1->row != mat_2->row)
        return -1;
    double differ = 0;
    for (size_t i = 0; i < mat_1->col * mat_2->row; i++)
        differ = fmax(differ, fabs(mat_1->data[i] - mat_2->data[i]) / fmin(mat_1->data[i],
    return differ;
}
```

矩阵乘法的计时

本次Project中，我使用了gettimeofday函数来计算矩阵乘法的运行时间，而非clock函数，因为clock函数计算的是处理器的CPU时间。而本次Project中使用了OpenMP进行多线程加速，使得计算结果不准确。而且gettimeofday函数的精度也更高，可以达到$10^{-6}s$。

C语言的函数在第一次调用时总是速度过慢，是因为第一次调用时代码和数据还不在缓存中，所以应当进行“缓存预热”，即在测速时丢弃第一次运行的结果，只计算后面的运行时间。

普通的矩阵乘法

这次Project中的matmul_plain函数，我使用的是ikj循环的方式通过三重循环来计算矩阵乘法，时间复杂度为$O(n^3)$。

```
Matrix *matmul_plain(Matrix *mat_1, Matrix *mat_2) //使用三重循环计算矩阵乘法
{
    if (mat_1 == NULL || mat_2 == NULL)
        return NULL;
    if (mat_1->data == NULL || mat_2->data == NULL)
        return NULL;
    if (mat_1->col != mat_2->row)
        return NULL;
    Matrix *ret = CreateMatrix(mat_1->row, mat_2->col, NULL);
    for (size_t i = 0; i < mat_1->row; i++)
        for (size_t j = 0; j < mat_1->col; j++)
            for (size_t k = 0; k < mat_2->col; k++)
                ret->data[i * ret->col + k] += mat_1->data[i * mat_1->col + j] * mat_2->da
    return ret;
}
```

这是随着矩阵大小逐渐增加，矩阵乘法的运行时间(每种规模的矩阵计算5次取平均值)

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
时间/s	0.000051	0.10645	4.890845	>600

可以发现，当矩阵规模达到$8K \times 8K$时，传统的矩阵乘法就已经需要至少10分钟才能计算出结果了。

矩阵乘法的提速

之后要进行矩阵乘法的加速，使运算速度更快，效率更高

编译优化

C语言编译器有一些优化编译的选项，合理运用可以提升程序的效率，缩短执行时间。在这次Project中，我开启了O3优化。

这是开启优化后运行时间的对比。(每种规模的矩阵计算5次取平均值)

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
开启优化前/s	0.000051	0.010645	4.890845	>600
开启优化后/s	0.000002	0.000307	0.090641	115.529935

可以发现，开启O3优化后，速度提升了几乎30倍。

使用Strassen算法

Strassen算法是一个用来加速矩阵乘法的算法，其原理为首先将需要计算的矩阵划分成四个小矩阵。

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

然后创建10个矩阵$S_1 - S_{10}$。

$$\begin{aligned} S_1 &= B_{12} - B_{22} \\ S_2 &= A_{11} + A_{12} \\ S_3 &= A_{21} + A_{22} \end{aligned}$$

$$
\begin{aligned}
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

然后再次利用Strassen递归算出7个矩阵$P_1 - P_7$。

$$
\begin{aligned}
P_1 &= A_{11} \times S_1 \\
P_2 &= S_2 \times B_{22} \\
P_3 &= S_3 \times B_{11} \\
P_4 &= A_{22} \times S_4 \\
P_5 &= S_5 \times S_6 \\
P_6 &= S_7 \times S_8 \\
P_7 &= S_9 \times S_{10}
\end{aligned}
$$

最后用$P_1 - P_7$算出C_{11}、C_{12}、C_{21}、C_{22}。

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_1 + P_5 - P_3 - P_7
\end{aligned}
$$

就可以计算出C矩阵了。

在进行Strassen算法时经常需要使用子矩阵，如果每次都将矩阵信息复制的话过于浪费，故选择记录子矩阵左上角元素的坐标，来减少复制元素带来的时间损耗。

设使用Strassen计算矩阵规模为n的矩阵乘法时的理论复杂度为T(n),那么$T(n) = 7T(\frac{n}{2}) + O(n)$,由主定理可以计算出$T(n) = O(n^{log_2 7})$，是比普通的矩阵乘法要快的。但是Strassen算法的常数较大应当在一个合适的时候退出递归，即当矩阵规模缩小到一定程度时，使用普通的ikj算法计算矩阵乘法而非继续递归。

以下是在不同初始矩阵规模和基础矩阵规模下的测试结果(每种规模的矩阵计算5次取平均值)：

矩阵规模	128×128	$1K \times 1K$	$8K \times 8K$
8	0.003092s	0.687389s	时间过长
16	0.000751s	0.154116s	63.247505s
32	0.000442s	0.080888s	37.090216s
64	0.000250s	0.058593s	24.819072s
128	0.000289s	0.059750s	21.829867s
256	无	0.065578s	23.632280s
512	无	0.075435s	27.627451s
1024	无	0.078591s	36.045516s

可以发现，当基础规模过小时，甚至有可能出现比普通ikj算法耗时更快的情况，所以选择合适的基础矩阵规模是很重要的。观察发现当基础矩阵规模为128时退出是一个较优的选择，故最后选择基础矩阵规模为128时退出。

这是优化后的对比结果(每种规模的矩阵计算5次取平均值)：

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
ikj算法/s	0.000051	0.010645	4.890845	>600
O3优化/s	0.000002	0.000307	0.090641	115.529935
Strassen算法/s	0.000003	0.000337	0.069799	27.348119

使用AVX指令集

使用AVX指令集加速运算，AVX指令集可以同时进行256位的计算，即同时进行8个float类型的加减乘法等。因为Strassen中既存在矩阵加减法也存在矩阵乘法，我会先尝试使用AVX加速矩阵加减法。

这是使用AVX优化矩阵加减法后的结果(每种规模的矩阵计算5次取平均值)：

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
ikj算法/s	0.000051	0.010645	4.890845	>600
O3优化/s	0.000002	0.000307	0.090641	115.529935
Strassen算法/s	0.000003	0.000337	0.069799	27.348119
AVX加速加减法/s	0.000009	0.000275	0.067669	23.598873

之后又尝试使用AVX加速了乘法(每种规模的矩阵计算5次取平均值)：

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
ikj算法/s	0.000051	0.010645	4.890845	>600
O3优化/s	0.000002	0.000307	0.090641	115.529935
Strassen算法/s	0.000003	0.000337	0.069799	27.348119
AVX加速加减法/s	0.000009	0.000275	0.067669	23.598873
AVX加速乘法/s	0.000002	0.000271	0.066548	23.754440

可以发现，相比于使用AVX加速加法运算，AVX对于乘法运算点加速并不明显，猜测是因为乘法运算调用了对元素进行复制的函数"_mm256_set1_ps"和先乘后加点，故最后只使用AVX加速了矩阵加减法。

使用OpenMP

OpenMP是一种用于共享内存并行系统的多线程程序设计方案，可以通过并行计算来优化for循环，以达到提升运算速度的效果。可以通过语句"#pragma omp parallel for"来优化for循环

这是使用OpenMP优化后的结果(每种规模的矩阵计算5次取平均值)：

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
ikj算法/s	0.000051	0.010645	4.890845	>600
O3优化/s	0.000002	0.000307	0.090641	115.529935
Strassen算法/s	0.000003	0.000337	0.069799	27.348119

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
AVX加速加减法/s	0.000009	0.000275	0.067669	23.598873
AVX加速乘法/s	0.000002	0.000271	0.066548	22.490622
OpenMP加速/s	0.000100	0.000488	0.049943	12.461064

在数据范围小的时候开启线程和合并线程的时间消耗过大，但随着数据范围增大，多线程的优势逐渐显现，最后几乎可以优化一倍。

及时释放内存

在Strassen函数中我会申请很多内存，我会先为$S_1 - S_{10}$和$P_1 - P_7$都申请内存，之后在Strassen结束前统一释放，但是其实可以通过调整计算的顺序，更加及时的释放内存，减少计算时的负担

这是及时释放内存后的结果(每种规模的矩阵计算5次取平均值)：

矩阵规模	16×16	128×128	$1K \times 1K$	$8K \times 8K$
ikj算法/s	0.000051	0.010645	4.890845	>600
O3优化/s	0.000002	0.000307	0.090641	115.529935
Strassen算法/s	0.000003	0.000337	0.069799	27.348119
AVX加速加减法/s	0.000009	0.000275	0.067669	23.598873
AVX加速乘法/s	0.000002	0.000271	0.066548	22.490622
OpenMP加速/s	0.000100	0.000488	0.049943	12.461064
及时释放内存/s	0.000099	0.000359	0.028511	9.989306

将部分变量存储在寄存器中

可以使用register关键字，请求编译器将变量储存在寄存器中，这样可以更快的读取变量。

但是我在使用register关键字后计算时间并没有优化，后来发现原来O2优化会进行-fforce-mem优化，即在计算之前，强制将内存数据复制到寄存器后执行，而O3优化会调用O2优化，所以使用register关键字进行优化在O3优化下是一种画蛇添足的行为。

使用内联函数

函数被调用会开辟一个函数栈，不仅占空间而且会影响耗时，对于并不复杂的函数，可以使用inline关键词将函数设为内联函数，在编译时，编译器会对内联函数进行内联扩展，起到优化性能的作用。在这个Project中，除了Straeen函数为递归函数，无法设为内联函数外，其他都可以通过设为内联函数来加速运算。

我在添加内联函数后发现计算速度并没有提升，发现原来O3优化会进行-finline-functions优化，即将简单的函数内联到被吊用函数中，所以已经默认进行了函数内联，我的行为又称为了一种画蛇添足，不禁让人感叹编译器的强大。

矩阵乘法的正确性

我写了一个similar函数，用来对比两个矩阵对应元素之间的差，并返回误差占原数的比例的最大值。

```
double similar(Matrix *mat_1, Matrix *mat_2) //比较两个矩阵是否相似
{
    if (mat_1 == NULL || mat_2 == NULL)
        return -1;
    if (mat_1->col != mat_2->col || mat_1->row != mat_2->row)
        return -1;
    double differ = 0;
    for (size_t i = 0; i < mat_1->col * mat_2->row; i++)
        differ = fmax(differ, fabs(mat_1->data[i] - mat_2->data[i]) / fmin(mat_1->data[i],
    return differ;
}
```

矩阵中数据的类型

因为担心单精度浮点数在计算过程中误差过大，我的程序可以通过更改data_Type对应的类型将矩阵中的数据类型改为双精度浮点数，并进行计算比较。

Part 2 - Code

CMakeLists.txt

```
cmake_minimum_required (VERSION 3.2)

project(Project4)

SET(CMAKE_C_COMPILER /usr/local/bin/gcc-12)
SET(CMAKE_CXX_COMPILER /usr/local/bin/g++-12)

find_package(OpenMP)
if(NOT OpenMP_FOUND)
    message("OpenMP not found.\nTry to set the flags")
    SET(OpenMP_C_FLAGS -I/usr/local/opt/libomp/include)
    SET(OpenMP_C_LIB_NAMES libomp)
    SET(OpenMP_CXX_FLAGS -I/usr/local/opt/libomp/include)
    SET(OpenMP_CXX_LIB_NAMES libomp)
    SET(OpenMP_libomp_LIBRARY /usr/local/opt/libomp/lib/libomp.dylib)
endif()

SET(BLAS_C_FLAGS -I/usr/local/opt/openblas/include)
SET(BLAS_C_LIB_NAMES openblas)
SET(BLAS_CXX_FLAGS -I/usr/local/opt/openblas/include)
SET(BLAS_CXX_LIB_NAMES openblas)
SET(BLAS_openblas_LIBRARY /usr/local/opt/openblas/lib/libopenblas.dylib)

SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR})
add_compile_options(-Wall)
add_compile_options(-fopenmp)
add_compile_options(-lopenblas)
add_compile_options(-O3)
add_compile_options(-mavx)
add_compile_options(-mfma)
add_executable(test main.c Matrix.c)

find_package(OpenMP)
if(OpenMP_C_FOUND)
    message("OpenMP found.")
    target_link_libraries(test PUBLIC OpenMP::OpenMP_C)
endif()
message(${BLAS_C_FLAGS})
find_package(BLAS REQUIRED)
if(BLAS_FOUND)
    message("OpenBLAS found.")
    target_include_directories(test PUBLIC /usr/local/opt/openblas/include)
    target_link_libraries(test PUBLIC ${BLAS_openblas_LIBRARY})
endif(BLAS_FOUND)
```

Matrix.h

```h
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <immintrin.h>
#include <math.h>
#include <omp.h>

#define _OMP_THREAD_ 4

#ifndef deleteMat
#define deleteMat(Mat) (deleteMatrix(Mat), Mat = NULL)
#endif

typedef float data_Type;

typedef struct Matrix
{
    size_t row;
    size_t col;
    data_Type *data;
} Matrix;

Matrix *CreateMatrix(size_t, size_t, data_Type *); //创建一个_row行_col列的矩阵，并更新其中元素

void deleteMatrix(Matrix *); //删除矩阵,释放空间

Matrix *readFile(FILE *); //从文件读入矩阵

Matrix *matmul_plain(Matrix *, Matrix *); //使用三重循环计算矩阵乘法

void plus_for_Strassen(Matrix *, size_t, Matrix *, size_t, Matrix *, size_t, size_t); // S

void minus_for_Strassen(Matrix *, size_t, Matrix *, size_t, Matrix *, size_t, size_t); //

Matrix *Strassen(Matrix *, size_t, Matrix *, size_t, size_t); // Strassen算法，返回A*B的结果

Matrix *matmul_improved(Matrix *, Matrix *); //优化矩阵乘法

void printMatrix(Matrix *); //输出矩阵

double similar(Matrix *, Matrix *); //比较两个矩阵是否相似
```

Matrix.c

```c
#include "Matrix.h"

Matrix *CreateMatrix(size_t _row, size_t _col, data_Type *_data) //创建一个_row行_col列的矩阵
{
    Matrix *ret = malloc(sizeof(Matrix));
    ret->row = _row;
    ret->col = _col;
    if (_data == NULL)
        ret->data = aligned_alloc(256, _row * _col * sizeof(data_Type));
    else
        ret->data = _data;
    return ret;
}

void deleteMatrix(Matrix *mat) //删除矩阵,释放空间
{
    if (mat == NULL)
        return;
    if (mat->data != NULL)
        free(mat->data);
    free(mat);
    return;
}

Matrix *readFile(FILE *file) //从文件读入矩阵
{
    Matrix *mat = malloc(sizeof(Matrix));
    fscanf(file, "%zu %zu", &mat->row, &mat->col);
    mat->data = aligned_alloc(256, sizeof(data_Type) * mat->row * mat->col);
    if (sizeof(data_Type) == 8)
        for (size_t i = 0; i < mat->row * mat->col; i++)
            fscanf(file, "%lf", &mat->data[i]);
    if (sizeof(data_Type) == 4)
        for (size_t i = 0; i < mat->row * mat->col; i++)
            fscanf(file, "%f", &mat->data[i]);
    return mat;
}

Matrix *matmul_plain(Matrix *mat_1, Matrix *mat_2) //使用三重循环计算矩阵乘法
{
    if (mat_1 == NULL || mat_2 == NULL)
        return NULL;
    if (mat_1->data == NULL || mat_2->data == NULL)
        return NULL;
    if (mat_1->col != mat_2->row)
        return NULL;
```

```
    Matrix *ret = CreateMatrix(mat_1->row, mat_2->col, NULL);
    for (size_t i = 0; i < mat_1->row; i++)
        for (size_t j = 0; j < mat_1->col; j++)
            for (size_t k = 0; k < mat_2->col; k++)
                ret->data[i * ret->col + k] += mat_1->data[i * mat_1->col + j] * mat_2->da
    return ret;
}

void plus_for_Strassen(Matrix *mat_1, size_t startIndex_1, Matrix *mat_2, size_t startInde
{
    if (sizeof(data_Type) == 8)
    {
#pragma omp parallel for num_threads(_OMP_THREAD_)
        for (size_t i = 0; i < size; i++)
        {
            for (size_t j = 0; j < size; j += 4)
            {
                _mm256_store_pd((mat_3->data + startIndex_3 + i * mat_3->col + j), _mm256_
            }
        }
    }
    if (sizeof(data_Type) == 4)
    {
#pragma omp parallel for num_threads(_OMP_THREAD_)
        for (size_t i = 0; i < size; i++)
        {
            for (size_t j = 0; j < size; j += 8)
            {
                _mm256_store_ps((mat_3->data + startIndex_3 + i * mat_3->col + j), _mm256_
            }
        }
    }
}

void minus_for_Strassen(Matrix *mat_1, size_t startIndex_1, Matrix *mat_2, size_t startInd
{

    if (sizeof(data_Type) == 8)
    {
#pragma omp parallel for num_threads(_OMP_THREAD_)
        for (size_t i = 0; i < size; i++)
        {
            for (size_t j = 0; j < size; j += 4)
            {
                _mm256_store_pd((mat_3->data + startIndex_3 + i * mat_3->col + j), _mm256_
            }
```

```
        }
    }
    if (sizeof(data_Type) == 4)
    {
#pragma omp parallel for num_threads(_OMP_THREAD_)
        for (size_t i = 0; i < size; i++)
        {
            for (size_t j = 0; j < size; j += 8)
            {
                _mm256_store_ps((mat_3->data + startIndex_3 + i * mat_3->col + j), _mm256_
            }
        }
    }
}

Matrix *Strassen(Matrix *A, size_t index_A, Matrix *B, size_t index_B, size_t size) // Str
{
    if (size <= 128)
    {
        Matrix *ret = CreateMatrix(size, size, NULL);
        memset(ret->data, 0, ret->col * ret->row * sizeof(data_Type));
#pragma omp parallel for num_threads(_OMP_THREAD_)
        for (size_t i = 0; i < size; i++)
        {
            for (size_t j = 0; j < size; j++)
            {
                for (size_t k = 0; k < size; k++)
                {
                    ret->data[i * ret->col + k] += A->data[index_A + i * A->col + j] * B->
                }
            }
        }
        return ret;
    }
    size_t new_size = (size >> 1);
    size_t a11 = index_A, a12 = a11 + new_size, a21 = a11 + new_size * A->col, a22 = a21 +
    size_t b11 = index_B, b12 = b11 + new_size, b21 = b11 + new_size * B->col, b22 = b21 +

    Matrix *S1 = CreateMatrix(new_size, new_size, NULL);
    minus_for_Strassen(B, b12, B, b22, S1, 0, new_size);
    Matrix *P1 = Strassen(A, a11, S1, 0, new_size);
    deleteMat(S1);

    Matrix *S2 = CreateMatrix(new_size, new_size, NULL);
    plus_for_Strassen(A, a11, A, a12, S2, 0, new_size);
    Matrix *P2 = Strassen(S2, 0, B, b22, new_size);
```

```
    deleteMat(S2);

    Matrix *S3 = CreateMatrix(new_size, new_size, NULL);
    plus_for_Strassen(A, a21, A, a22, S3, 0, new_size);
    Matrix *P3 = Strassen(S3, 0, B, b11, new_size);
    deleteMat(S3);

    Matrix *S4 = CreateMatrix(new_size, new_size, NULL);
    minus_for_Strassen(B, b21, B, b11, S4, 0, new_size);
    Matrix *P4 = Strassen(A, a22, S4, 0, new_size);
    deleteMat(S4);

    Matrix *S5 = CreateMatrix(new_size, new_size, NULL);
    plus_for_Strassen(A, a11, A, a22, S5, 0, new_size);
    Matrix *S6 = CreateMatrix(new_size, new_size, NULL);
    plus_for_Strassen(B, b11, B, b22, S6, 0, new_size);
    Matrix *P5 = Strassen(S5, 0, S6, 0, new_size);
    deleteMat(S5);
    deleteMat(S6);

    Matrix *S7 = CreateMatrix(new_size, new_size, NULL);
    minus_for_Strassen(A, a12, A, a22, S7, 0, new_size);
    Matrix *S8 = CreateMatrix(new_size, new_size, NULL);
    plus_for_Strassen(B, b21, B, b22, S8, 0, new_size);
    Matrix *P6 = Strassen(S7, 0, S8, 0, new_size);
    deleteMat(S7);
    deleteMat(S8);

    Matrix *S9 = CreateMatrix(new_size, new_size, NULL);
    minus_for_Strassen(A, a11, A, a21, S9, 0, new_size);
    Matrix *S10 = CreateMatrix(new_size, new_size, NULL);
    plus_for_Strassen(B, b11, B, b12, S10, 0, new_size);
    Matrix *P7 = Strassen(S9, 0, S10, 0, new_size);
    deleteMat(S9);
    deleteMat(S10);

    Matrix *C = CreateMatrix(size, size, NULL);
    memset(C->data, 0, size * size * sizeof(data_Type));
    size_t c11 = 0, c12 = c11 + new_size, c21 = c11 + new_size * size, c22 = c21 + new_siz
    plus_for_Strassen(P5, 0, P4, 0, C, c11, new_size);
    minus_for_Strassen(C, c11, P2, 0, C, c11, new_size);
    plus_for_Strassen(C, c11, P6, 0, C, c11, new_size);
    deleteMat(P6);
    plus_for_Strassen(P1, 0, P2, 0, C, c12, new_size);
    deleteMat(P2);
    plus_for_Strassen(P3, 0, P4, 0, C, c21, new_size);
```

```
    deleteMat(P4);
    plus_for_Strassen(P5, 0, P1, 0, C, c22, new_size);
    deleteMat(P5);
    deleteMat(P1);
    minus_for_Strassen(C, c22, P3, 0, C, c22, new_size);
    deleteMat(P3);
    minus_for_Strassen(C, c22, P7, 0, C, c22, new_size);
    deleteMat(P7);
    return C;
}

Matrix *matmul_improved(Matrix *mat_1, Matrix *mat_2) //优化矩阵乘法
{
    if (mat_1 == NULL || mat_2 == NULL)
        return NULL;
    if (mat_1->data == NULL || mat_2->data == NULL)
        return NULL;
    if (mat_1->col != mat_2->row)
        return NULL;
    if (mat_1->col != mat_1->row || mat_2->col != mat_2->row)
        return NULL;
    return Strassen(mat_1, 0, mat_2, 0, mat_1->col);
}

void printMatrix(Matrix *mat) //输出矩阵
{
    if (mat == NULL)
        return;
    if (mat->data == NULL)
        return;
    for (size_t i = 0; i < mat->row * mat->col; i++)
    {
        if (i % mat->col == 0)
            printf("\n");
        printf("%.1f ", mat->data[i]);
    }
    printf("\n");
}

double similar(Matrix *mat_1, Matrix *mat_2) //比较两个矩阵是否相似
{
    if (mat_1 == NULL || mat_2 == NULL)
        return -1;
    if (mat_1->col != mat_2->col || mat_1->row != mat_2->row)
        return -1;
    double differ = 0;
```

```
    for (size_t i = 0; i < mat_1->col * mat_2->row; i++)
        differ = fmax(differ, fabs(mat_1->data[i] - mat_2->data[i]) / fmin(mat_1->data[i],
    return differ;
}
```

Part 3 - Result & Verification

我会比较在不同规模下，普通矩阵乘法，优化后的矩阵乘法和OpenBLAS矩阵乘法的速度差距（全部开启了O3优化），计算5次取平均值，这是数据类型为单精度浮点数时的结果。

	16×16	64×64	256×256	$1K \times 1K$	$4K \times 4K$	$8K \times 8K$	$16K \times 16K$
matmul_plain/s	0.0000018	0.0000724	0.0024406	0.1121380	12.9422574	111.9896614	$\times$
matmul_improved/s	0.0001132	0.0003374	0.0015878	0.0445666	1.3719404	10.2487874	79.3426786
OpenBLAS/s	0.0000022	0.0000284	0.0004496	0.0104096	0.4378020	3.5735278	26.9740332

在所有比较中，Similar函数的返回值都在万分之三以下，说明准确率基本正确，而即使我应用了所有能够想到的优化方法，仍然是OpenBLAS速度的三倍之多，仍有许多不足。

之后我将数据类型改为双精度浮点数并进行比较。

	16×16	64×64	256×256	$1K \times 1K$	$4K \times 4K$	$8K \times 8K$
matmul_plain/s	0.0000044	0.0001414	0.0043354	0.3410770	29.0534780	249.8883548
matmul_improved/s	0.0000052	0.0002240	0.0025376	0.0685068	3.1187230	19.7339492
OpenBLAS/s	0.0000034	0.0000512	0.0005292	0.0216948	1.0577760	7.1512328

可以发现，双精度浮点数的速度约为单精度浮点数的两倍，但是准确率明显升高，误差在1e-6以下。

Part 4 - Difficulties & Solutions

访存优化

连续的内存访问速度会更快，所以我们需要尽量减少内存访问的跳转，具体可以通过交换for循环枚举的顺序得到最后的最优方案，经过枚举和分析发现，对于 $A \times B = C$， 最快的方法是先枚举A的每一行，之后枚举A的列，最后枚举B的列，所以最后选择这种方法作为矩阵乘法的朴素方法。

OpenMP的使用

因为电脑是MAC，编译器为Clang，无法适配OpenMP，所以后来下载了gcc编译器进行编译，并配置了OpenMP的路径。

分配内存

使用aligned_alloc而非malloc分配内存，这样分配的内存能首地址是需求值的倍数，可以使AVX的访问速度更快，加快运算速度。

Strassen算法

Strassen算法会进行大量递归，在递归过程中会调用大量空间，其中还会使用大矩阵的一些子矩阵，如果将数据再多复制一份的话会损耗很大一块空间，还会影响运算速度。所以最后我使用子矩阵的左上角元素的地址和矩阵大小来表示子矩阵，这样可以加快运算速度。