CS205 C/C++ Program Design Project3

> Name:章志轩
>
> SID:12010526

> Time:2023/04/20

Abstract: 工欲善其事必先利其器。设计好的数据结构将有效提高算法的运行效率以及降低代码的复杂度。在卷积神经网络（CNNs）中数据块是定义保存数据的格式，所有输入、输出以及卷积核都遵循着相同的存储方法。如何设计数据块，如何设计高效实用的数据块是本次项目的主要内容，而对内存的维护则是重点中的重点，将在本次项目着重关注。

I. 项目介绍

在上个项目中我已经实现了卷积操作(链接：https://github.com/15775011722/CS205-Sustech-Cpp/blob/main/project/project3/CS205%20CC%2B%2B%20Program%20Design%20Project3.md)，当时我设计了一个名为Tensor的结构体作为data blob用于存储数据：

```
typedef struct {
    float *data; // 数据指针
    struct {
        size_t n;
        size_t channel;
        size_t width;
        size_t height;
    } shape;
} Tensor;
```

由于当时使用的是确定类型`float`，所以数据块设计很简单。此次设计的数据块是个类，要求支持多类型，至少包括`unsigned char`, `short`, `int`, `float`, `double`, 且要求重载至少包含`=`, `==`, `+`, `-`, `*`在内的运算符，如此还要便于使用。为此我考虑过在project2中使用过的`union`联合体，当时使用`union`存储数据实现多类型的向量点乘。但根据当时的经验，尽管union可以自主选择保存的数据类型，但在使用上并不方便。因为在每次运算时都需要重新判断保存的类型是哪一种，假设有4种可能的类型，那么两个数据相加就会有4*4=16种可能，这对于代码复杂度以及实现上多不方便。我也考虑过使用`void`指针，以此兼容不同类型数据，但由于我并不熟悉它的用法，最后还是选择放弃。最终我选择使用C++的高级特性：模板类。它能让用户在创建时选择输入的数据类型，而类针对输入的类型执行对应的运算（类似于java的泛型）。我将其命名为`Mat`，是matrix的缩写，是我模仿`opencv::Mat`类设计的数据块。

```
template <typename _Tp>
class Mat
{
private:
    shared_ptr<_Tp> data;
    long long rows;
    long long cols;
    long long step;
    int elementSize;
    long long channels;
```

```
    long long size;

public:
    ...
}
```

接下来我会讲解我的具体实现。

II. 环境配置&语言

系统环境：Win10系统下的WSL装载的Ubuntu系统，AMD 锐龙 5 4600U处理器；

IDE：VScode-x64-1.75.1

语言：C++（g++编译器，版本C++11）

III. 代码设计与实现

变量设计

代码如上述所示，每一个变量的含义如下：

`data`：数据块保存的数据的头指针

`rows`：数据块的行数

`cols`：数据块的列数

`step`：数据块的步长，由于数据块在此处固定是二维矩阵，故step就是一行大小

`elementSize`：每一个元素所占的byte

`channels`：数据块的通道数

`size`：数据总量（单位是个，不是byte）

避免内存重复释放

当多个对象指向同一地址，又各自有释放操作(包括主动释放(堆)与编译器帮忙释放(栈))，就会导致重复释放问题。为避免重复释放且减轻用户的使用难度，我使用智能指针`shared_ptr`来帮助管理数据内存。`shared_ptr`是`<memory>`库中的类，是属于c++的新特性。它有内置计数器能够计算所有指向同一地址的智能指针数量，当对象被释放时它会判断是否还有其它智能指针在使用这个地址，如果没有就将指针指向的地址一并释放，否则就保留。由于项目要求使用软拷贝，故会存在多个对象指向同一地址，此时用智能指针进行管理就会很方便。也因此，在我的析构函数中就不需要执行`delete`操作避免误删。

构造器

为方便类的使用，我设计五种构造器供用户选择，它们是：

```
Mat();
Mat(const long long rows, const long long cols, const long long channels);
Mat(const long long rows, const long long cols, const long long channels, _Tp
*data);
Mat(const Mat& m);
// 强转设计构造器
template<typename T2>
Mat(const Mat<T2>& m);
```

需要注意的是第三个构造器并不完善，它能接受的指针只有保存在堆中的指针数组(`new`)，要是在栈中就会有重复释放。为此最安全的做法是使用第二个构造器，然后使用`getData().get()`获得指针，然后进行赋值。更具体的说，就是第三个构造器有bug，给矩阵赋值的操作还是交给`setData()`(测试代码`main.cpp`中使用的方法)函数或者`getData().get()`(操作符重载时使用的方法)然后进行赋值较为安全。

错误检查

以及`setData`等函数辅助多方面创建对象。需要注意的是，为避免指针越界，在赋值指针时会对非法的修改进行一次错误检查`handleError(ILLEGAL_MODIFY)`。它是我自定义的函数，实现在`ErrorMsg.hpp`中。该头文件定义了我自己设置的错误类型，当检测到对应的错误，就会打印信息，并强行终止程序`exit()`。

```
template<typename _Tp>
void Mat<_Tp>::setData(_Tp *data, long long length)
{
    if(size != length)
    {
        handleError(ILLEGAL_MODIFY);
    }
    this->data = shared_ptr<_Tp>(data);
}
```

```
// ErrorMsg.hpp
#pragma once
#include <iostream>
#include <stdlib.h>
using namespace std;

#define SHAPE_NOT_MATCH 0
#define ILLEGAL_MODIFY 1

#define ERROR_DESCRIPTION(type, description) \
    case type: \
        return description; \

static
string errorDescription(int type)
{
    switch (type)
    {
    ERROR_DESCRIPTION(SHAPE_NOT_MATCH, "矩阵大小不匹配!");
```

```
    ERROR_DESCRIPTION(ILLEGAL_MODIFY, "非法的结构修改!");
    default:
        return "未知错误!";
    }
}

static
void handleError(int type)
{
    cout << "Error: " << errorDescription(type) << endl;
    exit(1);
}
```

如此就能快速定位错误的位置(而不是让错误的程序运行下去得到未知的结果)。

强转构造器

我对强转类型的思考起源于拷贝构造函数。拷贝构造函数十分简单，只需要指针指过去就可以完成。但这又存在一个问题，当我想将数据块中的存储格式从int转换为float时需要先创建一个白板，然后挨个赋值。而且在后续进行复杂运算时取操作符`=`时也可能出现两边数据类型不兼容导致报错。这样的类不好用且不实用。为此我实现了强转构造器，使得编译器能根据数据类型自动帮我进行数据的转换。代码如下：

```
template<typename T2>
Mat(const Mat<T2>& m) {
    this->rows = m.getRows();
    this->cols = m.getCols();
    this->step = m.getStep();
    this->size = m.getSize();
    this->channels = m.getChannels();
    this->elementSize = sizeof(_Tp);
    this->data = shared_ptr<_Tp>(new _Tp[this->size]);
    T2 *tmp = m.getData().get();
    _Tp *tmp2 = this->data.get();
    for (long long i = 0; i < m.getSize(); i++)
    {
        tmp2[i] = (_Tp)(tmp[i]);
    }
}
```

如此我就可以灵活的对数据块(Mat对象)进行操作。

```
Mat<short> t1;
Mat<int> t2(1, 1, 1);
Mat<float> t4(t2);
Mat<double> t5(t4);
```

操作符重载

重载操作符，我考虑到与强转构造器相同的情况，为此我使用友元函数实现了不同数据类型之间的`+-*==!=`，以及成员函数实现的`=`。

=操作符重载

前面提到的拷贝构造函数，实际使用的就是重载后的`=`操作符。

```
template<typename _Tp>
Mat<_Tp>& Mat<_Tp>::operator = (const Mat& m)
{
    if (this->data == m.data)
    {
        return *this;
    }
    this->rows = m.getRows();
    this->cols = m.getCols();
    this->step = m.getStep();
    this->channels = m.getChannels();
    this->size = m.getSize();
    this->elementSize = m.getElementSize();
    this->data = shared_ptr<_Tp>(m.getData());
    return *this;
}
```

实现类型自动选择

我们知道，当一个int类型变量+-*or/一个float类型变量，那么结果会是float类型变量。为了实现类型的自动选择，我设计了一个`TypeSel.hpp`。其中采用了`declval`表达式生成对应类型值，并将得到的数据类型传给`Type`。这样当我使用`Result<T1, T2>`时就会获得自动转换的类型，这样就满足不同类型运算的输出结果类型自动选择。

```
// TypeSel.hpp
#include <iostream>
using namespace std;

template<typename T1,typename T2>
struct ResultT
{
    using Type = decltype(std::declval<T1>() + std::declval<T2>());
};

template<typename T1, typename T2>
using Result = typename ResultT<T1, T2>::Type;
```

+-操作符重载

使用友元函数重载，会进行一次错误检查`handleError(SHAPE_NOT_MATCH)`检查进行运算的两个变量是否形状符合要求，不符合就报错并终止进程。同时此处本来想使用SIMD，但在尝试后发现由于数据类型的不确定，我很难确定一次取多少数据，且两个对象的运算需要同时考虑各自的类型保存格式(浮点与整型)。所以考虑再三，认为如此操作代码复杂度太高，无奈放弃。但由于数据保存在内存中是连续的，因此还可以采取笨方法每次循环取8个，分别相加。如此做到加快运算(包括O3，omp等加速方法)。

```
template<typename T1, typename T2>
Mat<Result<T1, T2>>& operator + (const Mat<T1>& m1, const Mat<T2>& m2)
{
    if (m1.rows != m2.rows || m1.cols != m2.cols || m1.channels != m2.channels)
    {
        handleError(SHAPE_NOT_MATCH);
    }
    Mat<Result<T1, T2>> *matrix = new Mat<Result<T1, T2>>(m1.rows, m1.cols,
m1.channels);
    Result<T1, T2> *sum = matrix->data.get();
    T1 *p1 = m1.data.get();
    T2 *p2 = m2.data.get();
    long long remainder = (matrix->size >> 3) << 3; // 除8再乘8
    if (remainder != 0) {
        // #pragma omp parallel for schedule(dynamic)
        for (long long i = 0; i < matrix->size; i+=8)
        {
            sum[i] = (Result<T1, T2>)(p1[i] + p2[i]);
            sum[i+1] = (Result<T1, T2>)(p1[i+1] + p2[i+1]);
            sum[i+2] = (Result<T1, T2>)(p1[i+2] + p2[i+2]);
            sum[i+3] = (Result<T1, T2>)(p1[i+3] + p2[i+3]);
            sum[i+4] = (Result<T1, T2>)(p1[i+4] + p2[i+4]);
            sum[i+5] = (Result<T1, T2>)(p1[i+5] + p2[i+5]);
            sum[i+6] = (Result<T1, T2>)(p1[i+6] + p2[i+6]);
            sum[i+7] = (Result<T1, T2>)(p1[i+7] + p2[i+7]);
        }
    }
    for (long long i = remainder; i < matrix->size; i++)
    {
        sum[i] = (Result<T1, T2>)(p1[i] + p2[i]);
    }
    return *matrix;
}
```

(减法与加法类似，符号改为-即可。)

***操作符重载**

data在内存中保存的格式是(channels, rows, cols)。这与Opencv中的Mat保存形式不同(不同通道的放在一起)。实际上opencv::Mat的保存格式更加符合数据定义，因为通道是作为元素的属性而不是维度。然而在deep learning中卷积操作是不同卷积核分开卷积的(最后再加在一起)，因此为保证取数据时地址的连贯性以及提高计算效率，将不同通道分开保存是更有利的，也就是将channels也视为一个维度。如此进行矩阵乘法将更容易设计与提速。以下是代码实现：

```
template<typename T1, typename T2>
Mat<Result<T1, T2>>& operator * (const Mat<T1>& m1, const Mat<T2>& m2)
{
    if (m1.cols != m2.rows || m1.channels != m2.channels)
    {
        handleError(SHAPE_NOT_MATCH);
    }
    Mat<Result<T1, T2>> *matrix = new Mat<Result<T1, T2>>(m1.rows, m2.cols,
m1.channels);
    Result<T1, T2> (*sum)[matrix->rows][matrix->cols]
        = (Result<T1, T2> (*)[matrix->rows][matrix->cols])matrix->data.get();
    T1 (*p1)[m1.rows][m1.cols] = (T1 (*)[m1.rows][m1.cols])m1.data.get();
    T2 (*p2)[m2.rows][m2.cols] = (T2 (*)[m2.rows][m2.cols])m2.data.get();
    long long remainder = (matrix->cols >> 3) << 3;  // 除8再乘8
    // #pragma omp parallel for schedule(dynamic)
    for (long long c = 0; c < matrix->channels; c++)
    {
        for (long long i = 0; i < m1.rows; i++)
        {
            for (long long j = 0; j < m1.cols; j++)
            {
                if (remainder != 0) {
                    for (long long k = 0; k < m2.cols; k+=8)
                    {
                        sum[c][i][k] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j][k]);
                        sum[c][i][k+1] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+1]);
                        sum[c][i][k+2] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+2]);
                        sum[c][i][k+3] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+3]);
                        sum[c][i][k+4] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+4]);
                        sum[c][i][k+5] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+5]);
                        sum[c][i][k+6] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+6]);
                        sum[c][i][k+7] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j]
[k+7]);
                    }
                }
                for (long long k = remainder; k < matrix->cols; k++)
                {
                    sum[c][i][k] += (Result<T1, T2>)(p1[c][i][j] * p2[c][j][k]);
                }
            }
        }
    }
    return *matrix;
}
```

==与!=操作符重载

关系运算符的返回值是布尔类型，实现起来也相对容易。这里同样实现了不同数据类型的关系运算符重载，并且只要数据类型不同，那么两者就不相等。只有当数据类型相同，形状相同，且数据的值(不是地址)相同，两者才算相等。代码如下：

```
template<typename T1, typename T2>
bool operator == (const Mat<T1>& m1, const Mat<T2>& m2)
{
    bool sameType = is_same<T1, T2>::value;
    bool sameShape = (m1.rows==m2.rows) && (m1.cols==m2.cols) &&
(m1.channels==m2.channels);
    if (!(sameType && sameShape))
        return false;
    T1 *p1 = m1.data.get();
    T2 *p2 = m2.data.get();
    for (long long i = 0; i < m1.size; i++)
    {
        if (p1[i] != p2[i])
            return false;
    }
    return true;
}

template<typename T1, typename T2>
bool operator != (const Mat<T1>& m1, const Mat<T2>& m2)
{
    return !(m1 == m2);
}
```

IV. 使用

`main.cpp`是测试代码，包含头文件`<Mat.hpp>`，而`Mat.hpp`就是我实现的`Mat`类也就是数据块类。由于模板类设计与实现分文件写会存在些问题(编译器不知道模板类用什么类型的参数套用模板)，会导致编译无法通过。所以被逼无奈我将实现函数一并写在头文件中(尽管这样是不好的)。而在`Mat.hpp`中又引用了另外两个头文件`<ErrorMsg.hpp>`和`<TypeSel.hpp>`，它们都是简短的代码，且实现专门的功能，是我在解决问题过程中发现的优雅的方法。因此最终只有一个文件需要编译成.o文件，也就是说代码执行方法是`g++ main.cpp &&./a.out`即可。同时为方便展示数据块，我也实现了display()函数，用来在终端打印数据块。

由于没领到ARM架构的开发板，所以我只在本机进行测试。不同环境运行结果可能不同。

V. 总结&缺陷&收获的经验

内存管理是门学问，而我还只是入门。好的内存管理可以增加程序的鲁棒性，提高运行效率以及合理有效的利用有限的内存空间。然而我的内存管理并没有做到面面俱到，就比如构造器入参，如果直接输入数据指针就会产生错误，要是调用set函数就不会。以及对于自定义错误信息的打印，一开始想法很好，希望能检测所有类型的错误，至少能检测非法的使用以及内存泄漏。但最后由于时间上的问题只完成了两条。同样是由用户输入可能导致出错的是模板类的传入类型。由于模板类的开放性，用户可能会输入奇怪的数据类型比如string。因为我重载的操作符实际上并不能满足除整型与浮点型以外的数据，所以会出错。对此的改进想

法是用前面说的ErrorMsg.hpp去处理错误，或者对模板类Mat再进行一次封装，模仿opencv::Mat的输入结构通过传参(8U_FC等)限制用户的输入。

在本次项目学习中，我探索了多方面内容，对智能指针、模板类、数据兼容等都了解了很多。其中智能指针十分好用，但要注意不要与普通指针混合使用。模板类的书写格式是本次项目我遇到的最大的麻烦(比处理内存问题都多)，为实现不同类型数据的运算、赋值等操作，我修改了许多版本的代码，比如成员函数的重载方法我发现不行，必须得用友元函数才能识别T1类型和T2类型。但赋值操作又只能是成员函数，为此我又设计了强转。最后就是又发现模板类的申明与定义只能写在一起，分文件会导致编译器无法确定模板使用的类型。我也算是对模板又爱又恨了。

总结，本次项目学会了内存管理的方法(智能指针、小心重载函数)、模板类的复杂使用(多类型操作)等内容，在代码中还存在没处理好的内存问题，用户限制输入问题以及没完成的错误处理。同时没条件且没时间对不同系统进行测试，这将影响我代码的鲁棒性。

感谢于老师的教诲与各位学助的帮助。

VI. 源码

https://github.com/15775011722/CS205-Sustech-Cpp/tree/main/project/project4/code