CS205 · C/C++ Programming

Matrix multiplication

何泽安 (He Zean)
12011323
Sep. 26, 2021

Contents

1 需求分析

本程序通过命令行参数获取两个存有需要相乘的矩阵的 txt 文件，进行矩阵乘法后需要将结果以类似于输入文件的格式保存到 txt 文件中。考虑到课程定位及进度，我认为本项目注重的是矩阵乘法后结果每一位的精度问题，而暂不需要考虑溢出，同时，本程序的定位是不需要处理数 GB 的特大矩阵——其难以加载到内存中，需要用特殊的方式直接对文件进行读写且较为复杂（类似的问题在《编程珠玑》中有所讨论），故可以选择直接将输入文件的内容读取到内存中。要求中提到“分别为 double/float 实现矩阵乘法”，但现阶段我们并没有学到处理这两种基本数据类型的特殊优化方式，即两种乘法除矩阵内存储数据类型不同，其余均一致，故我使用模版类来方便切换数据类型。本程序还具有以下 features。

1.1 文件输入输出

除竞赛中可能较常用的 FILE* 等写法，使用 *fstream* 进行文件读写更为常用且规范。但在写本程序时，注意到其并不会对指定的输入路径抛出显式异常，而仅以 *good()* 等方法标识出来。因此，我们首先需要一个函数来判断是否成功加载了两个矩阵 txt 文件（此项目偏重于计算，而非特别完备的异常处理机制，故省略检查文件内是否均为数字）。此外，输出时若目标路径下无指定文件，则创建之；否则将其内容覆盖写入。

1.2 矩阵可乘性判断

虽然给出的测试文件均为方阵，且矩阵规模出现在文件名中，此时一种投机取巧的方法就是直接读取文件名中的数字并开辟对应大小的数组，但鲁棒的程序必须假设输入的名字中包含的大小信息是错误的，需要自己判断（实际上，这也使得我们的程序不再受输入文件名的规范性影响）。同时，根据矩阵运算规则，在判断完两个矩阵的规模后，若不符合可以直接退出程序。

1.3 答案正确性判定 *

项目要求比较 float 和 double 型矩阵的运算精度，为此我写了一个简单的脚本，使用 NumPy 计算“标准答案”[1]与本程序结果进行分析。

1.4 计时器

修饰器是 Python 中的一个极为好用的语法糖，为了以类似的优雅方式实现此功能，在参考了 StackOverflow[2]的思想后，通过定义类的构造与析构方法，并定义宏，实现了比函数首尾手动计算时间差优雅的计时方法。

1.5 随机矩阵生成

为了对比矩阵乘法的不同实现的性能差异，现有的三组样例不足获取足够对比数据，且在计时上可能存在较大误差，因此需要使用自定的矩阵更全面的测试性能与程序鲁棒性。

[1]除非特别实现了高精度的数据结构，计算机进行小数运算有不可避免的误差。NumPy 默认使用的是 64 位的浮点数（相当于 double）类型来存储数据，故其计算结果可用于比较本程序算出结果是否有错误（较大偏差），也用于分析 float 较 double 的误差。

[2]How to implement decorators in C and C++ (stackoverflow.com/questions/4667293/how-to-implement-decorators-in-c-and-c)

2 代码实现

本项目路径下包含以下文件：

① util.hpp util.cpp 用于检测输入数据 ② Timer.hpp Timer.cpp 优雅的计时器实现

③ Mat.hpp Mat.tpp 定义矩阵模版类，便于切换 float/double

④ check.py NumPy 实现答案检验 ⑤ ThreadPool.hpp 线程池的实现[3]（未放入报告）

⑥ main.cpp, CMakeLists.txt 及测试文件

关于各关键函数的说明请阅读第 2.3 节。

2.1 Util

```hpp
// util.hpp

#pragma once

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

void check_input(std::ifstream &, std::ifstream &);
size_t file_rows(std::ifstream &);
size_t file_cols(std::ifstream &);
```

```cpp
// util.cpp

#include "util.hpp"

void check_input(std::ifstream &file1, std::ifstream &file2) {
    if (!file1.is_open() || !file2.is_open()) {
        if (!file1.good())
            std::cerr << "Error on "
                      << "Mat 1 (argv[1]) : "
                      << strerror(errno) << std::endl;
        if (!file2.good())
            std::cerr << "Error on "
                      << "Mat 2 (argv[2]) : "
                      << strerror(errno) << std::endl;
        throw std::invalid_argument("Invalid input file name");
    }
}

size_t file_rows(std::ifstream &f) {
    size_t row_cnt = 0;
    std::string tmp;
```

[3] A simple C++11 Thread Pool implementation (github.com/progschj/ThreadPool)

```
    while (getline(f, tmp, '\n')) row_cnt++;
    f.clear();
    f.seekg(0L, std::ios_base::beg);
    return row_cnt;
}

size_t file_cols(std::ifstream &f) {
    size_t col_cnt = 0;
    std::string line, tmp;
    std::getline(f, line, '\n');
    std::stringstream ssl(line);
    while (ssl >> tmp) col_cnt++;
    f.clear();
    f.seekg(0L, std::ios_base::beg);
    return col_cnt;
}
```

check_input 在主函数尝试打开 ifstream 后检查两个输入流的状态，如果有错误，通过 *strerror(errno)* 提示用户[4]，并通过抛出异常供主函数捕获的方式退出程序。

file_rows 和 *file_cols* 通过快速移动光标的方式检查文件的行数，及每行的元素数（空格分隔），在做了输入一定是矩阵（且均为数字）的假设后，不必逐行检查。

2.2 Timer

```
// Timer.hpp

#pragma once

#include <iostream>
#include<iomanip>
#include <chrono>
#include <unistd.h>

#define TIMER Timer stopwatch;

using namespace std;

class Timer {
private:
    chrono::time_point<chrono::system_clock> start_;
    constexpr static auto unit_ =
            static_cast<double>(chrono::microseconds::period::num)
            / chrono::microseconds::period::den;

public:
    Timer();
    ~Timer();
```

[4]How to get error message when ifstream open fails (stackoverflow.com/questions/17337602/how-to-get-error-message-when-ifstream-open-fails)

```
};
```

```cpp
// Timer.cpp

#include "Timer.hpp"

Timer::Timer() {
    this->start_ = chrono::system_clock::now();
}

Timer::~Timer() {
    auto end = chrono::system_clock::now();
    cout << "Time spent: " << setiosflags(ios::fixed) << setprecision(8)
         << static_cast<double>((end - this->start_).count()) * unit_
         << " sec" << endl;
}
```

这里利用了构造函数和析构函数管理时间变量算出 Timer 实例从生成到销毁的时间间隔并打印出来，也利用了函数中声明的变量（Timer 实例）存放在栈内存中，会在函数 return 时释放（调用析构函数）。此外，利用宏定义方便调用。

2.3 Mat

```cpp
// Mat.hpp

#pragma once

#include <cblas.h>
#include <immintrin.h>
#include <iostream>
#include <fstream>
#include <random>
#include <sstream>
#include <string>
#include <thread>

#include "Timer.hpp"
#include "ThreadPool.hpp"

#define THREADS thread::hardware_concurrency()

template<typename T>
class Mat {
private:
    size_t col_{};
    size_t row_{};
    T **data_;

```

```
    void show_ln(size_t);
    static Mat<T> strassen(const Mat<T> &, const Mat<T> &);

public:
    [[nodiscard]] size_t getCol() const;
    [[nodiscard]] size_t getRow() const;
    T **getData() const;

    Mat(size_t, size_t);
    Mat(size_t, size_t, ifstream &);
    ~Mat();

    Mat<T> operator+(const Mat<T> &);
    Mat<T> operator-(const Mat<T> &);

    static Mat<T> sub_mat(const Mat<T> &, size_t, size_t, size_t, size_t);
    static Mat<T> merge(Mat<T> &, Mat<T> &, Mat<T> &, Mat<T> &);

    static Mat<T> dot_n3(const Mat<T> &, const Mat<T> &);
    static Mat<T> dot_strassen(const Mat<T> &, const Mat<T> &);
    static Mat<T> dot_change_ord(const Mat<T> &, const Mat<T> &);
    static Mat<T> _mm_dot_float(const Mat<T> &, const Mat<T> &);
    static Mat<T> dot_mul_threads(const Mat<T> &, const Mat<T> &);
    static Mat<T> dot_blas(const Mat<T> &, const Mat<T> &);

    static void rand(size_t, size_t, const string &);

    void show();
    void save(const string &);
};

#include "Mat.tpp"
```

```
// part of <Mat.tpp>

template<typename T>
size_t Mat<T>::getCol() const {
    return col_;
}

template<typename T>
size_t Mat<T>::getRow() const {
    return row_;
}

template<typename T>
T **Mat<T>::getData() const {
    return data_;
}
```

```cpp

template<typename T>
Mat<T>::Mat(size_t row, size_t col) {
    this->col_ = col;
    this->row_ = row;
    this->data_ = new T *[row];
    for (size_t i = 0; i < row; ++i) {
        this->data_[i] = new T[col];
        for (size_t j = 0; j < col; ++j) this->data_[i][j] = 0;
    }
}

template<typename T>
Mat<T>::Mat(size_t row, size_t col, ifstream &f) {
    TIMER  // it prompts the user how long IO cost: load a Mat from a txt
    this->col_ = col;
    this->row_ = row;
    this->data_ = new T *[row];
    for (size_t i = 0; i < row; ++i) {
        this->data_[i] = new T[col];
        string line;
        std::getline(f, line, '\n');
        std::stringstream ssl(line);
        int col_i = 0;
        while (ssl >> this->data_[i][col_i++]);
    }
    std::cout << "Initialize Mat (" << col << "*" << row << ") finished. ";
    // the timer info will be attached here
    f.clear();
    f.seekg(0L, std::ios_base::beg);
}

template<typename T>
Mat<T>::~Mat() {
    for (size_t i = 0; i < this->row_; ++i)
        delete[] this->data_[i];
    delete[] this->data_;
}

template<typename T>
Mat<T> Mat<T>::operator+(const Mat<T> &op) {
    if (this->col_ != op.getCol() || this->row_ != op.getRow())
        throw invalid_argument("size not same");
    Mat<T> sum(this->row_, this->col_);
    for (size_t i = 0; i < this->row_; ++i)
        for (size_t j = 0; j < this->col_; ++j)
            sum.getData()[i][j] = this->data_[i][j] + op.getData()[i][j];
    return sum;
}

```

```cpp
template<typename T>
Mat<T> Mat<T>::operator-(const Mat<T> &op) {
    if (this->col_ != op.getCol() || this->row_ != op.getRow())
        throw invalid_argument("size not same");
    Mat<T> sum(this->row_, this->col_);
    for (size_t i = 0; i < this->row_; ++i)
        for (size_t j = 0; j < this->col_; ++j)
            sum.getData()[i][j] = this->data_[i][j] - op.getData()[i][j];
    return sum;
}
```

以下均为 <Mat.tpp> 的代码片段：

2.3.1 朴素矩阵乘法

```cpp
template<typename T>
Mat<T> Mat<T>::dot_n3(const Mat<T> &m1, const Mat<T> &m2) {
    TIMER
    Mat<T> prod(m1.getCol(), m2.getRow());
    for (size_t i = 0; i < m1.getRow(); ++i) {
        for (size_t j = 0; j < m2.getCol(); ++j) {
            for (size_t k = 0; k < m1.getCol(); k++)
                prod.getData()[i][j] += (m1.getData()[i][k] * m2.getData()[k][j]);
        }
    }
    cout << ">>> Product calculated: by <dot_n3>. ";
    return prod;
}
```

使用三重 for 循环嵌套，易知其复杂度为 $O(n^3)$，为本程序实现的最慢算法。但也因其是最基本的算法，我们将其引入作为性能比较的基准。

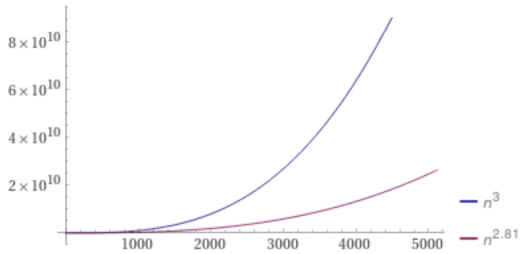
2.3.2 调换乘法顺序：访存优化

```cpp
template<typename T>
Mat<T> Mat<T>::dot_change_ord(const Mat<T> &m1, const Mat<T> &m2) {
    TIMER
    Mat<T> prod(m1.getRow(), m2.getCol());
    for (size_t i = 0; i < m1.getRow(); ++i) {
        for (size_t k = 0; k < m2.getRow(); ++k) {
            auto op = m1.getData()[i][k];
            for (size_t j = 0; j < m2.getCol(); ++j)
                prod.getData()[i][j] += op * m2.getData()[k][j];
        }
    }
    cout << ">>> Product calculated: by <dot_change_ord>. ";
    return prod;
}
```

矩阵乘法的性能瓶颈主要在于 IO，而非 CPU 算力。在本程序中，矩阵数据使用二维数组存储，除某一列外，乘法所需访问的元素大部分在内存中是不连续的（尤其是当其作为被乘矩阵时，需要按列访问），而每次计算中都尝试将当前访问内存地址的后一段内存这造成搬入缓存，矩阵朴素矩阵乘法造成了大量的缓存未命中。因此减少读取（访问内存地址的跳转）能提升乘法效率[5]。使用一维数组能更好的减少跳转次数，但考虑到可能输入测试样例过大，即当矩阵为 n×m 时，一维数组需要连续的长度为 $n \times m \times$ sizeof(T) 的空闲内存，在计算机内存紧张时，OS 难以找到这样的内存地址，可能会造成（暂时不清楚是什么）错误；而二维数组将数据分散开来，降低了这种风险，也将矩阵的表达更符合直观认知、规范，方便后期实现其他方法。

2.3.3 Strassen 算法

Strassen 算法和 Coppersmith–Winograd 算法是矩阵乘算法优化的两只最常用的算法。本程序实现了较易编写的 Strassen 算法。其不断的将矩阵分块（以中线分割，上下、左右分割为四块，因此限制了两个相乘的矩阵必须均为方阵，且大小为 2 的幂）并递归直至子矩阵足够小，将复杂度将到了 $O(n^{log7}) \approx O(n^{2.81})$。



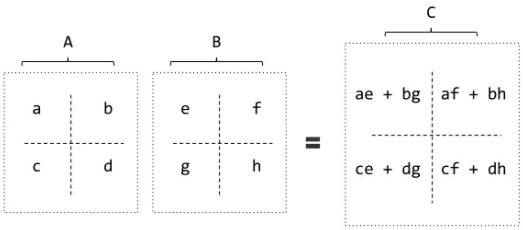

```cpp
template<typename T>
Mat<T> Mat<T>::dot_strassen(const Mat<T> &m1, const Mat<T> &m2) {
    TIMER
    if (m1.getRow() != m1.getCol() || m2.getRow() != m2.getCol()) {
        cout << "Strassen algo can only handle square matrices" << endl;
        return Mat<T>(0, 0);
    }
    size_t n = m1.getRow();
    if (n & (n - 1)) {  // a method to check whether n is 2^x
        cout << "Matrices' size must be a power of two" << endl;
        return Mat<T>(0, 0);
    }

    cout << ">>> Product calculated: by <dot_strassen>. ";
    return Mat<T>::strassen(m1, m2);
}

template<typename T>
Mat<T> Mat<T>::strassen(const Mat<T> &m1, const Mat<T> &m2) {
    size_t n = m1.getCol();

    if (n <= 256) {
        Mat<T> prod(m1.getRow(), m2.getCol());
```

[5]调整循环顺序加速矩阵乘法 (zhuanlan.zhihu.com/p/146250334)

```
        for (size_t i = 0; i < m1.getRow(); ++i) {
            for (size_t k = 0; k < m2.getRow(); ++k) {
                auto op = m1.getData()[i][k];
                for (size_t j = 0; j < m2.getCol(); ++j)
                    prod.getData()[i][j] += op * m2.getData()[k][j];
            }
        }
        return prod;
    }
    n /= 2;
    auto A11 = Mat<T>::sub_mat(m1, 0, 0, n, n);
    auto A12 = Mat<T>::sub_mat(m1, 0, n, n, n);
    auto A21 = Mat<T>::sub_mat(m1, n, 0, n, n);
    auto A22 = Mat<T>::sub_mat(m1, n, n, n, n);

    auto B11 = Mat<T>::sub_mat(m2, 0, 0, n, n);
    auto B12 = Mat<T>::sub_mat(m2, 0, n, n, n);
    auto B21 = Mat<T>::sub_mat(m2, n, 0, n, n);
    auto B22 = Mat<T>::sub_mat(m2, n, n, n, n);

    auto S1 = B12 - B22;
    auto S2 = A11 + A12;
    auto S3 = A21 + A22;
    auto S4 = B21 - B11;
    auto S5 = A11 + A22;
    auto S6 = B11 + B22;
    auto S7 = A12 - A22;
    auto S8 = B21 + B22;
    auto S9 = A11 - A21;
    auto S10 = B11 + B12;

    auto P1 = strassen(A11, S1);
    auto P2 = strassen(S2, B22);
    auto P3 = strassen(S3, B11);
    auto P4 = strassen(A22, S4);
    auto P5 = strassen(S5, S6);
    auto P6 = strassen(S7, S8);
    auto P7 = strassen(S9, S10);

    auto C11 = P5 + P4 - P2 + P6;
    auto C12 = P1 + P2;
    auto C21 = P3 + P4;
    auto C22 = P5 + P1 - P3 - P7;

    return Mat<T>::merge(C11, C12, C21, C22);
}

template<typename T>
Mat<T> Mat<T>::sub_mat(const Mat<T> &m, size_t rowst, size_t colst, size_t rowlen, size_t collen
    ) {
```

```
    Mat<T> sub(rowlen, collen);
    for (size_t i = 0; i < rowlen; ++i)
        for (size_t j = 0; j < collen; ++j)
            sub.data_[i][j] = m.getData()[i + rowst][j + colst];
    return sub;
}

template<typename T>
Mat<T> Mat<T>::merge(Mat<T> &m11, Mat<T> &m12, Mat<T> &m21, Mat<T> &m22) {
    Mat<T> mer(m11.getRow() * 2, m11.getCol() * 2);
    for (size_t i = 0; i < m11.getRow(); ++i) {
        for (size_t j = 0; j < m11.getCol(); ++j) {
            mer.getData()[i][j] = m11.getData()[i][j];
            mer.getData()[i][j + m11.getCol()] = m12.getData()[i][j];
            mer.getData()[i + m11.getRow()][j] = m21.getData()[i][j];
            mer.getData()[i + m11.getRow()][j + m11.getCol()] = m22.getData()[i][j];
        }
    }
    return mer;
}
```

作为分治算法，如果简单的任其递归到子矩阵为一个数字，可想而知大量的创建和销毁 Mat 对象反而会大幅拖慢算法效率（其中分配堆内存空间占用了大量计算时间，从而掩盖了 Strassen 算法的优势）。本算法复杂度下降不多，在矩阵较小时（该界限将在下面的性能测试中讨论），可以直接使用朴素矩阵乘，算法稍慢，但节省 IO 开销。



$$
\begin{aligned}
P1 &= A(F - H)\\
P2 &= (A + B)H\\
P3 &= (C + D)E\\
P4 &= D(G - E)\\
P5 &= (A + D)(E + H)\\
P6 &= (B - D)(G + H)\\
P7 &= (A - C)(E + F)
\end{aligned}
$$

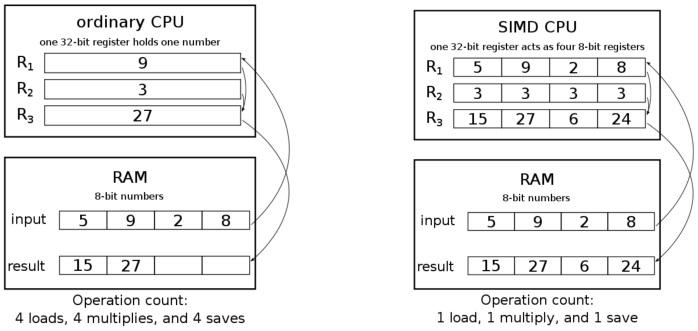
$$
AB = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}
$$

2.3.4 SIMD：硬件级优化

```
// Acknowledgement: Prof. Shiqi Yu, Speedup your program (2021 Spring)
float _mm_vec_dot(const float *v1, const float *v2, const size_t len) {
    float s[8] = {0};
    __m256 a, b;
    __m256 c = _mm256_setzero_ps();

    for (size_t i = 0; i < len; i += 8) {
        a = _mm256_load_ps(v1 + i);
        b = _mm256_load_ps(v2 + i);
        c = _mm256_add_ps(c, _mm256_mul_ps(a, b));
    }
    _mm256_store_ps(s, c);

```

```
14      auto sum = s[0] + s[1] + s[2] + s[3] + s[4] + s[5] + s[6] + s[7];
15      for (size_t i = len - 1; i >= len - len % 8; i--) sum += v1[i] * v2[i];
16      return sum;
17  }
18
19  template<>
20  Mat<float> Mat<float>::_mm_dot_float(const Mat<float> &m1,
21                                       const Mat<float> &m2) {
22      TIMER
23      Mat<float> prod(m1.getRow(), m2.getCol());
24      for (size_t i = 0; i < m1.getRow(); ++i)
25          for (size_t j = 0; j < m2.getCol(); ++j) {
26              auto v1 = static_cast<float *>(std::aligned_alloc(256,
27                                                                max(static_cast<int>(m1.getCol() *
                                                                    sizeof(float)), 256)));
28              auto v2 = static_cast<float *>(std::aligned_alloc(256,
29                                                                max(static_cast<int>(m2.getRow() *
                                                                    sizeof(float)), 256)));
30              memcpy(v1, m1.getData()[i], m1.getCol() * sizeof(float));
31              for (size_t k = 0; k < m2.getRow(); ++k) v2[k] = m2.getData()[k][j];
32              prod.data_[i][j] = _mm_vec_dot(v1, v2, m2.getRow());
33              delete[] v2;
34          }
35
36      cout << ">>> Product calculated: by <_mm_dot_float>. ";
37      return prod;
38  }
```

本机为 Intel x86 架构，CPU 设计有专门的寄存器可以一次保存 8 个 float[6]，使用该类寄存器可以使数据向量化，并使用一个控制器控制多个处理器，同时对一组数据中的每一个分别执行相同的操作从而实现空间上的并行性，即*单指令流多数据流 (Single Instruction Multiple Data)*[7]。对于 $\mathrm{Vec}(1,k)\cdot\mathrm{Vec}^T(1,k)$，这从原来的需要循环 k 次降低到了循环 $\frac{k}{8}$ 次。



这里有一点需要特别注意，即 26、27 行必须使用 aligned_alloc 使得内存对齐，否则（当矩阵规模较小，本机实验为 128×128）在将数据载入寄存器时会出现 EXC_BAD_ACCESS (code=EXC_I386_GPFLT) 的问题，经查阅[8]发现此问题是由从未对齐的内存地址中读入 SSE 寄存器导致的。

[6]Speedup your program | C/C++：从基础语法到优化策略 (xuetangx.com/course/sustc08091004451/7753997)
[7]SIMD (en.wikipedia.org/wiki/SIMD)
[8]What's the meaning of exception code "EXC_I386_GPFLT"? (stackoverflow.com/questions/19651788/whats-the-meaning-of-

2.3.5 BLAS

```cpp
template<typename T>
Mat<T> Mat<T>::dot_blas(const Mat<T> &m1, const Mat<T> &m2) {
    TIMER
    Mat<T> prod(m1.getRow(), m2.getCol());
    for (size_t i = 0; i < m1.getRow(); ++i)
        for (size_t j = 0; j < m2.getCol(); ++j) {
            auto v2 = new T[m2.getRow()];
            for (size_t k = 0; k < m2.getRow(); ++k)
                v2[k] = m2.getData()[k][j];
            prod.data_[i][j] = cblas_sdot(m2.getRow(), m1.getData()[i], 1, v2, 1);
            delete[] v2;
        }

    cout << ">>> Product calculated: by <dot_blas>. ";
    return prod;
}
```

本程序调用 OpenBLAS 库，其是一种优化了的 BLAS (Basic Linear Algebra Subprograms) 实现并会在编译时根据目标硬件进行优化，生成运行效率很高的程序。简单起见，我们仅使用 Blas 运算向量点积：其集多种优化方法（如向量化、内存对其、分块）于一体，性能优于 2.3.5 节只使用向量化（BLAS）。显然有更好的调用方法（sgemm 等），但需要修改 Mat 中的数据结构，工程量较大故暂未考虑。

2.3.6 多线程

```cpp
template<>
Mat<float> Mat<float>::dot_mul_threads(const Mat<float> &m1, const Mat<float> &m2) {
    TIMER
    Mat<float> prod(m1.getRow(), m2.getCol());
    // due to the time limit, I reused _mm_vec_dot, thus only support float
    /*
     * rather than using the manual way like the following:
     *      auto *threads = new thread[THREADS];
     *      for (int i = 0; i < THREADS; ++i) ...
     * I use ThreadPool, see: https://github.com/progschj/ThreadPool
     */
    ThreadPool pool(THREADS);
    for (size_t i = 0; i < m1.getRow(); i++)
        for (size_t j = 0; j < m2.getCol(); j++)
            pool.enqueue(calc_tar_pos, &prod, &m1, &m2, i, j);
        cout << ">>> Product calculated: by <dot_mul_threads>. ";
        return prod;
}

static void calc_tar_pos(Mat<float> *tar, const Mat<float> *m1,
                         const Mat<float> *m2, size_t row, size_t col) {
    auto v1 = static_cast<float *>(std::aligned_alloc(256,
```

exception-code-exc-i386-gpflt)

```
                                                    max(static_cast<int>(m1->getCol() * sizeof
                                                        (float)), 256)));
    auto v2 = static_cast<float *>(std::aligned_alloc(256,
                                                    max(static_cast<int>(m2->getRow() * sizeof
                                                        (float)), 256)));
    for (size_t k = 0; k < m2->getRow(); ++k) v2[k] = m2->getData()[k][col];
    tar->getData()[row][col] =
            _mm_vec_dot(v1, v2, m1->getCol());
    delete[] v2;
}
```

矩阵乘法并不会改变两个输入矩阵的数，也即计算结果矩阵的不同位置的元素的过程是互不影响的。上文指出 IO 为主要运算瓶颈，指的是相较于 CPU 乘法运算的时间，IO 读取两个被乘数的时间开销占比较大。但此时 CPU、RAM 占用率依然不高。多线程[9]的实现相当于将计算矩阵乘法细分为多个任务并均匀下发给多个计算单元，效率预计大幅提高。

2.4 main.cpp

```
#include "Mat.hpp"
#include "util.hpp"

#define MAT_TYPE double
//#define MAT_TYPE float

using namespace std;

int main(int argc, char **argv) {
    // rand matrix generator, can commit the following code to ignore the necessary args
//    Mat<float>::rand(2, 3, "mat-2x3.txt");
//    Mat<float>::rand(3, 6, "mat-3x6.txt");
//    Mat<float>::rand(8, 8, "mat-8x8.txt");

    if (argc != 4) {
        cerr << "3 arguments expected (input 1, input 2, output), got "
             << argc - 1 << endl;
        return -1;
    }

    // attempting to receive two input matrices
    ifstream matf1(argv[1]);
    ifstream matf2(argv[2]);
    try {
        check_input(matf1, matf2);
    } catch (const invalid_argument &) {
        return -1;
    }

```

[9]请注意，为了方便线程管理，我从 GitHub 上找到了 C++11 标准实现的线程池，而非手动创建线程，但可能正因如此，后面演示 O3 时使得编译器难以优化。

```
    // checking multiplication validity
    size_t mat1_c = file_cols(matf1), mat2_c = file_cols(matf2);
    size_t mat1_r = file_rows(matf1), mat2_r = file_rows(matf2);
    if (mat1_c != mat2_r) {
        cerr << "The input two matrices cannot be multiplied: "
             << "Mat 1 (" << mat1_r << "*" << mat1_c << ")\t"
             << "Mat 2 (" << mat2_r << "*" << mat2_c << ")";
        return -1;
    }

    auto m1 = Mat<MAT_TYPE>(mat1_r, mat1_c, matf1);
    auto m2 = Mat<MAT_TYPE>(mat2_r, mat2_c, matf2);
    cout << endl;

    for (int i = 0; i < 10; i++)
        Mat<MAT_TYPE>::dot_n3(m1, m2);
    cout << endl;
    auto prod1 = Mat<MAT_TYPE>::dot_n3(m1, m2);
    prod1.show();

    for (int i = 0; i < 10; i++)
        Mat<MAT_TYPE>::dot_change_ord(m1, m2);
    cout << endl;
    auto prod2 = Mat<MAT_TYPE>::dot_change_ord(m1, m2);
    prod2.show();
    prod2.save(argv[3]);

    for (int i = 0; i < 10; i++)
        Mat<MAT_TYPE>::dot_strassen(m1, m2);
    cout << endl;
    auto prod3 = Mat<MAT_TYPE>::dot_strassen(m1, m2);
    prod3.show();

    for (int i = 0; i < 10; i++)
        Mat<MAT_TYPE>::_mm_dot_float(m1, m2);
    cout << endl;
    auto prod4 = Mat<MAT_TYPE>::_mm_dot_float(m1, m2);
    prod4.show();

    for (int i = 0; i < 10; i++)
        Mat<MAT_TYPE>::dot_blas(m1, m2);
    cout << endl;
    auto prod5 = Mat<MAT_TYPE>::dot_blas(m1, m2);
    prod5.show();

    for (int i = 0; i < 10; i++)
        Mat<MAT_TYPE>::dot_mul_threads(m1, m2);
    cout << endl;
    auto prod6 = Mat<float>::dot_mul_threads(m1, m2);
    prod6.show();
```

```
80
81      prod6.save(argv[3]);
82
83      matf1.close();
84      matf2.close();
85      return 0;
86  }
```

① 头部的宏定义是为了方便切换 float/double 模式：只需在此注释其中之一，另一个即替换 main 中 Mat<T> 的数据类型，使模版类切换模式。但要注意 <_mm_dot_float>，<dot_mul_threads> 和 <dot_blas> 由于寄存器类型及调用第三方库的原因只支持 float 型矩阵的运算。

② 11 行附近代码为生成测试数据用，一般情况下保持被注释。

③ 包含基本的检查：输入参数个数，输入矩阵，大小等。

④ 六个重复的乘法运算在单次运行程序时只需保留一个，其余注释。用 for 循环预计算 10 次的目的主要是将数据载入缓冲以减小误差和多次采样以计算平均时间。

⑤ 81 行的 save 仅展示将数据存入文件，实际搭配上方唯一未被注释的函数作变量名的修改。

2.5 check.py

```python
import numpy as np

def txt2arr(path: str, delimiter: str = ' ') -> np.array:
    with open(path, 'r', encoding='utf-8') as f:
        mat = f.read()
    row_list = mat.splitlines()
    data_list = [[float(i)
                  for i in row.strip().split(delimiter)]
                 for row in row_list]
    return np.array(data_list)

def RMSE(x: np.array, y: np.array) -> float:
    return (np.sum((x - y)**2) / len(x)**2)**0.5

if __name__ == '__main__':
    mat1 = txt2arr('mat-A-2048.txt')
    mat2 = txt2arr('mat-B-2048.txt')
    my_prod = txt2arr('out2048.txt')
    # print(np.dot(mat1, mat2))  # to check if the answer is generally correct
    print('RMSE = ', RMSE(np.dot(mat1, mat2), my_prod))
```

在对比 double 与 float 的精度差异时，我想到以拟合标准差（RMSE）作为指标：

$$\mathrm{RMSE} = \sqrt{\mathrm{MSE}} = \sqrt{\frac{\mathrm{SSE}}{n}} = \sqrt{\frac{1}{n}\Sigma i}$$

其中，MSE（均方方差）是数理统计中均方误差是指参数估计值与参数值之差平方的期望值，即本程序运算结果与作为参考答案的 NumPy 运算结果的差异。RMSE 的值越小，说明本程序结果越接近 NumPy 结果。

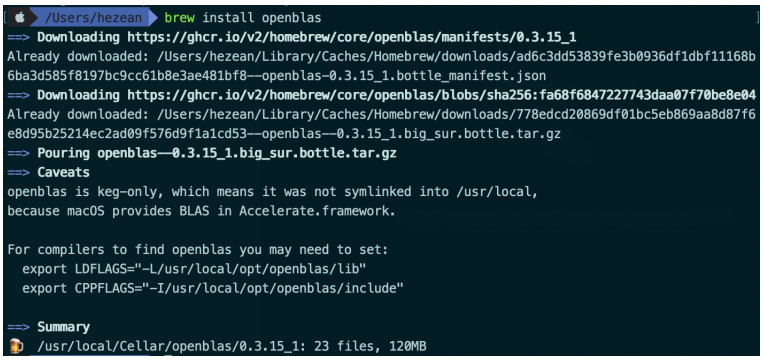
另一种一种方法是用 Java 的 BigDecimal 类计算，获得极为精确的的参考答案，但不是本项目的重点，后因耗时过多而放弃。

```java
import java.math.BigDecimal;

public class Mat {
    public static BigDecimal[][] mul(BigDecimal[][] matrix1, BigDecimal[][] matrix2) {
        int row1, row2, col1, col2;
        row1 = matrix1.length;
        row2 = matrix2.length;
        col1 = matrix1[0].length;
        col2 = matrix2[0].length;
        BigDecimal[][] mulMatrix = new BigDecimal[row1][col2];
        assert row2 == col1 : "Math Error";

        for (int i = 0; i < row1; i++)
            for (int j = 0; j < col1; j++)
                for (int k = 0; k < col1; k++)
                    mulMatrix[i][j] = mulMatrix[i][j].add(matrix1[i][k].multiply(matrix2[k][j]))
                        ;
        return mulMatrix;
    }
}
```

3 测试样例及分析

```
Platform: CMake 3.20.4 OpenBLAS 0.3.15_1
Hardware: MacBook Pro 13, 2020 (2 GHz Quad-Core Intel Core i5, 16 GB 3733 MHz LPDDR4X)
$ cd /path/to/project/ && cmake . && make && ./matmul <3 args>
```

运行此程序需要安装 OpenBLAS，并配置 CMake 使之 include 相关文件。并因为 AVX2 指令集，本程序的 _mm_dot_float 只能运行在 intel 芯片的计算机上。



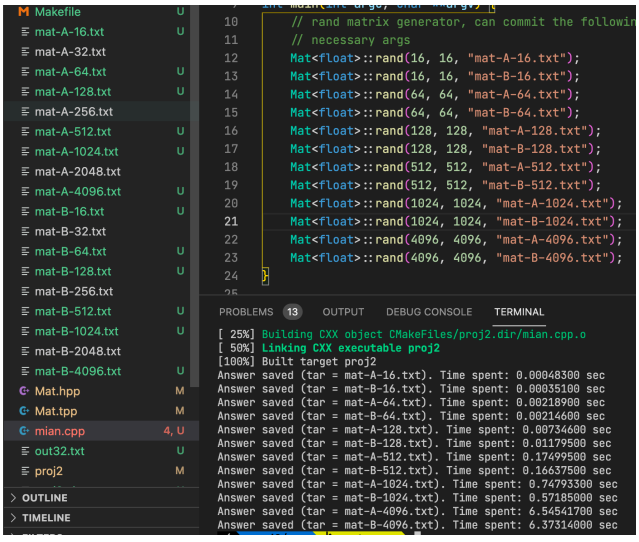
```
cmake_minimum_required(VERSION 3.20)
project(matmul)

set(CMAKE_CXX_STANDARD 20)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -mavx -I/usr/local/opt/openblas/include")
set(LDFLAGS "-L/usr/local/opt/openblas/lib")

set(USE_BLAS OpenBLAS)
set(BLAS_LIBRARY /usr/local/opt/openblas/lib/libblas.dylib)

add_executable(matmul main.cpp Timer.cpp Mat.tpp util.cpp ThreadPool.hpp)

target_link_libraries(matmul ${BLAS_LIBRARY})
```

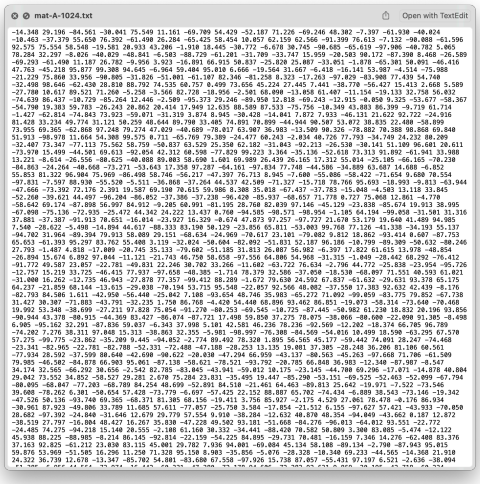
关于精度和性能的测试样本均为 Mat<T>::rand 生成，以 NumPy 运算结果作为参考。

3.1 程序附加功能

```
proj2/src  master ±+  ./proj2 mat-C-0.txt mat-B-32.txt out1.txt
Error on Mat 1 (argv[1]) : No such file or directory
proj2/src  master ±+  ./proj2 mat-A-32.txt mat-B-31.txt out1.txt
Error on Mat 2 (argv[2]) : No such file or directory
proj2/src  master ±+  ./proj2 mat-A-31.txt mat-B-31.txt out1.txt
Error on Mat 1 (argv[1]) : No such file or directory
Error on Mat 2 (argv[2]) : No such file or directory
```

检测输入文件是否有效





随机测试矩阵生成

```
proj2/src  master ±+  ./proj2 mat-A-32.txt mat-B-256.txt out1.txt
The input two matrices cannot be multiplied: Mat 1 (32*32)      Mat 2 (256*256)%
proj2/src  master ±+  ./proj2 mat-2x3.txt mat-3x6.txt out1.txt
Initialize Mat (3*2) finished. Time spent: 0.00004000 sec
Initialize Mat (6*3) finished. Time spent: 0.00002400 sec

>>> Product calculated: by <dot_change_ord>. Time spent: 0.00000100 sec
-720.63   -2011.52   7381.92   5309.08   -817.65   5481.14
5968.22   -7839.58   8506.15   7760.01   644.97    5813.77
Answer saved (tar = out1.txt). Time spent: 0.00032600 sec
```

矩阵可乘性判断

out2048.txt — Open with TextEdit

```
5252342.290 5053432.250 5087801.580 5006596.300 5185736.240 5085108.770 5122403.840 5107456.240 5100696.670 4986783.210
5179252.860 5227450.160 4954678.640 5186517.320 5016024.130 5086792.110 5034601.550 5048987.360 5125421.170 5183793.000
5173591.170 5168722.730 5225956.850 5148351.350 5181384.010 5165839.890 5047797.470 5037823.020 5138836.100 5143938.110
5078131.380 5084697.480 5003599.430 5066776.480 5071675.470 5138037.810 5077775.830 5066137.010 5059699.250 4996654.170
5198302.350 5144538.480 5142902.820 5093786.480 5093557.080 5193329.490 5131372.120 5011720.830 5164276.980 5069446.260
5073968.070 5066349.620 5123903.210 5031033.550 5121444.660 5151897.980 5074951.460 4963520.610 5064791.120 4970008.460
5141884.760 5099495.280 5080104.150 5106041.170 5088050.450 4939459.230 5158605.400 5093232.820 5164157.180 5149617.890
5090575.300 5229102.780 4928759.320 5108197.910 5002165.860 5279201.720 5191367.660 5205247.670 5097415.120 5055068.410
5093059.560 5139758.430 5209775.530 5147525.510 5078004.880 5010893.800 5121471.990 5037652.110 5054324.380 5143689.440
5075481.550 5152887.200 5166776.020 5070605.860 5118337.470 5046065.150 5139212.470 5186727.140 5093527.690 5170155.120
5121507.210 5160789.450 5198960.740 5003207.570 5152996.470 5215701.650 5252099.590 5064638.860 5163374.020 5187791.050
5167186.030 5229941.260 5045870.410 5046071.810 5044371.310 5151135.980 5115303.010 5199625.670 5182627.720 5209605.420
5177428.980 5035655.820 4969694.910 5123624.110 5039352.770 5258726.760 5052817.290 5105121.910 5075773.020 5216832.960
5102888.820 5130858.440 5177199.600 4991952.860 5190530.750 5107458.600 5031405.720 5044584.230 5154873.790 5137812.500
5101147.800 5065173.210 5055713.840 5236948.540 4996532.330 5131240.210 5173626.190 5170893.180 5089864.850 5087427.610
5157267.810 5128547.170 5111882.340 5124604.750 5217837.460 5092290.890 5268690.680 5075408.750 4994743.790 5104569.650
5097440.770 5155921.190 5097026.390 5091445.470 5013626.590 5165796.600 5153209.580 5120561.850 5001815.300 5114867.940
5194627.510 5171335.070 5126607.480 5244155.740 4971194.020 5105657.240 5120918.750 5065453.910 5117169.340 5158262.690
4985591.280 5121015.410 5266648.170 5008135.080 5137412.780 5153537.290 5068034.530 5169700.100 5083412.650 5125023.720
5121347.370 5100279.980 5155154.800 5205806.030 5077907.570 5095979.410 5120664.130 5097431.430 5129282.390 5050127.090
4991166.360 5168110.430 5244338.360 5225574.800 5058778.940 5071157.020 5217730.160 5100445.780 4988404.880 5018989.990
5139532.880 5157050.560 5158880.000 5131992.000 5125132.480 5087754.980 5170542.380 5086490.110 4949039.350 5045782.840
5200929.920 5091278.730 5215750.760 4952580.960 5192708.520 5069840.290 5051824.020 5118123.110 5135946.900 4930261.090
5052393.890 5144512.110 4862772.300 5074920.330 5081033.750 5145991.800 5059843.990 5118255.150 5024737.770 5029910.830
5090539.240 5128381.560 5241280.430 4935199.930 5173697.640 5061024.940 5192224.600 5111880.540 5054940.380 5057193.060
5161625.240 5173685.080 5042364.750 5020669.030 5107095.390 5119416.650 4946280.700 5205796.650 5006801.390 5091063.230
5194725.010 5194797.620 5211528.900 5223971.210 5072454.730 5116604.760 5192063.730 5195964.720 5075980.340 5119653.520
5137859.490 5167653.550 5093227.940 5121031.320 5212870.550 5181075.560 5091764.860 4969722.780 5078944.710 4973659.310
5171402.460 5039032.940 4988241.380 5115016.470 5169021.220 5023413.470 5166963.160 5121283.720 5157197.950 4916838.170
5135807.520 5133358.030 5138694.880 5103929.840 5096159.810 5184165.780 5064688.080 5306704.710 5157681.550 5144847.640
4941731.590 5103176.930 5117940.800 5165372.010 5039854.560 5154911.230 5117752.480 5147910.800 5086889.550 5018512.460
5142877.060 5074958.170 5224077.920 4974334.160 5074487.560 5080533.540 5133031.890 5004076.550 5024207.900 5049186.800
5073381.100 4952330.640 5131808.880 5089318.270 4910740.460 5021681.210 5147714.600 5054417.830 4961512.100 5096004.720
5066244.260 5047982.550 5071831.120 5204201.000 5076339.950 5119785.210 5108811.500 5191279.930 5201434.900 5024706.350
5105708.010 5032234.520 5164747.370 5259934.100 5247821.090 4970439.870 5095139.860 5174350.280 5129168.130 5105735.220
5063932.920 5152501.750 5059940.210 4970935.240 5145557.260 5082007.780 4932023.420 5164515.820 5084460.760 5140530.200
5208884.960 5075930.610 5074408.180 5078880.730 5231785.840 5167250.240 5126860.440 5112271.140 5034925.550 5160999.570
5050378.910 5163262.280 5151578.380 5182356.910 5078802.480 5014953.170 4992186.470 5295681.150 5082730.910 5069699.500
5054840.680 5105383.230 5139350.010 5187008.270 5132264.390 5062505.030 5100936.780 4869335.040 5180838.660 5164258.790
5032551.010 5113716.090 5099822.950 5076223.280 5154210.210 5063948.730 5273238.930 5103844.060 5143901.240 5123764.540
5157751.170 5187028.480 5156899.590 5147747.010 5145743.020 5169631.410 5083573.430 5117171.020 5158452.310 5087408.200
5070175.010 5070613.170 5173704.600 5247814.180 5113959.750 5033153.190 5109887.150 5134402.890 5157095.860 5149187.330
5074846.100 5136467.110 5032237.710 5066392.470 5076299.560 5054634.700 5103133.840 5186035.160 5144129.150 5159717.190
4966879.830 5094251.440 5193774.200 5166963.250 5046828.130 5166033.210 5246556.580 5142792.350 5168600.200 4963852.920
5018123.100 4947172.470 5174601.540 5109163.100 5089334.380 5072908.530 5187970.710 5124089.290 4985867.660 5011641.250
5167851.010 5163728.110 5094419.430 5048910.520 5234124.220 5086443.570 5012999.320 5199839.160 5105451.790 5102685.270
5171183.580 5158555.120 5126766.330 5071096.390 5164395.330 5101632.420 5170544.540 5214461.840 5070263.370 5061780.730
5141279.820 4986244.970 5002703.880 5130707.150 5195513.910 5082263.120 4973124.940 5175994.000 5113365.480 5086552.430
5112557.250 5228965.990 5083659.200 5074962.310 5021872.270 5075335.910 5012794.220 5125732.340 5059658.630 4985153.200
5197383.700 4998011.160 5142447.420 5279977.880 5046938.130 5084661.640 5260412.110 5101932.730 5153600.880 5065737.710
5124513.150 5025958.720 5000325.150 5069290.870 5008161.030 5048667.410 5046747.820 5289502.320 5000741.280 5192477.360
5093316.940 5055821.580 5025224.730 5005443.700 5046795.280 5119346.880 4972325.750 5135597.020 5260061.120 5090313.120
5159027.390 5093808.500 5045111.290 5135222.150 5090832.970 5293520.770 4966785.010 5100292.230 5051059.650 5221136.790
5033081.790 5143016.810 5149071.270 5028104.750 5009649.270 5058223.660 5102814.520 5107594.800 5005601.300 5148780.430
5160831.010 5067918.990 5192507.340 5151995.630 5049838.610 5178310.550 5242090.960 4911482.960 4971152.460 5226980.810
5165804.540 5094820.350 5069366.490 5078281.620 5063524.600 5078887.220 5076890.430 5122802.820 5088720.550 5031055.810
5018508.340 5018173.380 5071359.430 5123649.050 4986939.960 5039760.930 5070900.930 5083931.340 5192631.890 5023742.360
5084604.570 5167676.310 5075430.380 5008563.520 5114732.660 4977078.700 5011198.490 5132824.390 5098590.550 5252526.630
```

计算结果保存为文件

3.2 正确性验证

本程序所有函数实现（均选择 float 或 double）算出结果在小数点后两位无明显差异，故不重复附图。

```
"/Users/hezean/Documents/GitHub/SUSTech-CSUG/CS205 C++/proj/proj2/src/proj2" mat-A-2048.txt mat-B-2048.txt out2048.txt
Initialize Mat (2048*2048) finished. Time spent: 1.62767900 sec
Initialize Mat (2048*2048) finished. Time spent: 1.59910100 sec

 >>> Product calculated: by <dot_strassen>. Time spent: 63.60757100 sec
5252342.29   5053432.25   5087801.58   ...   5078388.40   5001864.77   5172207.06
5087195.58   4995317.88   5043294.43   ...   4972799.48   4868584.08   5114608.45
5232809.33   5175333.17   5066051.67   ...   5083236.05   5026657.89   5222874.31
...
5254304.21   5022093.36   5103064.68   ...   5116843.85   5002038.89   5161521.93
5155669.19   5057525.70   5082871.68   ...   5109429.86   5006605.10   5141488.58
5160491.47   5101746.54   5141126.60   ...   5112892.08   5002619.57   5181736.57
Answer saved (tar = out2048.txt). Time spent: 1.47088200 sec

Process finished with exit code 0
```

double 模式下的结果部分预览

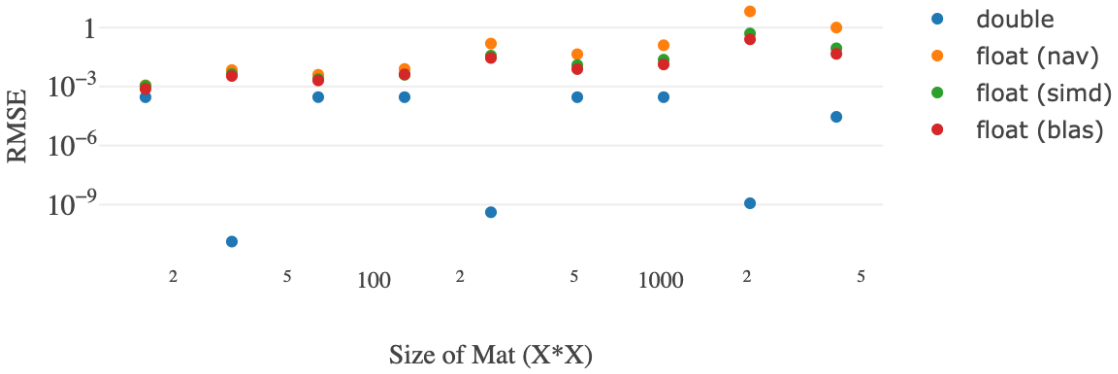
```
[[5252342.29 5053432.25 5087801.58 ... 5078388.4  5001864.77 5172207.06]
 [5087195.58 4995317.88 5043294.43 ... 4972799.48 4868584.08 5114608.45]
 [5232809.33 5175333.17 5066051.67 ... 5083236.05 5026657.89 5222874.31]
 ...
 [5254304.21 5022093.36 5103064.68 ... 5116843.85 5002038.89 5161521.93]
 [5155669.19 5057525.7  5082871.68 ... 5109429.86 5006605.1  5141488.58]
 [5160491.47 5101746.54 5141126.6  ... 5112892.08 5002619.57 5181736.57]]
RMSE =  1.1736362226873855e-09
```

与 NumPy 的结果对比，小数点后两位均一致，RMSE 可见 10^{-9} 级，即认为两者一致（程序正确性有保障）

```
"/Users/hezean/Documents/GitHub/SUSTech-CSUG/CS205 C++/proj/proj2/src/proj2" mat-A-2048.txt mat-B-2048.txt out2048.txt
Initialize Mat (2048*2048) finished. Time spent: 2.09673500 sec
Initialize Mat (2048*2048) finished. Time spent: 1.96762900 sec

 >>> Product calculated: by <dot_strassen>. Time spent: 42.95168000 sec
5252339.50   5053456.50   5087809.00   ...   5078386.00   5001864.50   5172208.00
5087206.00   4995316.50   5043302.00   ...   4972799.00   4868584.00   5114607.50
5232831.00   5175318.50   5066068.00   ...   5083237.50   5026658.00   5222873.50
...
5254305.50   5022093.00   5103067.50   ...   5116858.50   5002034.00   5161519.00
5155671.00   5057522.50   5082871.00   ...   5109413.50   5006610.50   5141485.00
5160491.00   5101745.50   5141128.50   ...   5112875.50   5002609.50   5181746.50
Answer saved (tar = out2048.txt). Time spent: 1.47001400 sec

Process finished with exit code 0
```

float 模式下的结果部分预览，RMSE=6.601，显著大于 double 模式

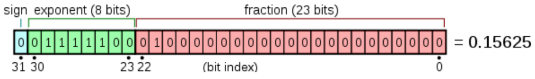
3.3 精度



	16	32	64	128	256	512	1024	2048	4096
double	2.88e-4	1.32e-11	2.91e-4	2.89e-4	4.09e-10	2.88e-4	2.89e-4	1.17e-9	2.89e-5
float (NAV)	1.16e-3	6.89e-3	3.97e-3	7.82e-3	1.53e-1	4.40e-2	1.26e-1	6.60e+0	1.00e+0
float (SIMD)	1.09e-3	4.35e-3	2.38e-3	3.98e-3	3.73e-2	1.23e-2	2.32e-2	5.03e-1	8.81e-2
float (BLAS)	7.67e-4	3.45e-3	2.05e-3	4.17e-3	2.84e-2	7.71e-3	1.33e-2	2.53e-1	4.59e-2

以上数据为不同大小的矩阵使用不同模式算出结果与 NumPy 结果差异的 RMSE，RMSE 越小指标越好。其中使用 OpenBLAS 和 SIMD（AVX2 指令集）只在 float 模式下开启[10]（否则编译错误），但可能 OpenBLAS 做了特殊优化，其计算误差是 float 模式下最小的（见 float (BLAS)）、其次是向量化（见 float (SIMD)），但整体上所有 float 模式的精度不会相差过多，手动实现的乘法（见 float (NAV)）精度最差，且与 double 相差了至少一个数量级。除去 SIMD 和 BLAS 的优化，单论 float 与 double 的精度差别，主要在于其位数不同。

- `float`: single precision floating-point type, 32 bits



$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

- `double`: double precision floating-point type, 64 bits

可见 float 存底数的部分只有 23 位，$2^{23} = 8388608$，即 float 有七到八位的有效数字（底数部分超过 1.8388608×10^n 时只能再牺牲一位精度，表示为 $x \times 10^{n+1}$），这也解释了为什么上图（float 模式下的结果预览）小数点后均为 50 或 00——超过有效数字位数。同理，double 的底数能表示更多的有效位数，这时小数点后仍有较多的有效位，更精确。

注意到有三个样本点（官方三组测试样例，double 模式）的误差远小于其余样本，分析输入数据易发现，三个题目样例均为一位小数正数，而自己生成的测试文件为随机数输出的三位小数，有正有负，这也造成了 double 能很好的保存官方样例的矩阵计算的中间数据，在保存三位小数的结果时损失较少，而其余

[10]2021.9.29 更新：在 StackOverflow 上了解到可以通过 _mm256_add_pd 对 __m256 寄存器进行 double 类型的运算，但此时一个寄存器只能存 4 个 double。时间原因，未再对程序进行修改。

测试数据本身在计算过程中就有较多损失，故有三个样本点与其余精度数据相差较大。

3.4 效率

3.4.1 double 与 float 对比

```
Initialize Mat (512*512) finished. Time spent: 0.17050700 sec
Initialize Mat (512*512) finished. Time spent: 0.16589200 sec

>>> Product calculated: by <dot_n3>. Time spent: 1.57138600 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.52451500 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.55563600 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.57691900 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.58371600 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.45480800 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.43131100 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.44338300 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.43850900 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.44863900 sec
```

```
Initialize Mat (1024*1024) finished. Time spent: 0.61516400 sec
Initialize Mat (1024*1024) finished. Time spent: 0.62837600 sec

>>> Product calculated: by <dot_n3>. Time spent: 14.35710100 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.14097000 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.22857800 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.22876100 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.64915900 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.38464200 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.24697600 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.30221100 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.39109600 sec
>>> Product calculated: by <dot_n3>. Time spent: 14.11828900 sec
```

```
Initialize Mat (512*512) finished. Time spent: 0.18094500 sec
Initialize Mat (512*512) finished. Time spent: 0.17617100 sec

>>> Product calculated: by <dot_n3>. Time spent: 1.40319100 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.16263300 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.19340100 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.24963200 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.17396200 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.13953700 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.11583800 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.16648700 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.17710600 sec
>>> Product calculated: by <dot_n3>. Time spent: 1.16915300 sec
```

```
Initialize Mat (1024*1024) finished. Time spent: 0.79387100 sec
Initialize Mat (1024*1024) finished. Time spent: 0.75376400 sec

>>> Product calculated: by <dot_n3>. Time spent: 12.26710400 sec
>>> Product calculated: by <dot_n3>. Time spent: 11.05038900 sec
>>> Product calculated: by <dot_n3>. Time spent: 12.58839400 sec
>>> Product calculated: by <dot_n3>. Time spent: 11.05106100 sec
>>> Product calculated: by <dot_n3>. Time spent: 10.93687900 sec
>>> Product calculated: by <dot_n3>. Time spent: 11.23438200 sec
>>> Product calculated: by <dot_n3>. Time spent: 11.35320800 sec
>>> Product calculated: by <dot_n3>. Time spent: 11.01816800 sec
>>> Product calculated: by <dot_n3>. Time spent: 10.81626300 sec
>>> Product calculated: by <dot_n3>. Time spent: 10.79556600 sec
```

控制变量，为了体现时间差异，（不开启编译器优化的情况下）使用朴素乘法分别计算 512，2048 大小的矩阵 10 次。

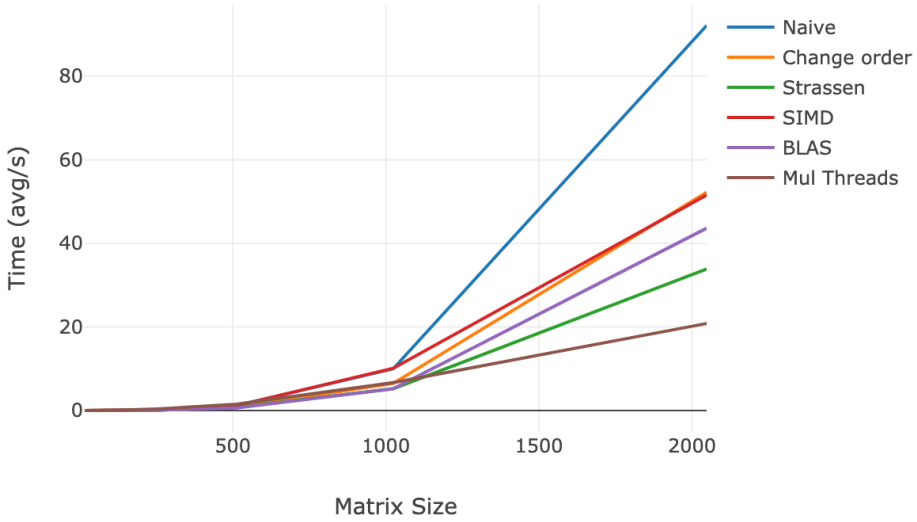
	512	1024
double (avg / ms)	1502.88	14304.77
float (avg / ms)	1192.39	11311.13

对于 512 规模的方阵，使用 float 比 double 节省了 20.66% 的时间，对于 1024 规模的方阵，使用 float 比 double 节省了 20.92% 的时间。关于此现象，一个合理的解释是 float 的 byte 数只有 double 的一半，在将数据从内存中取出供 ALU 运算时（包括存入缓存）的 IO 开销更少——CPU 在等待数据时被闲置，而使用 float 有益于内存密集的数据结构（主要是访问第一个被乘矩阵的一整行时），就 ALU 来说计算 float 和 double 的耗时没有太大差别。

本程序未实现 double 的 SIMD 运算，若实现，可以分析的是寄存器一次能存的 float 数量是 double 的两倍。

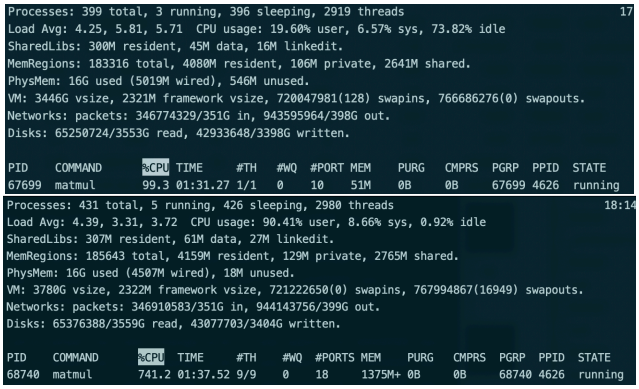
3.4.2 优化提升

关于 float 和 double 的效率分析主要见上，为了方便在此部分只使用 float 对比不同的乘法实现存在的显著差异。



其中朴素乘法的时间复杂度为 $O(n^3)$，耗时增长最快，与之对应的是 Strassen 算法凭借复杂度的优势在大矩阵运算时耗时明显减少（这里设置边界条件 256，对于小矩阵，Strassen 使用访存优化的方法计算）。对于将朴素乘法三重 for 循环（记为 ijk），通过调换乘法顺序（ikj）减少了 IO 开销也能做到很好的效率提升，若使用一维数组提升更加明显。SIMD 理论上计算向量点乘时性能能提升 8 倍，但在此例中因为需要创建新数组，将在内存中不连续了矩阵 B 的某列放入，这一步拖低了其效率。BLAS 的耗时虽没有 Strassen 短，但其精度更高，同样也是由于创建了新的临时数组存矩阵 B 的列而增加了耗时。多线程的方法使得 CPU 和 RAM 的利用率达到了最大（如下图所示，前面是单线程的方法，后面是多线程），因此大幅度提升效率（对于 2048 阶的矩阵，多线程的效率比朴素乘法提升了 77.39%），但在小矩阵运算时反而耗时更长，这是因为线程池

每次需要花费时间开辟线程，对于小矩阵，甚至做每一个向量点乘的时间都要小于开线程的时间，得不偿失。



3.4.3 编译器 O3 优化

相比起手动进行向量化等优化，g++ 自带的 O3 优化可以在编译时加入*-O3* 参数打开。这将降低编译速度并改变代码逻辑，当工程较为复杂时甚至可能导致逻辑改变错误而程序运行异常。本程序在使用 O3 优化后并无错误，结果正确。

```
Initialize Mat (2048*2048) finished. Time spent: 1.33691500 sec
Initialize Mat (2048*2048) finished. Time spent: 1.30208600 sec

>>> Product calculated: by <dot_n3>. Time spent: 17.79410100 sec

>>> Product calculated: by <dot_change_ord>. Time spent: 2.10350600 sec

>>> Product calculated: by <dot_strassen>. Time spent: 0.70183400 sec

>>> Product calculated: by <_mm_dot_float>. Time spent: 18.01669000 sec

>>> Product calculated: by <dot_blas>. Time spent: 18.22570600 sec

>>> Product calculated: by <dot_mul_threads>. Time spent: 11.47658600 sec
```

2048 阶矩阵乘法	初始化矩阵	朴素乘法	访存优化	Strassen	SIMD	BLAS	多线程
优化前 / s	2.614	92.134	52.241	33.883	51.556	43.623	20.835
开启 O3 / s	1.319	17.794	2.103	0.702	18.017	18.226	11.477
性能提升 / %	49.54	80.69	95.97	97.93	65.05	58.22	44.91

可见编译器集成多种优化方案确实比自己写代码更周全。但同时，自己写的一些代码（如手动分配了寄存器）影响了 O3 的选项，降低了性能提升率。对本项目起到明显提升的主要有以下几项[11]：

l -fdefer-pop 延迟栈的弹出时间。当完成一个函数调用，参数并不马上从栈中弹出，而是在多个函数被调用后，一次性弹出。

l -fdelayed-branch 这种技术试图根据指令周期时间重新安排指令。它还试图把尽可能多的指令移动到条件分支前，以便最充分的利用处理器的治理缓存。

l -fcprop-registers 因为在函数中把寄存器分配给变量，所以编译器执行第二次检查以便减少调度依赖性(两个段要求使用相同的寄存器) 并且删除不必要的寄存器复制操作。

[11] gcc -O0 -O1 -O2 -O3 四级优化选项及每级分别做什么优化 (blog.csdn.net/zgaoq/article/details/83039511)

l -fforce-mem 在做算术操作前，强制将内存数据 copy 到寄存器中以后再执行。这会使所有的内存引用潜在的共同表达式，进而产出更高效的代码，当没有共同的子表达式时，指令合并将排出个别的寄存器载入。

l -foptimize-sibling-calls 优化相关的以及末尾递归的调用。

l -finline-functions 内联简单的函数到被调用函数中。由编译器启发式的决定哪些函数足够简单可以做这种内联优化。

4 困难及解决

4.1 模版类的使用

对于模版类 Mat<T>，我一开始使用了常规的 hpp 和 cpp 文件，但是在编译阶段会报错：*Undefined symbols for architecture x86_64*。这是因为在编译过程中实例化模板时，编译器使用给定的模板参数 <typename T> 创建一个新类——模板不能编译为代码，只能编译实例化模板的结果。网上常见的解决方法是将.cpp 里的内容直接放进 hpp 里（使用内联函数），但这样不规范且不优雅。最终发现可以将 cpp 文件改为 tpp 文件[12]，开头不引用 hpp，并在 hpp 文件的下方使用 include 将其预编译进去。其实本质依然是前者，但这样更为优雅。

4.2 指针、引用与栈内存

一开始我忘记用指针或引用向函数传递参数，结果容易导致函数返回后栈内存释放，析构函数在尝试 delete 数组时产生异常。经过本次项目，我现在牢记 C++ 默认使用按值传递，合理的使用引用和指针也有助于提升程序速度，这点我在以后的编程中牢记并善加运用。

5 总结

在人工智能热潮的今天，为了提升神经网络等模型的精度，计算机科学家们竭尽一切的研究提升矩阵乘法这类底层的、大量基础运算的性能。相比起经过优化的 NumPy 等成熟的计算库（在验证答案时，计算 2048 规模矩阵只用将近 1 sec），本程序依然存在许多可以优化的方面，如二维数组改一维以更好的实现访存优化、使用 void 函数（传入指针）减少按值传递的时间开销等，有些优化方案也是在完成了大部分代码时查资料了解到的，受时间限制暂未能实现。

完成本项目的过程中探索了一些新的语言特性（如第四节所讨论），对 C++ 的掌握更加深入，更激动人心的是我在本项目中第一次真正用上了硬件级优化，其大幅提升了计算效率（将 2048 矩阵乘法从最长大约需要 92 秒，优化至最快仅需 0.7 秒！），带来了纯算法所无法比拟的性能提升，这也展现出了现代计算机科技的强大。但是自己的代码优化能力依旧有待提升，手写的 SIMD 和多线程仍不如编译器开启 O3 优化后效率高。本次项目打开了硬件优化极致提升性能的新大门，也为接下来的学习注入动力。

[12]Why can templates only be implemented in the header file? (stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file)