CS205 C/ C++ Programming - Project3

name: 廖铭骞

SID: 12012919

Part1 Analysis

文件的读取以及写入分析

C语言文件存取的原理

程序与数据的交互是以流的形式进行的，进行C语言文件的存取时，都会先打开数据流，完成操作之后便需要关闭数据流。当使用标准I/O函数（如本程序中使用的<stdio.h>头文件）进行文件读取时，系统会自动设置缓冲区，并通过数据流读写文件。即当进行文件读取时，不会直接对磁盘进行读取，而是先打开数据流，将磁盘上的信息拷贝到缓冲区内，程序再从缓冲区读取所需的数据。写入文件时同理，先将数据写入到缓冲区，在关闭数据流之后，才会将数据写入磁盘。

C语言文件（数据流）的打开与关闭

通过查看stdio.h头文件里面关于文件读取函数的内容，发现打开文件的函数原型为

```
/* Open a file and create a new stream for it.

   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern FILE *fopen (const char *__restrict __filename,
            const char *__restrict __modes) __wur;
```

该函数具有两个形参，一个是文件名，一个是打开文件的模式，返回类型是一个FILE类型的指针。在本程序中涉及到的打开文件模式有只读和只写，对应的模式为"r" 和 "w"， 在使用fopen函数之前先要声明了FILE指针指向打开的文件，如果文件打开失败则会返回空指针，所以可以根据返回的指针是否非空判定文件是否成功打开。

在操作结束之后，需要及时关闭数据流，如果没有及时关闭数据流，尽管在程序结束之后会自动关闭所有数据流，但文件打开过多会导致系统运行缓慢，这时就要自行手动关闭不再使用的文件，来提高程序整体的执行效率。关闭数据流可以通过调用 fclose函数实现。如果成功关闭，则会返回非零值。

C语言文件浮点数读写

文件的类型分为文本文件和二进制文件。由于官方给定的文件是文本文件的类。所以采用C语言当中有一种类似于scanf的方法，可以从文本文件中读取文件

```
extern int fscanf (FILE *__restrict __stream,
           const char *__restrict __format, ...) __wur;
```

可以通过传入数据流以及文件的读取格式，以及指定数据流读取去往的地址来实现文本文件数据的读取。在本程序中，由于需要进行的时候浮点数的计算，所以需要从文件当中读取浮点数，使用的格式为"%f"。类似地，文件的输出可以使用fputs函数来进行将字符串输出到文本当中。由于矩阵当中存储的是浮点数类型，所以先要将浮点数通过sprintf函数转化成字符串之后再写入到文件当中。sprintf函数原型如下：

```
/* Write formatted output to S.  */
extern int sprintf (char *__restrict __s,
            const char *__restrict __format, ...) __THROWNL;
```

以指定的格式对传入的待转化数据进行格式化之后输出到指定的字符串当中。

C语言文件读写与C++的比较

通过这两次project文件读写的实现，我体会到C++在文件读写方面的便捷性。在C++当中，对于文件的读写，只需要将`ifstream`对象或者`ofstream`对象与文件关联起来，创建流对象，便只需要直接使用该对象对文件进行操作了，这个对象已经封装好了识别读取类型的方法，只要调用该对象便会完美地匹配你的目标读取或写入类型。但在C语言当中，文件的读写都要自己去指定类型以及格式，而且在写入文件时还要将源数据流中的数据转化成为字符串才能很好地写入到文件当中。相比之下，C语言更贴近底层，采用文件指针的形式来进行文件的读写，不像C++可以直接使用封装好的方法判断文件是否成功打开等等操作。

头文件分析

为了使得整体代码的风格简洁便于调试，此次project使用了头文件来进行函数原型的声明，以及矩阵结构的实现。在本次项目中，头文件的作用主要在于使程序各部分之间保持信息的一致性。同时为防止头文件在编译程序时被重复引用，常引入#ifndef宏来进行保护，即在整个代码当中如果多次引用了该头文件，则还有一次会生效，生效后会定义一个宏变量，并在之后不会再进入被保护的代码区域。

在引用头文件的时候，产生了一个问题，什么时候需要使用引号，什么时候使用尖括号呢？通过查阅书籍，得知如果是自己编写的、非标准库的头文件，则使用引号引用，这样编译器会首先在程序的源文件中查找；如果是要引用标准库，则使用尖括号进行引用，这样引用会使编译器只在系统默认目录中查找。

矩阵的基本实现及分析

矩阵结构的声明

一个矩阵，具有的性质是它的行数（row）以及列数（col），同时一个矩阵里面的数据需要通过一个大小为数组（nums）来存储，声明代码如下：

```
struct Matrix{
    int row;
    int col;
    float * nums;
};
```

矩阵内存的分配

静态分配和动态分配的比较与选择

在程序中声明一个矩阵之后，如果需要在之后的程序中使用这个矩阵，就需要给它分配内存。而分配内存有两种选择，一种是静态分配内存，即在程序编译和链接的时候分配内存，另一种是动态分配内存，即在程序调用与执行的时候分配内存。此处尽管预先知道矩阵结构的大小，但是为了避免程序编写当中临时需要调整矩阵的结构体大小或是在程序运行的过程中导致空间的利用增大或者减少，这两种情况都将导致给矩阵静态分配内存的程序需要反复修改，又由于动态分配内存可以很好地根据程序的需要即时分配内存空间，所以在本次项目当中都将使用动态内存分配的方式对矩阵的空间进行分配。

动态分配矩阵内存的实现

在C语言当中，使用malloc函数实现内存的动态分配。通过查看该函数的原型，得知函数的返回类型是void*指针类型，所以在实际返回的时候需要使用目标的指针类型对返回的指针类型进行强制转换。malloc函数返回的是一段以字节为单位分配的一段内存块，所以需要使用一个matrix类型的指针来追踪这一段分配的内存。分配方式如下：

```
struct Matrix *mat1 = (struct Matrix*)malloc(sizeof(struct Matrix));
```

由于矩阵内部有一段指向矩阵内元素的数组还没有分配内存，所以在调用初始化矩阵方法的时候也需要为该数组动态分配内存。

```
mat -> nums = (float *)malloc(r * c * sizeof(float));//动态分配二维数组的内存
```

由于矩阵是float类型的矩阵，总共的元素个数为行数乘以列数，通过使用C语言的运算符sizeof(float)得到每一个元素所需要分配的字节数，得出给一个行数为r，列数为c的矩阵分配内存总共需要分配的字节总数为r * c * sizeof(float)

C语言中struct的使用与C++的比较

在C语言当中对于结构体无法在内部定义结构体初始化方法以及对本结构体进行操作的方法，而在C++当中可以这样实现。在C++当中，结构体相比C语言更像是一个类，里面可以实现声明内部数据以及对数据的操作。在C语言如果要初始化一个结构体中的各个数据则需要重新编写方法对其初始化，对内部数据的操作方法的编写也限制较大，这也是两种语言的不同点之一。

C语言动态分配内存与C++实现的对比

在C++当中，实现动态分配内存使用new运算符后接需要分配内存的类型即可实现内存的分配，new 运算符会自动地分配指定类型的内存，也自动计算需要分配内存的字节数目，和C++不同，C语言使用的是malloc函数，返回的是已分配内存的首地址，并没有明确的类型，所以需要使用强制类型转换获得目标类型的指针，更为底层地，C语言在动态分配内存的时候需要输出分配的字节数，由于可能存在字节计算出错的问题，所以在动态分配内存方面，使用C++编写程序也更加不容易出错。

C语言计时分析与实现

在本次project当中矩阵的乘法速度是十分重要的一项要求，要衡量矩阵的乘法速度就需要精准的计时函数来实现对矩阵乘法运行时间的测量。由于在运算小规模矩阵的时候不同算法的速度差距可能是微妙级别的，所以需要使用精度尽可能高的计时函数，此处使用头文件<sys/time.h>中的gettimeofday函数。

由于本程序是在Linux环境下运行，所以使用Linux环境下的计时函数gettimeofday()进行程序的计时，该计时函数精度较高，可以达到微秒级别。
通过查看计时函数的结构可以发现，该结构体的定义为

```
struct timeval
{
  __time_t tv_sec;      /* Seconds.  */
  __suseconds_t tv_usec;    /* Microseconds.  */
};
```

这个函数获取从1970年1月1日到现在经过的时间和时区（UTC时间），但是按照Linux的官方文档，该时区已经不再使用，所以在使用的时候传入NULL即可。

在需要计时的函数开头以及结尾分别获取一次当前时间，它们之间的差值即为程序运行的时间（微妙），显示时转化成为秒即可。
使用代码如下：

```
    gettimeofday(&read_t11, NULL);
   //function
    gettimeofday(&read_t12, NULL);
    read_time_use += (read_t12.tv_sec - read_t11.tv_sec) + (double)
(read_t12.tv_usec - read_t11.tv_usec) / 1000000.0;
```

各基本方法的分析与实现

初始化矩阵

由于需要读取文本文件对矩阵进行初始化，在初始化矩阵的方法当中，传入了一个FILE指针，该指针指向矩阵的源文本文件。

由于传入的矩阵并没有对内部的浮点数数组分配内存，所以首先需要根据传入的行数以及列数对矩阵内部元素进行初始化。在初始化之前需要对传入数据的合法性进行判断，如果行数和列数不全大于零，由于矩阵的行列数不应该为负数，所以此时不进行初始化，而是提醒输入非法并返回空的矩阵。

在分配内存完成之后按照矩阵的行和列依次读入文本文件的浮点数对矩阵内部元素进行初始化。

最后关闭文件，返回初始化完成的矩阵。

```
struct Matrix * init_file(struct Matrix* mat, int r, int c, FILE* fp){
    //输入合法性的判断
    if(r <= 0 || c <= 0){
        printf("The line or colomn input is improper!\n");
        return NULL;
    }

    //为矩阵结构各结构体变量初始化
    mat -> row = r;
    mat -> col = c;
```

```
    mat -> nums = (float *)malloc(r * c * sizeof(float));//按字节动态分配二维数组的内存
    //依次读入数据并为每一个元素赋值
    for(int i = 0; i < r; i++){
        for(int j = 0; j < c; j++){
            fscanf(fp, "%f", &(mat -> nums[i * c + j]));
        }
    };

    fclose(fp);//关闭文件
    return mat;//返回初始化完成的矩阵

}
```

复制矩阵

分析

对于一个结构体，复制的方式有两种，一种是直接使用等号进行浅拷贝，该拷贝过程按字节复制的，对于指针型成员变量只复制指针本身，而不复制指针所指向的目标，所以在复制的时候会导致两个结构体的指针指向同一块内存。在本程序中，显然使用浅拷贝无法获得两个相互无关但是内容一样的矩阵结构，所以需要使用深拷贝。

实现

为了使用深拷贝，且需要避免返回指向被调用函数内部自动变量的指针的问题（在Part4 有详细的说明），事先需要先新创建一个矩阵，这个创建矩阵的函数不需要对矩阵内部数组的元素进行赋值。

```
struct Matrix* init(struct Matrix* mat, int r, int c){
    if(r <= 0 || c <= 0){
        printf("The line or colomn input is improper!\n");
        return NULL;
    }

    //为矩阵结构各结构体变量赋值
    mat -> row = r;
    mat -> col = c;
    mat -> nums = (float *)malloc(r * c * sizeof(float));//动态分配二维数组的内存
    return mat;
}
```

之后同时传入新建矩阵以及源矩阵，对源矩阵内数组的每一个元素都逐一复制到新建矩阵当中，这样就可以得到两个值完全一样但是内存上毫无关联的矩阵了。实现代码如下：

```
bool cpyMatrix(struct Matrix *target, struct Matrix *src){
    if(src == NULL || src -> row <= 0 || src -> col <= 0){
        return false;
    }

    int row = src -> row;
```

```
    int col = src -> col;

    if(target == NULL && target -> row != row || target -> col != col){
        delMatrix(target);
        struct Matrix *tar;
        tar = (struct Matrix*)malloc(sizeof(struct Matrix));
        tar -> row = row;
        tar -> col = col;
        tar -> nums = (float*)malloc(row * col * sizeof(float));
        target = tar;
    }


    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            target -> nums[i * col + j] = src -> nums[i * col + j];
        }
    }
    return true;
}
```

释放矩阵

回收矩阵内存必要性分析

C语言当中局部变量在函数调用结束内存便会被自动释放，与局部变量不同，使用malloc动态分配的空间，它会存储在堆当中，如果没有及时释放，则该内存会一直被占用直到程序结束。

由于本程序当中矩阵的规模可能很大，需要占用的内存也很多，特别是在矩阵乘法优化所使用的Strassen算法当中需要为大量的矩阵分配内存，如果不及时回收内存，则在该函数调用完毕后会导致大量的内存泄漏，对程序本身运行存在极大的不利。所以需要及时地进行矩阵内存的回收。

释放矩阵函数实现

由于矩阵本身是通过结构体来实现的，所以在释放矩阵结构体的时候需不需要再重复释放内部的元素呢？即释放结构体内存的同时会不会连同结构体内所有的元素都同时释放呢？为此进行以下实验：

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

struct Matrix{
    char *nums;

    char arr[10];
};
```

```
int main(){
    struct Matrix *mat=(struct Matrix *)malloc(sizeof(struct Matrix));

    mat->nums="LONDON";

    strcpy(mat->arr,"mat");

    char *p1=mat->nums;
    char *p2=mat->arr;

    printf("nums=%s pointer of nums%p  arr=%s\n",mat->nums,mat->nums,mat->arr);

    free(mat);

    printf("nums=%s pointer of nums%p arr=%s\n",p1,p1,p2);
    return 0;
}
```

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/project3$ ./test
nums=LONDON pointer of nums0x7f61291dc008  arr=mat
nums=LONDON pointer of nums0x7f61291dc008 arr=
```

该实验结果表明，在释放结构体指针指向的内存的时候，即使调用free方法将该结构体的内存释放了，结构体内部的指针指向的内存依然没有被释放。

根据以上的实验结果，对于结构体内部的指针，在释放结构体之前要先进行内部指针指向内存的释放。

即使指针指向的内存已经释放，但是此时指针依然指向的是原有内存，但是此时内存已经不可再通过原指针被使用，所以原有的指针应该设置为指向空值NULL。这样子就可以保证矩阵结构已经完全被释放了。

释放矩阵函数的代码如下所示：

```
bool delMatrix(struct Matrix* mat){
    if(mat == NULL){
        return false;
    }

    free(mat -> nums);
    //即使内存已经释放，但指针仍然指向原有内存，应该置零
    mat -> nums = NULL;
    free(mat);
    mat = NULL;
    return true;

}
```

矩阵加法与减法（辅助函数）

为了便于一些特殊的矩阵乘法实现地更加便捷，在本程序中实现了两种矩阵的运算，分别是矩阵加法以及减法，两种方法实现都是通过将两个运算矩阵的对应位置进行运算之后加到传入的目标矩阵当中。由于是直接通过指针进行操作，这两种轻量级的方法在面对大规模输入矩阵的时候避免了大段地复制结构体。

字符串分割实现矩阵规模获取的函数（辅助函数）

由于矩阵乘法需要正确的矩阵规模用以初始化矩阵等操作，所以获取矩阵的规模至关重要。在本次项目当中，观察到所给的文本文件命名都有统一的规范，所以可以通过读取文件名来获取矩阵的规模。

可以通过指定分隔符（程序中为"-"号）来对文件名进行字符串分割，使用`strtok`函数实现该过程，通过查看其函数原型，可以发现通过传入一个字符串实参和分割符字符串实参，该函数可以返回一个分割之后的字符串，

```
extern char *strtok (char *__restrict __s, const char *__restrict __delim)
     __THROW __nonnull ((2));
```

以文件名`mat-A-32.txt`为例，我们需要获取以 "-"号分割的第三个字符串，所以对分割次数进行计数，当分割了两次的时候停止操作，返回当前得到的字符串，并将其通过C语言的函数`atoi`转成整数类型进行返回，这样就实现了通过文本文件名来获取矩阵的规模。

实现代码如下：

```
int matSize(char str[]){
    char *delim = "-.";
    char *p;
    strtok(str, delim);
    int cnt = 0;

    while ((p = strtok(NULL, delim))){
        if(cnt == 1){
            break;
        }
        cnt++;
    }
    return atoi(p);
}
```

矩阵乘法以及优化分析

朴素矩阵乘法

根据矩阵乘法定义：
假设矩阵A的大小为 m × n，矩阵B的大小为n × q则，两个矩阵相乘结果矩阵C的大小为 m × q，两个矩阵相乘的公式为：

$$C_{iq} = \sum_{k=1}^{n} A_{ik} B_{kq}$$

首先是采用朴素的矩阵乘法函数根据上述的公式直观地实现，使用三层循环遍历将两个矩阵相乘：

```
for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            for(int k = 0; k < row; k++){
                tar -> nums[i * col + j] += mat1 -> nums[i * col + k] * mat2 ->
nums[k * col + j];
            }
        }
    }
```

访存优化

进一步分析，根据课上学习的内容，数组是一段连续的内存，可以发现在运算的过程当中mat2[k * col + j]在读取内存数据的时候是不连续的，在二维数组当中，mat2[k * col +j]是按照列读取的顺序来读取的，这样会使得其在内存中不断地“跳跃”，引起访问效率的降低，这种效率的降低当输入的矩阵规模非常大的时候是非常显著的，而且在数据量很大的情况下会消耗巨大的读写内存，这种显著的性能降低情况会在之后的报告中给出具体的说明。

要同时实现顺序访问mat1以及mat2，又要满足下列公式

$$C_{iq} = \sum_{k=1}^{n} A_{ik} B_{kq}$$

所以可以通过改变循环的顺序来减少内存“跳跃”的次数，实现初步的优化：

此时将i的循环放在最外层，将k放在第二层进行循环，将 j的循环放在最内层这样的循环代码如下所示：

```
for(int i = 0; i < row; i++){
        for(int k = 0; k < row; k++){
            float tmp = mat1 -> nums[i * col + k];
            for(int j = 0; j < col; j++){
                tar -> nums[i * col + j] += tmp * mat2 -> nums[k * col + j];
            }
        }
    }
```

经过与标准三层循环程序运行三种矩阵规模结果的比对，结果完全一致，可以得到两种算法的正确性一致的结论。
并可以明显地感觉到运算速度的加快，详细的对比结果将在下文给出。

分块优化

借鉴在归并排序算法中的思想，矩阵的乘法也可以分而治之，通过对矩阵进行拆解，分别运算之后合并，这样计算从计算机层面思考一方面可以使得CPU更加快速地将矩阵从内存中搬运出来进行运算，同时在另一方面，运算更小规模的矩阵的时间也会显著减少，两方面结合，对矩阵进行分块运算可以加快运算的速度以及提高内存读取的效率。

代码实现如下所示，主要思想是把一个大矩阵分成`block_size`份，分别对每一部分进行运算之后进行合并：

```
int block_size = 4;
    for (int block_k = 0; block_k < size1; block_k += block_size) {
        for (int block_i = 0; block_i < size1; block_i += block_size) {
            for (int block_j = 0; block_j < size1; block_j += block_size) {
                for (int k = block_k; k < min(block_k + block_size, size1); k++) {
                    for (int i = block_i; i < min(block_i + block_size, size1);
i++) {
                        for (int j = block_j; j < min(block_j + block_size, size1);
j++) {
                            res[i * col + j] += mat1[i * col + k] * mat2[k * col +
j];
                        }
                    }
                }
            }
        }
    }
```

但是代码速度依然没有比一般的矩阵乘法要快，于是推测矩阵分块的数量对矩阵乘法速度也会有影响，因为分块之后的矩阵规模对单块矩阵的乘法运算会有影响，如果分块过大，则不能很快地进行单块矩阵的运算，另一方面，如果分块过小，虽然运算单个矩阵的速度变快，但是进行矩阵分块以及合并的过程会消耗大量的时间，两个方面都会不利于矩阵的运算，所以要选择合适的矩阵分块值以达到最好的分块矩阵乘法的运算效率，进一步分析原因，可能是对矩阵进行分块的时候，分得的内存不连续导致运算速度也有所减慢。

Strassen算法

基本思想以及正确性分析

进一步分析，要加快程序运算大规模矩阵时的速度，可以使用分块的思想，在矩阵规模较大的情况下，先花费一定时间对大规模矩阵进行分块操作，之后对这些小块进行乘法运算，最后将运算得到的结果进行合并处理，这种分而治之的思想在矩阵乘法当中的体现就是Strassen算法。

Strassen算法主要使用了分而治之的思想，通过设定基准条件（base case），即矩阵规模小于64*64的时候直接调用访存优化矩阵乘法进行结果返回。也设置了递归条件，通过将每次递归调用自身时传入的size参数为当前的size的一半来使得每次进行递归的结果都往基准条件的方向靠拢，这样也保证了递归一定能够结束，进而保证了算法的正确性。

算法步骤以及细节分析

由于官方所给的样本文件当中矩阵的规模都是2的幂，所以可以每次将一个大规模的矩阵分成4个 $n/2 \times n/2$的子矩阵进行运算。

从宏观上看，Strassen算法主要分为4个步骤：

1. 首先是将输入矩阵mat1 和 mat2以及输出矩阵tar分解成 $n/2 \times n/2$的子矩阵
2. 其次是创建两个结果矩阵记录需要记录矩阵乘法结果的步骤产生的中间矩阵。
3. 使用步骤一产生的子矩阵和步骤二产生的结果矩阵递归调用Strassen算法计算出7个矩阵的积$P_1, P_2, \ldots, P_7$。注意每一个矩阵都是$n/2$的
4. 通过步骤三产生的七个矩阵进行矩阵基本的加减运算计算出结果矩阵tar的四个子矩阵，最后进行合并组合成为矩阵tar进行返回。

使用Strassen方法计算较大规模矩阵的时候速度会有显著的提升，因为在一般的分块矩阵乘法运算当中，即使进行了分块将两个输入矩阵分解成为4个规模为原矩阵一半的矩阵如下所示，进行合并运算的时候依然需要进行如下所示的运算。

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

依然需要8次矩阵乘法以及4次的加法，即递归地调用8次规模为$n/2 \times n/2$的分块乘法。

而使用Strassen算法，在步骤3的时候只需要进行7次$n/2 \times n/2$的乘法。减少了一次矩阵乘法，这在处理规模非常的的矩阵乘法时节省的时间是非常多的，这在以下的报告当中会有详细的说明。

此处需要注意的是，由于Strassen算法需要初始化大量的子矩阵来进行矩阵运算或是存储矩阵运算结果，在这过程中尤其是创建子矩阵的过程需要复制比较多的数据，所以如果创建矩阵以及复制所消耗的时间超出Strassen算法的时间收益时，在本程序中限定是在矩阵规模小于64时，就使用普通的经过访存优化的矩阵乘法，这样就在矩阵规模非常小的时候也保证了算法的高效性。

同样地，由于在该算法当中新建了大量的矩阵结构，在递归调用该函数的时候，函数结束时也并不会释放占用的内存，所以如果不及时地进行矩阵结构内存的释放，则会导致严重的内存泄漏。所以在Strassen方法的最后都会调用释放矩阵的方法对矩阵结构进行彻底地释放，防止内存泄漏。

具体实现

Strassen函数的代码如下所示：

```
bool mulMatrix_strassen(struct Matrix *tar, struct Matrix *mat1, struct Matrix
*mat2, int size){

    int row = mat1 -> row;
    int col = mat1 -> col;
    int newSize = size / 2;


    if(tar == NULL || tar -> row != row || tar -> col != col){
        delMatrix(tar);
        struct Matrix *tar_ex;
        tar_ex = (struct Matrix*)malloc(sizeof(struct Matrix));
        tar_ex -> row = row;
        tar_ex -> col = col;
        tar_ex -> nums = (float*)malloc(row * col * sizeof(float));
        tar = tar_ex;
    }

    if(row <= 32){
        mulMatrix_ikj(tar, mat1, mat2);
        return true;
    }

    struct Matrix* A11 = (struct Matrix*) malloc(sizeof(struct Matrix));
    struct Matrix* A12 = (struct Matrix*) malloc(sizeof(struct Matrix));
    struct Matrix* A21 = (struct Matrix*) malloc(sizeof(struct Matrix));
    struct Matrix* A22 = (struct Matrix*) malloc(sizeof(struct Matrix));

    init(A11, newSize, newSize);
    init(A12, newSize, newSize);
    init(A21, newSize, newSize);
    init(A22, newSize, newSize);


    struct Matrix* B11 = (struct Matrix*) malloc(sizeof(struct Matrix));
    struct Matrix* B12 = (struct Matrix*) malloc(sizeof(struct Matrix));
    struct Matrix* B21 = (struct Matrix*) malloc(sizeof(struct Matrix));
    struct Matrix* B22 = (struct Matrix*) malloc(sizeof(struct Matrix));

    init(B11, newSize, newSize);
    init(B12, newSize, newSize);
    init(B21, newSize, newSize);
    init(B22, newSize, newSize);
```

```
struct Matrix* C11 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* C12 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* C21 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* C22 = (struct Matrix*) malloc(sizeof(struct Matrix));

init(C11, newSize, newSize);
init(C12, newSize, newSize);
init(C21, newSize, newSize);
init(C22, newSize, newSize);

struct Matrix* resA = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* resB = (struct Matrix*) malloc(sizeof(struct Matrix));

init(resA, newSize, newSize);
init(resB, newSize, newSize);


struct Matrix* P1 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* P2 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* P3 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* P4 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* P5 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* P6 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* P7 = (struct Matrix*) malloc(sizeof(struct Matrix));

init(P1, newSize, newSize);
init(P2, newSize, newSize);
init(P3, newSize, newSize);
init(P4, newSize, newSize);
init(P5, newSize, newSize);
init(P6, newSize, newSize);
init(P7, newSize, newSize);

struct Matrix* S1 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S2 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S3 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S4 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S5 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S6 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S7 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S8 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S9 = (struct Matrix*) malloc(sizeof(struct Matrix));
struct Matrix* S10 = (struct Matrix*) malloc(sizeof(struct Matrix));

init(S1, newSize, newSize);
init(S2, newSize, newSize);
init(S3, newSize, newSize);
init(S4, newSize, newSize);
init(S5, newSize, newSize);
init(S6, newSize, newSize);
init(S7, newSize, newSize);
init(S8, newSize, newSize);
init(S9, newSize, newSize);
init(S10, newSize, newSize);
```

```
    //对矩阵mat1 和 mat2进行分割
    for(int i = 0; i < newSize; i++){
        for(int j = 0; j < newSize; j++){
            A11 -> nums[i * newSize + j] = mat1 -> nums[i * col + j];
            A12 -> nums[i * newSize + j] = mat1 -> nums[i * col + j + row / 2];
            A21 -> nums[i * newSize + j] = mat1 -> nums[(i + newSize) * col + j];
            A22 -> nums[i * newSize + j] = mat1 -> nums[(i + newSize) * col + j +
newSize];

            B11 -> nums[i * newSize + j] = mat2 -> nums[i * col + j];
            B12 -> nums[i * newSize + j] = mat2 -> nums[i * col + j + row / 2];
            B21 -> nums[i * newSize + j] = mat2 -> nums[(i + newSize) * col + j];
            B22 -> nums[i * newSize + j] = mat2 -> nums[(i + newSize) * col + j +
row / 2];
        }
    }

    //bool mat_add(struct Matrix * res, struct Matrix *mat1, struct Matrix* mat2,
int size);
    //bool mat_sub(struct Matrix * res, struct Matrix * mat1, struct Matrix * mat2,
int size);
    //bool mulMatrix_strassen(struct Matrix *tar, struct Matrix *mat1, struct
Matrix *mat2, int size);
    /**/
    mat_sub(S1, B12, B22, newSize);

    mat_add(S2, A11, A12, newSize);

    mat_add(S3, A21, A22, newSize);

    mat_sub(S4, B21, B11, newSize);

    mat_add(S5, A11, A22, newSize);

    mat_add(S6, B11, B22, newSize);

    mat_sub(S7, A12, A22, newSize);

    mat_add(S8, B21, B22, newSize);

    mat_sub(S9, A11, A21, newSize);

    mat_add(S10, B11, B12, newSize);


    mulMatrix_strassen(P1, A11, S1, newSize);

    mulMatrix_strassen(P2, S2, B22, newSize);

    mulMatrix_strassen(P3, S3, B11, newSize);

    mulMatrix_strassen(P4, A22, S4, newSize);
```

```
    mulMatrix_strassen(P5, S5, S6, newSize);

    mulMatrix_strassen(P6, S7, S8, newSize);

    mulMatrix_strassen(P7, S9, S10, newSize);

    mat_add(resA, P4, P5, newSize);

    mat_sub(resB, resA, P2, newSize);

    mat_add(C11, resB, P6, newSize);

    mat_add(C12, P1, P2, newSize);

    mat_add(C21, P3, P4, newSize);

    mat_add(resA, P5, P1, newSize);

    mat_sub(resB, resA, P3, newSize);

    mat_sub(C22, resB, P7, newSize);


    for(int i = 0; i < newSize; i++){
        for(int j = 0; j < newSize; j++){
            tar -> nums[i * col + j] = C11 -> nums[i * newSize + j];
            tar -> nums[i * col + j + row / 2] = C12 -> nums[i * newSize + j];
            tar -> nums[(i + row / 2) * col + j] = C21 -> nums[i * newSize + j];
            tar -> nums[(i + row / 2) * col + j + row / 2] = C22 -> nums[i *
newSize + j];
        }
    }


    delMatrix(A11);
    delMatrix(A12);
    delMatrix(A21);
    delMatrix(A22);

    delMatrix(B11);
    delMatrix(B12);
    delMatrix(B21);
    delMatrix(B22);

    delMatrix(C11);
    delMatrix(C12);
    delMatrix(C21);
    delMatrix(C22);

    delMatrix(P1);
    delMatrix(P2);
    delMatrix(P3);
    delMatrix(P4);
    delMatrix(P5);
```

```
    delMatrix(P6);
    delMatrix(P7);

    delMatrix(S1);
    delMatrix(S2);
    delMatrix(S3);
    delMatrix(S4);
    delMatrix(S5);
    delMatrix(S6);
    delMatrix(S7);
    delMatrix(S8);
    delMatrix(S9);
    delMatrix(S10);

    delMatrix(resA);
    delMatrix(resB);

    return true;

}

bool mat_add(struct Matrix * res, struct Matrix *mat1, struct Matrix* mat2, int
size){

    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            res -> nums[i * (res -> col) + j] = mat1 -> nums[i * (mat1 -> col) + j]
+ mat2 -> nums[i * (mat2 -> col) + j];
        }
    }
    return true;
}

bool mat_sub(struct Matrix * res, struct Matrix * mat1, struct Matrix * mat2, int
size){
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            res -> nums[i * (res -> col) + j] = mat1 -> nums[i * (mat1 -> col) + j]
- mat2 -> nums[i * (mat2 -> col) + j];
        }
    }
    return true;
}
```

OpenBLAS 库的导入与使用

首先了解到BLAS(Basic Linear Algebra Subprograms)是一个应用程序接口标准，用以规范发布基础线性代数操作的数值库，而OpenBLAS是BLAS标准的一种基本实现，被广泛应用于科学计算，数据分析，深度学习算法等领域。

OpenBLAS库导入

要在Linux环境下下载并编译安装OpenBLAS库以便于之后的调用。为了导入OpenBLAS库，依次执行下列的命令。

1. 首先是通过命令`cd ~`切换到根目录
2. 之后使用`git clone git://github.com/xianyi/OpenBLAS.git`命令从GitHub上面拷贝一个 Git 仓库到本地，让自己能够查看该项目，或者进行修改。
3. 切换到OpenBLAS路径 `cd OpenBLAS`
4. 使用`sudo make install`将OpenBLAS库安装到默认的文件夹/opt/OpenBLAS当中

使用OpenBLAS库进行矩阵乘法运算

通过包含OpenBLAS实现了矩阵乘法函数的头文件<cblas.h>，并通过在`main`函数中使用`cblas_sgemm`进行传入矩阵的运算。该函数需要14个参数，这14个参数分别描述了相乘的矩阵是否要进行转置，以及矩阵的行和列等基本信息。

Cmake 的编写以及分析

Cmake是一个开源、跨平台的工具家族，用于构建、测试和打包软件。首先要指定Cmake的最低支持版本，通过以下命令查看本机的cmake版本，为3.16版本

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/project3$ cmake --version
cmake version 3.16.3
```

所以指定的是3.16版本。之后是要创建project的名字，此处名字为`matmul`，之后由于要使用到不属于当前目录的头文件，所以需要进行头文件的目录的添加，进而需要添加需要链接的头文件的目录以及名称。综合起来的CmakeLists.txt的代码如下所示：

```
cmake_minimum_required(VERSION 3.16)

project(matmul)

include_directories(/opt/OpenBLAS/include)

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

add_executable(matmul matmul.c mat.c)
```

之后只要运行`cmake .`就可以自动实现MakeFile的生成，之后可以直接通过用make命令编译源码生成可执行程序，自动实现各文件的链接以及动态库的调用，在处理一个包含有多级文件的C语言或者C++大工程的时候，将会大大地提高管理代码的效率。

同时由于运行cmake的时候cmake会检测内部文件的最新更改时间，如果没有文件在上次运行cmake命令之后被更改，则cmake将不会重新修改原来生成的makefile文件，否则会重新更新该文件，这样的机制也提高了cmake的编译效率。

程序的鲁棒性

文件读写的鲁棒性

由于使用的是fopen方法进行文件的读取，在文件读取成功的时候会返回该文件对应的FILE类型指针，如果打开失败则返回NULL。所以在本程序当中定义了FILE类型的指针变量，保存该函数的返回值，并在打开文件之后进行返回值的判断，根据返回值是否为空来判断文件是否成功打开。如果该指针是空值，则会报告文件打开异常之后便会退出程序。

这一操作保证了程序中的矩阵始终正常地存取了文本文件当中的浮点数，不然在接下来的操作中对于文件的操作是基于一个空指针来操作，极其容易导致程序崩溃。

矩阵初始化鲁棒性

在矩阵初始化的过程中，调用矩阵初始化方法并通过传入矩阵的行数以及列数对矩阵本身以及内部元素进行初始化。但是如果输入的行数或者列数为空则会导致矩阵不能正常初始化，此时该函数将提示错误信息并返回一个空值表示无法创建行数或者列数小于零的矩阵。

矩阵操作函数的鲁棒性

在复制矩阵或者是进行矩阵乘法运算的时候，容易出现输出的目标矩阵为空的情况，或者是该矩阵的大小与源矩阵不一致，此时在本程序当中，如果出现上述情况，首先将会通过指向原矩阵的指针将原内存块进行释放，之后重新初始化一个新的矩阵，将新矩阵的规模设置为与源矩阵一致，在将原指针指向这一块新的内存块，实现矩阵的纠正。这样保证了矩阵操作或是运算的合理性与可行性，使得程序更加健壮。

比较时间差距使用的方法

在比较不同算法以及优化策略时，对于随着矩阵规模的增大各算法时间的相对差距需要定义一个公式来进行对两组时间统一地比较，这里采用如下的公式进行比较：

$$\mathrm{E}=\frac{T_{long}-T_{short}}{T_{short}}\times 100\%(T_{long}\text{代表用时长的一组，}T_{short}\text{代表用时短的一组})$$

E的值越大，反映出两组所用时间的差距就越大，差距越明显，在后续的时间比较当中使用的都是这一公式。

Part2 Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <cblas.h>
#include "mat.h"
#pragma GCC optimize(2)//编译优化
```

```c
int main(int argc, char * argv[]){

    if(argc < 4){
        printf("The file cannot be received normally, please enter files
properly\n");
        exit(0);
    }


    char *ch1 = (char*) malloc(10 * sizeof(char));
    char *ch2 = (char*) malloc(10 * sizeof(char));

    strcpy(ch1, argv[1]);
    strcpy(ch2, argv[2]);

    int size1 = matSize(ch1);
    int size2 = matSize(ch2);


    if(size1 != size2){
        printf("The size of matrix 1 and the size of matrix 2 are not the
same\nPlease check again.\n");
        exit(0);
    }

    struct Matrix *mat1 = (struct Matrix*)malloc(sizeof(struct Matrix));
    struct Matrix *mat2 = (struct Matrix*)malloc(sizeof(struct Matrix));


    FILE *fp1, *fp2;

    fp1 = fopen(argv[1], "r");

    if(fp1 == NULL){
        printf("Error in opening the first file\n");
        exit(1);
    }else{
        init_file(mat1, size1, size1, fp1);
    }



    fp2 = fopen(argv[2], "r");

    if(fp2 == NULL){
        printf("Error in opening the second file\n");
        exit(1);
    }else{
        init_file(mat2, size1, size1, fp2);
    }


    struct Matrix *res = (struct Matrix*) malloc(sizeof(struct Matrix));
    res = init(res, mat1 -> row, mat1 -> col);
```

```
    // printf("res -> row = %d, res -> col = %d\n", res -> row, res -> col);
    struct timeval start, end;
    gettimeofday(&start, NULL);

    // mulMatrix_kij(res, mat1, mat2);

    // mulMatrix_ikj(res, mat1, mat2);
    // mulMatrix_tradition(res, mat1, mat2);
    // mulMatrix_block(res, mat1, mat2);

    mulMatrix_strassen(res, mat1, mat2, mat1 -> row);
    // cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, mat1 -> row, mat2 -> col, mat1 -> col, 1.0f, mat1 -> nums, mat1 -> col, mat2 -> nums, mat2 -> col, 0, res -> nums, res -> col);

    gettimeofday(&end, NULL);long timeuse =1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("The matrix multiplication used %lfs\n" , timeuse / 1000000.0);



    FILE *fout = fopen(argv[3], "w");
    if(fout == NULL){
        printf("Cannot create a file named: %s\n", argv[3]);
        exit(1);
    }
    int row = res -> row;
    int col = res -> col;
    char *str_res = (char*) malloc(10 * sizeof(char));
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            sprintf(str_res, "%.6f", res -> nums[i * col + j]);
            fputs(str_res, fout);
            fputc(' ', fout);
        }
        fputs("\n", fout);
    }

    fclose(fout);


    if(delMatrix(mat1)){
        printf("matrix1 has been successfully deleted\n");
    }

    if(delMatrix(mat2)){
        printf("matrix2 has been successfully deleted\n");
    };

    if(delMatrix(res)){
        printf("result matrix has been successfully deleted\n");
    }

    return 0;
}
```

Part3 Result and Verification

随机生成浮点数矩阵

由于在本次实验当中需要对不同规模的矩阵进行乘法时间的测量，在这个时间的比较当中，单单凭借官方所给的三种规模的矩阵不足以准确地反映不同算法对于不同规模的矩阵的最佳使用情况，所以需要通过随机生成不同规模的矩阵来增加采样范围，使得实验结果更具说服力。

随机数生成矩阵代码如下：

```
void random(struct Matrix* mat){
    int r = mat -> row;

    srand((unsigned)time(NULL));
    for(int i = 0; i < r; i++){
        for(int j = 0; j < r; j++){
            float value = (rand() % (100 * 10 - 1)) / 10.0;
            mat -> nums[i * r + j] = value;
        }
    }

}
```

通过传入矩阵的规模，在函数内部随机生成1~100的浮点数，并写入到传入的矩阵当中。

最终，在本次项目当中，另外生成了规模分别为512 * 512， 1024 * 1024 以及 4096 * 4096 的三个矩阵。

访存硬件级优化对矩阵乘法速度对比以及分析

由于数组是一段连续的内存，在本程序当中由于是使用一维数组来表示二维数组，通过模拟二维数组当中的行之间的跳转访问操作来实现。可以发现在运算的过程当中`mat2[k * col + j]`在读取内存数据的时候是不连续的，在二维数组当中，mat2[k * col +j]是按照列读取的顺序来读取的，这样会使得其在内存中不断地“跳跃”，引起访问效率的降低。下面进行实验探究在不同矩阵规模下通过改变内存访问的顺序对矩阵乘法速度的影响。

下面是三种访存优化方法矩阵的运算时间（由于使用朴素矩阵乘法运算4096 * 4096速度过慢，故不进行运算时间）：

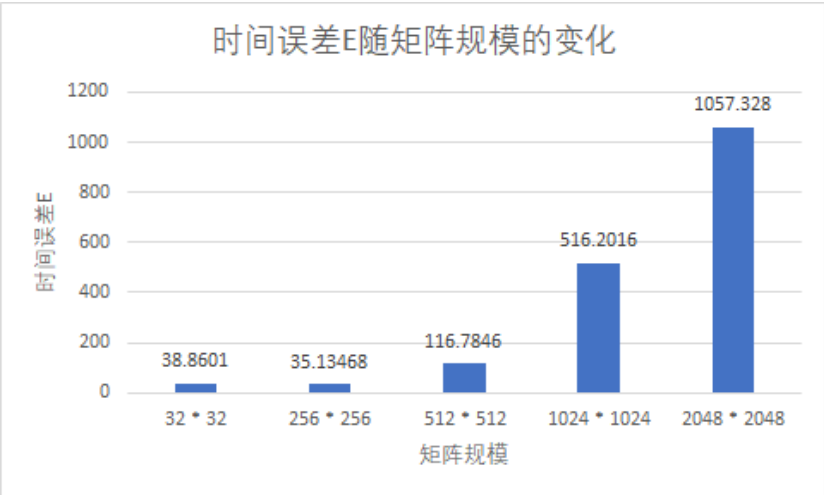
矩阵规模	32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048	4096 * 4096
ijk时间/s	0.000268	0.088847	0.988245	24.298455	375.722021	无
ikj时间/s	0.000193	0.065747	0.455865	3.943264	32.464609	260.30
kij时间/s	0.00014	0.063093	0.4585	4.285995	33.03083	261.149

可以看出，更改了矩阵乘法顺序的位置ikj和kij方法之间没有太多的区别，故选择ikj与朴素ijk顺序的乘法进行时间上的对比

通过绘制蝴蝶图，可以发现随着矩阵规模的增大，使用朴素矩阵乘法和使用访存优化的矩阵乘法速度差距也会越来越大，具体对比如图所示：

ijk时间/s	矩阵规模	ikj时间/s
0.000268	32 * 32	0.000193
0.088847	256 * 256	0.065747
0.988245	512 * 512	0.455865
24.29846	1024 * 102	3.943264
375.722	2048 * 204	32.46461

对两组时间进行比较，结果如下所示：



可以发现随着矩阵规模的增长，内存的访问顺序会在很大程度上限制矩阵乘法的速度，时间误差大致呈指数指数变化的关系，最后在运算通过优化矩阵乘法中的内存访问顺序可以很好地改善矩阵乘法的效率。

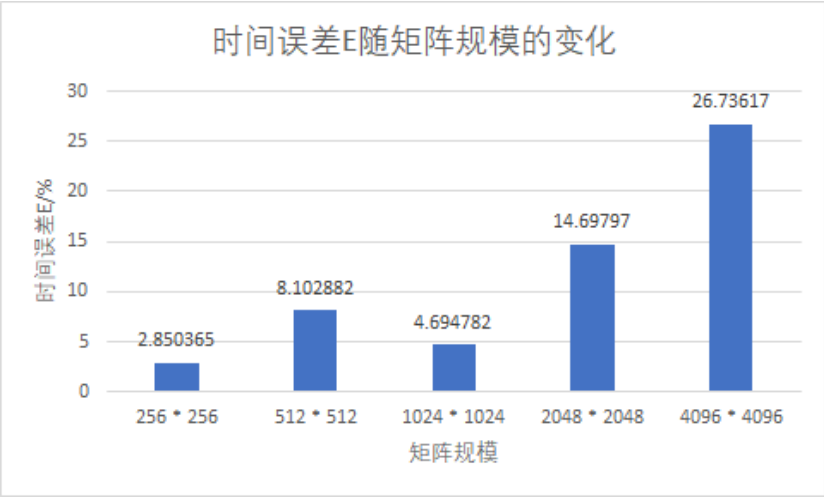
分析其中原因，结合在矩阵乘法运算当中对内存的访问顺序，在朴素的矩阵乘法当中，对于最内层的矩阵，由于最内层的变量k代表的是第二个矩阵的行数，所以对于每一次内层循环当中k的变化，程序会跳转地访问第二个矩阵的内存，在矩阵规模非常大的时候，矩阵的列数也会非常大，此时跳转访问内存的效率就会变得非常低。在最底层循环将原本访问`mat2`数组元素的过程从依次跳过`Column`个元素进行访问，变成了连续对`mat2`元素进行访问，在kij顺序当中尽管并不是连续地对`mat1`进行访问但是在矩阵规模很大的情况下，对`mat1`的访问远远没有对`mat2`的访问那么频繁，所以通过改变使用最频繁的矩阵内存的访问顺序可以在很大程度上提高矩阵运算的效率。进而使用ikj使得能在两种矩阵都同时满足顺序访问的需求，所以相比kij顺序的矩阵乘法，ikj顺序的表现也更加优秀。

使用Strassen算法释放子矩阵空间对矩阵乘法速度影响对比及分析

通过使用分块的思想进行Strassen算法的编写，涉及到通过空间换时间的问题，在算法运行的过程中需要递归地创建非常多的规模比原矩阵小的矩阵，以此来减少乘法所需的次数，考虑到在使用完这些矩阵之后对它们的空间进行回收也会消耗大量的时间，于是考虑将释放空间所用的时间与不对使用完的空间进行释放运行速度进行比较，探究使用过多的内存是否会拖慢程序运行的速度。实验结果如下：

释放空间	矩阵规模	未释放空间
0.000253	32 * 32	0.000162
0.056905	256 * 256	0.058527
0.367684	512 * 512	0.397477
2.768627	1024 * 102	2.898608
18.54047	2048 * 204	21.26554
131.021	4096 * 409	166.051

为了更加直观地看出内存泄漏在矩阵规模很大的时候对程序的拖累程度，做出时间误差随着矩阵规模变化的图像如下图：



可以看出，由于在矩阵规模较小的情况下调用的是朴素矩阵乘法，不涉及到Strassen算法的内容，所以矩阵规模从256 * 256 开始增长。可以看出，随着矩阵规模的增大，不及时回收子矩阵导致的内存泄漏对程序运行会有越来越显著的拖累，导致矩阵乘法会越来越慢。

分析其中原因，一方面是不释放空间会导致系统在程序运行的时候不断地寻找可使用的空间会耗费时间，且随着矩阵规模的增长，被占用的空间会呈指数级增长，这时候系统寻找可用空间耗费的时间也会越来越多，拖慢程序的运行。另一方面，由于不同规模的矩阵结构内部数组存储的元素数量也不同，在CPU分配内存空间的时候会涉及到数据的截取以及拼接问题，当占用的内存非常大而且未被释放时，就会使得CPU为不同规模的矩阵分配空间时数据的截取以及拼接操作会更加地困难，使得CPU运行效率变低，进而使程序运行速度更加慢。

由此也可以看出在使用完一块内存空间之后及时进行释放，防止内存泄漏对内存分配需求很高的程序的运行速度提升有很大的作用。

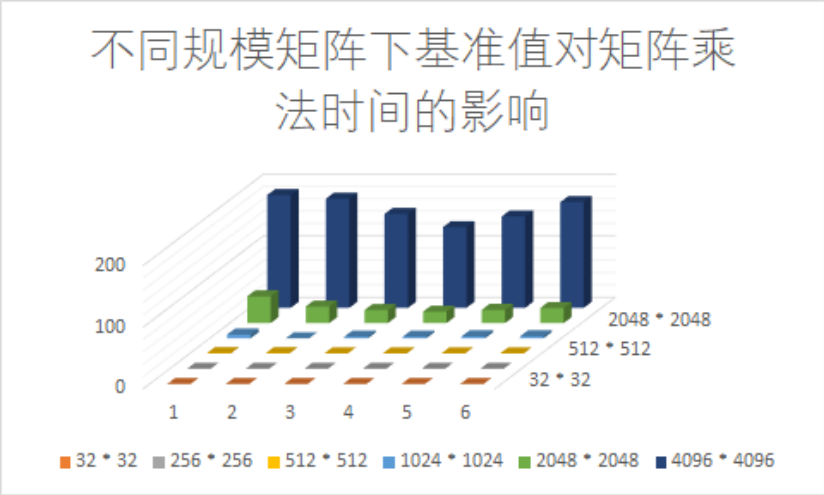
使用Strassen 算法递归基本条件值设置对矩阵乘法速度影响对比以及分析

在上面的实验可以看出，Strassen算法需要递归地动态分配非常多子矩阵的内存，在矩阵规模较小的情况下，直接进行运算的时间反而会少于分块进行计算的时间，所以考虑在Strassen算法和访存优化算法在运行效率上的取长补短。这一实验可以通过改变Strassen算法递归的基准条件进行。试验的结果如下表所示：

矩阵规模/基准值	4	8	16	32	256	512
32 * 32	0.00039	0.000277	0.0002	0.000253	0.000191	0.000149
256 * 256	0.122437	0.076364	0.059801	0.056905	0.064155	0.064627

矩阵规模/基准值	4	8	16	32	256	512
512 * 512	0.833879	0.70548	0.368306	0.367684	0.441689	0.451925
1024 * 1024	6.16822	5.543.9804993952	2.905321	2.768627	3.413362	3.513266
2048 * 2048	43.356809	26.944123	21.276548	18.540469	21.389535	24.492791
4096 * 4096	183.2321	177.2315	152.485619	131.021022	148.376083	171.581975

通过绘制成表格，可以更加直观地观察到递归基准值的选取对矩阵乘法的速度是有影响的（图中纵轴的单位是秒）



Strassen算法作为分治算法，以计算2048 * 2048 大小的矩阵乘法为例，如果递归值设置过小，假设为1， 则会导致在接近递归基准条件的时候会产生大量的大小为1的子矩阵，即单个数字。这样Strassen算法在动态分配堆空间所耗费的时间会非常多，掩盖了该算法原本的优势。所以在一个最适合的基准值处递归是非常有必要的。经过试验分析，从图中可以看出，在本电脑上测试最适合的基准值是第4组基准值，在本实验中为32，即在矩阵大小小于32时，停止递归，直接通过访存优化算法运算出来的结果时间是最快的。

使用编译优化对程序运行时速度加快原理分析

如果要提升程序的运行时间，进而加快矩阵乘法的速度，可以通过开启编译优化来进行实现。

下面是开启编译优化与未开启编译优化的程序运行时间进行对比（以Strassen算法为例）：

矩阵规模/基准值	32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 20
未开启编译优化	0.00024	0.075665	0.525403	3.93946	30.2746
开启编译优化	0.000253	0.056905	0.367684	2.768627	18.540469
优化效率/%	-5.138339921	32.96722608	42.89525788	42.2893008	63.289288

综合比较可以发现使用编译优化会在规模越大的矩阵乘法优化效果越显著。

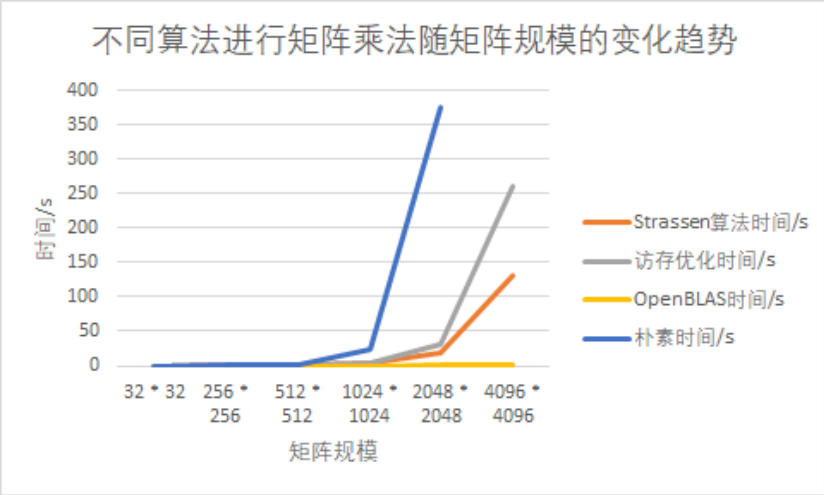
对于优化级别1来说，优化的主要是合并相同常量的时间以及优化汇编语言当中的循环与分支，在本次项目当中并不能起到很大的优化作用。如果使用优化级别2，由于在Strassen算法当中涉及到大量矩阵内部数据的读取，而该级别的优化可以通过指令`l -fforce-mem`强制把存放在内存位置中的所有变量都复制到寄存器中，从而使得所有的内存引用潜在的共同表达式，而处理器访问寄存器的速度要更加快，再者，该级别还可以指示编译器对函数调用保存和恢复寄存器，在Strassen算法当中有大量的递归函数调用，通过指令`l -foptimize-sibling-calls`可以优化相关的以及末尾递归的调用。除此之外，还能通过指令`l -fcprop-registers`使函数能够访问寄存器而且不必保存和恢复他们，由于在多次调用当中只进行一次寄存器的保存和恢复，所以优化级别2可以很好地优化程序，所以在本程序中使用编译优化级别2能够很好地加快程序的运行时间。

与使用OpenBLAS库进行运算的速度以及优化效率对比及分析

通过使用OpenBLAS库的矩阵乘法函数进行运算计时发现速度比起实现的方法来快了不少，通过前面提到的计算时间误差的方法分别比较OpenBLAS库与访存优化，以及Strassen算法的运行时间，得到下表：

矩阵规模	Strassen/s	访存优化时间/s	OpenBLAS时间/s	朴素时间/s
32 * 32	0.000253	0.000193	0.001148	0.000268
256 * 256	0.056905	0.065747	0.001925	0.088847
512 * 512	0.367684	0.455865	0.006794	0.988245
1024 * 1024	2.768627	3.943264	0.017891	24.298455
2048 * 2048	18.540469	32.464609	0.094559	375.722021
4096 * 4096	131.021022	260.303207	0.66195	NULL

通过绘制折线图可以更加直观地观察到不同算法在不同规模的矩阵乘法下的运行速度趋势：



可以发现随着矩阵规模的增大，各算法实现的矩阵乘法的时间消耗都呈指数级增长，经过一番对比，发现在运算规模很小的矩阵的时候，调用OpenBLAS库进行运算的速度还不及使用朴素算法进行矩阵乘法的时间，但是在运算矩阵的规模增大的时候，OpenBLAS库的优势就体现地淋漓尽致，几乎都是在相同的时间内完成了极大规模矩阵的乘法运算！

进一步进行优化效率的分析，以2048阶矩阵的乘法为例，利用Part2中提出的比较时间差距进而得出性能优化程度的公式，可以得到下面的表格：

矩阵规模	朴素乘法时间/s	访存优化时间/s	Strassen/s	OpenBLAS时间/s

2048 * 2048 矩阵规模	375.722021 朴素乘法时间/s	32.464609 访存优化时间/s	18.540469 Strassen/s	0.094559 OpenBLAS 时间/s
性能提升/倍(和朴素乘法对比)		10.57328034	19.26496854	3972.413647

通过比较，在运算2048 * 2048 规模的矩阵时，访存优化可以提升10倍的效率，而Strassen算法通过分而治之的思想通过不断地进行分块，以若干次的加法替代一次乘法可以提升将近20倍的效率，而OpenBLAS库的方法则可以提高将近4000倍的效率！

这不禁引起了我的好奇心，通过在网上查阅与OpenBLAS计算矩阵乘法的资料，了解到此库的矩阵乘法利用了下列优化方式：

1. 矩阵是以列优先存储的，这样就可以再访问内存时尽可能顺序访问。
2. 其次是有分块的思想，将一些向量提取出来，并在之后的计算当中多次对其复用，节省高速缓存的使用，进一步地，可以提取出一个矩阵块进一步优化。
3. 第三是使用寄存器变量，而不是直接操作内存，进一步减轻缓存的压力。
4. 再者是访存的优化，除了合理使用指针的移动来进行访存的优化，还可以将矩阵复制一份到连续的空间，这样就避免了在不连续的空间当中进行低效的访存，这样的意义就在于存储是连续的存储，读取也是连续的读取，通过结合在本次程序中的访存优化步骤，可以看出这样使得数据位于连续位置的优化是非常显著的。
5. 最后是利用单指令多数据（SIMD）的向量化指令，优化CPU的性能。

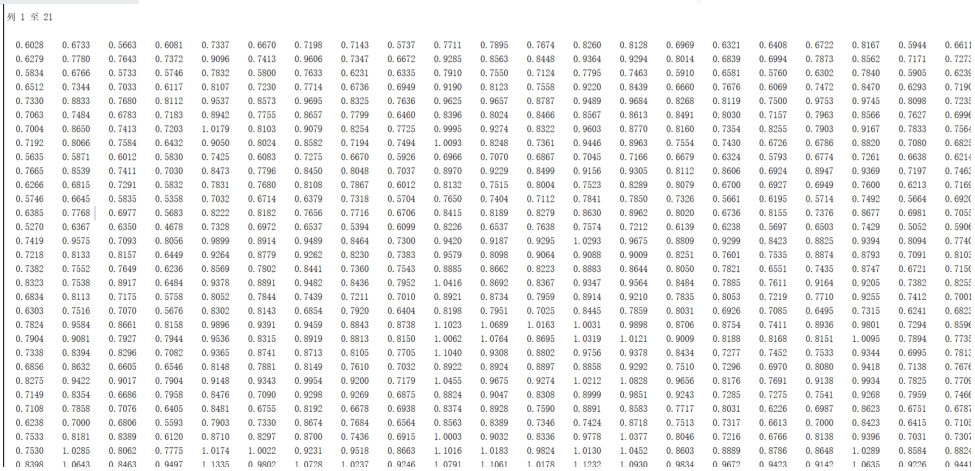
结果正确性测量及分析

首先将朴素矩阵乘法得到的结果与软件MATLAB的运算结果进行对比，结果如下图所示，观察得到结果几乎完全一样。

std_32.txt

```
60279.156250 67331.414062 56633.332031 60811.265625 73374.492188 66704.710938 71979.132812 7
62787.800781 77804.664062 76427.648438 73717.789062 90963.171875 74125.789062 96060.656250 7
58343.773438 67662.492188 57325.660156 57463.632812 78322.031250 57996.054688 76328.640625 6
65116.062500 73441.539062 70333.085938 61170.417969 81065.468750 72297.546875 77137.328125 6
73297.492188 88327.742188 76795.117188 81119.109375 95371.851562 85730.054688 96954.828125 8
70627.632812 74838.804688 67825.937500 71831.000000 89418.453125 77551.937500 86571.210938 7
70041.437500 86499.734375 74131.914062 72029.218750 101789.234375 81030.078125 90794.570312
71921.265625 80663.664062 75841.617188 64324.261719 90499.375000 80236.328125 85823.828125 7
56351.953125 58714.937500 60123.121094 58303.359375 74249.320312 60828.019531 72748.468750 6
76646.992188 85392.242188 74112.179688 70304.125000 84729.000000 77964.859375 84497.390625 8
62658.359375 68148.179688 72909.312500 58318.812500 78314.585938 76800.906250 81081.476562 7
57456.843750 66453.601562 58350.812500 53581.402344 70321.750000 67143.210938 63793.972656 7
63846.710938 77675.312500 69772.445312 56828.277344 82220.421875 81822.578125 76564.273438 7
52698.164062 63665.191406 63499.691406 46780.281250 73282.429688 69720.625000 65374.160156 5
74185.085938 95754.789062 70933.687500 80563.820312 98987.125000 89144.171875 94885.335938 8
72179.757812 81331.531250 81567.609375 64492.578125 92641.476562 87793.101562 92618.171875 8
73823.148438 75522.023438 76492.031250 62361.929688 85690.929688 78020.867188 84408.273438 7
83231.750000 75382.523438 89166.890625 64836.664062 93783.453125 88909.507812 94815.718750 8
68339.078125 81128.671875 71747.789062 57579.292969 80523.546875 78436.546875 74386.460938 7
63030.531250 75157.937500 70697.828125 56756.265625 83018.281250 81426.359375 68536.695312 7
78238.546875 95843.218750 86612.562500 81577.085938 98956.101562 93905.632812 94592.140625 8
79036.257812 90808.585938 79266.640625 79443.218750 95362.179688 83145.968750 89193.992188 8
73383.359375 83939.703125 82957.445312 70817.906250 93650.539062 87410.242188 87127.726562 8
68562.531250 86315.898438 66050.125000 65455.101562 81476.625000 78806.468750 81488.257812 7
```

（朴素乘法运行32 * 32矩阵结果）



（MATLAB运算32 * 32 相同矩阵结果）

进而以规模为256 * 256 的矩阵乘法为例，比较使用朴素算法与使用Strassen算法产生的txt文件中的精度差异，其中以朴素算法得到的浮点数为准进行误差计算。计算公式如下

$$\epsilon = \frac{\sum_{i=1}^{256}\sum_{j=1}^{256}\frac{|X_{ij}-Y_{ij}|}{X_{ij}}}{256 \times 256} \times 100\%$$

即求出标准文件中的数值与Strassen算法生成数值误差的平均值作为相对的误差进行判断。

编写程序进行比较：

```c
#include <stdio.h>
#include <stdlib.h>
#include<math.h>
int main(){
    FILE *fp1, *fp2;
    float cnt = 0.f;
    //打开两个文件
    fp1 = fopen("res_256.txt", "r");
    fp2 = fopen("std_256.txt", "r");
    //检测文件是否安全打开
    if(fp1 == NULL || fp2 == NULL){
        printf("Wrong in openning files.");
        exit(1);
    }else{
```

```
        for(int i = 0; i < 256; i ++){
            for(int j = 0; j < 256; j++){
                float f1, f2;
                fscanf(fp1, "%f", &f1);
                fscanf(fp2, "%f", &f2);
                cnt += fabsf(f1 - f2) / f1;
            }
        }

    }
    printf("cnt = %f\n", cnt / 256 * 256);
    return 0;
}
```

最终结果返回为

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/project3$ ./cmp
cnt = 0.031394
```

数据误差在3.13%，说明使用Strassen算法的精确度与使用朴素矩阵乘法相比会有损失。那么引起精度损失的原因是什么呢?

经过分析Strassen算法，发现Strassen 是利用若干次矩阵的加法替代了一次矩阵的乘法，而在浮点数的运算在加法和乘法的运算当中会存在精度损失的问题，在两个浮点数的加法当中，所需的小数点位数最多不会超过两加数的小数点位数 + 1，而两个数字相乘，由于浮点数尤其是float能够表示的精度有限，所以只能在小数点后六位进行截断，所以在进行若干次矩阵加法替代一次矩阵乘法的过程当中会存在着精度上的损失。所以在编写代码的时候，要综合考量比较重要的因素是时间还是精确度再考虑是否使用Strassen算法来加速矩阵乘法的过程。

Part4 Difficulties and Solutions

函数调用复制结构体速度慢问题及解决

在矩阵规模非常大的时候，通过调用矩阵乘法函数济进行运算的时候，开始在编写函数的时候形参设置成了

```
struct Matrix mulMatrix_tradition(struct Matrix mat1, struct Matrix mat2); //矩阵乘法
```

这样导致矩阵乘法运算的时间非常慢，结合课上所学内容，发现原因是调用函数的时候是按值传递，会复制一遍传入的参数，这样会严重拖慢程序的速度，改进之后将传入结构体本身改为传入指向该结构体内存的指针，这样就避免了复制矩阵结构体的问题，显著地加快了程序的运行速度。

复制字符串出错问题及解决

在最初编写程序的时候，为了在分割字符串寻找矩阵规模函数中不改变传入的参数值，想先使用一个C-风格的字符串`char str[10]`来存储命令行参数的复制值，但是在声明了字符串之后使用`strcpy()`函数时出现了

`*** stack smashing detected ***: terminated`的错误，经过排查，发现是声明字符串的时候申请内存空间不够导致的，因为不能预先得知输入命令行参数的长度，如果静态分配内存空间小会导致Linux的下栈溢出问题，分配内存空间大又会导致浪费，由于动态分配内存可以更合适地分配空间，于是后续改成使用设置为字符串地址的char指针，在声明时对其进行初始化动态分配内存空间，再调用`strcpy`进行复制，问题就得到了解决。

函数返回指向自动变量内存的指针问题及解决

最初调用函数的时候程序总是会崩溃，经过原因排查，发现是在被调用函数当中创建了自动变量来存储运算值，而在返回的时候直接将指向该自动变量内存的指针进行了返回，由于自动变量指向的内存在函数执行完毕之后就会被释放，所以返回的实际上是指向一块空内存空间的指针，在`main`函数使用该指针的时候就会导致系统崩溃。

后来通过在`main`函数当中事先创建目标矩阵，并为其动态分配内存空间之后作为函数实参传入被调用函数，并在被调用函数里面给目标矩阵内元素进行赋值，最后返回指向该目标函数内存块的指针，这样就有效避免了函数返回自动变量指向内存指针的问题。

OpenBLAS 库安装问题及解决

起初在安装OpenBLAS库时，会出现用户权限不够，无法安装的问题

```
/bin/sh: 1: cannot create /opt/OpenBLAS/include/openblas_config.h:
Permission denied
```

后通过

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/Week5$ su root
Password:
root@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/Week5#
```

命令`su root`获得root的执行权限之后可以成功安装。

第二个遇到的问题是最初安装的时候没有通过`cd ~`命令切换到根目录安装，导致OpenBLAS安装完成却无法正常被调用，经过切换到根目录之后，问题得到解决。

安装后找不到OpenBLAS库使用问题及解决

在矩阵乘法程序中包含OpenBLAS库的头文件<cblas.h>，之后在程序编译的时候显示找不到`cblas_sgemm`方法，报错如下：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/Users/86181/Desktop/大二课程/c++/C++Code/Week5$ g++ testnew.cpp -o test
/usr/bin/ld: /tmp/ccjlgie7.o: in function `TestLevel3()':
testnew.cpp:(.text+0x3e4): undefined reference to `cblas_sgemm'
collect2: error: ld returned 1 exit status
```

经过原因排查，发现是没有告诉编译器应该去往哪里寻找这个库，所以需要在编译链接程序的时候显式地告诉编译器OpenBLAS库所在的地址，通过查看OpenBLAS安装的目录，编写编译命令如下所示：

```
g++ -o main -L /opt/OpenBLAS/lib testnew.cpp -lopenblas
```

之后便可以正常调用OpenBLAS库进行一系列矩阵的运算了，问题得到解决。

Part 5 Thinking and Summary

通过这次使用C语言编写矩阵乘法的程序，除了对C语言更加熟悉，在编写C语言程序的时候也一直在记录与C++不同的地方，通过与上一次使用C++编写矩阵乘法进行对比，也对这两门编程语言的不同点以及相同点有了更深刻的认知，这些也都体现在第二部分的分析当中。同时感受到C语言更加贴近底层，如动态分配时还需要指定分配的字节数目，在使用文件指针的时候也许多在C++当中封装起来的方法，都依赖于编写者书写正确的类型。

在上一次的项目当中，并没有使用头文件以及cmake等用于管理代码的工具与方法，这次通过手动编写cmake，以及将函数的原型以及定义分开书写，也让我对C/C++的程序文件构成有了更深的理解，而且通过链接外部OpenBLAS库用于本地调用方法进行矩阵乘法的操作以及通过cmake了解到OpenBLAS内部架构以及实现也加深了对编写cmake以及对代码文件管理方式的理解。

在上一次的project当中，我是直接使用了二维数组进行矩阵信息的存储，并没有对底层内存实现有很深的理解。但是这一次通过使用一维数组来模拟二维数组的存储，经过一维数组进行二维数组内部元素的访问是要通过在行的基础上加上列的总数目在加上列数才能取到目标的数据，这样的操作也让我对数组的存储方式有了更深的理解。这一次通过一个结构体来代表一个矩阵的方式，由于C语言没有引用，起初通过直接传值的方式会复制大量的数据从而拖慢程序的运行，后来通过直接传递指针的方式传递地址，直接对指向的内存进行操作，通过箭头运算符来访问内部元素，都使我更加熟练了结构体的操作。综合使用数组以及结构体也更加让我了解了指针的本质以及相关操作。

在矩阵速度的优化当中，通过实现了访存优化，分块矩阵的优化以及Strassen算法，查看了解到OpenBLAS库的矩阵乘法操作以及探索了C语言在硬件层次的操作（SIMD）融合汇编语言的方式，也让我对计算机的内存机制以及CPU的运作更加地了解。通过对得到时间数据的综合分析以及提升效率的分析，我也更加明白熟悉计算机底层的实现逻辑对对优化上层的代码逻辑的重要性，上述方法都无一例外地利用了计算机底层的硬件逻辑进行优化，所以在之后的学习当中，我也会更加注重对代码在计算机底层，特别是CPU以及内存方面的优化策略。

通过查阅资料了解到进一步的优化还可以通过使用多线程实现，因为使用多线程可以更加充分地利用CPU的运算资源，在矩阵的乘法当中也可以通过将矩阵分块之后交给多个线程进行处理，这样处理的速度肯定是要优于单个线程的运行效率的。遗憾的是，由于之前没有对多线程有太深的了解，所以在此次项目当中不能使用这一技术来更优地实现矩阵乘法，但是我也会进一步学习多线程，在下一次的项目当中派上用场。

以上就是这次报告的全部内容，感谢老师的阅读！