Contents

1 Requirement Analysis

The brief requirement of this project is to design a class to store different types of data blocks involved in the CNN process, such as image data blobs, kernel data blobs, etc. This class needs to contain basic information about the data blobs, have some user-friendly functions, implement some operations in the CNN process and manage its memory properly.

1.1 The use of Class Template

The class supports storing different types of data. One way to achieve this is by using typedef in the header file to create classes with different aliases for each data type, but this makes the code long and if you need to change one place code may cause all the rest of the class code to be changed. Moreover, this does not meet the requirement of writing 'a class'. For classes that store different types of data, all operations are the same except for the type of data stored, so this program uses a C++ template class to achieve this function.

The template allows a function or class to be designed to work seamlessly with a diverse range of data types, without the need for tedious and repetitive rewrites for each type. By leveraging templates, C++ code can be made more concise, modular, and easier to maintain, while also boosting its flexibility and versatility.

Class templates are similar to the blueprints that guide the C++ compiler and are C++ compiler directives that only explain how to generate class and member function definitions, and when we need to use this one class, we can instantiate this one class implicitly or explicitly. Since class templates are not classes and functions, you cannot put template members in a separate implementation file, so this project integrates all template information in the header file.

```
template<typename T>
class DataBlobs
```

1.2 Private of the Class

The basic information of a data block is important and needs to be protected, so we put the basic information of a data block in the private area of the class.

Based on our knowledge that CNN is widely used as a deep learning tool in image processing, for image data as an example, we designed rows, columns, channels, and data as the basic parameters. The rows, columns, and channels are stored in the size_t data type. size_t is an unsigned integer, which ensures the correctness of our data to a certain extent. At the same time, the range of size_t can reach the maximum length of any object, making the data blob class able to accommodate data blocks with larger volumes of data.

Moreover, in order to improve memory utilization, we use a soft copy to deal with the copy of the same piece of data. Therefore, for proper memory management, we design a variable named ref_count to keep track of the number of times this data is referenced, which prevents us from double freeing the data in the destructor. ref_count is a size_t pointer because ref_count's address needs to be recorded and referenced by different data. The non-negativity of size_t ensures that the value of ref_count is correct to a certain extent. Also, the range of size_t can reach the maximum length of any object, allowing the same data to be referenced by many other data blobs.

In a program, we may create a very large number of data blobs for various reasons. To manage all the data blobs we create more precisely, we add the variable DataBlobs_num, which has a static data type of size_t, to keep track of the number of data blobs. The static data type was chosen because the DataBlobs_num variable is not unique to any one or a group of objects of this class, it is a property of this class and is owned jointly by all objects of this class.

```
private:
    size_t row;
    size_t col;
    size_t channel;
    T* data;
    size_t* ref_count;
    inline static size_t DataBlobs_num = 0;
```

1.3 Public of the Class

To make this data blob more user-friendly, we overload many operators for it, making it as easy to manipulate as numerical operations. Also, to make this data blob more convenient to use in the CNN process, we have added many functions for CNN-related operations to it.

1.3.1 Constructor

A total of 5 different constructors are written.

The first one is the default constructor and does not require any parameter input. It will create an empty Data Blob. The second one requires the input of the three most basic data parameters, row, column, and channel, for which the constructor will request a data space of this size and set all the data inside to zero. The third one requires rows, columns, channels, data, and matching ref_count (if this data has not been referenced you can pass NULL), these parameters complete the initialization of a data block. The fourth one is copy constructor. The last one requires an input of a file name (usually the file name of the image by default), which is the data block that generates file data. (Using OpenCV's imread)

```
template<typename T>
DataBlobs();
DataBlobs(const size_t row, const size_t col, const size_t channel);
DataBlobs(const size_t row, const size_t col, const size_t channel, T* data, size_t* ref_count);
DataBlobs(const DataBlobs& blob);
DataBlobs(const string filename);
```

1.3.2 Overloading operators

Many operators are overloaded.

I. Overloading = operator.

Copy a data block to another data block.

```
DataBlobs& operator=(const DataBlobs& blob);
```

II. Overloading +, -, *, / operators

A total of 5 different +, -, * operator overloading functions are written. 2 different / operator overloading are written.

Using the + operator as an example. The first function is a data blob plus a number, and the operation is to add this number to each element of the data blob. The second is a data blob plus another data block, and the operation is to add each number in this data blob to each number in the other data block. The next two functions are implementations of += in the above two situations. The last one is the friend function, which makes it possible to do the operation of a number plus a data blob. The - and / operator overloading is almost the same as the + one.

Considering the data blob as a matrix, there is no definition of the / operation between two matrices. Thus, no / between two data blobs. (reducing 3 functions)

```
//+ operator as an example
template<typename T>
DataBlobs operator+(T num) const;
DataBlobs operator+(const DataBlobs& blob) const;
DataBlobs& operator+=(T num);
DataBlobs& operator+=(const DataBlobs& blob);
template<typename Tp>
friend DataBlobs operator+(Tp num, const DataBlobs<Tp>& blob);
```

III. Overloading ++, -- operators

The increment of a matrix is to add one to each element of the matrix, and the subtraction is to subtract one from each element.

```
//++ operator as an example
//Prefix
DataBlobs& operator++();
//Suffix
DataBlobs operator++(int);
```

IV. Overloading ==, != operator

Compare the rows, columns, channels, and every element in the data between the two data blocks to see if they are equal.

```
bool operator==(const DataBlobs& blob);
bool operator!=(const DataBlobs& blob);
```

V. Overloading << operator

Outputs the data of a data block in the form of a square, according to rows and columns to correspond to the position.

```
template<typename Tp>
friend std::ostream& operator<<(std::ostream& , DataBlobs<Tp>& blob);
```

VI. Overloading () operator

We overload this operator so that the data block has the ability to directly access the data at the specified location by the row, column, and channel.

```
template<typename T>
T& operator()(size_t r, size_t c, size_t ch);
```

1.3.3 CNN related Functions

The functions for convolution-related operations are static functions. Because for convolution-related operations, it is more a function of a class than an object unique to a data blob.

I. Padding Function

This function focuses on padding zeroes to the data correctly based on the size of the convolution kernel, making the output data of the convolution equal in size to the original data.

```
static bool padding(const DataBlobs& input, const size_t ker_r, const size_t ker_c, DataBlobs& pad);
```

II. Rearrange the data for img2Col

In order to realize the img2Col algorithm, we need to rearrange the data.

```
static bool rearrange_img2Col(const DataBlobs& input, const size_t ker_r, const size_t ker_c, DataBlobs& rearrange
```

III. 4 Convolution Functions

4 convolution functions are included in this class. The first one is the normal one. The second one is the unloop one, which calculates 8 elements at one time. The third one is the SIMD one, which calculates 256 bits size of elements at one time. The fourth one is the one using img2Col algorithm, which converts convolution operations to the multiplication between the input matrix and the kernel matrix. (The detailed realization of each function will be explained in the Code implementation part later.)

```
static bool cnn(const DataBlobs& pad, const DataBlobs& kernel, DataBlobs& output);
static bool cnn_unloop(const DataBlobs& pad, const DataBlobs& kernel, DataBlobs& output);
static bool cnn_avx2_ker3x3(const DataBlobs& pad, const DataBlobs& kernel, DataBlobs& output);
static bool cnn_img2Col(const DataBlobs& rearrange, const size_t input_row, const size_t input_col,
const DataBlobs& kernel, DataBlobs& output);
```

IV. Output Function

Two functions are written. One is to output the data directly from the calling object itself and the other one is to output the data based on the data of the input parameter(a static function). These functions are able to identify the format of the output file by the function name and thus output the data. (using OpenCV's imwrite)

```
bool output(const string filename) const;
static bool output(const DataBlobs& blob, const string filename);
```

1.3.4 Functions for data of 3 channels

Three functions are written here to get the data of each channel from the RGB adjacently stored data, and one function is written to merge the data of the three channels into the RGB adjacently stored data.

```
bool getRed(DataBlobs& r) const;
bool getGreen(DataBlobs& g) const;
bool getBlue(DataBlobs& b) const;
static bool CombineRGB(const DataBlobs& r, const DataBlobs& g, const DataBlobs& b, DataBlobs& rgb);
```

1.3.5 Auxiliary Functions

Auxiliary Functions make the data blob more user-friendly and safer.

I. Get basic information about the data blob

Since the information in the data block is protected in the private area, making it inaccessible directly outside the class, we need some public function to return the value in the private.

```
const size_t getRow();
const size_t getCol();
const size_t getChannel();
T* getData();
size_t* getRefCount();
```

II. Get the total number of data blobs

Since Datablobs_num is a static variable, we use a static function to return its value.

```
static size_t getDataBlobsNum();
```

III. Check if the Data Blob is empty

```
bool empty();
```

1.3.6 Destructor

Correctly free memory by checking if ref_count is equal to 1.

```
~DataBlobs();
```

1.4 Memory and Safety Management

1.4.1 Check the input parameter of a function

Checking the correctness of the input parameters of a function guarantees the security of the program and the accuracy of the operations.

In the constructors, we check if the row, column, or channel input in the functions is zero. And check if the input data is valid. These checks can guarantee the correctness of the creation of a data blob.

When overloading computation operators, we first check if the data of the data blob and the input data blob's data are valid. Secondly, for the + and - operations, we need to check if the sizes of the two data blobs are the same. For the * operations, we need to check if the column of the left data blob is equal to the row of the left data blob. For the / operation, we need to check if the number divided by this data block is zero.

When overloading the () operator to access the data of the data blob, we need to check if the index is out of the range of the data. When overloading << operator and the output() function to output the data of the data blob, we need to check if the data is valid for outputting.

In the functions dealing with the data of channel 3, we need to check if the channel of the data blob is 3. In the function related to the operations of CNN, we need to check if the channel of the data blob is 1. The Convolution functions written can not deal with the adjacent stored RGB data. When doing the Convolution operation of the adjacent stored RGB data, we separate the data of each channel, do the convolution operation to each channel and combine the result data to form an adjacent stored RGB data.

Every time when we encounter a wrong input, we will use ostream object cerr to output error information to the standard error stream. Compared with cout, cerr is output directly without buffering, which means the error messages can be sent directly to the display without waiting for a buffer or a new line break to be displayed. To stop the program, the exit() function is called.

```
if(...)//check the error condition
{
    cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
    cerr << "error message" << endl;
    exit(EXIT_FAILURE);
}
```

1.4.2 Considering the return type of a function

In a function that needs to create a new memory in the heap to store data and output that data, we make sure that the generated data is accepted by the DataBlobs by passing in the output DataBlobs. Since a function with DataBlobs or DataBlobs& as the return type is not accepted by a variable of the data type DataBlobs, it will cause a memory leak, and we cannot detect whether the return of a function is accepted correctly, we take the method of passing the accepting object as a parameter to the function and make the data accepted by the correct object inside the function, which can prevent memory leaks.

1.4.3 Free the memory correctly

In this project, we use the soft copy method to copy data from one data blob to another, which means the same data in the same memory is shared with many data blobs. This method can improve memory utilization but can cause problems when freeing the memory. Thus, we use a size_t pointer named ref_count to record the times the data is being referenced. In the destructor, we need to check if the value of ref_count is equal to one. If not, we should not delete the data as there are other data blobs that are using the data. If so, we delete the data and also delete the ref_count.

2 Code Implementation

2.1 Constructors

Five constructors are mentioned below.

The implementation of the constructor requires an input of a file name using the imread() function in OpenCV library.

```
template<typename T>
DataBlobs<T>::DataBlobs(){
    cout << "Constructor() is invoked" << endl;
    this->row = 0;
    this->col = 0;
    this->channel = 0;
    this->data = NULL;
    this->ref_count = NULL;
    DataBlobs_num++;
}
template<typename T>
DataBlobs<T>::DataBlobs(const size_t row, const size_t col, const size_t channel){
    //Safety check
    T* data = new T[row*col*channel];
    memset(data, 0, sizeof(T)*row*col*channel);
    this->row = row;
    this->col = col;
    this->channel = channel;
    this->data = data;
    this->ref_count = new size_t(1);
    DataBlobs_num++;
}
template<typename T>
DataBlobs<T>::DataBlobs(const size_t row, const size_t col, const size_t channel, T* data, size_t* ref_count){
    //Safety check
    if(ref_count==NULL)
        this->ref_count = new size_t(1);
    else
    {
        this->ref_count = ref_count;
        ++*this->ref_count;
    }
    this->row = row;
    this->col = col;
    this->channel = channel;
    this->data = data;
    DataBlobs_num++;
}
template<typename T>
DataBlobs<T>::DataBlobs(const DataBlobs<T>& blob){
    this->row = blob.row;
    this->col = blob.col;
    this->channel = blob.channel;
    this->data = blob.data;
    if((blob.ref_count)==NULL)
        this->ref_count = NULL;
    else
    {
        this->ref_count = blob.ref_count;
        ++*this->ref_count;
```

```
    }
    DataBlobs_num++;
}
template<typename T>
DataBlobs<T>::DataBlobs(const string filename){
    //Safety check
    Mat image = imread(filename);
    T* data = new T[(image.rows)*(image.cols)*(image.channels())];
    this->row = image.rows;
    this->col = image.cols;
    this->channel = image.channels();
    size_t row_stride = (this->col)*(this->channel);
    if(this->channel == 1)
        for(int i = 0; i < this->row; i++)
            for(int j = 0; j  < this->col; j++)
                data[i*row_stride + j] =(T)image.at<uchar>(i,j);
    // Convert and store BGR channels as RGB channels
    else if(this->channel == 3)
        for(int i = 0; i < this->row; i++)
            for(int j = 0; j < this-> col; j++)
            {
                data[i*row_stride + j*3] = static_cast<T>(image.at<Vec3b>(i,j)[2]);
                data[i*row_stride + j*3 + 1] = static_cast<T>(image.at<Vec3b>(i,j)[1]);
                data[i*row_stride + j*3 + 2] = static_cast<T>(image.at<Vec3b>(i,j)[0]);
            }
    this->data = data;
    this->ref_count = new size_t(1);
    DataBlobs_num++;
}
```

2.2 Operators overloading

2.2.1 = operator

```
template<typename T>
DataBlobs<T>& DataBlobs<T>::operator=(const DataBlobs<T>& blob){
    if(this == &blob)
        return *this;
    this->row = blob.row;
    this->col = blob.col;
    this->channel = blob.channel;
    if(this->ref_count == NULL){
        this->data = blob.data;
        this->ref_count = blob.ref_count;
    }
    else if(*(this->ref_count)==1){
        delete[] this->data;
        this->data = blob.data;
        delete[] this->ref_count;
        this->ref_count = blob.ref_count;
    }
    else{
        this->data = blob.data;
        --*this->ref_count;
        this->ref_count = NULL;
        this->ref_count = blob.ref_count;
    }
    if(this->ref_count != NULL)
        ++*this->ref_count;
    return *this;
}
```

2.2.2 +, -, *, / operators

The code implementations of overloading +, -, *, / operators are similar. Thus, we only display the + operator overloading here.

```
template<typename T>
DataBlobs<T> DataBlobs<T>::operator+(T num) const{
    //Safety check
    T* data = new T[(this->row)*(this->col)*(this->channel)];
    for(int i = 0; i < (this->row)*(this->col)*(this->channel); i++)
        data[i] = this->data[i] + num;
    DataBlobs B(this->row, this->col, this->channel, data, NULL);
    return B;
}
template<typename T>
DataBlobs<T> DataBlobs<T>::operator+(const DataBlobs<T>& blob) const{
```

```
    T* data = new T[(this->row)*(this->col)*(this->channel)];
    for(int i = 0; i < (this->row)*(this->col)*(this->channel); i++)
        data[i] = this->data[i] + *(blob.data + i);
    DataBlobs B(this->row, this->col, this->channel, data, NULL);
    return B;
}
template<typename T>
DataBlobs<T>& DataBlobs<T>::operator+=(T num){
    //Safety check
    T* data = new T[(this->row)*(this->col)*(this->channel)];
    for(int i = 0; i < (this->row)*(this->col)*(this->channel); i++)
        data[i] = this->data[i] + num;
    if(*(this->ref_count)==1){
        delete[] this->data;
        this->data = data;
    }
    else{
        this->data = data;
        --*this->ref_count;
        this->ref_count = NULL;
        this->ref_count = new size_t(1);
    }
    return *this;
}
template<typename T>
DataBlobs<T>& DataBlobs<T>::operator+=(const DataBlobs<T>& blob){
    //Safety check
    T* data = new T[(this->row)*(this->col)*(this->channel)];
    for(int i = 0; i < (this->row)*(this->col)*(this->channel); i++)
        data[i] = this->data[i] + *(blob.data+i);
    if(*(this->ref_count)==1){
        delete[] this->data;
        this->data = data;
    }
    else{
        this->data = data;
        --*this->ref_count;
        this->ref_count = NULL;
        this->ref_count = new size_t(1);
    }
    return *this;
}
template<typename Tp>
friend DataBlobs operator+(Tp num, const DataBlobs<Tp>& blob){
    return blob + num;
}
```

By the way, in order to improve the speed of the multiplication between two matrices, we use the function cblas_sgemm() in the openblas library to help. The corresponding code is mentioned below. Since the data type of the input data of the function should be float, we need to transform our data to float first and then do the multiplication.

```
template<typename T>
DataBlobs<T> DataBlobs<T>::operator*(const DataBlobs& blob) const{//openblas
    //Safety Check
    T* data = new T[(this->row)*(blob.col)];
    float* left = new float[(this->col)*(this->row)];
    float* right = new float[(blob.row)*(blob.col)];
    float* res = new float[(this->row)*(blob.col)];
    //transform the data type of the data
    for(int i = 0; i < (this->col)*(this->row); i++)
        left[i] = static_cast<float>(*(this->data + i));
    for(int i = 0; i < (blob.row)*(blob.col); i++)
        right[i] = static_cast<float>(*(blob.data + i));
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, this->row, blob.col, this->col, 1.0f, left, this->col,
    for(int i = 0; i < (this->row)*(blob.col); i++)
        data[i] = static_cast<T>(res[i]);
    DataBlobs B(this->row, blob.col, 1, data, NULL);
    delete[] left;
    delete[] right;
    delete[] res;
    return B;
}
```

2.2.3 ++, -- operators

As the process of ++ and -- operators overloading are very similar, only ++ overloading is mentioned below.

```
//Prefix
template<typename T>
```

```
DataBlobs<T>& DataBlobs<T>::operator++(){
    //Safety check
    size_t num = (this->row)*(this->col)*(this->channel);
    if(*(this->ref_count) == 1){
        for(int i = 0; i < num; i++)
            *(this->data + i) = *(this->data + i) + 1;
    }
    else{
        T* data = new T[num];
        if(data == NULL){
            cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
            cerr << "Fail to allocate memory for the data" << endl;
            exit(EXIT_FAILURE);
        }
        for(int i = 0; i < num; i++)
            data[i] = *(this->data + i) + 1;
        this->data = data;
        --*(this->ref_count);
        this->ref_count = NULL;
        this->ref_count == new size_t(1);
    }
    return *this;
}
//Suffix
template<typename T>
DataBlobs<T> DataBlobs<T>::operator++(int){
    DataBlobs old = *this;
    operator++();
    return old;
}
```

2.2.4 ==, != operators

```
template<typename T>
bool DataBlobs<T>::operator==(const DataBlobs& blob){
    if(this->row != blob.row)
        return false;
    if(this->col != blob.col)
        return false;
    if(this->channel != blob.channel)
        return false;
    size_t num = (this->row)*(this->col)*(this->channel);
    for(int i=0; i<num; i++)
        if(*(this->data + i) != *(blob.data + i))
            return false;
    return true;
}
template<typename T>
bool DataBlobs<T>::operator!=(const DataBlobs& blob){
    if(*this == blob)
        return false;
    else
        return true;
}
```

2.2.5 << operator

```
template<typename T>
std::ostream& operator<<(std::ostream & os, DataBlobs<T>& blob){
    //Safety Check
    for(size_t c = 1; c <= blob.channel; c++){
        for(size_t i = 0; i < blob.row; i++){
            for(size_t j = 0; j < blob.col; j++)
                os << blob(i, j, c) << " ";
            os << endl;
        }
        os << endl;
    }
    return os;
}
```

2.2.6 () operator

```
template<typename T>
T& DataBlobs<T>::operator()(size_t r, size_t c, size_t ch){
    //Safety check
```

```
    return this->data[r * this->col * this->channel + c * this->channel + (ch-1)];
}
```

2.3 CNN related Functions

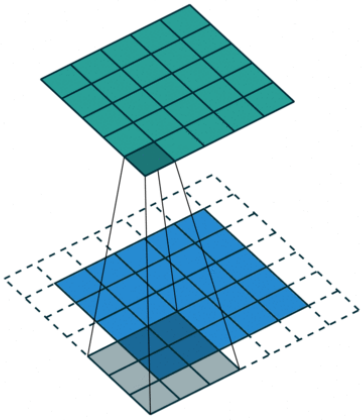
2.3.1 Padding

In order to let the output image maintains the same size as the input image, the function will pad the edges of the data with zeros. If the kernel is nxn we wrap (n-1)/2 circle(s) of zeros around the data. (n must be odd)

```
template<typename T>
bool DataBlobs<T>::padding(const DataBlobs& input, const size_t ker_r, const size_t ker_c, DataBlobs& pad){
    //Safety check
    size_t input_row = input.row;
    size_t input_col = input.col;
    size_t pad_row = input_row + (ker_r - 1);
    size_t pad_col = input_col + (ker_c - 1);
    T* pad_data = new T[pad_row*pad_col]();//0
    if(pad_data == NULL){
        cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
        cerr << "Fail to allocate memory for padding data" << endl;
        exit(EXIT_FAILURE);
    }
    size_t row_bias = (ker_r - 1) / 2;
    size_t col_bias = (ker_c - 1) / 2;
    for(int i = row_bias; i < input_row + row_bias; i++){
        memcpy(pad_data + i*pad_col + col_bias, input.data + (i - col_bias)*input_col, input_col*sizeof(T));
    }
    pad.row = pad_row;
    pad.col = pad_col;
    pad.channel = 1;
    if(pad.ref_count==NULL){
        pad.data = pad_data;
        pad.ref_count = new size_t(1);
    }
    else if(*(pad.ref_count)==1){
        delete[] pad.data;
        pad.data = pad_data;
    }
    else{
        pad.data = pad_data;
        --*pad.ref_count;
        pad.ref_count = new size_t(1);
    }
    return true;
}
```

2.3.2 Normal Convolution

A convolution operation can be seen as a sliding window that moves the kernel over the input data and calculates the dot product between the kernel and the data within the window, resulting in an output value. By moving the kernel over the input data, multiple output values can be obtained, which constitute an output feature map.



The normal convolution operation uses 4 layers of looping to complete the operation. However, we can reduce the layers of the loop from 4 to 2. For the convolution of an element, we can get the row-index = (k / kernel_col) and the column-index = (k % kernel_col). For the convolution of the whole image, which we can get the row-index = (t / input_width) and the column-index = (t % input_width). Thus, we can reduce the dimension of the loop from 4 to 2.

```
template<typename T>
bool DataBlobs<T>::Convolution(const DataBlobs<T>& pad, const DataBlobs<T>& kernel, DataBlobs<T>& output){
    //Safety check
    size_t pad_row = pad.row;
    size_t pad_col = pad.col;
    size_t kernel_row = kernel.row;
    size_t kernel_col = kernel.col;
    size_t input_row = pad_row - (kernel_row - 1);
    size_t input_col = pad_col - (kernel_col - 1);
    T* output_data = new T[input_row*input_col];
    //Convolution operation
    for(size_t t = 0; t < input_row*input_col; t++)
    {
        size_t j = t / input_col;
        size_t i = t % input_col;
        for(size_t k =0; k<kernel_col * kernel_row;k++)
        {
            *(output_data+ t) += (*(pad.data + (j * pad_col) + i + ((k / kernel_col)*pad_col) + (k % kernel_col)))
        }
    }
    //Output the result
    output.row = input_row;
    output.col = input_col;
    output.channel = 1;
    if(output.ref_count==NULL){
        output.data = output_data;
        output.ref_count = new size_t(1);
    }
    else if(*(output.ref_count)==1){
        delete[] output.data;
        output.data = output_data;
    }
    else{
        output.data = output_data;
        --*output.ref_count;
        output.ref_count = new size_t(1);
    }
    return true;
}
```

2.3.3 Unloop Convolution

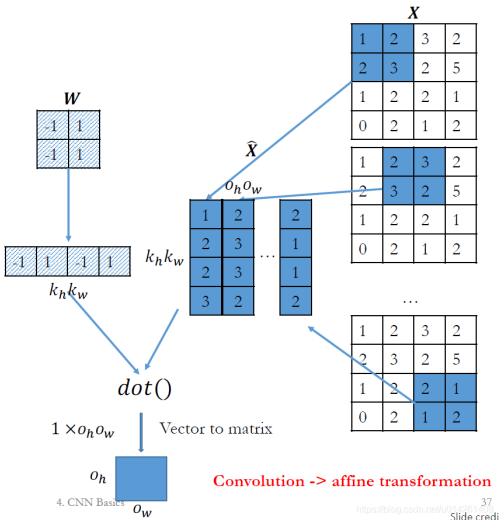
In the unloop function, we calculate 8 elements at one time. Some repeated calculated value are calculated outside of the loop in order to reduce the time of the repeated calculations. By the way, in order to deal with all the elements in the case that the input width is not a multiple of 8, we can only reduce the loop dimension to 3.

```
template<typename T>
bool DataBlobs<T>::cnn_unloop(const DataBlobs<T>& pad, const DataBlobs<T>& kernel, DataBlobs<T>& output){
    //Safety check
    size_t pad_row = pad.row;
    size_t pad_col = pad.col;
    size_t kernel_row = kernel.row;
    size_t kernel_col = kernel.col;
    size_t input_row = pad_row - (kernel_row - 1);
    size_t input_col = pad_col - (kernel_col - 1);
    T* kernel_data = kernel.data;
    T* pad_data = pad.data;
    T* output_data = new T[input_row*input_col];
    //Convolution operation
    for(size_t i = 0; i < input_row; i++)
    {
        size_t stride = i*input_col;
        size_t j = 0;
        for(; j < (input_col/8)*8; j+=8)
        {
            size_t out_index = stride + j;
            for(size_t m = 0; m<kernel_row;m++)
            {
                for(size_t n = 0,k = 0; n < kernel_col; n++,k++)
                {
                    size_t pad_index = (i * pad_col) + j + (m*pad_col) + n;
```

```
                    *(output_data+out_index) += (*(pad_data + pad_index)) * (*(kernel_data+k));
                    *(output_data+out_index+1) += (*(pad_data + pad_index + 1)) * (*(kernel_data+k));
                    *(output_data+out_index+2) += (*(pad_data + pad_index + 2)) * (*(kernel_data+k));
                    *(output_data+out_index+3) += (*(pad_data + pad_index + 3)) * (*(kernel_data+k));
                    *(output_data+out_index+4) += (*(pad_data + pad_index + 4)) * (*(kernel_data+k));
                    *(output_data+out_index+5) += (*(pad_data + pad_index + 5)) * (*(kernel_data+k));
                    *(output_data+out_index+6) += (*(pad_data + pad_index + 6)) * (*(kernel_data+k));
                    *(output_data+out_index+7) += (*(pad_data + pad_index + 7)) * (*(kernel_data+k));
                }
            }
        }
        if((input_col % 8) != 0)
        {
            for(int w=j;w<(input_col % 8)+j;w++)
            {
                for(int m = 0; m<kernel_row;m++)
                {
                    for(int n = 0,k = 0; n < kernel_col; n++,k++)
                    {
                        *(output_data+(stride)+w)+=(*(pad_data+(i*pad_col)+w+m*pad_col+n))*(*(kernel_data+k));
                    }
                }
            }
        }
    }
    //Output the result
    output.row = input_row;
    output.col = input_col;
    output.channel = 1;
    if(output.ref_count==NULL){
        output.data = output_data;
        output.ref_count = new size_t(1);
    }
    else if(*(output.ref_count)==1){
        delete[] output.data;
        output.data = output_data;
    }
    else{
        output.data = output_data;
        --*output.ref_count;
        output.ref_count = new size_t(1);
    }
    return true;
}
```

2.3.4 img2Col Convolution

The core idea of the img2Col algorithm is to convert the convolution operation into the multiplication of two matrices. (The rearranged data matrix and the kernel data matrix) As the algorithms for dealing with the multiplication between two matrices are very mature, thus we can take advantage of those algorithms to improve the speed of the convolution operation.



The principle of the img2Col algorithm is drawn precisely above. Firstly, as what is drawn on the right of the image, the data covered by the kernel-sized sliding window need to be flattened into a column of vector. Then move the kernel-sized sliding window to another area of data and flatten the new area of data into a column of the vector. Repeated those processes until the kernel-sized window moves along the whole

data. Secondly, as what is drawn in the middle of the image, connect all the column vectors generated in the previous step to form a new matrix, whose row is equal to the size of the kernel and the column is equal to the size of the output image size. Thirdly, as what is drawn on the left of the image, flatten the kernel into a column vector. Finally, do the matrix multiplication between the two matrices. The result is the result of the convolution operation in a flattened form.

By the way, in order to speed up the process of multiplication between two matrices, we use the function cblas_sgemm() in openblas library to help.

The Function of rearranging the data.

```
template<typename T>
bool DataBlobs<T>::rearrange_img2Col(const DataBlobs& input, const size_t ker_r, const size_t ker_c,
    DataBlobs& rearrange)
{
    //Safety check
    size_t input_row = input.row;
    size_t input_col = input.col;
    size_t pad_row = input_row + (ker_r - 1);
    size_t pad_col = input_col + (ker_c - 1);
    size_t re_row = pad_row * pad_col;
    size_t re_col = ker_r * ker_c;
    T* pad_data = new T[pad_row*pad_col]();
    if(pad_data == NULL){
        cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
        cerr << "Fail to allocate memory for padding data" << endl;
        exit(EXIT_FAILURE);
    }
    T* re_data = new T[re_row*re_col];
    if(re_data == NULL){
        cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
        cerr << "Fail to allocate memory for padding data" << endl;
        delete[] pad_data;
        exit(EXIT_FAILURE);
    }
    size_t row_bias = (ker_r - 1) / 2;
    size_t col_bias = (ker_c - 1) / 2;
    //padding
    for(int i = row_bias; i < input_row + row_bias; i++){
        memcpy(pad_data + i*pad_col + col_bias, input.data + (i - col_bias)*input_col, input_col*sizeof(T));
    }
    //rearrange data
    size_t t = 0;
    for(size_t i = 0; i < input_row; i++){
        for(size_t j = 0; j < input_col; j++){
            for(size_t m = 0; m < ker_r; m++){
                for(size_t n = 0; n < ker_c; n++, t++){
                    re_data[t] = *(pad_data + i*pad_col + j + m*ker_c + n);
                }
            }
        }
    }
    rearrange.row = re_row;
    rearrange.col = re_col;
    rearrange.channel = 1;
    if(rearrange.ref_count==NULL){
        rearrange.data = re_data;
        rearrange.ref_count = new size_t(1);
    }
    else if(*(rearrange.ref_count)==1){
        delete[] rearrange.data;
        rearrange.data = re_data;
    }
    else{
        rearrange.data = re_data;
        --*rearrange.ref_count;
        rearrange.ref_count = new size_t(1);
    }
    delete[] pad_data;
    return true;
}
```

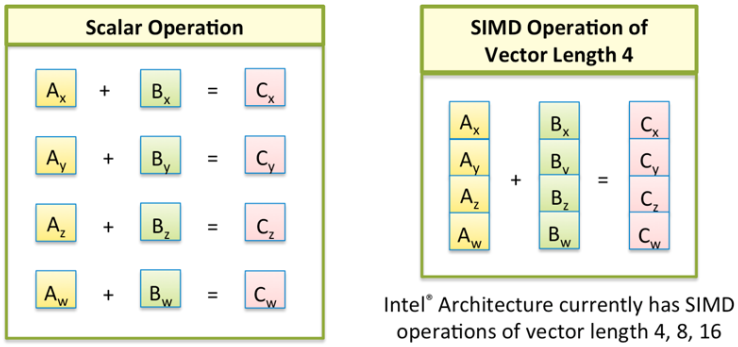
The Function of flattening the kernel data and do the matrix multiplication.

```
template<typename T>
bool DataBlobs<T>::cnn_img2Col(const DataBlobs& rearrange, const size_t input_row, const size_t input_col,
const DataBlobs& kernel, DataBlobs& output){
    //Safety check
    DataBlobs<T> ker((kernel.row)*(kernel.col),1,1,kernel.data,kernel.ref_count);
    DataBlobs<T> res;
    //Convolution operation
```

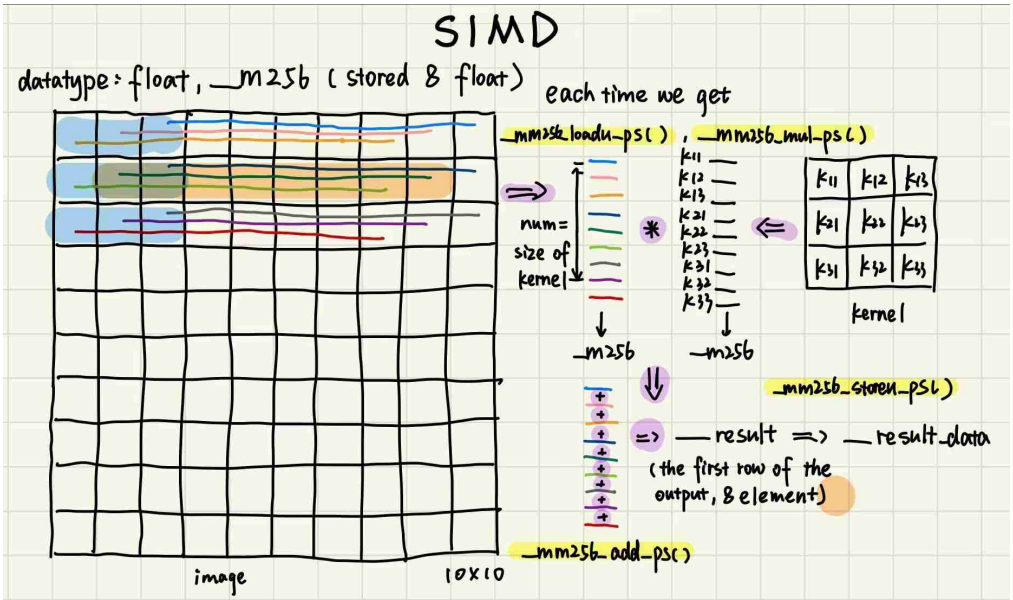
```
    res = rearrange * ker;//invoke the * overloading function
    //Output the result
    output.row = input_row;
    output.col = input_col;
    output.channel = 1;
    if(output.ref_count==NULL){
        output.data = res.data;
        output.ref_count = res.ref_count;
        ++*(output.ref_count);
    }
    else if(*(output.ref_count)==1){
        delete[] output.data;
        output.data = res.data;
        delete[] output.ref_count;
        output.ref_count = res.ref_count;
        ++*(output.ref_count);
    }
    else{
        output.data = res.data;
        --*output.ref_count;
        output.ref_count = res.ref_count;
        ++*(output.ref_count);
    }
    return true;
}
```

2.3.5 SIMD Convolution

In SIMD architecture, a single instruction can operate multiple data elements at the same time, making full use of register space. It can greatly reduce the number of instruction executions and improves the running efficiency of the program.



Using datatype float, a 10x10 image, and a 3x3 kernel as an example. The steps of the realization are drawn in the following image.



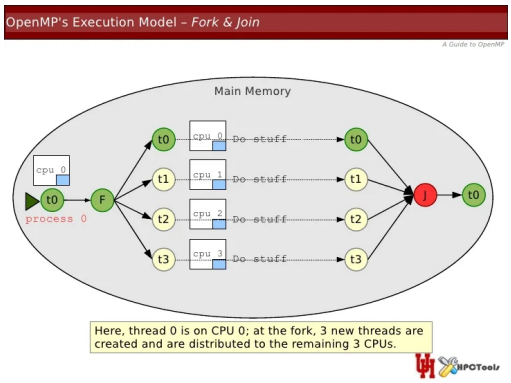
Firstly, considering the size of the register variable, we get num = 32 / sizeof(T) continuous elements (float: 8) at one time and stored them into the register through a datatype named __m256 using _mm256_loadu_ps function. The total of the group (kernel 3x3: 9 groups) and the position of the data is according to the size of the kernel size. (kernel 3x3: coinciding with 3x3) Secondly, store the kernel data separately as __m256 datatype. Thirdly, we can use _mm256_mul_ps to multiply the image data with kernel data. Fourthly, we _mm256_add_ps to add all the results. Finally, we use _mm256_storeu_ps to store the data in the register back to the memory. (Remember to use _mm256_setzero_ps to set the data named res to zero at the beginning of every operation).

```
template<typename T>
bool DataBlobs<T>::cnn_avx2_ker3x3(const DataBlobs<T>& pad, const DataBlobs<T>& kernel, DataBlobs<T>& output){
    //Safety check
    size_t pad_row = pad.row;
    size_t pad_col = pad.col;
    size_t kernel_row = kernel.row;
    size_t kernel_col = kernel.col;
    size_t input_row = pad_row - (kernel_row - 1);
    size_t input_col = pad_col - (kernel_col - 1);
    size_t num = 32 / sizeof(T);
    T* kernel_data = kernel.data;
    T* input_data = pad.data;
    T* output_data = new T[input_row*input_col];
#ifdef WITH_AVX2
    __m256 im11,im12,im13,im21,im22,im23,im31,im32,im33;
    __m256 ker11,ker12,ker13,ker21,ker22,ker23,ker31,ker32,ker33;
    __m256 res;
    T* Ker11=static_cast<T*>(aligned_alloc(256,32));
    T* Ker12=static_cast<T*>(aligned_alloc(256,32));
    T* Ker13=static_cast<T*>(aligned_alloc(256,32));
    T* Ker21=static_cast<T*>(aligned_alloc(256,32));
    T* Ker22=static_cast<T*>(aligned_alloc(256,32));
    T* Ker23=static_cast<T*>(aligned_alloc(256,32));
    T* Ker31=static_cast<T*>(aligned_alloc(256,32));
    T* Ker32=static_cast<T*>(aligned_alloc(256,32));
    T* Ker33=static_cast<T*>(aligned_alloc(256,32));
    for(int i=0;i<num;i++)
    {
        *(Ker11+i)=*(kernel_data);
        *(Ker12+i)=*(kernel_data+1);
        *(Ker13+i)=*(kernel_data+2);
        *(Ker21+i)=*(kernel_data+3);
        *(Ker22+i)=*(kernel_data+4);
        *(Ker23+i)=*(kernel_data+5);
        *(Ker31+i)=*(kernel_data+6);
        *(Ker32+i)=*(kernel_data+7);
        *(Ker33+i)=*(kernel_data+8);
    }
    ker11 = _mm256_loadu_ps(Ker11);
    ker12 = _mm256_loadu_ps(Ker12);
    ker13 = _mm256_loadu_ps(Ker13);
    ker21 = _mm256_loadu_ps(Ker21);
    ker22 = _mm256_loadu_ps(Ker22);
    ker23 = _mm256_loadu_ps(Ker23);
    ker31 = _mm256_loadu_ps(Ker31);
    ker32 = _mm256_loadu_ps(Ker32);
    ker33 = _mm256_loadu_ps(Ker33);
    T*temp=static_cast<T*>(aligned_alloc(256,32));
    for(size_t i=0; i<input_row;i++)
    {
        size_t j=0;
        for(; j<(input_col/num)*num; j+=num)
        {
            res =  _mm256_setzero_ps();
            im11 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (0*pad_col) + 0);
            im12 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (0*pad_col) + 1);
            im13 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (0*pad_col) + 2);
            im21 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (1*pad_col) + 0);
            im22 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (1*pad_col) + 1);
            im23 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (1*pad_col) + 2);
            im31 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (2*pad_col) + 0);
            im32 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (2*pad_col) + 1);
            im33 = _mm256_loadu_ps(input_data + (i * pad_col) + j + (2*pad_col) + 2);
            res = _mm256_add_ps(res, _mm256_mul_ps(im11, ker11));
            res = _mm256_add_ps(res, _mm256_mul_ps(im12, ker12));
            res = _mm256_add_ps(res, _mm256_mul_ps(im13, ker13));
            res = _mm256_add_ps(res, _mm256_mul_ps(im21, ker21));
            res = _mm256_add_ps(res, _mm256_mul_ps(im22, ker22));
            res = _mm256_add_ps(res, _mm256_mul_ps(im23, ker23));
            res = _mm256_add_ps(res, _mm256_mul_ps(im31, ker31));
            res = _mm256_add_ps(res, _mm256_mul_ps(im32, ker32));
            res = _mm256_add_ps(res, _mm256_mul_ps(im33, ker33));
```

```
            _mm256_storeu_ps(temp, res);
            for(size_t k=0;k<num;k++)
                *(output_data+(i*input_col)+j+k) = *(temp+k);
        }
        if((input_col % num) != 0)
        {
            for(size_t w=j;w<(input_col % num)+j;w++)
            {
                for(size_t k =0; k<kernel_col*kernel_row;k++)
                {
                    *(output_data+(i*input_col)+w)+=(*(input_data+(i*pad_col)+w+
                    (k/kernel_col)*pad_col+(k%kernel_col)))*(*(kernel_data+k));
                }
            }
        }
    }
    output.row = input_row;
    output.col = input_col;
    output.channel = 1;
    if(output.ref_count==NULL){
        output.data = output_data;
        output.ref_count = new size_t(1);
    }
    else if(*(output.ref_count)==1){
        delete[] output.data;
        output.data = output_data;
    }
    else{
        output.data = output_data;
        --*output.ref_count;
        output.ref_count = new size_t(1);
    }
    return true;
#else
    cout << "AVX2 is not supported" << endl;
    return false;
#endif
}
```

2.3.6 OpenMP Convolution

On a multi-core computer, OpenMP can map multiple threads to different CPU cores so that different threads can execute different tasks at the same time. This parallel execution can significantly improve the running efficiency of programs because it can fully utilize the multi-core processing power of the computer.



Adding the following instruction can suggest the program runs the operation in multiple threads. (usually used before the loop)

```
#pragma omp parallel for
```

2.3.7 Output

The implementation of the output functions using the imwrite() function in OpenCV library.

```
template<typename T>
bool DataBlobs<T>::output(const string filename) const{
    //Safety check
```

```
    unsigned char* data = new unsigned char[(this->row)*(this->col)*(this->channel)];
    size_t row_stride = (this->col)*(this->channel);
    if(this->channel == 3){
        for(int i = 0; i < this->row; i++)//Converting RGB to BGR
            for(int j = 0; j < this-> col; j++)
            {
                data[i*row_stride + j*3] = static_cast<unsigned char>(*(this->data + i*row_stride + j*3 + 2));
                data[i*row_stride + j*3 + 1] = static_cast<unsigned char>(*(this->data + i*row_stride + j*3 + 1));
                data[i*row_stride + j*3 + 2] = static_cast<unsigned char>(*(this->data + i*row_stride + j*3));
            }
        Mat image(this->row, this->col, CV_8UC3, data);
        imwrite(filename, image);
    }
    else if(this->channel == 1){
        for(int i = 0; i < (this->row)*(this->col); i++)
            data[i] = static_cast<unsigned char>(*(this->data + i));
        Mat image(this->row, this->col, CV_8UC1, data);
        imwrite(filename, image);
    }
    delete[] data;
    return true;
}

template<typename T>
bool DataBlobs<T>::output(const DataBlobs& blob, const string filename){
    //Safety check
    unsigned char* data = new unsigned char[(blob.row)*(blob.col)*(blob.channel)];
    size_t row_stride = (blob.col)*(blob.channel);
    if(blob.channel == 3){
        for(int i = 0; i < blob.row; i++)//Converting RGB to BGR
            for(int j = 0; j < blob.col; j++)
            {
                data[i*row_stride + j*3] = static_cast<unsigned char>(*(blob.data + i*row_stride + j*3 + 2));
                data[i*row_stride + j*3 + 1] = static_cast<unsigned char>(*(blob.data + i*row_stride + j*3 + 1));
                data[i*row_stride + j*3 + 2] = static_cast<unsigned char>(*(blob.data + i*row_stride + j*3));
            }
        Mat image(blob.row, blob.col, CV_8UC3, data);
        imwrite(filename, image);
    }
    else if(blob.channel == 1){
        for(int i = 0; i < (blob.row)*(blob.col); i++)
            data[i] = static_cast<unsigned char>(*(blob.data + i));
        Mat image(blob.row, blob.col, CV_8UC1, data);
        imwrite(filename, image);
    }
    delete[] data;
    return true;
}
```

2.4 Functions for data of 3 channels

2.4.1 Get data of R, G, B

```
template<typename T>
bool DataBlobs<T>::getRed(DataBlobs<T>& r) const{
    //Safety Check
    T* data = new T[(this->row)*(this->col)];
    if(data == NULL){
        cerr << "File " << __FILE__ << "Line " << __LINE__ << "Func " << __FUNCTION__ << "()" << endl;
        cerr << "Fail to allocate memory for data" << endl;
        exit(EXIT_FAILURE);
    }
    int row_stride = (this->col)*(this->channel);
    for(int i = 0; i < this->row; i++)
        for(int j = 0; j < this-> col; j++)
            data[i*this->col + j] = *(this->data + i*row_stride + j*3);
    if((r.ref_count!=NULL)&&(--*r.ref_count)==0)
    {
        delete[] r.data;
        delete[] r.ref_count;
    }
    r.row = this->row;
    r.col = this->col;
    r.channel = 1;
    r.data = data;
    r.ref_count = new size_t(1);
    return true;
}
template<typename T>
bool DataBlobs<T>::getGreen(DataBlobs<T>& g) const;
template<typename T>
```

```
bool DataBlobs<T>::getBlue(DataBlobs<T>& g) const;
```

2.4.2 Combine the RGB data

```
template<typename T>
bool DataBlobs<T>::CombineRGB(const DataBlobs<T>& r, const DataBlobs<T>& g, const DataBlobs<T>& b,
    DataBlobs<T>& rgb)
{
    //Safety check
    T* data = new T[(r.row)*(r.col)*3];
    if((rgb.ref_count!=NULL)&&(--*rgb.ref_count)==0)
    {
        delete[] b.data;
        delete[] b.ref_count;
    }
    rgb.row = r.row;
    rgb.col = r.col;
    rgb.channel = 3;
    size_t row_stride = (rgb.col) * 3;
    size_t row_stride_ = (rgb.col);
    for(int i = 0; i < rgb.row; i++)
        for(int j = 0; j < rgb.col; j++){
            data[i*row_stride + j*3] = *(r.data + i*row_stride_ + j);
            data[i*row_stride + j*3 + 1] = *(g.data + i*row_stride_ + j);
            data[i*row_stride + j*3 + 2] = *(b.data + i*row_stride_ + j);
        }
    rgb.data = data;
    rgb.ref_count = new size_t(1);
    return true;
}
```

2.5 Auxiliary Functions

2.5.1 Get basic information

```
const size_t getRow(){
    return this->row;
}
const size_t getCol(){
    return this->col;
}
const size_t getChannel(){
    return this->channel;
}
T* getData(){
    return this->data;
}
size_t* getRefCount(){
    return this->ref_count;
}
```

2.5.2 Get the total number of Data Blobs

```
template<typename T>
size_t DataBlobs<T>::getDataBlobsNum(){
    return DataBlobs_num;
}
```

2.5.3 Check if empty

```
bool empty(){
    if(this->row == 0||this->col == 0||this->channel == 0||this->data == NULL||this->ref_count == NULL)
        return true;
    else
        return false;
}
```

2.6 Destructor

```
template<typename T>
DataBlobs<T>::~DataBlobs(){
    DataBlobs_num--;
    if(ref_count!=NULL&&(--*ref_count)==0)
    {
        delete[] this->data;
        this->data = NULL;
        delete[] this->ref_count;
        this->ref_count = NULL;
    }
}
```

2.7 Complie the file

```
//Compile the file on x86
g++ main.cpp -DWITH_AVX2 -mavx -lopenblas `pkg-config --cflags --libs opencv4` -std=c++17
//Compile the file on arm
g++ main.cpp -DWITH_NEON -lopenblas `pkg-config --cflags --libs opencv4` -std=c++17
```

3 Experiments & Analysis

3.1 Check the correctness of each function

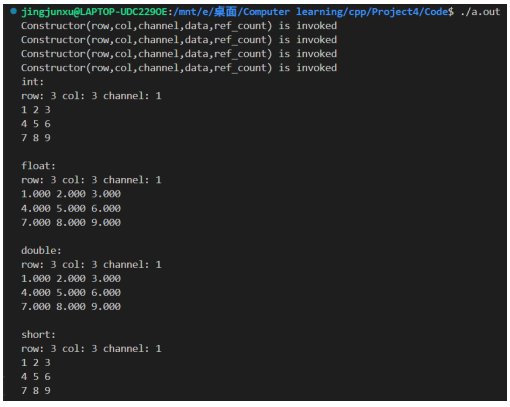
3.1.1 Constructors

DataBlobs(row, col, channel, data, ref_count):

We create four 3x3 data blobs in datatype short, int, float, and double through the constructor with complete parameters(row, col, channel, data, ref_count) and output the results.

```
int* data1 = new int[9]{1,2,3,4,5,6,7,8,9};
float* data2 = new float[9]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
double* data3 = new double[9]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
short* data4 = new short[9]{1,2,3,4,5,6,7,8,9};
DataBlobs data_int(3,3,1,data1,NULL);
DataBlobs data_float(3,3,1,data2,NULL);
DataBlobs data_double(3,3,1,data3,NULL);
DataBlobs data_short(3,3,1,data4,NULL);
```

The results are as follows. It is obvious that the constructor works well.



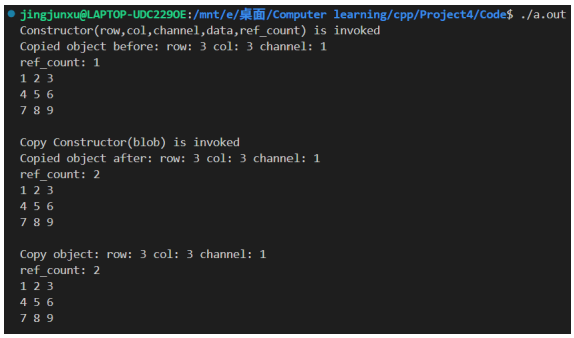
Copy Constructor:

To check the copy constructor, we create a DataBlob first and then use that DataBlob to construct a new Datablob. Finally, we output the information of the two Datablobs.

```
DataBlobs d1(3,3,1,data1,NULL);
cout << "Copied object before: " << d1;
```

```
DataBlobs d2(d1);
cout << "Copied object after: " << d1;
cout << "Copy object: " << d2;
```

The result is as follows. From the output, we can find that the data of these two DataBlobs are the same, and as the data is shared by two DataBlobs, the value of the ref_count is two, which is also correct.



DataBlobs(filename):

We read an image called ”gray.jpg”. If the image is read correctly, the output of the image must be correct.

```
DataBlobs<float> image("gray.jpg");
image.output("grayoutput.jpg");
```

The results are as follows. It is obvious that these two images are the same.

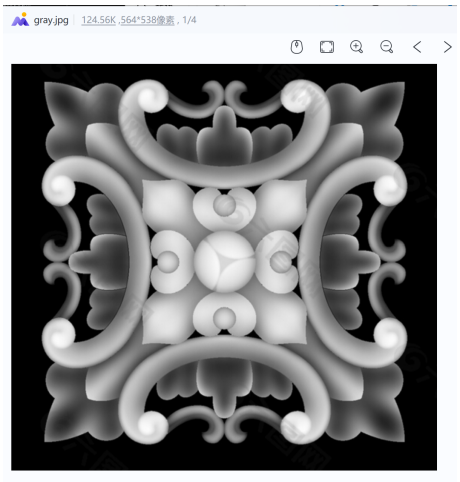


Figure 1: Input

Figure 2: Output

3.1.2 Operators Overloading

=:

We use = to copy the value of a DataBlob to another Datablob.

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs<int> d3;
d3 = d1;
cout << "d1: " << endl << d1;
cout << "d3: " << endl << d3;
```

The result is as follows. From the output, we can find that the data of these two DataBlobs are the same, and as the data is shared by these two DataBlobs, the value of the ref_count is two, which is also correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 2
1 2 3
4 5 6
7 8 9

d3:
row: 3 col: 3 channel: 1
ref_count: 2
1 2 3
4 5 6
7 8 9
```

+, -, *, / :

Doing calculations with numbers:

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs<int> d3;
cout << "d1: " << endl << d1;
d3 = d1 + 1;
cout << "d1 + 1 = " << endl << d3;
d3 = d1 - 1;
cout << "d2 - 1 = " << endl << d3;
d3 = d1 * 2;
cout << "d1 * 2 = " << endl << d3;
d3 = d1 / 2;
cout << "d1 / 2 = " << endl << d3;
```

The results are as follows. It obvious to find out that they are all correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

d1 + 1 =
row: 3 col: 3 channel: 1
ref_count: 1
2 3 4
5 6 7
8 9 10

d2 - 1 =
row: 3 col: 3 channel: 1
ref_count: 1
0 1 2
3 4 5
6 7 8

d1 * 2 =
row: 3 col: 3 channel: 1
ref_count: 1
2 4 6
8 10 12
14 16 18

d1 / 2 =
row: 3 col: 3 channel: 1
ref_count: 1
0 1 1
2 2 3
3 4 4
```

Friend function for + and * operator overloading:

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs<int> d3;
cout << "d1: " << endl << d1;
d3 = 1 + d1;
cout << "1 + d1 = " << endl << d3;
d3 = 2 * d1;
cout << "2 * d1 = " << endl << d3;
```

The results are as follows. They are correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

1 + d1 =
row: 3 col: 3 channel: 1
ref_count: 1
2 3 4
5 6 7
8 9 10

2 * d1 =
row: 3 col: 3 channel: 1
ref_count: 1
2 4 6
8 10 12
14 16 18
```

Doing calculation between two matrices:

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs d2(3,3,1,data2,NULL);
DataBlobs<int> d3;
cout << "d1: " << endl << d1;
cout << "d2: " << endl << d2;
d3 = d1 + d2;
cout << "d1 + d2 = " << endl << d3;
d3 = d2 - d1;
cout << "d2 - d1 = " << endl << d3;
d3 = d1 * d2;
cout << "d1 * d2 = " << endl << d3;
```

The results are as follows. They are correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

d2:
row: 3 col: 3 channel: 1
ref_count: 1
132 53 456
852 55 66
4 6 52

d1 + d2 =
row: 3 col: 3 channel: 1
ref_count: 1
133 55 459
856 60 72
11 14 61

d2 - d1 =
row: 3 col: 3 channel: 1
ref_count: 1
131 51 453
848 50 60
-3 -2 43

d1 * d2 =
row: 3 col: 3 channel: 1
ref_count: 1
1848 181 744
4812 523 2466
7776 865 4188
```

+=, -=, /=, *=

Doing calculations with numbers:

```
DataBlobs d1(3,3,1,data1,NULL);
cout << "d1: " << endl << d1;
d1 += 1;
cout << "d1 += 1. d1:" << endl << d1;
d1 -= 1;
cout << "d1 -= 1. d1:" << endl << d1;
d1 *= 3;
cout << "d1 *= 3. d1:" << endl << d1;
d1 /= 2;
cout << "d1 /= 2. d1:" << endl << d1;
```

The results are as follows. They are correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

d1 += 1. d1:
row: 3 col: 3 channel: 1
ref_count: 1
2 3 4
5 6 7
8 9 10

d1 -= 1. d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

d1 *= 3. d1:
row: 3 col: 3 channel: 1
ref_count: 1
3 6 9
12 15 18
21 24 27

d1 /= 2. d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 3 4
6 7 9
10 12 13
```

Doing calculations between two matrices:

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs d2(3,3,1,data2,NULL);
cout << "d1: " << endl << d1;
cout << "d2: " << endl << d2;
d1 += d2;
cout << "d1 += d2. d1:" << endl << d1;
d1 -= d2;
cout << "d1 -= 1. d1:" << endl << d1;
d1 *= d2;
cout << "d1 *= d2. d1:" << endl << d1;
```

The results are as follows. They are correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

d2:
row: 3 col: 3 channel: 1
ref_count: 1
132 53 456
852 55 66
4 6 52

d1 += d2. d1:
row: 3 col: 3 channel: 1
ref_count: 1
133 55 459
856 60 72
11 14 61

d1 -= 1. d1:
row: 3 col: 3 channel: 1
ref_count: 1
1 2 3
4 5 6
7 8 9

d1 *= d2. d1:
row: 3 col: 3 channel: 1
ref_count: 1
1848 181 744
4812 523 2466
7776 865 4188
```

++, --:

Prefix:

We first create two DataBlobs that share the same data. Then we do prefix increment to one DataBlob and do prefix decrement to another DataBlob.

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs d2(3,3,1,data1,d1.getRefCount());
DataBlobs<int> d3;
cout << "d1: " << endl << d1;
cout << "d2: " << endl << d2;
d3 = ++d1;
cout << "++d1: " << endl << d1;
```

```
cout << "d3 = ++d1 " << endl << d3;
d3 = --d2;
cout << "--d2: " << endl << d2;
cout << "d3 = --d2 " << endl << d3;
```

The results are as follows. Initially, as the two DataBlobs d1 and d2 share the same data, the value of the data and the ref_count of the data are all the same. Then, we do the prefix increment, assign the result to another DataBlob d3, and output the information of both the DataBlobs d1 and d3. Doing the increment of the data in d1, which change the value of the data. However, the original data is shared by d1 and d2. Thus, we need to minus one from the value of the ref_count and create a new ref_count for the new data. (The address of the ref_count indicates the success of the operation toward ref_count in the suffix increment function) Next, we do the prefix decrement to d2 and assign the result to DataBlob b3. As the value of the ref_count of this data is one, there is no need to create a new ref_count for the data after decrement. (The address of the ref_count indicates the success of the operation toward ref_count in the suffix decrement function.) By the way, as it is suffix increment and suffix decrement, the data of the d3 is the result after the operations.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 2 0x55f9632e8f10
1 2 3
4 5 6
7 8 9

d2:
row: 3 col: 3 channel: 1
ref_count: 2 0x55f9632e8f10
1 2 3
4 5 6
7 8 9

zaizheli
++d1:
row: 3 col: 3 channel: 1
ref_count: 2 0x55f9632e9370
2 3 4
5 6 7
8 9 10

d3 = ++d1
row: 3 col: 3 channel: 1
ref_count: 2 0x55f9632e9370
2 3 4
5 6 7
8 9 10

--d2:
row: 3 col: 3 channel: 1
ref_count: 2 0x55f9632e8f10
0 1 2
3 4 5
6 7 8

d3 = --d2
row: 3 col: 3 channel: 1
ref_count: 2 0x55f9632e8f10
0 1 2
3 4 5
6 7 8
```

Suffix:

We first create two DataBlobs that share the same data. Then we do suffix increment to one DataBlob and do suffix decrement to another DataBlob.

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs d2(3,3,1,data1,d1.getRefCount());
DataBlobs<int> d3;
cout << "d1: " << endl << d1;
cout << "d2: " << endl << d2;
d3 = d1++;
cout << "d1++: " << endl << d1;
cout << "d3 = d1++ " << endl << d3;
d3 = d2--;
cout << "d2--: " << endl << d2;
cout << "d3 = d2-- " << endl << d3;
```

The results are as follows. This is almost the same as the prefix one. The only difference is that the d3 is getting the data before the operations.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 2 0x5571d9872ee0
1 2 3
4 5 6
7 8 9

d2:
row: 3 col: 3 channel: 1
ref_count: 2 0x5571d9872ee0
1 2 3
4 5 6
7 8 9

++d1:
row: 3 col: 3 channel: 1
ref_count: 2 0x5571d9873340
2 3 4
5 6 7
8 9 10

d3 = ++d1
row: 3 col: 3 channel: 1
ref_count: 2 0x5571d9873340
2 3 4
5 6 7
8 9 10

--d2:
row: 3 col: 3 channel: 1
ref_count: 2 0x5571d9872ee0
0 1 2
3 4 5
6 7 8

d3 = --d2
row: 3 col: 3 channel: 1
ref_count: 2 0x5571d9872ee0
0 1 2
3 4 5
6 7 8
```

==, != operators:

We create two same DataBlobs and check if they are the same using the == and!= operators.

```
DataBlobs d1(3,3,1,data1,NULL);
DataBlobs d2(d1);
if(d1 == d2)
    cout << "d1 == d2" << endl;
else
    cout << "d1 != d2" << endl;
if(d1 != d2)
    cout << "d1 != d2" << endl;
else
    cout << "d1 == d2" << endl;
```

The results are as follows. They are correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1 == d2
d1 == d2
```

<< operator:

We create a DataBlob and output the information of this DataBlob.

```
DataBlobs d1(3,3,1,data1,NULL);
cout << d1;
```

The result is as follows. It is correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
row: 3 col: 3 channel: 1
ref_count: 1 0x5638b87def10
1 2 3
4 5 6
7 8 9
```

() operator:

We create a DataBlob and use () operator to get access the the data on the specific position.

```
DataBlobs d1(3,3,1,data1,NULL);
cout << "d1: " << endl << d1;
cout << "d1(0,0,1): " << d1(0,0,1) << endl;
cout << "d1(1,1,1): " << d1(1,1,1) << endl;
```

The results are as follows. They are correct.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
d1:
row: 3 col: 3 channel: 1
ref_count: 1 0x5621a6599f10
1 2 3
4 5 6
7 8 9

d1(0,0,1): 1
d1(1,1,1): 5
```

3.1.3 Convolution Functions

In order to check the correctness of our convolution operations. We do the convolution operation using different functions, output the results, and compared qualitatively with the convolution operation result in MATLAB.

When doing the convolution operation of the 3 channels image, we choose to separate the data of the 3 channels first and use the specific kernel to do the convolution operation with the specific channel of data. After finishing the 3 convolution operations, we combine the data together in order to output the image correctly.

The reasons why we choose to do this in this way are as follows: First of all, when doing the convolution operation of 3-channel data, it is common to have different kernels for the different channels of the data. Secondly, the code of the convolution function will be more simple.

For the normal convolution:

```
float* kernel_data = new float[9]{-1, 0 ,1, -2 ,0, 2, -1, 0, 1};
DataBlobs kernel(3,3,1,kernel_data,NULL);
DataBlobs<float> input("gray.jpg");
DataBlobs<float> r1,g1,b1,r2,g2,b2,r3,g3,b3,Output;
input.getRed(r1);
input.getGreen(g1);
input.getBlue(b1);
DataBlobs<float>::padding(r1,3,3,r2);
DataBlobs<float>::padding(g1,3,3,g2);
DataBlobs<float>::padding(b1,3,3,b2);
DataBlobs<float>::cnn(r2,kernel,r3);
DataBlobs<float>::cnn(g2,kernel,g3);
DataBlobs<float>::cnn(b2,kernel,b3);
DataBlobs<float>::CombineRGB(r3,g3,b3,Output);
Output.output("cnn_output.jpg");
```

The result is as follows:

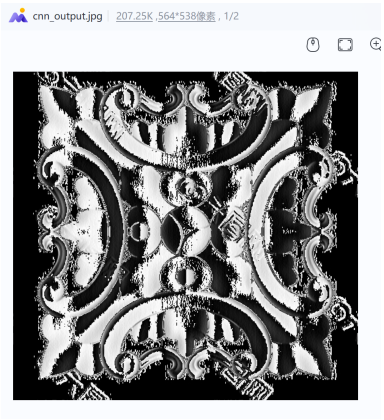


Figure 3: Normal Convolution

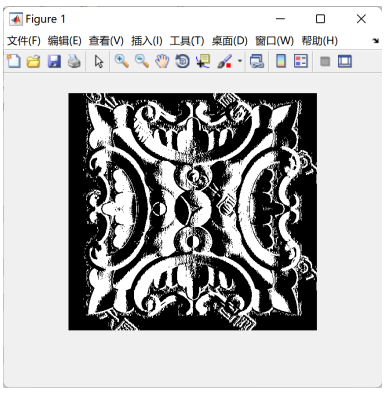


Figure 4: MATLAB Convolution

For the unloop function:

```
float* kernel_data = new float[9]{-1, 0 ,1, -2 ,0, 2, -1, 0, 1};
DataBlobs kernel(3,3,1,kernel_data,NULL);
DataBlobs<float> input("gray.jpg");
DataBlobs<float> r1,g1,b1,r2,g2,b2,r3,g3,b3,Output;
input.getRed(r1);
input.getGreen(g1);
```

```
input.getBlue(b1);
DataBlobs<float>::padding(r1,3,3,r2);
DataBlobs<float>::padding(g1,3,3,g2);
DataBlobs<float>::padding(b1,3,3,b2);
DataBlobs<float>::cnn_unloop(r2,kernel,r3);
DataBlobs<float>::cnn_unloop(g2,kernel,g3);
DataBlobs<float>::cnn_unloop(b2,kernel,b3);
DataBlobs<float>::CombineRGB(r3,g3,b3,Output);
Output.output("cnn_unloop_output.jpg");
```

The result is as follows:

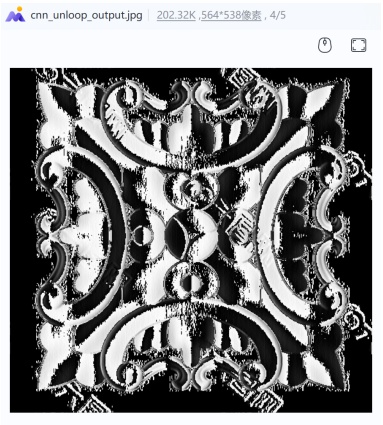


Figure 5: Unloop Convolution

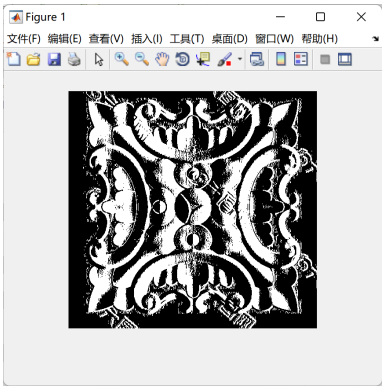


Figure 6: MATLAB Convolution

For SIMD function:

```
float* kernel_data = new float[9]{-1, 0 ,1, -2 ,0, 2, -1, 0, 1};
DataBlobs kernel(3,3,1,kernel_data,NULL);
DataBlobs<float> input("gray.jpg");
DataBlobs<float> r1,g1,b1,r2,g2,b2,r3,g3,b3,Output;
input.getRed(r1);
input.getGreen(g1);
input.getBlue(b1);
DataBlobs<float>::padding(r1,3,3,r2);
DataBlobs<float>::padding(g1,3,3,g2);
DataBlobs<float>::padding(b1,3,3,b2);
DataBlobs<float>::cnn_avx2_ker3x3(r2,kernel,r3);
DataBlobs<float>::cnn_avx2_ker3x3(g2,kernel,g3);
DataBlobs<float>::cnn_avx2_ker3x3(b2,kernel,b3);
DataBlobs<float>::CombineRGB(r3,g3,b3,Output);
Output.output("cnn_simd_output.jpg");
```

The result is as follows:

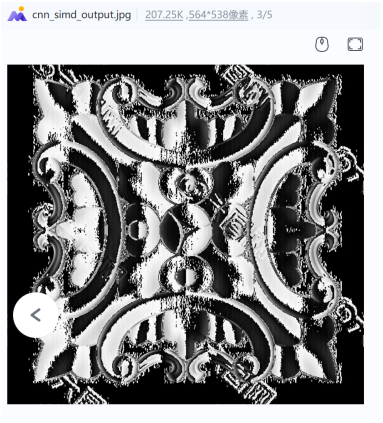


Figure 7: SIMD Convolution

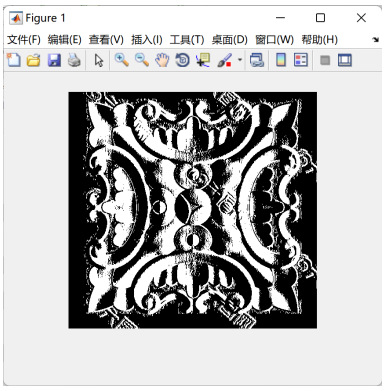


Figure 8: MATLAB Convolution

For img2Col function:

```
float* kernel_data = new float[9]{-1, 0 ,1, -2 ,0, 2, -1, 0, 1};
DataBlobs kernel(3,3,1,kernel_data,NULL);
DataBlobs<float> input("gray.jpg");
DataBlobs<float> r1,g1,b1,r2,g2,b2,r3,g3,b3,Output;
input.getRed(r1);
input.getGreen(g1);
input.getBlue(b1);
DataBlobs<float>::rearrange_img2Col(r1,3,3,r2);
DataBlobs<float>::rearrange_img2Col(g1,3,3,g2);
DataBlobs<float>::rearrange_img2Col(b1,3,3,b2);
DataBlobs<float>::cnn_img2Col(r2,input.getRow(),input.getCol(),kernel,r3);
DataBlobs<float>::cnn_img2Col(g2,input.getRow(),input.getCol(),kernel,g3);
DataBlobs<float>::cnn_img2Col(b2,input.getRow(),input.getCol(),kernel,b3);
DataBlobs<float>::CombineRGB(r3,g3,b3,Output);
Output.output("cnn_img2Col_output.jpg");
```

The result is as follows:

Figure 9: Img2Col Convolution

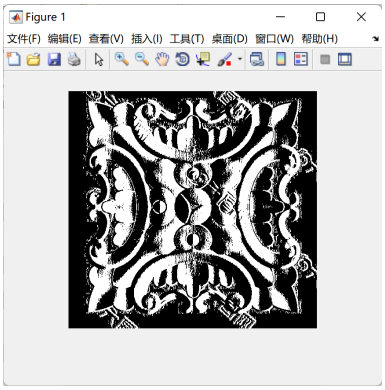


Figure 10: MATLAB Convolution

For the MATLAB convolution:

```
% Read the image
img = imread('gray.jpg');

% define the kernel
kernel = [-1 0 1; -2 0 2; -1 0 1];

% convolution operation
result = conv2(img, kernel, 'same');

% Show the result
imshow(result);
```

Based on the comparison between the convolution operations in the DataBlibs class and the convolution operations in MATLAB. We can deduce that our convolution-related functions are all correct. Moreover, as the image is 3 channels (recognize by the imread() of OpenCv), we can also deduce the functions related to the Datablob of 3 channels data are correct.

3.1.4 Auxiliary Functions

Get basic information about the data blob:

```
DataBlobs d1(3,3,1,data1,NULL);
cout << "row: " << d1.getRow() << " col: " << d1.getCol() << " channel: " << d1.getChannel() << endl;
cout << "ref_count: " << *(d1.getRefCount()) << " " << d1.getRefCount() << endl;
for(int i = 0; i < 9; i++)
    cout << *(d1.getData()+i) << " ";
cout << endl;
```

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code$ ./a.out
row: 3 col: 3 channel: 1
ref_count: 1 0x5560a7885f10
1 2 3 4 5 6 7 8 9
```

Get the total number of Data Blobs:

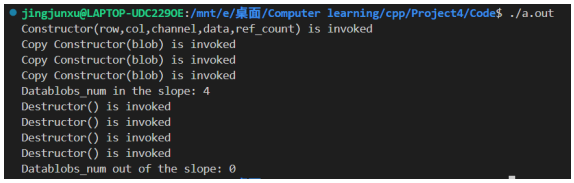
```
DataBlobs data_int(3,3,1,data1,NULL);
DataBlobs data_int1(data_int);
DataBlobs data_int2(data_int);
DataBlobs data_float(3,3,1,data2,NULL);
DataBlobs data_float1(data_float);
DataBlobs data_double(3,3,1,data3,NULL);
DataBlobs data_short(3,3,1,data4,NULL);
DataBlobs data_short1(data_short);
cout << "Datablobs<short> num: " << DataBlobs<short>::getDataBlobsNum() << endl;
cout << "Datablobs<int> num: " << DataBlobs<int>::getDataBlobsNum() << endl;
cout << "Datablobs<float> num: " << DataBlobs<float>::getDataBlobsNum() << endl;
cout << "Datablobs<double> num: " << DataBlobs<double>::getDataBlobsNum() << endl;
```

Check if the data blob is empty:

```
DataBlobs<float> blob1;
if(blob1.empty())
    cout << "blob1 is empty" << endl;
else
    cout << " blob1 is not empty" << endl;
DataBlobs blob2(3,3,1,data1,NULL);
if(blob2.empty())
    cout << "blob2 is empty" << endl;
else
    cout << "blob2 is not empty" << endl;
```

3.1.5 Destructor

```
{
    DataBlobs blob1(3,3,1,data1,NULL);
    DataBlobs blob2(blob1);
    DataBlobs blob3(blob2);
    DataBlobs blob4(blob3);
    cout << "Datablobs_num in the slope: " << DataBlobs<int>::getDataBlobsNum() << endl;
}
cout << "Datablobs_num out of the slope: " << DataBlobs<int>::getDataBlobsNum() << endl;
```



3.2 Compare the speed of different convolution operations

In this experiment, we compare the speed of the convolution operation between the SIMD function and the img_2Col function. (The algorithms of these two functions are mentioned in the Code Implement part)

Since this program is running under Linux environment, we use the Linux environment timing function gettimeofday() to time the program. The timing function has a high accuracy of microseconds. This function gets the elapsed time and time zone (UTC time) from January 1, 1970 to the present, but according to the official Linux documentation, the time zone is no longer used, so just pass NULL when using it.

The difference between the current time before the convolution operation and the current time after the convolution operation is completed is the difference between them, which is the program's running time.

```
#include <sys/time.h>
struct timeval start, end;
gettimeofday(&start, NULL);
//Convolution operation
gettimeofday(&end, NULL);
printf("Time Cost : %lf s\n",(end.tv_sec - start.tv_sec) + (double)(end.tv_usec - start.tv_usec) / 1000000.0);
```

We generate the corresponding data blobs according to the size of the specified data scale, respectively 128x128, 256x256, 512x512, 1024x1024, 2056x2056, and 4096x4096. We do convolution operations among the normal, SIMD, and img2Col convolution functions in the case of a 3x3 kernel.

The results are shown in the following image.

朴素卷积，SIMD卷积，img2Col卷积平均时间						
	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
cnn	0.0017263	0.006564	0.0271464	0.0702042	0.2535935	0.906978
cnn_SIMD	0.0003218	0.0012584	0.0049715	0.0162583	0.0661326	0.2241817
cnn_img2Col	0.0018999	0.0088069	0.00512286	0.0269957	0.1756575	0.423265

In order to see the results more clearly, we use the formula below to calculate the speed improvement rate of the two functions. (Suppose η is the Speed improvement rate, x_1 is the time cost of the normal convolution function and x_2 is the time cost of the Research Subject.)

$$\eta = (x_1 - x_2) \div x_1 \times 100\%$$

SIMD与img2Col对卷积的速度提升率的比较						
	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
cnn_SIMD	0.813589758	0.808287629	0.816863378	0.76841414	0.739218079	0.752825647
cnn_img2Col	-0.100561895	-0.341697136	0.811287685	0.615468875	0.307326489	0.533323851

From the results, we can find out that the average speed improvement of the SIMD function is approximately 80%, while the performance of the img2Col is neither stable nor satisfied when compared with the SIMD function.

There are some disadvantages of the img2Col. First of all, it needs to rearrange the data. The process of rearranging the data is very time-costing as it is accessing the data in discontinuous memory. Moreover, the rearrangement of the data is almost like the whole process of the convolution operation. (without the calculation part) Secondly, we are using the template to implement the class as well as the member functions in this project. Thus, it needs some time to transform the data type of the data into a float in order to satisfy the parameter requirement of the function cblas_sgemm(). Although openblas library has provided different functions for different data types, it is very difficult to recognize the data type of the data inside the function. Another way is to implement the specialization of the template class. However, this is too tedious to rewrite so many functions.

SIMD also has limits. A single SIMD function is not universal to every kernel, it can only deal with a specific size of the kernel.

In conclusion, according to my research up to date, the best way to speed up the convolution operation is the SIMD way. Since the commonly used size of the kernel is not varied (usually 3x3 or 5x5), the limit of the SIMD can be ignored. Anyway, the img2Col does have its own value as it does speed up the convolution operation in some circumstances and the time cost of rearranging the data may be ignored if the rearranged data is used in many convolution operations which overall have a much larger time cost.

3.3 Comparison under x86 and ARM

3.3.1 Compare the speed of the matrix multiplication

In this experiment, we compare the speed of the matrix multiplication between the x86 and the arm, which aims to compare the computational capability between the two CPU architectures.

The original idea of the experiment was first to use the Random() function to generate different sizes of the random float data and generate the DataBlobs using the data. Then we do the matrix multiplication and calculate the time cost. Finally, we write the result into a CSV file. Compare the time cost between x86 and arm and make some conclusions.

```cpp
#include <fstream>
int main()
{
    ofstream outFile;
    outFile.open("TimeCost_x86.csv", std::ios::out | std::ios::trunc);
    for(int num = 1; num<=10; num++)
    {
        float* data1 = Random(4096,4096);
        float* data2 = Random(4096,4096);
        DataBlobs d1(4096,4096,1,data1,NULL);
        DataBlobs d2(4096,4096,1,data2,NULL);
        outFile << num << ",";
        struct timeval start, end;
        gettimeofday(&start, NULL);
        d1 * d2;
        gettimeofday(&end, NULL);
        outFile << (end.tv_sec - start.tv_sec) + (double)(end.tv_usec - start.tv_usec) / 1000000.0 << endl;
    }
    outFile.close();
    return 0;
}
```

However, I fail to find a computer with an arm CPU architecture. The following is my prediction of the results and the analysis of this experiment based on the information online.

The computational capability of x86 is likely to be better than the arm. The reasons are as follows.

Firstly, x86 and arm have different Instruction sets. The x86 architecture uses the Complex Instruction Set Computer (CISC) instruction set, while ARM uses the Reduced Instruction Set Computer (RISC) instruction set. When it comes to matrix multiplication, x86 architectures typically have a richer instruction set, including instructions for vectorization and parallel computation. This may result in some cases where x86 can perform matrix multiplication more efficiently.

Secondly, the data type in this experiment is float, different x86, and ARM processors may have different floating point performances. In general, x86 processors typically offer higher performance in floating-point computing, especially server-class processors optimized for scientific computing and large-scale data processing. However, some high-performance ARM processors are also capable of delivering fairly good floating-point performance.

3.3.2 Compare the speed of SIMD convolution operation

In this experiment, we compare the speed of the SIMD convolution operation between the x86 and the arm, which aims to compare the SIMD differences between the two CPU architectures.

The original idea of the experiment was first to use the Random() function to generate different sizes of the random float data and generate the DataBlobs using the data. Then we do the SIMD convolution operation and calculate the time cost. Finally, we write the result into a CSV file.

```cpp
ofstream outFile;
outFile.open("TimeCost_x86.csv", std::ios::out | std::ios::trunc);
float* kernel_data = new float[9]{-1, 0 ,1, -2 ,0, 2, -1, 0, 1};
DataBlobs kernel(3,3,1,kernel_data,NULL);
DataBlobs<float> res;
for(int num = 1; num<=10; num++)
{
    float* data1 = Random(128,128);
    DataBlobs d1(128,128,1,data1,NULL);
    outFile << num << ",";
    struct timeval start, end;
    gettimeofday(&start, NULL);
    DataBlobs<float>::cnn_avx2_ker3x3(d1,kernel,res);
    gettimeofday(&end, NULL);
    outFile << (end.tv_sec - start.tv_sec) + (double)(end.tv_usec - start.tv_usec) / 1000000.0 << endl;
}
outFile.close();
```

However, I fail to find a computer with an arm CPU architecture. The following is my prediction of the results and the analysis of this experiment based on the information online.

The SIMD of x86 is likely to be better than the arm. The reason may be that avx2 and neon have different vector widths. The AVX2 has a vector width of 256 bits, while NEON's vector width is typically 128 bits. This means that AVX2 can process more data at the same time in one instruction execution, and thus may have higher throughput in a given case.

4 Difficulties & Solutions

4.1 Difficulties in memory management

The function that does some operation to the data of the DataBlob or copies data from one DataBlob to another requires some operation toward the parameter ref_count.

In the function that does some operations to the data of the DataBlob. Firstly, if the value of the ref_count is one, there is no need to create a new ref_count. Using the original one makes no difference. Secondly, if the value of the ref_count is bigger than one, then we need to minus one from the value of the original ref_count and create a new size_t, whose value is one, to the ref_count. Particularly noteworthy is that we need to set the value of the ref_count to NULL before assigning a new size_t to it. If you do not do so, it is likely that the address of the new size_t has the same address as the original ref_count, which causes the problem.

```
//The code before
--*(this->ref_count);
this->ref_count = new size_t(1);

//The code after
--*(this->ref_count);
this->ref_count = NULL;
this->ref_cout = new size_t(1);
```

In the function that copies data from a DataBlob to another. The operations toward the ref_count are similar, except we should assign the ref_count from the DataBlob being copied to the Datablob and add one to the value of the ref_count.

4.2 Difficulties in invoking the OpenBlas library

Invoking the OpenBlas library has some difficulties in downloading the library, linking the library, and using the functions of the library correctly.

We use the input of the instruction in the terminal to download the Openblas library in Ubuntu.

```
sudo apt update
sudo apt install libopenblas-dev
```

To import the Openblas library in C++, we include 'cblas.h' head file. Then when we compile the file, we should add '-lopenblas' to link the library.

The parameters of the cblas_sgemm() function are very long. The definition of the function is as follows.

```
void cblas_sgemm(
    const enum CBLAS_ORDER Order,
    const enum CBLAS_TRANSPOSE TransA,
    const enum CBLAS_TRANSPOSE TransB,
    const int M,
    const int N,
    const int K,
    const float alpha,
    const float *A,
    const int lda,
    const float *B,
    const int ldb,
    const float beta,
    float *C,
    const int ldc
);
```

Order: Specifies the storage order of the matrices. It can be either CblasRowMajor or CblasColMajor.

TransA: Specifies whether matrix A should be transposed before the multiplication. It can be CblasNoTrans for no transpose, CblasTrans for transpose, or CblasConjTrans for conjugate transpose.

TransB: Specifies whether matrix B should be transposed before the multiplication. It follows the same options as TransA.

M: Specifies the number of rows of matrix A and matrix C.

N: Specifies the number of columns of matrix B and matrix C.

K: Specifies the number of columns of matrix A and the number of rows of matrix B.

alpha: Specifies the scalar value to scale the matrix product.

A: Pointer to the first element of matrix A.

lda: Specifies the leading dimension (stride) of matrix A. If Order is CblasRowMajor, lda should be at least K. If Order is CblasColMajor, lda should be at least M.

B: Pointer to the first element of matrix B.

ldb: Specifies the leading dimension (stride) of matrix B. If Order is CblasRowMajor, ldb should be at least N. If Order is CblasColMajor, ldb should be at least K.

beta: Specifies the scalar value to scale matrix C before adding the matrix product.

C: Pointer to the first element of matrix C.

ldc: Specifies the leading dimension (stride) of matrix C. If Order is CblasRowMajor, ldc should be at least N. If Order is CblasColMajor, ldc should be at least M.

4.3 Difficulties in invoking the OpenCV library

Invoking the OpenCV library has some difficulties in downloading the library and linking the library.

We use the input of the instruction in the terminal to download the OpenCV library in Ubuntu.

```
sudo apt update
sudo apt install libopencv-dev
```

To import the Openblas library in C++, we include 'opencv2/opencv.hpp' head file. Then when we compile the file, we should add 'pkg-config –cflags –libs opencv4' to link the library.

4.4 Difficulties in writing CMakeLists.txt

In this project, we invoke some libraries to help us implement some specific features and use several extra commands in the terminal to implement the SIMD operation. Thus, it is tedious to write all the commands in the command window by ourselves. Writing a CMakeLists.txt is a good way to solve this problem.

However, I failed to link the openblas library using CMakeLists.txt. When I use the command find_package(OpenBLAS REQUIRED) and target_link_libraries(pro OpenBLAS::OpenBLAS) to find the openblas package and link the openblas, there are some error messages. The error messages are shown below.

```
jingjunxu@LAPTOP-UDC229OE:/mnt/e/桌面/Computer learning/cpp/Project4/Code/build$ cmake ..
CMake Error at CMakeLists.txt:15 (find_package):
  By not providing "FindOpenBLAS.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "OpenBLAS",
  but CMake did not find one.

  Could not find a package configuration file provided by "OpenBLAS" with any
  of the following names:

    OpenBLASConfig.cmake
    openblas-config.cmake

  Add the installation prefix of "OpenBLAS" to CMAKE_PREFIX_PATH or set
  "OpenBLAS_DIR" to a directory containing one of the above files.  If
  "OpenBLAS" provides a separate development package or SDK, be sure it has
  been installed.


-- Configuring incomplete, errors occurred!
```

Here is my CMakeLists.txt.

```
cmake_minimum_required(VERSION 3.12)

add_definitions(-DWITH_AVX2)
add_definitions(-mavx)
#add_definitions(-O3)

```

```
set(CMAKE_CXX_STANDARD 17)

project(pro)

ADD_EXECUTABLE(pro main.cpp)

find_package(OpenBLAS REQUIRED)
target_link_libraries(pro OpenBLAS::OpenBLAS)

find_package(OpenCV REQUIRED)
if (OpenCV_FOUND)
    message("OpenCV Found")
    include_directories(${OpenCV_INCLUDE_DIRS})
    target_link_libraries(pro ${OpenCV_LIBS})
else()
    message("Can't find OpenCV library")
endif()

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    message("OpenMP found.")
    target_link_libraries(pro OpenMP::OpenMP_CXX)
endif()
```

4.5 Difficulties in checking the correctness of the Convolution operation

Initially, I want to use the filter2D() function in OpenCV to check the correctness of my convolution operation function. However, the result of the filter2D() function is very different from my own convolution functions and also different from the result of MATLAB.

For the OpenCV convolution:

```
cv::Mat image = cv::imread("gray.jpg");
 cv::Mat Kernel = (cv::Mat_<float>(3, 3) << -1, 0, 1,
                                            -2, 0, 2,
                                            -1, 0, 1);
cv::Mat result;
cv::filter2D(image, result, -1, Kernel);
imwrite("opencv_output.jpg", result);
```

For the MATLAB convolution:

```
% Read the image
img = imread('gray.jpg');

% define the kernel
kernel = [-1 0 1; -2 0 2; -1 0 1];

% convolution operation
result = conv2(img, kernel, 'same');

% Show the result
imshow(result);
```

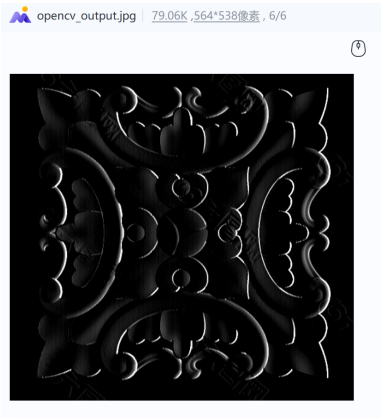


Figure 11: OpenCV Convolution

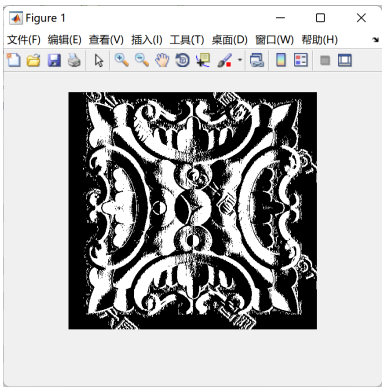


Figure 12: MATLAB Convolution

The result from the filter2D() in OpenCV is very different from the result from the conv2() in MATLAB and I can not figure out the reasons for the differences. The possible reasons are as follows.

Data type: OpenCV and MATLAB may use different default data types to represent images.

Border handling: When performing the convolution operation, it is necessary to handle the borders of the image. OpenCV and MATLAB may employ different default strategies for handling border pixels. OpenCV typically uses border pixel replication, while MATLAB may use zero-padding or symmetric padding. This can result in different convolution results at the image borders.

Convolution order: OpenCV and MATLAB may have different default orders for the convolution operation. In OpenCV's filter2D function, the default convolution order is vertical-first. In MATLAB's conv2 function, the default convolution order is horizontal-first. This difference in order can lead to discrepancies in the results.

5 Brief Summary

I gained a lot from this project.

First of all, I learn how to design a class considering both the performance and the safety. For the intelligence of the class, a variety of constructors were designed and various operators were overloaded to make the class more user-friendly. For the safety of the class, various scenarios are taken into account to prevent memory leaks, prevent memory from being freed multiple times in the circumstance of soft copy, and detect the total number of data blocks in the program to ensure that there are no remaining blocks of memory that are not freed when all the memory needs to be freed.

Secondly, this project has helped me to practice my ability to import other libraries and use them. In this project, I imported Openblas and OpenCV libraries. I used Openblas to optimize the matrix multiplication, and OpenCV to read and write the image data and do the convolution operation.

Thirdly, I continued my research from the last project to study how to speed up the convolution operation. In this project, I tried a new algorithm called img2Col, which is to convert the process of convolution into the calculation of matrix multiplication. This project uses existing libraries named openblas for the optimization of matrix multiplication to speed up the process of convolution. I compared this algorithm with the SIMD implementation of my last project and found that although SIMD has some limitations, its stability and acceleration performance is much better than that of img2Col. As far as my current research is concerned, SIMD is the optimal way to implement convolution.

Finally, I compared the speed of implementing matrix multiplication in x86 and arm architectures with the speed of implementing SIMD, in order to compare the differences in the computational ability and the performance of SIMD implementation between the two CPU architectures. The process of the research deepens my understanding of both x86 and arm CPU architectures.