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.

1. **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

1. **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

1. **实现程序**
   1. **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.

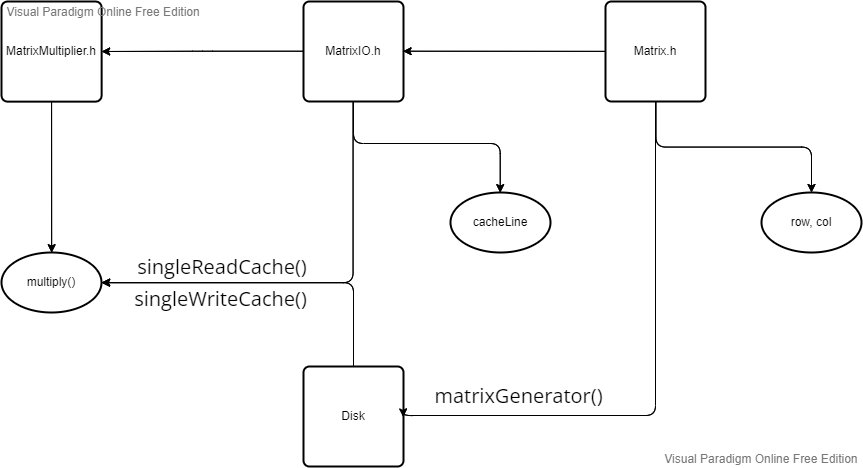


Fig 1. 模块之间的联系

接下来将详细介绍每个模块的实现思路和部分代码.

* 1. **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>())){}

* 1. **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];

    }

* 1. **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.

* 1. **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:

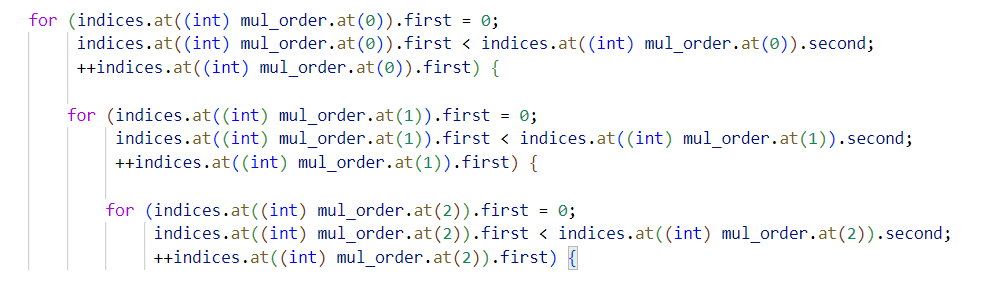


Fig 2. For循环嵌套结构(使用截图原因是希望保持格式)

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:

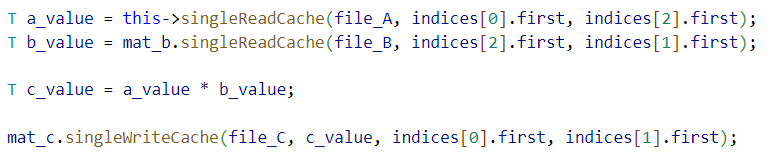


Fig 3.单元素乘法

到此为止, 主要的函数声明和定义介绍完毕.

* 1. **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.

1. **测试结果**
   1. **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:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

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.

* 1. **大数据测试(外部乘法)**

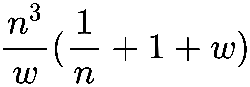
为节约运行时间, 以100×100的矩阵乘法为例, 设置cacheLine大小为3136(随机) < 100×100. 乘法过程中需要与磁盘进行IO读写. 运行完成后, 统计信息如下表所示:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

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”.

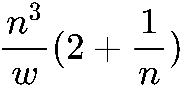
理论计算:

**Ijk:** 理论的总miss hits为



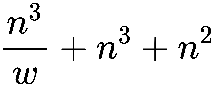
即, 1000322次. 实验结果60897次, 实验结果和理论值相差很大, 可能原因是N不够大.

**Ikj:** 理论的总miss hits为



即641次, 实验结果为903次, 大于理论值.

**Jik:** 理论公式



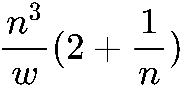
理论为1010318次, 实验为60897次.

**Jki:** 理论公式



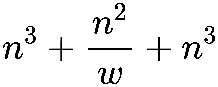
理论为: 1020000次, 实验为90597次

**Kij:** 理论公式



理论为641次: 实验为1794次

**Kji:** 理论公式



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.

1. **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.