```
__kernel void Convolution(__global float* inImage,
                          int kernelWidth,
                          __global float* kernelData,
                          __global float* outImage,
                                                int imgsize,
                                                int nKernels,
                                                int inImagesize,
                                                int outImagesize,
                                                 __global float* bias) {
    int c = get_global_id(0),
        r = get_global_id(1);
    int outputs    = get_global_size(0);
        int imgWidth   = get_global_size(1);
        int inimg, outimg, ker;
        outimg = c*imgWidth + (r/outImagesize) * outImagesize + r%outImagesize;
        outImage[outimg] = 0;
        for(int i = 0; i != nKernels; ++i){
                for (int m = 0; m < kernelWidth; m++) {
                        for (int n = 0; n < kernelWidth; n++) {
                        inimg = i * inImagesize * inImagesize +
(r/outImagesize+m)*inImagesize+r%outImagesize+n;
                        ker = (c * nKernels * kernelWidth * kernelWidth) + (i * kernelWidth *
kernelWidth) + m * kernelWidth + n;
                        outImage[outimg] += inImage[inimg] * kernelData[ker];
                        //printf("%d %d %d %d %d        ",c,r,inimg,ker,outimg);
                         }
```

```
                }
            }
            float ans = outImage[outimg]+bias[c];
            outImage[outimg] = ans ;
            //printf("%d %d %d %d %d₩n",c,r,inimg,ker,outimg);
            //printf("%d,%d outImage[%d]=%f₩n",c,r,outimg,outImage[outimg]);
}
__kernel void Pooling(__global float* inImage,
                      __global float* outImage,
                                  int inImgWidth,
                                  int outImgWidth) {

    int i = get_global_id(0),
    j = get_global_id(1);

        float a, b, c, d, fst_max, snd_max;

        int a1,b1,c1,d1;
        a1=i * inImgWidth * inImgWidth + 2 * (j / outImgWidth) * inImgWidth + 2 * (j%outImgWidth);
        b1=i * inImgWidth * inImgWidth + 2 * (j / outImgWidth) * inImgWidth + 2 * (j%outImgWidth) + 1;
        c1=i * inImgWidth * inImgWidth + (2 * (j / outImgWidth) + 1) * inImgWidth + 2 * (j%outImgWidth);
        d1=i * inImgWidth * inImgWidth + (2 * (j / outImgWidth) + 1) * inImgWidth + 2 * (j%outImgWidth) + 1;
        a = inImage[a1];
```

```
        b = inImage[b1];
        c = inImage[c1];
        d = inImage[d1];
        fst_max = ((a > b) ? a : b);
        snd_max = ((c > d) ? c : d);
        int outimg = i * outImgWidth * outImgWidth + (j / outImgWidth) * outImgWidth  + j%outImgWidth;
        outImage[outimg] = fst_max > snd_max ? fst_max : snd_max;
        //printf("%d %d %d %d %d %d₩n",i,j,a1,b1,c1,d1);

}
__kernel void Fconnect(__global float* inImage,
                                __global float* outImage,
                                                __global float* kernelData,
                                                __global float* bias,
                                               int inImgWidth) {

        int i = get_global_id(0);
        int size = get_global_size(0);
        outImage[i] = 0;
        float ans = 0;
        for(int a = 0; a != inImgWidth; ++a){
                outImage[i] += inImage[a] * kernelData[ i * inImgWidth + a];
                //printf("%d %f ",i,outImage[i]);
        }
```

```
        ans = outImage[i] + bias[i];
        outImage[i] = (size==10)? ans : (ans>0? ans : 0);
        //outImage[i] = ans;

}

__kernel void Pad(__global float* inImage,
                  __global float* outImage,
                                  int inImagesize,
                                  int outImagesize,
                                  int pad) {

        int c = get_global_id(0),
        r = get_global_id(1);
        outImage[( c+ pad) * outImagesize * outImagesize + r + pad]=inImage[c * inImagesize * inImagesize + r];

        }
```

```
#include "Layer.h"

#include <fstream>
#include <iostream>
#include <sstream>
using namespace std;
#define CL_DEBUG
//#define TEST
cl_platform_id     cpPlatform;
cl_device_id       cdDevice;
cl_context         cxGPUContext;
cl_command_queue cqCommandQueue;
cl_program          program;

cl_kernel kernel_conv, kernel_pool, kernel_FC;

void openclRetTackle(cl_int retValue, char* processInfo)
{
        char* errInfo = nullptr;
        switch (retValue)
        {
        case   0:errInfo = "CL_SUCCESS"; break;
        case -1:errInfo = "CL_DEVICE_NOT_FOUND"; break;
        case -2:errInfo = "CL_DEVICE_NOT_AVAILABLE"; break;
        case -3:errInfo = "CL_COMPILER_NOT_AVAILABLE"; break;
        case -4:errInfo = "CL_MEM_OBJECT_ALLOCATION_FAILURE"; break;
```

```
case -5:errInfo = "CL_OUT_OF_RESOURCES"; break;

case -6:errInfo = "CL_OUT_OF_HOST_MEMORY"; break;

case -7:errInfo = "CL_PROFILING_INFO_NOT_AVAILABLE"; break;

case -8:errInfo = "CL_MEM_COPY_OVERLAP"; break;

case -9:errInfo = "CL_IMAGE_FORMAT_MISMATCH"; break;

case -10:errInfo = "CL_IMAGE_FORMAT_NOT_SUPPORTED"; break;

case -11:errInfo = "CL_BUILD_PROGRAM_FAILURE"; break;

case -12:errInfo = "CL_MAP_FAILURE"; break;

case -13:errInfo = "CL_MISALIGNED_SUB_BUFFER_OFFSET"; break;

case -14:errInfo = "CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST"; break;

case -30:errInfo = "CL_INVALID_VALUE"; break;

case -31:errInfo = "CL_INVALID_DEVICE_TYPE"; break;

case -32:errInfo = "CL_INVALID_PLATFORM"; break;

case -33:errInfo = "CL_INVALID_DEVICE"; break;

case -34:errInfo = "CL_INVALID_CONTEXT"; break;

case -35:errInfo = "CL_INVALID_QUEUE_PROPERTIES"; break;

case -36:errInfo = "CL_INVALID_COMMAND_QUEUE"; break;

case -37:errInfo = "CL_INVALID_HOST_PTR"; break;

case -38:errInfo = "CL_INVALID_MEM_OBJECT"; break;

case -39:errInfo = "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR"; break;

case -40:errInfo = "CL_INVALID_IMAGE_SIZE"; break;

case -41:errInfo = "CL_INVALID_SAMPLER"; break;

case -42:errInfo = "CL_INVALID_BINARY"; break;

case -43:errInfo = "CL_INVALID_BUILD_OPTIONS"; break;

case -44:errInfo = "CL_INVALID_PROGRAM"; break;

case -45:errInfo = "CL_INVALID_PROGRAM_EXECUTABLE"; break;
```

```
            case -46:errInfo = "CL_INVALID_KERNEL_NAME"; break;
            case -47:errInfo = "CL_INVALID_KERNEL_DEFINITION"; break;
            case -48:errInfo = "CL_INVALID_KERNEL"; break;
            case -49:errInfo = "CL_INVALID_ARG_INDEX"; break;
            case -50:errInfo = "CL_INVALID_ARG_VALUE"; break;
            case -51:errInfo = "CL_INVALID_ARG_SIZE"; break;
            case -52:errInfo = "CL_INVALID_KERNEL_ARGS"; break;
            case -53:errInfo = "CL_INVALID_WORK_DIMENSION"; break;
            case -54:errInfo = "CL_INVALID_WORK_GROUP_SIZE"; break;
            case -55:errInfo = "CL_INVALID_WORK_ITEM_SIZE"; break;
            case -56:errInfo = "CL_INVALID_GLOBAL_OFFSET"; break;
            case -57:errInfo = "CL_INVALID_EVENT_WAIT_LIST"; break;
            case -58:errInfo = "CL_INVALID_EVENT"; break;
            case -59:errInfo = "CL_INVALID_OPERATION"; break;
            case -60:errInfo = "CL_INVALID_GL_OBJECT"; break;
            case -61:errInfo = "CL_INVALID_BUFFER_SIZE"; break;
            case -62:errInfo = "CL_INVALID_MIP_LEVEL"; break;
            case -63:errInfo = "CL_INVALID_GLOBAL_WORK_SIZE"; break;
            case -64:errInfo = "CL_INVALID_PROPERTY"; break;
            }
            if (retValue != CL_SUCCESS)
            {
#if (defined CL_DEBUG) || (defined CL_VERBOSE)
                        fprintf(stderr, "%s Error! %s ₩n", processInfo, errInfo);
#endif
                        system("pause");
```

```
                exit(-1);
        }
        else
        {
#ifdef CL_VERBOSE
                printf("%s Success!₩n", processInfo);
#endif
        }
}

void printCLDeviceInfo()
{
        cl_uint uintRet;
        cl_ulong ulongRet;
        clGetDeviceInfo(cdDevice, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(uintRet), &uintRet,
NULL);
        printf("CL_DEVICE_MAX_COMPUTE_UNITS %d₩n", uintRet);
       clGetDeviceInfo(cdDevice, CL_DEVICE_MAX_MEM_ALLOC_SIZE, sizeof(ulongRet), &ulongRet,
NULL);
        printf("CL_DEVICE_MAX_MEM_ALLOC_SIZE %d₩n", ulongRet);
}
int openclInit()
{
        cl_int ret;
        //µÃµ½Æ½Ì¨ID
        openclRetTackle(clGetPlatformIDs(1, &cpPlatform, NULL), "clGetPlatFormIDs");
```

```
	//µÃµ½GPUÉè±¸ID
	openclRetTackle(clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_CPU, 1, &cdDevice, NULL), "clGetDeviceIDs");
	//»ñÈ¡GPUÉè±¸ÉÏÎÂÎÄ
	cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ret);

#if (defined CL_DEBUG) || (defined CL_VERBOSE)
	printCLDeviceInfo();
#endif

	openclRetTackle(ret, "clCreateContext");
	//¿ª±ÙÈÎñ¶ÓÁÐ
	cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, CL_QUEUE_PROFILING_ENABLE, &ret);
	openclRetTackle(ret, "clCreateCommandQueue");

	return CL_SUCCESS;
}
int openclCreateKernelFromFile(cl_program* cpProgram, cl_kernel* clKernel, const char* clFileName, const char* kernelName, int flag)
{
	cl_int ret;
	cl_uint count = 1;
	size_t sourceLength = 0;
	char *sourceString = NULL;
	FILE *fp;
```

```
	fp = fopen(clFileName, "rb");
	if (fp == NULL) {
		fprintf(stderr, "Read source file %s failed, does it exist?₩n", clFileName);
		exit(-1);
	}
	fseek(fp, 0, SEEK_END);
	sourceLength = ftell(fp);
	fseek(fp, 0, SEEK_SET);
	sourceString = (char*)malloc((sourceLength + 1) * sizeof(char));//no non-ascii characters in .cl file please
	memset(sourceString, 0, (sourceLength + 1) * sizeof(char));
	if (fread(sourceString, sourceLength, 1, fp) != 1) {
		fprintf(stderr, "Cannot read source file %s.₩n", clFileName);
		exit(-1);
	}

	if (flag == 0)
	{
		(*cpProgram) = clCreateProgramWithSource(cxGPUContext, count, (const char**)(&sourceString), &sourceLength, &ret);
		openclRetTackle(ret, "clCreateProgramWithSource");
		ret = clBuildProgram(*cpProgram, 1, &cdDevice, NULL, NULL, NULL);
		if(ret!=0){
		size_t log_size;
		char *program_log;
		/* Find size of log and print to std output */
```

```
            clGetProgramBuildInfo(*cpProgram, cdDevice, CL_PROGRAM_BUILD_LOG,
                    0, NULL, &log_size);
            program_log = (char*)malloc(log_size + 1);
            program_log[log_size] = '₩0';
            clGetProgramBuildInfo(*cpProgram, cdDevice, CL_PROGRAM_BUILD_LOG,
                    log_size + 1, program_log, NULL);
            printf("%s₩n", program_log);
            free(program_log);
            }
            openclRetTackle(ret, "clBuildProgram");
        }
        else { int a = 0; }
        (*clKernel) = clCreateKernel(*cpProgram, kernelName, &ret);

        char *infoString = (char*)malloc((strlen(kernelName) + 40) * sizeof(char));
        memset(infoString, 0, sizeof((strlen(kernelName) + 40) * sizeof(char)));
        strcat(infoString, "clCreateKernel ");
        strcat(infoString, kernelName);
        openclRetTackle(ret, infoString);

        free(infoString);
        free(sourceString);
        fclose(fp);

        return 0;
}
```

```
int ReverseInt(int i)
{
	unsigned char ch1, ch2, ch3, ch4;
	ch1 = i & 255;
	ch2 = (i >> 8) & 255;
	ch3 = (i >> 16) & 255;
	ch4 = (i >> 24) & 255;
	return((int)ch1 << 24) + ((int)ch2 << 16) + ((int)ch3 << 8) + ch4;
}

void ReadMNIST(int NumberOfImages, int DataOfAnImage, float *arr, string name)
{
	//arr = new float**[NumberOfImages];
	//ifstream file (name,ios::binary);
	ifstream file(name, ios::binary);
	if (file.is_open())
	{
		cout << "start read" << endl;
		int magic_number = 0;
		int number_of_images = 0;
		int n_rows = 0;
		int n_cols = 0;
		file.read((char*)&magic_number, sizeof(magic_number));
		magic_number = ReverseInt(magic_number);
		file.read((char*)&number_of_images, sizeof(number_of_images));
```

```
            number_of_images = ReverseInt(number_of_images);
            file.read((char*)&n_rows, sizeof(n_rows));
            n_rows = ReverseInt(n_rows);
            file.read((char*)&n_cols, sizeof(n_cols));
            n_cols = ReverseInt(n_cols);
            //#pragma simd
            for (int i = 0; i<NumberOfImages; ++i)
            {
                        //arr[i] = new float*[DataOfAnImage];
                        for (int r = 0; r<n_rows; ++r)
                        {
                                    //arr[i][r] = new float[DataOfAnImage];
                                    for (int c = 0; c<n_cols; ++c)
                                    {
                                                unsigned char temp = 0;
                                                file.read((char*)&temp, sizeof(temp));
                                                arr[i*n_cols*n_rows + r*n_rows + c] = (float)temp/256.0 ;
                                    }
                        }
            }
}
else
{
            printf("can not find data₩n");
            exit(1);
}
```

```
}
void ReadMNIST_Label(int NumberOfImages, float *arr, string name)
{
	/*arr = new float *[NumberOfImages];*/
	ifstream file(name, ios::binary);
	if (file.is_open())
	{
		int magic_number = 0;
		int number_of_images = 0;

		file.read((char*)&magic_number, sizeof(magic_number));
		magic_number = ReverseInt(magic_number);
		file.read((char*)&number_of_images, sizeof(number_of_images));
		number_of_images = ReverseInt(number_of_images);

		for (int i = 0; i<NumberOfImages; ++i)
		{
			//arr[i] = new float[10];
			unsigned char temp = 0;
			file.read((char*)&temp, sizeof(temp));
			int label = (int)temp;
			for (int j = 0; j<10; j++)
			{
				if (j == label)
					arr[i * 10 + j] = 1.0;
				else
```

```
                                        arr[i * 10 + j] = 0.0;
                                }
                        }
                }
}

int findIndex(float* p)
{
        int index = 0;
        float Max = p[0];
        for (int i = 1; i<10; i++)
        {
                float v = p[i];
                if (p[i] > Max)
                {
                        Max = p[i];
                        index = i;
                }
        }
        return index;
}

void prepareCNeurons(int nNeurons, int nKernels, int kernelWidth, int outimgsize, int inimgsize, string filePath_conv, string filePath_bias, vector<shared_ptr<CNeuron>> &cns) {
        int kernelSize = kernelWidth*kernelWidth;
```

```
ifstream fin_conv, fin_bias;
fin_conv.exceptions(ifstream::failbit | ifstream::badbit);
fin_bias.exceptions(ifstream::failbit | ifstream::badbit);
try {
        fin_conv.open(filePath_conv);
        fin_bias.open(filePath_bias);
        vector<float*> kernelsData;
        vector<float*> bias;
        float* data_conv = new float[nNeurons*nKernels*kernelSize];
        float* data_bias = new float[nNeurons];
        for (int i = 0; i < nNeurons; ++i) {
                string line;
                string strBias;
                getline(fin_bias, strBias);
                 // float bias = stof(strBias);
                data_bias[i] = stof(strBias);
                //data_bias[i] = 1;
                // cout << bias << " ";
                getline(fin_conv, line);
                stringstream iss(line);
                for (int j = 0; j < nKernels; ++j) {
                        for (int k = 0; k < kernelSize; ++k) {
                                string str;
                                getline(iss, str, ' ');
                                data_conv[i*nKernels*kernelSize + j*kernelSize + k] =
stof(str);
```

```
                                        //data_conv[i*nKernels*kernelSize + j*kernelSize + k] =
1;
                                        //    cout << weights_conv[k] << " ";
                                        //weights_conv[k] = 1;
                                    }
                                }
                            }
                                kernelsData.push_back(data_conv);
                                bias.push_back(data_bias);
                                //create neuron based on kernel data
                                CNeuron cn(kernelsData,bias, kernelWidth, cxGPUContext,
cqCommandQueue, kernel_conv, kernel_pool, kernel_FC, nNeurons, nKernels, inimgsize, outimgsize);

                                //create vector of neurons for convolution layer
                                cns.push_back(make_shared<CNeuron>(cn));

                                //should clean up that float* shit

                        fin_conv.close();
                }
                catch (ifstream::failure error) { cerr << error.what() << endl; }
}

/*void preparePNeurons(int nNeurons, string filePath, vector<shared_ptr<PNeuron>> &pns) {
                ifstream fin_pool;
                fin_pool.exceptions(ifstream::failbit | ifstream::badbit);
```

```
	try {
		fin_pool.open(filePath);
		vector<float*> bias;
		float* data = new float[nNeurons];
		for (int i = 0; i < nNeurons; ++i) {
			string strBias;
			getline(fin_pool, strBias);
			//   float bias = stof(strBias);
			//data[i] = stof(strBias);
			data[i] = 1;
			// cout << bias << " ";
		}
		bias.push_back(data);
		//create neuron based on bias
		PNeuron pn(bias, cxGPUContext, cqCommandQueue, kernel_conv, kernel_pool,
nNeurons);

		//create vector of neurons for pool layer
		pns.push_back(make_shared<PNeuron>(pn));
		fin_pool.close();
	}
	catch (ifstream::failure error) { cerr << error.what() << endl; }
}
*/
```

```
int init_cl() {
	cout << "initial opencl...₩n";
	openclInit();
	//printCLDeviceInfo();
	openclCreateKernelFromFile(&program, &kernel_conv, "conv.cl", "Convolution", 0);
	openclCreateKernelFromFile(&program, &kernel_pool, "conv.cl", "Pooling", 1);
	openclCreateKernelFromFile(&program, &kernel_FC, "conv.cl", "Fconnect", 1);
	return 0;
}

Neuron::Neuron(const cl_context & context, const cl_command_queue & commandQueue, const cl_kernel &kernel_conv, const cl_kernel &kernel_pool, const cl_kernel &kernel_FC) :
	context(context),
	commandQueue(commandQueue),
	kernel_conv(kernel_conv),
	kernel_pool(kernel_pool),
	kernel_FC(kernel_FC) {
}

CNeuron::CNeuron(vector<float*>kernelsdata, vector<float*> poolbias, int kernelWidth, const cl_context &context,
	const cl_command_queue &commandQueue, const cl_kernel &kernel_conv, const cl_kernel &kernel_pool, const cl_kernel &kernel_FC,int nNeurons,int nKernels, int inImgsize, int outImgsize) :
   Neuron(context,commandQueue, kernel_conv, kernel_pool, kernel_FC),
   kernelWidth(kernelWidth),
```

```
    nNeurons(nNeurons),
    nKernels(nKernels),
    outImgsize(outImgsize),
    inImgsize(inImgsize){
    cl_int ret;
    setKernels(kernelsdata, kernelWidth, nNeurons, nKernels);
    setpoolBias(poolbias, nNeurons);
    featureBuf = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * nNeurons * outImgsize * outImgsize , NULL, &ret);
    openclRetTackle(ret,"create feature buffer");
}

PLayer::PLayer(const cl_context &context, const cl_command_queue &commandQueue, const cl_kernel &kernel_pool, int nNeurons, int Imgsize) :
        context(context),
        commandQueue(commandQueue),
        kernel_pool(kernel_pool),
        nNeurons(nNeurons),
        Imgsize(Imgsize){
        cl_int ret;
        featureBuf = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * nNeurons * Imgsize * Imgsize, NULL, &ret);
        openclRetTackle(ret, "create feature buffer");
};
```

```
void CNeuron::convolve(cl_mem & inFMaps) {
	int convImgWidth = 1;
	int convImgHeight = 1;
	int feature_size = inImgsize * inImgsize;
	if (kernelWidth != 1) {
		convImgWidth = inImgsize - kernelWidth + 1;
		convImgHeight = inImgsize - kernelWidth + 1;
		size_t range_width[] = { nNeurons,convImgWidth * convImgHeight };
		size_t range_height[] = { 1,1 };
		//just to init buffer by zeros
		cl_int ret;



		/*
			cl_mem buf;
			float *img = new float[28 * 28];
			for (int i = 0; i != 28 * 28; ++i) {
				img[i] = 1;
			}
			buf=clCreateBuffer(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(float) * 28 * 28, img, &ret);
			float *data = new float[28 * 28];
			clEnqueueReadBuffer(cqCommandQueue, buf, CL_TRUE, 0, sizeof(float)*28*28, data, 0, NULL, NULL);
			for (int i = 0; i != 28 * 28; ++i) {
				cout << data[i];
```

```
                }
                */


                openclRetTackle(clSetKernelArg(kernel_conv, 1, sizeof(int), &kernelWidth), "clSetKernelArg 1");
                openclRetTackle(clSetKernelArg(kernel_conv, 3, sizeof(cl_mem),&featureBuf), "clSetKernelArg 3");
                openclRetTackle(clSetKernelArg(kernel_conv, 4, sizeof(int), &feature_size), "clSetKernelArg 4");
                openclRetTackle(clSetKernelArg(kernel_conv, 5, sizeof(int), &nKernels), "clSetKernelArg 5");
                openclRetTackle(clSetKernelArg(kernel_conv, 6, sizeof(int), &inImgsize), "clSetKernelArg 6");
                openclRetTackle(clSetKernelArg(kernel_conv, 7, sizeof(int), &convImgHeight), "clSetKernelArg 7");
                openclRetTackle(clSetKernelArg(kernel_conv, 8, sizeof(cl_mem), &Bias), "clSetKernelArg 8");


                openclRetTackle(clSetKernelArg(kernel_conv, 0, sizeof(cl_mem), &inFMaps), "clSetKernelArg 0");
                openclRetTackle(clSetKernelArg(kernel_conv, 2, sizeof(cl_mem), &kernelBuf), "clSetKernelArg 2");
                openclRetTackle(clEnqueueNDRangeKernel(commandQueue, kernel_conv, 2, NULL, range_width, range_height, 0, NULL, NULL), "clEnqueueNDRangeKernel  kernelconv");
                clFinish(commandQueue);
                int a = 1;
#if (defined TEST)
                float *data = new float[convImgWidth*convImgHeight*nNeurons];
```

```
                clEnqueueReadBuffer(cqCommandQueue, featureBuf, CL_TRUE, 0, sizeof(float) * convImgWidth * convImgHeight * nNeurons, data, 0, NULL, NULL);
                for (int i = 0; i != nNeurons; ++i) {
                        for (int j = 0; j != convImgWidth; ++j) {
                                for (int k = 0; k != convImgWidth; ++k) {
                                        printf("%f ", data[i*convImgWidth*convImgWidth + j*convImgWidth + k]);
                                }
                                printf("₩n");
                        }
                        printf("₩n");
                }
                delete[] data;
#endif
                //delete[] img;
                //delete[] data;
        }
        else {
                size_t range_width[] = { nNeurons };
                size_t range_height[] = { 1 };
                //just to init buffer by zeros
                cl_int ret;

                openclRetTackle(clSetKernelArg(kernel_FC, 1, sizeof(cl_mem), &featureBuf), "clSetKernelArg 1");
                openclRetTackle(clSetKernelArg(kernel_FC, 4, sizeof(int), &nKernels), "clSetKernelArg 4");
```

```
		openclRetTackle(clSetKernelArg(kernel_FC, 0, sizeof(cl_mem), &inFMaps), "clSetKernelArg 0");
		openclRetTackle(clSetKernelArg(kernel_FC, 2, sizeof(cl_mem), &kernelBuf), "clSetKernelArg 2");
		openclRetTackle(clSetKernelArg(kernel_FC, 3, sizeof(cl_mem), &Bias), "clSetKernelArg 3");
		openclRetTackle(clEnqueueNDRangeKernel(commandQueue, kernel_FC, 1, NULL, range_width, range_height, 0, NULL, NULL), "clEnqueueNDRangeKernel kernelFC");
		clFinish(commandQueue);
#if (defined TEST)
		float *data = new float[convImgWidth*convImgHeight*nNeurons];
		clEnqueueReadBuffer(cqCommandQueue, featureBuf, CL_TRUE, 0, sizeof(float) * convImgWidth * convImgHeight * nNeurons, data, 0, NULL, NULL);
		for (int i = 0; i != nNeurons; ++i) {
			for (int j = 0; j != convImgWidth; ++j) {
				for (int k = 0; k != convImgWidth; ++k) {
					printf("%f ", data[i*convImgWidth*convImgWidth + j*convImgWidth + k]);
				}
				printf("₩n");
			}
			printf("₩n");
		}
		delete[] data;
#endif
		//delete[] img;
```

```
                    //delete[] data;
            }
}


void CNeuron::setKernels(vector<float*>kernelsdata, int kernelWidth,int nNeurons,int nKernels) {
    cl_int ret;
    kernelWidth   = kernelWidth;
    kernelsData = kernelsdata;
    kernelBuf = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, sizeof(float)
* kernelWidth * kernelWidth * nNeurons * nKernels, (void*)kernelsData[0], &ret);
    openclRetTackle(ret, "clCreateBuffer const");
}


/*PNeuron::PNeuron(vector<float*> poolbias, const cl_context & context, const cl_command_queue
& commandQueue, const cl_kernel &kernel_conv, const cl_kernel &kernel_pool, int nNeurons) :
    Neuron(context,commandQueue,kernel_conv,kernel_pool){
             setpoolBias(poolbias, nNeurons);
}
*/
void CNeuron::setpoolBias(vector<float*> poolbias,int nNeurons) {
            cl_int ret;
            poolBias = poolbias;
            Bias = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, sizeof(float)
```

```
* nNeurons, (void*)poolBias[0], &ret);
		openclRetTackle(ret, "clCreateBuffer const");
}
/*
void PNeuron::setpoolBias(vector<float*> poolbias, int nNeurons) {
		cl_int ret;
		poolBias = poolbias;
		Bias = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, sizeof(float)
* nNeurons, (void*)poolBias[0], &ret);
		openclRetTackle(ret, "clCreateBuffer const");
}
*/

void PLayer::pool(cl_mem & inFMaps)
{
	cl_int ret;
	int inWidth = Imgsize * 2;
	int poolImgWidth = Imgsize;
	int poolImgHeight = Imgsize;
	size_t range_width[] = { nNeurons, poolImgWidth * poolImgHeight };
	size_t range_height[] = { 1,1 };



	openclRetTackle(clSetKernelArg(kernel_pool, 0, sizeof(cl_mem), &inFMaps), "clSetKernelArg 0");
	openclRetTackle(clSetKernelArg(kernel_pool, 1, sizeof(cl_mem), &featureBuf), "clSetKernelArg 1");
```

```
    openclRetTackle(clSetKernelArg(kernel_pool, 2, sizeof(int), &inWidth), "clSetKernelArg 2");
    openclRetTackle(clSetKernelArg(kernel_pool, 3, sizeof(int), &Imgsize), "clSetKernelArg 3");

    openclRetTackle(clEnqueueNDRangeKernel(commandQueue, kernel_pool, 2, NULL, range_width, range_height, 0, NULL, NULL), "clEnqueueNDRangeKernel    kernelpool");
    clFinish(commandQueue);
//    kernel_pool.setArg(0, sizeof(cl_mem), (void*)&poolImgBuf);


#if (defined TEST)
    float *data = new float[poolImgWidth * poolImgHeight * nNeurons];
    clEnqueueReadBuffer(cqCommandQueue, featureBuf, CL_TRUE, 0, sizeof(float) * poolImgWidth * poolImgHeight * nNeurons, data, 0, NULL, NULL);
    for (int i = 0; i != nNeurons; ++i) {
            for (int j = 0; j != poolImgWidth; ++j) {
                    for (int k = 0; k != poolImgHeight; ++k) {
                            printf("%f ", data[i * poolImgWidth * poolImgHeight + j * poolImgWidth + k]);
                    }
                    printf("₩n");
            }
            printf("₩n");
    }
    delete[] data;
#endif
```

```
}




Layer::~Layer() {}

Layer::Layer() {}



ILayer::ILayer(int imgsize, const cl_context &context) {

	cl_int ret;

	featureBuf = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * imgsize * imgsize, NULL, &ret);

	openclRetTackle(ret, "create feature buffer");


}


void ILayer::activate(float* inImage, const cl_context &context, const cl_command_queue &commandQueue, int imagesize) {

	cl_int ret;

	ret = clEnqueueWriteBuffer(commandQueue, featureBuf, CL_TRUE, 0, sizeof(float)*imagesize*imagesize, inImage, 0, NULL, NULL);

	openclRetTackle(ret, "create input feature buffer");

}
```

```
OLayer::OLayer() {}

void OLayer::activate(cl_mem & prevFeatureMaps) { }



HiddenLayer::~HiddenLayer() {}
HiddenLayer::HiddenLayer() {}

void HiddenLayer::activate(cl_mem & prevFeatureMaps)
{
   cout << "Virtual method of base class called. This shouldn't happen.";
}

CLayer::CLayer(vector<shared_ptr<CNeuron>> neurons) :
   neurons(neurons) { }

//CCLayer::CCLayer(vector<shared_ptr<CNeuron>> neurons) :
//          neurons(neurons) { }

void CLayer::activate(cl_mem & prevFeatureMaps) {

          neurons[0].get()->convolve(prevFeatureMaps);

}
```

```
/*
void CCLayer::activate(FeatureMaps prevFeatureMaps) {
	featureMaps.width = prevFeatureMaps.width - neurons[0].get()->kernelsize() + 1;
	featureMaps.height = prevFeatureMaps.height - neurons[0].get()->kernelsize() + 1;

	for (size_t i = 0; i < neurons.size(); i++) {
		featureMaps.buffers.push_back(neurons[i].get()->connect(prevFeatureMaps));
	}
}
*/



void PLayer::activate(cl_mem & prevFeatureMaps) {

		pool(prevFeatureMaps);

}
```

```
#define __CL_ENABLE_EXCEPTIONS


#include "layer.h"


//#include <opencv₩highgui.h>

//#include<opencv2/core/core.hpp>

//#include<opencv2/highgui/highgui.hpp>

#include <CL₩cl.hpp>


#include <iostream>

#include <fstream>

#include <ctime>

#include <vector>


using namespace std;

//using namespace cv;


extern cl_context           cxGPUContext;

extern cl_command_queue cqCommandQueue;

extern cl_kernel kernel_pool;

/*

shared_ptr<cl_mem> create() {

        float * a = new float[10];

        for (int i = 0; i != 10; ++i) {

                a[i] = i;

        }
```

```
	cl_mem	p = clCreateBuffer(cxGPUContext, CL_MEM_COPY_HOST_PTR, sizeof(float) * 10, a,
NULL);
	delete[] a;
	return(make_shared<cl_mem>(p));
}
*/
int main(int argc, char** argv)
{
	int test_image = 10000;
	float *inImage = new float[28 * 28];
	float *output = new float[10];
	float *true_out = new float[10];
	int inImgWidth = 28;
	int inImgHeight = 28;

	float *data = new float[10000 * 28 * 28];
	float *label = new float[10000 * 10];

	//input data
	ReadMNIST(test_image, inImgHeight, data, "t10k-images.idx3-ubyte");
	ReadMNIST_Label(test_image, label, "t10k-labels.idx1-ubyte");
/*	std::ofstream fout("mnist.txt");
	if (!fout)
	{
		std::cout << "ÎÄ¼þ»ÄÜ´ò¿ª" << std::endl;
	}
```

```
        else
        {
                for (int i = 0; i != 28; ++i) {
                        for (int j = 0; j != 28; ++j) {
                                fout << data[28*28*3 + i * 28 + j] << " ";
                                }
                        fout << std::endl;
                }
                fout.close();
        }
  */
  //initial opencl
    int ans = init_cl();
  //FeatureMaps P;

 // P.buffers.push_back(create());
 // float *b = new float[10];
 // clEnqueueReadBuffer(cqCommandQueue, *P.buffers[0].get(), CL_TRUE, 0, sizeof(float) * 10, b, 0, NULL, NULL);
 // for (int i = 0; i != 10; ++i) {
//              cout << b[i] << " ";
 // }


  cout << "CNN layers are preparing...₩n";
  //0 conv layer neurons
```

```
 int kernelWidth0 = 5;
 vector<shared_ptr<CNeuron>> cns0;
 //prepareCNeurons(2, 1, 5, 24, 28, "conv1.txt", "conv1_bias.txt", cns0);
 prepareCNeurons(20, 1, 5, 24, 28, "conv1.txt","conv1_bias.txt", cns0);

 //1 conv layer neurons
 int kernelWidth1 = 5;
 vector<shared_ptr<CNeuron>> cns1;
// prepareCNeurons(5, 2, 5, 8, 12, "conv2.txt", "conv2_bias.txt", cns1);
 prepareCNeurons(50, 20, 5, 8, 12, "conv2.txt", "conv2_bias.txt", cns1);

 //2 conv layer neurons (22 layer in matlab code)
 int kernelWidth2 = 1;
 vector<shared_ptr<CNeuron>> cns2;
// prepareCNeurons(5, 80, 1, 1, 4, "ip1.txt","ip1_bias.txt",cns2);
 prepareCNeurons(500, 800, 1, 1, 4, "ip1.txt","ip1_bias.txt", cns2);

 //3 (Out) conv layer neurons
 int kernelWidth3 = 1;
 vector<shared_ptr<CNeuron>> cns3;
// prepareCNeurons(10, 5, 1, 1, 1, "ip2.txt","ip2_bias.txt", cns3);
 prepareCNeurons(10, 500, 1, 1, 1, "ip2.txt","ip2_bias.txt", cns3);


 //init layers
 shared_ptr<ILayer> iLayer(make_shared<ILayer>(28, cxGPUContext));
```

```
shared_ptr<CLayer> cLayer0(make_shared<CLayer>(cns0));
shared_ptr<PLayer> pLayer0(make_shared<PLayer>(cxGPUContext, cqCommandQueue, kernel_pool, 20, 12));
shared_ptr<CLayer> cLayer1(make_shared<CLayer>(cns1));
shared_ptr<PLayer> pLayer1(make_shared<PLayer>(cxGPUContext, cqCommandQueue, kernel_pool, 50, 4));
shared_ptr<CLayer> cLayer2(make_shared<CLayer>(cns2));
shared_ptr<CLayer> outCLayer(make_shared<CLayer>(cns3));

int flag = 0, total = 10000;
cout << "Layers are ready. Let's run!₩n";
int ture_label = 0, test_label = 0;
//cnn run
double start, stop, durationTime, totaltime = 0;
for (int i = 0; i != total; ++i) {
        for (int j = 0; j != 28 * 28; ++j) {
                inImage[j] = data[i * 28 * 28 + j];
               // inImage[j] = 1;
              //   cout << inImage[j] << " ";
               // if (j % 28 == 0) { cout << endl; }
        }
       // cout << endl;
        for (int k = 0; k != 10; ++k) {
                true_out[k] = label[i*10+k];
        }
```

```
start = clock();
iLayer->activate(inImage, cxGPUContext, cqCommandQueue, inImgWidth);
cLayer0->activate(iLayer->getFeature());
pLayer0->activate(cLayer0->getFeature());

cLayer1->activate(pLayer0->getFeature());
pLayer1->activate(cLayer1->getFeature());

cLayer2->activate(pLayer1->getFeature());
outCLayer->activate(cLayer2->getFeature());
stop = clock();

durationTime = ((double)(stop - start)) / CLK_TCK;
// cout << "µ¥´ïºÄÊ±£º" << durationTime << " s" << endl;
totaltime += durationTime;
clEnqueueReadBuffer(cqCommandQueue, outCLayer->getFeature(), CL_TRUE, 0, sizeof(float) * 10, output, 0, NULL, NULL);
/*for (size_t w = 0; w != 10; w++) {
        cout << output[w] << " ";
}
*/
ture_label = findIndex(true_out);
test_label = findIndex(output);
if (ture_label == test_label) { flag++; }
if (i % 100 == 0) {
```

```
                        cout << "examples:" << i << "    true label: " << ture_label << " test label: " << test_label << " right num: " << flag << endl;
                }

    }
    cout << "³ÌÐò¹ÄÊ±£º" << totaltime << " s" << endl;
    cout << "Æ½¾ùÃ¿´Î¹ÄÊ±£º" << totaltime/ total << " s" << endl;
    cout << "×¼È·ÂÊ:" << float(flag) / total << endl;
    /*
    char* x = new char[32];

    FeatureMaps out = outPLayer->getFeatureMaps();
    for (size_t i = 0; i < out.buffers.size(); i++) {
        cl::Buffer *o = out.buffers[i].get();
        Mat image = Mat::zeros(Size(out.width, out.height), CV_32FC3);
        commandQueue.enqueueReadBuffer(*o, CL_TRUE, 0, sizeof(cl_float) * 3 * out.width * out.height, image.data);
        sprintf(x, "output%d.png", i);
        image.convertTo(image, CV_8UC3);
        imwrite(x, image);
    }

    delete[] x;
    */
    cout << "Done!₩n";
    delete[] data;
```

```
    delete[] label;
    delete[] inImage;
    delete[] output;
    delete[] true_out;
    system("pause");
    return 0;
}
```

```
#pragma once
#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <memory>
//#include <opencv₩highgui.h>
#include <CL₩cl.h>
#include <fstream>

using namespace std;
//using namespace cv;




struct FeatureMaps {
   vector<shared_ptr<cl_mem>> buffers;
   int width;
   int height;
};

class Neuron {
protected:
   //const stuff
   const cl_context          &context;
```

```
    const cl_command_queue &commandQueue;
    const cl_kernel &kernel_conv;
    const cl_kernel &kernel_pool;
    const cl_kernel &kernel_FC;
public:
    Neuron(const cl_context &context, const cl_command_queue &commandQueue, const cl_kernel &kernel_conv, const cl_kernel &kernel_pool, const cl_kernel &kernel_FC);
};

class CNeuron : public Neuron {
private:

    int                 kernelWidth;
    int                 nNeurons;
    int                 nKernels;
    int                 outImgsize;
    int                 inImgsize;
    vector<float*> kernelsData;

    cl_mem kernelBuf;
    vector<float*> poolBias;
    cl_mem Bias;

public:
    CNeuron(vector<float*>kernelsdata, vector<float*> poolbias, int kernelWidth, const cl_context &context, const cl_command_queue &commandQueue,
```

```
        const cl_kernel &kernel_conv, const cl_kernel &kernel_pool, const cl_kernel &kernel_FC, int nNeurons, int nKernels, int inimgsize, int outimgsize);
    int kernelsize() { return kernelWidth; };

    void convolve(cl_mem & inFMaps);

    void setKernels(vector<float*>kernelsData, int width, int nNeurons, int nKernels);
    void setpoolBias(vector<float*>poolbias, int nNeurons);
    vector<float*> kernel() { return kernelsData; };
    cl_mem featureBuf;
};

/*class PNeuron : public Neuron {
private:

    vector<float*> poolBias;
    cl_mem Bias;
public:
    PNeuron(vector<float*> poolBias, const cl_context &context, const cl_command_queue &commandQueue, const cl_kernel &kernel_conv, const cl_kernel &kernel_pool, int nNeurons);

    shared_ptr<cl_mem> PNeuron::pool(const shared_ptr<cl_mem> buffer, int outWidth, int outHeight, float poolCoef);
    void setpoolBias(vector<float*>poolbias, int nNeurons);
};
*/
```

```
class Layer {
protected:
   FeatureMaps featureMaps;
public:
   ~Layer();
   Layer();
   FeatureMaps getFeatureMaps() { return featureMaps; };
};

class ILayer : public Layer {
private:
   cl_mem featureBuf;
public:
   ILayer(int imgsize, const cl_context &context);
   void activate(float* inImage, const cl_context &context, const cl_command_queue &commandQueue, int imagesize);
   cl_mem & getFeature() { return featureBuf; };
};

class OLayer : public Layer {
public:
   OLayer();
   void activate(cl_mem & getFeatureMaps);
};
```

```
class HiddenLayer : public Layer {
protected:
public:
   virtual ~HiddenLayer();
   HiddenLayer();
   virtual void activate(cl_mem & prevFeatureMaps);
};

class CLayer : public HiddenLayer {
protected:
   vector<shared_ptr<CNeuron>> neurons;
public:
   CLayer(vector<shared_ptr<CNeuron>> neurons);
   void activate(cl_mem    & prevFeatureMaps) override;
   cl_mem & getFeature() { return neurons[0].get()->featureBuf; };
};

class PLayer : public HiddenLayer {
private:
            const cl_context          &context;
            const cl_command_queue &commandQueue;
            const cl_kernel &kernel_pool;
            int nNeurons;
            int Imgsize;
            cl_mem featureBuf;
```

```
public:
	PLayer(const cl_context &context, const cl_command_queue &commandQueue, const cl_kernel &kernel_pool, int nNeurons, int imgsize);
	void activate(cl_mem & prevFeatureMaps) override;
	void pool(cl_mem & inFMaps);
	cl_mem & getFeature() { return featureBuf; };
};
/*
class CCLayer : public HiddenLayer {
protected:
	vector<shared_ptr<CNeuron>> neurons;
	float poolCoef;
public:
	CCLayer(vector<shared_ptr<CNeuron>> neurons);
	void activate(FeatureMaps prevFeatureMaps) override;
};
*/
void openclRetTackle(cl_int retValue, char* processInfo);
void printCLDeviceInfo();
int openclInit();
int openclCreateKernelFromFile(cl_program* cpProgram, cl_kernel* clKernel, const char* clFileName, const char* kernelName, int flag);
int init_cl();
int ReverseInt(int i);
void ReadMNIST(int NumberOfImages, int DataOfAnImage, float *arr, string name);
void ReadMNIST_Label(int NumberOfImages, float *arr, string name);
```

```
int findIndex(float* p);

void prepareCNeurons(int nNeurons, int nKernels, int kernelWidth, int outimgsize, int inimgsize, string filePath_conv, string filePath_bias, vector<shared_ptr<CNeuron>> &cns);

//void preparePNeurons(int nNeurons, string filePath, vector<shared_ptr<PNeuron>> &pns);
```