**Shanxi University**

School of Computer and Information Technology

《**parallel computing**》experimental report

**Project name：**

Parallel implementation of image classification based on CNN pre-training network

Name : 徐 世 琦

Student ID: 201801004239

Class : Data Science and Big Data Technology Class 2

Mentor : Zhiwei Qiao

Update date: 2021/ 6 / 10

**catalogue**

[1. Project Purpose 1](#_Toc74045584)

[2. Experimental environment 1](#_Toc74045585)

[1) First step: Monolayer convolution 1](#_Toc74045586)

[2) Second step: Accumulate layer by layer to get the final convolution result 2](#_Toc74045587)

[3) Third step: Pooling processing 3](#_Toc74045588)

[4) Fourth step: Fully Connected layer operation 4](#_Toc74045589)

[4 Project Code and Result 5](#_Toc74045590)

[1）Kernel function Defination 5](#_Toc74045591)

[2）Main Funcion calling kernel 5](#_Toc74045592)

[3）Result display 12](#_Toc74045593)

[5 Personal contribution 13](#_Toc74045594)

[6 Experience sharing 13](#_Toc74045595)

[7 others 14](#_Toc74045596)

[8 References 14](#_Toc74045597)

# Project Purpose

What we want to design is parallel implementation of image classification based on CNN pre-training network .Usually a CNN Forward Propagation includes numerous operation which called “layer” such as Convolutional layer ,Pooling layer and Fully Connected layer, somehow it is sometimes slow to be run with cpus when the matrix is so large , therefore we want use CUDA to accelerate operations with numerous CUDA cores.

# Experimental environment

**CUDA version :** 10.0

**System environment** ：windows 10

**Development Environment** ：Visiual Stdio 2017

**Third-party library** ：opencv 4.5

**GPU Attributes**:

model : MX130

SMs : 3

CUDA cores:386

Architecture：Maxwell

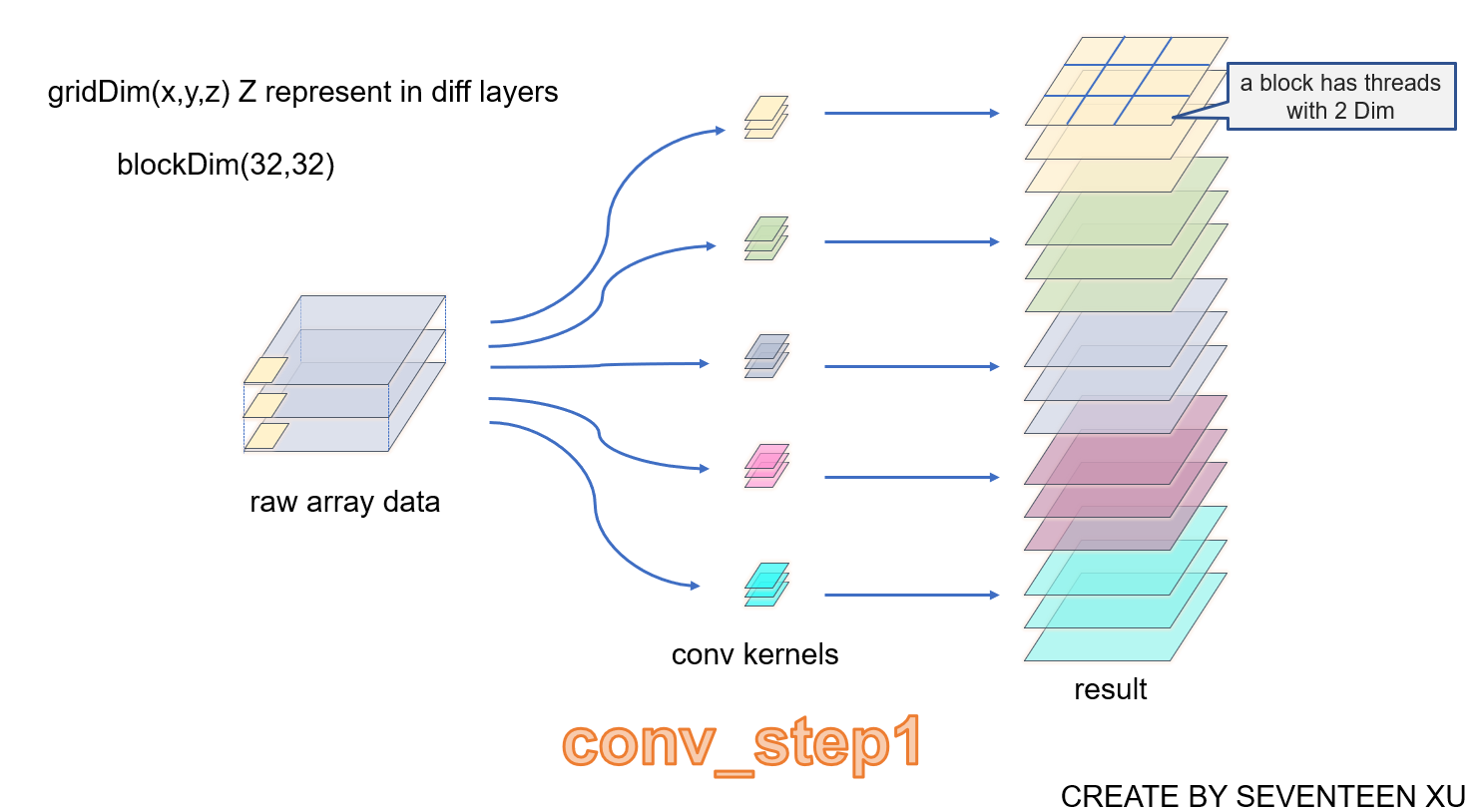
1. **Project design process**

We divide the entire CNN Forward Propagation into four parts ,for each part we have designed a kernel function to match with. Here are specific parts we design:

## 1) First step: Monolayer convolution

**Purpose：**Get the result of single layer convolution

**Approach：**Do single-layer convolution in convolution, use all layers of N convolution cores andcorresponding layers of featuremap to do single-layer convolution operation, and generate a conv with the same size of featuremap and the same number of layers as the total number of convolution conv\_ temp array. Picture 3-1 shows the processes depicted above.



Pic 3-1

dim gridDim (block x coordinate , block y coordinate , the number of convolution kernels )

dim blockDim( width in a tile，Height ≤ 32 ).

Because the number of threads in a block is limited (a block can only have 1024 threads at most), the block in a featuremap layer is a two-dimensional structure, and the threads are divided according to the operation results. For example, if the size of a single layer is 32 \* 32, the number of threads is 32 \* 32.

Blockdim is the number of convolution cores. Each thread needs to calculate the convolution data of N layers and save it to the result. Each thread is responsible for computing multiple data.

## 2) Second step: Accumulate layer by layer to get the final convolution result

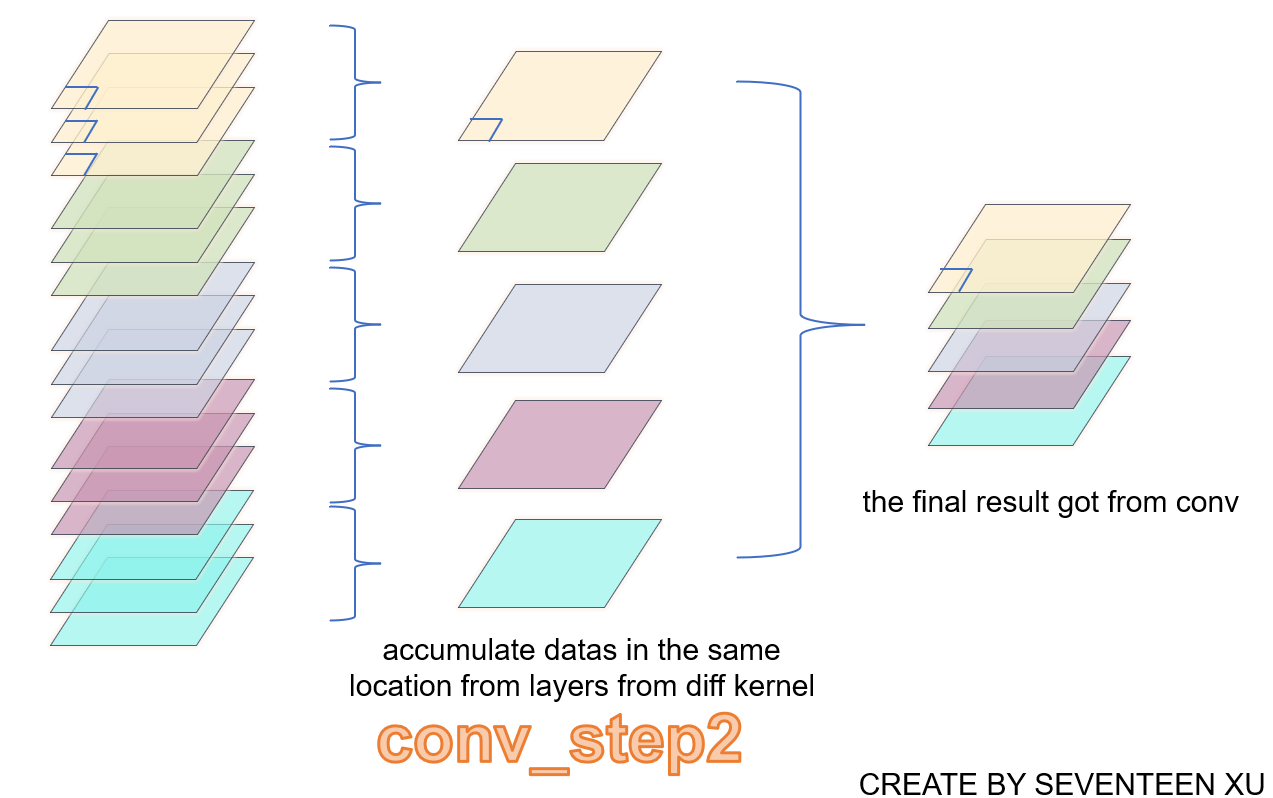
**Purpose** :Summing the layers at the same position in conv\_temp longitudinally and summing them into a result is stored in out\_gpu.

**Approaches:**

GridDim is three dimensional, like (block\_x,block\_y,kernel\_num).

BlockDim is two dimensional, like (x,y).

Picture 3-2 shows the processes depicted above.



Pic 3-2

## 3) Third step: Pooling processing

**Purpose:**

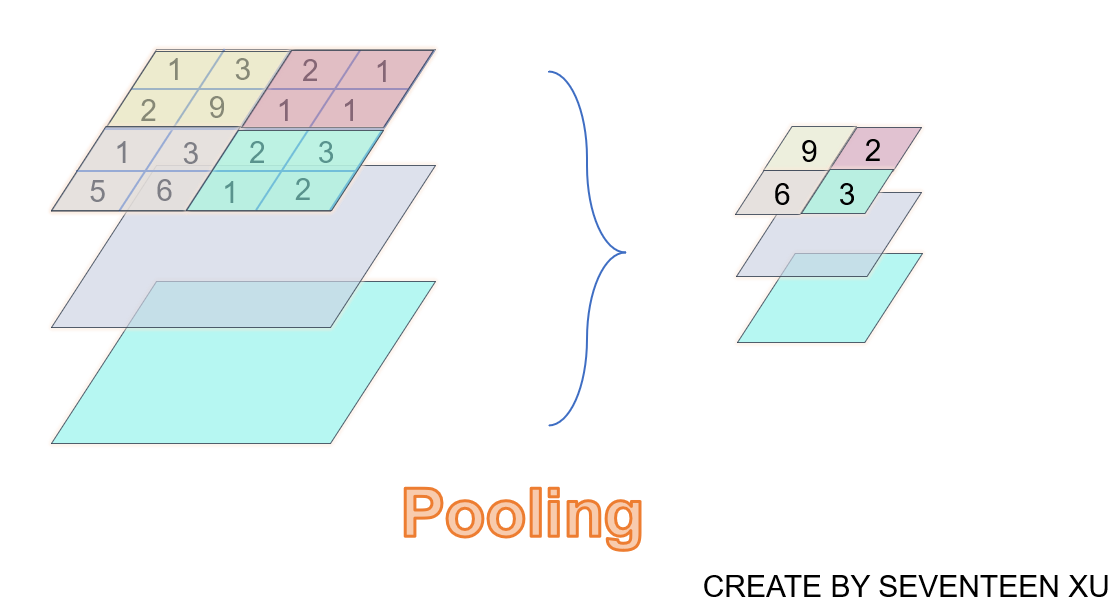
Maintaining the main features while reducing the amount of data to prevent overfitting.

**Approaches:**

Calculates the maximum area covered by the pooled window and assigns a value to the output.

GridDim is a three-dimensional representation of the number of layers of feature.

BlockDim is a two-dimensional (x,y) representation of the thread arrangement in the block corresponding to the column and column data in the feature. Picture 3-1 shows the processes depicted above



Pic 3-3

## 4) Fourth step: Fully Connected layer operation

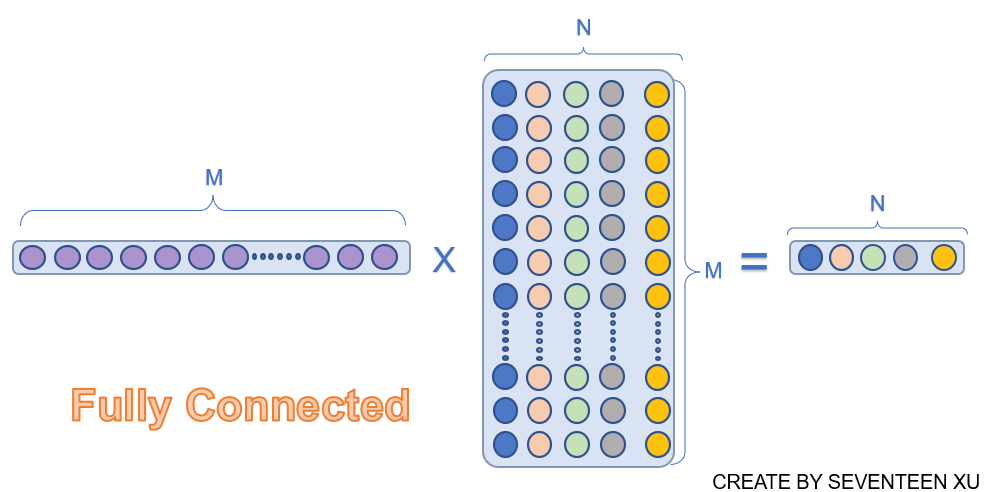
**Purpose:**

Connect all the features and send the output value to the classifier. The Fully Connected Layers (FC) plays the role of "classifier" in the whole convolutional neural network, and plays the role of mapping the learned "distributed feature representation" to the sample label space.

**Approaches:**

A matrix is obtained by connecting the pooled feature with the weight matrix. The thread cannot put a lot of data to the shared Mem at one time, so the matrix is divided into multiple tiles and multiplied by the weight matrix using tile idea. Each time, take a tile data, multiply it with the corresponding weight matrix, and finally generate a one-dimensional result.

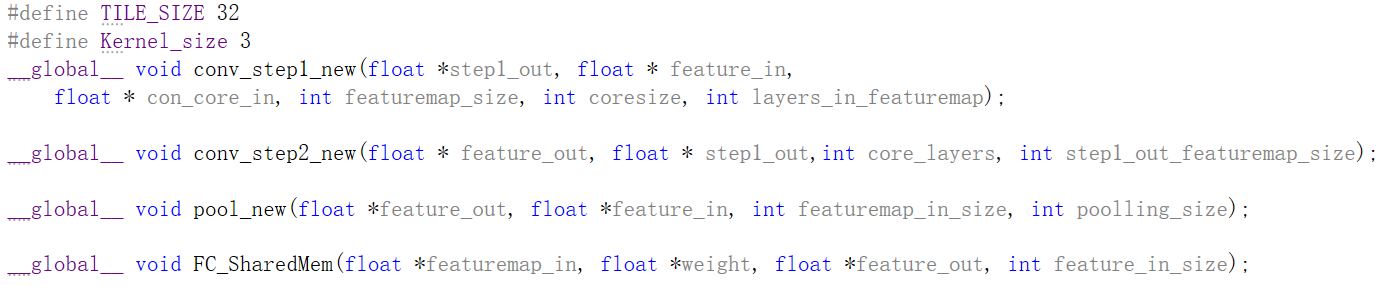
Picture 3-4 shows the processes depicted above



Pic 3-4

# 4 Project Code and Result

## 1）Kernel function Defination

Kernel function defined in cuda\_cnn.h

## 2）Main Funcion calling kernel

This is the main function that we implemented the entire image classification based on CNN pre-training network . we read the pic data with the help of opencv library ,then we initial kernel weight artificially, through Convolutional layer ,Pooling layer and Fully Connected layer,the raw image data will be transfer to a result matrix representing predict value for different class. The index of the argmax of the result matrix is the class the CNN predict.

**The whole main function shows below:**

void FullCnnProject() {

//opencv读取图片 512\*512

Mat raw\_img = imread("Lena.jpg");

Mat grey\_img,small\_img;

printf("图片大小为 %d行 %d列 %d通道 \n ", raw\_img.rows, raw\_img.cols, raw\_img.channels());

cout << "像素类型为："<< raw\_img.type() << endl;

//imshow("读取图像展示", img);

//waitKey(1500);

//转换成灰度图

//cvtColor(raw\_img, grey\_img, CV\_BGR2GRAY);

//printf("图片大小为 %d行 %d列 %d通道 \n ", grey\_img.rows, grey\_img.cols, grey\_img.channels());

resize(raw\_img, small\_img, Size(64, 64), INTER\_AREA);

printf("图片缩放后的大小为 %d行 %d列 %d通道 \n ", small\_img.rows, small\_img.cols, small\_img.channels());

//将图像像素数值转换成浮点型

//我们自己定义float型数组，然后将数据转换过来就行

int pic\_input\_size = small\_img.rows \* small\_img.cols\*raw\_img.channels();

float \* img\_input=(float \*)malloc(pic\_input\_size \*sizeof(float));

//循环 assignment

for (int channel = 0; channel < small\_img.channels(); channel++)

{

for (int i = 0; i < small\_img.rows; i++)

{

for (int j = 0; j < small\_img.cols; j++)

{

img\_input[channel\*small\_img.rows \* small\_img.cols+i\*small\_img.cols + j] =

(float)small\_img.data[channel\*small\_img.rows \* small\_img.cols+i\*small\_img.cols + j] / 256;

}

}

}

//imshow("缩小图像展示", small\_img);

//waitKey(1500);

for (int channel = 0; channel < small\_img.channels(); channel++)

{

for (int i = 0; i < small\_img.rows; i++)

{

for (int j = 0; j < small\_img.cols; j++)

{

printf("%f \t", img\_input[channel\*small\_img.rows \* small\_img.cols + i \* small\_img.cols + j]);

}

printf(" \n \n");

}

printf(" \n 图像一层 \n");

}

cout << "##########################以上为图像数组输出########################### " << "\n"<< "\n";

//////////////////////////////////////////////////////////////////////

/////// 接下来就是传入整个卷积网络的运算了！！！ /////////

//////////////////////////////////////////////////////////////////////

///////////////////// 一 卷积 step1 ////////////////////////////////////

//首先 原图像数组已经有了

int input\_3d\_size =small\_img.channels()\* small\_img.rows\*small\_img.cols;

int kernel\_3d\_size = 10 \* small\_img.channels() \* 3\*3;

//step1\_out大小

int step1\_out\_size = 10 \* small\_img.channels() \* small\_img.rows\*small\_img.cols;

float \*kernel\_in\_cpu = (float\*)malloc(kernel\_3d\_size \* sizeof(float));

float \*feature\_out\_cpu = (float\*)malloc(step1\_out\_size \* sizeof(float));

// 10个 kernel 赋值

for(int num=0;num<10;num++){

for (int layer = 0; layer < small\_img.channels(); layer++){

for(int i = 0; i < Kernel\_size; i++) {

for (int j = 0; j < Kernel\_size; j++) {

kernel\_in\_cpu[num \* Kernel\_size \* Kernel\_size\*small\_img.channels()

+ layer\* Kernel\_size \* Kernel\_size

+ i \* Kernel\_size + j] = i + 1;

}

}

}

}

//输出结果调试

for (int num = 0; num < 10; num++) {

for (int layer = 0; layer < small\_img.channels(); layer++) {

for (int i = 0; i < Kernel\_size; i++) {

for (int j = 0; j < Kernel\_size; j++) {

printf("%f \t", kernel\_in\_cpu[num \* Kernel\_size \* Kernel\_size\*small\_img.channels()

+ layer \* Kernel\_size \* Kernel\_size

+ i \* Kernel\_size + j] );

}

}

}

printf(" \n 一个 卷积核的数据 \n ");

}

float \*feature\_in\_gpu, \*kernel\_in\_gpu, \*out\_gpu;

cudaMalloc((void\*\*)&feature\_in\_gpu, sizeof(float)\*input\_3d\_size);

cudaMalloc((void\*\*)&kernel\_in\_gpu, sizeof(float)\*kernel\_3d\_size);

cudaMalloc((void\*\*)&out\_gpu, sizeof(float)\*step1\_out\_size);

printf("\n cudaMalloc 完成\n");

// 将CPU中的数据传到 GPU内存中

cudaMemcpy(feature\_in\_gpu, img\_input, sizeof(float)\*input\_3d\_size, cudaMemcpyHostToDevice);

cudaMemcpy(kernel\_in\_gpu, kernel\_in\_cpu, sizeof(float)\*kernel\_3d\_size, cudaMemcpyHostToDevice);

printf("\n cudaMemcpy 完成\n");

//设计传入的线程 启用线程

dim3 dimGrid(2,2,10);

dim3 dimBlock(32,32);// 线程的形状

//调用CUDA API时我们应该传入的是device的内存指针！！！

//否则直接就调用不了kernel

conv\_step1\_new<<<dimGrid, dimBlock >>>(out\_gpu, feature\_in\_gpu, kernel\_in\_gpu,64,3,3);

printf("\n函数调用完成\n");

//将GPU计算好的内存返回给CPU

//copy back

cudaMemcpy(feature\_out\_cpu, out\_gpu, sizeof(float)\*step1\_out\_size, cudaMemcpyDeviceToHost);

//输出featrue\_out\_cpu 结果 调试用

printf("---------------------------------------\n");

printf("-------输出featrue\_out\_cpu结果-----------\n");

printf("---------------------------------------\n");

/\*for (int num = 0; num < 10; num++) {

for (int layer = 0; layer < small\_img.channels(); layer++) {

for (int i = 0; i < small\_img.rows; i++) {

for (int j = 0; j < small\_img.cols; j++) {

printf("%f \t", feature\_out\_cpu[num \* small\_img.rows\* small\_img.cols\*small\_img.channels()

+ layer \* small\_img.rows\* small\_img.cols

+ i \* small\_img.cols + j]);

}

}

printf(" \n 一个层卷积层的结果 \n ");

}

}\*/

printf("\n --------- conv step 1 全部完成 ------ \n");

cudaFree(feature\_in\_gpu);

cudaFree(kernel\_in\_gpu);

//out\_gpu不释放继续往下一个函数传值

////////////////////////////////////////////////////////////////////////////////////////

/////// conv step 2 我们保留了之前的 gpu\_out指针 //////////////

////////////////////////////////////////////////////////////////////////////////////////

int conv\_setp2\_out\_size = small\_img.rows\*small\_img.cols \* 10;

float \* conv\_step2\_out\_cpu, \*conv\_step2\_out\_gpu;

conv\_step2\_out\_cpu = (float\*)malloc(conv\_setp2\_out\_size \* sizeof(float));

cudaMalloc((void\*\*)&conv\_step2\_out\_gpu, sizeof(float)\*conv\_setp2\_out\_size);

dim3 gridDim(2,2,10);

dim3 blockDim(32,32);

conv\_step2\_new<<<gridDim , blockDim>>>(conv\_step2\_out\_gpu, out\_gpu, 3, 64);

cudaMemcpy(conv\_step2\_out\_cpu, conv\_step2\_out\_gpu, sizeof(float)\*conv\_setp2\_out\_size, cudaMemcpyDeviceToHost);

printf("---------------------------------------\n");

printf("-------输出conv\_step2\_cpu结果-----------\n");

printf("---------------------------------------\n");

for (int num = 0; num < 10; num++) {

for (int layer = 0; layer < 1; layer++) {

for (int i = 0; i < small\_img.rows; i++) {

for (int j = 0; j < small\_img.cols; j++) {

printf("%f \t", feature\_out\_cpu[num \* small\_img.rows\* small\_img.cols \* 1

+ layer \* small\_img.rows\* small\_img.cols

+ i \* small\_img.cols + j]);

}

}

printf(" \n 第%d个层卷积层的结果 \n ", num);

}

}

//在这里可以释放上一步的输出的gpu指针

////////////////////////////////////////////////////////////////////

//////////////////// 池化 操作 conv\_step2\_out\_gpu ////////////////////////////////

////////////////////////////////////////////////////////////////////

//池化操作没有 权重矩阵的输入

int pool\_out\_size = 8 \* 8 \* 10;

float \* pool\_out\_cpu, \*pool\_out\_gpu;

//开辟内存空间

cudaMalloc((void\*\*)&pool\_out\_gpu, sizeof(float)\*pool\_out\_size);

pool\_out\_cpu = (float\*)malloc(pool\_out\_size \* sizeof(float));

//kernel函数调用

dim3 gridDim\_pool(1, 1, 10);

dim3 gridBlock\_pool(8, 8);

pool\_new<<<gridDim\_pool, gridBlock\_pool >>>(pool\_out\_gpu, conv\_step2\_out\_gpu, 64, 8);

//结果拷回cpu

cudaMemcpy(pool\_out\_cpu, pool\_out\_gpu, sizeof(float)\*pool\_out\_size, cudaMemcpyDeviceToHost);

for (int num = 0; num < 10; num++) {

for (int layer = 0; layer < 1; layer++) {

for (int i = 0; i < 8; i++) {

for (int j = 0; j < 8; j++) {

printf("%f \t", pool\_out\_cpu[num \* 64 \* 1

+ i \* 8 + j]);

}

}

printf(" \n 第%d个层的结果 \n ", num);

}

}

printf(" \n 池化操作运行成功！！！NB \n ");

// 释放可以释放掉的指针

cudaFree(conv\_step2\_out\_gpu);

////////////////////////////////////////////////////////////////////////

///////////// 全连接层(矩阵运算) /////////////////////////

////////////////////////////////////////////////////////////////////////

//定义开辟数据空间的大小和一些宏变量

int weight\_size = 8 \* 8 \* 10 \* 10;

int fc\_out\_size = 10;

float \* fc\_out\_cpu, \*fc\_out\_gpu,\* weight\_in\_cpu,\*weight\_in\_gpu;

//开辟对应的空间

fc\_out\_cpu = (float \*)malloc(fc\_out\_size \* sizeof(float));

weight\_in\_cpu = (float \*)malloc(weight\_size \* sizeof(float));

cudaMalloc((void\*\*)&fc\_out\_gpu, sizeof(float)\*fc\_out\_size);

cudaMalloc((void\*\*)&weight\_in\_gpu, sizeof(float)\*weight\_size);

//给权重数组赋初值 一定要在 mencopy之前！

for (int raw = 0; raw < 10; raw++) {

for (int i = 0; i < 8 \* 8 \* 10; i++)

{

weight\_in\_cpu[raw \* 8 \* 8 \* 10 + i] = raw \*0.1;

}

}

for (int raw = 0; raw < 10; raw++) {

for (int i = 0; i < 8 \* 8 \* 10; i++)

{

printf(" %f \t",weight\_in\_cpu[raw \* 8 \* 8 \* 10 + i]);

}

printf("\n");

printf("#########################################");

printf("########### 一个数据 ############\n\n");

}

cudaMemcpy(weight\_in\_gpu, weight\_in\_cpu, sizeof(float)\*weight\_size, cudaMemcpyHostToDevice);

//设计传入 线程结构，调用kernel函数

dim3 dimGrid\_fc(10);

dim3 dimBlock\_fc(1);

FC\_SharedMem <<<dimGrid\_fc, dimBlock\_fc>>> (pool\_out\_gpu, weight\_in\_gpu, fc\_out\_gpu, 8 \* 8 \* 10);

//结果拷贝回cpu

cudaMemcpy(fc\_out\_cpu, fc\_out\_gpu, sizeof(float)\*fc\_out\_size, cudaMemcpyDeviceToHost);

int best\_index=-1;

float temp = 0.0;

for (int i = 0; i < fc\_out\_size; i++)

{

printf("预测第 %d 个类别的 得分是 %f \n", i, fc\_out\_cpu[i]);

if (fc\_out\_cpu[i] > temp) {

temp = fc\_out\_cpu[i];

best\_index = i;

}

}

printf("\n最后预测的是第%d类!!!!\n", best\_index);

cudaFree(weight\_in\_gpu);

cudaFree(fc\_out\_gpu);

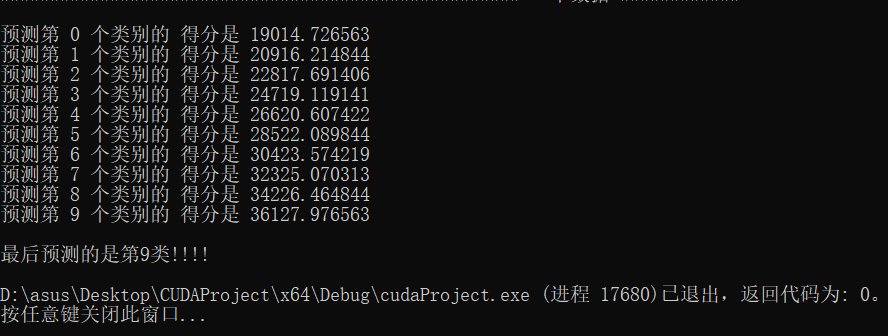
free(weight\_in\_cpu);

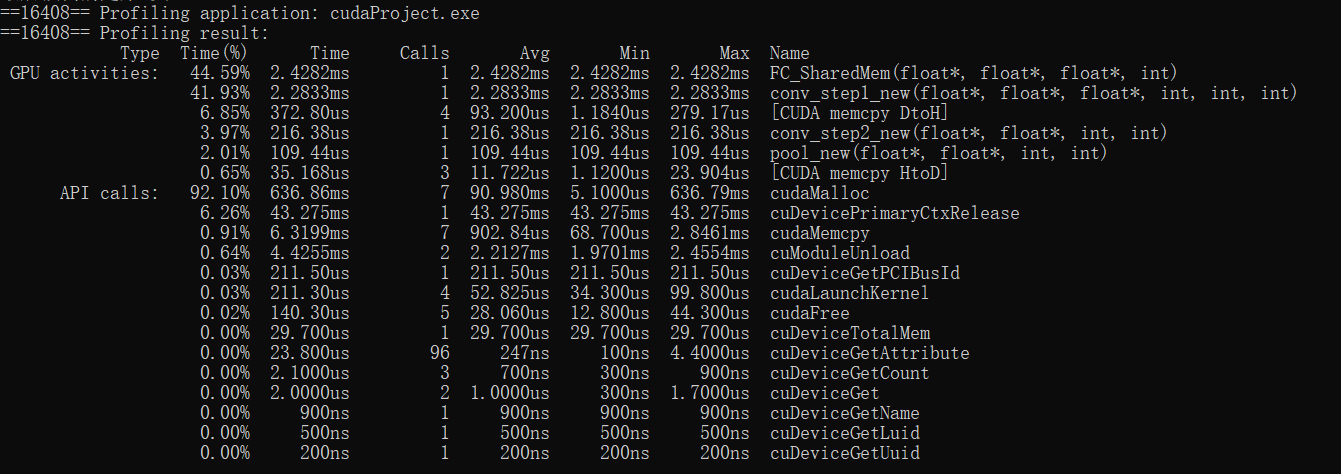
free(fc\_out\_cpu);

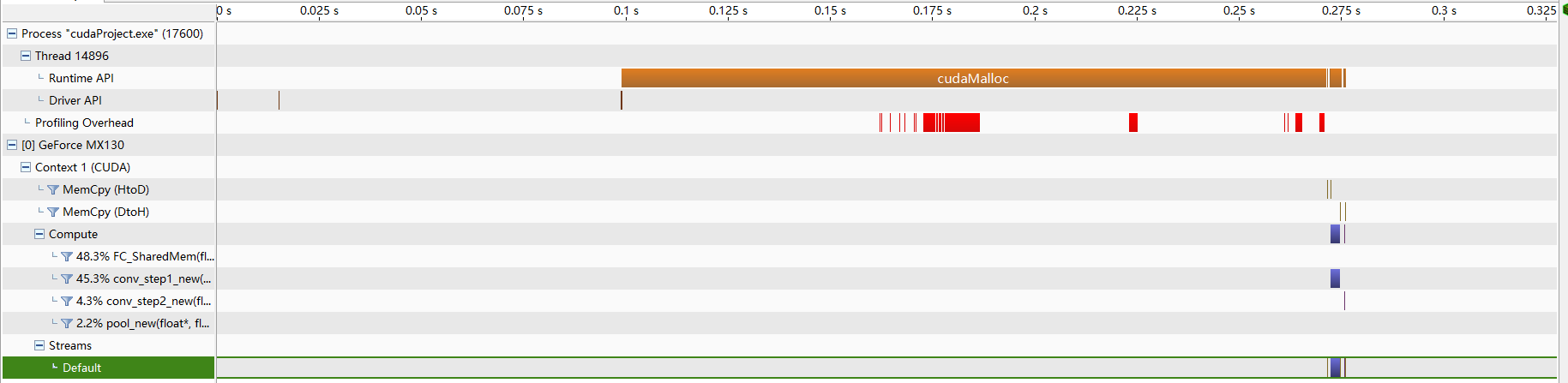
}

## 3）Result display

**We assume there are 10 categories in total and the programme output as below:**



**This shows the gpu time with the help of nvprof:**

**This shows gpu time visiually under the help of nvvp**

# 5 Personal contribution

**贴一段代码即可**

# 6 Experience sharing

# 7 others

**Main** **contributor** :**Seventeen Xu(徐世琦)**

**Other** **contributors: 李若彤 李瑞格 郭旭红**

**This project is open now but follows GPL 2.0 license**

**The entire project has been upload to github repo ,welcome to visit and give a star Thanks.**

<https://github.com/seventeenxsq/CUDA_CNN_Forward.git>

# 8 References

.《Programming\_Massively\_Parallel\_Processors》

《CUDA\_C\_Programming\_Guide》

https://forums.developer.nvidia.com/