

# CS205:C/C++ Program - Project Report 4

## A class for Matrix

姓名: 张闻城

学号: 12010324

## 目录

### 1. 需求分析

- 1.1 `Mat` 类当中应该有哪些属性?
- 1.2 如何避免内存硬拷贝?
- 1.3 如何合理地进行内存管理?
- 1.4 如何让 `Mat` 类支持多种数据类型
- 1.5 如何实现支持多通道矩阵的 `Mat` 类?

### 2. 代码

- 2.1 `CMakeList.txt`
- 2.2 `mat.hpp`
  - 2.2.1 `Mat` 类的声明
  - 2.2.2 命名空间 `util`
- 2.3 `mat.tpp`
  - 2.3.1 构造器
  - 2.3.2 析构器
  - 2.3.3 运算符重载
  - 2.3.4 其他函数

### 3. 测试结果及实验验证

- 3.1 内存管理
  - 3.1.1 拷贝构造函数赋值
  - 3.1.2 运算符 `operator=()` 赋值
- 3.2 矩阵运算
  - 3.3.1 加减法
  - 3.3.2 乘法

### 4. 困难及解决

- 4.1 模板类的使用
  - 4.1.1 头文件的编写
  - 4.1.2 模板友元函数
- 4.2 运算结果误差 

### 5. 总结及反思

## 1. 需求分析

### 1.1 `Mat` 类当中应该有哪些属性?

首先, `Mat` 类应该有一些基本的属性用来表示矩阵的信息, 比如行数 `rows`、列数 `cols`、通道数 `channels` 以及存储矩阵各个元素值的数组(指针) `data`。另外, 这次的proj要求实现 **ROI** 的获取, 同时要在获取 **ROI** 的过程中避免内存硬拷贝。因此还需要其他属性来辅助完成这两个要求。在参考了 **OpenCV**<sup>1</sup> 之后, 我添加了一些用于辅助 **ROI** 实现的属性, 例如: `steps`, `beginRowIndex`, `beginColIndex` 等等。详细的实现方法将会在后面解释。

## 1.2 如何避免内存硬拷贝？

避免内存硬拷贝的最简单的方法是让两个 `Mat` 对象共享同一块存数据的内存，即使两个对象的 `data` 指向同一块堆中的内存。但是这种方式会造成一些内存管理方面的问题，例如：当两个 `Mat` 对象共享同一块数据时，在某一个对象的析构函数被调用之后，这块位于堆中的数据被释放了。之后若通过另一个对象再去访问这块数据的时候，便有可能产生错误(通常为 `Segmentation Fault`)，此外还有可能产生内存多次释放的问题。因此需要对内存进行严谨的管理。

## 1.3 如何合理地进行内存管理？

正如再上一个问题最后所提到的，如果想要避免内存的硬拷贝，将会出现某些内存管理问题。主要原因是由于 `C++` 中类的析构器机制。例如：如果在某个函数体内部创建了一个 `Mat` 对象，并且将这个对象 `return` 出去。在 `return` 语句后，这个对象的析构器将会自动被调用。如果 `Mat` 类的析构器中有一些将当前对象的 `data` 所指向的堆内存释放的操作，在析构函数被调用后这块内存就会被释放掉。此时如果再通过这个函数 `return` 出来的对象访问这块内存，便有可能会出现 `Segmentation Fault` 的错误，因为这块内存已经被释放。该问题的解决办法是再给 `Mat` 类添加一个属性 `refCntPtr`。这个属性是一个指向 `data` 最后一个元素的下一个内存的一个指针。这个指针指向的内存会储存共享当前这一块 `data` 内存的 `Mat` 对象的个数。只有在没有 `Mat` 对象使用这块 `data` 时，这块内存才会被释放。

## 1.4 如何让 `Mat` 类支持多种数据类型

泛型程序设计是一种算法在实现时不指定具体要操作的数据的类型的程序设计方法。在 `C++` 中，利用模板这种技术可以很好的进行泛型程序设计，实现对多种数据类型的支持。

## 1.5 如何实现支持多通道矩阵的 `Mat` 类？

实现多通道矩阵的一般做法是令矩阵的每个元素都储存每个通道的数据。但个人认为这种存储方式会导致同一通道内的元素分布不连续，从而造成访问时效率有所下降。

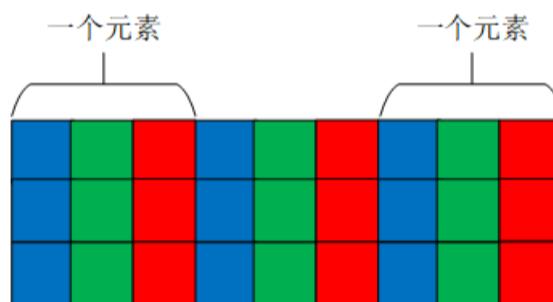


图1.(a) 多通道矩阵示意图

另一种思路是将不同通道的矩阵合成为一个大矩阵，只不过大矩阵的行数为原来小矩阵的行数乘以通道数。这种实现方法的一个好处是不同通道的矩阵间元素的分布是连续的，在访问时效率会相对高一些。

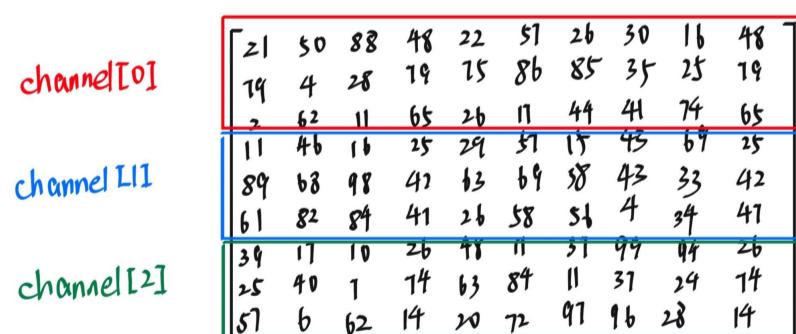


图1.(b) 多通道矩阵示意图

## 2. 代码

文件目录层级：

```
| CMakeLists.txt  
|   file  
|     image  
|       txt  
|   include  
|     mat.hpp  
|   src  
|     main.cpp  
|     mat.tpp
```

## 2.1 CMakeList.txt

```
1 cmake_minimum_required(VERSION 3.16)  
2 project(Project_4 CXX)  
3  
4 set(CMAKE_CXX_STANDARD 20)  
5 set(CMAKE_BUILD_TYPE DEBUG)  
6  
7 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ")  
8 set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/build)  
9  
10 option(ENABLE_OMP "use OpenMP" OFF)  
11 if (CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "x86_64")  
12   message("-- x86_64 CPU architecture detected")  
13   option(ENABLE_AVX2 "use AVX2 IS in x86 CPU" ON)  
14 endif ()  
15 option(ENABLE_NEON "use NEON IS in ARM CPU" OFF)  
16 option(ENABLE_O3 "use compiler O3 optimization" ON)  
17  
18 if (ENABLE_O3)  
19   if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU"  
20       OR "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")  
21     message("-- Turn on compiler O3 optimization.")  
22     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3")  
23   endif ()  
24 endif ()  
25  
26 if (ENABLE_OMP)  
27   message("Use OpenMP")  
28   find_package(OpenMP REQUIRED)  
29   if (OPENMP_FOUND)  
30     message("-- OpenMP Found.")  
31     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")  
32     add_definitions(-D_ENABLE_OMP)  
33   else ()  
34     message("-- OpenMP Not Found.")  
35   endif ()  
36 endif ()  
37  
38 if (ENABLE_AVX2)  
39   message("-- Use AVX2.")  
40   add_definitions(-D_ENABLE_AVX2)  
41   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -mavx2 -mfma")  
42 endif ()  
43  
44 if (ENABLE_NEON)  
45   message("-- Use NEON.")
```

```

46     add_definitions(-D_ENABLE_NEON)
47 endif()
48
49 find_package(Threads)
50
51 include_directories(include)
52 aux_source_directory(src DIR_SRCS)
53
54 add_executable(${PROJECT_NAME} ${DIR_SRCS})
55
56 target_link_libraries(${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
57
58 message("-- CXX_FLAGS: ${CMAKE_CXX_FLAGS}")
59 message("-- O3 optimization = ${ENABLE_O3}")
60 message("-- OpenMP = ${ENABLE_OMP}")
61 message("-- NEON = ${ENABLE_NEON}")
62 message("-- AVX2 = ${ENABLE_AVX2}")

```

在 `CMakeLists.txt` 中添加了一些 `option(ENABLE_AVX2, ENABLE_NEON, ENABLE_OMP)`(参考了Pro. Yu的Github开源项目<sup>2</sup>)，以便于用户在不同的平台能够利用不同的方式(指令集)加速矩阵的一些计算的过程。通过这些 `option`，CMake会在运行时根据选项定义新的宏(`_ENABLE_AVX2`, `_ENABLE_NEON`, `_ENABLE_OMP`)，用于开启不同的加速选项。同时还会检测当前环境是否能使用对于的指令集，若当前环境不包含某一指令集，即使用户打开了对应的加速选项，程序也不会开启相应的加速选项。

## 2.2 mat.hpp

### 2.2.1 Mat类的声明

```

1 template<typename Tp>
2 class Mat {
3
4 public:
5     Mat();
6     Mat(const Mat<Tp> &other);
7     Mat(size_t rows_, size_t cols_, int channels_);
8     Mat(size_t rows_, size_t cols_, int channels_, initializer_list<string> pathList);
9     explicit Mat(const string &path);
10    ~Mat();
11
12    Tp &operator()(size_t i, size_t j, int channelIndex = 0);
13    Mat<Tp> operator+(const Mat<Tp> &other) const;
14    Mat<Tp> operator-(const Mat<Tp> &other) const;
15    Mat<Tp> operator*(Mat<Tp> &other) const;
16    Mat<Tp> operator*(const int &multiplier) const;
17    Mat<Tp> &operator=(const Mat &other);
18    bool operator==(const Mat &other) const;
19    bool operator!=(const Mat &other) const;
20
21    template<typename Tp_>
22        friend Mat<Tp_> operator*(const int &multiplier, const Mat<Tp_> &other);
23    template<typename Tp_>
24        friend ostream &operator<<(ostream &os, const Mat<Tp_> &mat);
25    template<typename Tp_>
26        explicit operator Mat<Tp_>() const;
27

```

```

28     [[nodiscard]] Mat<Tp> subMat(size_t _rows_, size_t _cols_, size_t _beginRowIndex_, size_t
29     _beginColIndex_) const;
30     [[nodiscard]] Mat<Tp> roi(size_t rowIndex, size_t colIndex, size_t rows_, size_t cols_)
31     const;
32     [[nodiscard]] Mat<Tp> clone() const;
33     [[nodiscard]] Tp &at(size_t i, size_t j, int channelIndex = 0) const;
34     [[nodiscard]] Tp *rowPtr(size_t rowIndex, int channelIndex = 0) const;
35     [[nodiscard]] Tp *colPtr(size_t colIndex, int channelIndex = 0) const;
36
37     void print() const;
38     Mat<Tp> transfer();
39     void writeTo(const char *path) const;
40
41     static Mat<Tp> txtRead(const string &path);
42     static Mat<Tp> rand(size_t rows_, size_t cols_, int channels_, Tp bottom, Tp top);
43
44     [[nodiscard]] size_t getRows() const;
45     [[nodiscard]] size_t getCols() const;
46     [[nodiscard]] size_t getSteps() const;
47     [[nodiscard]] size_t getBeginRowIndex() const;
48     [[nodiscard]] size_t getBeginColIndex() const;
49     [[nodiscard]] size_t getDataRows() const;
50     [[nodiscard]] size_t getDataCols() const;
51     [[nodiscard]] int getChannels() const;
52     [[nodiscard]] Tp **getData() const;
53     [[nodiscard]] int *getRefCntPtr() const;
54     [[nodiscard]] DataType getDataType() const;
55
56     void setRefCntPtr(int *ptr);
57     void setRows(size_t _rows_);
58     void setCols(size_t _cols_);
59     void setSteps(size_t _steps_);
60     void setBeginRowIndex(size_t _beginRowIndex_);
61     void setBeginColIndex(size_t _beginColIndex_);
62     void setDataRows(size_t _dataRows_);
63     void setDataCols(size_t _dataCols_);
64     void setData(Tp **_data_);
65     void setChannels(int channels_);
66     void setDataType(DataType dataType_);
67
68 private:
69     size_t rows{};
70     size_t cols{};
71     size_t steps{};
72     size_t beginRowIndex{};
73     size_t beginColIndex{};
74     size_t dataRows{};
75     size_t dataCols{};
76     int channels;
77     Tp **data{};
78     int *refCntPtr{};
79     DataType dataType;
80
81     void thread_product(const Mat<Tp> &, Mat<Tp> &) const;
82
83     static void singleThreadMul(const Mat &, const Mat &, Mat &, size_t, size_t);
84 };

```

各个属性的详细细节：

- `rows` 和 `cols`: 这两个属性分别表示当前对象在 `data` 中所需要的某一块区域(可能是子矩阵，也可能就是这个 `data` 所指向的整体的矩阵)的行数和列数。
- `data`: 指向储存矩阵元素内存的指针。该内存全部位于堆中。且内存都是连续的。
- `dataRows` 和 `dataCols`: 这两个属性表示被 `data` 指向的矩阵的实际的行数和列数。这两个属性所表示的含义与 `rows` 和 `cols` 不同。
- `beginRowIndex` 和 `beginColIndex`: 这两个属性分别表示当前对象在 `data` 中所需要的某一块区域(可能是子矩阵，也可能就是这个 `data` 所指向的整体的矩阵)中第一个元素在 `data` 所指向的整体矩阵中的行下标和列下标。

示例：

	B.data									
	21	50	88	48	22	51	26	30	16	48
0	19	4	28	79	75	86	85	35	25	19
1	2	62	11	65	26	11	44	41	74	65
2	11	46	16	25	29	51	85	43	69	25
3	89	68	98	42	43	69	38	43	33	42
4	61	82	84	47	26	58	56	4	34	41
5	39	17	10	26	48	11	31	99	94	26
6	25	40	1	74	63	84	11	31	24	14
7	57	6	62	14	20	72	97	16	28	14
8	9x10									

图2.(a)

两个 `Mat` 对象 `A` 和 `B`。并且 `A.data` 和 `B.data` 都指向上图所示的矩阵(的首元素)。`A` 和 `B` 的 `dataRows` 的值都为 `9`, `dataCols` 的值都为 `10`，因为两个 `Mat` 对象 `data` 都指向一块 `9` 行 `10` 列的数组(一维方式存储)。

`A` 表示的是图中红色区域内的矩阵。由于红色区域矩阵的行数和列数分别为 `3` 和 `4`，因此 `A.rows = 3`, `A.cols = 4`。由于红色区域矩阵的首元素的相对于整个大矩阵的行下标和列下标分别为 `1` 和 `2`，因此 `A.beginRowIndex = 1`, `A.beginColIndex = 2`。

	B.data									
	0	1	2	3	4	5	6	7	8	9
0	21	50	88	48	22	51	26	30	16	48
1	19	4	28	79	75	86	85	35	25	19
2	2	62	11	65	26	11	44	41	74	65
3	11	46	16	25	29	51	85	43	69	25
4	89	68	98	42	43	69	38	43	33	42
5	61	82	84	47	26	58	56	4	34	41
6	39	17	10	26	48	11	31	99	94	26
7	25	40	1	74	63	84	11	31	24	14
8	57	6	62	14	20	72	97	16	28	14
9	9x10									

图2.(b)

`B` 表示的是图中绿色区域内的矩阵。由于绿色区域矩阵的行数和列数分别为 `5` 和 `3`，因此 `B.rows = 5`, `B.cols = 3`。由于绿色区域矩阵的首元素的相对于整个大矩阵的行下标和列下标分别为 `4` 和 `3`，因此 `B.beginRowIndex = 4`, `B.beginColIndex = 3`。

	B.data									
	0	1	2	3	4	5	6	7	8	9
0	21	50	88	48	22	51	26	30	16	48
1	19	4	28	79	75	86	85	35	25	19
2	2	62	11	65	26	11	44	41	74	65
3	11	46	16	25	29	51	85	43	69	25
4	89	68	98	42	43	69	38	43	33	42
5	61	82	84	47	26	58	56	4	34	41
6	39	17	10	26	48	11	31	99	94	26
7	25	40	1	74	63	84	11	31	24	14
8	57	6	62	14	20	72	97	16	28	14
9	9x10									

图2.(c)

- `steps`: 与 `dataCols` 有相同的值，用于访问位于同一列但不同行的元素(参考 [OpenCV<sup>3</sup>](#))。

例：在上面的例子里从矩阵 `A`(红色区域内)的第一行最后一个元素到下一行的第一个元素时，需要跨过 `data` 中的十个元素。

- `refCntPtr`: 指向 `data` 最后一个元素的下一个内存的一个指针。这个指针指向的内存会储存共享当前这一块 `data` 内存的 `Mat` 对象的个数。在上边的例子中，`A` 和 `B` 的 `*refCntPtr` 的值都为 `2`，因为 `A` 和 `B` 共享了同一块 `data` 内存。

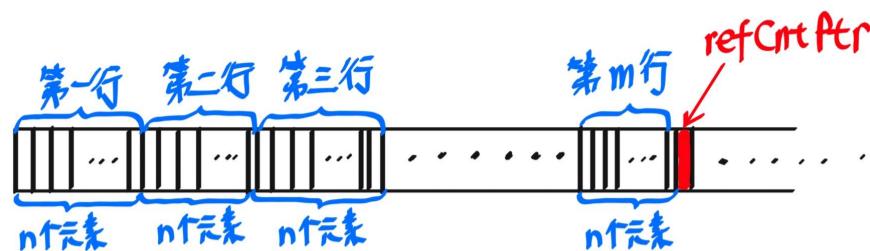


图3.  $m \times n$ 矩阵的refCntPtr的内存示意图

## 2.2.2 命名空间util

命名空间util中定义了一些用于辅助Mat类的函数。由于定义在Mat类中不太符合设计模式，因此定义在了命名空间中

```

1  namespace util {
2      string *tpStr = new string [6>{"uchar", "short", "int", "long", "float", "double"};
3
4      template<typename Tp>
5      Tp **allocate(size_t rows, size_t cols);
6      template<typename Tp>
7      void read(const string &path, Tp **dest, size_t rows, size_t cols);
8      string tpToString(DataType dataType);
9      DataType dataType(const type_info &typeInfo);
10     size_t getRowsOfTXTFile(const string &filePath);
11     size_t getColsOfTXTFile(const string &filePath);
12     void error();
13 }
14
15 template<typename Tp>
16 Tp **util::allocate(size_t rows, size_t cols)
17 {
18     auto *oneDimensionArray = static_cast<Tp *>(aligned_alloc(512, rows * cols * sizeof(Tp) +
19     sizeof(int)));
20     for (int i = 0; i < rows * cols; i++) {
21         oneDimensionArray[i] = {};
22     }
23     auto **res = static_cast<Tp **>(malloc(sizeof(Tp *) * rows));
24     for (size_t i = 0; i < rows; i++) {
25         res[i] = oneDimensionArray + i * cols;
26     }
27     return res;
}

```

命名空间util中最重要的函数是allocate()。这个函数会分配特定大小的堆内存，用于储存矩阵元素。且这块内存的首地址会以512字节对齐，这样在取数据的时候效率比较高。由于在将硬盘中的数据中加载到内存的过程中，计算机会按对齐的地址一次取2的整数次幂个字节的数据。若不对齐的话，对于一个数据可能要进行多次由硬盘加载到内存的操作，效率会有所损失。如下图的例子：在内存不对齐的情况下，程序需要先从0x0003中取该int的一部分数据，再从0x0004-0x0006取剩下的int的数据，最后还需要两部分数据合并再加载到内存中。在内存对齐的情况下，程序只需要取一次数据。因此，内存对齐有利于提高数据访问的效率。

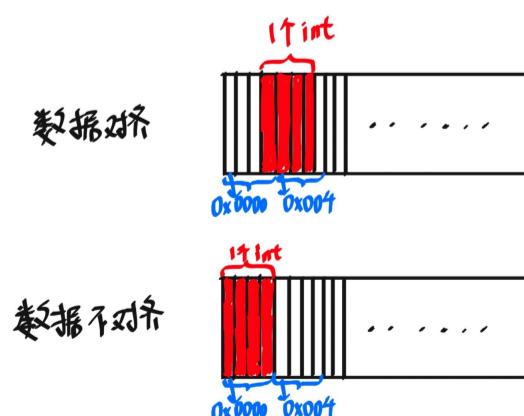


图4. 内存对齐与不对齐的对比

矩阵的元素都是以一维数组的方式存储的。同时还会利用一个二级指针，另二级指针所指向的各个一级指针分别指向矩阵元素每一行的首元素。

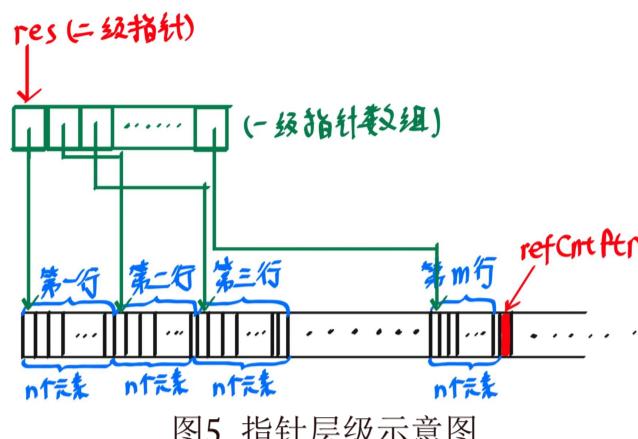


图5. 指针层级示意图

其他的函数大多用于debug，因此在此不过多提及。

## 2.3 mat.tpp

### 2.3.1 构造器

- `Mat()`:

```
1 template<typename Tp>
2 Mat<Tp>::Mat()
3     : rows(0), cols(0), steps(0), beginRowIndex(0), beginColIndex(0), dataRows(0),
4       dataCols(0), channels(1),
5       data(nullptr), refCntPtr(nullptr), dataType(util::dataType(typeid(Tp)))
```

`Mat`类的默认构造函数，会将所有属性的初始化。

- `Mat(const Mat<Tp> other)`:

```
1 template<typename Tp>
2 Mat<Tp>::Mat(const Mat<Tp> &other)
3     : rows(other.rows), cols(other.cols), steps(other.steps),
4       beginRowIndex(other.beginRowIndex),
5       beginColIndex(other.beginColIndex), dataRows(other.dataRows), dataCols(other.dataCols),
6       channels(other.channels),
7       data(other.data), refCntPtr(other.refCntPtr), dataType(util::dataType(typeid(Tp)))
8 {
9     *(this->refCntPtr) += 1;
10 }
```

`Mat`类的拷贝构造函数。在拷贝的过程中对于矩阵元素采用浅拷贝的方式，即令 `this` 对象和 `other` 对象的 `data` 指向同一块堆数据。同时由于这块 `data` 被指向的次数增加，需要对 `*refCntPtr` 进行加一的操作。

- `Mat(size_t rows_, size_t cols_, int channels_)`:

```

1 template<typename Tp>
2 Mat<Tp>::Mat(size_t rows_, size_t cols_, int channels_)
3     : rows(rows_), cols(cols_), steps(cols_), beginRowIndex(0), beginColIndex(0),
4     dataRows(rows_), dataCols(cols_),
5     channels(channels_), data(nullptr), refCntPtr(nullptr),
6     dataType(util::dataType(typeid(Tp)))
7 {
8     this->data = util::allocate<Tp>(this->channels * this->rows, this->cols);
9     setRefCntPtr((int *)(&this->data[this->channels * this->dataRows - 1][this->dataCols]));
10    *this->getRefCntPtr() = 1;
11 }

```

构造一个有指定 `rows`, `cols` 和 `channels` 的 `Mat` 对象。`data` 会被分配内存空间, 且全部被初始化。被分配给 `data` 的内存空间的尺寸是 `rows_ × cols_ × channels_`, 目的是为了实现多通道矩阵(如何实现支持多通道矩阵的 `Mat` 类?)。另外还会对其他属性进行初始化。

- `Mat(size_t rows_, size_t cols_, int channels_, initializer_list<string> pathList):`

```

1 template<typename Tp>
2 Mat<Tp>::Mat(size_t rows_, size_t cols_, int channels_, initializer_list<string> pathList)
3     : rows(rows_), cols(cols_), steps(cols_), beginRowIndex(0), beginColIndex(0),
4     dataRows(rows_),
5     dataCols(cols_), channels(channels_), data(nullptr), refCntPtr(nullptr),
6     dataType(util::dataType(typeid(Tp)))
7 {
8     if (channels_ <= 0) {
9         cout << "Invalid number of channels. Exit." << endl;
10        exit(EXIT_FAILURE);
11    }
12    if (this->channels != pathList.size()) {
13        cout << "Inconsistent number of channels and path of files. Exit." << endl;
14        exit(EXIT_FAILURE);
15    }
16    this->data = util::allocate<Tp>(this->channels * this->rows, this->cols);
17    int channelIndex = 0;
18    for (const string &beg: pathList) {
19        if (this->rows != util::getRowsOfTXTFile(beg) || this->cols !=
20            util::getColsOfTXTFile(beg)) {
21            cout << "Inconsistent specified size and data's size in file. Exit." << endl;
22            exit(EXIT_FAILURE);
23        }
24        util::read(beg, &(this->data[this->rows * channelIndex]), this->rows, this->cols);
25        channelIndex++;
26    }
27    setRefCntPtr((int *)(&this->data[this->channels * this->dataRows - 1][this->dataCols]));
28    *this->getRefCntPtr() = 1;
29 }

```

该构造器用于构造一个有特定矩阵元素和指定通道数的 `Mat` 对象。矩阵各元素的具体值会分别从 `pathList` 中的一个或者多个路径中读取。参数 `rows_` 和 `cols` 的值应该与存储矩阵元素的文件(如 `.txt` 文本文件)中数据的行数和列数相匹配。此外参数 `channels_` 的值应该与 `pathList` 中的路径的个数相同。

例：如果想构造一个三通道矩阵，且各个通道的元素的值分别存储在路径 `file/txt/mat-A-32.txt`, `file/txt/mat-B-32.txt` 和 `file/txt/mat-C-32.txt` 中，则可以通过下面的语句构造 `Mat` 对象。

```
1 Mat<float> mat_3_channels(32, 32, 3, {"file/txt/mat-A-32.txt", "file/txt/mat-B-32.txt",
2 "file/txt/mat-C-32.txt"});
```

- `Mat(const string &path):`

```
1 template<typename Tp>
2 Mat<Tp>::Mat(const string &path)
3     : beginRowIndex(0), beginColIndex(0), channels(1), data(nullptr), refCntPtr(nullptr),
4       dataType(util::dataType(typeid(Tp)))
5 {
6     this->rows = util::getRowsOfTXTFile(path);
7     this->cols = util::getColsOfTXTFile(path);
8     this->dataRows = this->rows;
9     this->dataCols = this->cols;
10    this->steps = this->cols;
11    this->data = util::allocate<Tp>(this->rows, this->cols);
12    util::read(path, this->data, this->rows, this->cols);
13    setRefCntPtr((int *)(&this->data[this->dataRows - 1][this->dataCols]));
14    *this->getRefCntPtr() = 1;
15 }
```

与上一个构造器类似，只不过只创建一个单通道矩阵，并且从指定路径 `path` 中读取矩阵的元素，且会自动计算矩阵的行数和列数。

以上所有的构造器在构造器定义的最后一步都会令属性 `refCntPtr` 指向 `data` 数据的尾部，并且将其指向的值初始化为 `1` 或者加 `1`。

## 2.3.2 析构器

`~Mat():`

```
1 template<typename Tp>
2 Mat<Tp>::~Mat()
3 {
4     if (this->data == nullptr) {
5         cout << "-- [data] has not been allocated memory. Return." << endl;
6         return;
7     }
8     if (this->refCntPtr == nullptr) {
9         cout << "-- [refCntPtr] has not been allocated memory. Return." << endl;
10    }
11    cout << "-- \033[1m\033[34m~Mat()\033[0m: refCnt: " << *this->refCntPtr << " --> ";
12    *this->refCntPtr -= 1;
13    cout << *this->refCntPtr << endl;
14    if (*this->refCntPtr <= 0) {
15        free(this->data[0]);
16        free(this->data);
17        cout << "\033[1m\033[34m~Mat()\033[0m: Free memory successfully." << endl;
18    }
19 }
```

`Mat`类中的析构器在内存管理中起着至关重要的作用。当某一个`Mat`对象的析构器被调用时，析构器首先会将当前对象的`refCntPtr`所指向的值减1。若减1之后`refCnt`值大于0，说明仍有其他`Mat`对象在共享当前对象的`data`所指向的内存，因此此时这些内存不需要被释放。若减1之后`refCnt`的值等于0，则说明没有`Mat`对象还共享当前对象的`data`所指向的内存，因此此时这些内存可以被安全地释放(利用C中的`free()`释放的原因是因为这些内存都是由C中的`aligned_alloc()`所分配的内存)。通过调整析构器按以上方式运作，内存可以被小心，谨慎地管理，并且不需要担心内存泄漏的问题。

### 2.3.3 运算符重载

- `operator()(size_t i, size_t j, int channelIndex)`

```
1 template<typename Tp>
2 inline Tp &Mat<Tp>::operator()(size_t i, size_t j, int channelIndex)
3 {
4     return *(&(*this->data[channelIndex] * this->rows + this->beginRowIndex)[this->beginColIndex])
5     + j + i * this->steps);
6 }
```

通过运算符`()`来访问矩阵中的某一个通道上的某一个元素。由于某个`Mat`对象所代表的举证可能只是`data`中的某一部分(子矩阵)，因此在访问矩阵的元素时需要通过一系列简单的计算来定位当想要的位置。由于访问矩阵元素是频繁的操作，该函数需要被调用，所以为该函数添加`inline`关键词，建议编译器在调用该函数的地方将函数调用直接替换成函数体，减少函数的压栈和出栈所造成的内存开销，一定程度上可提高程序运行效率。

- `operator+(const Mat<> &other)`

```
1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::operator+(const Mat<Tp> &other) const
3 {
4     if (this->rows != other.rows || this->cols != other.cols || this->channels != other.channels) {
5         cout << "Inconsistent size of two matrices for addition. Exit" << endl;
6         exit(EXIT_FAILURE);
7     }
8     cout << "In Mat<Tp>::operator+(): ";
9     Mat<Tp> res(other.rows, other.cols, other.channels);
10    util::Timer timer("Matrix<Tp> addition");
11 #if defined(_ENABLE_OMP)
12 #pragma omp parallel for num_threads((int)thread::hardware_concurrency())
13 #endif
14    for (int k = 0; k < this->channels; k++) {
15        for (size_t i = 0; i < this->rows; i++) {
16            for (size_t j = 0; j < this->cols; j++) {
17                res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
18            }
19        }
20    }
21    return res;
22 }
23
24 template<>
25 Mat<float> Mat<float>::operator+(const Mat<float> &other) const
26 {
27     if (this->rows != other.rows || this->cols != other.cols || this->channels != other.channels) {
28         cout << "Inconsistent size of two matrices for addition. Exit" << endl;
29         exit(EXIT_FAILURE);
30     }
```

```

31     Mat<float> res(other.rows, other.cols, other.channels);
32     util::Timer timer("Mat<float> addition");
33 #if defined(_ENABLE_OMP)
34 #pragma omp parallel for num_threads((int)thread::hardware_concurrency())
35 #endif
36
37 #if defined(_ENABLE_AVX2)
38     for (int k = 0; k < this->channels; k++) {
39         for (size_t i = 0; i < this->rows; i++) {
40             for (size_t j = 0; j < this->cols; j += 8) {
41                 __m256 v1, v2;
42                 __m256 r = _mm256_setzero_ps();
43                 v1 = _mm256_loadu_ps(this->rowPtr(i, k) + j);
44                 v2 = _mm256_loadu_ps(other.rowPtr(i, k) + j);
45                 r = _mm256_add_ps(v1, v2);
46                 _mm256_storeu_ps(res.rowPtr(i, k) + j, r);
47             }
48             for (size_t j = this->cols / 8 * 8; j < this->cols; j++) {
49                 res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
50             }
51         }
52     }
53 #elif defined(_ENABLE_NEON)
54     for (int k = 0; k < this->channels; k++) {
55         for (size_t i = 0; i < this->rows; i++) {
56             for (size_t j = 0; j < this->cols; j += 4) {
57                 float32x4_t v1 = vdupq_n_f32(0.0f);
58                 float32x4_t v2 = vdupq_n_f32(0.0f);
59                 float32x4_t r = vdupq_n_f32(0.0f);
60                 v1 = vld1q_f32(this->rowPtr(i, k) + j);
61                 v2 = vld1q_f32(other.rowPtr(i, k) + j);
62                 r = vaddq_f32(v1, v2);
63                 vst1q_f32(res.rowPtr(i, k) + j, r);
64             }
65             for (size_t j = this->cols / 4 * 4; j < this->cols; j++) {
66                 res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
67             }
68         }
69     }
70 #else
71     for (int k = 0; k < this->channels; k++) {
72         for (size_t i = 0; i < this->rows; i++) {
73             for (size_t j = 0; j < this->cols; j++) {
74                 res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
75             }
76         }
77     }
78 #endif
79     return res;
80 }
81
82 template<>
83 Mat<int> Mat<int>::operator+(const Mat<int> &other) const
84 {
85     if (this->rows != other.rows || this->cols != other.cols) {
86         cout << "Inconsistent size of two matrices for addition. Exit" << endl;
87         exit(EXIT_FAILURE);
88     }

```

```

89     Mat<int> res(other.rows, other.cols, other.channels);
90     util::Timer timer("Mat<int> Addition");
91 #if defined(_ENABLE_OMP)
92 #pragma omp parallel for num_threads((int)thread::hardware_concurrency())
93 #endif
94 #if defined(_ENABLE_AVX2)
95     for (int k = 0; k < this->channels; ++k) {
96         for (size_t i = 0; i < this->rows; ++i) {
97             for (size_t j = 0; j < this->cols; j += 8) {
98                 __m256i v1, v2;
99                 __m256i r = _mm256_setzero_si256();
100                v1 = _mm256_loadu_si256((__m256i *) (this->rowPtr(i, k) + j));
101                v2 = _mm256_loadu_si256((__m256i *) (other.rowPtr(i, k) + j));
102                r = _mm256_add_epi32(v1, v2);
103                _mm256_storeu_si256((__m256i *) (res.rowPtr(i, k) + j), r);
104            }
105            for (size_t j = this->cols / 8 * 8; j < this->cols; ++j) {
106                res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
107            }
108        }
109    }
110    for (size_t i = (this->rows * this->cols * this->channels) / 8 * 8;
111        i < this->rows * this->cols * this->channels; ++i) {
112        *(res.data[0] + i) = *(this->data[0] + i) + *(other.data[0] + i);
113    }
114 #elif defined(_ENABLE_NEON)
115     for (int k = 0; k < this->channels; ++k) {
116         for (size_t i = 0; i < this->rows; ++i) {
117             for (size_t j = 0; j < this->cols; j += 4) {
118                 int32x4_t v1 = vdupq_n_s32(0.0f);
119                 int32x4_t v2 = vdupq_n_s32(0.0f);
120                 int32x4_t r = vdupq_n_s32(0.0f);
121                 v1 = vld1q_s32(this->rowPtr(i, k) + j);
122                 v2 = vld1q_s32(other.rowPtr(i, k) + j);
123                 r = vaddq_s32(v1, v2);
124                 vst1q_s32(res.rowPtr(i, k) + j, r);
125             }
126             for (size_t j = this->cols / 4 * 4; j < this->cols; ++j) {
127                 res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
128             }
129         }
130     }
131 #else
132     for (int k = 0; k < this->channels; ++k) {
133         for (size_t i = 0; i < this->rows; ++i) {
134             for (size_t j = 0; j < this->cols; ++j) {
135                 res.at(i, j, k) = this->at(i, j, k) + other.at(i, j, k);
136             }
137         }
138     }
139 #endif
140     return res;
141 }
```

“+”运算符对于矩阵的意义是将所有相同位置上的元素相加求和，得到一个新的矩阵。在该项目中，对矩阵加法做了一定的加速处理。第一种方法是利用**OpenMP**，将`for`循环内的语句拆分到多个线程中并行计算。第二种方法是利用**SIMD**，使用单一指令操作多条数据。在关闭编译器**O3**优化的情况下，两种方法都有一定明显的收益(若开启**O3**优化，由于编译器会自动对代码逻辑和计算过程进行大量的优化，因此在这种情况下使用**OMP**和**SIMD**的效果都不明显)。

由于`Mat`类是支持多种数据类型的矩阵的，因此在使用**SIMD**加速矩阵运算的过程中，需要根据不同的数据类型，使用不同的**SIMD**指令进行加速。这就要求在加速之前必须知道当前`Mat`对象在创建时的具体的数据类型是什么。如何才能获取该信息呢？我想到了利用模板函数实例化的方法。编写多个数据类型确定的`operator+( )`函数(即`template<Tp> operator()`模板函数的实例化)，在不同的示例化的`operator+( )`函数中，根据当前已实例化的数据类型选择不同的**SIMD**指令加速。这样在`Mat`对象被创建好后需要运算时，程序便会自动调用数据类型相一致的实例化函数来进行计算(如果相应的数据类型已被实例化)。例：

```
1 Mat<float> mat_float();
2 mat_float + mat_float;
3 /*-----*/
4 Mat<int> mat_int();
5 mat_int + mat_int;
```

在第一个加法运算中，实例化的模板函数`Mat<float>::operator+( )`将会被调用；在第二个加法运算中，实例化的模板函数`Mat<int>::operator+( )`将会被调用。若元素的数据类型无法使用**SIMD**进行加速，则调用`Mat<Tp>::operator+( )`进行计算。

此外该程序还对不同的平台采用了不同的**SIMD**指令加速：**x86**平台采用**AVX2**指令集加速，**arm**平台采用**NEON**指令集加速。该过程的实现依赖于**CMakeLists.txt**的编写。通过在**CMakeLists.txt**添加自定义选项来让用户自定义加速方式。

由于不同的数据类型的**SIMD**加速只需要改变使用的相关**SIMD**指令函数，因此对于`float`和`int`以外的数据类型的加速的具体代码在此没有过多提及，但核心的思路是相同的。详细情况可查看`mat.tpp`源文件

- `operator-(const Mat<> &other)`

```
1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::operator-(const Mat<Tp> &other) const
3 {
4     if (this->rows != other.rows || this->cols != other.cols) {
5         cout << "Inconsistent size of two matrices for addition. Exit" << endl;
6         exit(EXIT_FAILURE);
7     }
8     Mat<Tp> res(other.rows, other.cols, other.channels);
9     util::Timer timer("Mat<Tp> subtraction");
10    for (size_t i = 0; i < this->rows * this->cols * this->channels; i++) {
11        *(res.data[0] + i) = *(this->data[0] + i) - *(other.data[0] + i);
12    }
13    return res;
14 }
```

思路与`operator+( )`的实现大同小异，在此不展开叙述，详细内容可以查看源文件`mat.tpp`。

- `operator*( )`:

-两个矩阵的相乘：

```
1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::operator*(Mat<Tp> &other) const
3 {
4     if (this->cols != other.rows) {
5         cout << "Inconsistent matrices for multiplication. Exit." << endl;
6         exit(EXIT_FAILURE);
7     }
```

```

8     Mat<Tp> res(other.rows, other.cols, other.channels);
9     if (thread::hardware_concurrency() >= 1) {
10         util::Timer timer("Mat<Tp> Multiplication-Threads");
11         this->thread_product(other, res);
12         return res;
13     }
14     util::Timer timer("Mat<Tp> Multiplication");
15 #if defined(_ENABLE_OMP)
16 #pragma omp parallel for num_threads((int)thread::hardware_concurrency())
17 #endif
18     for (int p = 0; p < this->channels; ++p) {
19         for (size_t i = 0; i < this->rows; i++) {
20             for (size_t k = 0; k < this->cols; ++k) {
21                 Tp tmp = this->at(i, k, p);
22                 for (size_t j = 0; j < other.cols; ++j) {
23                     res.at(i, j, p) += tmp * other.at(k, j, p);
24                 }
25             }
26         }
27     }
28     return res;
29 }
30
31 template<typename Tp>
32 inline void Mat<Tp>::thread_product(const Mat<Tp> &other, Mat<Tp> &res) const
33 {
34     size_t threadsNum = thread::hardware_concurrency();
35     size_t rowNumOfEachThread = this->rows / threadsNum;
36     auto *threads = new thread[threadsNum]{};
37     for (size_t i = 0; i < threadsNum; ++i) {
38         threads[i] = std::thread(Mat<Tp>::singleThreadMul, ref(*this), ref(other), ref(res), i *
39 rowNumOfEachThread,
40                                     rowNumOfEachThread);
41     }
42     for (size_t i = 0; i < threadsNum; ++i) {
43         threads[i].join();
44     }
45 }
46
47 template<typename Tp>
48 void Mat<Tp>::singleThreadMul(const Mat<Tp> &mat_A, const Mat<Tp> &mat_B, Mat<Tp> &mat_C, const
49 size_t _beginRowIndex_,
50                                 const size_t rowNumToCal)
51 {
52     for (int p = 0; p < mat_A.channels; ++p) {
53         for (size_t i = 0; i < rowNumToCal; ++i) {
54             for (size_t k = 0; k < mat_A.cols; ++k) {
55                 Tp temp = mat_A.at(_beginRowIndex_ + i, k);
56                 for (size_t j = 0; j < mat_B.cols; ++j) {
57                     mat_C.at(_beginRowIndex_ + i, j, p) += temp * mat_B.at(k, j, p);
58                 }
59             }
60         }
61     }
62 }
```

在该 `Mat` 类中，矩阵乘法主要采用了多线程的加速方法。该加速方法最快可以达到  $2048 \times 2048$  矩阵乘法耗时 **263ms**。考虑到适应于不同的CPU，在矩阵乘法进行运算之前还会对CPU线程数进行检查：如果线程数大于1，则直接使用多线程加速；反之则使用 **I/O** 访存优化矩阵乘法，减少有线程开启和销毁带来的开销(原本计划是对于不同数据类型的矩阵乘法，使用不同**SIMD**进行加速，类似于矩阵加法。但由于时间关系，该环节目前还未得到实现)。

矩阵乘法加速方式在 *Project 2* 和 *Project 3* 中已经有过详细的研究和说明，不是本次项目中的重点。因此本次项目中只选用了 *Project 2* 和 *Project 3* 中加速效果最好的方式来加速。

-矩阵与常数相乘

```
1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::operator*(const int &multiplier) const
3 {
4     Mat<Tp> res(this->rows, this->cols, this->channels);
5 #if defined(_ENABLE_OMP)
6 #pragma omp parallel for num_threads((int)thread::hardware_concurrency())
7 #endif
8     for (int k = 0; k < this->channels; k++) {
9         for (size_t i = 0; i < this->rows; i++) {
10            for (size_t j = 0; j < this->cols; j++) {
11                res.at(i, j, k) = this->at(i, j, k) * multiplier;
12            }
13        }
14    }
15    return res;
16 }
17
18 //该函数声明为Mat类的友元函数
19 template<typename Tp>
20 Mat<Tp> operator*(const int &multiplier, const Mat<Tp> &mat)
21 {
22     return mat * multiplier;
23 }
```

该 `Mat` 类还支持矩阵与常数的乘法运算。对于 常数  $\times$  矩阵这样形式的运算，采用友元函数的方式进行运算符重载。并且在计算时可以开启**OMP**选项，利用**OMP**进行加速。

- `operator<<()`

```
1 template<typename Tp>
2 ostream &operator<<(ostream &os, const Mat<Tp> &mat)
3 {
4     for (int k = 0; k < mat.channels; k++) {
5         os << "Data type is [" << util::tpToString(mat.dataType) << "]. Elements in channel[" <<
6         k << "]:" << endl;
7         for (size_t j = 0; j < mat.cols; j++) {
8             os << "_____";
9         }
10        os << "__" << endl;
11        for (size_t i = 0; i < mat.rows; i++) {
12            os << "|";
13            for (size_t j = 0; j < mat.cols; j++) {
14                os << mat.at(i, j, k) << "\t";
15            }
16            os << "|" << endl;
17        }
18        for (size_t j = 0; j < mat.cols; j++) {
```

```

18         os << "-----";
19     }
20     os << "---" << endl;
21 }
22 os << "rows = " << mat.rows << ". cols = " << mat.cols << endl;
23 os << "steps = " << mat.steps << endl;
24 os << "beginRowIndex = " << mat.beginRowIndex << ". beginColIndex = " << mat.beginColIndex
<< endl;
25 os << "dataRows = " << mat.dataRows << ". dataCols = " << mat.dataCols << endl;
26 os << "channels = " << mat.channels << endl;
27 os << "refCntPtr = " << mat.refCntPtr << ". refCnt = " << *(mat.refCntPtr) << endl;
28 return os;
29 }

```

重载`<<`输出运算符，对矩阵对象的一系列信息进行打印输出。

- `operator=()`

```

1 template<typename Tp>
2 Mat<Tp> &Mat<Tp>::operator=(const Mat &other)
3 {
4     if (this == &other) {
5         return *this;
6     }
7     if (this->data != nullptr && this->refCntPtr != nullptr) {
8         cout << "operator(): refCnt: " << *this->refCntPtr << " --> ";
9         *this->refCntPtr -= 1;
10    cout << *this->refCntPtr << endl;
11    if (*this->refCntPtr <= 0) {
12        cout << "In operator(): ";
13        free(this->data[0]);
14        free(this->data);
15        this->data = nullptr;
16
17        cout << "operator(): Free memory successfully." << endl;
18    }
19 }
20 this->rows = other.rows;
21 this->cols = other.cols;
22 this->steps = other.steps;
23 this->beginRowIndex = other.beginRowIndex;
24 this->beginColIndex = other.beginColIndex;
25 this->dataRows = other.dataRows;
26 this->dataCols = other.dataCols;
27 this->channels = other.channels;
28 this->data = other.data;
29 this->refCntPtr = other.refCntPtr;
30 cout << "operator(): refCnt: " << *this->refCntPtr << " --> ";
31 *(this->refCntPtr) += 1;
32 cout << *this->refCntPtr << endl;
33 return *this;
34 }

```

同 `Mat` 类的拷贝构造函数类似，赋值运算符 `=` 仍然采用浅拷贝的方式将对矩阵对象进行赋值操作。在函数尾部对 `*refCntPtr` 进行加1的操作是因为 `this` 矩阵对象开始与 `other` 矩阵对象共享一块 `data` 内存空间。同时在对 `this` 对象的 `data` 进行赋值之前，需要根据 `data` 的情况做不同的处理(上方代码块的7-17行)：若当前对象的 `data` 为 `nullptr`，则说明该对象不代表任何矩阵，则直接将 `other` 的 `data` 赋值给 `this` 的 `data`，令这两个 `Mat` 对象享用同一块内存；若当前对象的 `data` 不是 `nullptr`，将另一个矩阵赋值给当前矩阵意味着当前矩阵不再引用原来的 `data` 内存，即这块 `data` 内存的 `refCnt` 需要减1，随后还需要判断这块内存是否还有其他矩阵在引用并判断是否需要释放内存(上方代码的11-17行)，以避免内存泄漏。

- `operator==()` 和 `operator!=()`

```
1 template<typename Tp>
2 bool Mat<Tp>::operator==(const Mat &other) const
3 {
4     if (this->rows != other.rows || this->cols != other.cols || this->channels != other.channels) {
5         return false;
6     }
7     for (int k = 0; k < this->channels; k++) {
8         for (size_t i = 0; i < other.rows; i++) {
9             for (size_t j = 0; j < other.cols; j++) {
10                 if (this->at(i, j, k) != other.at(i, j, k)) {
11                     return false;
12                 }
13             }
14         }
15     }
16     return true;
17 }
18
19 template<typename Tp>
20 bool Mat<Tp>::operator!=(const Mat &other) const
21 {
22     if (this->rows != other.rows || this->cols != other.cols || this->channels != other.channels) {
23         return false;
24     }
25     if (*this == other) {
26         return false;
27     } else {
28         return true;
29     }
30 }
```

判断两个矩阵是否相等和是否不等。

- 自定义类型转换

```
1 template<typename Tp>
2 template<typename Tp_>
3 Mat<Tp>::operator Mat<Tp_>() const
4 {
5     Mat<Tp_> res(this->rows, this->cols, this->channels);
6     for (int k = 0; k < res.getChannels(); k++) {
7         for (size_t i = 0; i < res.getRows(); i++) {
8             for (size_t j = 0; j < res.getCols(); j++) {
9                 res.at(i, j, k) = (Tp_) this->at(i, j, k);
10            }
11        }
12    }
13 }
```

```

12     }
13     return res;
14 }
```

将一矩阵对象强制转换为`Mat<Tp>`类型的矩阵，其中的操作是在函数内部创建一个`Mat<Tp>`对象，再将原来的`Mat<Tp>`对象的每一个元素的值强制转换为`Tp`类型再赋值给`Mat<Tp>`对象的相应位置。对于类型转换不直接将`data`中的数据强制转换为`Tp`类型，而是先创建一个`Mat<Tp>`的对象，再将`this->data`中的数据强制转换为`Tp`类型后再赋值给`Mat<Tp>`对象的原因是：如果原来的数据类型和转换后数据类型所占的字节数不一样，容易造成在访问矩阵元素时的数值不正确的问题。例：将`double`转为`int`。原本访问矩阵元素是8个字节为一个元素，转换后是4个字节为一个元素，显然转换后数值会不正确。

### 2.3.4 其他函数

- `roi()`

```

1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::roi(size_t rowIndex, size_t colIndex, size_t rows_, size_t cols_) const
3 {
4     if (rows_ >= this->rows || cols_ >= this->cols) {
5         cout << "Invalid size of sub-matrix. Return [nullptr]." << endl;
6         return (Mat<Tp> *) nullptr;
7     }
8     Mat<Tp> res;
9     res.rows = rows_;
10    res.cols = cols_;
11    res.steps = this->steps;
12    res.beginRowIndex = this->beginRowIndex + rowIndex;
13    res.beginColIndex = this->beginColIndex + colIndex;
14    res.dataRows = this->dataRows;
15    res.dataCols = this->dataCols;
16    res.channels = this->channels;
17    res.data = this->data;
18    res.refCntPtr = this->refCntPtr;
19    *(this->refCntPtr) += 1;
20    return res;
21 }
```

对矩阵截取感兴趣区域。为避免内存硬拷贝，该函数会设置`beginRowIndex`和`beginColIndex`为感兴趣区域(子矩阵)在`data`这个整体的矩阵中相对的行下标和列下标；设置`rows`和`cols`为感兴趣区域的行数和列数；设置`dataRows`和`dataCols`为`data`整体矩阵的行数和列数；最后再将`data`指向父矩阵`data`的首地址(即让父矩阵和子矩阵的`data`指向同一块内存)。由于这块`data`又被一个`Mat`对象引用，因此需要对这块`data`尾部的`*refCntPtr`进行加1的操作。

- `clone()`

```

1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::clone() const
3 {
4     Mat<Tp> res;
5     res.rows = this->rows;
6     res.cols = this->cols;
7     res.steps = this->cols;
8     res.beginRowIndex = 0;
9     res.beginColIndex = 0;
10    res.dataRows = this->rows;
11    res.dataCols = this->cols;
12    res.channels = this->channels;
```

```

13     res.data = util::allocate<Tp>(res.rows * res.channels, res.cols);
14     for (int k = 0; k < res.channels; k++) {
15         for (size_t i = 0; i < this->rows; i++) {
16             for (size_t j = 0; j < this->cols; j++) {
17                 res.at(i, j, k) = this->at(i, j, k);
18             }
19         }
20     }
21     res.setRefCntPtr((int *) (&(res.data[res.channels * res.dataRows - 1][res.dataCols])));
22     *(res.getRefCntPtr()) = 1;
23     return res;
24 }
```

对 `Mat` 对象进行克隆的操作。该函数会创建一个与 `this` 矩阵完全相同的矩阵，并且会对内存进行深拷贝，并将 `*refCntPtr` 初始化为1。

- `subMat()`

```

1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::subMat(size_t _rows_, size_t _cols_, size_t _beginRowIndex_, size_t
3 _beginColIndex_) const
4 {
5     return this->roi(_beginRowIndex_, _beginColIndex_, _rows_, _cols_).clone();
6 }
```

获取 `this` 矩阵的子矩阵。与 `roi()` 不同的是，该函数会对内存进行深拷贝(先利用 `roi()` 获取矩阵的局部再利用 `clone()` 创建一个新的与之相同的矩阵)。

- `transfer()`

```

1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::transfer()
3 {
4     util::Timer timer("Mat<Tp> transformation");
5     if (this->rows == this->cols) {
6 #if defined(_ENABLE_OMP)
7 #pragma omp parallel for num_threads((int)thread::hardware_concurrency())
8 #endif
9         for (int k = 0; k < this->channels; ++k) {
10             for (size_t i = 0; i < this->rows; ++i) {
11                 for (size_t j = 0; j < this->cols; ++j) {
12                     if (j > i) {
13                         Tp temp = this->at(i, j, k);
14                         this->at(i, j, k) = this->at(j, i, k);
15                         this->at(j, i, k) = temp;
16                     }
17                 }
18             }
19         }
20         return *this;
21     }
22     Mat<Tp> res(this->cols, this->rows, this->channels);
23     for (int k = 0; k < this->channels; ++k) {
24         for (size_t i = 0; i < this->rows; ++i) {
25             for (size_t j = 0; j < this->cols; ++j) {
26                 res.at(i, j, k) = this->at(j, i, k);
27             }
28         }
29     }
30     return res;
31 }
```

```

27         res.at(j, i, k) = this->at(i, j, k);
28     }
29 }
30
31 return res;
32 }
```

对矩阵进行转置操作。可开启**OMP**选项进行加速。如果当前矩阵的行数与列数相等，则直接在该矩阵上进行转置操作，将当前矩阵返回；如果当前矩阵行数与列数不等，则需要创建另外一个矩阵，并将该矩阵返回。

- `at()`, `rowPtr()`, `colPtr()`

```

1 template<typename Tp>
2 inline Tp &Mat<Tp>::at(size_t i, size_t j, int channelIndex) const
3 {
4     return *(&(this->data[channelIndex] * this->dataRows + this->beginRowIndex)[this-
5 >beginColIndex]) + j + i * this->steps);
6 }
7
8 template<typename Tp>
9 Tp *Mat<Tp>::rowPtr(size_t rowIndex, int channelIndex) const
10 {
11     return &(this->at(rowIndex, 0, channelIndex));
12 }
13
14 template<typename Tp>
15 Tp *Mat<Tp>::colPtr(size_t colIndex, int channelIndex) const
16 {
17     return &(this->at(0, colIndex, channelIndex));
18 }
```

`at()`与重载后的`operator()`所实现的功能相同，都用于访问矩阵特定位置上的元素。`rowPtr()`用于获取矩阵某一行首元素的地址；`colPtr()`用于获取矩阵某一列首元素的地址。

- `writeTo()`

```

1 template<typename Tp>
2 void Mat<Tp>::writeTo(const char *path) const
3 {
4     FILE *fp;
5     if ((fp = fopen(path, "w")) == nullptr) {
6         printf("Fail to open or create output file -> %s. Exit.", path);
7         exit(EXIT_FAILURE);
8     }
9     for (int k = 0; k < this->channels; ++k) {
10         fputs("Channel[", fp);
11         const char channelIndex = k + '0';
12         fputs((const char *)(&(channelIndex)), fp);
13         fputs("]:\r\n", fp);
14         for (size_t i = 0; i < this->rows; i++) {
15             for (size_t j = 0; j < this->cols; j++) {
16                 char *buffer = (char *) malloc(sizeof(char) * 32);
17                 sprintf(buffer, "%.1f", (float) this->at(i, j, k));
18                 fputs(buffer, fp);
19                 if (j < this->cols - 1) fputs(" ", fp);
20                 else fputs("\r\n", fp);
21             }
22         }
23     }
24 }
```

```
21         free(buffer);
22     }
23 }
24 fclose(fp);
25 }
26 }
```

将矩阵输入到指定路径的文件当中。

- `rand()`

```
1 template<typename Tp>
2 Mat<Tp> Mat<Tp>::rand(size_t rows_, size_t cols_, int channels_, Tp bottom, Tp top)
3 {
4     Mat<Tp> res(rows_, cols_, channels_);
5     default_random_engine engine(time(nullptr));
6     for (int k = 0; k < res.channels; k++) {
7         for (size_t i = 0; i < res.rows; i++) {
8             for (size_t j = 0; j < res.cols; j++) {
9                 uniform_real_distribution<double> generator(bottom, top);
10                res.at(i, j, k) = generator(engine);
11            }
12        }
13    }
14    return res;
15 }
```

生成一个指定行数和列数，以及通道数的矩阵。矩阵各个元素的值随机生成，元素的取值范围是`[bottom, top]`。

## 3. 测试结果及实验验证

测试实验中用到的矩阵数据(文本文件)来自于*Project 2*和*Project 3*中提供的文本数据。

### 3.1 内存管理

#### 3.1.1 拷贝构造函数赋值

```
1 Mat<int> mat_A("file/txt/mat-A-32.txt");
2 Mat<int> mat_B = mat_A.roi(2, 1, 7, 4);
3 mat_B.print();
4 Mat<int> mat_C = mat_B.roi(0, 2, 4, 2);
5 mat_C.print();
```

对`roi()`截取矩阵内容的正确性以及内存管理的安全性进行验证，结果如下图所示：

```
-- ~Mat(): refCnt: 3 --> 2
-- data = 0x55ecb4e513c0
Data type is [int]. Elements in channel[0]:
-----
| 62.0   11.0   65.0   26.0 |
| 46.0   16.0   25.0   29.0 |
| 68.0   97.0   42.0   63.0 |
| 82.0   84.0   47.0   26.0 |
| 17.0   10.0   26.0   85.0 |
| 40.0   7.0    74.0   77.0 |
| 6.0    62.0   14.0   42.0 |
-----
rows = 7. cols = 4
steps = 32
beginRowIndex = 2. beginColIndex = 1
dataRows = 32. dataCols = 32
channels = 1
refCntPtr = 0x55ecb4e51200. refCnt = 2
```

```
-- ~Mat(): refCnt: 4 --> 3
-- data = 0x55ecb4e513c0
Data type is [int]. Elements in channel[0]:
-----
| 65.0   26.0 |
| 25.0   29.0 |
| 42.0   63.0 |
| 47.0   26.0 |
-----
rows = 4. cols = 2
steps = 32
beginRowIndex = 2. beginColIndex = 3
dataRows = 32. dataCols = 32
channels = 1
refCntPtr = 0x55ecb4e51200. refCnt = 3
```

```
-- ~Mat(): refCnt: 3 --> 2
-- ~Mat(): refCnt: 2 --> 1
-- ~Mat(): refCnt: 1 --> 0
~Mat(): Free memory successfully.
```

图6. (a) mat\_B.print()

(b) mat\_C.print()

(c)main函数结束时输出

由图可知三个 Mat 对象的 data 都指向同一块内存。

图5.(a)(b) 中的第一行均为析构器被调用的原因是：在 roi() 内部会先创建一个矩阵 res，并将 this->data 的值赋予 res.data(两个矩阵公用 data 内存)，因此需要将 refCnt 加 1。在 roi() 尾部将 res return 出去后，roi() 内部的 res 的析构器会被调用，此时会对 res 的 refCnt 进行减 1 的操作(因为 roi() 内部的 res 将被销毁，即引用 data 内存的对象少一个)。图5.(c) 在 main 函数结束时，mat\_C, mat\_B, mat\_A 的析构器分别被执行。在执行前两个对象的析构器时，由于将 \*refCntPtr 减 1 后的值不为 0，表明还有 Mat 对象在引用其 data 的内存，因此不会将 data 的内存进行释放。在执行第三个对象的析构器时，将 \*refCntPtr 减 1 后的值为 0，表明这块 data 的内存以及没有 Mat 对象引用，此时将其内存释放。

对于多通道的矩阵的测试：

```
1 Mat<int> mat_A(32, 32, 2, {"file/txt/mat-A-32.txt", "file/txt/mat-B-32.txt"});
2 Mat<int> mat_B = mat_A.roi(4, 3, 6, 7);
3 mat_B.print();
4 Mat<int> mat_C = mat_B.roi(0, 1, 4, 5);
5 mat_C.print();
```

```
-- ~Mat(): refCnt: 3 --> 2
-- data = 0x55992325d200
Data type is [int]. Elements in channel[0]:
-----
| 42.0   63.0   69.0   38.0   43.0   33.0   74.0 |
| 47.0   26.0   58.0   9.0    56.0   4.0    34.0 |
| 26.0   85.0   72.0   13.0   42.0   71.0   91.0 |
| 74.0   77.0   31.0   70.0   91.0   56.0   2.0  |
| 14.0   42.0   49.0   8.0    52.0   0.0    37.0 |
| 57.0   58.0   23.0   71.0   74.0   66.0   57.0 |
-----
Data type is [int]. Elements in channel[1]:
-----
| 22.0   87.0   95.0   3.0    25.0   76.0   88.0 |
| 13.0   97.0   92.0   46.0   10.0   20.0   65.0 |
| 21.0   4.0    46.0   52.0   58.0   15.0   50.0 |
| 44.0   79.0   92.0   38.0   65.0   55.0   97.0 |
| 24.0   38.0   88.0   11.0   42.0   49.0   75.0 |
| 31.0   30.0   3.0    40.0   91.0   33.0   24.0 |
-----
rows = 6. cols = 7
steps = 32
beginRowIndex = 4. beginColIndex = 3
dataRows = 32. dataCols = 32
channels = 2
refCntPtr = 0x55992325d000. refCnt = 2
```

```
-- ~Mat(): refCnt: 4 --> 3
-- data = 0x55992325d200
Data type is [int]. Elements in channel[0]:
-----
| 63.0   69.0   38.0   43.0   33.0 |
| 26.0   58.0   9.0    56.0   4.0  |
| 85.0   72.0   13.0   42.0   71.0 |
| 77.0   31.0   70.0   91.0   56.0 |
-----
Data type is [int]. Elements in channel[1]:
-----
| 87.0   95.0   3.0    25.0   76.0 |
| 97.0   92.0   46.0   10.0   20.0 |
| 4.0    46.0   52.0   58.0   15.0 |
| 79.0   92.0   38.0   65.0   55.0 |
-----
rows = 4. cols = 5
steps = 32
beginRowIndex = 4. beginColIndex = 4
dataRows = 32. dataCols = 32
channels = 2
refCntPtr = 0x55992325d000. refCnt = 3
```

```
-- ~Mat(): refCnt: 3 --> 2
-- ~Mat(): refCnt: 2 --> 1
-- ~Mat(): refCnt: 1 --> 0
~Mat(): Free memory successfully.
```

图7. (a) mat\_B.print()

(b) mat\_C.print()

(c)main函数结束时输出

### 3.1.2 运算符 operator=() 赋值

```

1 Mat<int> mat_A("file/txt/mat-A-32.txt");
2 Mat<int> mat_B;
3 mat_B = mat_A.roi(2, 1, 7, 4);
4 mat_B.print();
5 Mat<int> mat_C;
6 mat_C = mat_B.roi(0, 2, 4, 2);
7 mat_C.print();

```

The figure consists of three terminal windows side-by-side. 
 (a) mat\_B.print(): Shows the original 7x4 matrix from file mat-A-32.txt. The output includes memory statistics (refCnt), operator() calls, and the matrix itself.

62.0	11.0	65.0	26.0
46.0	16.0	25.0	29.0
68.0	97.0	42.0	63.0
82.0	84.0	47.0	26.0
17.0	10.0	26.0	85.0
40.0	7.0	74.0	77.0
6.0	62.0	14.0	42.0

rows = 7. cols = 4  
steps = 32  
beginRowIndex = 2. beginColIndex = 1  
dataRows = 32. dataCols = 32  
channels = 1  
refCntPtr = 0x56293f79b200. refCnt = 2

 (b) mat\_C.print(): Shows the 4x2 ROI from mat\_B. The output includes memory statistics and the matrix.

65.0	26.0
25.0	29.0
42.0	63.0
47.0	26.0

rows = 4. cols = 2  
steps = 32  
beginRowIndex = 2. beginColIndex = 3  
dataRows = 32. dataCols = 32  
channels = 1  
refCntPtr = 0x56293f79b200. refCnt = 3

 (c) main函数结束时输出: Shows the destruction of mat\_B and mat\_C, and the successful deallocation of memory.

-- ~Mat(): refCnt: 3 --> 2  
-- ~Mat(): refCnt: 2 --> 1  
-- ~Mat(): refCnt: 1 --> 0  
~Mat(): Free memory successfully.

图8. (a) mat\_B.print()

(b) mat\_C.print()

(c)main函数结束时输出

运行结果及原因与通过拷贝构造函数进行“赋值”操作相同，不再赘述。

此外还需要测试当被赋值的对象原来已经引用了某块内存时的处理过程。

```

1 Mat<int> mat_A("file/txt/mat-A-32.txt");
2 Mat<int> mat_B("file/txt/mat-B-32.txt");
3 mat_A = mat_B;

```

The terminal output shows the assignment of mat\_B to mat\_A. It highlights the destruction of mat\_B and the creation of mat\_A, including the deallocation of shared memory.

-- operator(): refCnt: 1 --> 0  
operator(): Free memory successfully.  
operator(): refCnt: 1 --> 2  
-- ~Mat(): refCnt: 2 --> 1  
-- ~Mat(): refCnt: 1 --> 0  
~Mat(): Free memory successfully.

图9. 输出结果

当 `operator=()` 被调用时(代码块第3行)，由于 `mat_A` 的 `data` 已经指向了一块内存(`file/txt/mat-A-32.txt`)，所以先对该 `data` 尾部的 `refCnt` 减1。此时 `refCnt` 为0，代表没有 `Mat` 对象引用这块内存，所以需要释放掉以防止内存泄漏。

```

1 Mat<int> mat_A("file/txt/mat-A-32.txt");
2 Mat<int> mat_B = mat_A;
3 Mat<int> mat_C("file/txt/mat-B-32.txt");
4 mat_A = mat_C;

```

```

-- operator=(): refCnt: 2 --> 1
operator=(): refCnt: 1 --> 2
-- ~Mat(): refCnt: 2 --> 1
-- ~Mat(): refCnt: 1 --> 0
~Mat(): Free memory successfully.
-- ~Mat(): refCnt: 1 --> 0
~Mat(): Free memory successfully.

```

图10. 输出结果

在这个例子中当代码块第4行的 `operator=()` 被调用时，由于 `mat_A.data` 和 `mat_B.data` 都指向同一块内存，即使 `mat_A` 被 `mat_C` 赋值，不再引用这块内存后，仍然有 `mat_B` 在引用这块内存，所以这块内存并不会在 `operator=()` 中被释放。在图9中，第一句 `refCnt` 减1代表 `mat_A` 不再引用 `mat_B.data` 指向的内存，这块内存的 `refCnt` 减1；第二句 `refCnt` 加1代表 `mat_A.data` 又指向了 `mat_C.data` 指向的内存，`mat_C.data` 的 `refCnt` 加1。

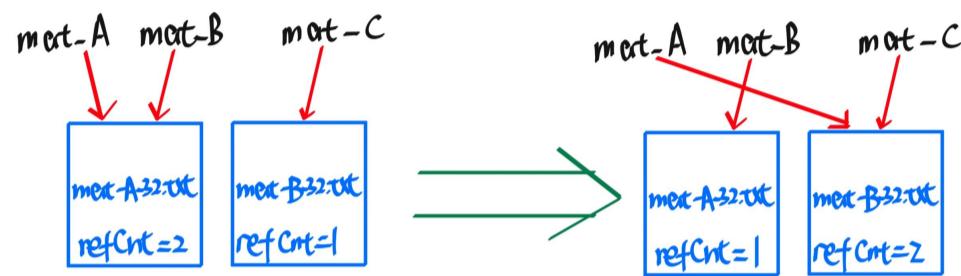


图11. 赋值过程示意图

## 3.2 矩阵运算

本地测试环境：

*OS: Ubuntu 20.04.1 LTS on Windows 10 x86-64*

*CPU: AMD Ryzen 9 5900HS with Radeon Graphics @ 3.30 GHz*

*RAM: 16.0 GB (15.4 GB available)*

*GPU: NVIDIA GeForce RTX 3060 Laptop GPU*

远程测试环境：

*OS: Linux*

*CPU Architecture: aarch64 64-bit*

*RAM: 2.9 GB*

### 3.3.1 加减法

以下测试均在关闭编译器O3优化的情况下测试

```

1 Mat<int> mat_A_int = Mat<int>::rand(4096, 4096, 1, -100, 100);
2 Mat<int> mat_B_int = Mat<int>::rand(4096, 4096, 1, -100, 100);
3 for (int i = 0; i < 5; i++) mat_A_int + mat_B_int;
4 Mat<float> mat_A_float = Mat<float>::rand(4096, 4096, 1, -100, 100);
5 Mat<float> mat_B_float = Mat<float>::rand(4096, 4096, 1, -100, 100);
6 for (int i = 0; i < 5; i++) mat_A_float + mat_B_float;

```

-- Procedure Mat<int> Addition consumed 168.55ms.	-- Procedure Mat<int> Addition consumed 42.1387ms.
-- Procedure Mat<int> Addition consumed 170.492ms.	-- Procedure Mat<int> Addition consumed 41.9177ms.
-- Procedure Mat<int> Addition consumed 176.372ms.	-- Procedure Mat<int> Addition consumed 42.0354ms.
-- Procedure Mat<int> Addition consumed 172.201ms.	-- Procedure Mat<int> Addition consumed 41.9157ms.
-- Procedure Mat<int> Addition consumed 171.554ms.	-- Procedure Mat<int> Addition consumed 42.2382ms.
-- Procedure Mat<float> Addition consumed 172.944ms.	-- Procedure Mat<float> Addition consumed 44.017ms.
-- Procedure Mat<float> Addition consumed 174.077ms.	-- Procedure Mat<float> Addition consumed 43.9537ms.
-- Procedure Mat<float> Addition consumed 180.868ms.	-- Procedure Mat<float> Addition consumed 46.027ms.
-- Procedure Mat<float> Addition consumed 173.783ms.	-- Procedure Mat<float> Addition consumed 44.5162ms.
-- Procedure Mat<float> Addition consumed 173.819ms.	-- Procedure Mat<float> Addition consumed 44.91ms.

图12. (a) x86平台下未开启AVX2指令集加速

-- Procedure Mat<int> Addition consumed 390.349ms.
-- Procedure Mat<int> Addition consumed 388.888ms.
-- Procedure Mat<int> Addition consumed 389.668ms.
-- Procedure Mat<int> Addition consumed 389.203ms.
-- Procedure Mat<int> Addition consumed 390.637ms.
-- Procedure Mat<float> Addition consumed 401.235ms.
-- Procedure Mat<float> Addition consumed 401.211ms.
-- Procedure Mat<float> Addition consumed 401.22ms.
-- Procedure Mat<float> Addition consumed 401.593ms.
-- Procedure Mat<float> Addition consumed 402.702ms.

(b) x86平台下开启AVX2指令集加速

-- Procedure Mat<int> Addition consumed 183.428ms.
-- Procedure Mat<int> Addition consumed 183.138ms.
-- Procedure Mat<int> Addition consumed 184.547ms.
-- Procedure Mat<int> Addition consumed 183.469ms.
-- Procedure Mat<int> Addition consumed 183.529ms.
-- Procedure Mat<float> Addition consumed 191.435ms.
-- Procedure Mat<float> Addition consumed 192.067ms.
-- Procedure Mat<float> Addition consumed 190.842ms.
-- Procedure Mat<float> Addition consumed 191.132ms.
-- Procedure Mat<float> Addition consumed 190.43ms.

图13. (a) ARM平台下未开启NEON指令集加速

(b) ARM平台下开启NEON指令集加速

如图7和图8中的结果所示，对于不同数据类型的矩阵，程序调用了对应类型的operator+()函数来进行计算(例子中体现为Mat<int> addition和Mat<float> addition)。在开启SIMD加速选项后，不论是x86还是arm平台，矩阵计算的效率都得到明显提升。

与在本地的个人电脑相比，在远程服务器上进行的测试效率明显下降。其主要原因是远程服务器的配置较低。通过命令查询，远程服务器的CPU只有两个核心，而本地测试环境有8个核心。因此远程运算效率要明显低于本地。

```
[root@ecs001-0021-0011 build]# cat /proc/cpuinfo
processor : 0
BogoMIPS : 200.00
Features : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhpm cpuid asimdrdm jscvt fcma dcpop asimdd
p asimdfhm
CPU implementer : 0x48
CPU architecture: 8
CPU variant : 0x1
CPU part : 0xd01
CPU revision : 0

processor : 1
BogoMIPS : 200.00
Features : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhpm cpuid asimdrdm jscvt fcma dcpop asimdd
p asimdfhm
CPU implementer : 0x48
CPU architecture: 8
CPU variant : 0x1
CPU part : 0xd01
CPU revision : 0
```



图14. (a) 远程服务器CPU信息

(b) 本地CPU信息

### 3.3.2 乘法

以下测试均在开启编译器O3优化的情况下测试

```
1 Mat<int> mat_A_int = Mat<int>::rand(2048, 2048, 1, -100, 100);
2 Mat<int> mat_B_int = Mat<int>::rand(2048, 2048, 1, -100, 100);
3 for (int i = 0; i < 5; ++i) mat_A_int * mat_B_int;
4 Mat<float> mat_A_float_2channel(2048, 2048, 2, {"file/txt/mat-A-2048.txt", "file/txt/mat-B-2048.txt"});
5 Mat<float> mat_B_float_2channel(2048, 2048, 2, {"file/txt/mat-B-2048.txt", "file/txt/mat-A-2048.txt"});
6 for (int i = 0; i < 5; ++i) mat_A_float_2channel * mat_B_float_2channel;
```

-- Procedure Mat<Tp> Multiplication-Threads consumed 236.711ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 257.395ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 307.199ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 231.758ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 310.552ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 491.226ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 534.493ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 536.262ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 497.111ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 477.474ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 273.301ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 2286.58ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 2302.66ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 2253.99ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 2302.51ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 2287.94ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 3740.14ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 3889.32ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 4147.12ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 3922.31ms.	-- Procedure Mat<Tp> Multiplication-Threads consumed 3977.4ms.
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--

(a) x86\_64平台下矩阵乘法耗时

(b) ARM平台下矩阵乘法耗时

相较于x86\_64平台，ARM平台下矩阵乘法运算速度较为缓慢的原因同矩阵加法。

## 4. 困难及解决

### 4.1 模板类的使用

#### 4.1.1 头文件的编写

在刚开始使用模板类时，会产生`undefined reference to [function]`的链接错误。在查阅相关资料后<sup>4</sup>，发现该错误的原因是：编译器在编译C++代码时，是一个一个以`*.cpp`为一个基本单元进行编译的。根据代码找到`#include`声明，翻译代码产生`*.o`文件。对于在`#include`文件中却少定义的部分，如果在其他源文件中进行了妥善的定义，则会在链接的过程中产生引用关系。对于模板类和模板函数，由于在编译的时候并不知道具体的类型参数是什么，实际上没有任何有用的内容存在于`.o`文件当中。而在使用模板类的地方指定了类型参数，编译器这才开始根据模板代码产生有用的`*.o`编码，可是这些内容放在了使用模板的代码产生的`*.o`文件当中。如果编使用模板代码的时候，通过`#include`包含“看不到”模板的实现代码，这些所有的缺失，到链接阶段就无法完成。

该问题的解决办法是：在头文件中就完成函数的声明和定义；或者将`*.cpp`文件的后缀修改为`*.tpp`，并在头文件的尾部将`*.tpp`文件`#include`；或者直接将函数的所有声明和定义都放在头文件当中去。

#### 4.1.2 模板友元函数

在声明模板友元函数时，如果类型参数直接使用`Mat`模板类的类型参数，在编译时会出现`friend declaration [function] declares a non-template function`的错误。这是因为友元函数不属于类成员，只是相当于一个可以访问类成员的类外部定义的普通函数。其本质上认识一个类外部的“普通”函数。因此在类内部声明时需要像在类外部声明模板函数那样声明模板友元函数<sup>5</sup>，且模板友元函数的类型参数名称不能与类模板的类型参数相同。例：

```

1 template<typename Tp>
2 class Mat {
3     template<typename Tp_>
4     friend ostream &operator<<(ostream &os, const Mat<Tp_> &mat);
5 }
```

### 4.2 运算结果误差

在进行矩阵乘法运算结果正确性的实验时，发现不论用什么方法，矩阵乘法的运算结果(主要是多线程)与标准(朴素矩阵乘法)运算结果相比，总有一些误差。虽然误差不是特别大，测试过程( $2048 \times 2048$ 矩阵乘法)中平均误差为`0.686865`，但是这种问题在Project 3同样采用了多线程和SIMD加速的运算过程中却没有产生。目前猜测的可能原因是代码中有些细节的地方有错误，但如果是这种错误的话，运算结果( $2048 \times 2048$ )的误差应该会比较大(因为矩阵规模较大，误差会累积)。另外一种原因是由于本次矩阵乘法中的多线程没有使用互斥锁的手段保证线程同步，导致在不同线程在处理数据的时候可能会使得数据产生一些误差。

该问题在Project报告提交时仍未解决。

## 5. 总结及反思

---

C++作为一门可以直接在硬件层面上操作内存的语言，内存管理一直是使用C++的过程中最令程序员头疼的问题。本次Project中，个人认为最主要要解决的问题是矩阵类的内存管理问题。由于在本次Project中涉及到了矩阵内存浅拷贝的问题，因此需要对data指向的内存何时需要被释放的逻辑进行严谨、缜密的思考与测试。通过一段时间的思考以及对OpenCV进行适当的参考后，该项目实现了较为安全的内存管理。从本次Project学习到的对于内存管理的思考模式，在我今后其他科目的学习会起到一定的启发。

## 参考资料

---

1. OpenCV: OpenCV Tutorials [←](#)
2. ShiqiYu/libfacedetection: An open source library for face detection in images. The face detection speed can reach 1000FPS. (github.com) [←](#)
3. OpenCV: cv::Mat attribute `steps` [←](#)
4. c++ - Why can templates only be implemented in the header file? - Stack Overflow [←](#)
5. c++ - overloading friend operator<< for template class - Stack Overflow [←](#)