# CS205 C/C++Program Design - Project 3




# 姓名：张琦

# 学号：12010123


# 日期：2021.10.21

# Part 1 - 分析问题

## 1.1 : 任务

1. 使用C语言设计一个能够实现矩阵相乘的程序，该程序计算的矩阵需要程序从`txt`文档中获取，矩阵相乘的结果也需要存储在新生成的`txt`文档中。
2. 设计一个用于表示矩阵的结构体，结构体中包含矩阵的行数，列数和矩阵所包含的元素。
3. 实现函数：
    1. 创建矩阵`creat()`
    2. 删除矩阵`delete()`

3. 复制矩阵 `copy()`
4. 矩阵乘法 `mul()`
5. 打印矩阵 `print_matrix()`
6. 将矩阵写入txt文档 `write_matrix()`

4. 提出方法给程序加速
5. 当输入在Linux控制台分别输入

   `./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`

   时，分别实现矩阵相乘，并将正确的计算结果输出到指定的`txt`文档中
6. 给矩阵乘法算法提速，并比较本次使用的提速方法与`Project2`中使用的提速方法的区别
7. 将优化后程序的总执行时间与调用`OpenBLAS`实现矩阵乘法所需的时间进行比较

## 1.2 基本思路与具体实现

### 1.2.1 文件的输入与输出

本次Project要求使用纯C语言编程，C语言中最常用的文件操作结构是定义在头文件`<stdio.h>`中的`FILE`。

`FILE`这个结构包含了文件操作的基本属性，对文件的操作都要通过这个结构的指针来进行。其中操作函数`fscanf()`可以实现将数据从流中按格式读取数据，`fprintf()`可以将数据按格式输出到流中，刚好可以满足我们对按空格分隔元素的矩阵这种具有一定格式的数据组读入和输出的需求。

C语言不能调用`Vector`，很难实现在读取数据的同时同时记录矩阵的行数和列数。如果先遍历元素直到`EOF`,会使程序运行所需的时间增长。所以为了方便程序在读入时能够通过`mollac()`提前在堆中为矩阵数组分配内存空间，我规定所有矩阵文件的第一行用于记录矩阵的行数和列数。在读取矩阵的元素前，先读取矩阵的行和列。在向txt文件输出矩阵时也先输出矩阵的行数和列数。

以32 x 32矩阵为例：

```
32 32
21.2 50.1 88.3 48.1 22.6 57.2 26 30.5 16.9 54.1 46.9 21.3 34.7 49.4 26.3
79.2 4 28.2 79.1 75.2 86.5 85.7 35.7 25 25.9 31.9 80.5 43.6 58.1 23.7 29
2.5 62.4 11.2 65.6 26.1 17.8 44.6 41.4 74.5 31.4 22.1 6 21.8 5.9 58.5 48.
11.1 46.1 16.7 25.5 29.8 57.8 85 43.2 69.5 9.6 77.7 29.2 20.5 86.3 76.3 6
89.4 68.9 97 42.5 63.1 69.1 38 43.6 33.1 74.6 22.9 66.2 32.4 97.7 13.6 8.
61.2 82.3 84.2 47.5 26.5 58.2 9.8 56.9 4.4 34.5 38.6 66.9 81.1 23.1 77.1
39.9 17 10.1 26 85.5 72.4 13.5 42.5 71.8 91.3 57.8 53.5 31 16.8 46.6 88.2
25.5 40.1 7.6 74.1 77.5 31.2 70.6 91.1 56.8 2.3 81 15.2 33.6 65.7 90.4 22
```

这样以来，还没有读入元素就可以动态申请内存，预防段错误(Segmentation Fault)

```
fscanf(fileprt, "%lld %lld\n", &row, &column);

float *elements = (float *)malloc(sizeof(float) * row * column);
```

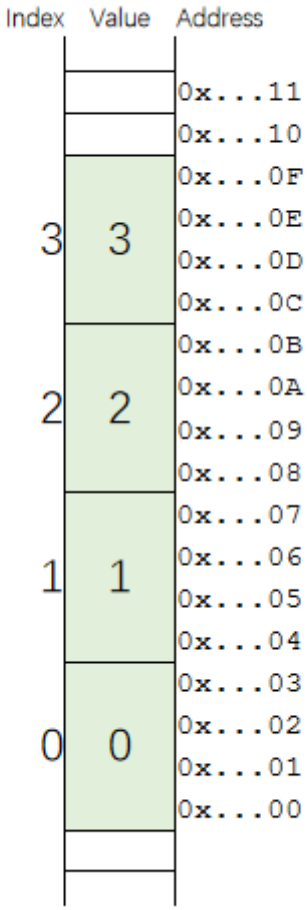
Notice:

1. 当 `fprintf()` 被指定为 `"w"` 类型时，会在当前目录寻找目标文件，若没有则会创建一个目标文件。
2. 正如 `Project2` 所说的，动态申请内存才能满足超大矩阵的存储,使用临时变量容易出现栈溢出错误(StackOverFlowError)

### 1.2.2 矩阵的存储方式

在 `Project2` 当中，我选用 `float **` 即二维数组的方式存储矩阵的元素。在听取了于老师课上的建议后，这次选用 `float *` 一维数组来存储元素。因为通过指针实现的二维数组用于存储浮点数的地址在内存中不一定是连续的，但是用指针实现的一维数组地址一定连续。

图片来源：Lecture5 PPT




所以理论上，内存连续的一维数组更有利于内存的寻址，实现矩阵乘法所需的时间应该更短。

### 1.2.3 矩阵计算结果的验证

在 `Project2` 中我利用了 `MATLAB` 来验证矩阵计算结果的准确性，但只通过眼睛观察结果并不具有说服力。所以本次项目我采用构造精度比较辅助函数来判断结果是否准确，对照结果来源于利用 `OpenBLAS` 矩阵乘法库得到的计算结果。

百分误差的计算公式：（图片来源：百度百科）

$$\delta = \frac{|a-b|}{a} \times 100\%$$

$a$表示真实值，这里指使用OpenBLAS计算的结果。

$b$表示$a$的近似值，这里指使用矩阵乘法算法计算的结果

由数学知识知： 当矩阵的尺寸很大时，最大百分误差和平均百分误差均小于0.00001%，而当矩阵的尺寸很小时，最大百分误差和平均百分误差趋近于0，说明矩阵之间的差别非常小，可以说两矩阵相同。

误差分析结构体：

```
typedef struct Error
{
    float max;
    float average;
} Error;
```

误差分析函数：

```
Error accuracyCheck(Matrix A, Matrix B)
{
    if (A.row != B.row || A.col != B.col)
    {
        printf("A and B have different size");
        Error err;
        err.max = -1;
        err.average = -1;
        return err;
    }
    else
    {
        long long size = A.col * A.row;
        Error err;
        err.max = 0;     //峰值
        err.average = 0; //平均值
        for (long long i = 0; i < size; i++)
        {

            // fabs求浮点数x的绝对值
            float delta = fabs(A.elements[i] - B.elements[i]) / B.elements[i];
            if (err.max < delta)
            {
                err.max = delta;
            }
            err.average += delta;
        }
        err.average = err.average / (size);
        return err;
    }
}
```

### 1.2.4 计时

参考博客：[https://blog.csdn.net/fz_ywj/article/details/8109368](https://blog.csdn.net/fz_ywj/article/details/8109368)

C语言中最常用的计时器是利用`<time.h>`中的`clock()`函数来实现的。因为程序在矩阵小的时候会很快结束，我希望计时器有更高的精度，并以秒的形式表示，所以我希望使用宏定义常数`CLK_TCK`来得到对应的秒数，但是Linux系统下的`<time.h>`头文件中常数`CLK_TCK`没有定义（或许是失效）

```
   XSI-conformant systems.  */
#define CLOCKS_PER_SEC  ((__clock_t) 1000000)

#if (!defined __STRICT_ANSI__ || defined __USE_POSIX) \
    && !defined __USE_XOPEN2K
/* Even though CLOCKS_PER_SEC has such a strange value CLK_TCK
   presents the real value for clock ticks per second for the system.  */
extern long int __sysconf (int);
# define CLK_TCK ((__clock_t) __sysconf (2)) /* 2 is _SC_CLK_TCK */
#endif

#ifdef __USE_POSIX199309
/* Identifier for system-wide realtime clock.  */
```

所以最后我选用了Linux C函数`gettimeofday()`来实现计时器

```
struct timeval start, end;

gettimeofday(&start, NULL);

///code part

gettimeofday(&end, NULL);

long timeuse = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec;

printf("The Required Time = %f Seconds\n", timeuse / 1000000.0);
```

### 1.2.5 创建矩阵和删除矩阵的函数

我一共编写了四种用于创建矩阵的函数：

1 . Matrix creat1(long long *row*, long long *col*, float **arr*) : 创建一个指定大小的空矩阵

```
Matrix creat1(long long row, long long col)
{

    float *elements = (float *)malloc(sizeof(float) * row * col);
    Matrix matrix = {row, col, elements};
    return matrix;
}
```

2. Matrix creat2(FILE fileprt, *const char filename*)： 从目标txt文件中读入矩阵

```
Matrix creat2(FILE *fileprt, const char *filename)
{
    long long row, column = 0;

    fileprt = fopen(filename, "r");

    fscanf(fileprt, "%lld %lld\n", &row, &column);

    float *elements = (float *)malloc(sizeof(float) * row * column);

    for (long long i = 0; i < row * column; i++)
    {
        fscanf(fileprt, "%f", &elements[i]);
    }

    Matrix matrix = {row, column, elements};

    fclose(fileprt);

    return matrix;
}
```

3. Matrix creat3(long long *row*, long long *col*)： 创建一个指定大小的单位矩阵

```
Matrix creat3(long long row, long long col)
{
    float *arr = (float *)malloc(row * col * sizeof(float));
    for (long long i = 0; i < row * col; i++)
    {
        arr[i] = 1.0;
    }
    Matrix matrix = {row, col, arr};
    return matrix;
}
```

4. Matrix creat4(long long *row*, long long *col*, float *bound*)： 随机生成一个指定大小的矩阵，矩阵的元素大于0且存在上界Bound

```
Matrix creat4(long long row, long long col, float bound)
{
    float *arr = (float *)malloc(row * col * sizeof(float));
    for (int i = 0; i < row * col; i++)
    {
        arr[i] = 0 + 1.0 * rand() / RAND_MAX * bound;
    }
    Matrix matrix = {row, col, arr};
    return matrix;
}
```

5. void delete (Matrix *matrix*)： 删除目标矩阵

本次project的矩阵是一个结构体，创建的结构体只是一个存储了矩阵行数，列数和元素地址的临时变量，占用的内存空间很小。所以考虑删除目标矩阵是将目标矩阵的行数列数置0，再释放指针所指向的内存空间。

矩阵的元素存储在结构体的float * 成员当中，在删除矩阵时一定要将对应的矩阵释放，否则可能会造成内存泄露。

```
void delete (Matrix matrix)
{
    float *ptr = matrix.elements;
    free(ptr);
    matrix.row = 0;
    matrix.col = 0;
}
```

注：这里的矩阵删除函数存在更好的实现形式，因为删除的矩阵结构体仍然在占用栈内存，但没有想到除了使用 stuct  Matrix * 之外更好的方法，所以只保留了简单的做法。

### 1.2.6 加速

仅介绍了提速方法，具体时间分析和与Project2选用方法的区别在下文

1. 内存寻址优化1 （一维数组）： 选用了一维数组而不是二维数组存储浮点数
2. 内存寻址优化2（改变循环的顺序）： 像`Project2`一样，改变循环的顺序，但改变的是一维数组的循环顺序
3. 多线程加速（Openmp指令）：相较于`thread>`头文件中的线程操作，`Openmp`指令更加简单易懂，效率更高。
4. 编译器优化（O3加速）：操作简易但高效的gcc优化指令

## 1.3 我的操作环境

- 系统：`Ubuntu20.04（64bit）`
- 编译器：`gcc 9.3.0`
- CPU: `I7-10750H 2.60GHZ`

# Part 2 - 源代码

## 2.1 Source1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <omp.h>
#include <math.h>
#include "header.h"
#include <cblas.h>

#pragma GCC optimize(3)

const enum CBLAS_ORDER Order = CblasRowMajor;
```

```
const enum CBLAS_TRANSPOSE TransA = CblasNoTrans;
const enum CBLAS_TRANSPOSE TransB = CblasNoTrans;
const float alpha = 1;
const float beta = 0;
int main(int argc, char **argv)
{

    #pragma omp parallel for schedule(dynamic)

        // i7 10750H 线程数为15
        for (int i = 0; i < 15; i++)
        {
            printf("Thread Check number %d from thread = %d\n", i,
omp_get_thread_num());
        }

    //直接使用time.h计时存在问题，故使用过新的计时方法，
    //参考博客： https://blog.csdn.net/fz_ywj/article/details/8109368

    struct timeval start, end;

    const char *filename1 = argv[1];
    const char *filename2 = argv[2];
    const char *filename3 = argv[3];

    FILE *fileprt = NULL;
    Matrix matrix1 = creat2(fileprt, filename1);

    Matrix matrix2 = creat2(fileprt, filename2);

    float *result = (float *)malloc(matrix1.row * matrix2.col * sizeof(float));

    gettimeofday(&start, NULL);

    cblas_sgemm(Order, TransA, TransB, matrix1.row, matrix2.col,
                matrix1.col, alpha, matrix1.elements, matrix1.col,
                matrix2.elements, matrix2.col, beta, result, matrix2.col);


    Matrix matrix3 = mul_fast(matrix1, matrix2);

    gettimeofday(&end, NULL);

    long timeuse = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec -
start.tv_usec;

    printf("The Required Time = %f Seconds\n", timeuse / 1000000.0);

    Matrix matrix3 = mul_fast(matrix1, matrix2);

    Matrix correct = creat5(matrix1.row, matrix2.col, result);

    Error error = accuracyCheck(matrix3, correct);

    printf("The max error: %f%%\n", error.max);
    printf("The average error: %f%%", error.average);

    write_matrix(correct,fileprt,filename3 );  //使用openBlas计算
```

```
    write_matrix(matrix3, fileprt, filename3);
}
```

## 2.2 Source2.c

```
#include <stdio.h> //输入输出
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <omp.h> //OPENMP加速
#include "header.h"

#pragma GCC optimize(3)

Error accuracyCheck(Matrix A, Matrix B)
{
    if (A.row != B.row || A.col != B.col)
    {
        printf("A and B have different size");
        Error err;
        err.max = -1;
        err.average = -1;
        return err;
    }
    else
    {
        long long size = A.col * A.row;
        Error err;
        err.max = 0;      //峰值
        err.average = 0; //平均值
        for (long long i = 0; i < size; i++)
        {

            // fabs求浮点数x的绝对值
            float delta = fabs(A.elements[i] - B.elements[i]) / B.elements[i];
            if (err.max < delta)
            {
                err.max = delta;
            }
            err.average += delta;
        }
        err.average = err.average / (size);
        return err;
    }
}

void delete (Matrix matrix)
{
    float *ptr = matrix.elements;
    free(ptr);
    matrix.row = 0;
```

```
    matrix.col = 0;
}

Matrix creat1(long long row, long long col)
{

    float *elements = (float *)malloc(sizeof(float) * row * col);
    Matrix matrix = {row, col, elements};
    return matrix;
}

//假设文件的第一行保存了矩阵的 行数 和  列数
//从文件中读入矩阵

///数据读入存在bug

Matrix creat2(FILE *fileprt, const char *filename)
{
    long long row, column = 0;

    fileprt = fopen(filename, "r");

    fscanf(fileprt, "%lld %lld\n", &row, &column);

    float *elements = (float *)malloc(sizeof(float) * row * column);

    for (long long i = 0; i < row * column; i++)
    {
        fscanf(fileprt, "%f", &elements[i]);
    }

    Matrix matrix = {row, column, elements};

    fclose(fileprt);

    return matrix;
}

//默认创建单位矩阵
//尝试设置数量矩阵
Matrix creat3(long long row, long long col)
{
    float *arr = (float *)malloc(row * col * sizeof(float));
    for (long long i = 0; i < row * col; i++)
    {
        arr[i] = 1.0;
    }

    //默认设计为单位矩阵

    Matrix matrix = {row, col, arr};

    return matrix;
}

///生成随机矩阵 , 默认为大于等于0的数
Matrix creat4(long long row, long long col, float bound)
{
```

```
    float *arr = (float *)malloc(row * col * sizeof(float));
    for (int i = 0; i < row * col; i++)
    {
        arr[i] = 0 + 1.0 * rand() / RAND_MAX * (bound);
    }
}

Matrix creat5(long long row, long long col, float *arr)
{
    Matrix matrix = {row, col, arr};
    return matrix;
}

void copy(Matrix mat1, Matrix mat2)
{ // mat1是复制完后的原数组，mat2是复制目标
    mat1.col = mat2.col;
    mat1.row = mat2.col;
    mat1.elements = mat2.elements; //存在bug，mat2内存释放后，mat1的矩阵失效
}

//参考博客：https://blog.csdn.net/flying_saker/article/details/83350731

///先尝试使用一维数组的朴素三重循环

Matrix mul_slow(Matrix mat1, Matrix mat2)
{

    if (mat1.col == mat2.row)
    {

        float *arr = (float *)malloc(mat1.row * mat2.col * sizeof(float));

#pragma omp parallel for schedule(dynamic)
        for (long long i = 0; i < mat1.row; i++)
        {
            for (long long j = 0; j < mat2.row; j++)
            {
                for (long long k = 0; k < mat1.col; k++) // mat1.col == mat2.row
                {
                    arr[mat2.row * i + j] += mat1.elements[mat1.col * i + k] *
mat2.elements[mat2.col * k + j];
                }
            }
        }
        Matrix result = {mat1.row, mat2.col, arr};

        return result;
    }
    else
    {
        printf("Can not mul because Mat1's row is not equal to Mat2's column");
    }
}

Matrix mul_fast(Matrix mat1, Matrix mat2)
{
    if (mat1.col == mat2.row)
    {
```

```
        float *arr = (float *)malloc(mat1.row * mat2.col * sizeof(float));

#pragma omp parallel for schedule(dynamic)
        for (long long i = 0; i < mat1.row; i++)
        {
            for (long long k = 0; k < mat2.row; k++)
            {

                float s = mat1.elements[mat1.col * i + k];
                for (long long j = 0; j < mat1.col; j++) // mat1.col == mat2.
                {
                    arr[mat2.row * i + j] += s * mat2.elements[mat2.col * k +
j];
                }
            }
        }
        Matrix result = {mat1.row, mat2.col, arr};

        return result;
    }
    else
    {
        printf("Can not mul because Mat1's row is not equal to Mat2's column");
    }
}

void print_matrix(Matrix Matrix)
{

    for (long long i = 0; i < Matrix.row * Matrix.col; i++)
    {
        if ((i + 1) % Matrix.col == 0) //列的最后一位
        {
            printf("%f\n", Matrix.elements[i]);
        }
        else
        {
            printf("%f ", Matrix.elements[i]);
        }
    }
}

void write_matrix(Matrix matrix, FILE *fileprt, const char *filename)
{

    fileprt = fopen(filename, "w");

    //写入行数 和 列数
    fprintf(fileprt, "%lld  %lld\n", matrix.row, matrix.col);

    for (long long i = 0; i < matrix.row * matrix.col; i++)
    {
        if ((i + 1) % matrix.col == 0) //列的最后一位
        {

            fprintf(fileprt, "%lf\n", matrix.elements[i]);
        }
```

```
        else
        {
            fprintf(fileprt, "%lf ", matrix.elements[i]);

        }
    }
    fclose(fileprt);
}
```

## 2.3 head.h

```
#pragma once

typedef struct Matrix

{

  long long row;

  long long col;

  float *elements;


} Matrix;



typedef struct Error

{

  float max;

  float average;

} Error;



Error accuracyCheck(Matrix *A*, Matrix *B*);



void delete (Matrix *matrix*);



Matrix creat1(long long *row*, long long *col*);



Matrix creat2(FILE **fileprt*, const char **filename*);
```

```
Matrix creat3(long long *row*, long long *col*);

Matrix creat4(long long *row*, long long *col*, float *bound*);

Matrix creat5(long long *row*, long long *col*, float **arr*);

void copy(Matrix *mat1*, Matrix *mat2*);

Matrix mul_slow(Matrix *mat1*, Matrix *mat2*);

Matrix mul_fast(Matrix *mat1*, Matrix *mat2*);

void print_matrix(Matrix *Matrix*);

void write_matrix(Matrix *matrix*, FILE **fileprt*, const char **filename*);
```

# Part 3 - 结果的检验和优化

## 3.1.计算结果检验

利用上文的百分误差计算公式以及OpenBLAS来验证矩阵乘法算法是否精确：

同上文，分别创建float指针申请待检验矩阵和正确的矩阵的内存，利用自己编写的矩阵乘法得到待检验矩阵，利用 OpenBLAS的`cblas_sgemm()`来得到正确的矩阵，最后将结果带入误差检验函数`Error accuracyCheck(Matrix A, Matrix B)`来计算百分误差。

具体实现代码：（matrix3是待检测矩阵，correct是正确答案矩阵）

```
const enum CBLAS_ORDER Order = CblasRowMajor;
const enum CBLAS_TRANSPOSE TransA = CblasNoTrans;
const enum CBLAS_TRANSPOSE TransB = CblasNoTrans;
const float alpha = 1;
const float beta = 0;

const char *filename1 = argv[1];
const char *filename2 = argv[2];
const char *filename3 = argv[3];

FILE *fileprt = NULL;
Matrix matrix1 = creat2(fileprt, filename1);

Matrix matrix2 = creat2(fileprt, filename2);

float *result = (float *)malloc(matrix1.row * matrix2.col * sizeof(float));

cblas_sgemm(Order, TransA, TransB, matrix1.row, matrix2.col,
            matrix1.col, alpha, matrix1.elements, matrix1.col,
            matrix2.elements, matrix2.col, beta, result, matrix2.col);

Matrix matrix3 = mul_fast(matrix1, matrix2);

Matrix correct = creat5(matrix1.row, matrix2.col, result);

Error error = accuracyCheck(matrix3, correct);

printf("The max error: %f%%\n", error.max);
printf("The average error: %f%%", error.average);
```

依次验证提供的32 x 32、256 x 256 和 2048 x 2048矩阵：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ gcc source1.c source2.c -lopenblas -fopenmp -o matmul
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-32.txt mat-B-32.txt out32.txt
The max error: 0.000000%
The average error: 0.000000%marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-256.txt mat-B-256.txt ou
t256.txt
The max error: 0.000001%
The average error: 0.000000%marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt
out2048.txt
The max error: 0.000004%
The average error: 0.000001%marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$
```

从结果可以看出无论是小尺寸矩阵还是大尺寸矩阵，计算结果的百分误差都非常小，均小于0.00001%，所以可以证明矩阵乘法的算法是正确的。

## 3.2.程序加速：

### 3.2.1 内存寻址优化1：改变浮点数的存储方式

不同于Project2，在Project3中我选用了float* 而不是 float ** 来存储浮点数。因为相较于通过float **对多个float *申请内存，只通过float * 申请一次的内存一定是连续的。当矩阵的尺寸很大时，连续调用不连续的内存地址会增加程序的负担，导致程序的运行时间增长。

此外代码量肉眼可见地减少，编译所需的时间也会减少。

所以理论上程序运行所需的时间会增长

project2矩阵内存申请代码：

```
int size = row;
double **answer;
answer = new double *[size];
for (int i = 0; i < size; i++)
{
    answer[i] = new double[size];
}
```

project3矩阵内存申请代码：

```
float *elements = (float *)malloc(sizeof(float) * row * col);
```

以下是均使用慢速矩阵乘法对2048x2048矩阵进行计算的时间统计：

project2：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project02$ ./matmul mat-A-2048.txt mat-B-2048.txt out2048.txt

程序完成运行：
总时长：  313.024 s
float类型矩阵的输出时长：  154.555 s
double类型矩阵的输出时长：  156.82 s
```

project3：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt

The Required Time = 201.908785 Seconds
```

结果出乎意料，一维数组所需的时间竟然要高于二维数组。可能是因为两次Project我采取了不同的头文件和方法读入浮点数和输出浮点数，导致总时间的失去了可比性。

如果要单纯比较存储方式的区别，我应该再写一个结构体表示矩阵，将存储方式改为float **，但这样的话我所有的函数都要重新修改，由于时间限制，我没有来得及实现。

### 3.2.2 内存寻址优化2：改变朴素三重循环的循环顺序

受到Project2的启发，尽管是用的一维数组存储矩阵，也应该可以通过改变循环的顺序来实现内存寻址优化。

朴素IJK三重循环：

```
for (long long i = 0; i < mat1.row; i++)
{
    for (long long j = 0; j < mat2.row; j++)
    {
        for (long long k = 0; k < mat1.col; k++) // mat1.col == mat2.row
        {
            arr[mat2.row * i + j] += mat1.elements[mat1.col * i + k] * mat2.elements[mat2.col * k + j];
        }
    }
}
Matrix result = {mat1.row, mat2.col, arr};
```

IKJ寻址优化后三重循环：

```
for (long long i = 0; i < mat1.row; i++)
{
    for (long long k = 0; k < mat2.row; k++)
    {

        float s = mat1.elements[mat1.col * i + k];
        for (long long j = 0; j < mat1.col; j++) // mat1.col == mat2.
        {
            arr[mat2.row * i + j] += s * mat2.elements[mat2.col * k + j];
        }
    }
}
Matrix result = {mat1.row, mat2.col, arr};
```

对2048x2048矩阵进行计算的时间统计：

朴素IJK三重循环:

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt

The Required Time = 201.908785 Seconds
```

IKJ寻址优化后三重循环:

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 51.630978 Seconds
```

和Project2的结论一样，改变循环顺序的效果非常显著，计算时间缩短了接近4倍！

### 3.2.3: 多线程加速（Openmp指令）

在project2中我也利用了并行计算的方法来尝试给矩阵乘法加速，但是当时的尝试是失败的，因为引入<*thread*>头文件来使用多线程非常复杂，和CPU的设置也相关。

如果没有提前设置CPU给程序执行分配的线程数，程序虽然是双线程执行的，但CPU只会使用单线程来执行这个程序，这就是为什么当时我的程序执行总时长在引入线程后变化不大，如下图所示。

Project2 提速前：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project02$ ./matmul mat-A-2048.txt mat-B-2048.txt out204
8.txt
float类型矩阵的输出时长：  58.3515 s
double类型矩阵的输出时长：  52.6459 s
```

Project2 提速后：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project02$ ./matmul mat-A-2048.txt mat-B-2048.txt out204
8.txt
double类型矩阵的输出时长：  111.97 s
float类型矩阵的输出时长：  120.104 s
总时长：  121.897 s
```

在Project3中，我选用了Openmp指令来实现多线程加速。

Linux系统下安装的gcc编译器自带OpenMp。OpenMp提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的pragma来指明自己的意图，由此编译器可以自动将程序进行并行化，操作极其简单。

我们选择在最外层循环前使用OpenMP指令来实现多线程，并编写了一个线程测试程序来查看程序在执行时是否真的存在多个线程。

```
#pragma omp parallel for schedule(dynamic)

//i7 10750H 线程数为15
 for (int i = 0; i < 15; i++)
 {
     printf("Thread Check number %d from thread = %d\n", i, omp_get_thread_num());
 }
```

`#pragma omp parallel for schedule(dynamic)` 指令表示让程序自己来分配线程数

```
#pragma omp parallel for schedule(dynamic)
        for (long long i = 0; i < mat1.row; i++)
        {
            for (long long k = 0; k < mat2.row; k++)
            {

                float s = mat1.elements[mat1.col * i + k];
                for (long long j = 0; j < mat1.col; j++) // mat1.col == mat2.
                {
                    arr[mat2.row * i + j] += s * mat2.elements[mat2.col * k + j];
                }
            }
        }
        Matrix result = {mat1.row, mat2.col, arr};

        return result;
```

多线程时间测试：
未使用多线程：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 51.630978 Seconds
```

使用多线程后：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
Thread Check number 1 from thread = 7
Thread Check number 12 from thread = 7
Thread Check number 13 from thread = 7
Thread Check number 14 from thread = 7
Thread Check number 7 from thread = 0
Thread Check number 5 from thread = 5
Thread Check number 11 from thread = 11
Thread Check number 0 from thread = 1
Thread Check number 2 from thread = 8
Thread Check number 3 from thread = 3
Thread Check number 8 from thread = 9
Thread Check number 6 from thread = 2
Thread Check number 9 from thread = 6
Thread Check number 10 from thread = 4
Thread Check number 4 from thread = 10
The Required Time = 39.966215 Seconds
```

从ThreadCheck可以看出，程序的循环中至少有11个线程在并行执行，且所需时间确实减少了接近1/5。尽管有11个线程在循环中并行执行程序，矩阵相乘所需的时间并不是像我们想象的一样直接下降为原来的 1/11，即线程数与程序执行所需的时间并不是线性相关的。

### 3.2.4： gcc编译优化

gcc编译优化在使用手册中的对应页数：https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

gcc编译器中存在编译优化指令，在编译时输入优化标识就可以调用编译优化指令，是操作简易但高效的程序加速方法。

打开优化标志会使编译器尝试以牺牲编译时间和调试程序的能力为代价来提高性能和/或代码大小，因为我们认为代码的编译时间不包含在程序的运行时间当中，所以调用优化指令后，控制台上显示的运行时间理论上是会下降的。

当优化标识被启用之后，gcc编译器将会试图改变程序的结构（当然会在保证变换之后的程序与源程序语义等价的前提之下），以满足某些目标，如：代码大小最小或运行速度更快（只不过通常来说，这两个目标是矛盾的，二者不可兼得）。

gcc和g++编译器是很强大的，可以选择性地优化我们编写的代码，从而提升程序运行的速度。在c++中我们还可以使用inline关键字将反复调用的函数声明为inline函数来加速程序，但本次项目使用的是C语言，不能使用.

调用指令：

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

本次Project中，我们尝试使用gcc的o3加速，以下为效果测试：

o3加速前：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
Thread Check number 1 from thread = 7
Thread Check number 12 from thread = 7
Thread Check number 13 from thread = 7
Thread Check number 14 from thread = 7
Thread Check number 7 from thread = 0
Thread Check number 5 from thread = 5
Thread Check number 11 from thread = 11
Thread Check number 0 from thread = 1
Thread Check number 2 from thread = 8
Thread Check number 3 from thread = 3
Thread Check number 8 from thread = 9
Thread Check number 6 from thread = 2
Thread Check number 9 from thread = 6
Thread Check number 10 from thread = 4
Thread Check number 4 from thread = 10
The Required Time = 39.966215 Seconds
```

o3加速后：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ gcc source1.c source2.c -fopenmp -o matmul
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 26.761378 Seconds
```

时间下降了接近1/3，可以看出编译器优化的效果显著！

## 3.3.：将结果与使用OpenBLAS计算矩阵乘法所需的时间进行比较

使用OpenBLAS前所需的时间：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ gcc source1.c source2.c -fopenmp -o matmul
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 26.761378 Seconds
```

使用OpenBLAS后所需的时间：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 25.205373 Seconds
```

让我疑惑的是，加速的时间并不明显。 OpenBLAS是优秀的开源库，作为产品级开源库，质量不应该这么低。

思考后，我发现我将2048 x 2048的矩阵写入txt文档的时间也放入了时间统计中，但OpenBLAS加速的过程只有矩阵乘法。

为了保证结果的准确性，我又做了一次时间测试，这次仅比较了两次矩阵乘法所需的时间，不将fscanf的fprinf的输出时间计入总时间。

使用OpenBLAS进行矩阵乘法所需的时间：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 0.266254 Seconds
```

使用改进后的矩阵乘法算法进行矩阵乘法所需的时间：

```
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$
marsy@DESKTOP-62OEA15:/mnt/d/CPP/Project03$ ./matmul mat-A-2048.txt mat-B-2048.txt out20
48.txt
The Required Time = 0.368212 Seconds
```

可以看出，时间下降了接近1/3,说明多次优化后的矩阵乘法算法仍然没有使用OpenBLAS进行矩阵乘法高效，对OpenBLAS进行恰当的设置之后，可能会更加高效，所以开源库的力量真的很强大。

# Part 4 - 遇到的困难及解决方法

本次project和上一次project的内容近似，所以这里只提及了与上次project不同的部分。

## 4.1 ： 从GitHub安装OpenBLAS计算库

本次Project要求我们安装OpenBLAS这个开源计算库，这是我第一次在Linux系统中通过git指令从github中安装库，我深深的感受到了Linux系统和Github的遍历程度和对于程序员的重要性。

我本以为要找到Linux虚拟系统的具体位置手动安装OpenBLAS

但其实在Ubuntu的虚拟Linux系统中键入以下指令就能完成安装

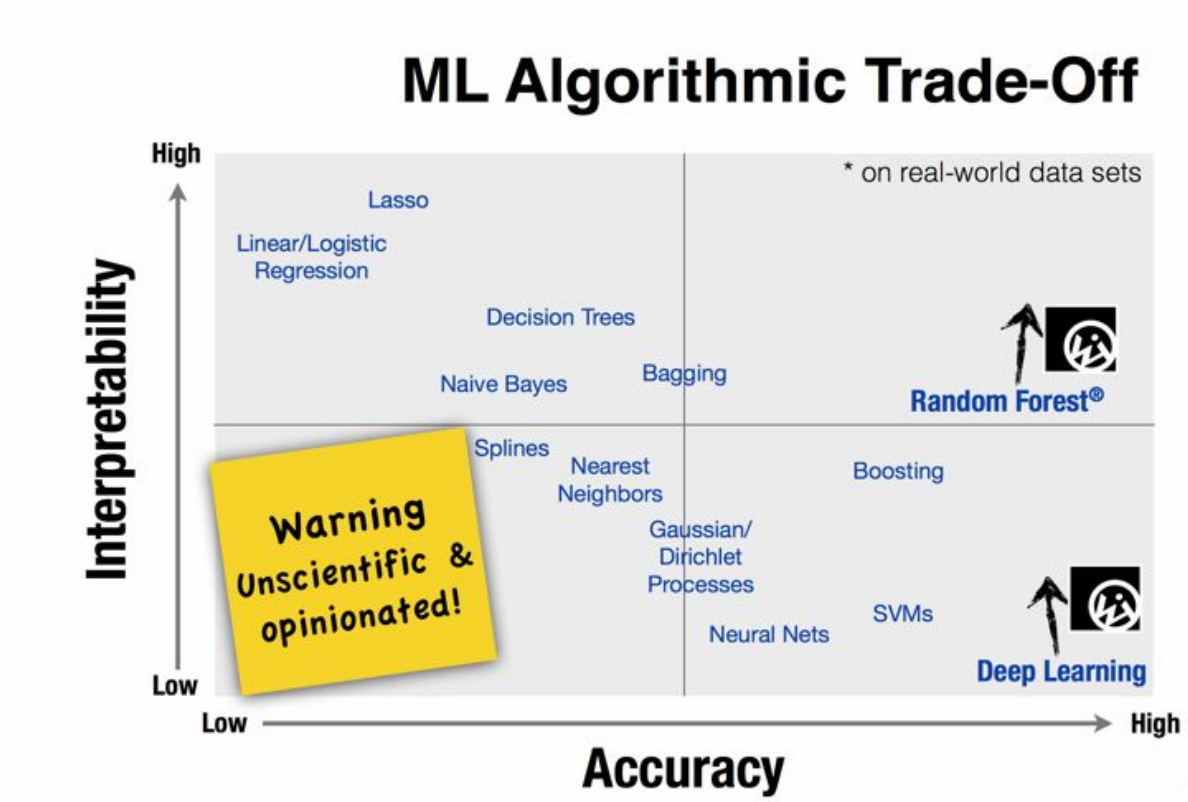
```
git clone https://github.com/xianyi/OpenBLAS.git
cd OpenBLAS
make -j8
sudo make PREFIX=/usr/local/OpenBLAS install
```

使用OpenBLAS时只需要在头文件中引入头文件<cblas.h>

所以利用Github安装优秀的开源库和项目真的比我们想象中的容易很多，git真的是一个革命性的开源分布式版本控制系统。

## 4.2： 如何检验矩阵乘法算法是否正确

算法正确性的验证从来就不是一个简单的问题。算法是正确的当且仅当对任意一个合法的输入经过有限步执行之后算法应给出正确的结果。键入某个合法的输入从而得到正确的结果并不是一件难事，但是对任意一个合法的输入都能得到正确的结果很难验证。




若不涉及到软件测试的相关知识，我选择了最简单的近似对拍的方法来验证结果的准确性。

假设OpenBLAS的矩阵乘法所得到的的结果一定是正确值。当正确答案与待检验结果每一个对应元素的差值的所取平均值都小于一个特定值，即当矩阵的尺寸很大时，最大百分误差和平均百分误差均小于0.00001%，而当矩阵的尺寸很小时，最大百分误差和平均百分误差趋近于0，说明矩阵之间的差别非常小，可以说两矩阵相同，此时我们可以说矩阵乘法算法是正确的。

利用的百分误差计算公式：

$$\delta = \frac{|a-b|}{a} \times 100\%$$