# CS205 C/ C++ Programming Project 3: Matrix Structure and Multiplication in C

**Name:** 眭和（SUI He)

**SID:** 12012929



# Part 1 - Requirement

1. The programming language can only be C, not C++. Please save your source code into `*.c` files, and compile them using a C compiler such as gcc (not g++). Try to use Makefile or CMake to manage your source code.
2. Design a `struct` for matrices, and the `struct` should contain the data of a matrix, the number of columns, the number of rows, etc.

3. Implement some functions to
   - create a matrix
   - delete a matrix
   - copy a matrix (copy the data from a matrix to another)
   - multiply two matrices
   - some other functions needed
4. When you run the program as follows, it will output the result into an output file such as out32.txt . The data files are the same with those in Project 2.
   `$./matmul mat-A-32.txt mat-B-32.txt out32.txt`
   `$./matmul mat-A-256.txt mat-B-256.txt out256.txt`
   `$./matmul mat-A-2048.txt mat-B-2048.txt out2048.txt`
5. Try to improve the speed of matrix multiplication. Introduce how you improve it in the report. You should explicitly introduce the differences between this one and Project 2.
6. Compare the speed of your implementation with OpenBLAS (https://www.openblas.net/).

# Part 2 - Analysis

## 存储矩阵的`struct`

在本次project中，要求设计一个`struct`来存储矩阵信息，包括矩阵的行数、列数以及矩阵中各个元素的信息。又因为`C`的`struct`中不能放方法，因此在我的矩阵结构体中，只存放了三个变量分别是行数，列数和一个指向元素信息的`float *`指针。指针指向一个一维的`float`数组存储数据，按照行的顺序存储，在第一行的数据存储完成后紧接着存储第二行的数据，依次类推。

```
struct Matrix
{
    int row;
    int column;
    float *data;
};
```

## 文件的读写

本次文件读入和Project 2 有一定的区别。在project 2 中矩阵全为方形的，因此我采取了较为简便的读取方法，直接读取的文件名中的矩阵规模，因此可以直接确定数组大小直接进行读入。但在本次project中，矩阵的行数和列数是不确定的，需要根据文件内容进行确定。因此我采取了对文件扫描两遍的方法，第一遍先确定矩阵的列数和行数，然后创建好数组，再进行第二遍扫描读入数据。这样虽然花费了更多的时间读入数据，但也能更方便地对文件的错误情况进行处理：

我首先依次扫描每个元素，并且检查是否有换行符，检测到换行符即可确定该矩阵列数，然后继续计算文件中总共有多少个数，判断总个数能否整除列数，如果能够整除，则说明矩阵的元素在数量上是正确的，并可以根据总数除以列数得到行数。如果不能够整除，则说明文件是有问题的，可以在计算前就停止程序，有效防止问题产生。然后再将文件指针重置到开头，读入数据。

文件的写入还是与project2相似，唯一的区别是要判断什么时候换行，即输出的总数能整除列数时，需要输出`\n`。

# 矩阵部分函数的实现

## 创建一个矩阵

根据上文矩阵结构体内的数据，可以得知创建一个矩阵，需要矩阵的行数、列数、及存储矩阵的各元素值的数组的地址。因此可以使用`malloc`为矩阵分配一块空间，并依次对各个变量进行赋值。

```
struct Matrix *creatMatrix(const int row, const int column, float *const
data)
{
    struct Matrix *matrix = (struct Matrix *)malloc(sizeof(struct Matrix));
    matrix->row = row;
    matrix->column = column;
    matrix->data = data;
    return matrix;
};
```

## 删除一个矩阵

若一个矩阵不再需要，可以对其进行删除操作，释放占用的空间。若对于规模非常大的矩阵不及时清理，则会造成空间浪费，还有可能使程序变得非常慢。由于矩阵的内部还有一个指向数据数组的指针，而且该数组占用了绝大部分空间，所以需要先用`free()`对数组进行空间释放，再对结构体进行空间释放才能释放完全。

```
void deleteMatrix(struct Matrix *matrix)
{
    if (matrix != NULL)
    {
        if (matrix->data != NULL)
            free(matrix->data);
        free(matrix);
    }
}
```

## 复制一个矩阵

有时需要对一个数组进行复制操作，但在复制的过程不能仅仅将一个数组的指针复制到另一个矩阵的结构体中，否则修改其中一个矩阵的值会使另一个矩阵的值一起改变，因此，需要使用深复制，即遍历每一个位置进行复制。同时，`dest`可能是一个已经经过初始化的矩阵，规模可能与被复制的矩阵不同，所以要先对数组进行释放再创建一个新的。

```
void copyMatrix(const struct Matrix *const src, struct Matrix *const dest)
{
    dest->row = src->row;
    dest->column = src->column;
    free(dest->data);
    dest->data = (float *)malloc(sizeof(float) * dest->row * dest->column);
    for (int counter = 0; counter < dest->row * dest->column; ++counter)
        dest->data[counter] = src->data[counter];
}
```

### 矩阵的转置

转置也是对矩阵的常用操作。这个操作需要先生成一个新的数组，对数组进行赋值。因为数组是通过一维数组存储，因此需要计算好新数组内每个位置对应了原数组的哪个位置，需要对矩阵转置过程有清楚的认识。最后交换矩阵行和列的值，即可完成矩阵转置的过程。

```
void transpose(struct Matrix *const matrix)
{
    float *data = malloc(sizeof(float) * matrix->column * matrix->row);
    int index = 0;
    for (int i = 0; i < matrix->column * matrix->row; ++i)
    {
        if (i % matrix->row == 0)
            index = i / matrix->row;
        else
            index += matrix->column;
        data[i] = matrix->data[index];
    }
    free(matrix->data);
    matrix->data = data;
    int temp = matrix->row;
    matrix->row = matrix->column;
    matrix->column = temp;
}
```

## 矩阵乘法的计时

在project2 中计时采用的是`clock()`函数计时，而在本次我选择了`sys/time.h`库中的结构体`timeval`以及`gettimeofday()`方法获取时间。主要原因有以下两点：

1. `clock()`只能精确到1ms，在计算1s内的时间时误差可能相对较大，而`gettimeofday()`可以精确到1微秒。在本次project中`OpenBLAS`矩阵运算速度较快，需要更精确的计时方法。
2. `clock()`计时测量的是处理器的CPU时间，而不是实际的时间或者时钟，而`OpenBLAS`中的矩阵运算包含了大量的优化，包括多线程，导致`clock()`计时测得的时间不准。这一部分也会在下一部分进行详细阐释。

## 矩阵乘法的提速

在project2中，采用的是二维数组存储数据，同时`ijk`的朴素计算方法以及对第二个矩阵进行转置后运算减小内存访问开销，以及最后尝试多种循环方式，发现`ikj`方式循环在内存访问上开销最少，能最大程度上对数组进行连续访问，花费在内存访问上的时间更少。因此提速的关键很大一部分就是减少内存访问开销。

### 使用一维数组存储数据

考虑到一维数组是线性存储数据的，理论上在内存访问时间上应该比二维数组要快。因此，在本次project上尝试用一维数组存储数据进行计算。

在该方法中首先需要判断两个矩阵结构体是否为空或第一个矩阵的列数是否等于第二个矩阵的行数，即两个矩阵不能相乘，即返回`NULL`并返回提示信息。

然后，根据`ikj`的循环方式对矩阵进行计算，原理在之前报告已有详细阐释，在这不做过多赘述。但值得一提的是对于在循环中重复的值，需要可以提到循环外面防止多次计算。比如在k循环内的A矩阵中的值，在下一层循环中每次都需要利用，即可放在循环外。此外，数组的下标因为是一维数组存储，每次都需要计算数组下标，其中涉及大量乘法。但仔细观察可以发现，每次计算的都是数组中线性的一段，只需要找到这段的头下标，那么在下一层循环中只需要每次自增一次而不必都做一次大数乘法再加上一个j的值(`(i - 1) * result->column + j - 2`)。因为乘法的运算消耗肯定是比加法要多的，所以尽可能地减少乘法运算也应能节省一定时间。

```
struct Matrix *multiply(const struct Matrix *const a, const struct Matrix
*const b)
{
    if (a == NULL || b == NULL || a->column != b->row)
    {
        printf("Can not multiply these two matrixes!\n");
        return NULL;
    }
    struct Matrix *result = creatMatrix(a->row, b->column, (float
*)malloc(sizeof(float) * a->row * b->column));
    for (int i = 0; i < result->column * result->row; ++i)
        result->data[i] = 0;
    for (int i = 1; i <= result->row; ++i)
    {
        for (int k = 1; k <= a->column; ++k)
        {
            float value = a->data[(i - 1) * a->column + k - 1];
            int index_b = (k - 1) * b->column - 1;
            int index_result = (i - 1) * result->column - 1;
            for (int j = 1; j <= result->column; ++j)
                result->data[++index_result] += value * b->data[++index_b];
        }
    }
    return result;
}
```

以下是计算计算矩阵运行时间的结果(每种规模的矩阵测试5遍取平均值)

| 矩阵规模 | 32 | 256 | 2048 | 4096 |
| --- | --- | --- | --- | --- |
| 时间/ms | 0 | 59 | 28978 | 238181 |

## 开启编译优化

C语言编译器拥有一些用于优化编译的选项，合理地应用这些编译选项可以有效提升程序执行效率，缩短执行时间。编译优化可改变代码逻辑，但当工程较为复杂时可能导致逻辑改变错误导致结果异常。但本程序经核对并未产生错误。

程序运行时间对比如下（时间单位为ms):

| 矩阵规模 | 32 | 256 | 2048 | 4096 |
| --- | --- | --- | --- | --- |
| 开启前 | 0 | 59 | 28978 | 238181 |
| 开启后 | 0 | 3 | 2825 | 23911 |

对程序优化明显，在大规模矩阵的情况下速度基本是原来的十分之一。

进一步提速以及OpenBLAS的对比将在下一部分展示

# Part 3 - Difficulties & Solutions

## `ikj` 写法时的一个错误

首先，先列出一个使用上面方法计算矩阵乘法的一个错误。在初次写完这个函数后，运行发现输出的结果中有时会产生某些元素不对的情况（并且集中在输出的第一行）。而且这种情况是重复执行多次才会出现一次这样的情况，即大部分情况结果是正确的，偶尔会产生错误结果。这令我很费解，并且找了很久的bug。

后来，我仔细看了函数的执行部分，在计算每个元素值的时候，使用的运算符号是`+=`，我的本意是矩阵元素一开始全为0，然后根据矩阵乘法定义不断加上对应的乘积。然而这个的基础是建立在数组元素一开始全为0的情况下。而我的数组的存储区域是通过`malloc`申请的内存，这个过程仅仅是将这片内存划给了数组，并没有改变原先地址上的值。因此，该内存上有些位置初始就为一个随机的数，导致后续的运算出现错误。因此，该问题只需在计算前对数组进行循环归0即可。这也提醒我下次如果申请了一片内存并且不是对元素进行覆盖式地赋值，一定要先将数组初始化！

## 尝试采用Strassen算法进行提速

### Strassen算法介绍

1. 将$A, B, C$分别分别为4个子矩阵（花费时间$\Theta(1)$）。

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

2. 如下创建10个$\frac{n}{2} \times \frac{n}{2}$的矩阵$S_1, S_2, S_3, \ldots, S_10$(花费时间$\Theta(n^2)$)。

$$\begin{aligned} S_1 &= B_{12} - B_{22} \\ S_2 &= A_{11} + A_{12} \\ S_3 &= A_{21} + A_{22} \\ S_4 &= B_{21} - B_{11} \\ S_5 &= A_{11} + A_{22} \\ S_6 &= B_{11} + B_{22} \\ S_7 &= A_{12} - A_{22} \\ S_8 &= B_{21} + B_{22} \\ S_9 &= A_{11} - A_{21} \\ S_{10} &= B_{11} + B_{12} \end{aligned}$$

3. 递归地计算7个矩阵积$P_1, P_2, \ldots, P_7$，每个矩阵$P_i$都是$\frac{n}{2} \times \frac{n}{2}$的。

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

注意，上述公式中只有中间一列需要计算。

4. 通过$P_i$计算$C_{11}, C_{12}, C_{21}, C_{22}$，花费时间$\Theta(n^2)$。

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

综合可得如下递归式：

$$
T(n) = \begin{cases} \Theta(1) & \text{if n} = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if n} > 1 \end{cases}
$$

进而求出时间复杂度为：$T(n) = \Theta(n^{\log_2 7})$，时间复杂度比第一种算法$T(n) = \Theta(n^3)$小。

## Strassen算法实现

Strassen算法内有大量的创建矩阵以及分割矩阵以及合并矩阵的过程。如果每次都要对切割的矩阵进行复制或者对子矩阵合并成一个大矩阵会花费$O(n^2)$的时间。浪费大量时间在矩阵的复制上肯定不合适。

于是我决定采用**不复制数组**的方式，每次进行矩阵切割的时候，我并不创建新的矩阵，而是采用原先的矩阵，只是**记录子矩阵左上角元素的下标。**之后在进行矩阵的加减法时，给矩阵加法的函数内传入的参数有原先的矩阵，子矩阵的`start_index`，以及子矩阵的规模。拥有这些参数，就已经足以在给定的原有矩阵下计算子矩阵的加减法。相较复制的方法能够省去非常多的时间。

使用Strassen算法的关键是需要注意要仔细应用递归，而且因为在计算过程中不断递归新建了非常多的矩阵，需要进行内存管理，稍后也会测试不进行内存管理的后果。

## Strassen算法递归的基本情况的选定

因为Strassen算法采用的是递归的算法，而递归必须选择一个适当的情况作为离开递归的出口，称为基本情况。为了确定递归的最好的基本情况（即矩阵规模下降到多少的时候采用常规的算法），进行了一些测试（测试开启编译优化）。以下是测试结果：

| 基本情况矩阵规模 | 256 | 2048 | 4096 |
| --- | --- | --- | --- |
| 1 | 607ms | - | - |
| 2 | 134ms | 43382ms | - |
| 4 | 47ms | 15034ms | - |
| 8 | 24ms | 7891ms | 55522ms |
| 16 | 14ms | 4931ms | 34319ms |
| 32 | 7ms | 2507ms | 17812ms |
| 64 | 4ms | 1763ms | 11868ms |
| 128 | 4ms | 1686ms | 11827ms |
| 256 | - | 1475ms | 10075ms |
| 512 | - | 1372ms | 9917ms |

可以很容易地发现，在递归基本情况为很小的时候，比如最小的子矩阵为`1*1`的矩阵，即使用该算法递归到`1*1`才使用常规算法计算，这个时候时间耗费非常的大，甚至在算`256*256`规模的矩阵是常规算法的100多倍。究其原因，是因为Strassen算法里面创建了非常多的矩阵，如果一直递归到规模为1的矩阵才进行直接计算，运算效率还不如常规算法，因为常规算法几步就算完的过程，strassen需要额外创建十几个新的数组，而创建数组非常耗时间，而在矩阵规模非常大的情况下，如果递归到非常深，会有非常多的这样的浪费，致使最终运算速度比常规算法还要慢非常多。从数据结果我们也能看出来随着基本情况矩阵规模的增大，运算速度逐渐赶上并且超越了常规算法，也印证了其时间复杂度优于常规算法。

事实上，每个规模的矩阵都有最适合其计算的基本情况，基本情况定的太小会导致计算时间反而过低，基本情况太大会导致strassen最后趋近于常规算法。具体计算时还是需要根据实际情况选择合适的基本情况。在本次project中，由于时间有限，无法依次测出最合适的情况，对于这三种大小的矩阵，先让基本情况取128。**在之后的测试也都是以128为基本情况进行运算的。**

## Strassen算法的内存管理

由于在Strassen算法中每次递归都会新建大量的矩阵，在不需要用之后，应当及时对矩阵进行释放。而如果不及时对其进行释放，是否会对其速度产生影响呢。在接下来的测试中，我将分成两边，一边的测试均为及时进行了内存回收，另一边从到到尾不进行回收。

| 矩阵规模 | 256 | 2048 | 4096 |
| --- | --- | --- | --- |
| 释放内存 | 4ms | 1706ms | 11885ms |
| 不释放内存 | 4ms | 2081ms | 14566ms |

从数据可以很明显地发现如果不及时释放内存，计算速度变慢了。可能是大量内存占用导致了内存访问变慢。因此，对于不需要的矩阵，一定需要及时释放！

## Strassen算法实现代码

> 该方法中计算子矩阵加法和减法的函数详见下一部分

```
struct Matrix *strassen(struct Matrix *a, int a_start_index, struct Matrix *b, int b_start_index, int size)
{
    struct Matrix *result = creatMatrix(size, size, malloc(sizeof(float) * size * size));
    if (size == 128)
    {
        for (int i = 0; i < 16384; ++i)
            result->data[i] = 0;
        for (int i = 1; i <= 128; ++i)
        {
            for (int k = 1; k <= 128; ++k)
            {
                float value = a->data[a_start_index + (i - 1) * a->column + k - 1];
                int index_b = b_start_index + (k - 1) * b->column - 1;
                int index_result = (i - 1) * result->column - 1;
                for (int j = 1; j <= 128; ++j)
                    result->data[++index_result] += value * b->data[++index_b];
            }
        }
        return result;
    }

    int subSize = size / 2;
    int subNum = subSize * subSize;
    struct Matrix *S1 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S2 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S3 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S4 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S5 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S6 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S7 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S8 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S9 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    struct Matrix *S10 = creatMatrix(subSize, subSize, malloc(sizeof(float) * subNum));
    int a11 = findIndex(a, size, a_start_index, 1);
    int a12 = findIndex(a, size, a_start_index, 2);
    int a21 = findIndex(a, size, a_start_index, 3);
    int a22 = findIndex(a, size, a_start_index, 4);
    int b11 = findIndex(b, size, b_start_index, 1);
    int b12 = findIndex(b, size, b_start_index, 2);
```

```
    int b21 = findIndex(b, size, b_start_index, 3);
    int b22 = findIndex(b, size, b_start_index, 4);
    minus(subSize, b, b12, b, b22, S1, 0);
    add(subSize, a, a11, a, a12, S2, 0);
    add(subSize, a, a21, a, a22, S3, 0);
    minus(subSize, b, b21, b, b11, S4, 0);
    add(subSize, a, a11, a, a22, S5, 0);
    add(subSize, b, b11, b, b22, S6, 0);
    minus(subSize, a, a12, a, a22, S7, 0);
    add(subSize, b, b21, b, b22, S8, 0);
    minus(subSize, a, a11, a, a21, S9, 0);
    add(subSize, b, b11, b, b12, S10, 0);

    struct Matrix *P1 = strassen(a, a11, S1, 0, subSize);
    deleteMatrix(S1);
    struct Matrix *P2 = strassen(S2, 0, b, b22, subSize);
    deleteMatrix(S2);
    struct Matrix *P3 = strassen(S3, 0, b, b11, subSize);
    deleteMatrix(S3);
    struct Matrix *P4 = strassen(a, a22, S4, 0, subSize);
    deleteMatrix(S4);
    struct Matrix *P5 = strassen(S5, 0, S6, 0, subSize);
    deleteMatrix(S5);
    deleteMatrix(S6);
    struct Matrix *P6 = strassen(S7, 0, S8, 0, subSize);
    deleteMatrix(S7);
    deleteMatrix(S8);
    struct Matrix *P7 = strassen(S9, 0, S10, 0, subSize);
    deleteMatrix(S9);
    deleteMatrix(S10);
    multiOperations(subSize, P5, 0, P4, 0, P2, 0, P6, 0, result,
findIndex(result, size, 0, 1), '+');
    deleteMatrix(P6);
    add(subSize, P1, 0, P2, 0, result, findIndex(result, size, 0, 2));
    deleteMatrix(P2);
    add(subSize, P3, 0, P4, 0, result, findIndex(result, size, 0, 3));
    deleteMatrix(P4);
    multiOperations(subSize, P5, 0, P1, 0, P3, 0, P7, 0, result,
findIndex(result, size, 0, 4), '-');
    deleteMatrix(P1);
    deleteMatrix(P3);
    deleteMatrix(P5);
    deleteMatrix(P7);
    return result;
}
```

## 与OpenBLAS的比较

本次矩阵乘法用到的OpenBLAS的函数为`cblas`中的`cblas_sgemm`函数，该函数共有14个参数，分别如下（参考自官方文档）：

1. `Order`：指定矩阵乘法在行主序(`CblasRowMajor`)或是列主序(`CblasColMajor`)下进行
2. `TransA`：左乘矩阵是否转置(`CblasTrans / CblasNoTrans`)
3. `TransB`：右乘矩阵是否转置(`CblasTrans / CblasNoTrans`)
4. `M`：左乘矩阵或结果矩阵的行数。如果左乘矩阵转置，则表示转置后的列数

5. `N`： 右乘矩阵或结果矩阵的列数。如果右乘矩阵转置，则表示转置后的行数
6. `K`： 左乘矩阵的列数或右乘矩阵的行数
7. `alpha`： 矩阵乘法的系数倍率之一 ($C = \alpha * A \times B + \beta * C$)
8. `A`： 以一维数组存储的左乘矩阵的首位指针(内存是否对齐均可)
9. `lda`
10. `B`： 以一维数组存储的右乘矩阵的首位指针(内存是否对齐均可)
11. `ldb`
12. `beta`： 矩阵乘法的系数倍率之一（$C = \alpha * A \times B + \beta * C$ ）
13. `C`： 以一维数组存储的结果矩阵的首位指针（内存是否对齐均可）
14. `ldc`

因此，我们可以通过以下方式调用OpenBLAS库：

> cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, A->row, B->column, A->column, alpha, A->data, A->column, B->data, B->column, beta, result2->data, result2->column);

| 算法 | 32 | 256 | 2048 | 4096 |
|---|---|---|---|---|
| ikj算法 | 0ms | 3ms | 2859ms | 23173ms |
| Strassen算法 | - | 4ms | 1694ms | 11827ms |
| OpenBLAS | 0ms | 0ms | 60ms | 414ms |

可以看出OpenBLAS的运算速度是非常快的，即便是2048的矩阵甚至是4096的矩阵都仅仅需要几十、几百毫秒就能运算完毕，相比自己的程序还是有非常大的差距的。OpenBLAS在编译的时候，根据cpu构架进行优化，所以执行非常地快。而OpenBLAS有大量对于底层的优化，目前实力有限，还无法对其深究。

## 计时问题

在计算OpenBLAS用时时，还遇到了问题。一开始使用的是`clock()`函数进行计时，但最后得到的结果和其他同学测得的结果相差一个数量级，经过查询资料得知`clock()`函数测量的是处理器的CPU时间，而不是实际的时间或者时钟，当开启多线程的时候，就会发现实际的时间比CPU的时间快多了。

`clock()`函数的功能: 这个函数返回从"开启这个程序进程"到"程序中调用`clock()`函数"时之间的CPU时钟计时单元（clock tick）数。当程序单线程或者单核心机器运行时，这种时间的统计方法是正确的。但是如果要执行的代码多个线程并发执行时就会出问题，因为最终end-begin将会是多个核心总共执行的时钟嘀嗒数，因此造成时间偏大。

由此可知，以后如果多线程程序计时将不能使用`clock()`函数，否则结果会偏大。应该选择记录当前准确时间的计时方法。

# Part 4 - Source Code

> 代码由CMakeLists.txt, matrix.c, matrix.h, main.c组成
>
> 代码中导入了cblas库，在使用前需要将CMakeLists.txt中的第3、4行分别改为头文件和库的地址

> main.c中第4行需要修改为cblas库所在地址
>
> 只需要先执行`cmake .`再执行`make`即可生成可执行文件
>
> 三种算法的结果分别为result1, result2, result3，可以在main.c中自行修改打印的结果

## CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)

project(matmul)

include_directories(/opt/OpenBLAS/include)

link_libraries(/opt/OpenBLAS/lib/libopenblas.so)

aux_source_directory(. DIR_SRCS)

add_executable(matmul ${DIR_SRCS})
```

## matrix.h

```
#pragma once

#pragma GCC optimize(3, "Ofast", "inline")
#include <stdio.h>
#include <stdlib.h>

struct Matrix
{
    int row;
    int column;
    float *data;
};

struct Matrix *creatMatrix(const int, const int, float *const);
void deleteMatrix(struct Matrix *);
void copyMatrix(const struct Matrix *const, struct Matrix *const);
struct Matrix *multiply(const struct Matrix *const, const struct Matrix
*const);
void printMatrix(const struct Matrix *const, FILE *);
void transpose(struct Matrix *const);
struct Matrix *readFile(FILE *);
void *add(int size, struct Matrix *, int, struct Matrix *, int, struct
Matrix *, int);
void *minus(int, struct Matrix *, int, struct Matrix *, int, struct Matrix
*, int);
int findIndex(struct Matrix *, int, int, int);
void *multiOperations(int, struct Matrix *, int, struct Matrix *, int,
struct Matrix *, int, struct Matrix *, int, struct Matrix *, int, char);
struct Matrix *strassen(struct Matrix *, int, struct Matrix *, int, int);
```

## matrix.c

```c
#include "matrix.h"

struct Matrix *creatMatrix(const int row, const int column, float *const data)
{
    struct Matrix *matrix = (struct Matrix *)malloc(sizeof(struct Matrix));
    matrix->row = row;
    matrix->column = column;
    matrix->data = data;
    return matrix;
};

void deleteMatrix(struct Matrix *matrix)
{
    if (matrix != NULL)
    {
        if (matrix->data != NULL)
            free(matrix->data);
        free(matrix);
    }
}

void copyMatrix(const struct Matrix *const src, struct Matrix *const dest)
{
    dest->row = src->row;
    dest->column = src->column;
    free(dest->data);
    dest->data = (float *)malloc(sizeof(float) * dest->row * dest->column);
    for (int counter = 0; counter < dest->row * dest->column; ++counter)
        dest->data[counter] = src->data[counter];
}

struct Matrix *multiply(const struct Matrix *const a, const struct Matrix *const b)
{
    if (a == NULL || b == NULL || a->column != b->row)
    {
        printf("Can not multiply these two matrixes!\n");
        return NULL;
    }
    struct Matrix *result = creatMatrix(a->row, b->column, (float *)malloc(sizeof(float) * a->row * b->column));
    for (int i = 0; i < result->column * result->row; ++i)
        result->data[i] = 0;
    for (int i = 1; i <= result->row; ++i)
    {
        for (int k = 1; k <= a->column; ++k)
        {
            float value = a->data[(i - 1) * a->column + k - 1];
            int index_b = (k - 1) * b->column - 1;
            int index_result = (i - 1) * result->column - 1;
            for (int j = 1; j <= result->column; ++j)
                result->data[++index_result] += value * b->data[++index_b];
        }
    }
```

```c
    return result;
}

void printMatrix(const struct Matrix *const matrix, FILE *file)
{
    for (int i = 0; i < matrix->row * matrix->column; ++i)
    {
        fprintf(file, "%f ", matrix->data[i]);
        if ((i + 1) % matrix->column == 0)
            fprintf(file, "\n");
    }
}

void transpose(struct Matrix *const matrix)
{
    float *data = malloc(sizeof(float) * matrix->column * matrix->row);
    int index = 0;
    for (int i = 0; i < matrix->column * matrix->row; ++i)
    {
        if (i % matrix->row == 0)
            index = i / matrix->row;
        else
            index += matrix->column;
        data[i] = matrix->data[index];
    }
    free(matrix->data);
    matrix->data = data;
    int temp = matrix->row;
    matrix->row = matrix->column;
    matrix->column = temp;
}

struct Matrix *readFile(FILE *file)
{
    float temp;
    char c;
    int row = 0;
    int column = 0;
    int counter = 0;
    char *str;
    while (fscanf(file, "%f", &temp) != EOF)
    {
        ++counter;
        ++column;
        fscanf(file, "%c", &c);
        if (c == '\n' || c == '\r')
            break;
    }
    while (fscanf(file, "%f", &temp) != EOF)
        ++counter;
    if (counter % column != 0)
        return NULL;
    row = counter / column;
    float *data = (float *)malloc(sizeof(float) * row * column);
    rewind(file);
    for (int i = 0; i < row * column; ++i)
        fscanf(file, "%f", &data[i]);
    return creatMatrix(row, column, data);
```

```
}

int findIndex(struct Matrix *matrix, int size, int start_index, int part)
{
    int length = size / 2;
    switch (part)
    {
    case 1:
        return start_index;
        break;
    case 2:
        return start_index + length;
    case 3:
        return start_index + matrix->column * length;
    default:
        return start_index + matrix->column * length + length;
    }
}

void *add(int size, struct Matrix *a, int a_start_index, struct Matrix *b,
int b_start_index, struct Matrix *result, int result_start_index)
{
    int a_index = a_start_index;
    int b_index = b_start_index;
    int result_index = result_start_index;
    for (int i = 0; i < size * size; ++i)
    {
        result->data[result_index] = a->data[a_index] + b->data[b_index];
        if ((i + 1) % size == 0)
        {
            a_index = a_index + a->column - size;
            b_index = b_index + b->column - size;
            result_index = result_index + result->column - size;
        }
        ++a_index;
        ++b_index;
        ++result_index;
    }
    return result;
}

void *minus(int size, struct Matrix *a, int a_start_index, struct Matrix
*b, int b_start_index, struct Matrix *result, int result_start_index)
{
    int a_index = a_start_index;
    int b_index = b_start_index;
    int result_index = result_start_index;
    for (int i = 0; i < size * size; ++i)
    {
        result->data[result_index] = a->data[a_index] - b->data[b_index];
        if ((i + 1) % size == 0)
        {
            a_index = a_index + a->column - size;
            b_index = b_index + b->column - size;
            result_index = result_index + result->column - size;
        }
        ++a_index;
        ++b_index;
```

```
        ++result_index;
    }
    return result;
}

void *multiOperations(int size, struct Matrix *a, int a_start_index, struct Matrix *b, int b_start_index, struct Matrix *c, int c_start_index, struct Matrix *d, int d_start_index, struct Matrix *result, int result_start_index, char thirdOperation)
{
    int a_index = a_start_index;
    int b_index = b_start_index;
    int c_index = c_start_index;
    int d_index = d_start_index;
    int result_index = result_start_index;
    for (int i = 0; i < size * size; ++i)
    {
        result->data[result_index] = a->data[a_index] + b->data[b_index] - c->data[c_index];
        switch (thirdOperation)
        {
        case '+':
            result->data[result_index] += d->data[d_index];
            break;
        case '-':
            result->data[result_index] -= d->data[d_index];
            break;
        }
        if ((i + 1) % size == 0)
        {
            a_index = a_index + a->column - size;
            b_index = b_index + b->column - size;
            c_index = c_index + c->column - size;
            d_index = d_index + d->column - size;
            result_index = result_index + result->column - size;
        }
        ++a_index;
        ++b_index;
        ++c_index;
        ++d_index;
        ++result_index;
    }
    return result;
}

struct Matrix *strassen(struct Matrix *a, int a_start_index, struct Matrix *b, int b_start_index, int size)
{
    struct Matrix *result = creatMatrix(size, size, malloc(sizeof(float) * size * size));
    if (size == 128)
    {
        for (int i = 0; i < 16384; ++i)
            result->data[i] = 0;
        for (int i = 1; i <= 128; ++i)
        {
            for (int k = 1; k <= 128; ++k)
            {
```

```
                float value = a->data[a_start_index + (i - 1) * a->column +
k - 1];
                int index_b = b_start_index + (k - 1) * b->column - 1;
                int index_result = (i - 1) * result->column - 1;
                for (int j = 1; j <= 128; ++j)
                    result->data[++index_result] += value * b-
>data[++index_b];
            }
        }
        return result;
    }

    int subSize = size / 2;
    int subNum = subSize * subSize;
    struct Matrix *S1 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S2 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S3 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S4 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S5 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S6 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S7 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S8 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S9 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    struct Matrix *S10 = creatMatrix(subSize, subSize, malloc(sizeof(float)
* subNum));
    int a11 = findIndex(a, size, a_start_index, 1);
    int a12 = findIndex(a, size, a_start_index, 2);
    int a21 = findIndex(a, size, a_start_index, 3);
    int a22 = findIndex(a, size, a_start_index, 4);
    int b11 = findIndex(b, size, b_start_index, 1);
    int b12 = findIndex(b, size, b_start_index, 2);
    int b21 = findIndex(b, size, b_start_index, 3);
    int b22 = findIndex(b, size, b_start_index, 4);
    minus(subSize, b, b12, b, b22, S1, 0);
    add(subSize, a, a11, a, a12, S2, 0);
    add(subSize, a, a21, a, a22, S3, 0);
    minus(subSize, b, b21, b, b11, S4, 0);
    add(subSize, a, a11, a, a22, S5, 0);
    add(subSize, b, b11, b, b22, S6, 0);
    minus(subSize, a, a12, a, a22, S7, 0);
    add(subSize, b, b21, b, b22, S8, 0);
    minus(subSize, a, a11, a, a21, S9, 0);
    add(subSize, b, b11, b, b12, S10, 0);

    struct Matrix *P1 = strassen(a, a11, S1, 0, subSize);
    deleteMatrix(S1);
    struct Matrix *P2 = strassen(S2, 0, b, b22, subSize);
    deleteMatrix(S2);
    struct Matrix *P3 = strassen(S3, 0, b, b11, subSize);
```

```
    deleteMatrix(S3);
    struct Matrix *P4 = strassen(a, a22, S4, 0, subSize);
    deleteMatrix(S4);
    struct Matrix *P5 = strassen(S5, 0, S6, 0, subSize);
    deleteMatrix(S5);
    deleteMatrix(S6);
    struct Matrix *P6 = strassen(S7, 0, S8, 0, subSize);
    deleteMatrix(S7);
    deleteMatrix(S8);
    struct Matrix *P7 = strassen(S9, 0, S10, 0, subSize);
    deleteMatrix(S9);
    deleteMatrix(S10);
    multiOperations(subSize, P5, 0, P4, 0, P2, 0, P6, 0, result,
findIndex(result, size, 0, 1), '+');
    deleteMatrix(P6);
    add(subSize, P1, 0, P2, 0, result, findIndex(result, size, 0, 2));
    deleteMatrix(P2);
    add(subSize, P3, 0, P4, 0, result, findIndex(result, size, 0, 3));
    deleteMatrix(P4);
    multiOperations(subSize, P5, 0, P1, 0, P3, 0, P7, 0, result,
findIndex(result, size, 0, 4), '-');
    deleteMatrix(P1);
    deleteMatrix(P3);
    deleteMatrix(P5);
    deleteMatrix(P7);
    return result;
}
```

## main.c

```
#include "matrix.h"
#include <stdbool.h>
#include <sys/time.h>
#include </opt/OpenBLAS/include/cblas.h>

int main(int args, char **argv)
{
    if (args < 4)
    {
        printf("File missing!\n");
        return 1;
    }
    FILE *input1, *input2, *output;
    input1 = fopen(argv[1], "r");
    input2 = fopen(argv[2], "r");
    if (input1 == NULL || input2 == NULL)
    {
        printf("File does not exist!\n");
        return 1;
    }

    struct Matrix *A = readFile(input1);

    fclose(input1);
    if (A == NULL)
```

```
    {
        printf("File1 Corrupted!\n");
        return 1;
    }
    struct Matrix *B = readFile(input2);
    fclose(input2);
    if (B == NULL)
    {
        printf("File2 Corrupted!\n");
        return 1;
    }
    printf("Read File Successfully!\n");

    struct timeval start_time, end_time;
    long long duration;

    gettimeofday(&start_time, NULL);
    struct Matrix *result1 = multiply(A, B);
    gettimeofday(&end_time, NULL);
    if (result1 == NULL)
        return 1;
    duration = 1000000 * (end_time.tv_sec - start_time.tv_sec) +
end_time.tv_usec - start_time.tv_usec;
    printf("The calculate time is: %lldms\n", duration / 1000);

    struct Matrix *result2;
    if (A->row == A->column && A->row == B->column && (A->row & (A->row -
1)) == 0 && A->row >= 128)
    {
        gettimeofday(&start_time, NULL);
        result2 = strassen(A, 0, B, 0, A->column);
        gettimeofday(&end_time, NULL);
        duration = 1000000 * (end_time.tv_sec - start_time.tv_sec) +
end_time.tv_usec - start_time.tv_usec;
        printf("The strassen calculate time is: %lldms\n", duration / 1000);
    }

    struct Matrix *result3 = creatMatrix(A->row, B->column, (float
*)malloc(sizeof(float) * A->row * B->column));
    const float alpha = 1;
    const float beta = 0;
    gettimeofday(&start_time, NULL);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, A->row, B-
>column, A->column, alpha, A->data, A->column, B->data, B->column, beta,
result3->data, result3->column);
    gettimeofday(&end_time, NULL);
    duration = 1000000 * (end_time.tv_sec - start_time.tv_sec) +
end_time.tv_usec - start_time.tv_usec;
    printf("The OpenBLAS calculate time is: %lldms\n", duration / 1000);

    output = fopen(argv[3], "w");
    if (output == NULL)
        printf("File does not exist!\n");
    printMatrix(result2, output);
    fclose(output);
    printf("Write File Successfully!\n");
    return 0;
}
```

# Part 5 - Result & Verification

在本部分，将验证部分函数的正确性以及矩阵乘法的结果。

## 矩阵的复制

原矩阵：

```
1 2 3
4 5 6
7 8 9
10 11 12
```

打印出复制的矩阵：

```
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
10.000000 11.000000 12.000000
```

改变原矩阵中的值后，复制出的矩阵仍然不改变

## 矩阵的转置

原矩阵：

```
1 2 3
4 5 6
7 8 9
10 11 12
```

打印出转置后的矩阵：

```
1.000000 4.000000 7.000000 10.000000
2.000000 5.000000 8.000000 11.000000
3.000000 6.000000 9.000000 12.000000
```

## 长方形矩阵的乘法

由于小数存在精度问题不方便准确验证，因此这里验算的均为整数数据

```
A =  1 2 3
     4 5 6
     7 8 9
     10 11 12

B =  3 6 8 3 1
     2 3 8 1 3
     4 6 8 9 3
```

result :

```
19.000000 30.000000 48.000000 32.000000 16.000000
46.000000 75.000000 120.000000 71.000000 37.000000
73.000000 120.000000 192.000000 110.000000 58.000000
100.000000 165.000000 264.000000 149.000000 79.000000
```

经过验证，结果正确。

若将B新增一行，输出：`Can not multiply these two matrixes!`

## 其他矩阵乘法验证

这一部分已经在Project 2进行过验证，由于数据量太大，结果无法上传到报告中。但经过验算，传统算法和Strassen算法以及OpenBLAS的计算结果在不考虑计算精度问题的情况下都是正确的。故不再在本报告中演示。