CG2023 HW1 Report

PB20111686 黄瑞轩

目标

本次实验的目的是实现一个矩阵类 Matrix，并且和 Eigen 库的计算做比较。加分项是实现稀疏矩阵 SparseMatrix。

Matrix 的实现思路

Matrix 类的数据是用一维数组存储二维数据，使用 RAII 的方式进行内存空间的申请与销毁：

```
// constructor with initializer list
Matrix(int r, int c) : rows(r), cols(c) { this->data = new T[c * r]; }
// destructor
~Matrix() { delete[] this->data; }
```

需要重载 Matrix 类的算符以支持矩阵与矩阵、矩阵与常量之间的运算。在重载运算符时，基本需要重复写一个二重循环，这里使用了一个范式把这个重复的二重循环提取出来：

```
template <typename F>
void _do_on_every_item(Matrix<T> &rhs, F opers) {
    this->_check_dim_issame(rhs);
    for (int i = 0; i < this->rows; i++)
        for (int j = 0; j < this->cols; j++)
            opers((*this)(i, j), rhs(i, j));
}
```

这里 `opers` 应该放一个 lambda 表达式，为了避免写 lambda 表达式的冗长类型，这里的做法是把其类型用一个模版类型 `F` 来代替。这样，比方说在后面的重载 `operator+=(Matrix& rhs)` 时，就只需要写

```
Matrix<T> &operator+=(Matrix<T> &rhs) {
    this->_do_on_every_item(rhs, [=](T &item, T r_item) { item += r_item; });
    return *this;
}
```

个人感觉代码还算易懂，`rhs` 提供了右操作数，lambda 表达式则清楚地表达了对每个元素做 += 运算。

类似 `operator+` 和 `col` 这样的函数，返回的是一个新的矩阵对象，如：

```
Matrix<T> col(int c) {
    this->_check_c_exceeds(c);
    auto col_copy = Matrix<T>(this->rows, 1);
    for (int i = 0; i < this->rows; i++)
        col_copy(i, 0) = (*this)(i, c);
    return col_copy;
}
```

直接这样写会导致在 return 的时候调用 `col_copy` 的析构函数，从而数据被释放，复制得到的对象内部的 `data` 是一个悬空指针。其实这里编译器帮我做了返回值优化（RVO），一开始没报错，使用编译选项 `-fno-elide-constructors` 禁止 RVO 后报错。解决办法是写一个拷贝复制函数 `Matrix(const Matrix<T>& m)`，这样会在复制的时候调用这个函数进行深拷贝。

SparseMatrix 的实现思路

稀疏矩阵的数据结构为

```
template <typename R>
std::map<int, std::vector<std::pair<int, R>>>
```

这样比较方便访问某一行，只需要花 $O(\log N)$ 的时间查询。但是不太方便访问某一列，需要遍历这一行的所有元素才能知道有没有列号为 c 的元素，如果稀疏矩阵每一行非零元素个数是 $O(\log N)$ 的，那么随机访问的时间复杂度是 $O(\log N)$ 的。

因为使用这种数据结构的 SparseMatrix 不支持下标那样的直接随机访问，所以其 `operator()` 重载实现仅返回值，不能通过其来赋值，赋值需要使用接口 `assign(int r, int c, R v)` 来实现。当 `v == 0` 时，需要查找 `(r, c)` 对应的元素是否存在，如果存在则需要删除之，如果不存在则不需要做任何操作。

测试与结果

（1）Matrix 类的正确性

这一部分使用 `Matrix<int>`，$A, B \in \mathbf{int}^{6\times 7}$，用 MT19937 算法随机为矩阵上各个位置填充 1～20 的随机数值（避免抛出除以 0 错误，如果需要检查这一部分，修改随机数范围即可）。

除了计算 $(A * B)/(A + 2)$ 之外，还测试了 submat 和 row 函数，结果如下：

```
---------------------------------------
The information of Matrix [ A ]
---------------------------------------
this Matrix has size (6 x 7)
the entries are:
|    12   13    4    5   18   16   14 |
|    13   14    6   14    4    7    2 |
|    17   13    9    6   16    6   16 |
|    13   18    7    5   14   11    4 |
|    19   13    1    8    3   17    9 |
|     6    6   15   15   19    7   13 |
```

```
--------------------------------------
The information of Matrix [ B ]
--------------------------------------
this Matrix has size (6 x 7)
the entries are:
|    16    7    4    2   12   20    6 |
|    11   17   19    2    8   18    8 |
|     1    1    6    5    1   15    2 |
|     4   14   13   15    2   20    5 |
|     2   15   19    7    7   14   13 |
|     4   13    4   19   13   18    8 |
--------------------------------------
The information of Matrix [ C = A * B ]
--------------------------------------
this Matrix has size (6 x 7)
the entries are:
|   192   91   16   10  216  320   84 |
|   143  238  114   28   32  126   16 |
|    17   13   54   30   16   90   32 |
|    52  252   91   75   28  220   20 |
|    38  195   19   56   21  238  117 |
|    24   78   60  285  247  126  104 |
--------------------------------------
The information of Matrix [ D = A + 2 ]
--------------------------------------
this Matrix has size (6 x 7)
the entries are:
|    14   15    6    7   20   18   16 |
|    15   16    8   16    6    9    4 |
|    19   15   11    8   18    8   18 |
|    15   20    9    7   16   13    6 |
|    21   15    3   10    5   19   11 |
|     8    8   17   17   21    9   15 |
--------------------------------------
The information of Matrix [ E = C / D ]
--------------------------------------
this Matrix has size (6 x 7)
the entries are:
|    13    6    2    1   10   17    5 |
|     9   14   14    1    5   14    4 |
|     0    0    4    3    0   11    1 |
|     3   12   10   10    1   16    3 |
|     1   13    6    5    4   12   10 |
|     3    9    3   16   11   14    6 |
--------------------------------------
The information of Matrix [ F = A[1, 2, 3, 4] ]
--------------------------------------
this Matrix has size (3 x 3)
the entries are:
```

```
|     6   14    4 |
|     9    6   16 |
|     7    5   14 |
---------------------------------------
The information of Matrix [ G = A.r[4] ]
---------------------------------------
this Matrix has size (1 x 7)
the entries are:
|    19   13    1    8    3   17    9 |
```

经检验是正确的。

（2）SparseMatrix 类的正确性

这一部分使用 `Matrix<int>`，$A, B \in \mathbf{int}^{10\times 10}$，为了实现稀疏化是这样处理的：先生成一个 1～10 的随机数，当这个随机数值为 10 时，这个位置为非零值，其值是随机数生成器再生成一次的结果。（这样做其实是线性的非零值数而不是 $\log$ 的，但是小规模测试且不是重点，姑且这么做）

计算 $A * B$ 的结果如下，注：在打印结果时，输出稀疏矩阵中所有非零元素的「行号，列号，元素值」。

```
---------------------------------------
The information of Matrix [ SA ]
---------------------------------------
this SparseMatrix has size (10 x 10)
the entries are:
「 0, 8, 6 」 「 1, 8, 7 」 「 2, 2, 2 」 「 2, 3, 10 」 「 3, 0, 1 」 「 3, 2, 3 」 「 4, 2, 1 」
「 6, 2, 1 」 「 6, 3, 1 」 「 8, 6, 7 」 「 9, 1, 7 」 「 9, 4, 8 」 「 9, 9, 2 」
---------------------------------------
The information of Matrix [ SB ]
---------------------------------------
this SparseMatrix has size (10 x 10)
the entries are:
「 2, 7, 7 」 「 2, 8, 4 」 「 3, 0, 4 」 「 3, 8, 5 」 「 4, 9, 3 」 「 5, 6, 4 」 「 6, 2, 5 」
「 8, 6, 1 」 「 9, 1, 5 」 「 9, 7, 6 」
---------------------------------------
The information of Matrix [ SC = SA * SB ]
---------------------------------------
this SparseMatrix has size (10 x 10)
the entries are:
「 3, 0, 4 」 「 6, 2, 5 」 「 8, 6, 7 」 「 9, 1, 35 」
```

经检验是正确的，而且相乘之后的零值也因为 `assign` 的实现被直接删除了。

（3）Matrix 类和 Eigen 库进行性能比较

这一部分使用 `Matrix<double>`，$A, B \in \mathbf{double}^{100\times100}$，各个位置都是随机生成的数值。为了进行性能比较，目标是计算 $(A * B)/(A + 2)$。

使用 `std::chrono` 的纳秒级计数器：

```
auto start = std::chrono::high_resolution_clock::now();
auto D3 = D1 * D2;
auto D4 = D1 + 2;
auto D5 = D3 / D4;
auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
std::cout << "My matrix calculating time: " << duration.count() << " ns.\n";

// 将 Matrix 的内容拷贝到 Eigen 矩阵里

start = std::chrono::high_resolution_clock::now();
auto E3 = E1.cwiseProduct(E2);
auto __E1 = E1.array() + 2;
auto E4 = __E1.matrix();
auto E5 = E3.cwiseQuotient(E4);
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
std::cout << "Eigen matrix calculating time: " << duration.count() << " ns.\n";
```

一次测试结果如下，经过重复测试，Eigen 的时间性能比我写的好 1000 倍左右（还是库厉害啊

```
My matrix calculating time: 800417 ns.
Eigen matrix calculating time: 1250 ns.
```