CS205 C/ C++ Programming - Project 3

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 Review Of last project

在上一次project中，我了解了C++语言矩阵的抽象表示方式、矩阵乘法性能的优化方法、浮点数误差以及矩阵乘法的误差分析方法、Python numpy的使用，学习了Google benchmark的使用，探究了内存连续遍历、多线程对矩阵乘法的优化作用，对乱码问题、命令行参数处理有了一个基本的认识。不足的是，未能实现Strassen算法，未了解到SIMD、CUDA等优化方式。

在本次Project当中，我将尝试改进上次project，解决上次未能解决的问题。

1-2 Goals and some problems to consider

- 1.使用C语言，而非C++语言。
  - C语言的编程思想是什么样的?
  - 如何使用CMake去管理一个C语言项目?
- 2.设计矩阵的结构体。并设计操作矩阵的函数
  - 如何用C语言去抽象表示一个矩阵?
  - 如何合理设计函数，使得函数确实能够修改矩阵、创建矩阵等，如何使用指针传值?（lecture5内容）
- 3.优化矩阵乘法的速度，并且提供比Project2更好的优化方式，与OpenBlas进行比较。
  - 如何调用OpenBlas库？CMake如何管理?
  - 有哪些更好的优化方式？原理是什么?
  - 如何实现这些优化方式?

Part 2 - Code

2-1 Project Hierarchy

Practice_Project3:
├─bin　　(二进制可执行文件)
├─include　（库文件的头文件）
│ └─**matmul_openblas.h**
│ └─**matmul_simd_openmp.h**
│ └─**matmul_strassen.h**
│ └─**matmul_trivial.h**
│ └─**matrix_multiplication.h**
│ └─**util.h**
├─lib　　（二进制库文件）
├─cmake-build-debug（cmake debug模式下的编译中间文件)
├─cmake-build-release (cmake release模式下的编译中间文件)
├─resources
│ └─input_data　　(mat-[AB]-[32,64,128,256,512,1024,2048].txt)
│ └─output_data　(mat-[C]-[32,64,128,256,512,1024,2048] [-by_python,cpp].txt)
├─python_help　（与上次project一样。提供基于numpy的生成矩阵、检查矩阵相似度、矩阵乘法等功能）
├─sample（可执行文件的源码）
| └─main.c
| └─matmul_benchmark.c
| └─CMakeLists.txt
├─src　（库文件的与源码）
│ └─matmul_openblas.c
│ └─matmul_simd_openmp.c
│ └─matmul_strassen.c

| └matmul_trivial.c
| └matrix_multiplication.c
| └util.c
| └**CMakeLists.txt**
└**CMakeLists.txt**

2-2 Code

- Note：由于report篇幅有限，本部分**仅包括了**
  - src中的**头文件代码**
  - 和**sample**的c文件代码
- src中对头文件函数的**实现可以在source.zip中找到。**
- 部分**关键实现代码在后文亦有详细的描述与展示。**

├─include

- | └matrix_multiplication.h

```
#ifndef P_PROJECT3_C_MATRIX_MULTIPLICATION_H
#define P_PROJECT3_C_MATRIX_MULTIPLICATION_H
#include "matmul_openblas.h"
#include "matmul_simd_openmp.h"
#include "matmul_strassen.h"
#include "matmul_trivial.h"
enum multiplication_mode {
    TRIVIAL,OPENBLAS,STRASSEN,SIMD,OPENMP,COMBINED
};
void file_matrix_multiplication(char *input1, char *input2, char *output,
enum multiplication_mode mode); //根据不同计算模式来计算矩阵乘法。
#endif //P_PROJECT3_C_MATRIX_MULTIPLICATION_H
```

- | └matmul_openblas.h

```
#ifndef P_PROJECT3_C_MATMUL_OPENBLAS_H
#define P_PROJECT3_C_MATMUL_OPENBLAS_H
#include <cblas.h> //同时要在matmul_openblas.c include，否则链接不到。这是因为本项目中的链接依赖关系是，让sample的main依赖于matmul_blas.dll，但是不直接依赖于openblas。这样可以将调库的过程封装到统一的接口，便于测试。
#include "matmul_trivial.h"//需要借用平凡矩阵的结构体定义
struct TrivialMatrix* matmul_blas(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB);
#endif //P_PROJECT3_C_MATMUL_OPENBLAS_H
```

- | └matmul_simd_openmp.h

```
#ifndef P_PROJECT3_C_MATMUL_SIMD_OPENMP_H
#define P_PROJECT3_C_MATMUL_SIMD_OPENMP_H
#include <intrin.h> //SIMD头文件
#include <omp.h>    //OpenMP头文件
#include "matmul_trivial.h" //需要借用平凡矩阵的结构体定义
struct TrivialMatrix* matmul_simd_openmp(const struct TrivialMatrix*
matrixA, const struct TrivialMatrix* matrixB);
struct TrivialMatrix* matmul_simd_openmp2(const struct TrivialMatrix*
matrixA, const struct TrivialMatrix* matrixB);
struct TrivialMatrix *matmul_simd(const struct TrivialMatrix* matrixA, const
struct TrivialMatrix* matrixB);
struct TrivialMatrix *matmul_simd2(const struct TrivialMatrix* matrixA,
const struct TrivialMatrix* matrixB);
struct TrivialMatrix *matmul_openmp(const struct TrivialMatrix* matrixA,
const struct TrivialMatrix* matrixB);
struct TrivialMatrix* matmul_openmp2(const struct TrivialMatrix* matrixA,
const struct TrivialMatrix* matrixB);
#endif  // P_PROJECT3_C_MATMUL_SIMD_OPENMP_H
```

- | └─matmul_strassen.h

```
#ifndef P_PROJECT3_C_MATMUL_STRASSEN_H
#define P_PROJECT3_C_MATMUL_STRASSEN_H
#include <intrin.h>
#include <omp.h>    //较小的矩阵使用simd和openmp
#include "matmul_trivial.h"  //需要借用平凡矩阵的结构体定义
// interface:
struct TrivialMatrix* matmul_strassen(const struct TrivialMatrix* matrixA,
const struct TrivialMatrix* matrixB);
// data models:
/**
 * usage:
 *  1. As a tuple of two size_t, indicating the size of a matrix/integer
rectangle/integer point/etc.
 *  2. As indices of matrix, serving as the iterator/pointer for the matrix.
 */
struct MatrixCoordinate {
    size_t rowIndex;
    size_t columnIndex;
};
struct MatrixCoordinate* createMatrixCoordinate(size_t rowIndex, size_t
columnIndex);
struct MatrixCoordinate* matrix_coordinate_addition(const struct
MatrixCoordinate coordinateA, const struct MatrixCoordinate coordinateB);
// MatrixCoordinate is a struct without *, so it need not delete.
/**
 * basic rules:
 * 1. Matrix can be seen as a view.
 * 2. Views can generate new computation results that is not a view, as long
as the operation does not change the model behind the
 * view.
 * 3. Destroying views does not change the model.
 * 4. views can change its pointing, as long as it is still a view of the
original matrix.
 */
struct MatrixView {
```

```
    const struct TrivialMatrix* data;
    struct MatrixCoordinate startingCoordinate;
    struct MatrixCoordinate viewSize;  // 5. The coordinate of matrix view
should not be pointer. Since they must not be changed
                                       // by others or change others.
};
struct MatrixView* createViewFromView(const struct MatrixView* view, struct
MatrixCoordinate startingCoordinate,
                                                struct MatrixCoordinate
viewSize);
struct MatrixView* createViewFromTrivialMatrix(const struct TrivialMatrix*
matrix, struct MatrixCoordinate startingCoordinate,
                                                struct MatrixCoordinate
viewSize);
struct MatrixView* ToView(const struct TrivialMatrix* matrix);
#define MatrixViewGetElement(view, i, j)
         \
    (TrivialMatrixGetElement((view)->data, (i) + (view)-
>startingCoordinate.rowIndex, \
                             (j) + (view)->startingCoordinate.columnIndex))
 // core function to explain how view works
// functions for view matmul
struct TrivialMatrix* view_view_strassen(const struct MatrixView*
matrixViewA, const struct MatrixView* matrixViewB);
/**
 * 本来可以把view中的元素拷贝出来形成新的矩阵，然后乘法，但是这就违背了我们设计view的初
衷。
 * 我们想尽可能避免拷贝。于是针对View重新设计一般乘法（矩阵规模较小的乘法）
 * @param matrixViewA
 * @param matrixViewB
 * @return A new Matrix of which data is generated by matrixViewA and
matrixViewB.
 */
struct TrivialMatrix* view_matmul_simd(const struct MatrixView* matrixViewA,
const struct MatrixView* matrixViewB);
struct TrivialMatrix* view_matmul_openmp(const struct MatrixView*
matrixViewA, const struct MatrixView* matrixViewB);
struct TrivialMatrix* view_element_by_element_linear_combinations(const
struct MatrixView* matrixViewA, const struct MatrixView* matrixViewB,
                                                                   float
alpha, float beta);
struct TrivialMatrix* view_addition(const struct MatrixView* matrixA, const
struct MatrixView* matrixB);
struct TrivialMatrix* view_subtraction(const struct MatrixView* matrixA,
const struct MatrixView* matrixB);
struct TrivialMatrix* view_scalar_multiplication(const struct MatrixView*
matrixA, float scalar);
// function for merging four TrivialMatrix instances.
// [first, second]
// [third, fourth]
struct TrivialMatrix* merge_matrices(const struct TrivialMatrix* first,
const struct TrivialMatrix* second,
                                     const struct TrivialMatrix* third,
const struct TrivialMatrix* fourth);
#endif  // P_PROJECT3_C_MATMUL_STRASSEN_H
```

- | └─matmul_trivial.h

```c
#ifndef P_PROJECT3_C_MATMUL_TRIVIAL_H
#define P_PROJECT3_C_MATMUL_TRIVIAL_H
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "util.h"
//common for struct
struct TrivialMatrix{
    float* data;
    size_t rowCount;
    size_t columnCount;
};
#define TrivialMatrixGetElement(matrix, i, j) ((matrix)->data[(matrix)->rowCount*(i)+(j)])
struct TrivialMatrix* _trivial_matrix(float* data, const size_t rowCount, const size_t columnCount);//完全从已有数据构造矩阵 //不建议调用
struct TrivialMatrix *create_zero_trivial_matrix(const size_t rowCount, const size_t columnCount) ;
struct TrivialMatrix* create_trivial_matrix(const size_t rowCount, const size_t columnCount);//会申请float*的空间,不过不会初始化为0
struct TrivialMatrix* create_from_existing_trivial_matrix(const struct TrivialMatrix* existing_matrix);//return a copy of the existing existing_matrix
void delete_trivial_matrix(struct TrivialMatrix* matrix);
struct StringBuilder* trivial_matrix_to_string(const struct TrivialMatrix* matrix);
struct TrivialMatrix* read_trivial_matrix_from_file(const char* file_name);//malloc a new one and return the address.
void write_trivial_matrix_to_file(const struct TrivialMatrix* matrix, const char* file_name);
//multiplication
struct TrivialMatrix* matmul_trivial(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB);
//addition and subtraction.
/**
 * Subtraction should not be treated as plus negative other, since negating a matrix takes time.
 * So I designed this function to do such θ(n^2) thing together.
 * @param matrixA
 * @param matrixB
 * @param alpha
 * @param beta
 * @return alpha*A + beta*B
 */
struct TrivialMatrix* matrix_element_by_element_linear_combinations(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB,
                                                                     float alpha, float beta);
struct TrivialMatrix* trivial_matrix_addition(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB);
struct TrivialMatrix* trivial_matrix_subtraction(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB);
struct TrivialMatrix* trivial_matrix_scalar_multiplication(const struct TrivialMatrix* matrixA, float scalar);
#endif //P_PROJECT3_C_MATRIX_TRIVIAL_H
```

- │└─util.h

```
#ifndef P_PROJECT3_C_UTIL_H
#define P_PROJECT3_C_UTIL_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//功能1：提供Deprecated注解
#if defined __GNUC__
#define __Matmul_Deprecated
#define Matmul_Deprecated__ __attribute__ ((deprecated))
#elif defined(__MSVC__)
#define __Matmul_Deprecated __declspec(deprecated)
#define Matmul_Deprecated__
#else
#define __Matmul_Deprecated
#define Matmul_Deprecated__
#endif

//功能2：提供时间处理，并保证1.多线程下准确计时。2.在毫秒级别准确。3.跨平台。
#ifdef WIN32
// windows
#include <Windows.h>
#include <stdint.h>
#pragma comment(lib, "winmm.lib ")
DWORD timer_start_time;  // private
DWORD timer_duration;    // public
#define TimerStart (timer_start_time = timeGetTime())
#define TimerEnd (timer_duration = timeGetTime() - timer_start_time)
#define TimerPrintln(msg) printf("%s :%lums\n", msg, timer_duration)
#define TimerEndPrintln(msg) TimerEnd; TimerPrintln(msg)
#define TimerEndPrintlnStart(msg) TimerEndPrintln(msg); TimerStart
#elif __linux__
#include <sys/time.h>
struct timeval timer_start_time;  // private
struct timeval timer_end_time;    // private
double timer_duration;              // public
#define TimerStart gettimeofday(&timer_start_time, NULL);
#define TimerEnd                            \
    gettimeofday(&timer_end_time, NULL); \
    timer_duration = 1000 * (timer_end_time.tv_sec -
timer_start_time.tv_sec) + 0.001* (timer_end_time.tv_usec -
timer_start_time.tv_usec);
#endif
//功能3：异常退出并在stderr输出错误信息。
void abort_with_message(const char* message);
//功能4：文件处理。
size_t file_len(FILE* file);

//功能4：StringBuilder。高效拼接char*
struct StringBuilder{
    char**stringArray;
    size_t arrayLength;
    //private
    size_t k;
    size_t strSize;
```

```
};
struct StringBuilder* _StringBuilder(char** stringArray, size_t
arrayLength);
struct StringBuilder* StringBuilder(size_t arrayLength);
struct StringBuilder* append_string(struct StringBuilder* stringBuilder,
const char* string);
void printf_string_builder(struct StringBuilder* stringBuilder);//may be
slow
char* stringBuilderToString(struct StringBuilder* stringBuilder);
#endif //P_PROJECT3_C_UTIL_H
```

├─sample

- main.c

```
//
// Created by 叶璨铭 on 2021/10/15.
//
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#ifdef WIN32
// windows
#include <Windows.h>
#define str_case_cmp _stricmp
#define str_cat strcat
#define str_cpy strcpy
#define _CRT_SECURE_NO_WARNINGS
#elif __linux__
// linux
#define str_case_cmp strcasecmp
#define str_cat strcat
#define str_cpy strcpy
#endif
#include "matrix_multiplication.h"
#include "util.h"
char* multiplication_mode_patterns[6] = { "trivial", "openblas", "strassen",
"simd", "openmp", "combined" };
void show_help() {
    // C语言不支持R"xx()xx"
    //但是两个字符串可以拼在一起，编译时处理
    puts("用法: ./matmul {input1.txt} {input2.txt} {output.txt} [-mode
trivial|openblas|strassen|simd|openmp|combined] [-help]");
    puts("功能: 从input1.txt和input2.txt读入两个矩阵matA和matB，计算矩阵乘法matA*matB，
并将结果放入output.txt中。");
    puts("选项:");
    puts("-mode:  指定矩阵乘法的运算策略。可以为
{TRIVIAL,OPENBLAS,STRASSEN,SIMD,CUDA,SPARSE}，不区分大小写");
    puts("   trivial已经包含了默认的内存连续性优化，但是不包括openmp的使用。基本上是上一个
Project的C版本移植。\n"
         "   openblas调用OpenBlas库来计算。是性能优化的目标。\n"
         "   strassen使用strassen算法来计算。\n"
         "   simd使用了avx2指令集来计算。暂不支持不是某些尺寸不是8的倍数的矩阵。\n"
         "   openmp使用了OpenMP并行接口来多线程计算。\n"
         "   combined尽可能结合以上模式的优点，提供一个接近于openblas的模式。\n"
         "   未来的计算可能还支持cuda和sparse模式。\n"
         "   cuda使用cuda指令来计算。sparse对于稀疏矩阵进行一定的优化。");
```

```
    puts("-help:  显示帮助信息。");
    puts("-t:  统计计算用时。单位为毫秒");
}
void error(const char* message, const char* argv0, const char* level) {
    struct StringBuilder* stringBuilder = StringBuilder(3);
    append_string(stringBuilder, argv0);
    append_string(stringBuilder, level);
    append_string(stringBuilder, message);
    fprintf_string_builder(stderr, stringBuilder);
}
void fatalError(const char* message, const char* argv0) {
    error(message, argv0, ": Fatal Error: ");
    puts("matrix multiplication terminated. ");
    show_help();
}
void optionError(const char* message, char* argv0) {
    error(message, argv0, ": Invalid arguments: ");
}
int main(int argc, char* argv[]) {
#ifdef WIN32
    SetConsoleOutputCP(CP_UTF8);
#endif
    enum multiplication_mode computationMode;
    int k = 0;
    char* in_out_filenames[3];  //连续的三个char*，那么这三个char*需要malloc，如果在原地是不行的
    bool timeBench = false;
    for(size_t i = 1; i < argc; i++) {
        char* arg = argv[i];
        if(arg[0] == '-') {
            if(strlen(arg) > 1)
                switch(arg[1]) {
                    case 'h':
                    case 'H': {
                        show_help();
                        return 0;
                    }
                    case 't':
                    case 'T': {
                        timeBench = true;
                        continue;
                    }
                    case 'm':
                    case 'M': {
                        arg = argv[++i];
                        computationMode = 0;
                        for(size_t j = 0; j < 6; j++) {
                            if(str_case_cmp(arg, multiplication_mode_patterns[j]) == 0)  //不加0，后果严重
                            {
                                computationMode = j;
                                break;
                            }
                        }
                        continue;
                    }
                    case '-': {
                        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;
            else {
                fatalError("Too many arguments!", argv[0]);
                return -1;
            }
        }
    }
    if(k < 3) {
        fatalError("Too few arguments!", argv[0]);
        return -1;
    }
    if(timeBench) {
        TimerStart;
    }
    file_matrix_multiplication(in_out_filenames[0], in_out_filenames[1],
in_out_filenames[2], computationMode);
    if(timeBench) {
        TimerEndPrintln(multiplication_mode_patterns[computationMode]);
    }
}
```

- matmul_benchmark.c

```
#include "matrix_multiplication.h"
#define SIZE_OF_PATH "2048"
#define PROJECT_DIR
"D:\\EnglishStandardPath\\Practice_File\\P_C_Cpp\\P_Project3_C\\"
#define pathA PROJECT_DIR"resources\\input_data\\mat-A-" SIZE_OF_PATH ".txt"
#define pathB PROJECT_DIR"resources\\input_data\\mat-B-" SIZE_OF_PATH ".txt"
#define out_path PROJECT_DIR"resources\\output_data\\mat-B-" SIZE_OF_PATH ".txt"
#define pathC PROJECT_DIR"resources\\output_data\\mat-C-" SIZE_OF_PATH ".txt"
```

```c
#define Five_Times(statement) statement statement statement statement statement
int main(int argc, char const *argv[])
{
    struct TrivialMatrix* a, *b, *c;
    TimerStart;
    Five_Times(a = read_trivial_matrix_from_file(pathA);)
    Five_Times(b = read_trivial_matrix_from_file(pathB);)
    TimerEndPrintlnStart("reading two matrices: ");

    c = matmul_openmp2(a, b);
    c = matmul_openmp2(a, b);
    TimerEndPrintlnStart("warming up: ");

    Five_Times(c = matmul_blas(a, b);)
    TimerEndPrintlnStart("matmul_blas: ");

    Five_Times(c = matmul_openmp2(a, b);)
    TimerEndPrintlnStart("matmul_openmp2: ");

    Five_Times(c = matmul_simd(a, b);)
    TimerEndPrintlnStart("matmul_simd: ");

    Five_Times(c = matmul_simd_openmp(a, b);)
    TimerEndPrintlnStart("matmul_simd_openmp2: ");

    Five_Times(c = matmul_strassen(a, b);)
    TimerEndPrintlnStart("matmul_strassen: ");

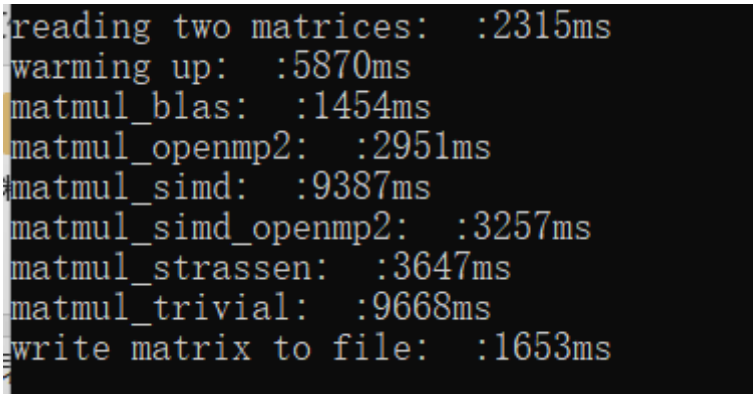
    Five_Times(c = matmul_trivial(a, b);)
    TimerEndPrintlnStart("matmul_trivial: ");

    Five_Times(write_trivial_matrix_to_file(c, pathC);)
    TimerEndPrintln("write matrix to file: ");
    return 0;
}
```

Part 3 - Result & Verification

3-1 探究不同优化方式在不同数据规模下的表现

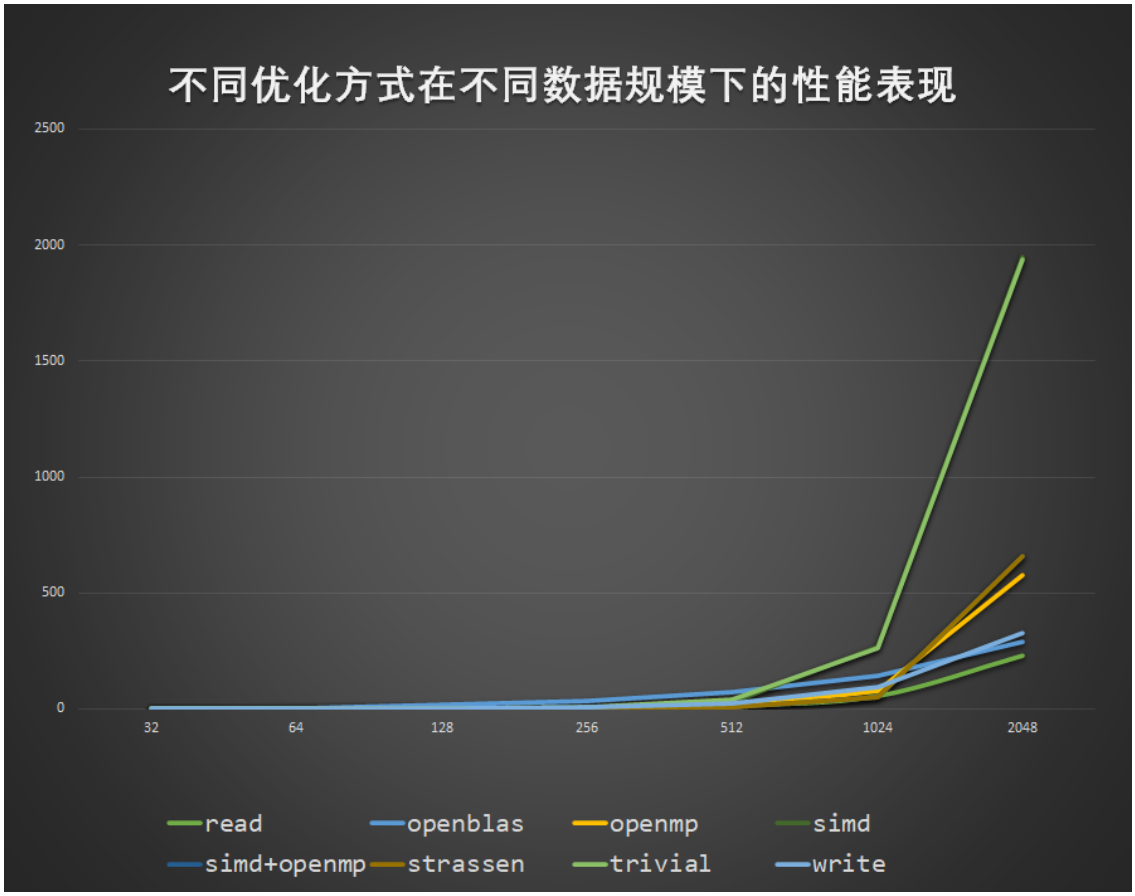
3-1-1 实验内容

- 修改matmul_benchmark.c，让size依次为32-2048
- 运行matmul_benchmark.exe。
- 

- 将显示的毫秒数据/5，填入表格中。

3-1-2 实验数据处理1 与实验结论1

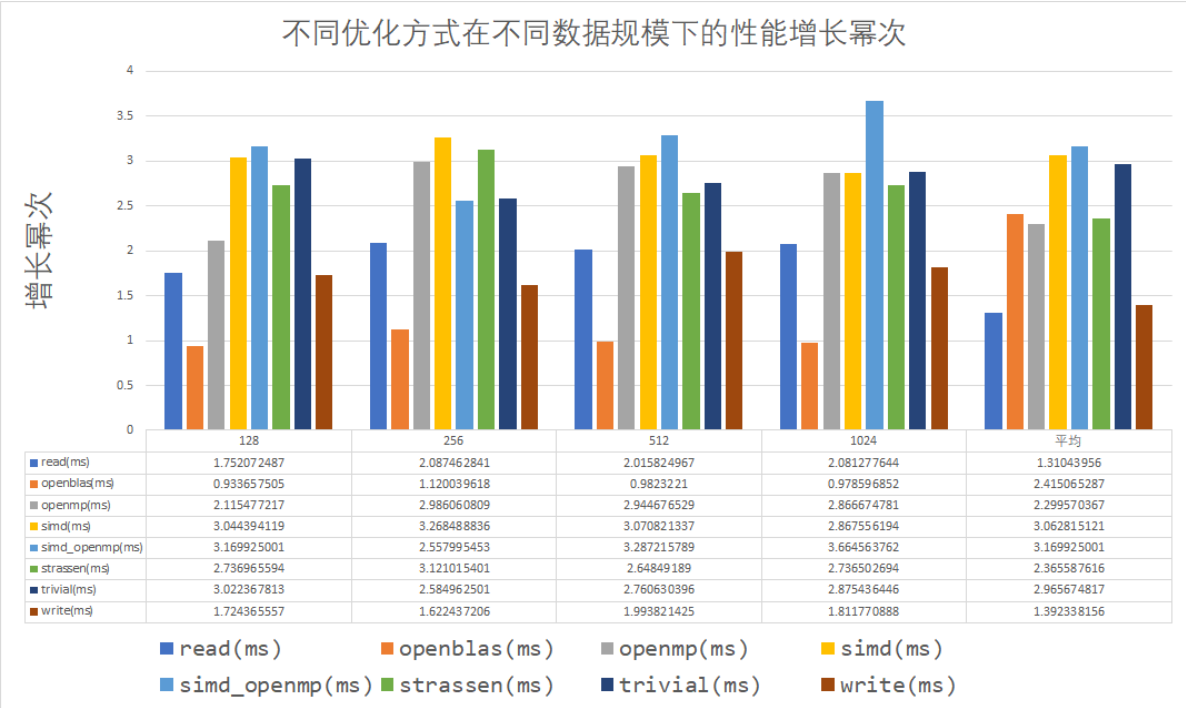
- 实验数据处理1：绘制折线图。



- 图像分析：
  - 在数据较小时，simd+openmp的效果最好，而openblas的表现并不是很好。
  - 当数据较小时，simd有效果，但是数据较大时，**simd与普通算法的曲线重合了，根本没有效果。**
  - 当数据超过1024时，平凡算法时间显著增长，**strassen和openmp均有较好的优化效果**，但是仍达不到openblas的水准。
- 实验结论：
  - **对外层循环进行的、内层循环无相互干扰的OpenMP使用对于矩阵乘法的优化很有效果，且优化倍数约为线程数（本次测试为16）。**
  - **对B矩阵每次取8列、同时计算8个C中元素的SIMD而言，该SIMD优化对于较大的矩阵没有效果。**
  - **带阈值为512的strassen算法可以在一定程度上提高大矩阵的运算速度。**
  - **openblas对于较小的矩阵进行乘法的效率不高，涉及这种矩阵的密集运算时不应频繁调用openblas，而应该尽可能地形成大矩阵的稀疏运算再调用openblas。**

3-1-3 实验数据处理2 与实验结论2

- 计算下一个规模下时间与这一次规模下时间的比，并取2的对数。



	128	256	512	1024	平均
read(ms)	1.752072487	2.087462841	2.015824967	2.081277644	1.31043956
openblas(ms)	0.933657505	1.120039618	0.9823221	0.978596852	2.415065287
openmp(ms)	2.115477217	2.986060809	2.944676529	2.866674781	2.299570367
simd(ms)	3.044394119	3.268488836	3.070821337	2.867556194	3.062815121
simd_openmp(ms)	3.169925001	2.557995453	3.287215789	3.664563762	3.169925001
strassen(ms)	2.736965594	3.121015401	2.64849189	2.736502694	2.365587616
trivial(ms)	3.022367813	2.584962501	2.760630396	2.875436446	2.965674817
write(ms)	1.724365557	1.622437206	1.993821425	1.811770888	1.392338156

- 实验结论：
  - simd、openmp的优化不能降低算法的复杂度。
  - 以512为阈值的strassen算法，在2048大小之下，能够不那么有效地降低算法复杂度
  - openblas的在2048的规模以下，可以看成是是O(n)的（平均值被前面512、256拉高了而已）

Part 4 - Difficulties & Solutions

4-1 如何使用CMake去管理一个C语言项目?

4-1-0 CMake的优势

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。[1] CMake对于我们本次写Project来说主要有以下好处：

- 一键编译运行或者debug，与IDE集成良好。
  - 无论是VsCode、CLion还是Visual Studio，都有CMake的插件或者直接支持。
  - cmake可以把gcc或者cl.exe的编译器选项，项目各个文件、文件夹之间的关系描述的一清二楚。
  - 即使没有用IDE，命令行中也是一到两行命令的事情。
- 跨平台编译。
  - 无论是Windows下的gcc、linux下的gcc还是mac下的gcc，
  - 无论是Windows下的gcc、clang还是msvc
  - CMakeLists.txt文件都不用进行修改（只需要扩展一个文件，而不是说需要准备几个不同版本的文件）。
- 与vcpkg包管理器的集成良好。
  - 见4-1-3

4-1-1 本次project的结构设计

- 所谓项目结构设计，需要将不同的模块分离。
  - 使得源码的位置是合理的，编程时关注点可以分离。
- 对于**矩阵乘法器**而言，它首先是一个可执行文件。
  - 这个可执行文件需要处理**命令行参数**，

    - 需要从**文件**中读取矩阵，写出矩阵，
    - 需要进行**矩阵乘法**
    - 需要进行**不同优化策略**下的矩阵乘法
    - 还需要对矩阵乘法进行**计时。**
- 可以看到，矩阵乘法器的较为复杂，应该分为不同的子模块。

4-1-2 如何用CmakeCMake基本语法实现project结构，并且成功编译?

- 在Cmake中，编译一个项目和项目的组织结构是分不开的。CMakeLists描述了项目的组织模式，也描述了编译的方式方法。
- 本次project中我采用了一种常用的CMake项目结构-sample、src、lib、bin、include结构。
- 首先，main.c是最后的可执行文件
    - 处理命令行参数，将命令行参数转为内存中的信息。
    - main.c调用matrix_multiplication库，把命令行参数的指令中提到的计算模式、文件路径告诉这个库，期待该库完成矩阵乘法任务和文件读写的任务。
    - ```
      add_executable(c_matmul main.c)
      target_link_libraries(c_matmul matrix_multiplication)
      ```
- 然后，**不同的计算模式对于矩阵的抽象方式可能是不同的**（比如稀疏矩阵需要使用ST表; 或者某些优化需要读入转置后的矩阵; 以及某些优化需要内存对齐），因此文件读入读出的任务亦分配到不同计算模式的实现当中。
    - matrix_multiplication库只是不同计算模式库的一个集成。
    - ```
      add_library(matrix_multiplication matrix_multiplication.c) #最后供main.c使用的是matrix_multiplication这个库
      target_link_libraries(matrix_multiplication #这个库包括了矩阵乘法的不同实现方式
              util
              matmul_trivial
              matmul_openblas
              matmul_strassen
              matmul_simd)
      ```
- 最后，每一个计算模式可以看做是一个“类”，包括了**结构体与结构体相关的方法。**
    - 这些计算模式都是子库。也需要使用cmake的add_library语法
    - 而util.c可以看做是一个静态方法库（我的意思是类似于Java中public方法都是static方法的类），不用写结构体，提供一些函数。

4-1-3 OpenBlas库如何导入?

- 在本次project中，openblas的安装对于大部分同学来说都是难点与痛点之一。
    - 其实，如果能够理解好lab课件，把openblas库inc、lib的位置搞清楚，其实openblas的手动安装并不难。
    - 不过，手动地编译openblas库并且处理很多细节，毕竟是一件很麻烦的事情。
- **Vcpkg**是微软开发的一款C/C++包管理器。[2]
    - 和CMake一样，Vcpkg也是开源、跨平台的。
    - 不只是Windows上能用，Linux上也完全能用。
    - 使用Vcpkg来安装库大大提高了效率。
- 下面以Windows下的Vcpkg和Cmake为例，记录一下如何安装OpenBlas并且在矩阵乘法项目中使用它。[2]
- 1.安装vcpkg

```
cd [欲安装位置]
git clone https://github.com/Microsoft/vcpkg.git
.\vcpkg\bootstrap-vcpkg.bat
```

- 2.用vcpkg安装OpenBlas

```
./vcpkg install openblas:x64-windows
./vcpkg integrate install
```

- 3.cmake命令行选项修改. 打开CLion（或者VsCode），在Cmake设置中增加命令行参数：

```
 -DCMAKE_TOOLCHAIN_FILE=[安装位置]/scripts/buildsystems/vcpkg.cmake
```

  如果使用Visual Studio作为IDE，则不需要进行这一步。

- 4.在CMakeLists.txt中，链接库到使用库的library或者executable.

```
add_library(matmul_openblas matmul_openblas.c)
find_package(OpenBLAS CONFIG REQUIRED)
target_link_libraries(matmul_openblas PRIVATE OpenBLAS::OpenBLAS)
```

- 5.在源码中 #include <cblas.h>

4-1-2 总CMakeLists的编写

- 1.语言标准限制
  - 本次project要求C语言，因此
  - ```
    set(CMAKE_C_STANDARD 11)
    ```

- 2.编译器选项，可以根据Debug、Release；GCC、MSVC来选择不同选项
  - ```
    if(MSVC)
        #utf-8支持
        add_compile_definitions("UNICODE")
        add_compile_definitions("_UNICODE")
        add_compile_options("/utf-8")
        #OpenMP 和 openmp中的SIMD 支持
        add_compile_options("/openmp:experimental")
        add_compile_options("/Qvec-report:2")
        if(CMAKE_BUILD_TYPE MATCHES Debug)
        else()
            add_compile_options("/O2")
        endif()
    elseif(CMAKE_COMPILER_IS_GNUC)
        #OpenMP支持
        add_compile_options("-fopenmp")
        if(CMAKE_BUILD_TYPE MATCHES Debug)
        else()
            add_compile_options("-Ofast")
        endif()
    endif()
    ```

- 3.项目设置

```
include_directories(
        ${PROJECT_SOURCE_DIR}/include/
)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
add_subdirectory(sample) #子CMakeLists
add_subdirectory(src) #子CMakeLists
```

4-2 如何设计矩阵结构体和相应的函数?

4-3 再论 程序性能效率理论 与 基准测试方法——优化与优化验证之基础

- 在上次project的3-2中，我讨论了google benchmark如何用于基准测试；
- 在上上次project中，我从算法的角度分析了程序效率与基准测试。
- 然而，
  - 本次project采用C语言，而非C++语言。Google benchmark基于c++11，而chrono库亦基于C++11。
  - 如果只从算法的角度分析，将难以从理论高度认识SIMD、OpenMP、访存优化等优化效果的原理。做优化出现各种神奇现象时无从下手。
- 因此
  - 我们需要一个C语言下的新的可靠的**基准测试方法**。
  - 参考计算机科学中**程序性能与效率的有关理论**，分为不同的角度看待程序的性能与效率。

4-3-1 程序性能效率理论

4-3-1-1 效率与性能之区别、提升效率的几个层次[3]

- 首先，高性能指的是一个程序尽可能快地执行特定的任务
  - 而高效率指的是程序运行时不做无用功。
  - 如果有些领域本质上不能快速执行，那么一个高效的程序可能有效率但是不快。
- 那么如何提高效率呢?
  - 提升效率分为语言层次和设计层次，
  - 其中，设计层次按照这本书的说法可以分为算法设计和其他的如缓存和对象池之类的设计层次。
- 这里我总结为:
  - C/C++语言层次（比如避免拷贝、循环展开、编译时计算等）
  - 数据结构与算法分析层次（比如Strassen算法）
  - 计算机组成原理设计层次
- 下面对不同的层次，重新审视Project1、2中认识不足的地方:

4-3-1-2 C/C++语言层次的高效[3]

- **在project2中测试矩阵乘法时，我有一个疑惑**:
  - 到底要不要开O2、O3优化去测试矩阵乘法的优化?
  - 它的优化会不会干扰我的优化，从而干扰实验的结论?
  - 反过来，有个同学还问过我：“我的愚蠢优化会不会干扰如此优秀的gcc编译器的优化，导致还不如躺平让它帮我优化？”

- 为了彻底回答这个问题，我们需要搞清楚
  - O2、O3优化属于提高效率的三大层次的哪一个层次?
  - 我的优化属于哪个层次?
- 经过初步调查[4]
  - O2\O3选项的优化**大部分是语言级别的优化**，但是不是说直接改你源码，更多的是从编译之后寄存器分配的效率等角度去考虑的。[5]
  - Ofast涉及破坏语言标准的优化。
  - 仔细调查之后，**O2、O3选项虽然涉及计算机组成原理的优化层次，但是没有或者较少涉及到本次project的焦点：向量化（simd）、并行化（openmp）。**
  - 更加不可能有"发现这段代码是经典的矩阵乘法代码，编译器经过智能查库得知正确的矩阵乘法要这样写，改你代码"这种妄想。
- 而在《C++高级编程》一书中，作者认为我们要"谨慎使用语言级优化"，因为
  - "这些优化远不如整体设计和程序选择的算法重要。"，可能浪费时间在语言细节。
  - "一些语言层次的技巧可以通过好的优化编译器自动进行"
- 当然，作为正在学习C/C++语言的学生，**我们也必须学会语言层次的优化**，写出一段较为大气的代码，而不是指望编译器完成所有的语言层次的优化。
- 因此，我的结论是
  - **本次project中，可以认为对矩阵乘法的几种优化思路不会与编译器的优化互相干扰。**
    - 因此，在进行基准测试时，可以开启编译器优化选项。
  - 开启与不开启编译器优化选项后程序运行时间的差别，可以作为一个程序员在语言层次优化做得好不好的依据之一。

4-3-1-3 数据结构与算法分析层次

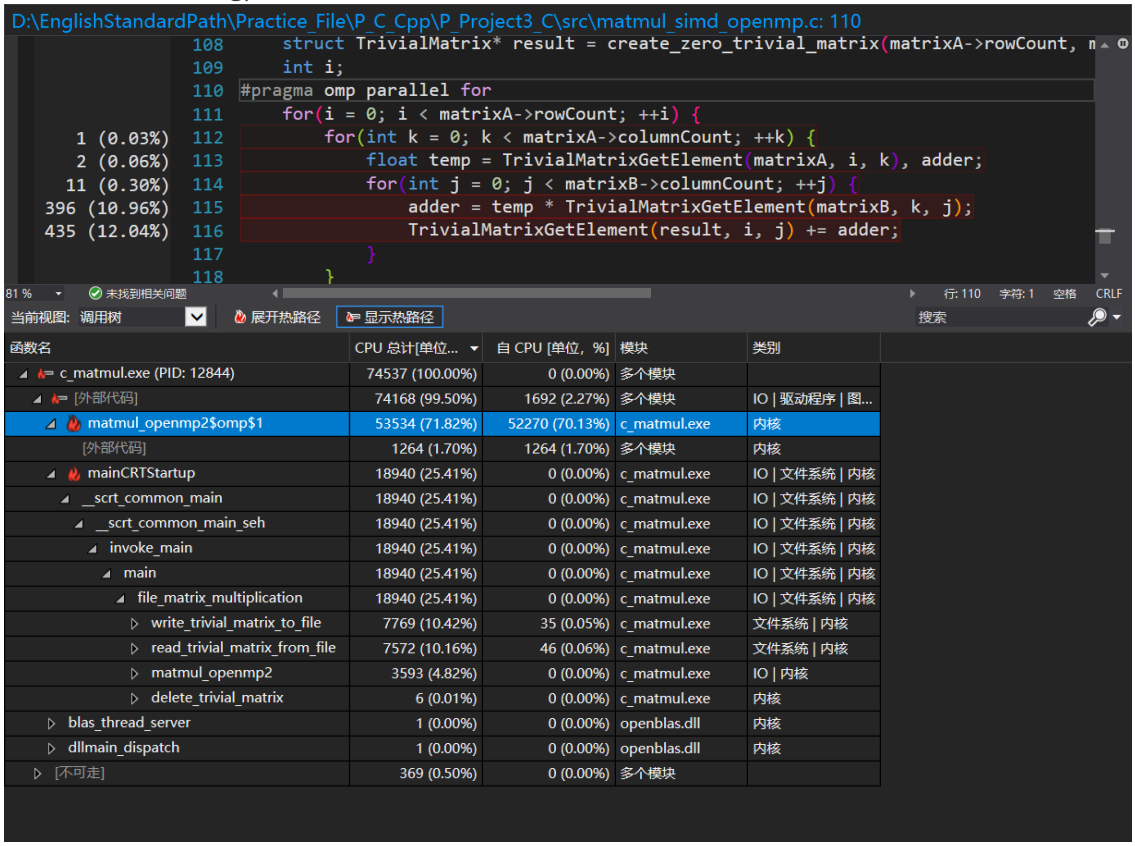
- 在project2中，我的报告中宣称，矩阵乘法的算法复杂度是$O(n^3)$, 但是最近学习了离散数学之后，产生一个疑问。
- 老师强调，要说明一个算法的复杂度，必须首先把**"input size"**搞清楚。[6]
  - input size 是 "minimum number of bits {0,1} neeed to encode the input of the problem."
  - 老师举的例子是：**对于自然数n，判断n是否为合数**
  - **T(n) = O(n).** 但是因为size(n)=logn
  - 所以$T(size(n)) = 2^{size(n)}$
  - T这个函数对于size x而言，**$T(x) = O(2^x)$**
  - 结果截然不同。
- 对于矩阵乘法，我分析也有这样的问题。
- 假设所分析的是$A_{mxk} \cdot B_{kxn}$
  - 对于T(n)， 不妨n是m，k，n中最大的，当然是$O(n^3)$
  - 矩阵的每一个元素的数据类型是float类型
    - float是4个字节，32个bit
    - 所以编码长度是32，与n无关
    - $O(n^3)$依然成立。
  - 然而我们需要质疑，size(n)这个函数，可以不以n为自变量吗?
- 答案是，完全可以。
  - **这次project中，于老师建议我们使用一维数组而非二维数组来进行矩阵表示。**
  - **考虑一维数组的长度而非二维数组的长度**
  - 那么32x32的矩阵的input size为 长度1024*元素编码量32
  - 对于一位数组的长度变量l

    - 我们可以认为矩阵加法的时间T(I) = O(I)， 矩阵乘法的时间不妨设为方阵，T(I) = O($I^{1.5}$)
    - 这依然符合定义。
- 那么，有的同学就说，“我都不用优化，只要更改变量，就可以把时间复杂度降低，太简单了。”实际上，
    - 我认为，变量的选择实际上与我们对于倍率的关注点是有关的
    - 比如，这次project中，我们考察矩阵从32x32到2048x2048的性能变化，倍率的增长是32增长到64、128...2048，
    - 而不是32x32增长到32x64再到32x128....2048x2048
    - 因此，我们依然以n为标准，而非I为标准。

4-3-1-4 计算机组成原理设计层次

将在4-4 矩阵乘法的优化 详细探讨。

4-3-2 C语言基准测试方法

- 剖析（profile）
- msvc 性能分析器与gprof
- 

4-4 矩阵乘法的优化

4-4-1 深入OpenMP——从基础语法、降速几十百倍（没有夸张）到真正的加速策略

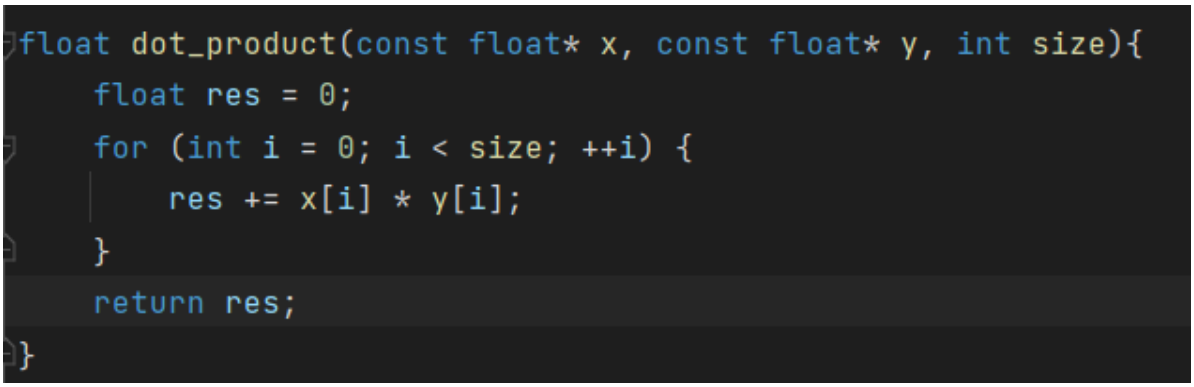
4-4-1-1 基础语法[7]

- OpenMP分为指令和函数两个部分可供使用
- #pragma omp for 可以让 for并行执行。

- ```
  //函数：线程数
  omp_get_num_threads() //易错函数，不能在并行体外调用。
  omp_set_num_threads(4)
  //当前线程
  omp_get_thread_num()
  ```

4-4-1-2 降速几十百倍及其原因

考虑于老师在学堂在线中提到的例子，尝试使用OpenMP计算向量点乘。[8]



如何优化呢？根据基础语法，我们可以加上指令#pragma omp parallel for并行化，同时#pragma omp critical保护数据。

```
//test_openmp.c
float dot_product3(const float* x, const float* y, int size){
    float res = 0, adder;
    int i;
    omp_set_num_threads(4);//默认是16个
#pragma omp parallel for
    for (i = 0; i < size; ++i) {
        adder = x[i] * y[i];
#pragma omp critical
        res += adder;
    }
    return res;
}
int main(){
    int test_size = 1000000;
    float* f2 = (float*)calloc(test_size,sizeof(float));
//    memset(f2,2,sizeof(float)*test_size);
    float* f1 = (float*)calloc(test_size,sizeof(float));
//    memset(f1,1,sizeof(float)*test_size); //WRONG：float数组不能用memset初始化值，C语言之大坑。
    for (int i = 0; i < test_size; ++i) {
        f1[i] =1;
    }
    for (int i = 0; i < test_size; ++i) {
        f2[i] =1;
    }
    dot_product(f1,f2,test_size);
    dot_product(f1,f2,test_size);
    dot_product(f1,f2,test_size);
    CLOCK_START;
    result = dot_product3(f1,f2,test_size);
    CLOCK_END("openmp");
    CLOCK_START;
```
```

```
    result = dot_product(f1,f2,test_size);
    CLOCK_END("trivial");
    free(f1);
    free(f2);
}
```

那么，理论上应该加速4倍对不对。然而

4-4-1-1-1 时钟不可靠

- ```
  openmp: 71ms, result: 1000000.000000
  trivial: 2ms, result: 1000000.000000
  ```
- OpenMP明显慢了30多倍。
- 是不是input size太小了？更改input size为100000000，结果更为惊人：
- ```
  openmp: 8448ms, result: 16777216.000000
  trivial: 82ms, result: 16777216.000000
  ```
- float溢出了，然后慢了100倍。
- 原来，时钟并不可靠，
  ```
  clock_t start;
  #define CLOCK_START (start = clock())
  #define CLOCK_END(msg) (printf(msg": %ld""ms, result: %f\n", (clock()-start), result))
  ```
- clock会计算CPU时间，而多线程会多次执行CPU。[9]

4-4-1-1-2 加锁减速

- 把时钟换为我在util.h中写好的安全时钟（见2-2 代码）
- 仍然有相似的结果。
- ```
  openmp: 104356ms, result: 1617.000000
  trivial: 27ms, result: 10000.000000          //几天前time
  openmp: 7294ms, result: 16777216.000000
  trivial: 3676ms, result: 16777216.000000    //数据来自几天前的代码，暂时无法复现，期间电脑环境、代码版本可能有些关键变化。结论可能有问题
  ```
- 原来，加锁导致速度的减慢

4-4-1-1-3 没有快

- 取消枷锁之后，答案错了，但是速度和trivial一样

  ```
  openmp: 2ms, result: 10000.000000
  trivial: 2ms, result: 10000.000000//数据来自几天前的代码，暂时无法复现,期间电脑环境、代码版本可能有些关键变化。结论可能有问题
  ```

- 经过查询，对于只有一层循环，且需要

4-4-1-3 真正的加速策略 需要逻辑推导
  ```

```
struct TrivialMatrix* matmul_openmp(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB) {
    int numberOfThreads; //读取线程数。
#pragma omp parallel
    {
        numberOfThreads = omp_get_num_threads();
    }
    if(matrixA->rowCount % numberOfThreads != 0) //行向量与列向量的长度需要是8的倍数
        return matmul_trivial(matrixA, matrixB);
    if(matrixA->columnCount != matrixB->rowCount) {
        abort_with_message( message: "Matrix multiplication of whose sizes do not match is not supported.");
    }
    struct TrivialMatrix* result = create_zero_trivial_matrix(matrixA->rowCount, matrixB->columnCount);
    //    for(int i = 0; i < matrixA->rowCount; ++i) {//做循环展开。展开的同时，发现不干扰，可以并行。于是可以omp parallel
    for(size_t i = 0; i < matrixA->rowCount; i += numberOfThreads) {
#pragma omp parallel firstprivate(i)
        {
            i +=
                omp_get_thread_num(); // i一次分配线程数，每个线程的i是私有的（从外层的i复制了一份）根据自己的线程编号来确定自己的i
            for(size_t k = 0; k < matrixA->columnCount; ++k) {
                float temp = TrivialMatrixGetElement(matrixA, i, k), adder;
                for(size_t j = 0; j < matrixB->columnCount; ++j) {
                    adder = temp * TrivialMatrixGetElement(matrixB, k, j);
                    TrivialMatrixGetElement(result, i, j) += adder;
                }
            }
        }
    }
    return result;
```

- 首先，并行发生在fori上
  - firstprivate表示把i这个变量复制给所有线程，每个线程对i的修改不影响其他线程的i，看到的i是自己的。
  - 网上有帖子说 pragma omp for 没有默认firstprivate(i)导致他出错，实际上经过查询，OpenMP2.0中pragma omp for
- 其次，代码可并行的原理来自于循环展开后的八段代码互相不会干扰。
  - 注意到**每个线程必定有不同的i**，因此，**每个线程必定有不同TrivialMatrixGetElement(result, i, j)**和不同的TrivialMatrixGetElement(matrixA, i, k)
  - 而 TrivialMatrixGetElement(matrixB, k, j)虽然可能相等，但是因为是读取而不是写出，所以是等价的。
  - 因此，这段代码是可并行的。
- 最后，并行可优化的原理是切分大任务，而不是切分小任务。
  - 参考[为什么用OpenMP并行计算会出现比单线程还要慢？该怎样正确使用OpenMP？ - 知乎 (zhihu.com)]

- 最后，经过仔细查询文档[7]，才确认这种写法与如下写法绝对是等价的，如果有问题，必定是其他问题出现了。

```
struct TrivialMatrix* matmul_openmp2(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB) {
    if(matrixA->columnCount != matrixB->rowCount) {
        abort_with_message( message: "Matrix multiplication of whose sizes do not match is not supported.");
    }
    struct TrivialMatrix* result = create_zero_trivial_matrix(matrixA->rowCount, matrixB->columnCount); //不会默认重制为0
    int i;
#pragma omp parallel for /*firstprivate(i)*/
    for(i = 0; i < matrixA->rowCount; ++i) {
        for(size_t k = 0; k < matrixA->columnCount; ++k) {
            float temp = TrivialMatrixGetElement(matrixA, i, k), adder;
            for(size_t j = 0; j < matrixB->columnCount; ++j) {
                // adder = temp * TrivialMatrixGetElement(matrixB, k, j);
                // TrivialMatrixGetElement(result, i, j) += adder;
                TrivialMatrixGetElement(result, i, j) += temp * TrivialMatrixGetElement(matrixB, k, j);
            }
        }
    }
    return result;
}
```

4-4-2 SIMD——从基础语法、降速策略到加速策略

- 基础语法：
  - **mm256可以与float[8]互转，然后在寄存器上，可以一次性就算一个**mm256, 也就是算了8个float
- 降速策略
  - 把内层循环当做是向量点乘，然后使用向量点乘的simd优化。结果：没有任何效果，变慢了10倍。受限于时间，未能深究原理。
  - ```
    __Matmul_Deprecated
    struct TrivialMatrix* matmul_simd_bad(const struct TrivialMatrix*
    matrixA, const struct TrivialMatrix* matrixB) {
        if(matrixA->columnCount % 8 != 0) //行向量与列向量的长度需要是8的倍数
            return matmul_trivial(matrixA, matrixB);
        if(matrixA->columnCount != matrixB->rowCount) {
            abort_with_message("Matrix multiplication of whose sizes do not
    match is not supported.");
        }
        //    size_t groupCount = matrixB->columnCount/8;
        struct TrivialMatrix* result = create_trivial_matrix(matrixA-
    >rowCount, matrixB->columnCount);
        //元素Cij是行向量Ai与列向量Bj的点乘
        float element[8]; //用float 8位数组取回 256bit寄存器的结果。
        __m256 a, b;      // Ai的8个元素和Bj的8个元素
        __m256 sum;
        for(size_t i = 0; i < matrixA->rowCount; ++i) {
            for(size_t j = 0; j < matrixB->columnCount; ++j) {
                sum = _mm256_setzero_ps();
                for(size_t k = 0; k < matrixA->columnCount; k += 8) {
                    //                result->data[i][j] += matrixA.data[i]
    [k]*matrixB.data[k][j];
                    a = _mm256_load_ps(&TrivialMatrixGetElement(matrixA, i,
    k));
                    b = _mm256_load_ps(&TrivialMatrixGetElement(matrixB, k,
    j));
                    sum = _mm256_add_ps(sum, _mm256_mul_ps(a, b));
                }
                _mm256_store_ps(element, sum);
                TrivialMatrixGetElement(result, i, j) =
                    element[0] + element[1] + element[2] + element[3] +
    element[4] + element[5] + element[6] + element[7];
            }
        }
        return result;
    }Matmul_Deprecated__
    ```
- 加速策略（至少平速，受限时间，未能查到更好的simd矩阵乘法的资料）
    ```

```
struct TrivialMatrix* matmul_simd(const struct TrivialMatrix* matrixA, const struct TrivialMatrix* matrixB) {
    if(matrixB->columnCount % 8 != 0) //A的行向量与B的列向量的长度需要是8的倍数
        return matmul_trivial(matrixA, matrixB);
    if (matrixA->columnCount != matrixB->rowCount) {
        abort_with_message( message: "Matrix multiplication of whose sizes do not match is not supported.");
    }
    __m256 b_line;
    __m256 c_results;
    float _a_element[8];
    __m256 a_element;
    struct TrivialMatrix* matrixC = create_trivial_matrix(matrixA->rowCount, matrixB->columnCount); //不会默认重制为0
    for (size_t i = 0; i < matrixA->rowCount; i++) {//固定A的一行
        for (size_t j = 0; j < matrixB->columnCount; j+=8) {//同时计算8个C的元素， 8个B的列也放一起
            c_results = _mm256_setzero_ps();
            for(size_t k=0; k <matrixB->rowCount; k++ ) {
                _a_element[7] = _a_element[6] = _a_element[5] = _a_element[4] = _a_element[3]
                    = _a_element[2] = _a_element[1] = _a_element[0] = TrivialMatrixGetElement(matrixA, i, k);
                a_element = _mm256_load_ps(_a_element);
                b_line = _mm256_load_ps(&TrivialMatrixGetElement(matrixB, k, j));
                c_results = _mm256_add_ps(c_results, _mm256_mul_ps(a_element, b_line));
            }
            _mm256_store_ps(&TrivialMatrixGetElement(matrixC, i, j), c_results);
        }
    }
    return matrixC;
}
```

- 逻辑，simd能够加速的原则，可能是要一次算出8个C中的元素，而不只是一次能够访问8个A、B的元素。[10]
- 所以尝试让A的行固定下，B一次走8列，同时算出8个C的元素。

4-4-3 Strassen算法实现[11]

```
#define Strassen_Threshold SQUARE(512)
struct TrivialMatrix* matmul_strassen(const struct TrivialMatrix* matrixA, const
struct TrivialMatrix* matrixB) {
    return view_view_strassen(ToView(matrixA), ToView(matrixB));
}
struct TrivialMatrix* view_view_strassen(const struct MatrixView* matrixViewA,
const struct MatrixView* matrixViewB) {
    // 1.size computation and checking
    size_t M = matrixViewA->viewSize.rowIndex;
    size_t K = matrixViewA->viewSize.columnIndex;
    size_t N = matrixViewB->viewSize.columnIndex;  // M*K matrix times K*N
matrix
#ifdef _DEBUG
    if(K != matrixViewB->viewSize.rowIndex)
        abort_with_message("Matrix multiplication of whose sizes do not match is
not supported.\n Error has occurred in function "
                           "view_view_strassen.");
        // debug use:
stringBuilderToString(trivial_matrix_to_string(matrixViewA->data))
#endif
    // 2.base case checking
    if(M * N <= Strassen_Threshold)
        return view_matmul_openmp(matrixViewA, matrixViewB);
    // 3.submatrices creating
    size_t halfM = M / 2;
    size_t halfK = K / 2;
    size_t halfN = N / 2;  // left up inclusive, right down exclusive. //not
mathematical 1 startingCoordinate
    struct MatrixView* A11 =
        createViewFromView(matrixViewA, *createMatrixCoordinate(0, 0),
```

```
                        *createMatrixCoordinate(halfM, halfK));  //不用减一，
0,1,2行的rowCount是3而非2 //FIXED
    struct MatrixView* A12 =
        createViewFromView(matrixViewA, *createMatrixCoordinate(0, halfK),
*createMatrixCoordinate(halfM, K - halfK));
    struct MatrixView* A21 =
        createViewFromView(matrixViewA, *createMatrixCoordinate(halfM, 0),
*createMatrixCoordinate(M - halfM, halfK));
    struct MatrixView* A22 =
        createViewFromView(matrixViewA, *createMatrixCoordinate(halfM, halfK),
*createMatrixCoordinate(M - halfM, K - halfK));
    struct MatrixView* B11 =
        createViewFromView(matrixViewB, *createMatrixCoordinate(0, 0),
*createMatrixCoordinate(halfK, halfN));
    struct MatrixView* B12 =
        createViewFromView(matrixViewB, *createMatrixCoordinate(0, halfN),
*createMatrixCoordinate(halfK, N - halfN));
    struct MatrixView* B21 =
        createViewFromView(matrixViewB, *createMatrixCoordinate(halfK, 0),
*createMatrixCoordinate(K - halfK, halfN));
    struct MatrixView* B22 =
        createViewFromView(matrixViewB, *createMatrixCoordinate(halfK, halfN),
*createMatrixCoordinate(K - halfK, N - halfN));
    // S1-S10 computing.
    struct TrivialMatrix* S[11] = { NULL,
                                    view_subtraction(B12, B22),
                                    view_addition(A11, A12),
                                    view_addition(A21, A22),
                                    view_subtraction(B21, B11),
                                    view_addition(A11, A22),
                                    view_addition(B11, B22),
                                    view_subtraction(A12, A22),
                                    view_addition(B21, B22),
                                    view_subtraction(A11, A21),
                                    view_addition(B11, B12) };
    // P1-P7 computing.
    struct TrivialMatrix* P[8] = { NULL,
                                   view_view_strassen(A11, ToView(S[1])),  //不要
交换
                                   view_view_strassen(ToView(S[2]), B22),
                                   view_view_strassen(ToView(S[3]), B11),
                                   view_view_strassen(A22, ToView(S[4])),
                                   matmul_strassen(S[5], S[6]),
                                   matmul_strassen(S[7], S[8]),
                                   matmul_strassen(S[9], S[10]) };
    struct TrivialMatrix* C11 =

 trivial_matrix_addition(trivial_matrix_subtraction(trivial_matrix_addition(P[5]
, P[4]), P[2]), P[6]);
    struct TrivialMatrix* C12 = trivial_matrix_addition(P[1], P[2]);
    struct TrivialMatrix* C21 = trivial_matrix_addition(P[3], P[4]);
    struct TrivialMatrix* C22 =

 trivial_matrix_subtraction(trivial_matrix_subtraction(trivial_matrix_addition(P
[5], P[1]), P[3]), P[7]);
    return merge_matrices(C11, C12, C21, C22);
}
```

- 使用了view来避免拷贝。
- 用阈值来决定选择。
- 使用stringBuilder来调试正确性。

参考文献

1. CMake Tutorial — CMake 3.22.0-rc1 Documentation
2. vcpkg - Open source C/C++ dependency manager from Microsoft
3. 格莱戈尔, M.), 凯乐普, & S.J.). (2012). C++高级编程(第2版). 清华大学出版社.
4. c++ - Is optimisation level -O3 dangerous in g++? - Stack Overflow
5. gcc -O0 -O1 -O2 -O3 四级优化选项及每级分别做什么优化_Aikenlan的博客-CSDN博客
6. QiWang, Lecture6.pptx
7. 2. OpenMP指令 | Microsoft Docs
8. C/C++：从基础语法到优化策略 - 南方科技大学 - 学堂在线 (xuetangx.com)
9. C 库函数 – clock() | 菜鸟教程 (runoob.com)
10. (24条消息) 数值计算优化方法C/C++(四)——矩阵乘法优化示例(访存优化和SIMD的使用)*artorias123*的博客-*CSDN*博客c++ 矩阵乘法优化
11. 详解矩阵乘法中的Strassen算法 - 知乎 (zhihu.com)