

CS205 C/C++ Programming - Project_2

Name: 刘乐奇 (Liu Leqi)

SID: 12011327

CS205 C/C++ Programming - Project_2

- 1 题目分析
 - 1.1 环境及工具
 - 1.2 文件读取写入
 - 1.3 矩阵合法性判断
 - 1.4 计时器
- 2 代码
 - 2.1 IO
 - 2.2 matrix
 - 2.3 mul
 - 2.3.1 朴素乘法
 - 2.3.2 ikj矩阵乘法
 - 2.3.3 Strassen算法
 - 2.3.4 SIMD优化
 - 2.3.5 O3优化
 - 2.3.6 OpenBLAS
 - 2.4 main
 - 2.5 CMakeLists.txt
- 3 测试及分析
 - 3.1 文件输入输出及矩阵复制操作
 - 3.2 ijk和ikj矩阵乘法
 - 3.3 Strassen算法
 - 3.4 OpenBLAS
 - 3.5 O3
- 4 困难及解决
 - 4.1 rewind()重置文件指针
 - 4.2 矩阵元素输入/输出
 - 4.3 矩阵拷贝
 - 4.3.1 memcpy()参数
 - 4.3.2 一维数组矩阵转化二维数组矩阵
 - 4.3.3 C中的i++和++i的效率区别
 - 4.3.4 SIMD中的内存对齐
 - 4.3.5 MinGW中unaligned_alloc()函数的问题
 - 4.3.6 Multiple definition
 - 4.3.7 结构体删除
- 5 与Project2对比
- 6 总结

1 题目分析

本程序与Project2一样，需要通过命令行参数获取两个存储在txt文件的原矩阵，在进行矩阵乘法后将结果存储到命令行所指定的新建txt文件中。虽然本程序只要求实现 `float` 类型的矩阵乘法，但是同样的，本程序还要考虑到计算速度和精度问题，以及文件的读取和写入。不同的是，本程序还需要实现矩阵的复制。

需要注意的是，该project要求完全使用C语言实现，也即不能使用c++中非C的特性。

1.1 环境及工具

本程序主要使用Windows系统，利用vscode及wsl编写并编译运行，使用的C语言标准为C11。同时也用了Clion、VS、VMware上的Ubuntu虚拟机、MinGW。

1.2 文件读取写入

在C语言中，对文本文件读写最常用的是利用 `FILE*` 文件指针和标准IO库 `<stdio.h>`。并且，在 `<stdio.h>` 中也提供了 `fEOF()` 和 `ferror()` 用于判断是否文件结尾和是否读写错误（但本程序直接用 `EOF` 判断文件结尾）。此外，若写入文件时目标路径下无指定文件，则创建之；否则将会报错，让用户检查是否误输入了同名文件。

1.3 矩阵合法性判断

由于矩阵内容和矩阵相乘需要符合一定条件，如内容需为数字、行列对应，因此在读取矩阵后需要对矩阵是否合法进行判断。若规范，则继续进行后续操作；否则直接退出程序。

在本程序中，将使用 `fscanf()` 的返回值判断是否成功读入了浮点数。

1.4 计时器

可以通过 `<time.h>` 中的 `clock()` 来计时，计算时间差并输出，能简便得出程序某部分的运行时间。

在本程序中，使用了宏定义了计时器的头和尾，使得计时代码较为美观。

2 代码

本程序包含以下文件：

- ① IO.h IO.c 矩阵读取、初始化、矩阵输出
- ② matrix.h matrix.c 矩阵结构体、矩阵合法性判断
- ③ mul.h mul.c 矩阵乘法
- ④ main.cpp, CMakeLists.txt

2.1 IO

头文件

```
//IO.h
#ifndef _IO_H
#define _IO_H

#include "matrix.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <sys/mman.h>
//#include <fcntl.h>

void valid_file(FILE *, const char *);

void close_file(FILE *);
```

```

size_t get_rows(FILE *);

size_t get_cols(FILE *);

void mat_init(FILE *, size_t, size_t, matp);

void mat_out_file(FILE *, matp);

//void map_read(const char *, matp);
//
//void map_write(const char *, matp);

#endif // _IO_H

```

源文件

```

//IO.c
#include "IO.h"
#include "matrix.h"

clock_t start, end;

//判断文件指针是否指向成功
void valid_file(FILE *fp, const char *filename) {
    if (fp == NULL) {
        fprintf(stderr, "Error: (%s) No such file.\n", filename);
        exit(EXIT_FAILURE);
    }
}

//关闭文件
void close_file(FILE *fp) {
    if (fp != NULL) {
        free(fp);
    } else {
        fprintf(stderr, "Error: The file pointer has been deleted.\n");
        exit(EXIT_FAILURE);
    }
}

//获取矩阵行数
size_t get_rows(FILE *fp) {
    rewind(fp);
    size_t row_cnt = 0;
    char tmp[1010];
    while (fgets(tmp, 1000, fp)) {
        if (tmp[strlen(tmp) - 1] == '\n') {
            row_cnt++;
        }
    }
    return row_cnt;
}

//获取矩阵列数
size_t get_cols(FILE *fp) {
    rewind(fp);

```

```

    size_t col_cnt = 0;
    char tmp;
    while ((tmp = fgetc(fp)) != '\n') {
        if (tmp == ' ') {
            col_cnt++;
        }
    }
    col_cnt++;
    return col_cnt;
}

//用于从文件中读取矩阵数据，初始化矩阵
void mat_init(FILE *fp, size_t row, size_t col, matp mat) {
    rewind(fp);

    START;
    mat->row = row;
    mat->col = col;
    mat->val = (float *) malloc(row * col * sizeof(float));
    size_t i, j;
    float tmp = 0;
    int che = 0;
    int control = 0;
    for (i = 0; i < row; ++i) {
        for (j = 0; j < col; ++j) {
            if ((che = fscanf(fp, "%f", &tmp)) != EOF) {
                if (che != 1) {
                    fprintf(stderr, "Error: For input not float.\n");
                    exit(EXIT_FAILURE);
                }
                *(mat->val + j + i * row) = tmp;
            } else {
                control = 1;
                break;
            }
        }
        if (control == 1) break;
    }
    END("mat_init");
}

//输出矩阵到文件中
void mat_out_file(FILE *fp, matp mat) {
    rewind(fp);

    START;
    size_t i, j;
    float tmp;
    for (i = 0; i < mat->row; ++i) {
        for (j = 0; j < mat->col; ++j) {
            tmp = *(mat->val + j + i * mat->row);
            fprintf(fp, "%.2f ", tmp);
        }
        fprintf(fp, "\n");
    }
    END("mat_out_file");
}

```

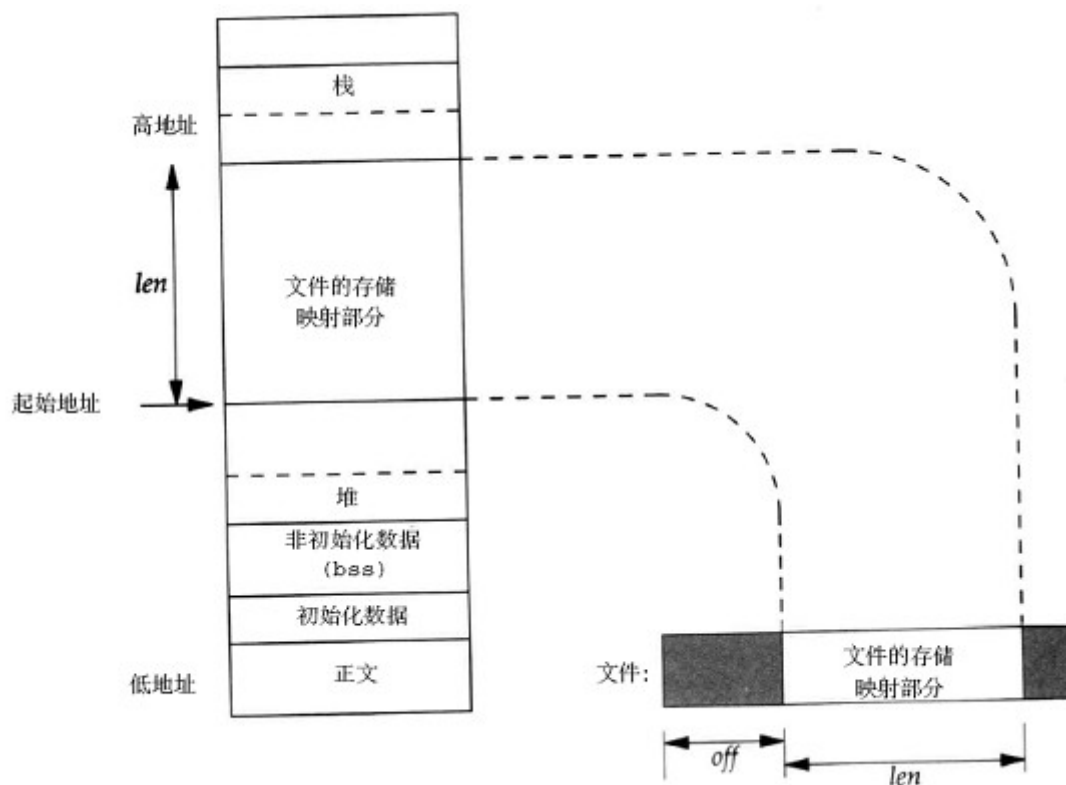
```

//内存映射，仅限于linux
//读
//void map_read(const char *filename, matp m) {
//    int fd = open(filename, O_RDONLY);
//    size_t len = lseek(fd, 0, SSEK_END);
//    float *buff = (float *) mmap(NULL, len, PORT_READ, MAP_PRIVATE, fd, 0);
//    close(fd);
//}
//写
//void map_write(const char *filename, matp m) {
//    size_t len = m->row * m->col;
//    int fd = open(filename, O_RDWR | O_CREAT, 00777);
//    lseek(fd, len - 1, SEEK_END);
//    write(fd, " ", 1);
//    char *buff = (char *) mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
//    close(fd);
//    memcpy(buff, m->val, len);
//    munmap(buff, len);
//}

```

需要注意的是，每次在使用文件指针前，要将其置于文件头，故要用到 `rewind()` 函数。

除了一般的输入输出方法，本程序还提供了专用于Linux系统的内存映射方法，直接将磁盘上的文件映射到内存（准确的说是虚拟内存）中，不需要其他额外空间，对内存映射区的修改可以与磁盘文件保持同步，故读写速度非常快。^{1 2} 但是可惜的是，由于没有Linux系统的电脑，也不敢用虚拟机，难以对该方法进行测试，无法确保能对文件读写成功，故使用时需注意（建议使用前先测试或完全别用）。



2.2 matrix

头文件

```
//matrix.h
#ifndef _MATRIX_H
#define _MATRIX_H

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

#define START start = clock()
#define END(message) end = clock();\
    printf("%s cost %.8lf ms.\n", message, (1000.0 * (double)(end-\
start))/CLOCKS_PER_SEC)

extern clock_t start, end;

typedef struct matrix_1 {
    size_t col;
    size_t row;
    float *val;
} mat, *matp;

typedef struct matrix_2 {
    size_t col;
    size_t row;
    float **val;
} mat_2, *matp_2;

void mat_del(matp);

void mat_del_2(matp_2);

void mat_copy(matp, const matp);

void mat_copy_to2(matp_2, const matp);

void mat_match_check(matp, const matp, const matp);

void mat_match_check_2(matp_2, const matp_2, const matp_2);

void mat_compare(const matp, const matp_2);

#endif // _MATRIX_H
```

源文件

```
//matrix.c
#include "matrix.h"

clock_t start, end;
```

//在结束时删除矩阵

```
void mat_del(matp mat) {
    if (mat == NULL) {
        fprintf(stderr, "Error: No such matrix to be deleted.\n");
        exit(EXIT_FAILURE);
    } else if (mat->val != NULL) {
        free(mat->val);
    }
}
```

//二维数组矩阵删除

```
void mat_del_2(matp_2 mat) {
    if (mat == NULL) {
        fprintf(stderr, "Error: No such matrix to be deleted.\n");
        exit(EXIT_FAILURE);
    } else if (mat->val != NULL) {
        for (int i = 0; i < mat->row; ++i) {
            free(mat->val[i]);
        }
        free(mat->val);
    }
}
```

//用于矩阵拷贝

```
void mat_copy(matp des, const matp src) {
    START;
    des->row = src->row;
    des->col = src->col;
    des->val = (float *) malloc(src->row * src->col * sizeof(float));
    size_t i, j;
    for (i = 0; i < des->row; ++i) {
        for (j = 0; j < des->col; ++j) {
            *(des->val + i * des->row + j) = *(src->val + i * src->row + j);
        }
    }
    END("mat_copy");
}
```

//用于将一维矩阵拷贝至二维矩阵

```
void mat_copy_to2(matp_2 des, const matp src) {
    START;
    des->row = src->row;
    des->col = src->col;
    des->val = (float **) malloc(src->row * sizeof(float *));
    size_t k;
    for (k = 0; k < src->row; ++k) {
        *(des->val + k) = malloc(src->col * sizeof(float));
    }
    size_t i, j;
    for (i = 0; i < src->row; ++i) {
        for (j = 0; j < src->col; ++j) {
            des->val[i][j] = *(src->val + i * src->row + j);
        }
    }
    END("mat_copy_to2");
}
```

//检测矩阵行列是否对应

```

void mat_match_check(matp des, const matp l_mat, const matp r_mat) {
    if (l_mat->row != r_mat->col) {
        fprintf(stderr, "Error: Not matched source matrices.\n");
        exit(EXIT_FAILURE);
    }
    if (l_mat->row != des->row || r_mat->col != des->col) {
        des->row = l_mat->row;
        des->col = r_mat->col;
        des->val = (float *) malloc(des->row * des->col * sizeof(float));
    }
}

//检测二维数组矩阵
void mat_match_check_2(matp_2 des, const matp_2 l_mat, const matp_2 r_mat) {
    if (l_mat->row != r_mat->col) {
        fprintf(stderr, "Error: Not matched source matrices.\n");
        exit(EXIT_FAILURE);
    }
    if (l_mat->row != des->row || r_mat->col != des->col) {
        des->row = l_mat->row;
        des->col = r_mat->col;
        des->val = (float **) malloc(des->row * des->col * sizeof(float **));
        for (int i = 0; i < des->row; ++i) {
            *(des->val + i) = (float *) malloc(des->row * des->col *
sizeof(float));
        }
    }
}

//比较确认一维转二维成功
void mat_compare(const matp m1, const matp_2 m2) {
    size_t i, j;
    for (i = 0; i < m1->row; ++i) {
        for (j = 0; j < m1->col; ++j) {
            if ((*m1->val + i * m1->row + j) - m2->val[i][j]) > 0.00001) {
                fprintf(stderr, "Error: matrix to 2-dimension wrong.\n");
                exit(EXIT_FAILURE);
            }
        }
    }
}

```

2.3 mul

头文件

```

//mul.h
#ifndef MUL_H
#define MUL_H

#include "matrix.h"

// #include <immintrin.h>
#include <blas.h>

void mat_mul_simple(matp, const matp, const matp);

```



```

void mat_mul_simple_2(matp_2, const matp_2, const matp_2);

void mat_mul_divide(matp, const matp, const matp);

void mat_mul_divide_2(matp_2, const matp_2, const matp_2);

float **init_array(size_t);

void del_array(size_t, float **);

void mul(size_t, float **, float **, float **);

void add(size_t, float **, float **, float **);

void sub(size_t, float **, float **, float **);

void strassen(size_t, float **, float **, float **);

// float vec_dot(const float *, const float *, const size_t);

// void mat_dot(matp_2, const matp_2, const matp_2);

void blas(matp, const matp, const matp);

#endif //MUL_H

```

源文件

```

//mul.c
#include "mul.h"

clock_t start, end;

```

2.3.1 朴素乘法

```

//mul.c
//朴素乘法
void mat_mul_simple(matp des, const matp l_mat, const matp r_mat) {
    START;
    size_t i, j, k;
    for (i = 0; i < l_mat->row; ++i) {
        for (j = 0; j < r_mat->col; ++j) {
            for (k = 0; k < l_mat->col; ++k) {
                *(des->val + i * l_mat->row + j) +=
                    *(l_mat->val + i * l_mat->row + k) * *(r_mat->val + k *
l_mat->row + j);
            }
        }
    }
    END("mat_mul_simple");
}

//二维数组
void mat_mul_simple_2(matp_2 des, const matp_2 l_mat, const matp_2 r_mat) {
    START;
    size_t i, j, k;

```

```

    for (i = 0; i < l_mat->row; ++i) {
        for (j = 0; j < r_mat->col; ++j) {
            for (k = 0; k < l_mat->col; ++k) {
                des->val[i][j] += l_mat->val[i][k] * r_mat->val[k][j];
            }
        }
    }
    END("mat_mul_simple_2");
}

```

最简单直观的矩阵乘法，按索引顺序也常被称为**ijk矩阵乘法**，虽然是 $O(n^3)$ 的时间复杂度，但非常容易想到和实现。

2.3.2 ikj矩阵乘法

```

//mul.c
//ikj矩阵乘法,一维数组
void mat_mul_divide(matp des, const matp l_mat, const matp r_mat) {
    START;
    size_t i, j, k;
    for (i = 0; i < l_mat->row; ++i) {
        for (k = 0; k < r_mat->row; ++k) {
            float tmp = *(l_mat->val + i * l_mat->row + k);
            for (j = 0; j < r_mat->col; ++j) {
                *(des->val + i * l_mat->row + j) += tmp * *(r_mat->val + k *
r_mat->row + j);
            }
        }
    }
    END("mat_mul_divide");
}

//二维数组实现
void mat_mul_divide_2(matp_2 des, const matp_2 l_mat, const matp_2 r_mat) {
    START;
    size_t i, j, k;
    for (i = 0; i < l_mat->row; ++i) {
        for (k = 0; k < r_mat->row; ++k) {
            float tmp = l_mat->val[i][k];
            for (j = 0; j < r_mat->col; ++j) {
                des->val[i][j] += tmp * r_mat->val[k][j];
            }
        }
    }
    END("mat_mul_divide_2");
}

```

在查阅有关矩阵乘法的相关资料中，我发现了与**ijk矩阵乘法**相类似，但计算用时却减少了的方法：**ikj矩阵乘法**。该算法的优势在于通过改变索引顺序，使得在访问内存时有了效率优化。

```

0,0 | 0,1 | 0,2 | 0,3
----+----+----+----
1,0 | 1,1 | 1,2 | 1,3
----+----+----+----
2,0 | 2,1 | 2,2 | 2,3

```

想象中的矩阵

```

0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3

```

实际内存中的矩阵

<code>x[0][0]</code>	<code>x[1][0]</code>	<code>x[2][0]</code>	实际的矩阵索引
<code>x[0][1]</code>	<code>x[1][1] etc...</code>		

在使用二维数组表示矩阵时，由上图可知，实际上矩阵元素在内存中的存储与我们想象中的不太一样，是一种线性的存储。尽管一维数组在内存中是连续的，但是二维数组的存储方式导致了若按第二个下标连续变化来访问元素，会使得电脑需要在内存中多次跳跃访问元素，易出现缓存缺失（cache misses）。³

实际上，**ikj**和**ijk**的用时都比较慢，除了由于时间复杂度都是 $O(n^3)$ ，也有内存访问不连续，这会 cache 命中率不高。若是用一维数组来表示矩阵，将会减轻访存问题的影响，但是实际上差异仍然存在，而且在大矩阵中尤为明显。（详细的分析将会在下面介绍）

2.3.3 Strassen算法

```
//mul.c
//Strassen算法
//初始化二维动态数组
float **init_array(size_t n) {
    float **arr = (float **) malloc(n * n * sizeof(float *));
    for (size_t i = 0; i < n; ++i) {
        arr[i] = (float *) malloc(n * sizeof(float));
    }
    return arr;
}

//删除二维动态数组
void del_array(size_t n, float **arr) {
    for (int i = 0; i < n; ++i) {
        free(arr[i]);
    }
    free(arr);
}

//二阶矩阵相乘
void mul(size_t n, float **A, float **B, float **C) {
    size_t i, j, k;
    for (i = 0; i < n; ++i) {
        for (k = 0; k < n; ++k) {
            float tmp = A[i][k];
            for (j = 0; j < 2; ++j) {
                C[i][j] += tmp * B[k][j];
            }
        }
    }
}

//矩阵加法：
void add(size_t n, float **A, float **B, float **C) {
    size_t i, j;
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            C[i][j] = A[i][j] + B[i][j];
}

//矩阵减法：
```

```

void sub(size_t n, float **A, float **B, float **C) {
    size_t i, j;
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            C[i][j] = A[i][j] - B[i][j];
}

//strassen主体
void strassen(size_t n, float **A, float **B, float **C) {
    if (n <= 256) {
        mul(n, A, B, C);
    } else {
        n >>= 1;
        float **A11 = init_array(n);
        float **A12 = init_array(n);
        float **A21 = init_array(n);
        float **A22 = init_array(n);

        float **B11 = init_array(n);
        float **B12 = init_array(n);
        float **B21 = init_array(n);
        float **B22 = init_array(n);

        float **M1 = init_array(n);
        float **M2 = init_array(n);
        float **M3 = init_array(n);
        float **M4 = init_array(n);
        float **M5 = init_array(n);
        float **M6 = init_array(n);
        float **M7 = init_array(n);

        float **AA = init_array(n);
        float **BB = init_array(n);

        float **C11 = init_array(n);
        float **C12 = init_array(n);
        float **C21 = init_array(n);
        float **C22 = init_array(n);

        size_t i, j;
        for (i = 0; i < n; ++i) {
            for (j = 0; j < n; ++j) {
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j + n];
                A21[i][j] = A[i + n][j];
                A22[i][j] = A[i + n][j + n];

                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j + n];
                B21[i][j] = B[i + n][j];
                B22[i][j] = B[i + n][j + n];
            }
        }

        sub(n, B12, B22, BB);
        strassen(n, A11, BB, M1);

        add(n, A11, A12, AA);
    }
}

```

```

strassen(n, AA, B22, M2);

add(n, A21, A22, AA);
strassen(n, AA, B11, M3);

sub(n, B21, B11, BB);
strassen(n, A22, BB, M4);

add(n, A11, A22, AA);
add(n, B11, B22, BB);
strassen(n, AA, BB, M5);

sub(n, A12, A22, AA);
add(n, B21, B22, BB);
strassen(n, AA, BB, M6);

sub(n, A11, A21, AA);
add(n, B11, B12, BB);
strassen(n, AA, BB, M7);

//C11 = M5 + M4 - M2 + M6
add(n, M5, M4, AA);
sub(n, M6, M2, BB);
add(n, AA, BB, C11);

//C12 = M1 + M2
add(n, M1, M2, C12);

//C21 = M3 + M4
add(n, M3, M4, C21);

//C22 = M5 + M1 - M3 - M7
sub(n, M5, M3, AA);
sub(n, M1, M7, BB);
add(n, AA, BB, C22);

for (i = 0; i < n; ++i) {
    for (j = 0; j < n; ++j) {
        C[i][j] = C11[i][j];
        C[i][j + n] = C12[i][j];
        C[i + n][j] = C21[i][j];
        C[i + n][j + n] = C22[i][j];
    }
}

del_array(n, A11);
del_array(n, A12);
del_array(n, A21);
del_array(n, A22);

del_array(n, B11);
del_array(n, B12);
del_array(n, B21);
del_array(n, B22);

del_array(n, M1);
del_array(n, M2);
del_array(n, M3);
del_array(n, M4);

```

```

del_array(n, M5);
del_array(n, M6);
del_array(n, M7);

del_array(n, AA);
del_array(n, BB);

del_array(n, C11);
del_array(n, C12);
del_array(n, C21);
del_array(n, C22);
}
}

```

Strassen算法也是一种分而治之的算法。与前面的简单分而治之算法相同，Strassen算法也需要将两个 $n \times n$ 的矩阵分别分割成 $\frac{n}{2} \times \frac{n}{2}$ 的矩阵。不同的是，Strassen算法利用了矩阵的加法和减法，通过矩阵的运算法则，减少了子矩阵相乘的次数（也即下图中递归式的系数减少），从而加快了算法速度。^{4 5}

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left. \begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dh \\ u = cf + dg \end{array} \right\} \begin{array}{l} 8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{array}$$

$$T(n) = 8T(n/2) + \Theta(n^2)$$

submatrices submatrix size work adding submatrices

$$n^{\log_2 8} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

- 1. Divide:** Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
- 2. Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
- 3. Combine:** Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

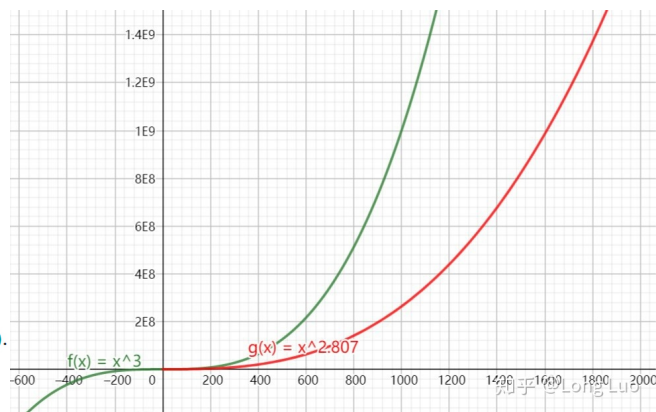
$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$n^{\log_2 7} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\log_2 7}).$$

Multiply 2×2 matrices with only 7 recursive mults.

$$\begin{array}{ll} P_1 = a \cdot (f - h) & r = P_5 + P_4 - P_2 + P_6 \\ P_2 = (a + b) \cdot h & s = P_1 + P_2 \\ P_3 = (c + d) \cdot e & t = P_3 + P_4 \\ P_4 = d \cdot (g - e) & u = P_5 + P_1 - P_3 - P_7 \\ P_5 = (a + d) \cdot (e + h) & \\ P_6 = (b - d) \cdot (g + h) & \\ P_7 = (a - c) \cdot (e + f) & \end{array}$$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!



根据 (<https://blog.csdn.net/handawnc/article/details/7987107>) 的分析，在矩阵规模比较小的情况下，Strassen算法并未真正降低用时，反而还使得在一定范围内，规模增加时，时间急剧增多。这是因为在Strassen算法中需要创建大量的动态数组，在分配堆内存时会占用大量计算时间（同时若未注意，容易发生内存泄露问题），即便只有 $O(n^{\log_2 7})$ 的算法复杂度，但其隐含的常数却比 $O(n^3)$ 的朴素算法要大得多。因此，在矩阵规模比较低（约 $n < 300$ ）的情况下Strassen算法并不具有优势，于是在矩阵规模比较低的情况下建议将其换成朴素矩阵乘法。

2.3.4 SIMD优化

```

//mul.c
//SIMD优化，基于intel的AVX指令集
float vec_dot(const float *v1, const float *v2, const size_t len) {
    if (len % 8 != 0) {
        fprintf(stderr, "Error: The size must be power of 8.\n");
    }
}

```

```

        exit(EXIT_FAILURE);
    }

    float s[8] = {0};
    __m256 a, b;
    __m256 c = _mm256_setzero_ps();

    for (size_t i = 0; i < len; i += 8) {
        a = _mm256_load_ps(v1 + i);
        b = _mm256_load_ps(v2 + i);
        c = _mm256_add_ps(c, _mm256_mul_ps(a, b));
    }
    _mm256_store_ps(s, c);
    return (s[0] + s[1] + s[2] + s[3] + s[4] + s[5] + s[6] + s[7]);
}

void mat_dot(matp_2 m3, const matp_2 l_mat, const matp_2 r_mat) {
    START;
    size_t i, j, k;
    float *v1 = (float *) aligned_alloc(256, l_mat->row * sizeof(float));
    float *v2 = (float *) aligned_alloc(256, r_mat->col * sizeof(float));
    for (i = 0; i < l_mat->row; ++i) {
        for (j = 0; j < r_mat->col; ++j) {
            for (k = 0; k < l_mat->col; ++k) *(v1 + k) = l_mat->val[j][k];
            for (k = 0; k < r_mat->row; ++k) *(v2 + k) = r_mat->val[k][j];
            m3->val[i][j] = vec_dot(v1, v2, l_mat->row);
        }
    }
    free(v1);
    free(v2);
    END("_mm_mat_dot");
}

```

由于作者本人在学堂在线上再次发现了于老师旧版的录课视频，在speed up your program⁶一节课中发现了SIMD优化。同时在查阅intel的参考文档⁷以及部分网上的教程^{8 9}后，得出了如上SIMD加速优化。

要注意的是，要使用AVX指令集，还需要在命令行中加上字段 `-mavx`：

```
gcc main.c matrix.c mul.c IO.c -mavx
```

抑或是在cmake中加上：

```
set(CMAKE_C_FLAGS "-mavx")
```

可惜的是，不知为何总是发生段错误（Segmentation fault），由于时间原因，作者未来得及找到bug原因，因此就此作罢。（目前猜测错误应该是在 `aligned_alloc()` 中可能内存申请不足，也有可能是 `memcpy()` 出现内存冲突）

2.3.5 O3优化

极力提高运行速度。可在命令行加上字段 `-O3`：

```
gcc main.c matrix.c mul.c IO.c -O3
```

抑或是在cmake中加上：

```
set(CMAKE_C_FLAGS "-O3")
```

2.3.6 OpenBLAS

```
//mul.c
//OpenBLAS
void blas(matp m3, const matp m1, const matp m2) {
    START;
    int M = m1->row, N = m2->col, K = m1->col;
    float alpha = 1.0f, beta = 0.0f;
    int lda = K, ldb = N, ldc = N;
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha, m1->val, lda, m2->val, ldb, beta, m3->val, ldc);
    END("blas");
}
```

利用第三方库 OpenBLAS 实现的矩阵乘法。调用需如下：

```
gcc -o matmul main.c mul.c matrix.c IO.c -I /opt/OpenBLAS/include/ -L/opt/OpenBLAS/lib -lopenblas
```

2.4 main

```
//main.c
#include "IO.h"
#include "matrix.h"
#include "mul.h"

clock_t start, end;

int main(int argc, char **argv) {
    if (argc != 4) {
        fprintf(stderr, "Error: The number of arguments is unexpected.\n");
        exit(EXIT_FAILURE);
    }

    FILE *fp_mat1 = fopen(*(argv + 1), "r");
    valid_file(fp_mat1, *(argv + 1));
    FILE *fp_mat2 = fopen(*(argv + 2), "r");
    valid_file(fp_mat2, *(argv + 2));

    size_t mat1_row = get_rows(fp_mat1), mat1_col = get_cols(fp_mat1);
    size_t mat2_row = get_rows(fp_mat2), mat2_col = get_cols(fp_mat2);

    mat mat1, mat2, mat_out;
    mat_init(fp_mat1, mat1_row, mat1_col, &mat1);
    mat_init(fp_mat2, mat2_row, mat2_col, &mat2);
    mat_match_check(&mat_out, &mat1, &mat2);

    mat_2 mat1_2, mat2_2, mat_out_2;
    mat_copy_to2(&mat_out_2, &mat_out);
    mat_copy_to2(&mat1_2, &mat1);
    mat_copy_to2(&mat2_2, &mat2);
```



```

mat_match_check_2(&mat_out_2, &mat1_2, &mat2_2);

//    //测试拷贝一维数组和二维数组的区别
//    mat test1;
//    mat_2 test2;
//    mat_copy(&test1, &mat1);
//    mat_copy_to2(&test2, &mat1);
//    mat_del(&test1);
//    mat_del_2(&test2);

//比较拷贝是否成功
mat_compare(&mat1, &mat1_2);
mat_compare(&mat2, &mat2_2);

//ijk
mat_mul_simple(&mat_out, &mat1, &mat2);
mat_mul_simple_2(&mat_out_2, &mat1_2, &mat2_2);
mat_compare(&mat_out, &mat_out_2);

//ikj
mat_mul_divide(&mat_out, &mat1, &mat2);
mat_mul_divide_2(&mat_out_2, &mat1_2, &mat2_2);
mat_compare(&mat_out, &mat_out_2);

//Strassen
START;
strassen(mat_out_2.row, mat1_2.val, mat2_2.val, mat_out_2.val);
END("strassen");

//SIMD
// mat_dot(&mat_out_2, &mat1_2, &mat2_2);

//OpenBLAS
blas(&mat_out, &mat1, &mat2);

FILE *fp_mat_out = fopen(*(argv + 3), "r");

if (fp_mat_out == NULL) {
    fp_mat_out = fopen(*(argv + 3), "w+");
    mat_out_file(fp_mat_out, &mat_out);
} else {
    fprintf(stderr, "Error: The file has existed. Please try again after
checked.\n");
    exit(EXIT_FAILURE);
}

mat_del(&mat1);
mat_del(&mat2);
mat_del(&mat_out);
mat_del_2(&mat1_2);
mat_del_2(&mat2_2);
mat_del_2(&mat_out_2);

close_file(fp_mat1);
close_file(fp_mat2);
close_file(fp_mat_out);
return 0;

```

```
}

```

2.5 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16.3)
project(matmul)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_C_STANDARD 11)

set(CMAKE_C_FLAGS "-O3 -mavx -I /opt/OpenBLAS/include/ -L/opt/OpenBLAS/lib -lopenblas")

aux_source_directory(. DIR_SRC)

add_executable(matmul ${DIR_SRC})

target_link_libraries(matmul ${BLAS_LIBRARY})

```

有时候并未用上cmake，而是直接用命令行了。

3 测试及分析

3.1 文件输入输出及矩阵复制操作

```
mat_init cost 0.27800000 ms.      mat_init cost 9.69200000 ms.      mat_init cost 554.00900000 ms.
mat_init cost 0.30500000 ms.      mat_init cost 9.56300000 ms.      mat_init cost 545.66100000 ms.
mat_out_file cost 0.18900000 ms.  mat_out_file cost 9.88000000 ms.  mat_out_file cost 608.15200000 ms.

```

(从左自右依次为32阶，256阶，2048阶)

Project3 (ms)	32阶	256阶	2048阶
IN (两个矩阵)	0.58ms	19.26ms	1143.11ms
OUT (一个矩阵)	0.189ms	9.880ms	688.152ms

Project2 (ms)	32阶	256阶	2048阶
IN (两个矩阵)	2.87ms	91.79ms	1554.24ms
OUT (一个矩阵)	4.80ms	67.68ms	2331.66ms

比较两次Project中文件读入（包括矩阵初始化）和文件输出，很显然本程序有更快的效果。猜测是因为C++的iostream流操作的速度要比C语言的stdio标准输入输出操作慢得多。

在本程序中，还实现了矩阵的拷贝（从一维数组到一维数组和从一维数组到二维数组）。

```
mat_copy cost 0.00900000 ms.      mat_copy cost 0.23000000 ms.      mat_copy cost 25.47000000 ms.
mat_copy_to2 cost 0.00600000 ms.  mat_copy_to2 cost 0.23400000 ms.  mat_copy_to2 cost 88.59700000 ms.

```

(从左自右依次为32阶，256阶，2048阶)

从一维数组 (ms)	32阶	256阶	2048阶
到一维数组	0.009ms	0.23ms	25.47ms
到二维数组	0.006ms	0.234ms	88.597ms

在矩阵比较大的时候从一维数组到二维数组的复制效率比较低，但是在比较小的时候相差无几，推测应该是拷贝时，二维数组比一维数组的内存跳跃要多得多，cache命中率也下降，故耗时更多。

另外，还有 `mat_compare()` 函数用来比较两个矩阵中的元素是否一样（在可接受的误差范围内）。

3.2 ijk和ikj矩阵乘法

```
mat_mul_simple cost 0.17200000 ms. mat_mul_simple cost 131.89700000 ms.
mat_mul_simple_2 cost 0.14200000 ms. mat_mul_simple_2 cost 71.55800000 ms.
mat_mul_divide cost 0.13800000 ms. mat_mul_divide cost 58.17100000 ms.
mat_mul_divide_2 cost 0.11000000 ms. mat_mul_divide_2 cost 50.59600000 ms.
mat_mul_simple cost 224639.09100000 ms.
mat_mul_simple_2 cost 71861.25800000 ms.
mat_mul_divide cost 30219.73000000 ms.
mat_mul_divide_2 cost 26061.67700000 ms.
```

(从左自右依次为32阶，256阶，2048阶)

在Project2中只用了一维数组模拟矩阵，且只实现了ikj算法，而本程序还使用了二维数组模拟矩阵，实现了ijk和ikj两种基础算法。

Project3 (ms)	32阶	256阶	2048阶
ijk (一维数组)	0.172	131.897	224639.091
ijk (二维数组)	0.142	71.558	71861.268
ikj (一维数组)	0.138	58.171	30219.730
ikj (二维数组)	0.110	50.596	26061.677

Project2 (ms)	32阶	256阶	2048阶
ikj (一维数组)	0.235	115.230	36418.200

比较两次Project，无论矩阵规模大小，仍然可以发现本程序用时显然较少，或许是因为C语言更偏向底层，抑或是gcc比g++有更多优化。

单从本次Project来看，同类矩阵模拟（即同时用一维数组或二维数组模拟）下，ijk要比ikj慢些，而且这种差异在矩阵规模增大时急剧加大，在计算2048阶矩阵时ijk已经比ikj落后一个数量级了。而用同种算法（即同时用ijk或ikj算法）下，与我预料的有所不同，竟然是二维数组的效果要好于一维数组（或许同编译器的自动优化有关）。

同时，在这部分测试中，还可以看到访存优化的重要性。同样与我预料中不一样的是，从ijk变成ikj时，对一维数组的优化要远远大于二维数组。

3.3 Strassen算法

```
strassen cost 1385.202000000 ms.
```

只有2048阶在比较时是有效的，因为当矩阵小于等于256阶时，Strassen算法将自动转为ikj算法。

对比于上面的朴素矩阵算法，strassen在大规模矩阵的用时要少得多。说明分而治之的思想在面对大规模问题时十分有效。

3.4 OpenBLAS

```
blas cost 0.76800000 ms. blas cost 33.00700000 ms. blas cost 2132.44300000 ms.
```

用时 (ms)	32阶	256阶	2048阶
OpenBLAS	0.768	33.007	2132.443

从结果我们可以看出，OpenBLAS对矩阵乘法的优化确实厉害，虽然小矩阵似乎有点落后，但对大规模的矩阵来说，用时要少很多。奇怪的是在2048阶下，OpenBLAS的 `cb1as_sgemm()` 函数表现得似乎不如Strassen算法，怀疑应该是编译器对Strassen算法有所优化。

3.5 O3

```
mat_mul_simple cost 87117.477000000 ms.
mat_mul_simple_2 cost 34128.814000000 ms.
mat_mul_divide cost 2323.600000000 ms.
mat_mul_divide_2 cost 2233.857000000 ms.
strassen cost 374.246000000 ms.
blas cost 1825.650000000 ms.
```

2048阶	ijk (一维数组)	ijk (二维数组)	ikj (一维数组)	ikj (二维数组)
开启O3 (ms)	87117.477	34128.814	2323.600	2233.857
未开启 (ms)	224639.091	71861.268	30219.730	26061.677
加速比 (%)	61.2	52.5	92.3	91.4
	Strassen	OpenBLAS		
开启O3 (ms)	374.246	1825.650		
未开启 (ms)	1385.202	2132.443		
加速比 (%)	73.0	14.4		

非常惊人！大部分速度相比未用O3时加快了差不多2倍，但是对于OPenBLAS来说加速不太多，可能是其内部原本已经加速很多了，以致于O3效果不明显。

4 困难及解决

4.1 rewind()重置文件指针

在检查矩阵列数时，发现了一个问题，就是代码看似没错的情况下竟然次次都在第一次读入就进入了EOF。

```
size_t get_cols(FILE* fp){  fp: 0x7ffbddcf90
    size_t col_cnt = 0;  col_cnt: 0
    char tmp;  tmp: -1 '\377'
    while((tmp = (char) fgetc(fp)) != '\n') {  fp: 0x7ffbddcf90
        if(tmp == ' '){  tmp: -1 '\377'
            col_cnt++;
        }
    }
    col_cnt++;
    return col_cnt;
}

size_t get_cols(FILE* fp){  fp: 0x7ffbddcf90
    rewind(fp);
    size_t col_cnt = 0;  col_cnt: 1
    char tmp;  tmp: 53 '5'
    while((tmp = fgetc(fp)) != '\n') {  tmp: 53 '5'  fp: 0x7ffbddcf90
        if(tmp == ' '){
            col_cnt++;
        }
    }
    col_cnt++;
    return col_cnt;
}
```

在查阅C语言相关资料¹⁰后，发现原来我在前一个函数 `get_rows()` 中已经将该文件指针读到了文件末尾，需要用 `rewind()` 函数将文件指针重置到文件开头。在修改后果然能运行正常了，也返回了正确的矩阵列数。

因此，之后在所有涉及文件指针操作的函数中，我都会先调用 `rewind()` 将文件指针先重置于文件开头。

4.2 矩阵元素输入/输出

一开始，对C语言的文件操作有点不熟悉，使用了 `fgetc()` 来读入矩阵的元素，但是这样会使得每一个浮点数都分裂开来（如21.3要读四次，变成 2 1 . 3 四个字符），使得数据变性，因此我改用了 `fscanf()` 来读入数据。

出于同样的原因，在输出数据时我将原本的 `fputc()` 改用了 `fprintf()`。

4.3 矩阵拷贝

4.3.1 memcpy()参数

在矩阵的拷贝函数中，我使用了 `memcpy()` 来复制储存矩阵的数组。但是在使用这个参数时，有许多需要注意的坑点。

在网上我们很容易找到 `memcpy()` 的函数原型：

```
void * memcpy ( void * destination, const void * source, size_t num);
```

需要注意的是，第三个参数 `size_t num` 所表示的不是几个元素 `elem`，而是字节数（也即 `elem*sizeof(type)`，`type`即是元素的类型）。

另外，若第一个参数 `void * destination` 所申请的内存空间字节大小小于 `size_t num`，则会报错。因此，我对矩阵的拷贝函数稍微做了调整，让目标矩阵在函数内部进行了初始化。

4.3.2 一维数组矩阵转化二维数组矩阵

在本程序中，一般使用一维数组来存储和表示矩阵，也提供了从一维数组矩阵到一维数组矩阵的拷贝函数。但是考虑到用一维数组表示矩阵不太直观，为增加可读性，故也提供了从一维数组到二维数组的拷贝。同时，其余大部分函数及操作也设计了用二维数组矩阵表示的操作。

4.3.3 C中的i++和++i的效率区别

虽然大多数情况下两个的结果都是一样的，但这两个的效率确实是有所区别。

在Stackoverflow上¹¹我找到了答案，由于 `i++` 需要有寄存器存储 `i` 的原值以便后续调用，而 `++i` 则没有这个顾虑，因此一般说来 `++i` 比 `i++` 要快些。

4.3.4 SIMD中的内存对齐

由于使用AVX指令集时，需要频繁地对内存地址进行操作。若是内存地址没有对齐，就容易使得内存操作目标发生段错误（Segmentation fault）。

```
float *v1 = (float *) malloc(l_mat->row * sizeof(float));
float *v2 = (float *) malloc(r_mat->col * sizeof(float));
```

最初的内存申请

因此，需要用到 `aligned_malloc()` 这个能对齐内存的函数（C11以来<stdlib.h>中），防止内存越界对齐。

```
float *v1 = (float *) aligned_alloc(256, l_mat->row * sizeof(float));
float *v2 = (float *) aligned_alloc(256, r_mat->col * sizeof(float));
```

更改后的内存申请

4.3.5 MinGW中无aligned_alloc()函数的问题

jetbrains的产品着实好用，代码自动补全和debug都是一等一的棒。因此很多时候我都是用clion来处理一些有bug的代码。但是在使用 `aligned_alloc()` 函数时，报了以下这个莫名其妙的错误。（该.c文件已 `#include <stdlib.h>`）

```
float *v1 = (float *) aligned_alloc(256, l_mat->row * sizeof(float));
float *v2 = (float *) aligned_alloc(256, r_mat->col * sizeof(float));
for (i = 0; i < l_mat->row; i++) {
    for (j = 0; j < r_mat->col; j++) {
        for (k = 0; k < l_mat->col; k++) {
```

Implicitly declaring library function 'aligned_alloc' with type 'void *(unsigned long long, unsigned long long)'
include the header <stdlib.h> or explicitly provide a declaration for 'aligned_alloc'
Create new function 'aligned_alloc(int,size_t)' Alt+Shift+Enter More actions... Alt+Enter

在clion中点进 `<stdlib.h>` 后我发现，竟然原来是因为该库里没有 `aligned_alloc()` 这个函数。翻遍 `<stdlib.h>` 库以后我只发现了一个比较类似的 `_aligned_malloc()`。

```
#if (!defined (_XMMINTRIN_H_INCLUDED) && !defined (_MM_MALLOC_H_INCLUDED)) || defined(_aligned_malloc)
#pragma push_macro("_aligned_free")
#pragma push_macro("_aligned_malloc")
#undef _aligned_free
#undef _aligned_malloc
_CRTIMP void __cdecl _aligned_free(void *_Memory);
_CRTIMP void __cdecl _aligned_malloc(size_t _Size, size_t _Alignment); ←
#pragma pop_macro("_aligned_free")
#pragma pop_macro("_aligned_malloc")
#endif
```

在网上搜索了一下，发现了 `_aligned_malloc()` 的定义¹²（且用该函数申请内存需要配合 `_aligned_free` 来释放内存）：

```
void * _aligned_malloc(size_t size, size_t alignment);
```

虽然与 `aligned_alloc()` 相类似:

```
void *aligned_alloc(size_t __alignment, size_t __size)
```

但是参数的位置是相反的。


令人更加奇怪的是, 按照微软的文档¹², `_aligned_malloc()` 是声明在头文件 `<malloc.h>`, 但是在我这却是声明在 `<stdlib.h>` 里, 且在未包含 `<malloc.h>` 下也成功调用了该函数。

再仔细寻找下发现, 原来是微软的MSVC不支持 `aligned_alloc()`¹³:

(同样令人感到奇怪, 因为这个文档是关于c++的, 而我的程序是.c, 且在该文档里也很清楚地写明了 `aligned_alloc()` 是自C11标准后就有的)

Memory allocation functions

C++

 Copy

```
// void* aligned_alloc(size_t alignment, size_t size); // Unsupported in MSVC
void* calloc(size_t nmem, size_t size);
void free(void* ptr);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
```

Remarks

These functions have the semantics specified in the C standard library. MSVC doesn't support the `aligned_alloc` function. C11 specified `aligned_alloc()` in a way that's incompatible with the Microsoft implementation of `free()`, namely, that `free()` must be able to handle highly aligned allocations.

最终, 我在GitHub上找到了一个更能说明原因的答案¹⁴。原来是我用的工具集合MinGW中没有 `aligned_alloc()`。换用wsl后果然问题就解决了。

4.3.6 Multiple definition

在运行程序时发现一个问题:

```
multiple definition of `start';
```

这个 `start` 变量是我拿来作计时的, 出现的原因也很简单, 就是因为我在头文件 `matrix.h` 里定义了这个变量, 而该头文件又被其他头文件调用, 最终在 `main.c` 中发生了多次重复定义相同变量。解决方法也很简单, 在头文件 `matrix.h` 中对该变量定义变为 `extern clock_t start, end;`, 并在各个使用到计时器的源文件中再次声明就好。¹⁵

4.3.7 结构体删除

由于我的结构体变量初始化不是通过 `malloc()` 等类似的函数进行的，故最后不必通过 `free()` 释放结构体，只需要释放结构体中的动态数组即可。

5 与Project2对比

在Project2中，我只实现了用一维数组存储矩阵，在本次Project中，我还使用了二维数组存储矩阵，使得部分程序可读性大大增加，同时数组的语法糖也方便我编写程序。此外，我还写了Linux系统中的内存映射读写文件（尽管未测试）、SIMD优化（尽管未完全实现）和实现了strassen算法。

对比测试数据发现，本次用C语言实现的同样的矩阵乘法算法，计算用时大大降低，且在文件IO中本程序也表现得更好了。

6 总结

矩阵乘法在许多工程计算当中都是十分重要的，如计算机视觉，神经网络等等。为了提升矩阵乘法的精度和速度，许许多多的人都在竭尽所能提升运算性能。既有从软件算法方面优化的，也有从硬件控制优化的。要想提升性能，其实两者缺一不可，必须要双管齐下。在本次Project中，既有从算法方面的优化（访存优化、Strassen算法），也有从硬件方面的优化（SIMD优化，O3优化），更有通过调用其他较为成熟的计算库来优化计算（OpenBLAS）。总的说来，在C/C++方面，仍有非常多的未知优化等待我们去探索。

-
1. 探寻C++最快的读取文件的方案 <https://byvoid.com/zhs/blog/fast-readfile/>
 2. C语言 mmap()函数(建立内存映射) 与 munmap()函数(解除内存映射) https://blog.csdn.net/qq_26093511/article/details/55102081
 3. Why does the order of the loops affect performance when iterating over a 2D array? <https://stackoverflow.com/questions/9936132/why-does-the-order-of-the-loops-affect-performance-when-iterating-over-a-2d-array>
 4. 详解矩阵乘法中的Strassen算法 <https://zhuanlan.zhihu.com/p/78657463>
 5. 算法导论-矩阵乘法-strassen算法 <https://www.cnblogs.com/zhoutaotao/p/3963048.html>
 6. C/C++: 从基础语法到优化策略(2021秋) 7.2 Speedup your program <https://www.xuetangx.com/learn/sustc08091004451/sustc08091004451/7753997/video/12724271>
 7. Intel® Intrinsics Guide https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX&ig_expand=6193,4242,4914,6804&text=%20
 8. Intel 内部指令 --- AVX和AVX2学习笔记 https://blog.csdn.net/just_sort/article/details/94393506
 9. 数值计算优化方法C/C++(三)——SIMD <https://blog.csdn.net/artorias123/article/details/86524899>
 10. 《C Primer Plus （第6版） 中文版》 第421页13.4
 11. Is there a performance difference between i++ and ++i in C? <https://stackoverflow.com/questions/24886/is-there-a-performance-difference-between-i-and-i-in-c>
 12. _aligned_malloc <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/aligned-malloc?view=msvc-160>
 13. <https://docs.microsoft.com/en-us/cpp/standard-library/cstdlib?view=msvc-160>
 14. No aligned malloc implementation on MinGW <https://github.com/ebassi/graphene/issues/83>
 15. multiple definition of xxxx问题解决及其原理 https://blog.csdn.net/mantis_1984/article/details/53571758