CS205 Project 4: A Class for Matrices

Name: Yankai Xu

SID: 12011525

Part 1 - Analysis

Introduction

This project is to design a class for matrices. The class should support different data types, basic operators and ROI.

To support ROI, two class are included: `Matrix` and `MatrixImpl`.

Classes

The class Matrix is the interface for operations, and holds pointers of MatrixImpl. These pointers are held by `std::shared_ptr` to avoid possible memory leak.

The class MatrixImpl hold the data of the Matrix.

template

To support different data types, class templates are widely used in both classes. Template T decides the data type stored in the matrix.

Matrices with different data types can do arithmetic operations together, as long as the data types can do the operation. This feature is implemented by `decltype` in compile time, and code inspectors in ide like clion support it well.

Matrices with different data types can also convert explicitly by copy constructor. This feature is implemented by doing static_cast to all element in matrix.

When doing matrices comparisons, the comparing of double and float need extra care. Template metaprogramming are used to define two `isEqual()` functions, if float or double appears in arguments, the function for comparing float is called. Otherwise, a simple function which returns the result of == is called to save time.

Channel

If a matrix has n channels, it owns n MatrixImpl pointers.

ROI

The semantic of ROI is similar to opencv. class Matrix have an overloaded function call operator to get another Matrix instance, which shares the same MatrixImpl with it's "father". The difference between these two Matrices is that there number of rows and cols are different, and the ROI matrix has an offset point to decide where to start in the matrix.

Constructors

Class Matrix has multiple useful constructors:

```
explicit Matrix(const size_t &rows = 0, const size_t &cols = 0, const
size_t channels = 1, const T &val = T());
```
Create a matrix with these parameters.

```
Matrix(const Matrix &rhs);
```
Copy all content from rhs to this. This construct(and copy assignment operator) are operating differently than the project required. It copies the whole matrix, not make two matrices share the same pointer. I didn't follow the project instruction there because all stl containers are copying there content when copy constructor and copy assignment are called. And we have pointer and reference to avoid copy, rvalue reference to transfer ownership. So, this constructor is copying the content of the matrix.
Copying an ROI will only copy the data inside the ROI region.

```
Matrix(Matrix &&rhs) = default;
```
Default move constructor. As std::shared_ptr is used in the matrix, default move constructor works well. Also, only matrices with same data type and channels can be moved.
Moving an ROI will do whatever default move constructors do.

```
explicit Matrix(const Matrix<U> &rhs);
```
convert data type U matrix to data type T matrix.

```
Matrix(const Matrix &other, const Range &row, const Range &col) :
pimpl{other.pimpl}, rows(row.size());
```
Create a ROI matrix based on matrix other. Same as operator().

Destructor

As Matrix's all members are either builtin type or stl class, default destructor works well too.

Operators

```
Matrix operator()(const Range &row, const Range &col);
```
Get ROI based on matrix. Range is similar to cv::Range.

```
std::vector<std::reference_wrapper<T>> &operator()(const size_t &row, const
size_t &col);
std::vector<T> operator()(const size_t &row, const size_t &col);
T &operator()(const size_t row, const size_t col, const size_t channel);
```

`const T &operator()(const size_t row, const size_t col, const size_t channel);` Get elements in row, col.

`Matrix<T> &operator=(const Matrix<T> &rhs);`

Copy assignment. Same as copy constructor mentioned before, and used Scott Meyers' advice to handle self assign problem. Besides, assign matrices with different data types is not allowed.

`Matrix<T> &operator=(Matrix<T> &&rhs) noexcept = default;`

Default move assignment. Works like move constructor.

`template<typename U>`
`bool operator==(const Matrix<U, RHS_CHANNEL> &rhs)`
Compare whether two matrices are equal. Two equal matrices must satisfy:

- Numbers of channels are same.
- Sizes are same.
- every value in lhs and rhs is equal.

In addition, two equal matrices don't appear to have the same data type.
Also, there are two special judges when two matrices have the same type. If lhs and rhs are the same matrix or are the roi of the same area of the same matrix, this function will return true immediately rather than compare values.

`operator +(), +(Matrix), +=(Matrix), -(), -(Matrix), -=(Matrix), *(U), *(Matrix), *=(U)`
Basic operators. As class Matrix is using template, SIMD isn't an easy choice. So, only openmp are used to accelerate.
Besides, compound assignment operators and scalar multiplication accept different data type, but doesn't change the type of the matrix. For example, `Matrix<int> a += Matrix<double>`, a is still `Matrix<int>`. That's different from built-in types.

`operator << and >>`
Operators for `std::basicstream`. If a matrix has multiple channels, the input should be rows * (cols * channels) matrix.

Friends and other

Friend classes

Both Matrix and MatrixImpl have their template class as friends. For example, Matrix and Matrix are friend classes, and MatrixImpl and Matrix are friends.

Function

Function swap are used to solve self assign problem. It's also almost a copy of Scott Meyers' advice.

Range check

Matrix will check every operation only in debug build. In release build, NDEBUG will be defined and no checking would be performed.

Part 2 - Code

```
// Matrix.hpp
#pragma once

#include "MatrixImpl.hpp"
#include <memory>
#include <iostream>
#include <sstream>

template<typename T>
class Matrix {
private:
    std::vector<std::shared_ptr<MatrixImpl<T>>> pImpls;
    size_t rows, cols, channels, offsetX = 0, offsetY = 0;

    template<typename U>
    bool isValueEqual(const Matrix<U> &rhs) const {
        if (channels != rhs.channels) return false;
        if (rows != rhs.rows || cols != rhs.cols) return false;
        for (size_t i = 0; i < channels; i++) {
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                if (!isEqual(pImpls[i]->base[j], rhs.pImpls[i]->base[j]))
            }
        }
        return true;
    }

    template<typename U>
    void copyImpl(const Matrix<U> &rhs) {
        for (size_t i = 0; i < channels; i++) {
            pImpls[i] = std::make_shared<MatrixImpl<T>>(*rhs.pImpls[i],
                                                        Range{offsetY, of
        }
    }

    // static methods for operator >>
    static void pushToImplBase(std::shared_ptr<MatrixImpl<T>> &impl, T &v
        impl->base.push_back(val);
    }

    static void setImplSize(std::shared_ptr<MatrixImpl<T>> &impl, size_t
        impl->rows = rows;
        impl->cols = cols;
    }

public:
    Matrix(const Matrix &rhs) : pImpls(rhs.channels), rows{rhs.rows},
                                cols{rhs.cols}, channels{rhs.channels}, o
        copyImpl(rhs);
    }

    template<typename U>
    explicit Matrix(const Matrix<U> &rhs) : pImpls(rhs.channels), rows{rh
                                            cols{rhs.cols}, channels{rhs.
                                            offsetY{rhs.offsetY} {
        copyImpl(rhs);
    }

    Matrix(Matrix<T> &&rhs) noexcept = default;

    explicit Matrix(const size_t &rows = 0, const size_t &cols = 0, const
            : pImpls(channels), rows{rows}, cols{cols}, channels{channels
        for (auto &v: pImpls) {
            v = std::make_shared<MatrixImpl<T>>(rows, cols, val);
        }
    }

    Matrix(const Matrix &other, const Range &row, const Range &col)
            : pImpls(other.pImpls), rows(row.size()), cols(col.size()), cl
              offsetY(col.start) {
#ifndef NDEBUG
        if (rows + offsetX > pImpls[0]->getNumberOfRows() || cols + offse
            throw std::invalid_argument("Invalid Range");
#endif
    }

    Matrix<T> &operator=(const Matrix<T> &rhs) {
        Matrix<T> temp(rhs);
        swap(temp);
        return *this;
    }

    Matrix<T> &operator=(Matrix<T> &&rhs) noexcept = default;

    std::vector<std::reference_wrapper<T>> operator()(const size_t &row,
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("inva
#endif
        std::vector<std::reference_wrapper<T>> res;
        for (std::shared_ptr<MatrixImpl<T>> &v: pImpls) {
            res.push_back(v->operator()(offsetX + row, offsetY + col));
        }
        return res;
    }

    std::vector<T> operator()(const size_t &row, const size_t &col) const
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("inva
#endif
        std::vector<T> res(channels);
        for (size_t i = 0; i < channels; i++) {
            res[i] = pImpls[i]->operator()(offsetX + row, offsetY + col);
        }
        return res;
    }

    T &operator()(const size_t row, const size_t col, const size_t channe
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("inva
#endif
        return (*pImpls[channel])(row + offsetX, col + offsetY);
    }

    const T &operator()(const size_t row, const size_t col, const size_t 
#ifndef NDEBUG
        if (row >= rows || col >= cols) throw std::invalid_argument("inva
#endif
        return (*pImpls[channel])(row + offsetX, col + offsetY);
    }

    Matrix operator()(const Range &row, const Range &col) const {
        return Matrix(*this, row, col);
    }

    bool operator==(const Matrix &rhs) const {
        if (this == &rhs) return true;
        if (channels != rhs.channels) return false;
        bool pImplFlag = true;
        for (size_t i = 0; i < channels; i++) {
            if (pImpls[i] != rhs.pImpls[i]) {
                pImplFlag = false;
                break;
            }
        }
        if (pImplFlag && rows == rhs.rows && cols == rhs.cols && offsetX 
            return true;
        return isValueEqual(rhs);
    }

    template<typename U>
    bool operator==(const Matrix<U> &rhs) const {
        return isValueEqual(rhs);
    }

    template<typename U>
    auto operator+(const Matrix<U> &rhs) const {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argu
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        typedef decltype(T() + U()) return_type;
        Matrix<return_type> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
```

```
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = pImpls[i]->base[j] + rhs.pImpls[
            }
        }

        return res;
    }

    template<typename U>
    Matrix<T> &operator+=(const Matrix<U> &rhs) {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argu
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                pImpls[i]->base[j] += rhs.pImpls[i]->base[j];
            }
        }
        return *this;
    }

    Matrix<T> &operator+() {
        return *this;
    }

    template<typename U>
    auto operator-(const Matrix<U> &rhs) const {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argu
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        typedef decltype(T() - U()) return_type;
        Matrix<return_type> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = pImpls[i]->base[j] - rhs.pImpls[
            }
        }
        return res;
    }

    template<typename U>
    Matrix<T> &operator-=(const Matrix<U> &rhs) {
#ifndef NDEBUG
        if (rows != rhs.rows || cols != rhs.cols) throw std::invalid_argu
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
```

```
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                pImpls[i]->base[j] -= rhs.pImpls[i]->base[j];
            }
        }
        return *this;
    }

    Matrix<T> operator-() const {
        Matrix<T> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = -pImpls[i]->base[j];
            }
        }
        return res;
    }

    template<typename U>
    Matrix<T> operator*(const U &rhs) const {
        Matrix<T> res(rows, cols, channels);
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                res.pImpls[i]->base[j] = pImpls[i]->base[j] * rhs;
            }
        }
        return res;
    }

    template<typename U>
    auto operator*(const Matrix<U> &rhs) const {
#ifndef NDEBUG
        if (rows != rhs.cols) throw std::invalid_argument("A.rows != B.co
        if (channels != rhs.channels)
            throw std::invalid_argument("A.channel != B.channel");
#endif
        typedef decltype(T() + U()) return_type;
        Matrix<return_type> res(rows, rhs.cols, channels);
        for (size_t c = 0; c < channels; c++) {
#pragma omp parallel for
            for (size_t i = 0; i < rows; i++) {
                for (size_t k = 0; k < cols; k++) {
                    const T &tmp = (*this)(i, k, c);
                    for (size_t j = 0; j < rhs.cols; j++) {
                        res(i, j, c) += tmp * rhs(k, j, c);
                    }
                }
            }
        }
        return res;
    }
    template<typename U>
    Matrix<T> &operator*=(const U &rhs) {
        for (size_t i = 0; i < channels; i++) {
#pragma omp parallel for
            for (size_t j = 0; j < pImpls[i]->base.size(); j++) {
                pImpls[i]->base[j] *= rhs;
            }
        }
        return *this;
    }

    void swap(Matrix<T> &other) {
        using std::swap;
        pImpls.swap(other.pImpls);
        swap(rows, other.rows);
        swap(cols, other.cols);
        swap(offsetY, other.offsetY);
        swap(offsetX, other.offsetX);
        swap(channels, other.channels);
    }

    friend std::istream &operator>>(std::istream &in, Matrix &mat) {
        for (size_t i = 0; i < mat.channels; i++) {
            mat.pImpls[i] = std::make_shared<MatrixImpl<T>>();
        }
#ifndef NDEBUG
        bool first = true;
#endif
        size_t rowCount = 0;
        for (std::string line; std::getline(in, line);) {
            std::istringstream iss(line);
            size_t channel = 0;
            size_t colCount = 0;
            rowCount++;
            for (T num; iss >> num;) {
                pushToImplBase(mat.pImpls[channel++], num);
                channel %= mat.channels;
                colCount++;
            }
#ifndef NDEBUG
            if (channel != 0) {
                throw std::invalid_argument("wrong channel numbers.");
            }
            if (!first && mat.cols != colCount / mat.channels) {
                throw std::invalid_argument("wrong input.");
            }
            first = false;
#endif
            for (size_t i = 0; i < mat.channels; i++) {
                setImplSize(mat.pImpls[i], rowCount, colCount / mat.chann
            }
            mat.cols = colCount / mat.channels;
        }
```

```cpp
        mat.rows = rowCount;
        mat.offsetY = 0;
        mat.offsetX = 0;
        return in;
    }

    friend std::ostream &operator<<(std::ostream &out, const Matrix<T> &m
        for (size_t i = 0; i < mat.rows; i++) {
            for (size_t j = 0; j < mat.cols; j++) {
                for (size_t k = 0; k < mat.channels; k++) {
                    out << mat(i, j, k) << " ";
                }
            }
            out << std::endl;
        }
        return out;
    }

    template<typename> friend
    class Matrix;

    template<typename> friend
    class MatrixImpl;
};

template<typename T>
void swap(Matrix<T> &a, Matrix<T> &b) {
    a.swap(b);
}



// MatrixImpl.hpp
#pragma once

#include <vector>
#include <stdexcept>
#include <type_traits>

template<typename L, typename R>
std::enable_if_t<std::disjunction_v<std::is_floating_point<L>, std::is_fl
isEqual(const L &lhs, const R &rhs) {
    return std::abs(lhs - rhs) <=
           (((std::abs(rhs) < std::abs(lhs)) ? std::abs(rhs) : std::abs(l
}

template<typename L, typename R>
std::enable_if_t<!std::disjunction_v<std::is_floating_point<L>, std::is_f
isEqual(const L &lhs, const R &rhs) {
    return lhs == rhs;
}

struct Range {
    size_t start, end;

    Range(size_t start, size_t end) : start(start), end(end) {
#ifndef NDEBUG
        if (start > end) throw std::invalid_argument("start > end");
#endif
    }

    [[nodiscard]] size_t size() const {
        return end - start;
    }
};

template<typename T>
class MatrixImpl {
private:
    size_t rows, cols;
    std::vector<T> base;

public:

    template<typename U>
    explicit MatrixImpl(const MatrixImpl<U> &rhs, const Range row, const
            : rows{row.size()}, cols{col.size()}, base(row.size() * col.s
        for (size_t i = 0; i < base.size(); i++) {
            base[i] = static_cast<T>(rhs.base[i]);
        }
    }

    explicit MatrixImpl(const size_t &row = 0, const size_t &col = 0, con
            : rows{row}, cols{col}, base(row * col, val) {}

     [[nodiscard]] size_t getNumberOfRows() const {
        return rows;
     }

    [[nodiscard]] size_t getNumberOfCols() const {
        return cols;
    }

    T &operator()(const size_t row, const size_t col) {
        return base[row * cols + col];
    }

    T operator()(const size_t row, const size_t col) const {
        return base[row * cols + col];
    }

    template<typename> friend
    class MatrixImpl;

    template<typename> friend
    class Matrix;
};
```

Part 3 - Result & Verification

Test case #1:

Test basic operators for same data type.

```
Matrix<float> a{ 2, 2, 3, 0.7f }, b{ 2, 2, 3, 1.6f };
Matrix<float> c{ a }, d{ a }, e{ a };
Matrix<float> f{ 2, 2, 2, 0.7f };

std::cout << (a + b) << std::endl;
std::cout << (a - b) << std::endl;
std::cout << -a << std::endl;
std::cout << (a * b) << std::endl;
std::cout << (a * 0.1f) << std::endl;
std::cout << (a == a) << " " << (a == f) << std::endl << std::endl;

c += b;
d -= b;
e *= 0.1f;
std::cout << c << std::endl;
std::cout << d << std::endl;
std::cout << e << std::endl;
```

Results:

```
2.3 2.3 2.3 2.3 2.3 2.3
2.3 2.3 2.3 2.3 2.3 2.3

-0.9 -0.9 -0.9 -0.9 -0.9 -0.9
-0.9 -0.9 -0.9 -0.9 -0.9 -0.9

-0.7 -0.7 -0.7 -0.7 -0.7 -0.7
-0.7 -0.7 -0.7 -0.7 -0.7 -0.7

2.24 2.24 2.24 2.24 2.24 2.24
2.24 2.24 2.24 2.24 2.24 2.24

0.07 0.07 0.07 0.07 0.07 0.07
0.07 0.07 0.07 0.07 0.07 0.07

1 0

2.3 2.3 2.3 2.3 2.3 2.3
2.3 2.3 2.3 2.3 2.3 2.3
```

```
-0.9 -0.9 -0.9 -0.9 -0.9 -0.9
-0.9 -0.9 -0.9 -0.9 -0.9 -0.9

0.07 0.07 0.07 0.07 0.07 0.07
0.07 0.07 0.07 0.07 0.07 0.07
```

Test case #2:

Test basic operators for different data type.

```
    Matrix<int> a{ 2, 2, 3, 2 };
    Matrix<double> b{ 2, 2, 3, 1.6 };
    Matrix<int> c{ a }, d{ a }, e{ a };
    Matrix<double> f{ 2, 2, 3, 2.0 };

    std::cout << (a + b) << std::endl;
    std::cout << (a - b) << std::endl;
    std::cout << -a << std::endl;
    std::cout << (a * b) << std::endl;
    std::cout << (a * 0.1f) << std::endl;
    std::cout << (f == a) << std::endl << std::endl;

    c += b;
    d -= b;
    e *= 0.1f;
    std::cout << c << std::endl;
    std::cout << d << std::endl;
    std::cout << e << std::endl;
```

Results:

```
3.6 3.6 3.6 3.6 3.6 3.6
3.6 3.6 3.6 3.6 3.6 3.6

0.4 0.4 0.4 0.4 0.4 0.4
0.4 0.4 0.4 0.4 0.4 0.4

-2 -2 -2 -2 -2 -2
-2 -2 -2 -2 -2 -2

6.4 6.4 6.4 6.4 6.4 6.4
6.4 6.4 6.4 6.4 6.4 6.4

0 0 0 0 0 0
0 0 0 0 0 0

1

3 3 3 3 3 3
3 3 3 3 3 3
```

```
0 0 0 0 0 0
0 0 0 0 0 0

0 0 0 0 0 0
0 0 0 0 0 0
```

Test case #3: Test constructors, assignment operators and swap.

```
    Matrix<double> a{ 1, 1, 3, 2.1 };
    Matrix<double> b{ a };
    Matrix<int> c{ a };
    b(0, 0, 0) = 1.1;
    std::cout << "a: " << a << std::endl;
    std::cout << "b: " << b << std::endl;
    std::cout << "c: " << c << std::endl;
    Matrix<double> d{ std::move(a) };
    std::cout << "d: " << d << std::endl;

    Matrix<double> e{ 1, 1, 3, 2 }, f, g;
    Matrix<int> h;
    f = e;
    f(0, 0, 0) = 0.5;

    std::cout << "e: " << e << std::endl;
    std::cout << "f: " << f << std::endl;
    g = std::move(e);
    std::cout << "g: " << g << std::endl;
//    h = e; Error!

    Matrix<int> i{ 1, 1, 3, 1 }, j{ 1, 1, 3, 2 };
    using std::swap;
    swap(i, j);
    std::cout << "i: " << i << std::endl;
    std::cout << "j: " << j << std::endl;
```

Result:

```
a: 2.1 2.1 2.1

b: 1.1 2.1 2.1

c: 2 2 2

d: 2.1 2.1 2.1

e: 2 2 2

f: 0.5 2 2

g: 2 2 2
```

```
i: 2 2 2

j: 1 1 1
```

Test case #4:

Test ROI.

```
    Matrix<int> a(2, 2, 1, 1);
    Matrix<int> b = a({ 0, 1 }, { 0, 1 });
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    b(0, 0)[0] = 2;
    std::cout << a << std::endl;
    std::cout << b << std::endl;
```

Results:

```
1 1
1 1

1

2 1
1 1

2
```

All tests above are proved to be the right answer.

Test case #5:

Test 1 channel 2048 * 2048 double matrix multiplication on X86 and ARM platforms.

Result on X86:

```
366.911ms
```

Result on ARM:

```
6597.2ms
```

The program runs slower on ARM server. The difference may come from:

- The cpu on ARM server only have 2 cores, and need to share computing power with other users. So, the thread number is a lot smaller than my cpu with 8 cores. And the calculation

time for each thread may be longer, too.
- When searching on the assembly code generated by g++ on ARM server, there is no neon instruction in the code. However, using optimize 3 when compile would generate simd instruction according to the document. I think this is because the gnu compiler didn't recognize the huawei Kunpeng-920 automatically, as gcc 7.3 is released in 2018, while Kunpeng-920 is published in 2019. The program would be much slower without simd.

Part 4 - Difficulties & Solutions

Channels

I tried 4 different methods when adding channels. The first one is using std::array. Its performance is good, but its size is set only in compile time. So I changed to std::vector. However, the multiplication runs 20 times slower than before. Then I tried to merge separate vectors into one, but the performance didn't improve much. Finally, letting matrix own several MatrixImpl pointers works well.

The operator() for channels is still a problem. Now I have two operator() returning reference:

- operator(size_t row, size_t col) : return a `std::vector<std::reference_wrapper<T>>`. The problem is the reference need to be accessed by get() methods, and we can't use an initializer list to change all elements in all channels at once. That's really annoying.
- operator(size_t row, size_t col, size_t channel) : return a reference. The problem is we need to use this operator many times when modifying all elements in all channels.

I haven't come up with a good solution yet.

Thread safety

Although I didn't let the copy constructor and assignment copying the ownership of MatrixImpl, there are still ROI doing this. So, accessing MatrixImpl is not thread safe. Maybe adding a lock to MatrixImpl would solve this problem, but it would cause performance problems.