Contents

1 Requirement Analysis

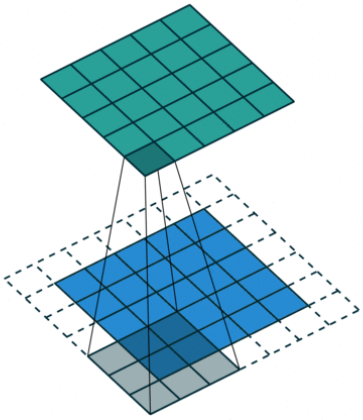
The concise purpose of the project is to design a function that implements the Convolutional Layer in Deep learning using C and to improve the efficiency of the program.

1.1 Background Knowledge

In a convolutional layer, multiple convolution kernels are typically set up. Each kernel is a small matrix (contains data) used for performing a convolution operation on the input data. The size of the kernel is the size of the matrix.

Padding refers to the process of adding extra ”padding” pixels (typically added with a value of zero) around the input image before performing convolution. (The dashed squares in the figure) The purpose of padding is to control the size of the output feature maps and preserve spatial information at the borders of the input image.

Stride refers to the number of pixels the kernel is moved in each step while performing convolution operation on the input image. The use of stride helps in reducing the size of the output feature map and also helps in controlling the computational complexity of the model. A stride of 1 means the kernel is shifted by one pixel at a time, and hence the output size will be the same as the input size. A stride of 2 means the kernel is shifted by two pixels at a time, resulting in a smaller output size.



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.

1.2 Design Data Structure

The Data structures named Image and Kernel are designed.

Image: The Image data structure contains 4 parts. The height, width, channels, and the data of an image. Considering using SIMD to improve the convolution operation of the images of channel 3, three float pointers storing the image data are set in the data structure. SIMD requires to get continuous data stored in the memory. If we store data of RGB adjacent to each other, it costs a lot to continuously restore the data during the convolution operation. (If the Channel is 1, the data2 and data3 will be set as NULL)

```
typedef struct {
    size_t width;
    size_t height;
    size_t channels;
    float* data1;
    float* data2;
    float* data3;
} Image;
```

Kernel: The Kernel data structure contains also 4 parts. The height, width, channels, and the data of the kernel. When implementing the convolution operation of the images of channel 3, the kernel data used to operate each channel are different in most applications. Thus, three float pointers storing the kernel data are packaged in a data structure, making it easy to find the specific data during the convolution operation. (If the Channel is 1, the data2 and data3 will be set as NULL)

```
typedef struct {
    size_t width;
    size_t height;
    size_t channels;
    float* data1;
    float* data2;
    float* data3;
} Kernel;
```

1.3 Convolution Function

First of all, we need to read an image (In this project, we choose to read jpg image). Secondly, we initialize the kernel. Thirdly, we pad the input image according to the size of the kernel. (The purpose is to make the output the same size as the input) Fourthly, we choose the relatively efficient way to implement the Convolution operation. Finally, we output the image in jpg form.

1.4 Testing the correctness

In this project, we use the convolution operation in MATLAB to test the correctness of my own convolution operation qualitatively.

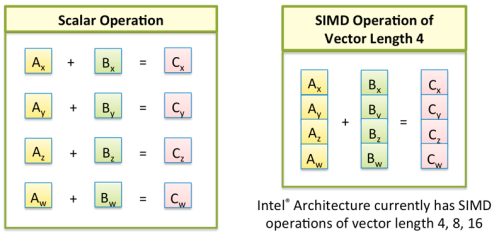
In MATLAB, we can use the imshow() function to show the image after the convolution operation. Then we can see if that image is as same as our own output image.

1.5 Improve the efficiency

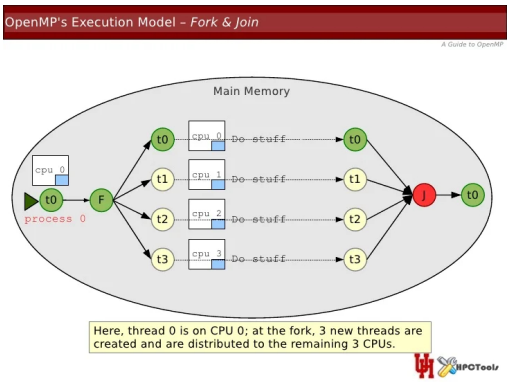
This project mainly improves the efficiency of the convolution operation in three ways: Reducing the number of looping, SIMD (single instruction multiple Data), and OpenMP (Open Multi-processing).

Loop costs a lot. Each loop requires repeated calculations and comparisons, operations that take up processor time and slow down program execution. In addition, loops need to store multiple variables and intermediate results in memory, which can increase memory usage and reduce program efficiency. Thus, lowering the dimension of the looping definitely improves the efficiency of the program.

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.



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.



2 Code Implementation

2.1 Image Operation

In this project, we choose to use a library called libjpeg in C to input and output an jpg image.

```
#include <jpeglib.h>
```

2.1.1 Input the Image

Firstly, we open the file, create jpeg-decompress-struct (decompress the image), and create jpeg-error-mgr (manage error handling during JPEG decompression). Secondly, we set some basic parameters to the jpeg-decompress-struct and start the compression. Thirdly, we store the data row by row and set some basic information to the image structure. Finally, we finish the decompression, close the file and free some memories.

```
bool input_jpeg_image(const char* filename, Image* image)
{
    if(image==NULL) {
        printf("Error Image(structure) input!");
```

```
        return false;
    }
    //Open JPEG file, create JPEG decoder object and JPEG error object
    FILE *infile;
    if ((infile = fopen(filename, "rb")) == NULL) {
        printf("Error opening JPEG file!");
        return false;
    }
    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;

    //Set basic parameters
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_decompress(&cinfo);
    jpeg_stdio_src(&cinfo, infile);
    jpeg_read_header(&cinfo, TRUE);
    //Start decompression
    jpeg_start_decompress(&cinfo);

    //Passing data into the Image structure
    int row_stride = cinfo.output_width * cinfo.output_components;
    unsigned char *buffer = (unsigned char*)malloc(row_stride*sizeof(unsigned char));
    if(cinfo.output_components==1)
    {
        cinfo.out_color_space = JCS_GRAYSCALE;
        cinfo.quantize_colors = FALSE;
        float *Data = (float*)aligned_alloc(256,row_stride*cinfo.output_height*sizeof(float));
        while (cinfo.output_scanline < cinfo.output_height) {
            int i = cinfo.output_scanline;
            jpeg_read_scanlines(&cinfo, &buffer, 1);
            for(int j=0;j<row_stride;j++)
                Data[i*row_stride+j]=(float)buffer[j];
        }
        image->data1 = Data;
        image->data2 = NULL;
        image->data3 = NULL;
    }
    else if(cinfo.output_components==3)
    {
        cinfo.out_color_space = JCS_RGB;
        float* red_channel = (float*)aligned_alloc(256,cinfo.output_width * cinfo.output_height * sizeof(float));
        float* green_channel = (float*)aligned_alloc(256,cinfo.output_width * cinfo.output_height * sizeof(float))
        float* blue_channel = (float*)aligned_alloc(256,cinfo.output_width * cinfo.output_height * sizeof(float));
        while (cinfo.output_scanline < cinfo.output_height)
        {
            jpeg_read_scanlines(&cinfo, &buffer, 1);
            int offset = (cinfo.output_scanline - 1) * cinfo.output_width;
            for (int i = 0, j = 0; i < row_stride; i += 3, j++) {
                red_channel[offset + j] =(float) buffer[i];
                green_channel[offset + j] =(float) buffer[i + 1];
                blue_channel[offset + j] =(float) buffer[i + 2];
            }
        }
        image->data1 = red_channel;
        image->data2 = green_channel;
        image->data3 = blue_channel;
    }
    else
    {
        printf("The channels of the image must be 1 or 3");
        return false;
    }
    image->width = cinfo.output_width;
    image->height = cinfo.output_height;
    image->channels = cinfo.output_components;

    //Complete the decompression process and release all related resources
    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);
    fclose(infile);
    free(buffer);

    return true;
}
```

2.1.2 Output the Image

Firstly, we create jpeg-compress-struct (compress the image), create jpeg-error-mgr (manage error handling during JPEG decompression) and open a file. Secondly, we connect the output stream with a compressed object, set some basic parameters to the jpeg-compress-struct, and start the compression. Thirdly, we transfer the data row by row and set some basic information to the compressor. Finally, we finish the compression, output the image, close the file and free some memories.

```
bool output_jpeg_image(const char* filename, Image* image)
{
    if(image==NULL) {
        printf("Error Image(structure) input!");
        return false;
    }
    //JPEG,JPEG
    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_compress(&cinfo);
    FILE* outfile;
    if ((outfile = fopen(filename, "wb")) == NULL) {
        printf("Error opening JPEG file!");
        return false;
    }

    //
    jpeg_stdio_dest(&cinfo, outfile);

    //
    cinfo.image_width = image->width;
    cinfo.image_height = image->height;
    cinfo.input_components = image->channels;
    jpeg_set_defaults(&cinfo);
    jpeg_set_quality(&cinfo, 90, TRUE);
    //
    jpeg_start_compress(&cinfo, TRUE);

    //image
    JSAMPROW row_pointer[1];
    int row_stride = cinfo.image_width * cinfo.input_components;
    unsigned char *buffer = (char*)malloc(row_stride*sizeof(unsigned char));
    if(image->channels==1)
    {
        cinfo.in_color_space = JCS_GRAYSCALE;
        while (cinfo.next_scanline < cinfo.image_height) {
            int i = cinfo.next_scanline;
            for(int j=0;j<row_stride;j++)
                buffer[j]=(unsigned char)image->data1[i*row_stride+j];
            row_pointer[0] = &buffer[0];
            jpeg_write_scanlines(&cinfo, row_pointer, 1);
        }
    }
    else if(image->channels==3)
    {
        cinfo.in_color_space = JCS_RGB;
        while (cinfo.next_scanline < cinfo.image_height) {
            int offset = cinfo.next_scanline * cinfo.image_width;
            for (int i = 0, j = 0; i < row_stride; i += 3, j++) {
                buffer[i] = (unsigned char)image->data1[offset + j];
                buffer[i + 1] =(unsigned char)image->data2[offset + j];
                buffer[i + 2] =(unsigned char)image->data3[offset + j];
            }
            row_pointer[0] = &buffer[0];
            jpeg_write_scanlines(&cinfo, row_pointer, 1);
        }
    }
    else
    {
        printf("The channels of the image must be 1 or 3");
        return false;
    }
    //
    jpeg_finish_compress(&cinfo);
    fclose(outfile);
    jpeg_destroy_compress(&cinfo);

    return true;
}
```

2.2 Kernel Initialization

We transfer the basic information of a kernel to the function and initialize the kernel through the function.

```
bool kernel_init(Kernel* kernel, const size_t height, const size_t width, const size_t ch,
const float* Data1, const float* Data2, const float* Data3)
{
    if(ch==1&&(Data1!=NULL))
    {
        kernel->data1 = Data1;
        kernel->data2 = NULL;
```

```
        kernel->data3 = NULL;
    }
    else if(ch==3&&(Data1!=NULL)&&(Data2!=NULL)&&(Data3!=NULL))
    {
        kernel->data1 = Data1;
        kernel->data2 = Data2;
        kernel->data3 = Data3;
    }
    else
    {
        printf("Error Input!");
        return false;
    }
    kernel->width = width;
    kernel->height = height;
    kernel->channels = ch;
    return true;
}
```

2.3 Padding

We transfer the data of the input image, the size of the input image, and the size of the kernel into the function. 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.

```
float* padding(const float* input_data, const size_t input_height,
const size_t input_width, const size_t kernel_height, const size_t kernel_width)
{
    if(input_data==NULL)
    {
        printf("NULL pointer!");
        return FALSE;
    }
    size_t pad_width = input_width + (kernel_width - 1);
    size_t pad_height = input_height + (kernel_height - 1);
    float* pad_data=(float*)calloc(pad_width*pad_height,sizeof(float));
    if(pad_data==NULL)
    {
        printf("Error Calloc for the padding data!");
        return NULL;
    }
    for(int i = ((kernel_width - 1) / 2); i < input_height + ((kernel_width - 1) / 2); i++){
        memcpy(pad_data + i * pad_width  + ((kernel_width - 1) / 2) , input_data + (i - ((kernel_width - 1) / 2))
    }
    return pad_data;
}
```

2.4 Convolution Operation

Five types of functions are mentioned in this part. They are the Ordinary Function (3 dimensions of loops), the Reduced Loops Function (2 dimensions of loops), the Unloop Function, the SIMD Function, and the SiMD with OpenMP Function.

2.4.1 Ordinary Function

```
bool cnnFunction_v1(const float* input_data, const size_t input_height, const size_t input_width, const float* ker
    const size_t kernel_width, float* output_data)
{
    //Safe Detection
    size_t pad_width=input_width+(kernel_width-1);
    for(int j = 0; j < input_height; j++)
    {
        for(int i = 0; i < input_width;i++)
        {
            for(int m = 0; m<kernel_height;m++)
            {
                for(int n = 0, k = 0; n < kernel_width; n++, k++)
                {
                    *(output_data+(j*input_width)+i) += (*(input_data + (j * pad_width) + i + (m*pad_width) + n))
                }
            }
        }
    }
    return true;
}
```

2.4.2 Reduced Loops Function

For the convolution of an element, it is necessary to traverse an entire convolution kernel and the corresponding data of the picture, which we can get the row-index = (k / kernel-width) and the column-index = (k % kernel-width). For the convolution of the whole image, which we can get the row-index = (t / input-width) and the column-index = (t % kernel-width). Thus, we can reduce the dimension of the loop from 4 to 2.

```
bool cnnFunction_v2(const float* input_data, const size_t input_height, const size_t input_width, const float* ker
    const size_t kernel_width, float* output_data)
{
    //Safe Detection
    size_t pad_width=input_width+(kernel_width-1);
    for(int t = 0; t < input_height*input_width; t++)
    {
        int j = t / input_width;
        int i = t % input_width;
        for(int k =0; k<kernel_height * kernel_width;k++)
        {
            *(output_data+(j * input_width)+i) += (*(input_data + (j * pad_width) + i + ((k / kernel_width)*pad_wi
        }
    }
    return true;
}
```

2.4.3 Unloop Function

Based on a three-dimensional loop we calculate 8 elements at one time. In order to reduce the time of calculating the repeated value (j*input-width), we set a value named stride at the top of the first loop. 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.

```
bool cnn_unloop(const float* input_data, const size_t input_height, const size_t input_width, const float* kernel_
    const size_t kernel_width, float* output_data)
{
//Safe Detection
bool cnn_unloop()
{
    for(int j = 0; j < input_height; j++)
    {
        int stride = j*input_width;
        int i = 0;
        for(; i < (input_width/8)*8;i+=8)
        {
            for(int m = 0; m<kernel_height;m++)
            {
                for(int n = 0,k = 0; n < kernel_width; n++,k++)
                {
                    *(output_data+(stride)+i) += (*(input_data + (j * pad_width) + i +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+1)) += (*(input_data + (j * pad_width) + (i+1) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+2)) += (*(input_data + (j * pad_width) + (i+2) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+3)) += (*(input_data + (j * pad_width) + (i+3) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+4)) += (*(input_data + (j * pad_width) + (i+4) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+5)) += (*(input_data + (j * pad_width) + (i+5) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+6)) += (*(input_data + (j * pad_width) + (i+6) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                    *(output_data+(stride)+(i+7)) += (*(input_data + (j * pad_width) + (i+7) +
                    (m*pad_width) + n)) * (*(kernel_data+k));
                }
            }
        }
        if((input_width % 8) != 0)
        {
            for(int w=i;w<(input_width % 8)+i;w++)
            {
                for(int m = 0; m<kernel_height;m++)
                {
                    for(int n = 0,k = 0; n < kernel_width; n++,k++)
                    {
                        *(output_data+(stride)+w) += (*(input_data + (j * pad_width) + w + m*pad_width + n)) *
                        (*(kernel_data+k));
```

```
                    }
                }
            }
        }
    }
}
return true;
}
```

2.4.4 SIMD Function

Based on a three-dimensional loop we calculate 8 elements at one time. We use the datatype _m256 and the function named _mm256_loadu_ps to get the continuous 8 floats (8*4 bytes*8 bits) in from the memory into the register. Because the size of the kernel is 3x3, thus we need to get 9 groups of data (each group has 8 floats) at once. As the image data are stored in the datatype _m256, so do the kernel data. Moreover, the data of the kernel should also be in the same size group with continuous memory allocated. Having set the data, we can use _mm256_add_ps to add and _mm256_mul_ps to multiple 2 groups of data (containing 8 floats) at once. After the operation, 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).

```
bool cnn_avx2_kernel3x3(const float* input_data, const size_t input_height, const size_t input_width,
const float* kernel_data, const size_t kernel_height,
    const size_t kernel_width, float* output_data)
{
    //Safe Detection
    size_t pad_width=input_width+(kernel_width-1);
#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;
    float*Ker11=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker12=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker13=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker21=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker22=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker23=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker31=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker32=(float*)(aligned_alloc(256,8*sizeof(float)));
    float*Ker33=(float*)(aligned_alloc(256,8*sizeof(float)));
    for(int i=0;i<8;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);
    float*temp=(float*)(aligned_alloc(256,8*sizeof(float)));
    for(size_t j=0; j<input_height;j++)
    {
        size_t i=0;
        for(; i<(input_width/8)*8; i+=8)
        {
            res =  _mm256_setzero_ps();
            im11 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (0*pad_width) + 0);
            im12 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (0*pad_width) + 1);
            im13 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (0*pad_width) + 2);
            im21 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (1*pad_width) + 0);
            im22 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (1*pad_width) + 1);
            im23 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (1*pad_width) + 2);
            im31 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (2*pad_width) + 0);
            im32 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (2*pad_width) + 1);
            im33 = _mm256_loadu_ps(input_data + (j * pad_width) + i + (2*pad_width) + 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<8;k++)
                *(output_data+(j*input_width)+i+k) = *(temp+k);
        }
        if((input_width % 8) != 0)
        {
            for(int w=i;w<(input_width % 8)+i;w++)
            {
                for(int k =0; k<kernel_height*kernel_width;k++)
                {
                    *(output_data+(j*input_width)+w) += (*(input_data +
                    (j * pad_width) + w + (k / kernel_width)*pad_width +
                    (k % kernel_width))) * (*(kernel_data+k));
                }
            }
        }
    }
    return TRUE;
#else
    printf("AVX2 is not supported");
    return FALSE;
#endif
}
```

The example is under the condition of the 3x3 kernel. There are functions written in 1x1 and 5x5 kernels in the source code. (named cnn_avx2_kernel5x5 and cnn_avx2_kernel1x1)

2.4.5 SIMD with OpenMP Function

We use the instruction 'pragma omp parallel for' before the loop to map multiple threads to different CPU cores so that different threads can execute the operation at the same time.

```
bool cnn_avx2_omp_kernel3x3(const float* input_data, const size_t input_height, const size_t input_width,
const float* kernel_data, const size_t kernel_height,
    const size_t kernel_width, float* output_data)
{
    //Safe Detection
    size_t pad_width=input_width+(kernel_width-1);
#ifdef WITH_AVX2
    //Ooperation in SIMD
    #pragma omp parallel for
    for(size_t j=0; j<input_height;j++)
    {
        size_t i=0;
        for(; i<(input_width/8)*8; i+=8)
        {
            //operations
            ......
    }
    return TRUE;
#else
    printf("AVX2 is not supported\n");
    return FALSE;
#endif
}
```

The example is under the condition of the 3x3 kernel. There are functions written in 1x1 and 5x5 kernels in the source code. (named cnn_avx2_omp_kernel5x5 and cnn_avx2_omp_kernel1x1)

2.5 Convolution Function

This function includes the functions mentioned above. We transfer an image into the input Image structure, the Kernel structure, and the output Image structure to the function. Inside the function, we pad the input data, operate the convolution operation, and package all the information to the output Image structure. Considering the different size of the kernel and the different size of the input data, we choose the relevantly efficient function to operate the convolution operation.

```
bool Convolution(const Image* input_image, const Kernel* kernel, Image* output_image)
{
    //Safe Detection
    if(input_image->channels==1)
    {
        //padding
        float* pad_data = padding(input_image->data1,input_image->height,
        input_image->width,kernel->height,kernel->width);

        //cnn
        float* data1 = (float*)malloc(sizeof(float)*input_image
        ->height*input_image->width);
        if(kernel->height==1&&kernel->width==1)
        {
            cnn_avx2_kernel1x1(pad_data,input_image->height,input_image->width,kernel->data1,kernel->height,
            kernel->width,data1);
            output_image->data1 = data1;
        }
        else if(kernel->height==3&&kernel->width==3)
        {
            if(input_image->width%8==0)
                cnn_avx2_omp_kernel3x3(pad_data,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
            else
                cnn_avx2_kernel3x3(pad_data,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
            output_image->data1 = data1;
        }
        else if(kernel->height==5&&kernel->width==5)
        {
            if(input_image->width%8==0)
                cnn_avx2_omp_kernel5x5(pad_data,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
            else
                cnn_avx2_kernel5x5(pad_data,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
            output_image->data1 = data1;
        }
        output_image->data2 = NULL;
        output_image->data3 = NULL;
        free(pad_data);
    }
    else if(input_image->channels==3)
    {
        //padding
        float* pad_data1 = padding(input_image->data1,input_image->height,input_image->width,
        kernel->height,kernel->width);
        float* pad_data2 = padding(input_image->data2,input_image->height,input_image->width,
        kernel->height,kernel->width);
        float* pad_data3 = padding(input_image->data3,input_image->height,input_image->width,
        kernel->height,kernel->width);

        //cnn
        float* data1 = (float*)malloc(sizeof(float)*input_image->height*input_image->width);
        float* data2 = (float*)malloc(sizeof(float)*input_image->height*input_image->width);
        float* data3 = (float*)malloc(sizeof(float)*input_image->height*input_image->width);
        if(kernel->height==1&&kernel->width==1)
        {
            cnn_avx2_kernel1x1(pad_data1,input_image->height,input_image->width,kernel->data1,
            kernel->height,kernel->width,data1);
            cnn_avx2_kernel1x1(pad_data2,input_image->height,input_image->width,kernel->data1,
            kernel->height,kernel->width,data2);
            cnn_avx2_kernel1x1(pad_data3,input_image->height,input_image->width,kernel->data1,
            kernel->height,kernel->width,data3);
            output_image->data1 = data1;
            output_image->data2 = data2;
            output_image->data3 = data3;
        }
        else if(kernel->height==3&&kernel->width==3)
        {
            if(input_image->width%8==0)
            {
                cnn_avx2_omp_kernel3x3(pad_data1,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
                cnn_avx2_omp_kernel3x3(pad_data2,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data2);
                cnn_avx2_omp_kernel3x3(pad_data3,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data3);
            }
            else
            {
                cnn_avx2_kernel3x3(pad_data1,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
                cnn_avx2_kernel3x3(pad_data2,input_image->height,input_image->width,kernel->data1,
```

```
                kernel->height,kernel->width,data2);
                cnn_avx2_kernel3x3(pad_data3,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data3);
            }
            output_image->data1 = data1;
            output_image->data2 = data2;
            output_image->data3 = data3;
        }
        else if(kernel->height==5&&kernel->width==5)
        {
            if(input_image->width%8==0)
            {
                cnn_avx2_omp_kernel5x5(pad_data1,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data1);
                cnn_avx2_omp_kernel5x5(pad_data2,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data2);
                cnn_avx2_omp_kernel5x5(pad_data3,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data3);
            }
            else
            {
                cnn_avx2_kernel5x5(pad_data1,input_image->height,input_image->width,kernel->data1
                ,kernel->height,kernel->width,data1);
                cnn_avx2_kernel5x5(pad_data2,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data2);
                cnn_avx2_kernel5x5(pad_data3,input_image->height,input_image->width,kernel->data1,
                kernel->height,kernel->width,data3);
            }
            output_image->data1 = data1;
            output_image->data2 = data2;
            output_image->data3 = data3;
        }
        else
        {
            printf("The size of the kernel must be 1x1, 3x3 or 5x5");
        }
        free(pad_data1);
        free(pad_data2);
        free(pad_data3);
        output_image->width = input_image->width;
        output_image->height = input_image->height;
        output_image->channels = input_image->channels;
        return true;
    }
}
```

2.6 Auxiliary Functions

2.6.1 Get Random Data

We obtain random numbers by calculating formulas and obtaining the seeds of random numbers through the system clock. And assign the first place of these numbers to a pointer. (In this project the range of the random float number is from 0 to 255)

```
float* Random(const size_t height, const size_t width)
{
    float* data = (float*)malloc(sizeof(float)*height*width);
    if(data==NULL)
    {
        printf("Malloc Error");
        return NULL;
    }
    srand((unsigned)time(NULL));
    for(int i=0;i<height*width;i++)
        *(data + i) = ((rand() / (float)RAND_MAX) + (rand()% 254 ));
    return data;
}
```

2.6.2 Print the Data

When the amount of data is small, we can print out the data in the form of a matrix for us to check if the result of the operation is correct.

```
bool printinfo(const float* data, const size_t height, const size_t width, const char* name)
{
```

```
    if(data==NULL)
    {
        printf("NULL Pointer!");
        return FALSE;
    }
    printf("%s:\n",name);
    for(int j=0; j<height;j++)
    {
        for(int i=0; i<width; i++)
            printf("%.2f ",*(data + j*width + i));
        printf("\n");
    }
    printf("\n");
    return true;
}
```

2.6.3 Calculate the Time Cost

When dealing with the same data, we use the length of the time cost through this operation to test the efficiency of each function.

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. time

```
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);
```

2.6.4 Write data into a CSV file

When conducting a large number of experiments which collect a large amount of data, we can write our data to a CSV file, which is more convenient for us to process the data.

If we do not have a file, we can create a csv named 'TimeCost.csv' through fopen() using the 'w' mode to write some basic information. (Just an Example)

```
FILE *fp;
    fp = fopen("TimeCost.csv", "w");
    if(fp == NULL) {
        printf("Failed to open file\n");
        return 1;
    }
    fprintf(fp,"%s\n","TimeCost");
    ......
```

If we already have a file, we can use mode 'a' to append new data after the data in that file.

```
FILE *fp;
    fp = fopen("TimeCost.csv", "a");
    if(fp == NULL) {
        printf("Failed to open file\n");
        return 1;
    }
    fprintf(fp,"%f,%f\n",Time1, Time2);
    ......
```

CSV file using ',' to separate the data into blocks in a row and using '\n' to change a row.

2.6.5 Package the Testing

Since there are many experiments to be performed, I packaged the experiments into a function test(). The function takes as arguments the function to be tested (using the function pointer), the name of the function to be tested, the data, the height, the width of the image, and the kernel.

This function contains the process of padding, convolution operations, calculating the time costs during the operation, and outputting the result. (printing out the data or writing the data into a CSV file)

```
bool test(bool (*cnn)(const float* input_data, const size_t input_height, const size_t input_width,
const float* kernel_data, const size_t kernel_height,
    const size_t kernel_width, float* output_data),const float* input_data, const size_t input_height,
    const size_t input_width, const float* kernel_data,
    const size_t kernel_height,const size_t kernel_width,const char* name)
{

    bool flag = false;
    //Padding
    float*pad_data = padding(input_data,input_height,input_width,kernel_height,kernel_width);
    //cnn
    float* cnn_data = (float*)calloc(input_height*input_width,sizeof(float));
    if(cnn_data==NULL)
    {
        printf("cnn_data calloc error!");
        return false;
    }
    struct timeval start, end;
    flag = gettimeofday(&start, NULL);
    if (!flag)
    {
        printf("Error: gettimeofday()\n");
        return false;
    }
    flag = cnn(pad_data, input_height, input_width, kernel_data, kernel_height, kernel_width, cnn_data);
    if(!flag)
    {
        printf("cnn Error!");
        return false;
    }
    flag = gettimeofday(&end, NULL);
    if (!flag)
    {
        printf("Error: gettimeofday()\n");
        return false;
    }
    //fprintf(fp,"%lf,",(end.tv_sec - start.tv_sec) +
    (double)(end.tv_usec - start.tv_usec) / 1000000.0);
    printf("%s\n",name);
    printf("Time Cost : %lf s\n",(end.tv_sec - start.tv_sec) + (double)(end.tv_usec - start.tv_usec)/1000000.0);
    printf("\n);
    free(cnn_data);
    free(pad_data);
}
```

2.6.6 Free the memory

The data of an image is usually very large. So we store the data in dynamic memories (The memory in a heap). Simply using free() to free the structure will not free the memory pointed to by the pointer inside the Image structure. Thus, there is a need for us to design a function to free the memory pointed to by the pointer inside the Image structure. By the way, the data of the kernel is usually not too big. Thus, we store the data of the kernel in static memories (The memory in the stack). There is no need to free the static memory manually, it will be free automatically when the program is finished.

```
bool Free_Image(Image* image)
{
    if(image==NULL)
    {
        printf("NULL pointer!");
        return false;
    }
    if(image->data1!=NULL)
    {
        free(image->data1);
        image->data1=NULL;
    }
    if(image->data2!=NULL)
    {
        free(image->data2);
        image->data2=NULL;
    }
    if(image->data3!=NULL)
    {
        free(image->data3);
        image->data3=NULL;
    }
```

```
    return true;
}
```

2.7 Compile the file

This project includes a library called libjpeg and need to implement SIMD instructions and OpenMP instructions. Thus, there are servals things needed to link when compiling the code.

```
gcc project3demo.c Func.c -ljpeg -DWITH_AVX2 -mavx
```

3 Experiments & Analysis

3.1 Efficiency Analysis

3.1.1 The Dimension of Loops

In this experiment, we want to figure out the effect of the number of layers of the loop on the convolution speed.

Four functions are designed. For the convolution of an element, the row-index = (k / kernel-width) and the column-index = (k % kernel-width). For the convolution of the whole image, the row-index = (t / input-width) and the column-index = (t % kernel-width). In this way, we can lower the dimension of the loop.

```
bool cnnFunction_v1()
{
    for(int j = 0; j < input_height; j++)
    {
        for(int i = 0; i < input_width;i++)
        {
            for(int m = 0; m<kernel_height;m++)
            {
                for(int n = 0,k = 0; n < kernel_width; n++,k++)
                {
                    *(output_data+(j*input_width)+i) +=
                    (*(input_data + (j * pad_width) + i + (m*pad_width) + n))
                    * (*(kernel_data+k));
                }
            }
        }
    }
}
bool cnnFunction_v2()
{
    for(int j = 0; j < input_height; j++)
    {
        for(int i = 0; i < input_width;i++)
        {
            for(int k =0; k<kernel_height * kernel_width;k++)
            {
                *(output_data+(j * input_width)+i) +=
                (*(input_data + (j * pad_width) + i + ((k / kernel_width)*pad_width) + (k % kernel_width)))
                * (*(kernel_data+k));
            }
        }
    }
}
bool cnnFunction_v3()
{
    for(int t = 0; t < input_height*input_width; t++)
    {
        int j = t / input_width;
        int i = t % input_width;
        for(int m = 0; m<kernel_height;m++)
        {
            for(int n = 0,k = 0; n < kernel_width; n++,k++)
            {
                *(output_data+(j*input_width)+i) +=
                (*(input_data + (j * pad_width) + i + (m*pad_width) + n))
                * (*(kernel_data+k));
            }
        }
    }
```

```
}
bool cnnFunction_v4()
{
    for(int t = 0; t < input_height*input_width; t++)
    {
        int j = t / input_width;
        int i = t % input_width;
        for(int k =0; k<kernel_height * kernel_width;k++)
        {
            *(output_data+(j * input_width)+i) +=
            (*(input_data + (j * pad_width) + i + ((k / kernel_width)*pad_width) + (k % kernel_width)))
            * (*(kernel_data+k));
        }
    }
}
```

In the experiment, we first use random() to create a certain size of random float image_data and a random 3x3 kernel_data. Then, we use test() to implement the convolution operation, calculate the time duration and store the time interval in the CSV file. The steps mentioned above will be carried out times for a certain size of image data size.

```
FILE *fp;
//open the file
fp = fopen("TimeCost.csv", "a");
for(int num = 1; num<=10; num++)
{
    //create random number
    float* image_data = (float*) malloc(128*128*sizeof(float));
    image_data = Random(128,128);
    float* kernel_data = (float*)malloc(3*3*sizeof(float));
    kernel_data = Random(3,3);
    fprintf(fp,"%d,",num);
    test(cnnFunction_v1,image_data,128,128,kernel_data,3,3,fp,"cnn_v1");
    test(cnnFunction_v2,image_data,128,128,kernel_data,3,3,fp,"cnn_v2");
    test(cnnFunction_v3,image_data,128,128,kernel_data,3,3,fp,"cnn_v3");
    test(cnnFunction_v4,image_data,128,128,kernel_data,3,3,fp,"cnn_v4");
    fprintf(fp,"\n");
    //free the memory
}
//close the file
```

The results are as follows.

对于不同的数据量，不同循环层数的卷积所用时间比较						
Function	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
cnn_v1(s)	0.0005945	0.0024136	0.0095929	0.0380648	0.1511122	0.6040479
cnn_v2(s)	0.0006313	0.0026448	0.0104002	0.0415641	0.1657193	0.6681645
cnn_v3(s)	0.0006612	0.0026523	0.0107056	0.042967	0.1713406	0.6866863
cnn_v4(s)	0.0006731	0.002731	0.011079	0.0443474	0.1773043	0.7116192

The results of the experiment are different from what we just envisioned at the beginning. The reason for the results is that the reduction in consumption through cycle reduction has no way to compensate for the increase in computation time due to the cycle reduction.

3.1.2 Unloop

In this experiment, we want to figure out the effect of the number of loops on the convolution speed.

Two Functions are designed. The first one is the same as the cnnFunction_v1 mentioned above. The second one calculates 8 elements in one loop. By the way, in order to deal with all the elements in the case that the input width is not a multiple of 8, another block of loops is set to deal with the remaining item(s).

```
bool cnnFunction_v1();
bool cnn_unloop();
```

The way of conducting the experiment is the same as the former experiment.

The results are as follows.

对于8的倍数的不同的数据量，解循环与不解循环卷积的时间比较						
Function	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
cnn_v1(s)	0.0006978	0.0024667	0.0095909	0.0379422	0.152315	0.6049463
Unloop(s)	0.0005326	0.0019161	0.0075812	0.0302822	0.1206239	0.4829237
速度提升率	0.236744053	0.2232132	0.209542379	0.201886027	0.208062896	0.201708152

When the size of the data is a multiple of 8, the speed improvement rate of the unloop function is approximately 21%.

对于8的倍数加7的不同的数据量，解循环与不解循环卷积的时间比较						
Function	15x15	73x73	135x135	263x263	519x519	1031x1031
cnn_v1(s)	0.0000102	0.0002474	0.0007386	0.0025631	0.0098216	0.0385328
Unloop(s)	0.0000083	0.0002042	0.000591	0.0020271	0.007841	0.0307507
速度提升率	0.18627451	0.174616006	0.19983753	0.209121767	0.201657571	0.201960408

When the size of the data is not a multiple of 8, the speed improvement rate of the unloop function is relatively low, especially when the size of the data is small. (approximately 18%) However, as the size of the data increases, the speed improvement gradually approaches 21%. (The average speed improvement rate above)

The reasons for the results are that when we reduce the number of loops, we reduce the cost of calling the loop (The cost of repeated calculations and comparisons). Thus, the speed improves. When the size of the data is not a multiple of 8, it costs time to finish the convolution operation in the normal way and some time to check.

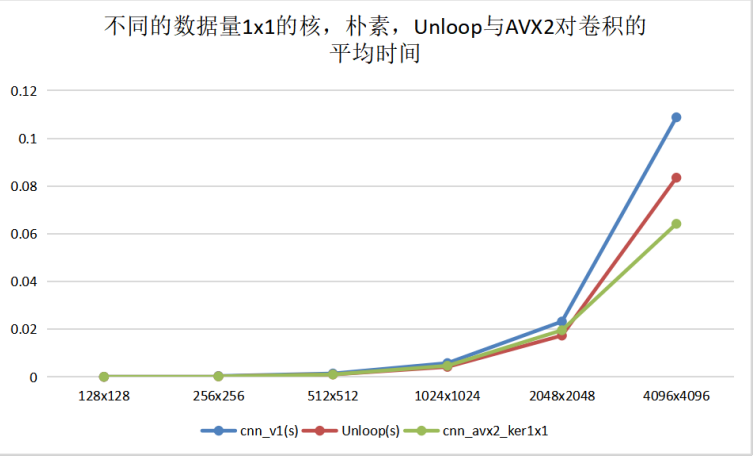
3.1.3 SIMD Analysis

This experiment aims to compare the efficiency between the Unloop function and the SIMD function in the case of 1x1, 3x3, and 5x5 kernels.

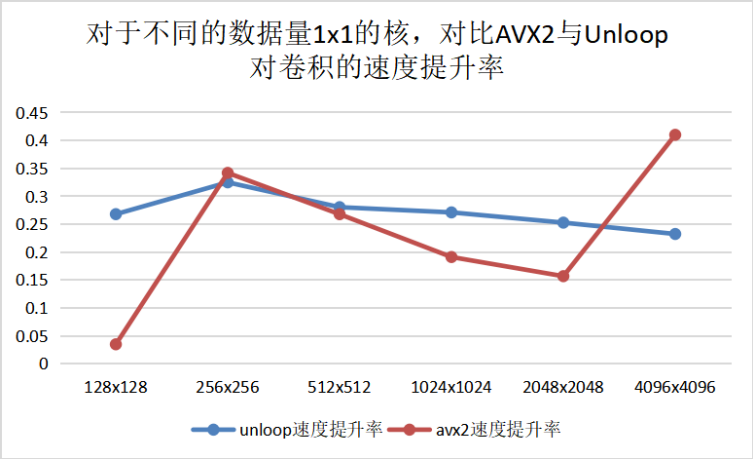
In the case of a 1x1 kernel, we compare the time cost of the following 3 functions.

```
bool cnnFunction_v1();
bool cnn_unloop();
bool cnn_avx2_kernel1x1();
```

The results are as follows.



Through the figure above, we can find that, as the size of the data increases, the differences between the time cost of the three functions become apparent.

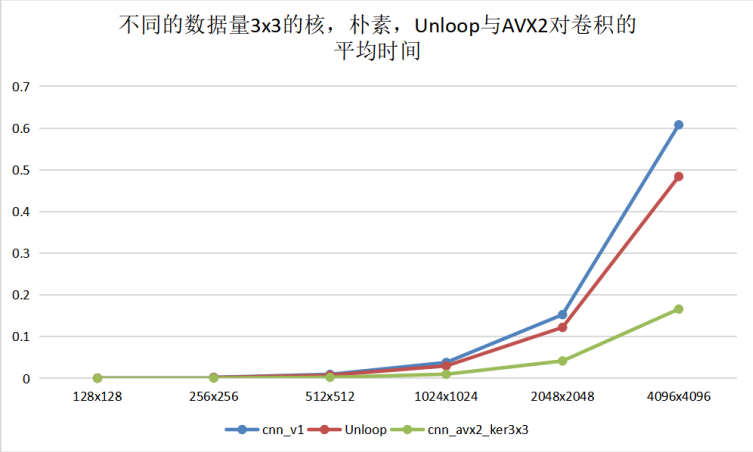


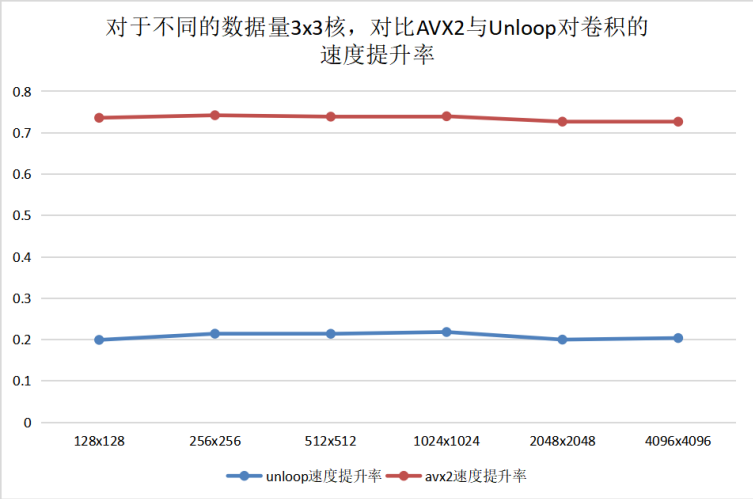
Through the figure above, we can find that, in the case of a 1x1 kernel, when the size of the data is small, the speed improvement of Unloop function is bigger than that of SIMD. However, when the size of the data is big, the speed improvement of the SIMD function becomes bigger than those of Unloop function.

In the case of a 3x3 kernel, we compare the time cost of the following 3 functions.

```
bool cnnFunction_v1();
bool cnn_unloop();
bool cnn_avx2_kernel3x3();
```

The results are as follows.





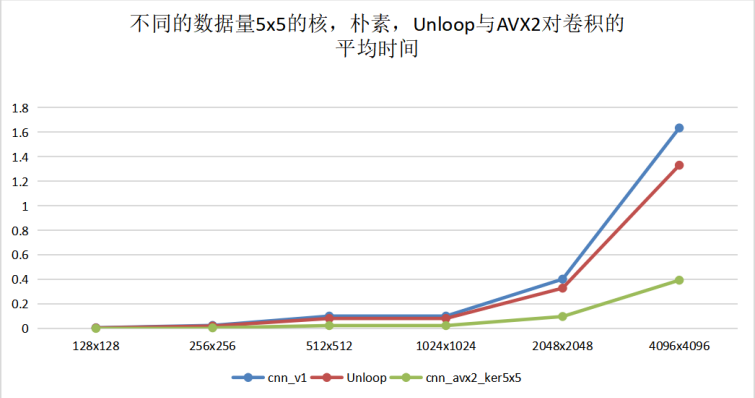
Through the figures above, we can find that, in the case of a 3x3 kernel, the speed improvement rate of the SIMD function is constantly bigger than that of the Unloop function. And the speed improvement rate of

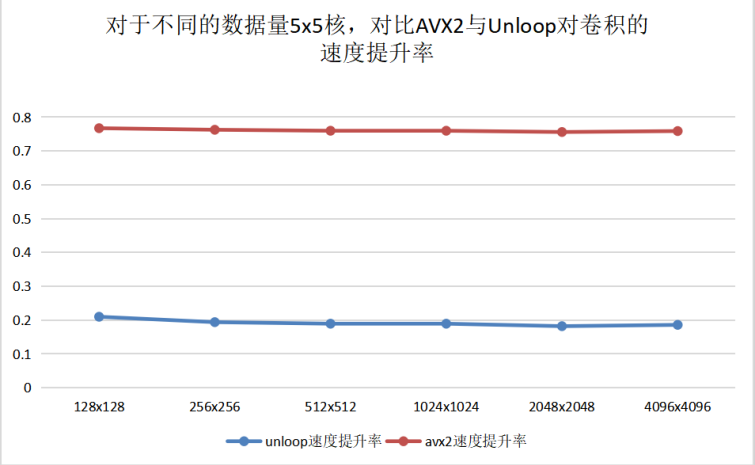
both SIMD and Unloop remain relatively constant among the different size of the data. (about 73% and 20% separately)

In the case of a 5x5 kernel, we compare the time cost of the following 3 functions.

```
bool cnnFunction_v1();
bool cnn_unloop();
bool cnn_avx2_kernel5x5();
```

The results are as follows.





Through the figures above, we can find that the result in the case of a 5x5 kernel is similar to the result in the case of a 3x3 kernel. (about 76% and 20% separately)

SIMD is a single instruction that can operate multiple data elements at the same time, making full use of register space. It saves processor time and resources by reducing the number of data reads and thus the number of memory accesses. However, implementing a SIMD has some costs in the preparation.

When the size of the kernel is small, the speed improvement due to the SIMD will not compensate for the costs of the preparation, resulting in the speed improvement rate of SIMD being smaller than Unloop at a small data size. When the size of the data gets bigger, the advantage of SIMD will be apparent.

When the size of the kernel is big, it is apparent that SIMD has a great advantage in efficiency compared to Unloop.

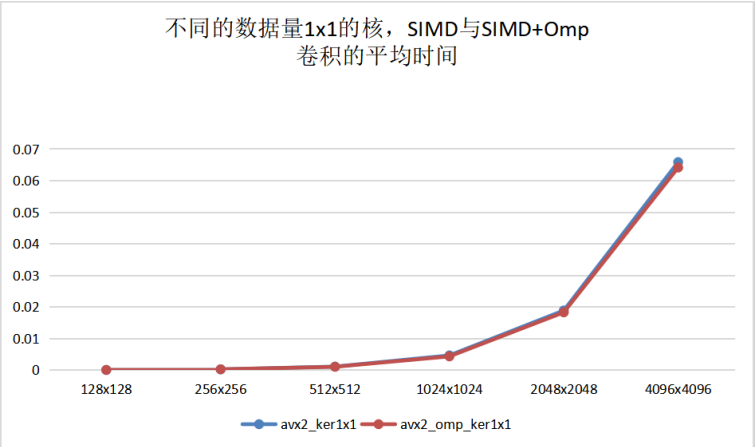
3.1.4 OpenMP Analysis

This experiment aims to compare the efficiency between SIMD and SIMD with OpenMP in the case of 1x1, 3x3 and 5x5 kernels.

In the case of a 1x1 kernel, we compare the time cost of the following 2 functions.

```
bool cnn_avx2_kernel1x1();
bool cnn_avx2_omp_kernel1x1();
```

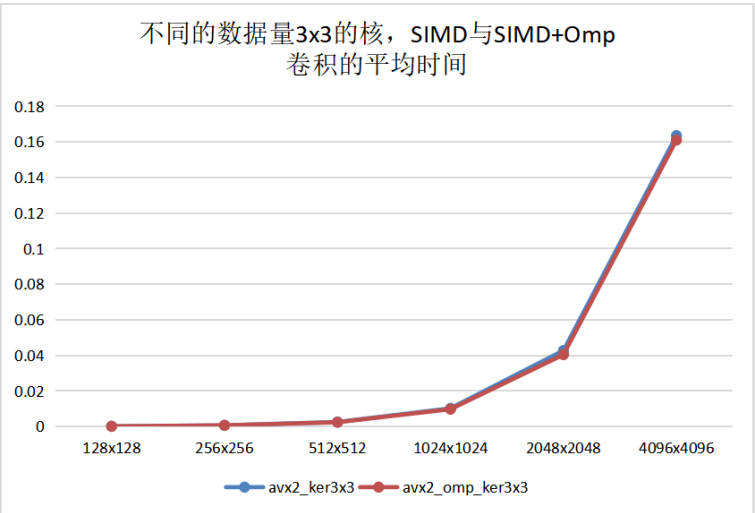
The results are as follows.



In the case of a 3x3 kernel, we compare the time cost of the following 2 functions.

```
bool cnn_avx2_kernel3x3();
bool cnn_avx2_omp_kernel3x3();
```

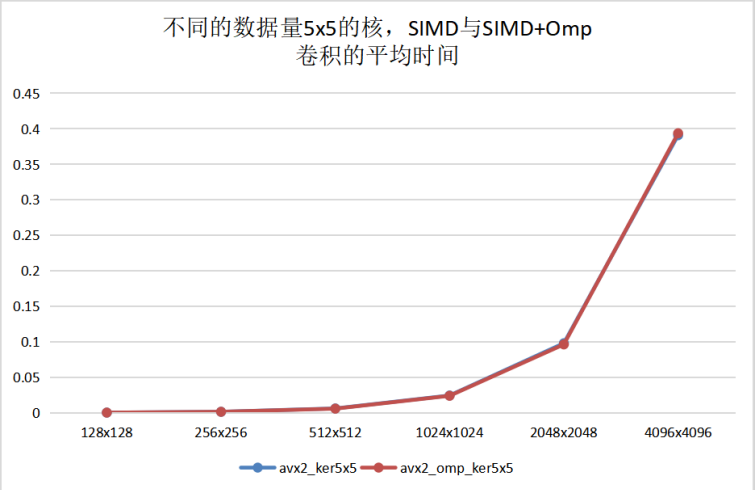
The results are as follows.



In the case of a 5x5 kernel, we compare the time cost of the following 2 functions.

```
bool cnn_avx2_kernel5x5();
bool cnn_avx2_omp_kernel5x5();
```

The results are as follows.



Through the figures listed above, we find that no matter the size of the data or the size of the kernel, combining OpenMP with SIMD will not bring extra efficiency compared to the single SIMD operation.

The reason for the result is that when performing convolution operations via SIMD, it is necessary to chunk the input data, which already adds additional copy and reorder overhead. In addition, in OpenMP, the input data needs to be copied again to avoid competition between threads, which may take up a lot of memory space and thus affect the efficiency of the program.

3.1.5 -O3

We compare the speed of the convolution operation with and without the -O3 instruction.

The results are as follows

O3的优化				
	unloop	omp_unloop	SIMD	SIMD_omp
加-O3	0.0596392	0.0600104	0.0191133	0.0166557
不加-O3	0.4940981	0.4931542	0.1663743	0.1611616
速度提升率	0.879296844	0.878313112	0.885118675	0.89665218

3.2 Correctness

3.2.1 Qualitative Method

We output the images after the convolution operation in C and Matlab to see if they are the same.

Convolution layer in C. (The example is an image of one channel)

```c
//Input the Image
Image input_image;
input_jpeg_image("gray.jpg",&input_image);
//Initialize the kernel
Kernel kernel;
float kernel_data[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
flag=kernel_init(&kernel,3,3,input_image.channels,kernel_data,NULL,NULL);
//Create the output image
Image output_image;
//Convolution operation
Convolution(&input_image,&kernel,&output_image);
//output the image
flag = output_jpeg_image("grayoutput.jpg",&output_image);
//free the memory
Free_Image(&input_image);
```

Convolution layer in MATLAB. (The example is an image of one channel)

```
img = imread('gray.jpg');
kernel = [-1 0 1; -2 0 2; -1 0 1];
result = conv2(img, kernel, 'same');
imshow(result);
```

The result in C and the result in MATLAB are as follows.

Figure 1: Output from C

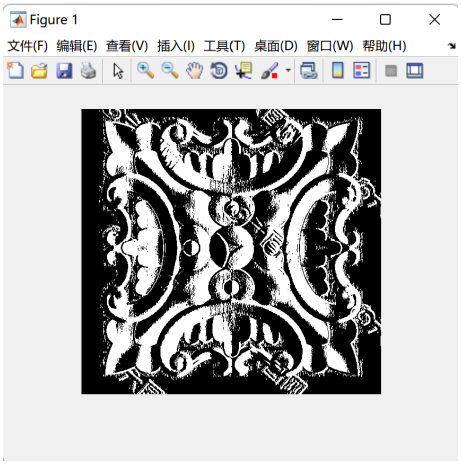


Figure 2: Output from MATLAB

From the result, it is likely that the Convolution layer in C is correct.

4 Difficulties & Solutions

4.1 Difficulties in input and output the image

Reading images in C is difficult as it does not contain any related function. This project uses a library called 'libjpeg' to read the data of a jpg image.

First, we need to download the library in Ubuntu and include the head file when running the program. (The instruction and the head file are as follows.)

```
sudo apt-get install libjpeg-dev
#include <jpeglib.h>
```

Then, we need to learn the grammar of decompressing and compressing the jpg. (What I have learned is mentioned in the 'input and output image' of the Code Implementation part)

4.2 Difficulties in Checking the correctness of the convolution operation

Checking if our convolution operation in C is difficult. Not only does it have no function of convolution operation, but it is also very difficult to find a library linked by C that can operate the convolution operation.

4.2.1 Attempt Intel Math Kernel Library

By searching on the internet, I found that the Math Kernel Library can conduct the convolution operation in C. However, having spent two to three hours, I still failed to install the library in my Ubuntu. Considering the time limit, I quit this method.

4.2.2 Attempt to check the correctness quantitatively

By searching on the internet, I found that we can operate convolution operations in MATLAB and can write the data of the images after the convolution operation in C and Matlab into Txt files, and use VSCode's file-matching feature to check if the results are correct.

Output data in C.

```
FILE *fp;
fp = fopen("C_output.txt", "w");
for(int i=0;i<input_image.height*input_image.width*input_image.channels;i++)
    fprintf(fp,"%f\n",*(input_image.data1+i));
```

Output data in MATLAB.

```
fileID = fopen('M_output.txt', 'w');
fprintf(fileID, '%d\n', result);
fclose(fileID);
```

However, the datatype of storing the image data in C through libjpeg differs from MATLAB. The original datatype of the jpg image reading by libjpeg is unsigned char (I make a type conversion from unsigned char to float in order to satisfy the requirement) and the datatype of the jpg image reading by MATLAB is an integer (which contains the negative numbers). Without the relevant knowledge of the Algorithm to read the image in both libjpeg and MATLAB and without plenty of time to learn, I quit this method.

5 Brief Summary

I gained a lot through this project.

This project focused on the implementation of a convolutional layer through C language and improving the efficiency of the program, which made me more familiar with C language, and made me more profoundly appreciate that C language is closer to the bottom. The perspective of how to improve efficiency is thinking from the bottom, such as the nature of computer s implementation operations, SIMD, and OpenMP operations.

Also, this project gave me a preliminary understanding of CNN, what convolution kernel, convolution, padding and stride are, and also gave me a preliminary understanding of the image processing field.