CS205 · C/C++ Programming

Project 4: Class Matrix

何泽安 (He Zean)
12011323
Nov. 28, 2021

Contents

1 需求分析 **1**

2 代码实现 **1**

2.1 检测内存泄露 1
2.2 SmartPtr 代理类 3
2.3 Matrix 类 6
2.3.1 构造器 7
2.3.2 深拷贝 9
2.3.3 填充元素 10
2.3.4 Getters & Setters 11
2.3.5 重载矩阵运算操作符 11
2.3.6 operator== 15
2.3.7 矩阵转置、合并通道 15
2.3.8 friend operator << 16
2.4 MatException 及其派生类 17
2.5 初始化设置 17

3 测试样例及分析 **18**

3.1 正确性验证 19
3.2 性能测试 22

4 困难及解决 **25**

4.1 避免构造与析构阶段的内存泄露 25
4.2 模版类友元函数的链接问题 26
4.3 类的继承 26
4.4 OpenMP 并行 for 下抛出异常 27

5 总结 **27**

1 需求分析

考虑到课程进度，本次 Project 需要注意的重点在于类的数据封装与内存管理。本次的 features 包括：

① 使用了模版类以支持多种数据类型，并为常用的数据类型设置了别名；支持 SIMD 的数据类型将会使用向量化进行加速计算（为了减少工作量，暂仅支持 AVX2、AVX512 及 NEON），而其他数据类型逐个计算，这为矩阵所存数据带来了极大的灵活性——甚至可以存储任何用户自定义类型，只要其支持相关运算（重载了相应运算符）。

② 通过 placement new/delete，在测试阶段记录内存的申请与释放，更加方便客观的检查代码是否出现内存泄露；保存了申请内存时的附加信息，易于定位出错代码。

③ “对比 ARM 与 x86 平台下的差异”启示我们需要跨平台的设计，向较于 Project 3 中需要手动为 SIMD 的选择添加宏，本次我们利用编译器自动定义的对应预处理宏[1]检测环境，更加方便。

④ 使用 SmartPtr 作为代理，将计数器与资源绑定、对 Matrix 类封装了内存管理的部分细节，使用更加方便优雅[2]；作为课程由基本语法进入面向对象的转折点，相较于前两次 *project* 致力于提升性能，本次 *project* 更重视在交付用户使用时面对不同输入的健壮性，因此引入了异常处理机制，使程序在出现数学错误时将下一步操作的选择权交给用户，并将提示异常的功能挪至异常类中，Matrix 类主体代码更加简洁。

2 代码实现 为节省空间，部分代码被省略/调整格式/与头文件合并，非核心代码被删除；复用的 project2 工具函数不重复附上。

2.1 检测内存泄露

栈内存由编译器自动分配释放，因此我们只需要关注堆内存是否出现泄漏。考虑到堆内存一般由程序员手动通过 new 关键字申请，我们只需要记录此申请动作，并在 delete 时删除对应的一条记录，程序结尾检测是否有未删除的记录即可[3]。查阅资料[4][5]发现：

① *new operator*：用于动态分配内存并进行初始化，不能被重载，其先申请内存，然后调用构造函数初始化。

② *operator new*：标准库的函数，只分配内存而不进行初始化，可以重载。

delete 关键字及运算符与之类似。自然的，我们可以通过重载 new/delete 运算符的方式来优雅地记录堆上内存的申请与释放，且记录的 new 关键字调用处方便调试。

```
#if not defined(NDEBUG)

struct new_ptr_info {
    static std::vector<new_ptr_info> new_ptr_pool;
    std::string call_file;  int call_line;  // record where calls `new` or `new[]`
    void *where;            size_t size;    // for leak report
};
std::vector<new_ptr_info> new_ptr_info::new_ptr_pool;

void *operator new(size_t size, const char *file, int line) {
    void *ptr = malloc(size);
```

[1]本次测试使用的 gcc 及 clang 均会在 x86 开启 SIMD 指令集支持后添加相应宏；ARM 下无需特殊指令，自动开启 NEON 支持。

[2]这种方式也使得开发可以明确的分为确保内存管理正确性和编写业务代码的两个步骤，SmartPtr 和上述的 Debug-new 均在后续开发前即完成了较为完善的局部测试（报告第三节省去此部分的介绍，仅通过整体运行反映其正确性）。

[3]思想参考 MemTrack: Tracking Memory Allocations in C++ (www.almostinfinite.com/memtrack.html)

[4]More Effective C++ (Item 8: Understanding the different meanings of new and delete)

[5]A Cross-Platform Memory Leak Detector (wyw.dcweb.cn/leakage.htm)

```
    new_ptr_info::new_ptr_pool.push_back(new_ptr_info{file, line, ptr, size});
    return ptr;
}

inline void *operator new[](size_t size, const char *file, int line) {
    return operator new(size, file, line);
}

void operator delete(void *ptr, const char *file, int line) {
    std::cerr << "(" << file << ": " << line << ") Constructor failed, deallocating by C++
        Runtime System."
    << std::endl;
    erase_if(new_ptr_info::new_ptr_pool, [ptr](new_ptr_info &p) { return p.where == ptr; });
    free(ptr);
}

inline void operator delete[](void *ptr, const char *file, int line) {
    operator delete(ptr, file, line);
}

void operator delete(void *ptr) {
    if (ptr == nullptr) return;
    erase_if(new_ptr_info::new_ptr_pool, [ptr](new_ptr_info &p) { return p.where == ptr; });
    free(ptr);
}

void operator delete[](void *ptr) { operator delete(ptr); }

void check_leak() {
    if (!new_ptr_info::new_ptr_pool.empty())  // prompt
    else std::cerr << "No leak detected, well done!" << std::endl;
}

#define new new(__FILE__, __LINE__)

#endif  // NDEBUG
```

值得注意的是 *Effective C++, Item 52* 提到，“这意味着如果要对所有与 *placement new* 相关的内存泄露宣战，我们必须同时提供一个正常形式的 operator delete（用于构造期间无任何异常被抛出）和一个 placement 版本（用于构造期间有异常被抛出），后者的额外参数必须和 operator new 一样”。因此上面为了在避免构造函数出现异常，提供了 void operator delete(void *ptr, const char *file, int line)，并为程序员正常调用 delete 提供了不带额外参数的 delete 操作符。

本代码受 NDEBUG 宏控制，仅在不注重性能的测试阶段使用，另外可配合系统的活动监视器辅助判断是否出现（明显的）内存泄露。我们可以将输出的信息重定向至文件中，方便进一步分析。而不为 delete 和 delete[] 打印信息是因为，在重载运算符后，C++ STL 等封装的大量 delete 操作也会使用其，这会导致大量我们不关心的信息被打印出来。

```
int main() {
```

```
    freopen("mem.log", "w", stderr);
    /* do something */
    check_leak();
    fclose(stderr);
}
```

另一种较不优雅但实现简单的检查是否析构所有堆内对象的方式是在类内定义静态计数器变量，并在构造函数中自增，析构函数中自减，程序结束前检查所有涉及的类的未释放对象计数器是否均为零。但其不仅麻烦且难以检测构造函数中的 new operator，另外，如果程序员在一处忘记释放内存，那他同样也有可能忘记管理计数器。

2.2 SmartPtr 代理类

相较于如同 OpenCV 中将引用计数器存为 int* 的方法，使用结构体将数据与其引用计数器绑定在一起更为优雅（虽然在一般情况下访问单个元素增加了一次解引用，但对于计算密集型的操作可以先将数据指针取出，分摊影响极小）。此外，将（带计数器的）数据使用 SmartPtr 代为管理有助于减少 Matrix 类中处理引用计数的代码重复以及封装细节以便开发。

使用时在矩阵中将每个通道的数据存进堆内存并由一个栈内存中的 SmartPtr 对象代为管理，这也利用了栈内存对象在出作用域后会自动析构的“trigger”性质。SmartPtr 中只有一个指针作为成员变量，因此将其按值传递的代价是相当低的。

```
#pragma once
#include "matexcept.hpp"
#define HERE __FILE__, __LINE__

namespace mat {
template<class T>
class SmartPtr {
    template<class V>
    friend class SmartPtr;

    template<class V>
    friend class Matrix;

    struct MatElem {
        int row;  int col;  // original 2d-matrix size, note that ROI may apply a bias
        T *data;  int ref_cnt;

        ~MatElem() { try { delete[] data;  data = nullptr; } catch (...) {} }
    };

    MatElem *m_data;

public:
    SmartPtr(int r, int c) : m_data(nullptr) {
        if (r <= 0 || c <= 0) throw InvalidArgs(std::initializer_list<int>{r, c});
        try {
            m_data = new MatElem{r, c, new T[r * c], 1};
```

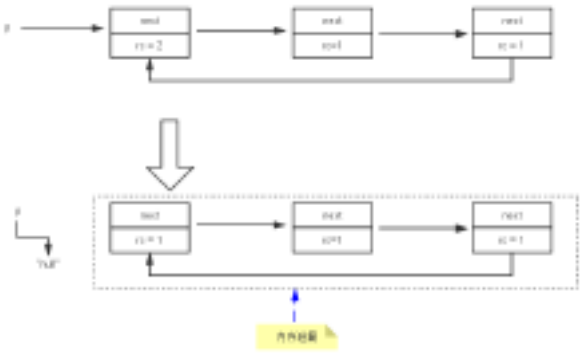
```
        } catch (...) {
            try { delete[] m_data->data;  m_data->data = nullptr; } catch (...) {}
            try { delete m_data;  m_data = nullptr; } catch (...) {}
            throw;
        }
    }

    SmartPtr(const SmartPtr &rhs) {  /* copy constructor */
        this->m_data = rhs.m_data;
        ++this->m_data->ref_cnt;
    }

    template<class V>  /* explicit cast the data managed by another smart_ptr */
    explicit SmartPtr(const SmartPtr<V> &op) {
        try {
            int row = op.m_data->row, col = op.m_data->col;
            this->m_data = new MatElem{row, col, new T[row * col], 1};
            auto thisData = m_data->data;
            auto opData = op.m_data->data;
#pragma unroll 16
            for (int i = 0; i < row * col; ++i)
                thisData[i] = static_cast<T>(opData[i]);
        } catch (...) { /* same as the above constructor SmartPtr(int, int) */ }
    }

    ~SmartPtr() {
        try {
            if (--m_data->ref_cnt == 0)  delete m_data;  m_data = nullptr;
        } catch (...) {}
    }

    SmartPtr &operator=(const SmartPtr &rhs) {
        if (this == &rhs || this->m_data == rhs.m_data) return *this;
        if (--m_data->ref_cnt == 0) delete m_data;
        m_data = rhs.m_data;
        ++m_data->ref_cnt;
        return *this;
    }

    T &operator()(int r, int c) {
        if (r <= 0 || c <= 0) throw InvalidArgs(std::initializer_list<int>{r, c});
        if (r > m_data->row) throw MatIndexOutOfBound(HERE, r, m_data->row);
        if (c > m_data->col) throw MatIndexOutOfBound(HERE, c, m_data->col);
        return m_data->data[r * m_data->col + c];
    }
};
}  // namespace mat
```

较为安全的内存管理 在构造函数中使用 try-catch 以处理可能的部分构造产生的内存泄露，析构函数“吸收”所有异常避免引发 terminate 导致的程序终止时大规模内存无法释放（更多分析见 4.1 节）。

有效性检查 在前面的 project 中只注重了矩阵乘法的性能提升，假设输入均有效。而本“工程”需要处理各种可能的输入，因此需要设置检查以保证鲁棒性[6]。相较于出错时打印提示并退出程序，这里选择抛出异常（见 2.4 节），并由主调函数决定如何处理，更加灵活。

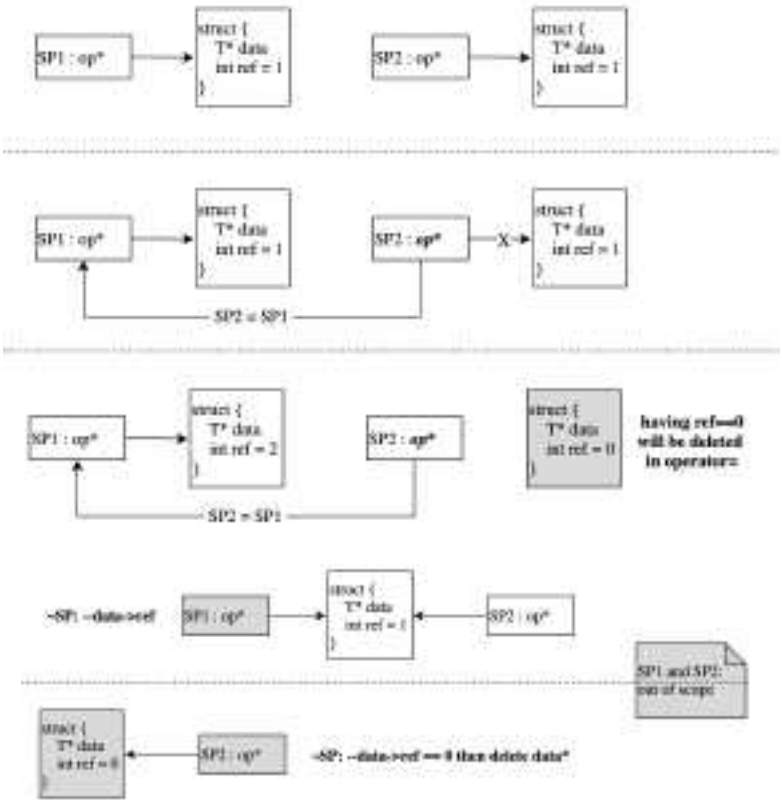
强制类型转换 提供了 SmartPtr(const SmartPtr &rhs) 作为拷贝构造器和 explicit SmartPtr(const SmartPtr<V> &op) 供强制类型转换（如 project 3 中讨论，在数据类型不便使用 SIMD 指令集时，循环展开能有效加速），但要注意的是对于非 primitive type <T>，用户在将其交给 SmartPtr 代为保管时需要提供对应的转换构造函数。

正确性 经多种测试，本代理类能很好的应对错误的参数、申请内存时的异常等情况，且在出作用域后能在正确的时机删除原始数据区，不漏删（见测试样例中的 leak report）不重删（故意在删除后不将指针设为 nullptr，再次删除必然导致系统杀死程序）。另外，注意到 memory 头文件中定义的 smart_ptr 存在典型的引用计数法的缺陷——无法释放环形引用，这是出于适用性考虑，其计数器在于指针，而非资源；而本代理类对计数器的管理较为“扁平化”，是 Matrix 访问“资源池”的代理，经过测试和理论分析，能有效避免内存不释放的问题。



典型的环形引用导致内存释放 (图源：简书)

[6]实际上，受时间所限，编写代码时所能想到的异常必然远不足以覆盖所有可能；另外，在真实的工程代码中，处理各种可能异常的代码量甚至超过业务代码量。



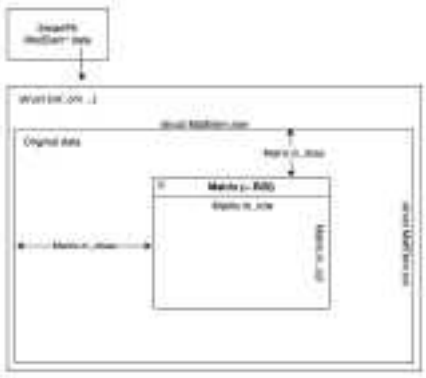
本 project 设计的 SmartPtr 类的行为图示，其中 SP 为 SmartPtr 对象（图源：我）

2.3 Matrix 类

```cpp
template<class T>
class Matrix {
    template<class V>
    friend class Matrix;

    int m_row;    int m_col;          // 当前ROI的大小
    int m_rbias;  int m_cbias;        // 相较于原矩阵，ROI的坐标偏置
    std::vector<SmartPtr<T>> m_data;  // 多个2d-matrix叠加为多通道
    /* public functions */
}  // in namespace mat
```

```cpp
typedef Matrix<unsigned char> MatUC8;    typedef Matrix<short> MatI16;
typedef Matrix<int> MatI32;              typedef Matrix<float> MatF32;
typedef Matrix<double> MatF64;
```

在选择将矩阵真正的数据放入“数据池”并由 SmartPtr 代理后，所有的 Matrix 都可以看作 ROI（而传统意义的原始矩阵即是一个 bias = 0，size = dataSize 的 ROI）。



使用 vector 存多个二维矩阵实现了多通道，而通道数直接通过 m_data.size() 即可获得，无需手动管理。这里 vector 中存的是 SmartPtr 的栈内对象，是为了让其可以自动销毁以此帮助我们减轻一部分内存管理的负担，且对象中仅存了一个指针，即 sizeof(SmartPtr<T>) = sizeof(SmartPtr<T>*)。又因为考虑到现实中通道数一般是远小于单个二维矩阵的元素个数的[7]（以图片为例，一般仅有 RGB 三个通道，而每个通道可能有上百万像素点），可以在每个 Matrix 类中保持 vector 的栈内对象而不必将其也共享给多个 Matrix。这使得了不同矩阵直接可以混用叠加通道，vector 提供的高效扩容机制使我们可以方便的增加/移除通道，而分摊开销较小。

在 SmartPtr 接管了引用计数器后，Matrix 类即可使用默认拷贝构造器、析构函数及赋值运算符[8][9]，这是因为除去四个 int 外，vector 默认是深拷贝的，会调用每个 SmartPtr 的拷贝构造器并正确管理计数器；在 matrix1 = matrix2 时，进行了 matrix1.m_data = matrix2.m_data 的赋值运算符的操作，其首先销毁 matrix2 中所有 SmatrPtr 对象（引用计数器自减），再对 matrix1 中 vector 的所有对象调用拷贝构造器（matrix1 指向的数据引用计数器自增）；Matrix 析构时，销毁 vector 的同时也将其中的所有 SmatrPtr 销毁，析构函数依然能起到管理计数器的作用。

```
1 Matrix(const Matrix<T> &m) = default;
2 Matrix<T> &operator=(const Matrix<T> &op) = default;
3 ~Matrix() noexcept = default;
```

2.3.1 构造器

```
1 Matrix(int r, int c, int ch = 1) noexcept(false)
2         :m_row(r), m_col(c), m_rbias(0), m_cbias(0) {
3     if (r <= 0 || c <= 0 || ch <= 0) throw InvalidArgs(std::initializer_list<int>{r, c, ch});
4     for (int i = 0; i < ch; ++i)
5         m_data.emplace_back(r, c);
6 }
```

形如第 3 行的有效性逻辑检查在后续还会大量出现，而第 5 行用 emplace_back 能将 SmartPtr 构造器所需的参数传递给 vector 并由其代为直接生成，而不用在 push_back 处生成一个临时对象（虽然区别及其细微，且均能保证内存管理的正确性）。如 3.1 节所提，在构造过程中可能出现的内存泄露均在 SmartPtr 的构造函数中被捕获并处理，但异常依然传出以供 C++ runtime system 释放 SmartPtr 及 Matrix 对象本身分

[7]本 project 均是基于此理解而完成的，与 OpenCV 不一致之处敬请谅解。

[8]实际上只是把管理计数器的工作交给 SmartPtr 完成，整体代码量并无减少，但从开发和逻辑的角度看，这样设计依然带来了一定的好处，并使代码解耦连更加方便。

[9]以下结论均是通过大量的实验而得出，但实验过程未做记录，故未在后文体现。

配的栈内存。

```
explicit Matrix(std::fstream &f) noexcept(false): m_rbias(0), m_cbias(0) {
    if (!f.is_open()) throw MatException(strerror(errno));
    m_row = file_rows(f);  m_col = file_cols(f);
    m_data.emplace_back(m_row, m_col);
    auto data = m_data.at(0).m_data->data;
    std::string line;  std::stringstream ss;
    for (int i = 0; i < m_row; ++i) {
        std::getline(f, line, '\n');  ss.clear();  ss << line;  int col_i = 0;
        while (ss >> data[i * m_col + col_i++]);
        if (col_i != m_col+1)
            throw MatException("Invalid file content: column number isn't fixed");
    }
    f.seekg(0L, std::ios_base::beg);
}
```

构造一个单通道矩阵，与 project 2 中的读取类似，要求矩阵元素存入文件时每行用换行符分隔，行内元素用空格分隔。只要矩阵所存元素重载了 operator>> 或为基本数据类型即可使用文件输入（否则无法通过编译，属于使用者的问题）。

```
Matrix(const Matrix<T> &m, int row, int col, int rbias = 0, int cbias = 0) noexcept(false) {
    if (row <= 0 || col <= 0 || rbias < 0 || cbias < 0)
        throw InvalidArgs(std::initializer_list<int>{row, col, rbias, cbias});
    for (auto &ch : m.m_data) {
        if (row + rbias > ch.m_data->row)
            throw MatException("ROI out of bound of the origin matrix: row+rbias > ori_row");
        if (col + cbias > ch.m_data->col)
            throw MatException("ROI out of bound of the origin matrix: col+cbias > ori_col");
    }
    m_row = row;      m_col = col;
    m_rbias = rbias;  m_cbias = cbias;
    m_data = m.m_data;
}
```

当不需要调整 ROI 视图大小时，可以通过 Matrix newMat = oldMat 使用默认拷贝构造器，而需要调整时，经检查参数合法后调节 ROI 参数并拷贝智能指针。

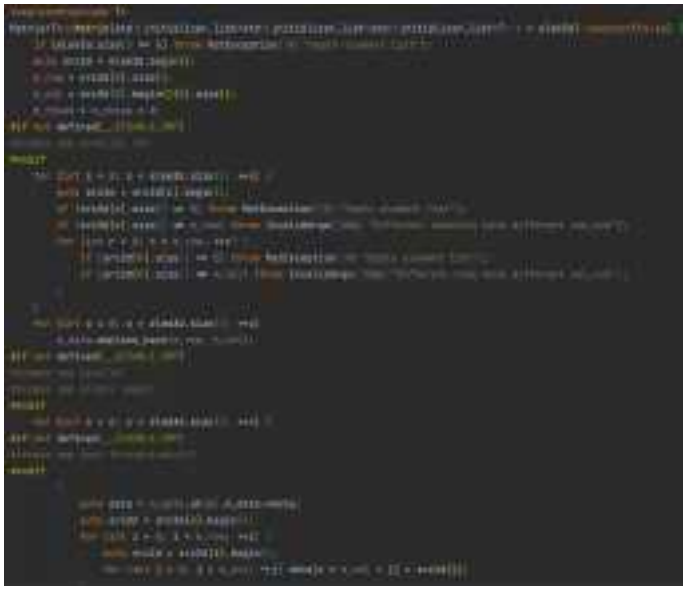
```
template<class V>
explicit Matrix(const Matrix<V> &m) noexcept(false)
        :m_row(m.m_row), m_col(m.m_col), m_rbias(m.m_rbias), m_cbias(m.m_cbias) {
#if not defined(__DISABLE_OMP)
#pragma omp parallel
#pragma omp single nowait
#endif
    for (int i = 0; i < m.m_data.size(); ++i)
#if not defined(__DISABLE_OMP)
```

```
#pragma omp task firstprivate(i)
#endif
        m_data.emplace_back(m.m_data.at(i));
}
```

提供了转换构造器，需要显式类型转换。emplace_back 会传入 SmartPtr<V> 型的引用并判断调用其转换构造器，将所有通道内数据生成一份转换类型后的副本（若两类型间不支持强制转换，将不通过编译）。第 12 行的 SmartPtr 转换构造器需要遍历所有数据，因此较为耗时，而使用 OpenMP 在一个线程中执行循环，但将循环中元素的处理委托给其他线程可以在保证通道顺序不变的条件下同时执行多个通道的转换，这对于多通道矩阵的类型转换能大幅提高效率，而前面的普通创建新矩阵的函数中，由于每个子任务耗时较少，不必使用并行化。

如 *More Effective C++* Item 18 （分期摊还预期的计算成本，超急评估）所述，在类型转换时将 ROI 以外的数据也进行了转换，这是考虑到当有必要进行类型转换时，后面可能多处需要用到同一矩阵的不同 ROI（Strassen 算法便是很好的例子）。

此外，还提供了接受初始化参数列表的构造器，可以用类似 NumPy 的形式直接以值初始化较小的矩阵。



2.3.2 深拷贝

上面的构造器均提供了共享数据的浅拷贝操作，当期望修改一个子矩阵而不影响原始矩阵的数据时，我们需要为其提供一份数据副本。

```
[[nodiscard]] static Matrix clone(const Matrix<T> &tar) {
    int channels = tar.channels();
    Matrix cpy(tar.m_row, tar.m_col, channels);
#pragma omp parallel       // #if not defined(__DISABLE_OMP)
#pragma omp single nowait  // #if not defined(__DISABLE_OMP)
    for (int i = 0; i < channels; ++i) {
#pragma omp task firstprivate(i)  // #if not defined(__DISABLE_OMP)
        {
            auto cpyData = cpy.m_data.at(i).m_data->data;
```

```
            auto srcData = tar.m_data.at(i).m_data->data;
            int srcCol = tar.m_data.at(i).m_data->col;
            for (int r = 0; r < tar.m_row; ++r) {
                memcpy(cpyData + r * tar.m_col,
                       srcData + (r + tar.m_rbias) * srcCol + tar.m_cbias,
                       sizeof(T) * tar.m_col);
            }
        }
    }
    return cpy;
}
```

2.3.3 填充元素

```
void fill(const T &elem) {
    int channels = m_data.size();
#pragma omp parallel       // #if not defined(__DISABLE_OMP)
#pragma omp single nowait  // #if not defined(__DISABLE_OMP)
    for (int i = 0; i < channels; ++i) {
#pragma omp task firstprivate(i)  // #if not defined(__DISABLE_OMP)
        {
            auto data = m_data.at(i).m_data->data;
            int srcCol = m_data.at(i).m_data->col;
            for (int r = 0; r < m_row; ++r) {
#pragma unroll 16
                for (int c = 0; c < m_col; ++c)
                    data[(r + m_rbias) * srcCol + c + m_cbias] = elem;
            }
        }
    }
}
```

使用并行化和循环展开加速，*void fillRand()* 与之类似（仅将第 13 行替换为以下几行），如果是用户自建类型，期望其 static_cast<T> 时能调起合适的函数生成“随机”对象。

```
// marcos TYPE_IS is defined as:
// #define TYPE_IS(to, tar) std::is_same<typename std::decay<to>::type, tar>::type::value
if (TYPE_IS(T, float) || TYPE_IS(T, double))
    data[(r + m_rbias) * srcCol + c + m_cbias]
            = static_cast<T>((static_cast<double>(rand() % 5000) / 160));
else data[(r + m_rbias) * srcCol + c + m_cbias] = static_cast<T>(rand() % 5000);
```

生成随机矩阵可以创建一个矩阵或调用以下函数（本 project 大部分短函数选择直接定义在头文件中，自动成为 inline）：

```
[[nodiscard]] static Matrix randMat(int r, int c, int ch = 1) {
    Matrix res(r, c, ch);
    res.fillRand();
    return res;
}
```

2.3.4 Getters & Setters

提供常用的 getters，并支持插入或移除通道的函数（我们一般不希望用户直接修改通道中的数据，但依然提供获取原始数据指针的函数，极不建议使用），配有异常处理。

2.3.5 重载矩阵运算操作符

对于 SIMD 不支持的数据类型，使用循环展开，且 SIMD 的使用无需在函数内判断，函数的调用可以利用最优匹配。以下 op 将创建一个新矩阵以存放计算结果，op= 将修改左值的原始数据。

```
Matrix operator+(const Matrix &op) const {
    if (m_row != op.m_row || m_col != op.m_col || channels() != op.channels())
        throw MatSizeMismatch(HERE, m_row, op.m_row, m_col, op.m_col,
                              channels(), op.channels(), '+');
    int ch = channels();
    Matrix sum(m_row, m_col, ch);
#pragma omp parallel for  // #if not defined(__DISABLE_OMP)
    for (int i = 0; i < ch; ++i) {
        auto lhsData = m_data.at(i).m_data->data;
        auto rhsData = op.m_data.at(i).m_data->data;
        auto sumData = sum.m_data.at(i).m_data->data;
        if (m_row == m_data.at(i).m_data->row && m_col == m_data.at(i).m_data->col
            && m_row == op.m_data.at(i).m_data->row && m_col == op.m_data.at(i).m_data->col) {
            int cnt = m_row * m_col;
#pragma unroll 16
            for (int i = 0; i < cnt; i++)
                sumData[i] = lhsData[i] + rhsData[i];
        } else {
            for (int r = 0; r < m_row; ++r) {
#pragma unroll 16
                for (int c = 0; c < m_col; ++c) {
                    sumData[r * m_col + c] = lhsData[(r + m_rbias) * m_col + m_cbias + c] +
```

```
                                                rhsData[(r + op.m_rbias) * m_col + op.m_cbias + c];
                }
            }
        }
    }
    return sum;
}

Matrix &operator+=(const Matrix &op) {
    /* similar to operator+(const Matrix &op) */
#pragma unroll 16
            for (int i = 0; i < cnt; i++)
                lhsData[i] += rhsData[i];
    /* similar to operator+(const Matrix &op) */
    return *this;
}

template<typename V>
Matrix operator+(const V &op) const {
    auto oppo = static_cast<T>(op);
    /* similar to operator+(const Matrix &op) */
}

template<typename V>
friend Matrix<T> operator+(const V &lhs, const Matrix<T> &rhs) { return rhs + lhs; }

template<typename V>
Matrix& operator+=(const V &op) { /* similar to operator+=(const Matrix &op) */ }
```

与之类似的还有减法及数乘的系列运算，为节省空间不在报告展示。

在 Lab 8 中，我们实践了加法运算的 SIMD，发现加法的提升效果远不及乘法的数据向量化，分析可能是因为每个乘法的操作所耗费的时间（CPU 时钟）比加法多，也就是说乘法运算是计算耗时占主导的，而加法运算的总耗时中，计算 / *IO* 较小。基于此进行了加法的循环展开和 SIMD 对比，发现耗时接近，因此不必实例化部分数据类型的 SIMD 加减法。

一般的矩阵乘法使用 OpenMP 并行化、ikj 访存优化及循环展开，力求在不支持 SIMD 的情况下提升性能。当 ROI 覆盖完整的原始数据区（或列数相等）时，每行的最后元素与下一行的首个元素在内存上连续，性能优于局部的 ROI。而对于 SIMD 支持的基本数据类型（方便起见，只写了 int、float 和 double 为例），选择实例化对应的模板函数。

```
#if not defined(__DISABLE_SIMD)
template<>
MatF32 MatF32::operator*(const MatF32 &op) const {
    /* matherr checks; the following codes are for single channel (an outer `for` is omitted) */
#if defined (__AVX512F__)
    for (int i = 0; i < m_row; ++i) {
        for (int j = 0; j < op.m_col; ++j) {
            __m512 res = _mm512_setzero_ps();
            for (int k = 0; k < m_col / 16; ++k) {
                res = _mm512_fmadd_ps(_mm512_loadu_ps(lhsData+(i+m_rbias)*lo_col+k*16+m_cbias),
                                      _mm512_loadu_ps(rhsData + j * rhs.m_col + k * 16),
                                      res);
            }
            prodData[i * op.m_col + j] = _mm512_reduce_add_ps(res);
            for (int k = m_col - (m_col % 16); k < m_col; ++k) {
                prodData[i * op.m_col + j] += lhsData[(i + op.m_rbias) * lo_col + k + m_cbias]
                                            * rhsData[j * rhs.m_col + k];
            }
        }
    }
```

```
#elif defined (__AVX2__)
        /* similar to AVX512 */
#elif defined (__ARM_NEON)
    for (int i = 0; i < m_row; ++i) {
        for (int j = 0; j < op.m_col; ++j) {
            float32x4_t res = vdupq_n_f32(.0f);
            for (int k = 0; k < m_col / 4; ++k) {
                res = vmlaq_f32(res,
                                vld1q_f32(lhsData + (i + m_rbias) * lo_col + k * 4 + m_cbias),
                                vld1q_f32(rhsData + j * rhs.m_col + k * 4));
            }
            prodData[i * op.m_col + j] = vaddvq_f32(res);
            for (int k = m_col - (m_col % 4); k < m_col; ++k) {
                prodData[i * op.m_col + j] += lhsData[(i + op.m_rbias) * lo_col + k + m_cbias]
                                            * rhsData[j * rhs.m_col + k];
            }
        }
    }
#else
        throw MatException("SIMD not supported, check the g++/gcc args");
#endif
    }  // the paired #omp parallel for <channels> is omitted
    return prod;
}
#endif
```

其余几种数据类型的代码类似。另外要注意的是 Intel 没有为 epi8 / 16 提供乘法运算函数，可以通过以下代码[10]模拟，但性能自然不如原生指令。

```
inline __m256i _mm256_mul_epi8(__m256i a, __m256i b) {
    __m256i dst_even = _mm256_mullo_epi16(a, b);
    __m256i dst_odd = _mm256_mullo_epi16(_mm256_srli_epi16(a, 8), _mm256_srli_epi16(b, 8));
    return _mm256_or_si256(_mm256_slli_epi16(dst_odd, 8), _mm256_and_si256(dst_even,
        _mm256_set1_epi16(0xFF)));
}
```

[10]参考：stackoverflow.com/questions/8193601/sse-multiplication-16-x-uint8-t 修改而得。

2.3.6 operator==

对于同类型 Matrix（如 Matrix<int> ?= Matrix<int>），会匹配第一个函数，当 ROI 大小一致且（共享同一数据区或每个元素均相同）即判定两矩阵相等；不同类矩阵比较应总认为是不相等的，这时会调用第二个函数。

在重载了 operator== 后，我们不必重载 operator!=，这是因为我们使用的较新版本的编译器能将 (mat1 != mat2) 自动转换为!(mat1 == mat2) 并调用 operator==[11]。

但是考虑到诸如浮点数的精度误差，我们判断两数相等一般是判断两数之差是否在一足够小的“误差范围”内，而 operator== 为二元运算符，将判断数据类型再决定对每个数据使用 == 或 float 式判断也不优雅。这种情况建议使用另一个函数（支持对无法重载 operator== 的数据类型，如 operator==(float, float)，明确指定判等条件以及误差范围）：

```
#include <functional>
bool elemEquals(const Matrix &op, std::function<bool(T &, T &)> const &eq =
                                      [](const T &e1, const T &e2) { return e1 == e2; }) const {
    /* change the code of operator== <line 9>: if (e1 != e2) ~> if (!eq(e1, e2)) */
}
```

2.3.7 矩阵转置、合并通道

[11] 与廖琪梅老师在 Lab 10 课下的讨论得出，若有误还请指正。

不使用并行化同时对多个通道计算是因为考虑到多个通道内相同位置元素会修改同一变量，且其存入目标使用的是 operator+=，多线程竞争数据可能导致脏读脏写。

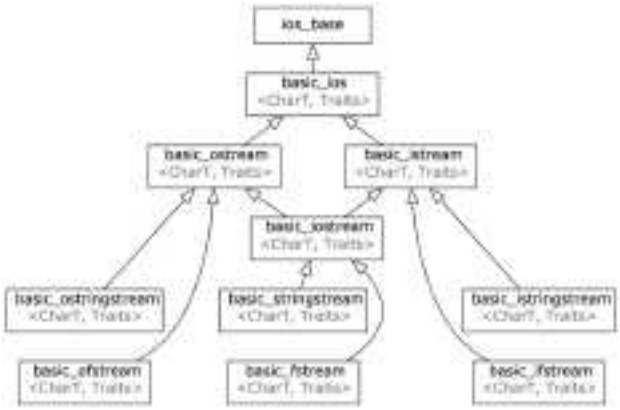
Nov. 28 更新 发现 AVX 指令集有将向量化的数据转换数据类型的功能，重载支持载入寄存器的数据类型的模板函数能进一步提速。

2.3.8 friend operator <<

Project2 中使用 Mat::show() 显示矩阵预览，但当我们想连续打印信息时必须频繁切换 cout 和 show()，代码编写繁琐，且无法指定输出流（写定为 cout），这里我们重载了 << 运算符：

```
template<class V>
friend std::ostream &operator<<(std::ostream &out, const Matrix<V> &mat);

template<class V>
friend std::fstream &operator<<(std::fstream &out, const Matrix<V> &mat);
```

输出到 ostream（如 cout、cerr）的是矩阵的 $6 \times 6 \times$ channel(s) 预览，而输出到文件（fstream）的是矩阵的所有元素值。这里利用了函数的最佳匹配来区分二者。



2.4 MatException 及其派生类

```cpp
class MatException : public std::exception {
protected:
    std::string msg;
    MatException() = default;
public:
    explicit MatException(const char *m) : msg(m) {}
    [[nodiscard]] const char *what() const noexcept override { return msg.c_str(); }
};

class InvalidArgs : public MatException { /* constructors */ };
class MatSizeMismatch : public MatException { /* constructors */ };
class MatIndexOutOfBound : public MatException  { /* constructors */ };
}  // namespace mat
```

2.5 初始化设置

通过取消同步流来细微提升 stdio 的读写性能、将随机数生成种子设为时间。该函数应该在主函数开头调用一次，但 static 的 lambda 表达式保证了即使多次调用该函数也只会总共执行一次。

```cpp
void init() {
    static auto init_once = []() -> int {
        std::ios_base::sync_with_stdio(false);
        std::cin.tie(nullptr), std::cout.tie(nullptr);
        srand(time(nullptr));
        omp_set_num_threads(omp_get_max_threads());  // #if not defined(__DISABLE_OMP)
```

```
        return 0;
    }();
}
```

3 测试样例及分析

在性能测试中使用了 ① MacBook Pro (x86) ② AliYun ECS T6 (x86) 以及学校提供的 ③ Huawei Cloud Server (ARM)。这里所考虑到的影响性能的主要四个方面在于单核算力、总线程数、内存读写性能及高速缓存大小[12]。如下图所示，华为与阿里云服务器分别有 2 线程及 4 线程，而 MacBook 的 8 线程能提供较好的并行计算能力，此外，架构及 SIMD 指令集不同意味着数据向量化的能力不同。

本次“跨平台”注重的是架构（指令集）的差异，而非考察对不同操作系统的适配，方便起见，本次只在 CentOS (ARM: gcc 7.3.0; x86: gcc 8.4.1) 及 macOS (Apple clang 13.0.0) 上进行测试。

(a) 华为服务器 (ARM)

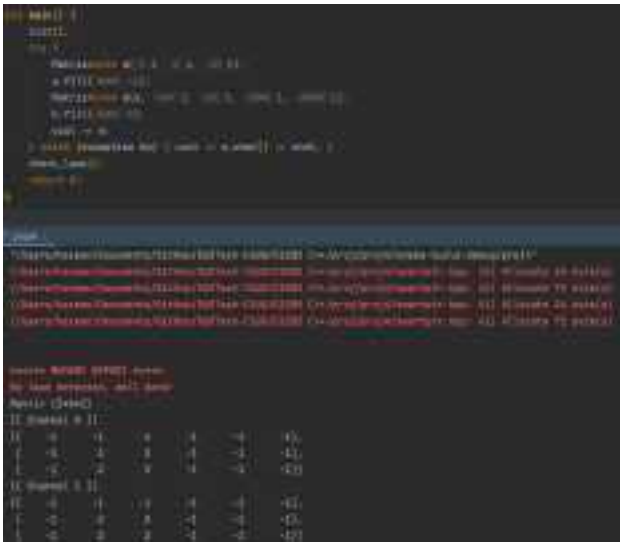
(b) 阿里云 ECS 突发性能 T6 实例 (x86)

(c) MacBook Pro (x86)

(d) 使用 CMake 构建项目，利用预编译宏

[12]对于小矩阵影响不明显；而对于单行所占内存接近缓存的大矩阵，缓存越大，对于访存优化较好的算法越有利。

3.1 正确性验证



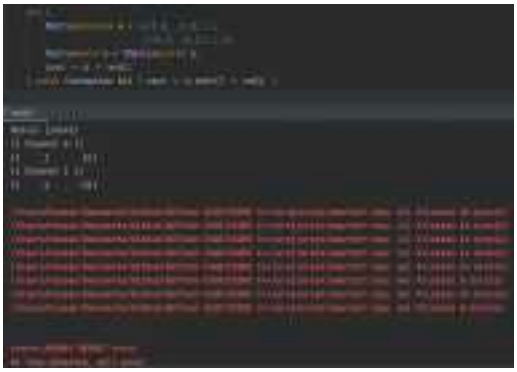
(a) 创建多通道矩阵，ROI 共享原始矩阵数据，memory_report 无内存泄露[13]

(b) ROI 等各项参数检测

(c) 字面值列表初始化矩阵：合法性检查（左）、创建成功（右）

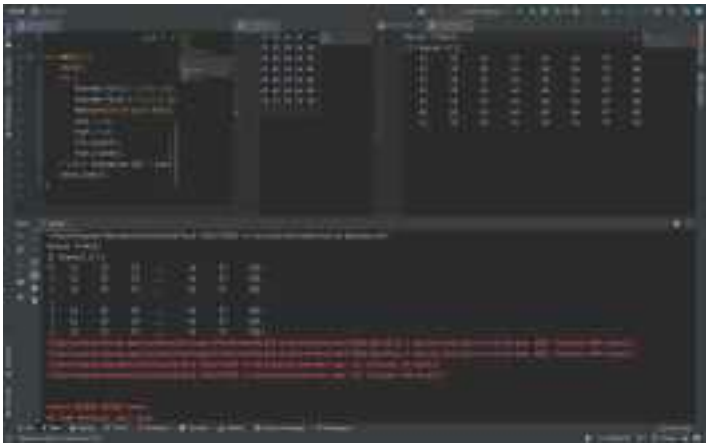
[13]观察 new 记录，两个 24bytes 为保存原始矩阵信息和数据指针的结构体，72bytes 为数据区，均为双通道的 Matrix a 所用；ROI b 不创建新数据（仅拷贝构造 SmartPtr）。

(d) 深拷贝后不影响原数据（clone 后获得数据副本，修改 Matrix b，a 中数据不变）

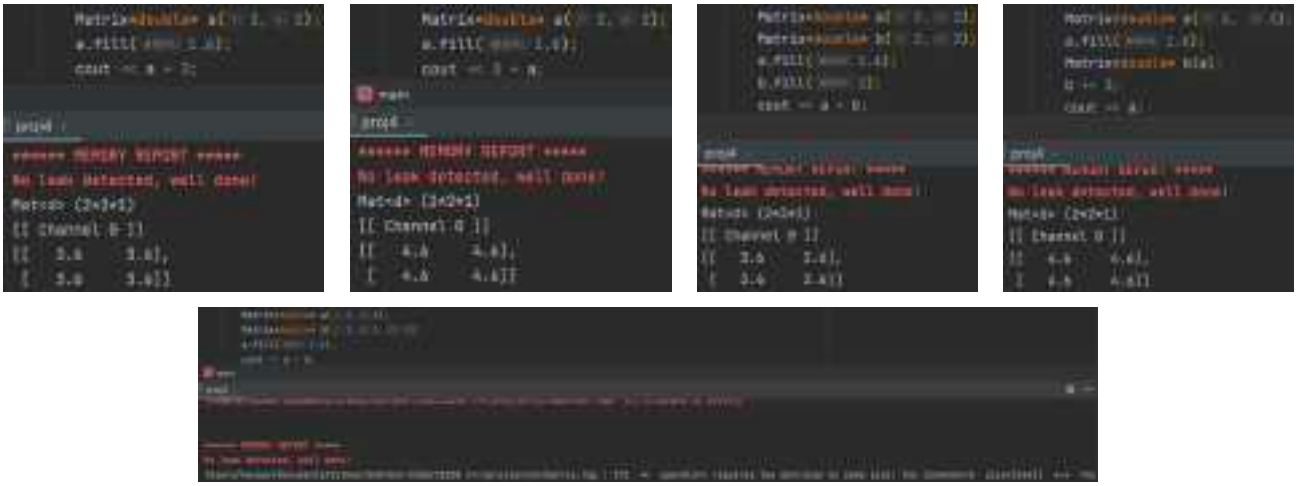


(e) 显式数据类型转换（原矩阵每通道数据区 16bytes，新矩阵每通道 8bytes）

(f) 文件输入：正常（左），文件异常（中），数据异常（右）

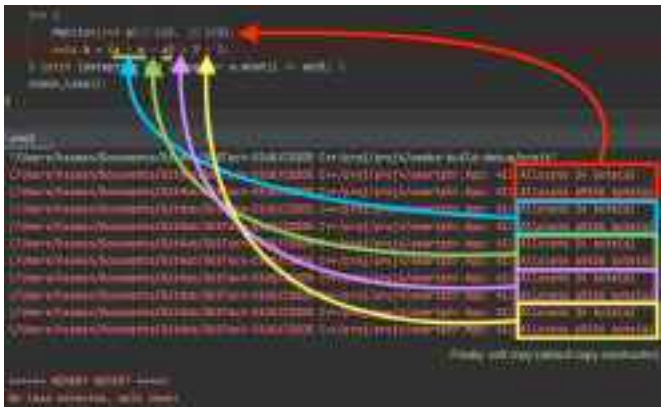


(g) 文件输出（完整模式，带有矩阵 Metadata，删除后可供读入文件）与控制台输出（预览模式）



(h) 加法系列操作，减法及乘法系列与之类似

(i) 矩阵相等的两种判断函数



(j) 复杂运算中的局部变量不会导致内存泄露

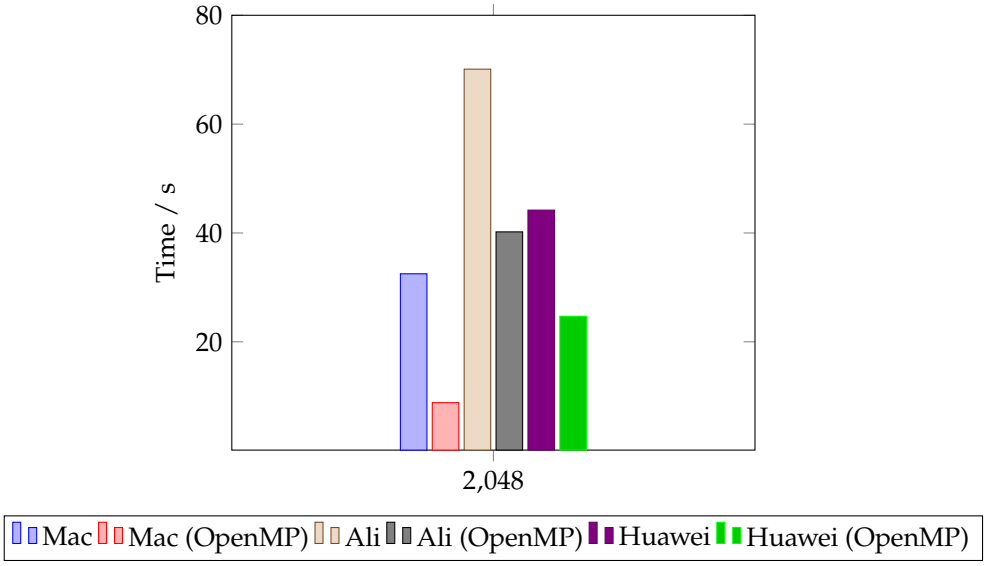
上图中虽然 SmartPtr 的栈内变量能自动释放数据区指针，但我们注意到所有运算中每步依然需要申请内存（系统级操作耗时较多！），*More Effective C++* Item 17 （考虑使用 lazy evaluation）则指出表达式的值不必直接算出，直到必须取值时才做相应计算。这启示我们，对于计算式复杂或矩阵往往只需要其小部分元素的值的应用场景，可以调整 Matrix 类的数据结构（比如保存两个 Matrix 指针和一个运算符 enum，或用栈记录当前矩阵等价于哪些矩阵的运算结果）来缓式计算表达式[14] 。

[14]本 project 中未能实现，但分析这样做至少能带来的好处是，在如上例子中，只需为第二条计算申请一个矩阵的内存

3.2 性能测试

使用 randMat 生成 2048 * 2048 大小的随机矩阵并保存至文件，手动删去文件头处的“Metadata”即可供读入并初始化为单通道矩阵。

operator* / Mat(2048 * 2048)	Plain (UC8)	OpenMP (UC8)	Plain (F32)	OpenMP (F32)
MacBook (x86, 8 threads)	32.840	8.563	33.281	9.075
Huawei (ARM, 2 threads)	66.004	38.587	75.212	43.047
Ali ECS (x86, 4 threads)	42.760	23.907	46.067	26.775



上图对 unsigned char 和 float 的乘法取平均，由图可见，三个测试环境的硬件不一导致基准性能不同，但在使用 OpenMP 后，4 核 8 线程的 Mac 性能提升约 4 倍，2 核 4 线程的阿里云和 2 核 2 线程的华为云服务器性能均提升 2 倍左右，这是由于“*在运行时，默认线程数通常等于操作系统看到的逻辑 CPU 核心数*”[15]，(但此时使用 top 命令查看 CPU 使用率，均显示每个核心使用约 100%，即使手动设置 num_threads 为总线程数也没能进一步带来性能提升，望老师指点)。

[15]https://support.huaweicloud.com/tuningtip-kunpenggrf/kunpengtuning_12_0086.html

operator* / Mat(2048 * 2048)	Plain (F32)	AVX2 (F32)	AVX512 (F32)	NEON (F32)
MacBook	33.281	7.003	3.807	-
Ali ECS	46.067	9.200	5.131	-
Huawei	75.212	-	-	26.854

上表中对 float 型 2048 阶矩阵乘法测速[16]，AVX2 将 8 个 float 打包为向量，但由于乘法运算中首先进行了一次暴力转置有部分耗时，最终将性能提升了将近 4.8 倍；AVX512 将 16 个 float 打包，性能约为 AVX2 下的 1.8 倍，排除没有加速的转置部分的时间，计算性能约为 AVX2 的两倍（印证了 project3 中得出的结论）。而 ARM 下将 4 个 float 打包进 128-bit 的寄存器，性能提升至 2.8 倍，理由同上。

事实上，本次 project 仅将 AVX 中的寄存器类型换为 NEON 中的，并没有很好的利用 ARM “寄存器小 (128-bit) 而多 (37 个)” 的特性，以及其支持的多寄存器 load / store 等操作。合理的调整代码行为，能更好的利用 ARM 架构的设计特性。

[16]ARM 架构的 128 位寄存器似乎不支持对 double 向量化（没有找到合适的函数）；AVX 系原生没有对 epi8 (unsigned char / char) 和 epi16 (short) 的乘法函数，即使是自己实现一个类似函数，效果也不及原生指令，故参考意义不大；int 和 float 虽然在计算方面可能略有差异（CPU 时钟周期），但矩阵乘法耗时依然是 IO 主导的，时间关系没有进一步对比测试。

	CISC
长 ②指令单周期执行 ③无 ，通过简单指令的 组合实现 ⑤指令译码采用硬布线逻辑	①指令变 期 ③有 成 复杂
流水线执行，流水线每周期	不易于实 行需要设
寄存器	用于特定
oad和Store指令完成数据在寄 部存储器之间的传输	处理器能 据
要求高，需要编译器对代码	对编译器

4 困难及解决

4.1 避免构造与析构阶段的内存泄露

当我们用 new 关键字来创建对象时，如果构造函数抛出异常，C++ 运行时系统会自动调用 delete 运算符（而非 operator delete，C++ 拒绝为部分构造的对象执行析构操作!），而此时我们只释放了分配给实例化对象需要的内存，而构造函数中可能已经进行了一些内存的动态分配，这时为了避免内存泄露，一种常用的传统方法是捕获所有的异常，然后执行一些清除代码，最后再重新抛出异常让其继续传递。

(a) 构造函数抛出异常可能导致内存泄露

(b) 构造函数内处理可能的内存泄露

一种更简便的方法是使用智能指针，本 project 设计的 SmartPtr 也起相同作用。

More Effective C++ Item 11 则提到，我们应该禁止异常流出 destructors 之外。析构函数除了程序员手动调用外，还有可能被异常处理机制（栈展开，stack-unwinding）所调用。而当析构函数因为这种原因被调用时，已经有一个异常处于作用状态中，假如此时析构函数再次抛出一个异常（“控制权基于 exception 的因素离开 destructor”），C++ 会调用 *terminate* 函数直接结束程序—— “甚至不等局部变量销毁”，这可能会造成极为显著的内存泄露；另一方面，抛出异常意味着析构函数是执行不全的。基于这两点原因，我们必须确保析构函数不会传出异常，所幸，一个简单的 *try-catch(...)* 块的包围即可以解决大部分问题。

在研究此方面问题时，我再次意识到将释放后的指针设为 nullptr 的重要性（并非为了回避重复释放! 该部分在未将指针置空时已经通过了周密的正确性检查）。测试发现，使用 try-catch 块包裹 delete 操作依然不能处理重复释放内存 (pointer being freed was not allocated) 的错误，网上没有找到相关解释，但猜测是因为 delete 与 new 同为系统级的操作，出错时并不引发 C++ 抛出异常，而是直接杀死程序 (interrupted

by signal 6: SIGABRT)。但我们知道 delete 一个 nullptr 是合法的操作，因此在上述代码的 catch 块中，我们尝试去释放完部分构造的对象所申请的内存后最好都将其设为 nullptr。

4.2 模版类友元函数的链接问题

用 lab 课上的常规方法为模版类声明友元函数时，链接器提示符号未定义，而解决方案[17]是为模版类 <T> 中的友元声明添加一句 *template <class U>*。

```
template<class U>
friend std::ostream &operator<<(std::ostream &out, const Mat<U> &mat);
```

这是由于我们在定义该友元函数时，会自然的加上 template 使之成为模版函数，而不带 template 的（类内）声明使链接器认为这是一个非模版函数（例如，在实例化一个 int 型的模版类时，链接器期望在其他代码中找到一个固定参数的非模版函数，其所有类型 T 被替换为 int，仅此而已），而加上 template <class T> 的标识提示链接器去寻找一个模版函数，且其所有实例化都是对应的类的友元。当友元函数较短小时，直接在类内定义则可以避免上述问题。

在为模板类提供转换构造函数时也遇到了类似的问题，即对于模板类 Demo<T> 而言，无法访问 Demo<V> 的私有成员变量，这可以通过 *template<class V> friend class Demo* 解决。

4.3 类的继承

对于语法 class <derived-class>: <access-specifier> <base-class>，一开始我不清楚 access-specifier 的作用，发现在 catch 块中总是无法正常捕获抛出的自定义异常。原先想用虚函数，但后来发现没必要也没有地方可写。

[17]Link error when using friend function in template linkedlist (stackoverflow.com/questions/8545495/link-error-when-using-friend-function-in-template-linkedlist)

图源：菜鸟教程 (www.runoob.com/cplusplus/cpp-inheritance.html)

4.4 OpenMP 并行 for 下抛出异常

在诱发抛出异常测试时，发现 OpenMP 块内的抛出的异常无法正常被主调函数处的 try-catch 块捕获。这是由于"A throw executed inside a parallel region must cause execution to resume within the same parallel region, and the same thread that threw the exception must catch it."[18]。一般的解决方案是在 task 内即解决所有的异常，但这无法达到我们想将所有异常传回主调处的目的。一种可能的方法是在 omp task 内捕获所有异常并写入 flag，task 块外依据 flag 判断是否需要抛出异常[19]。

```
bool raiseExcept = false;
std::string msg;
#pragma omp task <args>
{
    try { /* ... */ } catch (const std::exception& e) { raiseExcept = true;  msg = e.what(); }
}
if(raiseExcept) throw MatException(msg);
```

5 总结

本次 project 意味着我们正式开始了面向对象部分的部分，为了完成此次 project 我必须时刻提醒自己检查各种可能性。此外，本次 project 促使我查阅多本书籍及资料以更深入的了解 C++ vector STL、内存管理等机制、OpenMP 以及 SIMD，阅读书中凝练的知识点与利用新学到的知识亲手实现一些功能都使得这次 project 变得有趣。但随着了解的新知识越多，我越意识到 C++ 作为一个“语言联邦”的复杂深奥。

“ARM 的优势不在于性能强大而在于效率，ARM 采用 RISC 流水线指令集，在完成综合性工作方面根本就处于劣势，而在一些任务相对固定的应用场合其优势就能发挥得淋漓尽致。”[20]这句话可以理解为 ARM 计算机在易用性上对程序员不太友好，但对于由有足够经验等专业程序员开发的极致优化的程序来说，ARM 无疑是权衡所需性能和资金等投入的最佳选择。在对 ARM 和 x86 架构进行对比后，我对上文讨论的高级编程语言所不会接触到的底层也有了更深的理解，也开拓了 C++ 充分利用机器性能的角度。

[18]OpenMP 5.1 specification; What is the correct way for exception handling with OpenMP tasks? (StackOverflow)
[19]测试出此问题时考虑到时间关系，不再调整原有代码。
[20]鲲鹏服务器之 ARM 探知 (https://bbs.huaweicloud.com/forum/thread-64881-1-1.html)