CS205 C/ C++ Programming - Project 2

Name：叶璨铭(Ye CanMing)

SID：12011404

Part 1 - Analysis

> 在part1中，我将分析本次project要达到的目标。part4将会介绍遇到的问题以及解决方案的基本原理，随后，part3会对part4中作出的假设（比如xxx会导致矩阵乘法变快）进行验证。最后，我们在part5进行一个简短的总结。

1-1 The goal of this project

在上一个project当中，我学习了c++语言的基本设计目标、输入输出的方法、大整数的抽象数据类型、大整数乘法的算法以及如何调用专业的库并且编译程序，主要是复习和巩固了第一节课的内容。

在本次project中，我们要实现一个能从文本文件中读取矩阵，对矩阵进行乘法运算，并且将矩阵运算的结果输出到文本文件当中的c/c++程序。我们有以下基本目标与project1是一致的：

- 零开销、高效。计算速度可以接受。
- 计算结果正确。
- 能处理异常输入。

为了学习到和第一次project不一样的东西，我提出以下几个新的目标

- **本次project不调用现有的矩阵库来实现**，不然的话代码就和project1几乎一样，只是把cin、cout换成了fstream，BigInt换成了Mat
    - 尝试自己**实现高效的运算代码。**
- 除了了解c++语言的设计思想， 需要了解一下c语言的设计思想并且实践。
    - 尝试c风格数组、c风格字符串
- 程序就像系统的一个命令一样，可以接受命令行选项
    - 比如，按照project文档要求，我们需要float和double两种类型的矩阵来计算。
    - 可以**提供一个命令行选项 -type来指定是用float还是double来理解txt的数据。**
- 主要是**复习和巩固第二节课的内容**（计算机、c++如何表示整数、小数、字符）以及第三节（判断与循环），预习第四节的内容（数组、字符串初步）。

1-2 Some problems to consider

- 如何高效地进行矩阵乘法？（从多个层次、角度进行优化）
- 如何优雅地处理命令行选项？（判断与循环）
- 如何正确输出中文提示语？（从源码、编译器、char类型、控制台等多个层次探讨分析，问题出在哪里）
- 如何抽象矩阵？（数组、动态数组、变长容器的原理，初步探究模板）
- 如何让矩阵输入输出高效？（变长的处理、C语言、利用矩阵抽象）
- 如何让矩阵乘法高效？（语法层面、计算机组成原理层面、算法层面）

Part 2 - Code

2-1 Project Hierarchy

P_Project2:
├─build
├─data
│ └─mat-[ABC]-[32,64,128,256,512,1024,2048] [-by_python,cpp] [_float,double].txt
├─python_help
│ └─diff.py (见3-1-3)
│ └─matmul.py(见3-1-1)
│ └─random_matrix_generator.py(见3-1-2)
├─src
│ └─matrix_multiplication.hpp
| └─main.cpp
└─CMakeLists.txt

2-2 Code

- main.cpp

```
//
// Created by 叶璨铭 on 2021/9/25.
//
#include <cstdio>
#include <cstring>
#include <iostream>
#include "matrix_multiplication.hpp"

#ifdef WIN32
//windows
#include <Windows.h>
#define str_case_cmp _stricmp
#define str_cat strcat_s
#define str_cpy strcpy_s
#define _CRT_SECURE_NO_WARNINGS
#elif __linux__
// linux
#define str_case_cmp strcasecmp
#define str_cat strcat
#define str_cpy strcpy
#endif
namespace matrix_multiplication_main //命名空间，封装
{
    void show_help()
    {
        puts(
R"xx(用法: ./file_matrix_multiplication {input1.txt} {input2.txt}
{output.txt} [-type float|double] [-help]
        选项:
            -type:  指定运算对象的类型是浮点数还是双精度浮点数。本程序默认是双精度浮点
数。
            -help:  显示帮助信息。 )xx"); //msvc编译器会有问题，因为msvc的编码是
gbk，对utf-8代码页识别不良。改下cmakelist就好
    }
    void error(const char*message, const char* argv0, const char* level)
    {
        char buf[150];
        str_cpy(buf, argv0);
        str_cat(buf, level);
        str_cat(buf,message);
        puts(buf);
    }
    void fatalError(const char *message, const char* argv0) //简化字符串的拼接
    {
        error(message, argv0, ": Fatal Error: ");
        puts("matrix multiplication terminated. ");
        show_help();
    }
    void optionError(const char *message, char* argv0) //简化字符串的拼接
    {
        error(message, argv0, ": Invalid arguments: ");
    }
    bool float_mode = false;
```

```
    int main(int argc, char* argv[])
    {
        int k = 0;
        char *in_out_filenames[3];
        for (size_t i = 1; i < argc; i++)
        {
            char* arg = argv[i];
            if(arg[0]=='-')//判断这个参数是一个选项
            {
                if(strlen(arg)>1)
                    switch (arg[1]) {
                    case 'h':case 'H': {
                            show_help();
                            return 0;
                        }
                    case 't':case 'T': {
                            arg = argv[++i];
                            float_mode = str_case_cmp(arg, "float") == 0 ||
str_case_cmp(arg, "f") == 0;
                            continue;
                        }
                    case '-': { //不支持--风格的选项
                        char *temp = &arg[2];
                        char buf1[40];
                        str_cpy(buf1, "unrecognized command-line option: --
");
                        str_cat(buf1, temp);
                        optionError(buf1, argv[0]);
                        char buf2[40];
                        str_cpy(buf2, "Did you mean -");
                        str_cat(buf2, temp);
                        puts(buf2);
                        puts("Matrix multiplication terminated. ");
                        return -1;
                    }
                    default:{
                        char buf[20];
                        str_cpy(buf, "No option named -");
                        str_cat(buf, &arg[1]);
                        optionError(buf, argv[0]);
                        puts("Matrix multiplication terminated. ");
                        return -1;
                    }
                    }
                else{
                    optionError("invalid option: -", argv[0]);
                    return -1;
                }
            }else
            {
                if (k<3)
                    in_out_filenames[k++] = arg;//如果不是选项参数，当做文件名
                else
                {
                    fatalError("Too many arguments!", argv[0]);//检查文件参数的
数量
                    return -1;
                }
```

```
            }
        }
        if(k<3)
        {
            fatalError("Too few arguments!", argv[0]);//检查文件参数的数量
            return -1;
        }
//
matrix_multiplication::file_matrix_multiplication(in_out_filenames,
float_mode ? 1.0f : 1.0);//三目表达式有问题，会认为1.0f和1.0都是double。所以还是要显式地指定调用哪个泛型函数。
        if(float_mode)
            matrix_multiplication::file_matrix_multiplication<float>
(in_out_filenames);
        else
            matrix_multiplication::file_matrix_multiplication<double>
(in_out_filenames);
        return 0;
    }
}

int main(int argc, char*argv[])
{
#ifdef WIN32
    SetConsoleOutputCP(CP_UTF8);//把Windows默认控制台编码改为utf-8。需配合编译器的utf-8。
#endif
    return matrix_multiplication_main::main(argc, argv);
}
```

- matrix_multiplication.hpp

```
//
// Created by 叶璨铭 on 2021/9/25.
//
#ifndef P_PROJECT2_MATRIX_MULTIPLICATION_HPP
#define P_PROJECT2_MATRIX_MULTIPLICATION_HPP

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <stdexcept>
#include <sstream>
#include <type_traits> //判断模板的类型
#include <omp.h> //openmp 并发计算
namespace matrix_multiplication
{
    template<typename E>
    void file_matrix_multiplication(char* in_out_filenames[]);//对外暴露的接口只有这一个函数。其他实现隐藏。但是编译不通过，只好把.h与.cpp合二为一。TODO: 为什么。泛型编译的原理是什么?
}
namespace matrix_multiplication
{
    const int matrix_max_size = 4096;
    template<typename E>
```

```
    struct Matrix
    {
        E** data; //[行坐标][列坐标]
        int rowCount;
        int columnCount;
        Matrix(E **data, int rowCount, int columnCount) : data(data),
rowCount(rowCount), columnCount(columnCount) {}
        Matrix(int rowCount, int columnCount):rowCount(rowCount),
columnCount(columnCount){//初始化普通构造，其实只是把手动的构造一个确定大小的二维数组
的过程进行了合理的封装。
            data = new E*[rowCount];
            for (int i = 0; i < rowCount; ++i) {
                data[i] = new E[columnCount] {} ; //{}表示初始化为0
                // memset(data[i], 0, sizeof(E)*columnCount);
            }
        }
        ~Matrix(){
            for (int i = 0; i < rowCount; ++i) {
                delete [] data[i];
            }
            delete[] data;
        }
    };
    template<typename E>
    Matrix<E>* _concurrent_continuous_multiplication(const Matrix<E>&
matrix_a, const Matrix<E>& matrix_b){
        auto result = new Matrix<E>(matrix_a.rowCount,
matrix_b.columnCount);
#pragma omp parallel for
        for (int i = 0; i < matrix_a.rowCount; ++i) {
            for (int k = 0; k < matrix_a.columnCount; ++k){
                E temp = matrix_a.data[i][k];
                for (int j = 0; j < matrix_b.columnCount; ++j) {
                    result->data[i][j] += temp*matrix_b.data[k][j];
                }
            }
        }
        return result;
    }
    template<typename E>
    Matrix<E>* _continuous_multiplication(const Matrix<E>& matrix_a, const
Matrix<E>& matrix_b){
        auto result = new Matrix<E>(matrix_a.rowCount,
matrix_b.columnCount);//auto只是为了防止抄太多遍。
        //i和k如果是固定的，那么只是对j进行连续操作。
        for (int i = 0; i < matrix_a.rowCount; ++i) {
            for (int k = 0; k < matrix_a.columnCount; ++k){
                E temp = matrix_a.data[i][k];
                for (int j = 0; j < matrix_b.columnCount; ++j) {
                    result->data[i][j] += temp*matrix_b.data[k][j];
                }
            }
        }
        return result;//复制地址，而内存因为在堆空间上，所以没有被释放。释放的责任移交
给了外面调用我这个函数的函数。避免了对内容进行复制。
//        return new Matrix<E>{result, matrix_a.rowCount,
matrix_b.columnCount};
    }
```

```
    template<typename E>
    Matrix<E>* _trivial_multiplication(const Matrix<E>& matrix_a, const
Matrix<E>& matrix_b){
        auto result = new Matrix<E>(matrix_a.rowCount,
matrix_b.columnCount);
        for (int i = 0; i < matrix_a.rowCount; ++i) {
            for (int j = 0; j < matrix_b.columnCount; ++j) {
                for (int k = 0; k < matrix_a.columnCount; ++k) {
                    result->data[i][j] += matrix_a.data[i]
[k]*matrix_b.data[k][j];
                }
            }
        }
        return result;
    }
    template<typename E>
    Matrix<E>* operator*(const Matrix<E>& matrix_a, const Matrix<E>&
matrix_b) //因为我们不实现乘等于，所以要返回指针
    {
        if (matrix_a.columnCount != matrix_b.rowCount)
            throw std::runtime_error("Matrix multiplication of whose sizes
do not match is not supported.");
        return _concurrent_continuous_multiplication(matrix_a, matrix_b);
    }
    template<typename E>
    Matrix<E>* readMatrixFromFile(const char* filename)
    {
        FILE* file = fopen(filename, "r");
        int rowCount;
        char buffer[262144];//每一行可能有这么多个字节

        if (file == nullptr) {
            throw std::runtime_error("file not found.");
        }
        E** lines = new E*[matrix_max_size];
        E element; std::stringstream ss;
        int columnCountMax = 0;
        for(rowCount = 0;fgets(buffer, sizeof(buffer), file) !=
nullptr;++rowCount){
            int columnCount = 0;
            E* line = new E[matrix_max_size];
            char* token = strtok(buffer," "); //不能正则。这个思想是个指针，一步步
走，把字符改成\0。
            while(token!=nullptr){
                if (std::is_same<E, double>::value){  //泛型对具体类型执行不同操
作.
                    //不是fscanf，是sscanf
                    sscanf(token, "%lf", &element);
                } else{
                    sscanf(token, "%f", &element);
                }
//                lines[rowCount][columnCount++] = element;//栈上数组不可取。
                line[columnCount++] = element;
                token = strtok(nullptr, " ");
            }
            lines[rowCount] = line; //一行的数据读完了。
            if (columnCount>columnCountMax)
                columnCountMax = columnCount;
```

```
        }
        fclose(file);
        return new Matrix<E>{lines, rowCount, columnCountMax};
    }
    template<typename E>
    void writeMatrixToFile(const Matrix<E>* matrix, const char* filename)
    {
        FILE* file = fopen(filename, "w");
        for (int i = 0; i < matrix->rowCount; ++i) {
            int j;
            for (j = 0; j < matrix->columnCount-1; ++j) {
                if (std::is_same<E, double>::value){  //泛型对具体类型执行不同操
作
                    fprintf(file, "%lf ", matrix->data[i][j]);
                } else{
                    fprintf(file, "f ", matrix->data[i][j]);
                }
            }
            if (std::is_same<E, double>::value){  //泛型对具体类型执行不同操作
                fprintf(file, "%lf\n", matrix->data[i][j]);
            } else{
                fprintf(file, "%f\n", matrix->data[i][j]);
            }
        }
        fclose(file);
    }
     template<typename E>
     void file_matrix_multiplication(char* in_out_filenames[]) //TODO:为什么不
能是const
     {
         Matrix<E> matrix_a = *readMatrixFromFile<E>(in_out_filenames[0]);
         Matrix<E> matrix_b = *readMatrixFromFile<E>(in_out_filenames[1]);
         Matrix<E>* matrix_c = matrix_a*matrix_b;
         writeMatrixToFile(matrix_c, in_out_filenames[2]);
         delete matrix_c;
     }
}
#endif //P_PROJECT2_MATRIX_MULTIPLICATION_HPP
```

- matmul_benchmark.cpp

```
//
// Created by 叶璨铭 on 2021/9/29.
//
#include <iostream>
#include <sstream>
#include <benchmark/benchmark.h>
#include "matrix_multiplication.hpp"
#define matrix_type double
using namespace matrix_multiplication;
//测试乘法所需时间
static void testMultiplication_trivial(benchmark::State& state){
//    数据准备
    int size = state.range(0);
    char path_a_in[256];
    char path_b_in[256];
    char path_c_out[256];
```

```
    sprintf(path_a_in,"./data/mat-A-%d.txt", size);
    sprintf(path_b_in,"./data/mat-A-%d.txt", size);
    sprintf(path_c_out,"./data/mat-C-%d.txt", size);
    Matrix<matrix_type> matrix_a = *readMatrixFromFile<matrix_type>
(path_a_in);
    Matrix<matrix_type> matrix_b = *readMatrixFromFile<matrix_type>
(path_b_in);
    Matrix<matrix_type>* matrix_c;
//    开始测试
    for (auto _: state) {
        matrix_c = matrix_a*matrix_b;
    }
    writeMatrixToFile(matrix_c, path_c_out);
    delete matrix_c;
}
BENCHMARK(testMultiplication_trivial)->Arg(32)->RangeMultiplier(2)-
>Range(32, 2048)->Iterations(5)->Complexity(benchmark::oN);

//测试读入一个矩阵所需时间
static void testInput(benchmark::State& state){
//    数据准备
    int size = state.range(0);
    char path_a_in[256];
    sprintf(path_a_in,"./data/mat-A-%d.txt", size);
//    开始测试
    for (auto _: state) {
        Matrix<matrix_type> matrix_a = *readMatrixFromFile<matrix_type>
(path_a_in);
    }
}
BENCHMARK(testInput)->Arg(32)->RangeMultiplier(2)->Range(32, 2048)-
>Iterations(5)->Complexity(benchmark::oN);

//测试写出一个矩阵所需时间
static void testOutput(benchmark::State& state){
//    数据准备
    int size = state.range(0);
    char path_a_in[256];
    char path_a_out[256];
    sprintf(path_a_in,"./data/mat-A-%d.txt", size);
//    std::stringstream _path_a_out;
//    _path_a_out<<"./data/mat-A-"<<size<<"_output_test.txt";
//    char * path_a_out = _path_a_out.str().c_str();
    sprintf(path_a_out,"./data/mat-A-%d_output_test.txt", size);
    Matrix<matrix_type>* matrix_a = readMatrixFromFile<matrix_type>
(path_a_in);
//    开始测试
    for (auto _: state) {
        writeMatrixToFile(matrix_a, path_a_out);
    }
}
BENCHMARK(testOutput)->Arg(32)->RangeMultiplier(2)->Range(32, 2048)-
>Iterations(5)->Complexity(benchmark::oN);
BENCHMARK_MAIN();
```

- CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(P_Project2)

add_compile_options("$<$<C_COMPILER_ID:MSVC>:/utf-8>")
add_compile_options("$<$<CXX_COMPILER_ID:MSVC>:/utf-8>") #告诉msvc编译器，代码文件是utf-8编码的。

set(CMAKE_CXX_STANDARD 14)

add_executable(P_Project2
        main.cpp
        )

add_executable(R_Project2
        matmul_benchmark.cpp
)
find_package(benchmark REQUIRED)
target_link_libraries(R_Project2 PRIVATE benchmark::benchmark)

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    target_link_libraries(P_Project2 PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(R_Project2 PUBLIC OpenMP::OpenMP_CXX)
endif()
```

Part 3 - Result & Verification

3-1 正确性测试实验

3-1-0 实验原理与实验假设

“矩阵算的不对”或者“算得不准确”大体可以分为两种情况：

- 程序逻辑出现问题。
  - 这可能是因为语义与预期不一致，或者循环、递归的方法不能与矩阵的抽象形式进行适配。
  - 比如，使用python的numpy去计算矩阵(3-1-1)的是时候，容易犯这样的错误
    - ```
      matrixC = matrixA*matrixB                # wrong
      matrixC = np.matmul(matrixA, matrixB)    # correct
      ```
    - 前者的语义实际上是“矩阵元素逐个相乘”，与后面(3-1-2)中我们用到的 matrix*10 类似。目标是生成一个想要的矩阵。
    - 后者的语义才是我们所期望的矩阵乘法运算。
- 浮点数计算带来的误差
  - 于老师上课提到过，浮点数能表示的数是有限的。但是有理数（无限循环小数，有限小数）却又有可数无穷个，无论计算机（人类）用何种方式抽象有理数都不可能将所有情况表示清楚。[1]
  - 因此，必然带来误差。
  - 浮点数是一个很好的解决方案，已经是尽可能多的让有限的内存表示更多的数了。
    - 如果按照定点数来存储，会有很多一样的小数，但是有不同的存储方式
    - 如果按照分数（即两个整数来存储），也会相同小数不同存储方式的问题。
- 并发计算带来的不可复现误差
  - 根据IEEE浮点数规范，(a+b)+c 不保证与 a+(b+c)相等
      ```

    - 在本次project中，我们运用了多线程来执行并发计算，这就意味着，我们计算结果矩阵的某一个元素时，假设了(a+b)+c ==a+(b+c)， 从而认为并发计算（不保证顺序执行）的结果是近似正确的。
    - 然而实际上是有误差的。
    - 这种误差是不可复现的，因为每次并发计算的顺序行为都不相同，只能说有概率发生。
    - 这种误差的根本来源仍是浮点数计算带来的误差。

这三种情况，都是我们需要考虑的。如何分析其中的误差，其实，这是一个很复杂的科学问题。经过论文搜索[2]，主要有以下几点

- 在舍入操作之前，有一个中间状态是保留结果的
- 舍入操作是误差的来源。根据舍入的四种模式，可以计算出每次算术操作相对误差的最大值。
- 有上溢和下溢两种情况
- **矩阵的加减法、乘法的误差可以用矩阵的范数来刻画**
- 无论对于float还是double，有以下公式
- 有圈的是浮点截断值，没有圈的是真实值
- $\|\circ(A\cdot B)-(A\cdot B)\| \le \gamma_n \cdot \sqrt{\sum_{\substack{1\le i\le n\\ 1\le j\le n}} \left(\sum_{k=1}^{n}|a_{ik}b_{kj}|\right)^2}$ [2]
- 范数是一个矩阵对应的一个数。[2]

3-1-1 实验器材-参考答案生成

- 我们需要一个数学意义上的准确矩阵计算结果，才能够比较误差。当然，如果有些问题不存在数学意义的准确矩阵计算结果，那么我们就只能近似。
    - 这里我们虽然可以计算准确的矩阵，但是代价很高。
        - 我尝试用Java的BigDecimal写了一个简单的矩阵乘法，在运算2048 10分钟之后，java堆空间不足而退出。
    - 因此，我们认为，**python的numpy库作为成熟的矩阵运算库，其计算结果可以作为参考答案**（指定用double类型）。
- matmul.py

```
import numpy as np
if __name__ == '__main__':
    size = 2048
    #type = np.typeDict['f']
    type = np.typeDict['d']

    pathA = "../data/mat-A-%d.txt" % size
    pathB = "../data/mat-B-%d.txt" % size
    pathC = "../data/mat-C-{0:d}_by_python_{1}.txt".format(size, "float" if
type == np.typeDict['f'] else "double")
    matrixA = np.loadtxt(pathA, dtype=type, delimiter=" ")
    matrixB = np.loadtxt(pathB, dtype=type, delimiter=" ")

    # matrixC = matrixA*matrixB                # wrong
    matrixC = np.matmul(matrixA, matrixB)      # correct
    np.savetxt(pathC, matrixC, fmt='%f', delimiter=" ")
```

3-1-2 实验器材-数据生成

- 老师所给的数据当中，只有32、256和2048大小的。
  - 这里需要产生64, 128, 512, 1024大小的数据。
  - 再次使用numpy
- random_matrix_generator.py

```
import numpy as np
if __name__ == '__main__':
    for size in [64, 128, 512, 1024]:
        pathA = "../data/mat-A-%d.txt" % size
        pathB = "../data/mat-B-%d.txt" % size
        matrixA = np.random.rand(size, size) * 100 #每一个元素在[0,100]的范围内
        matrixB = np.random.rand(size, size) * 100
        np.savetxt(pathA, matrixA, fmt='%.1f', delimiter=" ") #和老师给的数据一
致，保留一位小数
        np.savetxt(pathB, matrixB, fmt='%.1f', delimiter=" ")
```

3-1-3 实验器材-计算结果差异比较

- 首先，我们比较元素的时候，如果元素之间差异很小，可认为是相等的。
  - 而对于两个矩阵而言"不相等"的元素越多，精确度就越低。
  - 考虑到矩阵大小的相对性，还需要除去总共有多少个元素。
  - 因此，我们可以这样来定义精确度：
  - accuracy = 1- errorCount/size ^{2} x 100%
- 其次，根据3-1-0查阅的资料，可以用范数之差除以准确范数来描述相对误差。（注意，范数本身的计算是用double来算的，本身也是有误差的，该论文没有指出方案。）
- 故形成python代码如下：

```
import numpy as np
import math

if __name__ == '__main__':
    # 参数
    theta = 0.1  # 每一个数认定相等的误差。
    for size in [32, 64, 128, 256, 512, 1024, 2048]:
        errorCount = 0  # 有多少个点是错误的。# 要清零!
        print("testing size %d"%size)
        # type = np.typeDict['f']
        type = np.typeDict['d']
        # 读取
        path_expected = "../data/mat-C-%d.txt" % size
        path_actually = "../data/mat-C-%d_float.txt" % size
        matrixA = np.loadtxt(path_expected, dtype=type, delimiter=" ")
        matrixB = np.loadtxt(path_actually, dtype=type, delimiter=" ")
        # 比较
        for i in range(size):
            for j in range(size):
                difference = math.fabs(matrixA[i][j] - matrixB[i][j])
                if difference > theta:
                    # print("Expected: {0:f}, actually {1:f}, at\n ({2:d},
{3:d})".format(matrixA[i][j], matrixB[i][j], i, j))
                    errorCount += 1
        print("Your result has %d elements that is different." % errorCount)
```

```
        print("Your computation accuracy is %f %%" % ((1 - (errorCount /
(size ** 2))) * 100))

        a = np.linalg.norm(matrixA)
        b = np.linalg.norm(matrixB)
        print("The norm values are expected {0:f} and actually {1:f}, and
the difference is {2:f}".format(a, b, a - b))
        print("Your computation accuracy is %f %%" % ((1 - (a - b) / a) *
100))
```

3-1-4 实验内容1：程序逻辑正确性验证

对python答案和c++答案进行diff.py, 结果如下：

- python double和c++ double对打

```
In[2]: runfile('D:/EnglishStandardPath/Practice_File/P_C_Cpp/P_Project2/python_help/diff.py', wdir='D:/Englis
testing size 32
Your result has 0 elements that is different.
Your computation accuracy is 100.000000 %
The norm values are expected 2576761.843612 and actually 2576761.837265, and the difference is 0.006347
Your computation accuracy is 100.000000 %
testing size 64
Your result has 0 elements that is different.
Your computation accuracy is 100.000000 %
The norm values are expected 10300483.950499 and actually 10300483.986501, and the difference is -0.036002
Your computation accuracy is 100.000000 %
```

使用表格进行数据整理：

数据量:	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
errorCount（相异阈值为0.1）	0	0	0	0	0	0	0
accuracy1	100%	100%	100%	100%	100%	100%	100%
norm difference	0.006347	-0.036002	-0.024216	-0.008653	-0.01546	-0.039922	-0.012758
accuracy2	100%	100%	100%	100%	100%	100%	100%

可以看到，相异阈值是很宽松的，两个矩阵在每个元素0.1的误差范围内，是近似相等的。因此，可以认为c++程序计算矩阵乘法的逻辑是正确的，编程没有错误。

3-1-5 实验内容2：float和double计算精度的差异比较

- c++ double 与 float对比，结果如下：

```
testing size 32
Your result has 0 elements that is different.
Your computation accuracy is 100.000000 %
The norm values are expected 2576761.837265 and actually 2576761.886708
Your computation accuracy is 100.000002 %
testing size 64
Your result has 466 elements that is different.
Your computation accuracy is 88.623047 %
The norm values are expected 10300483.986501 and actually 10300484.0086
Your computation accuracy is 100.000000 %
testing size 128
Your result has 2871 elements that is different.
Your computation accuracy is 82.476807 %
The norm values are expected 40804024.928926 and actually 40804025.1575
Your computation accuracy is 100.000001 %
testing size 256
Your result has 27302 elements that is different.
Your computation accuracy is 58.340454 %
The norm values are expected 163955432.405096 and actually 163955433.15
Your computation accuracy is 100.000000 %
testing size 512
```

使用表格进行处理：

数据量:	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
norm difference	-0.049444	-0.021586	-0.228665	-0.752011	-0.978473	-1.587534	-9.757679
accuracy norm	100%	100%	100%	100%	100%	100%	100%
errorCount(0.1)	0	466	2871	27302	220626	997184	4051253
accuracy(0.1)	100.00%	88.62%	82.48%	58.34%	15.84%	4.90%	3.41%
errorCount(1)	0	0	0	0	4163	411236	3208255
accuracy(1)	100%	100%	100%	100%	98%	60.78%	23.51%

分析：

- 从范数的角度来看，范数差始终很小，相比矩阵的大小微不足道。但是范数依然在不断增加。
    - 在上一个实验中，范数差并没有随着规模的增大而增大。
- 从相异阈值的角度来看，矩阵计算的准确率在逐渐下降，而且越降越快

结论：

- 本次实验表明，浮点数误差会随着计算量的增大而逐步增加。
    - 也可以理解为，**浮点数所表示的数越大，该数的值从定点的角度来看，就越不准确**
    - 比如说1e20+1等于1e20，那么从我们刚才的作差小于1角度来说，对于1来说它是不保证精确性的。
    - 比如，如果计算误差一下，让1e10变成了1.1e20, 那么他们的差就很容易大于1
    - 就算不是1.1e20，而是1.00001e20, 他们的差也依然大于1
- 本实验还表明，python的numpy矩阵乘法和c++我写的矩阵乘法，除了并发计算时有一定的随机性，是完全等价的。
    - 因为上一个实验中的误差远远小于这一次实验。上一次实验的范数是想等的。
- 但是这种误差是并不是特别严重。
- 从范数的理论分析来看，**本实验数据印证了，浮点数矩阵的计算误差是收敛的。**
    - 从实验数据来看，可以推断出**范数差增长的速率远远不如矩阵大小（范数）增长的速率**，
    - 因为即使范数差增加了，计算出来的精确度也依然能保持在100%

3-2 性能测试实验

3-2-0 实验原理

见4-4。

3-2-1 实验器材-时间测试框架

- 在上次project，我们讨论了进程测试和函数测试两种不同的测试方法，分析发现进程测试的误差较大。我们尝试了c++11标准的chrono库，并且写了一个stopwatch类来对函数进行计时。
  - 在这次project，我们尝试使用成熟的cpp性能测试框架-google benchmark这一工具来评测我们程序的性能。
  - google benchmark 是google开源的一个用来对代码片段进行基准测试的库，对于不同数据规模下程序能跑多快能够做很方便、准确的测定。
- 那么，如何使用google benchmark到我们的测试当中呢?
- 首先，我们要了解一下google benchmark的基本用法[3]
  - ```
    //benchmark_demo.cpp
    #include <iostream>
    #include <cstring>
    #include <benchmark/benchmark.h> //benchmark的头文件
    //被测试的函数
    static void test(benchmark::State& state){
        //数据准备
        auto i = state.range(0); //当前测试的数据规模
        //开始测试
        for (auto _: state) {//for内的代码会被计时。for会执行iterations指定的次数。
            int a[i];
            memset(a,0,sizeof(int)*i);
        }
    }
    BENCHMARK(test)->Arg(32)->RangeMultiplier(2)->Range(32, 2048)->Iterations(10); //Range2表示每次数据规模加倍；32和2048表示最小和最大的数据规模。
    BENCHMARK_MAIN(); //代替main，输出测试结果
    ```
- 其次，要成功编译程序。
  - ```
    sudo apt install libbenchmark-dev
    g++ benchmark_demo.cpp -o benchmark_demo -pthread -lbenchmark
    ./benchmark_demo [--benchmark_out="result.txt"]
    ```
- 最后，我们需要理解google benchmark的输出结果，为进一步数据分析作参考。
  - ```
    Benchmark                        Time             CPU   Iterations
    ----------------------------------------------------------------------
    test/32/iterations:10          340 ns          150 ns           10
    test/32/iterations:10          110 ns         80.0 ns           10
    test/64/iterations:10          120 ns         90.0 ns           10
    ```
  - Time 包括了等待io操作的阻塞时间，而CPU只是该进程的实际CPU时间，更加准确。[4]
  - 注意这些都是**平均时间。**
  - 我们**观察到，第一次32的时间无论是time还是CPU，都会远高于后面的时间，即使它的数据规模是最小的。**这说明，**第一次的数据是不准确的。**
    ```

- 这可能是因为cpu在第一次运行之后，有了缓存。[5]
- 所以，我们在Range前面增加了一个Arg(32)，强制先执行一次实验，让系统进行缓存.（不知道有没有别的更好的办法）
- 这样，之后的时间测试就是每个规模的测试都是由缓存的，时间测量的结果只与我们关心的算法处理相应规模数据的方式有关，减少了误差。

3-2-2 算法测试模型

- 问题规模
  - 本次问题规模的定义，就是方阵的行数或者列数。
- 倍率测试
  - 和上次project一样。不过这次我们通过google benchmark这一利器来实现倍率增长。

3-2-3 实验分析0-乘法速度与io速度

运行benchmark。获得double在连续、多线程算法的数据如下：

```
-------------------------------------------------------------------------------
Benchmark                                     Time             CPU   Iterations
-------------------------------------------------------------------------------
testMultiplication/32/iterations:5      1840780 ns      1841060 ns            5
testMultiplication/32/iterations:5       705340 ns       705560 ns            5
testMultiplication/64/iterations:5      1282440 ns      1282960 ns            5
testMultiplication/128/iterations:5     4225400 ns      4225740 ns            5
testMultiplication/256/iterations:5    20263000 ns     20263960 ns            5
testMultiplication/512/iterations:5   121132060 ns    111021140 ns            5
testMultiplication/1024/iterations:5  855369240 ns    823218380 ns            5
testMultiplication/2048/iterations:5 4913742840 ns   4779046120 ns            5
```

```
testInput/32/iterations:5               5002200 ns      1101540 ns            5
testInput/32/iterations:5               4740420 ns       994180 ns            5
testInput/64/iterations:5               6378260 ns      1972080 ns            5
testInput/128/iterations:5             15444260 ns      6741040 ns            5
testInput/256/iterations:5             45240880 ns     23979380 ns            5
testInput/512/iterations:5            166232740 ns     92021860 ns            5
testInput/1024/iterations:5           663384880 ns    366840380 ns            5
testInput/2048/iterations:5          2481224680 ns   1428505000 ns            5
```

```
testOutput/32/iterations:5              6477840 ns      1168760 ns            5
testOutput/32/iterations:5              6953980 ns      1236280 ns            5
testOutput/64/iterations:5              9631960 ns      2715340 ns            5
testOutput/128/iterations:5            22937380 ns      9047140 ns            5
testOutput/256/iterations:5            79581020 ns     35085340 ns            5
testOutput/512/iterations:5           302382040 ns    139194520 ns            5
testOutput/1024/iterations:5         1183304680 ns    539132260 ns            5
testOutput/2048/iterations:5         4746463800 ns   2175960280 ns            5
```

得到表格：

数据量	32	64	128	256	512	1024	2048
乘法占比	0.056883451	0.074160945	0.099171405	0.139663052	0.205396709	0.316562038	0.404708696
读取占比	0.382299951	0.36884204	0.362481412	0.311823491	0.281871353	0.245510897	0.204360146
写出占比	0.560816598	0.556997015	0.538347184	0.548513457	0.512731938	0.437927065	0.390931158

得出结论：

- **写出数据比读取数据要慢一倍左右。**
- **乘法算法随着问题规模增大增长的速度比读取、写出矩阵算法的增长的速度要快。**

3-2-4 实验分析1：探究 内存连续遍历 对矩阵乘法运算速度的优化。

接下来我们不再需要测试IO，只测试乘法。

结果如下：

数据量	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
平凡算法(ns)	260640	2021500	16998800	155229700	1408504420	12031226420	90144119320
连续性优化算法(ns)	176460	1513760	12910060	99875780	816831220	4155768640	32187978760

分析1：提速率 =(t(平凡算法)-t(连续性优化))/t(平凡算法)

数据量	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
提速率	0.322974217	0.251169923	0.240531096	0.356593616	0.420071951	0.654584787	0.642927581

可以看到，随着规模得增大，**连续性优化算法的优势越来越明显。**最后似乎要收敛在64%

分析2：复杂度

log2(倍率) 平凡算法	**2.95529579**	**3.071934797**	**3.190899803**	**3.181687572**	**3.094547717**	**2.905449665**
log2(倍率) 优化算法	3.100723395	3.09228731	2.951639159	3.031831238	2.347005425	2.953334742

无论是平凡算法还是优化算法，复杂度都是3次方。

3-2-5 实验分析2：探究 多线程 对矩阵乘法运算速度的优化。

数据如下：

数据量:	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
多线程连续	705340	1282440	4225400	20263000	121132060	855369240	4913742840
连续	176460	1513760	12910060	99875780	816831220	4155768640	32187978760

数据量:	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
提速率	-2.997166497	0.152811542	0.672704852	0.79711798	0.851704909	0.794173037	0.847342299

从实验数据可分析，多线程连续算法那在数据量小的时候优势不大，反而因为需要管理线程而速度慢（7个线程慢7倍）。

而在数据量较大的时候，**多线程算法可以提速80%左右。**

3-2-6 实验分析3：探究 float和double矩阵乘法运算速度的区别。

以上测试都是在double的情况下做的。

下面对float进行测试，与double表格合并得到数据如下：

数据量:	32	64	128	256	512	1024	2048
提速率:	-0.921995066	0.279264527	-0.141094334	0.234678972	-0.121412614	0.036011583	0.018372891

测试平台在Windows wsl下，使用g++编译。

得出结论，在该平台下，可以认为double与float计算速度相当，没有区别。

Part 4 - Difficulties & Solutions

4-1 如何像一个系统工具一样，给用户提供良好的使用接口?

一个良好的系统级矩阵乘法器，不仅要计算速度很快，还应该有良好的使用接口。

- 就像g++编译程序的时候，我们可以指定被编译的文件路径，编译的选项，是编译还是链接，还是异步到位等，而编译如果出错了，或者输入的命令行不对，g++也会把错误信息告诉我们。
- **矩阵乘法程序也应该实现这样的功能：可以指定是按照单精度浮点数模式，还是双精度浮点数模式来进行计算。**
- 在本次project，我将模仿g++的用户接口提示，自己实现一个较为通用而高效的命令行参数处理。

4-1-1 如何优雅地处理命令行选项?

- 什么是命令行选项？一般有什么规范?
  - 经过调查，命令行工具为了一个程序能够实现多种功能，往往会允许用户传递命令行参数。
    - 命令行参数能够改变命令行工具的状态与行为
    - 命令行工具不像函数，没有约定的函数参数传递方法。为什么不设计成函数，我没找到根源。
  - 正是因为命令行工具是不是函数，所以“第几个参数代表什么意思”这件事无法确定
    - 于是衍生出命令行选项的概念。
    - 命令行选项是特殊的一个参数，通常以'-'开头，跟上一个字母
      - 在Unix、linux系统上，命令行选项往往分为缩写和全称, 全称选项用两个减号表示。
      - 比如，"-h"是"--help"的缩写
      - 在Windows系统上，命令行选项可以以"/"开头。最新的powershell引入了.net技术，命令行参数可以传递对象（高于字符串的抽象概念），命令行工具可以返回对象（有结构的结果，不需要使用类似grep函数来进行解析）。
    - **一个选项有时候代表了“函数”里面的一个参数变量，将紧跟其后的一个或多个参数当成其值。**
    - 比如假设有一个结束进程的函数，它可能是这样写的：
      - ```
        int stop_process(const char* processName, const bool
        force_kill);//声明
        stop_process("explorer.exe", true);//调用
        stop_process( true, "explorer.exe");//错误，不符合函数调用的约定
        ```
    - 那么命令行工具，或者脚本语言往往是这样写的：
        ```

        - ```
          taskkill {/IM imagename} [/F] #帮助， 大括号表示必须，中括号表示可选
          taskkill -im explorer.exe -f #调用
          taskkill -f -im explorer.exe #正确，-f不期待参数，-im期待一个参数，-im和-f的相对顺序可以调换。
          ```

- g++有哪些功能
    - 在用户没有输入待编译的文件，或者输出文件与输入文件完全一样时，会有相应的提示
    - 对于编译命令选项，-O2会打开编译优化，-Wall会输出更多的报错信息。
    - 编译过程找到源码错误，会报错终止编译。
- 矩阵乘法程序的逻辑
    - 接受三个参数，表示矩阵输入和输出的文件。
    - **提供指定类型的选项-type，如果指定为float，就会按照float去读取矩阵进行乘法运算；如果指定为double，就用double。**
    - 提供-help。
- 实现方法
    - for循环。
        - 如果遇到了选项，判断是有参数选项还是无参数选项。
        - 有参数选项，则将循环提前后移多处理一格。
    - c语言的switch。
        - 与java不同，c语言的switch只能switch整数（包括char、bool等[1]），这是因为c语言的switch可以简单理解成是用跳转表实现的。[6]下标是整数方便。
        - java是通过计算对象的hashcode得到整数的。
        - 因此，我们不应该对整个char*进行switch
        - 只能switch字符
- c风格的字符串处理[1]
    - 为了提高处理速度，我尝试不使用cout、sstream、string, 学习原始的c语言处理字符串的方法。
    - 了解到c风格的字符串处理的基本思想如下。
    - 处理开始时，往往要设立一个“buffer”， buffer的大小是猜测的最大大小。
    - strcat可以把新的字符串拼接到buffer上面，注意，不是像+函数的逻s辑那样返回一个新的字符串，而是把buffer本身“+=”了，buffer不能是一个常量。
    - strlen是通过遍历直到找到\0来获得长度的，不能当做java的array.length来使用，尤其不能在for循环中用strlen来遍历char*

4-1-2 如何正确输出中文提示语

> 这里我们会从源码、编译器、char类型、控制台等多个层次探讨分析，输出中文乱码的问题到底出在哪里。

- 字符集与编码[7]
    - 字符集是一个映射，把字符与唯一的编号关联起来。
    - 编码也是一个映射，把字符与唯一的二进制码关联起来。
    - 为了节省空间，编码可以不按照字符集的编号来进行关联。
- 源码文件本身是用utf-8来存的，里面的每一个字符是二进制被解释为utf-8，再被编辑器解释为图形界面的结果。
- 编译器
    - 编译器g++按照utf-8来理解源码文件
    - msvc按照gbk来理解源码文件。
          ```

    - 目前看来，只要逻辑写的是对的，编译器按照正确的方式理解源码，字符串常量就能正确的被编译。[8]
- 终端
    - Windows控制台默认是gbk编码。
    - 可以通过Windows.h中的函数，在我们的代码中把我们这个程序的编码设置为utf-8.
    - 这样，编译时的常量，在运行时能够被终端正确理解而显示出来。

4-1-3 跨平台编译

- 4-1-1中，我们对字符串进行了strcat的处理
    - strcat函数有两个条件，dest的空间要能容纳src，且两者内存不能重叠。[9]
    - 但是strcat函数本身没有负起这个责任去检查（为了性能），导致了程序的不安全。
    - 微软msvc带头支持了**c11的新标准：strcat_s**
    - 这个函数可以实现近乎零开销的安全检查。
    - 但是在**linux下用gcc g++编译，是不能识别strcat_s的。**
    - 因此，我们需要进行条件编译。
- 具体方法就是，如果宏检测到当前环境是Windows，就把str_cat换成strcat_s，否则换成strcat
    -
```
#ifdef WIN32
//windows
#include <Windows.h>
#define str_case_cmp _stricmp
#define str_cat strcat_s
#define str_cpy strcpy_s
#define _CRT_SECURE_NO_WARNINGS
#elif __linux__
// linux
#define str_case_cmp strcasecmp
#define str_cat strcat
#define str_cpy strcpy
#endif
```
    - 有的同学是这样替换的，他在代码里面写的是strcat_s，而在Windows下不替换，linux下替换成strcat。
    - 这样也行，但是万一有一天linux的g++实现了strcat_s（毕竟是c11标准，不是微软瞎编的），#include 写的位置又不太对，那么**就会出现把系统库文件字符串修改的情况，是很危险的。**

4-3 如何抽象矩阵？如何让矩阵输入输出更加高效？

> 这里我们研究、探讨数组、动态数组、变长容器的原理，并初步探究模板如何使用。

数据结构是算法的载体，便于算法的表达。

- 在搜集资料的过程中，我看到网上很多种矩阵乘法的写法，其中还有一些使用到Strassen算法。它们大部分都是用最基本的指针、数组来表达矩阵的，在书写算法的过程中，由于有些算法逻辑比较复杂，网上的代码往往需要很多变量来存储不同的指针、变量、数组，并且复制粘贴了一些for代码，不仅导致代码难以维护、阅读，还涉及到一些cpp语法使得这些算法在不必要的地方进行了错误的操作（比如意外析构、复制）。
- 为了解决这些问题，防止我自己的代码出现这样的错误，而让算法更好地被数据结构所表达，
    - 通过查阅c++书籍、查看complex类、vector类的定义、搜索c语言的解决方案等，
    - 我仔细研究了在c/c++中如何对矩阵进行合理抽象。

4-3-1 数组

- 数组是连续的内存空间。[1]
  - 数组的这一特性，就决定了**数组的大小必须在某一时刻被确定。**
  - 如果数组的大小不确定，后面新的变量的空间怎么分配，分配前了，挡住了数组的去路，分配后了，又会浪费空间。
  - 无论是int a[10], 还是int* a = new int[10]，都不可避免地要限制申请内存的大小是有限的、定长度的。
  - 否则，我们就需要重新申请内存，然后把已有的数据搬运过去。
- **二维数组是连续的多个一维数组**
  - 二维数组的第二行数组，是紧接着第一行数组的内存空间申请而申请的，因此内存空间是连续的。
  - 从第一行第一个调到第一行第二个是很快的。但是从第一列的第一行跳跃到第二行就很困难了。
  - **因此，不必用一维数组来模拟二维数组，因为二维数组就是一维数组。**
- 有些高级的语法没有调查清楚，暂时认为
  - 函数内创建的栈上数组，不能成功返回。c++11中可以通过移动语义来解决这个问题。（p.65）[10]
  - 函数内创建的堆上数组，可以返回指针。指针的地址被赋值了，但是内容没有赋值。
  - 为了避免这些问题，有时候程序员，特别是C语言中会写void函数，但是第一个参数是要被修改的数组变量。这样，新的数组只能在函数的外面产生，函数只是帮忙做了个修改。

4-3-2 结构体

- **数组的大小，除了栈上数组，是不可知的。**
  - new的时候虽然保存了内存分配的信息，以供delete[]使用。
  - 但是据我目前搜索所知（暂时无法找到权威资料），我们无法获得这个值。
- **结构体可以把一个int值和数组关联在一起，从而保存长度信息**
  - 如果要坚持不用结构体，那么每次**调用函数**的时候，都要自己把**数组的大小信息**传递给函数，十分麻烦。
  - 当然，如果是char*，可以约定\0是数组的结束位置。但是不如保存长度简单。
- 结构体还可以把一些**经常要写的代码和数组绑定在一起。**
  - 比如，初始化二维数组的过程。如果我的矩阵乘法有很多个实现函数，那么每个函数我都要对结果的二维数组进行初始化，要把同样的代码复制粘贴很多次。
  - 如果使用结构体，就可以用构造函数取解决这个问题。
- 结构体还可以**对数组的生命周期进行控制**（p.151）[10]
  - 我们需要在最后一次使用数组的时候，把数组的空间释放。
  - 把数组用结构体封装，就可以保证如果在某个函数内在栈上创建一个结构体类型的矩阵，那么退出这个函数的时候，矩阵的内容就会被调用结构体的析构函数进行释放。

4-3-3 数据视图、c风格文件读写与矩阵的快速输入输出

- 经过初步研究，拿到一个文件时，**我们没办法在O（1）的复杂度内确认矩阵的大小。**
  - 我有一个同学是这样处理的。
    - 先读取一行，统计数量，然后将该数量认定为矩阵的行数与列数。
    - 显然，他的方法**局限于方阵。**
  - 我还有一个同学是这样处理的。
    - 使用vector<vector>，读到一个数据就push_back，最后就确认了大小。
    - 这种方法是靠谱的。但是vector可能速度不快（没有测试过）。
- **因此，我决定用c语言来完成文件读取的过程，并且适当地学习vector重新分配内存的方法**

    - c语言的文件读写不仅速度快[11],而且读取文件的操作也并不比fstream复杂
    - 通过fgets函数，我获得每一行的数据
    - 然后用过strtok函数，获得了每一个元素。
    - 此前在Java当中，通常是用String的split方法来进行的，那个效率其实也很高，没有说复制很多次数据。
    - 对于大小，我先假定了矩阵有4096那么大，然后申请内存空间，把所有元素放进来
    - 这一部分十分复杂，我尝试了很久，逻辑也不对。
    - 所以最后放弃了resize的过程，假定4096是最大值。
    - 用行数和列数来表示矩阵的实际大小（比4096小）。
- 缺点在于，没有实现resize，浪费了内存
- 该方法好处在于
    - 无论如何实现vector，最终计算机是有极限的。4096是我们程序做的一个限制。
        - 只要在4096之内，程序没有问题。
    - 对于小于4096的数组不做resize减少了内存复制的时间，
        - 对于矩阵乘法程序，只有两个矩阵一个结果
        - **浪费并不算严重，反而是性能有提升。**
- 这样的好处还在于**数据视图的实现**
    - 如果要取一个**子矩阵**，我们可以保持原本的矩阵不变，只改变columnCount和rowCount（以及起始位置也要记录）
    - 这就涉及到数据视图的概念
- 经过查询了解到，c++17引入了view的概念。string_view是string的视图，array_view是array的视图。
    - 可以方便地取子串、子数组。

4-4 如何高效地进行矩阵乘法?

> 有了矩阵的表示方法，我们将从多个层次、角度进行探究，如何高效地进行矩阵乘法。
>
> 这部分的内容会提出实验假设，并在Part3验证。

4-4-1 平凡算法

- 根据矩阵乘法的定义，如果一个矩阵A(m by n)与矩阵B(n by p)相乘，需要有m*p个n维向量进行点乘
- 因此时间复杂度是O(mnp)，考虑方阵，时间复杂度为O(n^3)

4-4-2 内存连续性优化——太神奇了

- 观察平凡算法

```
for (int i = 0; i < matrix_a.rowCount; ++i) {
    for (int j = 0; j < matrix_b.columnCount; ++j) {
        E sum = 0;
        for (int k = 0; k < matrix_a.columnCount; ++k) {
            sum += matrix_a.data[i][k]*matrix_b.data[k][j];
        }
        result[i][j] = sum;
    }
}
```

    - 我们发现，对于最内层的for k， k在发生变化，而访问的matrix_a也是连续地在变化

    - 因此，CPU能够提前缓存matrix_a.data的下几个值，提高运行速度。
    - 但是对于matrix_b， 变化是不连续的，需要跳过很多地址。
- 我们需要调换for循环的顺序
    - 但是调换的前提是改了之后结果不变。
    - 在上面的代码中，**i和j的for是可以互换，因为没有依赖关系，但是k和j就不行了。**
    - 怎么办呢?
- 首先，我们首先要**修改j这一层for的代码逻辑**，使其变成k-j可调换的。
    - 观察之后发现，之所以需要sum这个变量
    - **根本原因是result没有初始化为0。**
    - 如果我们在new初始化的时候，加上一个{}, 按照c++11标准，就可以自动、高效的初始化为0

```
data[i] = new E[columnCount] {} ;
```

    - 修改结果如下：

```
template<typename E>
Matrix<E>* _trivial_multiplication(const Matrix<E>& matrix_a, const Matrix<E>& matrix_b){
    auto result = new Matrix<E>(matrix_a.rowCount, matrix_b.columnCount);
    for (int i = 0; i < matrix_a.rowCount; ++i) {
        for (int j = 0; j < matrix_b.columnCount; ++j) {
            for (int k = 0; k < matrix_a.columnCount; ++k) {
                result->data[i][j] += matrix_a.data[i][k]*matrix_b.data[k][j];
            }
        }
    }
    return new Matrix<E>{result, matrix_a.rowCount, matrix_b.columnCount};
}
```

- 其次，我们调换j和k。同时提取出不变的A(i,k).
- 实验假设：修改后的速度快于修改前的速度。（当前理论知识无法预测到底会快几倍，需要实验测定）

4-4-3 多线程优化——充分利用CPU多核处理的优势

- 从矩阵乘法的角度，**之所以可以使用并发**
    - 是因为每个结果矩阵的元素，互相之间都是独立被计算出来的。
    - 即，C(1,2)的计算结果是否已经出现，完全不影响C(100,1024)的计算。
- 这里我们使用openmp这个库来进行并发执行
    - OpenMp提供了对并行算法的高层的抽象描述，我们只需要在程序中使用宏定义来告诉openmp我们要做什么，它就可以自动帮助我们完成相应的并发计算。
    - openmp的基本使用方法如下
        - **#pragma omp parallel**{} 让大括号内的代码被多个线程执行（执行多次，每一次之间没有关系）。
        - **#pragma omp parallel for** 让一个for被多个线程完成遍历（要求for的逻辑上没有先后顺序。多个线程各自有不同的任务，合起来完成了for）
            - 要求for的逻辑上没有先后顺序，正中矩阵乘法的下怀。
        - **#pragma omp critical**关键代码，一次只允许一个线程执行。与atomic类似。
    - 我们这里不需要critical和atomic。

4-4-4 算法层面的优化——从增长数量级的根本上解决问题

- Strassen算法是一个快于O（n^3）的算法。
- 其原理是，分治。
- 我们知道分治算法不一定快。如果只是简单的分块矩阵乘法，根据Master Theorem，依然是O（n^3）
- 而只要减少子问题的个数，无论分治时“治”的时间有多少，只要不超过n^2, 时间复杂度就能得到改善。
- Strassen算法就是利用了很多次矩阵加法减法运算（O（n^2））来减少了子问题的数量，让乘法减少
- 实现了算法复杂度的降低。

Part 5 - Summary

- 经过本次project，
  - 我学习了解了
    - 浮点数以及矩阵乘法的误差
    - python numpy的初步使用。
    - c++语言矩阵的抽象表示方式
    - 矩阵乘法性能的优化方法
    - c风格的字符串处理。
  - 体会了
    - 除了算法的升级之外，程序的其他方面的优化也是很重要的。
    - 第一次体验到课程介绍里面所说的“使程序快几十倍的方法“，意犹未尽。
- 不足之处
  - 时间有限，矩阵类没有写完全，只是一个结构体，最终未能实现Strassen算法。希望以后学了类之后能够很快地复现出来。
    - 矩阵类不会写，主要问题在于内存管理不懂、指针不懂、拷贝构造概念不理解等。
    - 希望后几节课听课之后能够掌握。
  - 如何只用c++11的std::thread手动实现类似于omp parallel for的功能呢？大概要使用函数代替for，因为thread接受的是函数。还要加锁。本次未能探索std::thread的用法，但是发现了openmp的强大之处。
  - 编码问题知道了怎么写对。但是依然没有完全认清其本质。
  - 经过仔细研究google benchmark的user manual， 仍然没有发现可以在一个test内进行分段计时的方法，因此不同的test有代码是重复的。
  - 报告太难写了，每一个点的问题细究下去都很复杂，很花时间。

参考文献

1. Lecture2.pptx, Lecture4.pptx ↩ShiqiYu/CPP: Lecture notes, projects and other materials for Course 'CS205 C/C++ Program Design' at Southern University of Science and Technology. (github.com) ↩ ↩ ↩ ↩ ↩

2. 曾霞. (2013). 浮点计算程序的误差分析. (Doctoral dissertation, 湖北大学). ↩ ↩ ↩

3. google/benchmark: A microbenchmark support library (github.com) ↩

4. https://blog.csdn.net/dgFlyNl/article/details/112893558 ↩

5. windows - What can cause a program to run much faster the second time? - Stack Overflow ↩

6. C语言switch执行原理,Switch 底层执行原理_抹茶牛奶泡芙的博客-CSDN博客 ↩

7. Unicode 和 UTF-8 有什么区别？ - 知乎 (zhihu.com) ↩

8. char本来不能表示中文，但是字符串在编译器的作用下似乎可以做到。时间有限，资料查阅亦有限，尚未理解其机制。 ↩

9. c语言strcat()/strcat_s()函数详解*aoxuestudy的专栏-CSDN博客*strcat_s()函数 ↩

10. 本贾尼·斯特劳斯特鲁普, 斯特劳斯特鲁普, 王刚,等. C++程序设计语言[M]. 机械工业出版社, 2016. ↩ ↩

11. c++ - Why are std::fstreams so slow? - Stack Overflow ↩