Project 1

Matrix Multiply

61520324 许睿

1. 问题介绍

常见的计算机将存储结构大致分为内存，缓存和外部存储三大区域. 其中 CPU 能直接对内存中的数据进行运算，而不能直接对外部存储的数据进行访问. 因此如果内存计算中需要存储在外部的数据时，需要 CPU 发动调度去外部存储介质中读取相应的数据，此步骤非常耗时. 因此出现了缓存，缓存根据局部性进行设计，将 CPU 最近可能将访问的数据存入. 本问题主要聚焦于外部存储中的矩阵乘法问题，矩阵乘法需要三重循环，由于矩阵的存储方式为线性存储，三重循环的顺序会极大影响矩阵乘法的效率和缓存的命中次数. 该实验的目的是统计缓存的未命中次数(miss hit)，探究不同乘法顺序对 miss hit 的影响.

2. 运行环境

- OS: Windows 10
- 最小 CMake 版本: 3.23
- 实现语言: C++
  - C++标准: std17 及以上
- IDE: CLion 2022.2

3. 设计创新点

- 可以人为指定乘法操作中 for 循环的顺序，不用写 6 个 for 循环实现
- 使用模板类型，适应更多的数据类型
- 使用模板偏特化实现多个数据类型的随机生成
- IO 操作，矩阵属性和矩阵乘法分开定义，方便管理

4. 实现程序

a) 程序框架

本程序分为 4 个头文件: Matrix.h 定义了矩阵的基本属性和矩阵生成器; MatrixIO.h 定义了缓存数组以及读写相关的函数; Order.h 定义了命名空间 Order，用于对字符串类型的乘法顺序进行转化和操作; MatrixMultipier.h 定义了常用的接口. Fig 1.给出了本程序的主要模块以及联系.

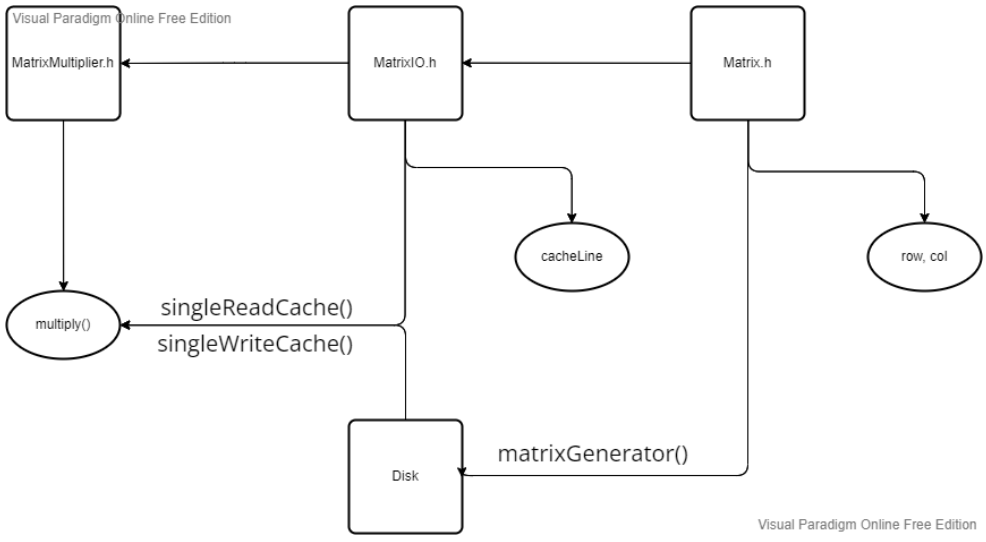

Fig 1. 模块之间的联系

接下来将详细介绍每个模块的实现思路和部分代码.

b) Matrix.h 实现

设计这个头文件的原意是将 IO 的读写，cache line 中的内容整合在一起，后来写到后面觉得有些凌乱，于是把和 IO 相关的代码分离了出去，于是本头文件只包含了矩阵的相关属性:

```
template<typename T>
class Matrix {
private:
    int row;
    int col;
    …
};
```

其中初始化行列使用的是成员函数`void init(int row, int col);`. 根据行列的值，可以调用成员函数`void matrixGenerator(const std::string &file_loc, int bottom = 0, int top = 10);`. 其中 file_loc 表示的输出文件路径, bottom 和 top 分别表示随机产生的数据的范围.

本文件的创新点在于：由于是模板实现，在随机产生数据的时候会受限于 C++11 的随机数函数，因此需要在编译时确定模板类型. 此处使用

```
template<typename T>
constexpr bool is_int() {
    return false;
}

template<>
constexpr bool is_int<int>() {
    return true;
}
```

进行模板类型推导(模板偏特化)，并在 if 的判断语句中使用 constexpr(C++17)在编译时期确定此时使用的模板类型是什么，从而确定随机产生的数据类型.

```
if constexpr (static_cast<bool>(is_int<T>())){}
```

c) MatrixIO.h 实现

本头文件是从 Matrix.h 头文件中脱离出来的，负责与 Disk 进行 IO 操作的封装. 它继承于 Matrix.h 以方便使用矩阵的相关属性. 私有成员包括缓存数组，以及记录读写位置和读写 miss hit 的变量.

```
template<typename T>
class MatrixIO : public Matrix<T> {
private:
    int cacheSize{};
```

```
    std::vector<T> cacheLine;

    std::vector<long long> read_p;
    std::vector<long long> write_p;

    int readBlockNum;  // the record of the number of blocks have been
read this turn
    int writeBlockNum; //

    int readMissTimes;
    int writeMissTimes;
    …
};
```

由于是主要用于与 disk 进行交互，因此其成员函数包括

```
    void init(int row, int col, int cache_size);
    void setCacheSize(int cache_size);
    // file to cache
    void cacheRead(const std::string &file_loc);
    // cache to file
    void cacheWrite(const std::string &file_loc);
    // cache to matrix single value
    T singleReadCache(const std::string &file_loc, int row_i, int
col_j);
    // matrix to cache single value
    void singleWriteCache(const std::string &file_loc, T value, int
row_i, int col_j);
    std::pair<int, int> getMissTimes();
    int getCacheSize();
    bool emptyCache();
```

其中最主要的是 singleReadCache()和 singleWriteCache()两个函数，用于在做乘法时直接从 cache 取出数据进行乘法并写回数据；而 cacheRead()和 cacheWrite()两个函数用于在前两个函数进行时，在以下时刻进行 cache 的文件读写：

- 当访问的矩阵下标大于 cache 时发生 miss hit，此时需要将 cache line 向前读取直到满足矩阵下标(此处以 singleReadCache()为例)

```
if (cache_ind >= this->cacheSize) {
        // update the cacheLine value when cache_ind is greater than
current cacheSize
        // until cache_ind < cache_ind
        while (cache_ind >= this->cacheSize) {
            this->readBlockNum += 1;
```

```
            cache_ind -= this->cacheSize;
            cacheRead(file_loc); // update cache value, file->cache
        }
        return this->cacheLine[cache_ind];
```

- 当访问的矩阵下标小于 0 时发生 miss hit，此时需要将 cache line 向后读取直到满足矩阵下标(此处以 singleReadCache()为例)

```
else {
        //cache_ind < 0
        // roll back to former file_p one by one, until cache_ind >= 0
        readMissTimes += readBlockNum;
        while (cache_ind < 0) {
            read_p.pop_back();
            read_p.pop_back();
            readBlockNum -= 1;
            readMissTimes += 1;
            cache_ind += cacheSize;
            cacheRead(file_loc);
        }
        return this->cacheLine[cache_ind];
    }
```

d) Order.h 实现

本头文件主要实现的是矩阵乘法中顺序的确定，方便自己能通过字符串指定所需的乘法顺序.

```
namespace Order {
    enum struct Sequence {
        I,
        J,
        K
    };
    std::vector<Sequence> getOrder(const std::string &order) {…}
    std::ostream &operator<<(std::ostream &out, Sequence seq) {…}
}
```

其中 getOrder()可以将输入的字符串转化为 Seqence 枚举类型，而下面的函数用于重载输出流，方便输出.

e) MatrixMultiplier.h 实现

MatrixMultiplier 继承于 MatrixIO，相当于在 IO 接口的基础上封装乘法操作. 本头文件主要的函数在于 multiply(). 创新之处在于能人为指定乘法顺序(是"ijk"还是"kij"等顺序). 该函数接受两个 MatrixMultiplier 对象，一个乘法顺序，两个输入文件和一个输出文件，返回为 MatrixMultiplier 对象. 函数中一个主要的数据结构是用于存储当前行列值(即 I, j, k)优先级和

限制值的向量:

```
    std::vector<std::pair<int, int> > indices(3, std::pair<int, int>(0,
0));

    // define constrain
    indices.at(0).second = this->getRow();
    indices.at(1).second = mat_b.getCol();
    indices.at(2).second = this->getCol();
```

向量中每一个元素为一个 pair，向量的下标 0, 1, 2 分别表示 I, j, k; pair 的第一个元素表示该行列值在 for 循环的优先顺序, 0 为最低优先级，即最外层循环, 2 为最高优先级，即最内层循环; pair 的第二个元素表示 I, j, k 分别的限制值，作为 for 循环退出的条件. For 循环的结构如下图 Fig 2.所示:

```
for (indices.at((int) mul_order.at(0)).first = 0;
     indices.at((int) mul_order.at(0)).first < indices.at((int) mul_order.at(0)).second;
     ++indices.at((int) mul_order.at(0)).first) {

    for (indices.at((int) mul_order.at(1)).first = 0;
         indices.at((int) mul_order.at(1)).first < indices.at((int) mul_order.at(1)).second;
         ++indices.at((int) mul_order.at(1)).first) {

        for (indices.at((int) mul_order.at(2)).first = 0;
             indices.at((int) mul_order.at(2)).first < indices.at((int) mul_order.at(2)).second;
             ++indices.at((int) mul_order.at(2)).first) {
```

Fig 2. For 循环嵌套结构(使用截图原因是希望保持格式)

如前所述, MatrixIO.h 中定义的 singleReadCache()和 singleWriteCache()用于乘法计算的读取和写入，如 Fig 3.所示:

```
T a_value = this->singleReadCache(file_A, indices[0].first, indices[2].first);
T b_value = mat_b.singleReadCache(file_B, indices[2].first, indices[1].first);

T c_value = a_value * b_value;

mat_c.singleWriteCache(file_C, c_value, indices[0].first, indices[1].first);
```

Fig 3.单元素乘法

到此为止，主要的函数声明和定义介绍完毕.

f) 算法思路

- 声明两个 MatrixMultiplier 对象，指定读写文件，将参数传入 multiply()，注意乘法顺序
- 在 multiply()中，先为两个 MatrixMultiplier 随机生成矩阵元素，并将相应的 cacheLine 读满
- 定义并根据传入的乘法顺序初始化下标向量
- 进入 for 循环，从相应的 cacheLine 中读取元素进行乘法并写入目标 cacheLine. 若 cacheLine 发生读写的越界，则要么回退读取文件，要么往前读取文件，直到满足下标要求.
- 退出 for 循环，返回 MatrixMultiply 结果对象，打印统计数据.

5. 测试结果

a) 小数据测试(内存操作)

以 3×3 矩阵为例, cacheLine 大小设置为 10(>9)，以检验乘法的正确性.

矩阵 A:

Fig 4. 小数据，矩阵 A

矩阵 B:

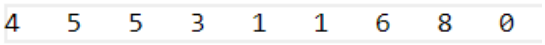

Fig 5. 小数据，矩阵 B

最终的输出结果为:

矩阵 C:

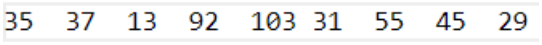

Fig 6. 小数据，矩阵 C

其中输出日志为:

multiply order i j k time spent 3 ms Matrix A matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix B matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix C matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0	multiply order i k j time spent 3 ms Matrix A matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix B matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix C matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0	multiply order j i k time spent 8 ms Matrix A matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix B matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix C matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0
multiply order j k i time spent 4 ms Matrix A matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix B matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix C matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0	multiply order k i j time spent 2 ms Matrix A matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix B matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix C matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0	multiply order k j i time spent 2 ms Matrix A matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix B matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0 Matrix C matrix row 3 | col 3 cacheSize 10 missTimes: readMissTimes 0| writeMissTimes 0

可以发现，各个矩阵的运算效率相仿，并且对磁盘的读写均为零.

因此从测试的角度而言，乘法结果和读写测试均正确.

b) 大数据测试(外部乘法)

为节约运行时间，以 100×100 的矩阵乘法为例，设置 cacheLine 大小为 3136(随机) < 100×100. 乘法过程中需要与磁盘进行 IO 读写. 运行完成后，统计信息如下表所示:

multiply order i j k time spent 25814 ms Matrix A matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 894| writeMissTimes 0 Matrix B matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 59997| writeMissTimes 0 Matrix C matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 0| writeMissTimes 6	multiply order i k j time spent 6481 ms Matrix A matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 3| writeMissTimes 0 Matrix B matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 597| writeMissTimes 0 Matrix C matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 0| writeMissTimes 303	multiply order j i k time spent 33253 ms Matrix A matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 597| writeMissTimes 0 Matrix B matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 59997| writeMissTimes 0 Matrix C matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 0| writeMissTimes 303

multiply order j k i time spent 646762 ms Matrix A matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 59997| writeMissTimes 0 Matrix B matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 597| writeMissTimes 0 Matrix C matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 0| writeMissTimes 30003	multiply order k i j time spent 6783 ms Matrix A matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 597| writeMissTimes 0 Matrix B matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 894| writeMissTimes 0 Matrix C matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 0| writeMissTimes 303	multiply order k j i time spent 729417 ms Matrix A matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 59997| writeMissTimes 0 Matrix B matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 3| writeMissTimes 0 Matrix C matrix row 100 | col 100 cacheSize 3136 missTimes: readMissTimes 0| writeMissTimes 30003

从表中数据可以看出来，矩阵 A 的读取 miss hit 最少的组合是"I, k, j"，只有 3 次 miss hits，矩阵 B 的读取 miss hits 最少的组合是"I, k, j"，"j, k, i"，而矩阵 C 的写入 miss hits 最少的组合是"I, j, k"，但是组合"I, k, j"，"j, I ,k"，"k, I, j"的 miss hits 也只有 303 次，因此综合而言 miss hits 最优的组合是"I, k, j".

理论计算:

Ijk: 理论的总 miss hits 为

$$\frac{n^3}{w}(\frac{1}{n}+1+w)$$

即, 1000322 次. 实验结果 60897 次，实验结果和理论值相差很大，可能原因是 N 不够大.

Ikj: 理论的总 miss hits 为

$$\frac{n^3}{w}(2+\frac{1}{n})$$

即 641 次，实验结果为 903 次，大于理论值.

Jik: 理论公式

$$\frac{n^3}{w}+n^3+n^2$$

理论为 1010318 次，实验为 60897 次.

Jki: 理论公式

$$n^2+n^2+n^3$$

理论为: 1020000 次，实验为 90597 次

Kij: 理论公式

$$\frac{n^3}{w}(2+\frac{1}{n})$$

理论为 641 次: 实验为 1794 次

Kji: 理论公式

$$n^3+\frac{n^2}{w}+n^3$$

理论为: 1000003 次，实验为 90003 次

从理论和实际来看，顺序为"ikj"和"kij"时，乘法操作的效率会比较高.

6. 实验感想

本次实验从实际出发，让我感受到了矩阵乘法的顺序对性能造成的影响. 实验中由于不想写 6 个 for 循环，于是设计了下标向量，方便进行操作，这让我对"程序员要避免造轮子"的箴言有了更深的理解. 其次是将 IO 操作和内存操作分离，给后面的实验提供了启示.