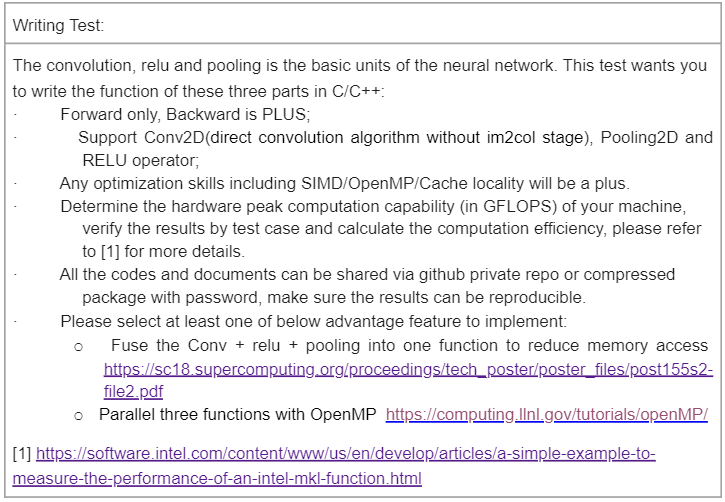
[Intel] Writing test report by Wu Hao:

Writing Test Detals：

I wrote the fuctions of the convolution, relu and pooling in C/C++ for forward inference via **OpenMP[1]** for parallel computing. In addition, **Fused Microkernel [2]** is applied to Convolution relu and pooling for exploiting better computation.

Information of my computer:

CPU: Intel® Core™ i7-8700 CUP@ 3.20GHz

RAM: 8G

Operation System: Microsoft Windows 10 LTSC (64bit)

IDE: Visual Studio 2019

Briefly, I introduce the basic logic of my code:

1. I use dynamic arrays as the basic type to complete 2D matrix reading and writing, as well as convolution relu and pooling operations.

2. The function fused\_conv defined in the code simultaneously completes the convolution relu and pooling operation. It can get the results of three functions in one loop.

3.OpenMP is used in the main loop(conv+relu+pooling).

Details of the code:

I named conv+relu+pooling fused function as fused\_conv in the code. It is defined as:

float\*\* fused\_conv(float\*\* feature,int &row\_i, int &col\_i, float\*\* filter, int size\_k, const unsigned stride, const unsigned pooling\_num, const float bias)

“feature” : the input of convolution layer.

“row\_i” and “col\_i”: the dimensions of rows and columns of “feature”.

“filter”: the convolutional kernel;

“size\_k”: convolutional kernel size;

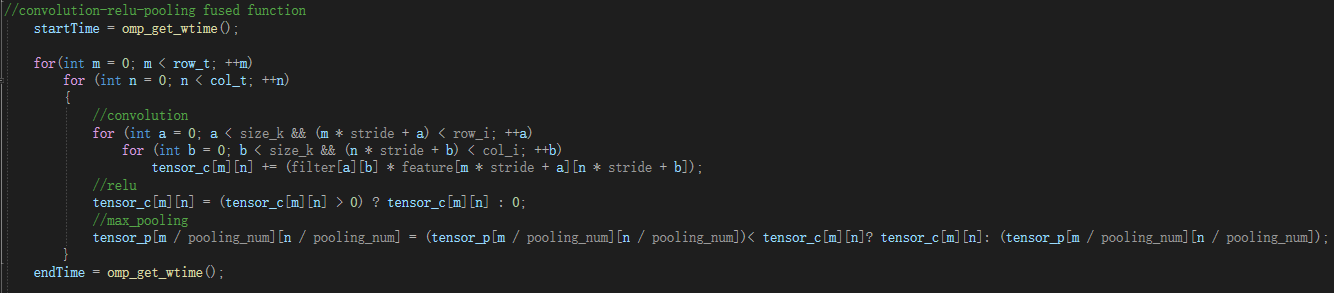
“stride”: convolutional stride;

“pooling\_num”: pooling\_num filter size;

“bias”: bias of convolutional layer.

The return value is a pointer to the 2D array of the conv+relu+pooling result.

Fused function and parallel computing code:



Each time an element (tensor\_c[m][n]) of convolution layer output matrix is obtained, the corresponding element of relu layer output matrix and pooling layer output matrix(tensor\_p) are updated synchronously.

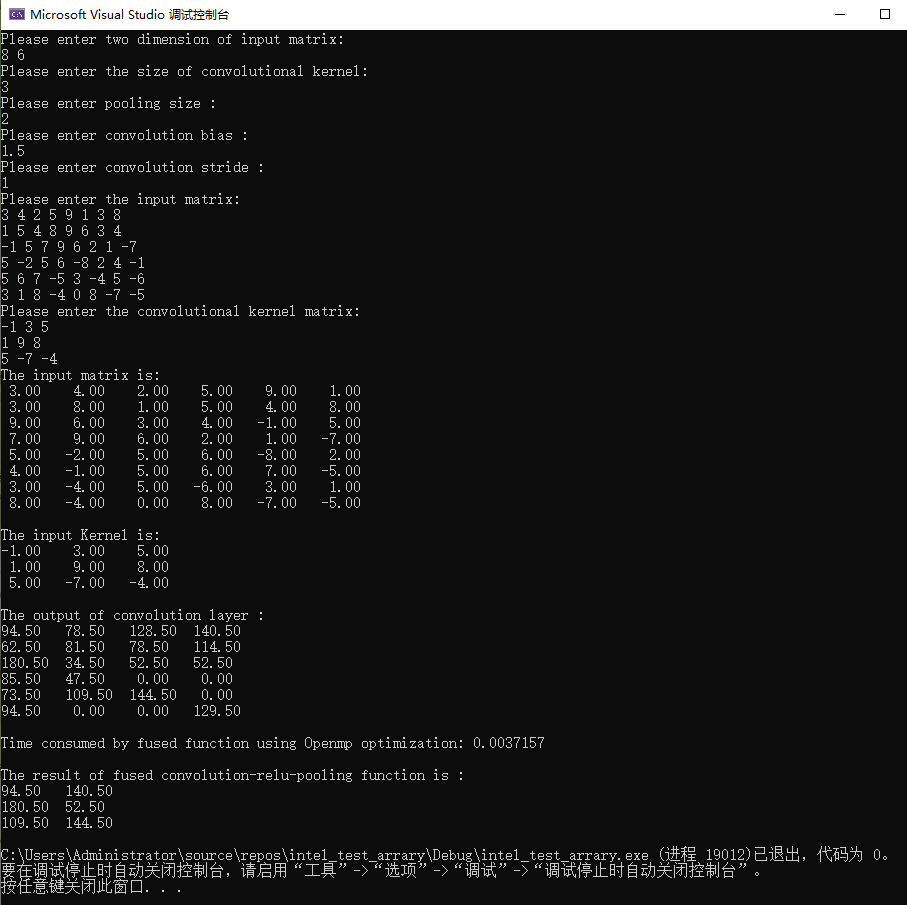
#pragma omp parallel for collapse(2)

The above code uses OpenMP to optimize the loop.

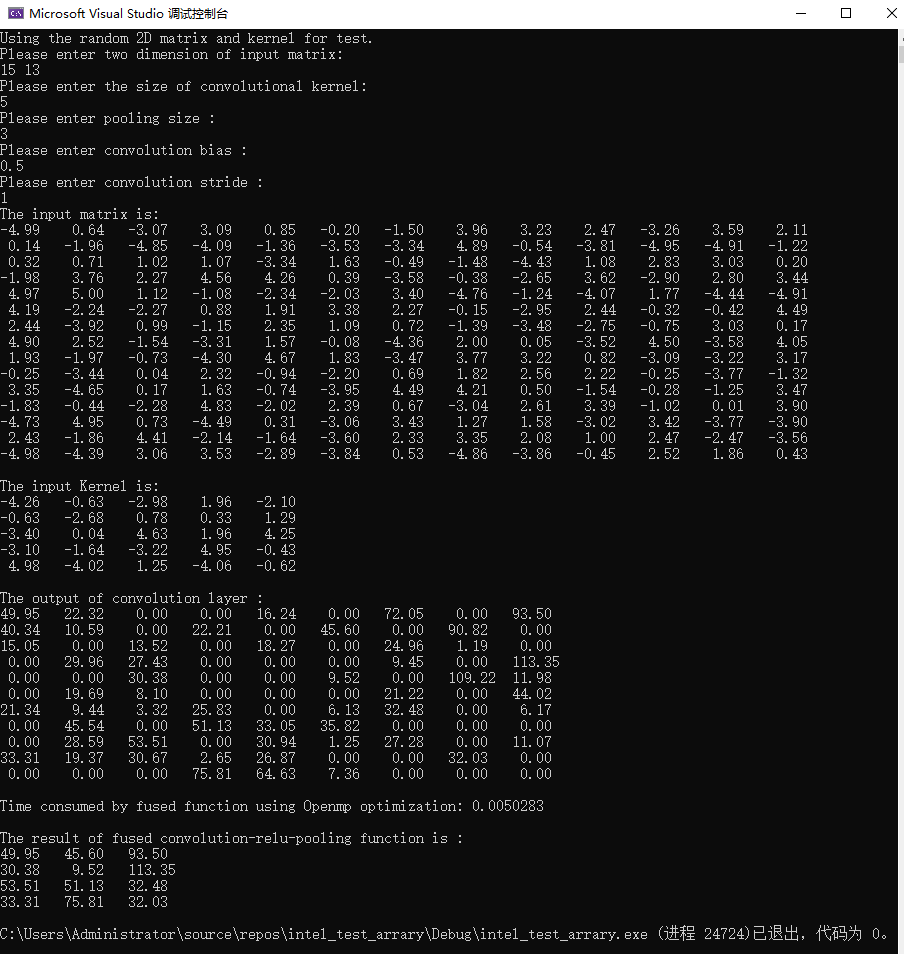
Test results:

The code allows mannual input of matrices and convolutional kernels, or randow generation of matrices and convolution kernels(convenient to test time complexity of large matrices). Here, two methods are used to test. When the convolution or pooling operation is out of bounds, padding 0 method is used to avoid it, so that 7\*7 feature and pooling size 2\*2 will get 4\*4 output.

Test1 : 8\*6 input matrix，3\*3 convolutional kernel, stride 1, bias 1.5, max-pooling size 2. Manual input.



Test2: 15\*13 input martix, 5\*5 convolutional kernel, stride 1, bias 0.5, max-pooling size 3. Random input.

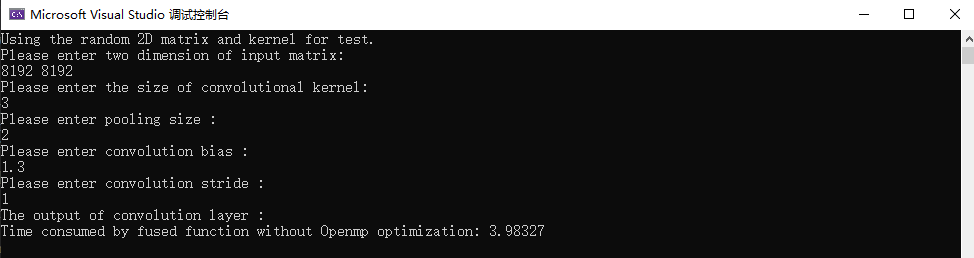


Test 3: Time comparison between parallel(OpenMP) and serial operation.

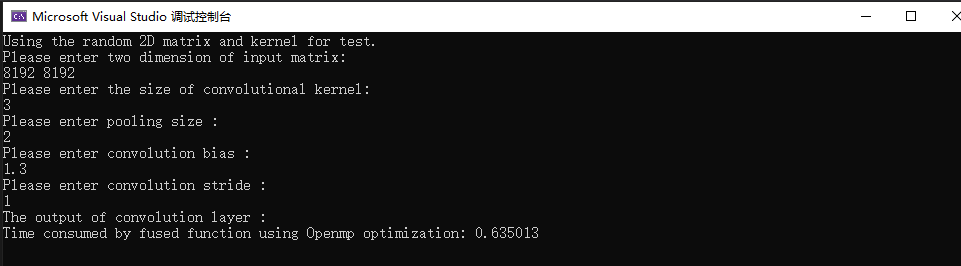
CPU: Intel® Core™ i7-8700 CUP@ 3.20GHz

8192\*8192 input martix, 3\*3 convolutional kernel, stride 1, bias 1.3, max-pooling size 2. Random input.

Time consumed by serial calculation (CPU): 3.98327 s



Time consumed by parallel computing (CPU): 0.635013s



Hardware Peak computation capability (in GFLOPS): 307.2 (theoretical peak performance)



Convolution :

(3\*3 additions + 3\*3multiplications)\* (8190\*8190 elements)/ 0.635s =1.92Gflops

Reference

[1] Board, O. A. R. "OpenMP application program interface version 3.0." *The OpenMP Forum, Tech. Rep*. 2008.

[2] Anderson, Michael, et al. "Tensorfolding: Improving convolutional neural network performance with fused microkernels." *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal.(SC)*. 2018.