**实验12 实验步骤3 softmax分类器**

**姓名：**徐陈天野 **学号：**2019141460119

**题目：**



**注：**经过分析代码与多次尝试后发现，需要在数据预处理时将data\_train和data\_test的所有值通过除以255都限制到0 ~ 1之间，否则后面有些值会因计算一直增大超出float型表示范围。而且，将data\_train和data\_test的所有值通过除以255都限制到0 ~ 1之间对算法的收敛与预测效果并无影响。

    // 把data\_train的所有值限制到 0 ~ 1 之间，防止后面部分值超出float型表示范围

    for(int r = 0; r < number\_of\_samples\_train; r++)

    {

        for(int c = 0; c < rows\*columns; c++)

        {

            data\_train[rows\*columns \* r + c] /=225;

        }

    }

    // 把data\_test的所有值限制到 0 ~ 1 之间，防止后面部分值超出float型表示范围

    for(int r = 0; r < number\_of\_samples\_test; r++)

    {

        for(int c = 0; c < rows\*columns; c++)

        {

            data\_test[rows\*columns \* r + c] /=225;

        }

    }

**任务1：**试完成“softmax.c”中的Todo部分，实现CPU版本的softmax分类器的训练和测试。

**源代码：**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include"mnist\_helper.h"

#include"time\_helper.h"

#include"utils.h"

/\*

\* Compile command \*

\* gcc -o softmax softmax.c -lm \*

\* OR \*

\* nvcc -o softmax softmax.c --run \*

\*/

int main(int argc, char \*argv[]) {

    const char \* train\_images\_file = "train-images-idx3-ubyte";

    const char \* train\_labels\_file = "train-labels-idx1-ubyte";

    const char \* test\_images\_file = "t10k-images-idx3-ubyte";

    const char \* test\_labels\_file = "t10k-labels-idx1-ubyte";

    float \*data\_train, \*data\_test;

    int \*labels\_train, \*labels\_test;

    int number\_of\_samples\_train, number\_of\_samples\_test, rows, columns;

    /\*

    \* \* \* \* Load training data  \* \* \* \*

    \* data\_train: float, 60000x784, each row represents a data sample \*

    \* labels\_train: int, 60000, data labels, [1,2,3,4,5,...] \*

    \* number\_of\_samples\_train: 60000 \*

    \* rows: 28, number of pixel rows in an image; columns: 28, number of pixel columns in an image \*

    \*/

    get\_dataset(train\_images\_file, train\_labels\_file, &data\_train, &labels\_train, &number\_of\_samples\_train, &rows, &columns);

    scale\_pixels(data\_train, number\_of\_samples\_train \* rows \* columns);

    printf("Training dataset: [%d %d %d] \n\n", number\_of\_samples\_train, rows, columns);

    /\*

    \* \* \* \* Load test data  \* \* \* \*

    \* data\_test: float, 10000x784, each row represents a data sample \*

    \* labels\_test: int, 10000, data labels, [1,2,3,4,5,...] \*

    \* number\_of\_samples\_test: 10000 \*

    \* rows: 28, number of pixel rows in an image; columns: 28, number of pixel columns in an image \*

    \*/

    get\_dataset(test\_images\_file, test\_labels\_file, &data\_test, &labels\_test, &number\_of\_samples\_test, &rows, &columns);

    scale\_pixels(data\_test, number\_of\_samples\_test \* rows \* columns);

    printf("\n Test dataset: [%d %d %d] \n", number\_of\_samples\_test, rows, columns);

    /\*

    \* Model initialization \*

    \* output = softmax(W\*input) \*

    \* W:10x784, input:784xn, output:10xn\*

    \*/

    int W\_rows = 10;

    int W\_columns = 784;

    float\* W = (float \*)malloc(W\_rows\*W\_columns\*sizeof(float));

    weight\_initialization(W, W\_rows, W\_columns);

    /\*

    \* Training data, activation and gradient buffers \*

    \*/

    float\* activations\_train = (float \*)malloc(W\_rows \* number\_of\_samples\_train \* sizeof(float));

    float\* data\_transposed\_train = (float \*)malloc(rows \* columns \* number\_of\_samples\_train \* sizeof(float));

    float \*delta = (float \*)malloc(W\_rows\*number\_of\_samples\_train\*sizeof(float));

    float \*W\_grad = (float \*)malloc(W\_rows\*W\_columns\*sizeof(float));

    /\*

    \* Test data and activation buffers \*

    \*/

    float\* activations\_test = (float \*)malloc(W\_rows \* number\_of\_samples\_test \* sizeof(float));

    float\* data\_transposed\_test = (float \*)malloc(rows \* columns \* number\_of\_samples\_test \* sizeof(float));

    /\*

    \* Data sample visualization \*

    \*/

    printf("label: %d\n", labels\_train[18]);

    for(int i=0; i<rows; i++)

    {

        for(int j=0; j<columns; j++)

        {

            printf("%s ", \*(data\_train+i\*columns+j + 18\*rows\*columns)>0?"#":" ");

        }

        printf("\n");

    }

    // 把data\_train的所有值限制到 0 ~ 1 之间，防止后面部分值超出float型表示范围

    for(int r = 0; r < number\_of\_samples\_train; r++)

    {

        for(int c = 0; c < rows\*columns; c++)

        {

            data\_train[rows\*columns \* r + c] /=225;

        }

    }

    // 把data\_test的所有值限制到 0 ~ 1 之间，防止后面部分值超出float型表示范围

    for(int r = 0; r < number\_of\_samples\_test; r++)

    {

        for(int c = 0; c < rows\*columns; c++)

        {

            data\_test[rows\*columns \* r + c] /=225;

        }

    }

    /\*

    \* data: [n,784], one image per row \*

    \* data\_transposed\_train | data\_transposed\_test: [784,n], one image per column\*

    \*/

    matrix\_transpose(data\_train, data\_transposed\_train, number\_of\_samples\_train, rows\*columns);

    matrix\_transpose(data\_test, data\_transposed\_test, number\_of\_samples\_test, rows\*columns);

    /\*

    \* Training loop \*

    \*/

    int epoch\_num = 1000;

    float learning\_rate = 0.5;

    float loss\_train, acc\_train, loss\_test, acc\_test;

    double time\_begin;

    for(int epoch=0; epoch<epoch\_num; epoch++)

    {

        time\_begin = cpuSecond();

        /\*

        \* Forward on training set \*

        \* data: [n,784], one image per row \*

        \* W:[10,784], data\_transposed\_train:[784,n], activations\_train: [10,n] \*

        \*/

        // Todo 1

        // activations\_train = W \* data\_transposed\_train

        memset(activations\_train, 0, W\_rows \* number\_of\_samples\_train \* sizeof(float));

        for(int r = 0; r < W\_rows; r++)

        {

            for(int c = 0; c < number\_of\_samples\_train; c++)

            {

                for(int i = 0; i < rows\*columns; i++)

                {

                    activations\_train[r \* number\_of\_samples\_train + c] += W[W\_columns \* r + i] \* data\_transposed\_train[number\_of\_samples\_train \* i + c];

                }

            }

        }

        /\*

        \* softmax normalization on activations \*

        \* activations: [10,n] \*

        \*/

        // Todo 2

        // softmax(activations\_train)

        for(int c = 0; c < number\_of\_samples\_train; c++)

        {

            float sum = 0;

            for(int r = 0; r < W\_rows; r++)

            {

                sum += exp(activations\_train[r \* number\_of\_samples\_train + c]);

            }

            for(int r = 0; r < W\_rows; r++)

            {

                activations\_train[r \* number\_of\_samples\_train + c] = exp(activations\_train[r \* number\_of\_samples\_train + c]) / sum;

            }

        }

        // Todo 3

        // loss on training set

        loss\_train = 0.;

        for(int i = 0; i < number\_of\_samples\_train; i++)

        {

            for(int k = 0; k < W\_rows; k++)

            {

                if(labels\_train[i] == k)

                    loss\_train += log(activations\_train[k \* number\_of\_samples\_train + i]);

            }

        }

        loss\_train /= (-number\_of\_samples\_train);

        // Todo 4

        // accuracy on training set

        int correct = 0;

        int res = -1;

        float max = 0;

        for(int i = 0; i < number\_of\_samples\_train; i++)

        {

            res = 0;

            max = activations\_train[i];

            for(int k = 0; k < W\_rows; k++)

            {

                if(activations\_train[k \* number\_of\_samples\_train + i] > max)

                {

                    max = activations\_train[k \* number\_of\_samples\_train + i];

                    res = k;

                }

            }

            if(res == labels\_train[i]) correct++;

        }

        acc\_train = correct / (float)number\_of\_samples\_train;

        // Todo 5

        // [Test] Forward on test set

        // activations\_test = W \* data\_transposed\_test

        memset(activations\_test, 0, W\_rows \* number\_of\_samples\_test \* sizeof(float));

        for(int r = 0; r < W\_rows; r++)

        {

            for(int c = 0; c < number\_of\_samples\_test; c++)

            {

                for(int i = 0; i < rows\*columns; i++)

                {

                    activations\_test[r \* number\_of\_samples\_test + c] += W[W\_columns \* r + i] \* data\_transposed\_test[number\_of\_samples\_test \* i + c];

                }

            }

        }

        // Todo 6

        // [Test] Softmax on activations\_test

        for(int c = 0; c < number\_of\_samples\_test; c++)

        {

            float sum = 0;

            for(int r = 0; r < W\_rows; r++)

            {

                sum += exp(activations\_test[r \* number\_of\_samples\_test + c]);

            }

            for(int r = 0; r < W\_rows; r++)

            {

                activations\_test[r \* number\_of\_samples\_test + c] = exp(activations\_test[r \* number\_of\_samples\_test + c]) / sum;

            }

        }

        // Todo 7

        // [Test] loss on test set

        loss\_test = 0.;

        for(int i = 0; i < number\_of\_samples\_test; i++)

        {

            for(int k = 0; k < W\_rows; k++)

            {

                if(labels\_test[i] == k)

                    loss\_test += log(activations\_test[k \* number\_of\_samples\_test + i]);

            }

        }

        loss\_test /= (-number\_of\_samples\_test);

        // Todo 8

        // [Test] accuracy on test set

        correct = 0;

        res = -1;

        max = 0;

        for(int i = 0; i < number\_of\_samples\_test; i++)

        {

            res = 0;

            max = activations\_test[i];

            for(int k = 0; k < W\_rows; k++)

            {

                if(activations\_test[k \* number\_of\_samples\_test + i] > max)

                {

                    max = activations\_test[k \* number\_of\_samples\_test + i];

                    res = k;

                }

            }

            if(res == labels\_test[i]) correct++;

        }

        acc\_test = correct / (float)number\_of\_samples\_test;

        // Reset gradients

        memset(delta, 0, W\_rows\*number\_of\_samples\_train\*sizeof(float));

        memset(W\_grad, 0, W\_rows\*W\_columns\*sizeof(float));

        // Todo 9

        // Compute delta

        // delta [10,n] = activations [10,n] - y [10,n]; (y[labels\_train\_set, :]=1)

        for(int c = 0; c < number\_of\_samples\_train; c++)

        {

            for(int r = 0; r < W\_rows; r++)

            {

                if(r == labels\_train[c])

                    delta[r \* number\_of\_samples\_train + c] = activations\_train[r \* number\_of\_samples\_train + c] - 1;

                else

                    delta[r \* number\_of\_samples\_train + c] = activations\_train[r \* number\_of\_samples\_train + c];

            }

        }

        // Todo 10

        // W\_grad [10,784] = delta [10,n] \* images\_train\_set [n,784]

        for(int r = 0; r < W\_rows; r++)

        {

            for(int c = 0; c < W\_columns; c++)

            {

                for(int i = 0; i < number\_of\_samples\_train; i++)

                {

                    W\_grad[r \* W\_columns + c] += delta[number\_of\_samples\_train \* r + i] \* data\_train[W\_columns \* i + c];

                }

            }

        }

        // Todo 11

        // Update, W = W - alpha \* W\_grad;

        for(int r = 0; r < W\_rows; r++)

        {

            for(int c = 0; c < W\_columns; c++)

            {

                W[r \* W\_columns + c] -= learning\_rate \* W\_grad[r \* W\_columns + c];

            }

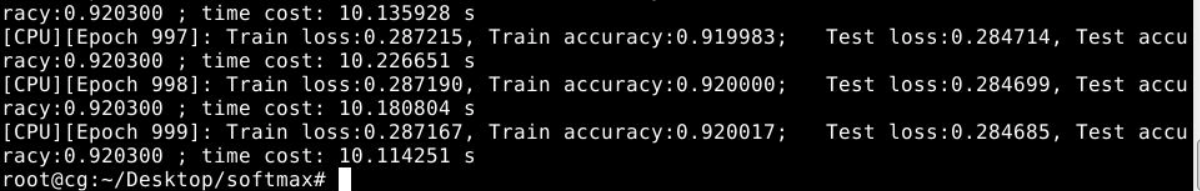
        }

        printf("[CPU][Epoch %d]: Train loss:%f, Train accuracy:%f;   Test loss:%f, Test accuracy:%f ; time cost: %lf s\n", epoch, loss\_train, acc\_train, loss\_test, acc\_test, cpuSecond()-time\_begin);

    }

    return 0;

}

**实验结果：**

由上图截图可见，CPU版本的softmax分类器训练和测试过程仅一个epoch就需要花费长达10s多的时间。完成这1000个epoch，总的训练过程花费了接近3个小时。

另外，可见softmax分类器分类效果较好，对MNIST手写体数字识别问题准确率达到了训练集上92.00%、测试集上92.03%的准确率。训练集上损失函数值下降到0.287，测试集上损失函数值下降到0.285，收敛表现也很好。

**任务2：**使用《GPU平台编程设计》中的方法，对该softmax分类器的训练和推理过程进行加速。

**注：**以下代码中将使用《GPU平台编程设计》中的方法对原方法进行改进的部分都包装成函数，置于main函数前，方便代码复用。

**源代码：**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include "mnist\_helper.h"

#include "time\_helper.h"

#include "utils.h"

#include "error\_check.h"

#define SDATA\_SIZE\_SOFTMAX\_R 10

#define SDATA\_SIZE\_SOFTMAX\_C 100

// 矩阵乘法加速

\_\_global\_\_ void kernel\_matrix\_multiply\_block(float \*M, float \*N, float \*P, int M\_rows, int M\_cols, int N\_rows, int N\_cols)

{

    int idx\_x = threadIdx.x;

    int idx\_y = threadIdx.y;

    int row = blockIdx.y\*16 + idx\_y;

    int col = blockIdx.x\*16 + idx\_x;

    float Pvalue = 0.0;

    \_\_shared\_\_ float M\_shared[16][16];

    \_\_shared\_\_ float N\_shared[16][16];

    for(int i=0; i<(M\_cols+1)/16; i++){

        if(i\*16 + idx\_x < M\_cols && row < M\_rows){

            M\_shared[idx\_y][idx\_x] = M[row\*M\_cols + i\*16 + idx\_x];

        }else{

            M\_shared[idx\_y][idx\_x] = 0.0;

        }

        if(i\*16 + idx\_y < N\_rows && col < N\_cols){

            N\_shared[idx\_y][idx\_x] = N[(i\*16 + idx\_y)\*N\_cols + col];

        }else{

            N\_shared[idx\_y][idx\_x] =0.0;

        }

        \_\_syncthreads();

        for(int k=0 ; k < 16 ; k++){

            Pvalue += M\_shared[idx\_y][k] \* N\_shared[k][idx\_x];

        }

        \_\_syncthreads();

    }

    if(row < M\_rows && col < N\_cols){

        P[row\*N\_cols + col] = Pvalue;

    }

}

void gpu\_matrix\_multiply(float \*M, float \*N, float \*P, int M\_rows, int M\_cols, int N\_rows, int N\_cols)

{

    float \*d\_M, \*d\_N, \*d\_P;

    CHECK(cudaMalloc((void \*\*)&d\_M, M\_rows\*M\_cols\*sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_N, N\_rows\*N\_cols\*sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_P, M\_rows\*N\_cols\*sizeof(float)));

    CHECK(cudaMemcpy(d\_M, M, M\_rows\*M\_cols\*sizeof(float), cudaMemcpyHostToDevice));

    CHECK(cudaMemcpy(d\_N, N, N\_rows\*N\_cols\*sizeof(float), cudaMemcpyHostToDevice));

    CHECK(cudaMemset(d\_P, 0, M\_rows\*N\_cols\*sizeof(float)));

    dim3 block(16, 16);

    dim3 grid((N\_cols-1)/block.y+1, (M\_rows-1)/block.x+1);

    kernel\_matrix\_multiply\_block<<<grid, block>>>(d\_M, d\_N, d\_P, M\_rows, M\_cols, N\_rows, N\_cols);

    CHECK(cudaDeviceSynchronize());

    CHECK(cudaMemcpy(P, d\_P, M\_rows\*N\_cols\*sizeof(float), cudaMemcpyDeviceToHost));

    CHECK(cudaFree(d\_M));

    CHECK(cudaFree(d\_N));

    CHECK(cudaFree(d\_P));

}

// 矩阵带系数的加法加速 （S = coefficientM \* M + coefficientN \* N）

\_\_global\_\_ void kernel\_matrix\_add\_with\_coefficient(float \*M, float \*N, float \*S, float coefficientM, float coefficientN)

{

    int idx = blockIdx.x \* blockDim.x + threadIdx.x;

    S[idx] = coefficientM \* M[idx] + coefficientN \* N[idx];

}

void gpu\_matrix\_add\_with\_coefficient(float \*M, float \*N, float \*S, float coefficientM, float coefficientN, int rows, int cols)

{

    float \*d\_M, \*d\_N, \*d\_S;

    CHECK(cudaMalloc((void \*\*)&d\_M, rows\*cols\*sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_N, rows\*cols\*sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_S, rows\*cols\*sizeof(float)));

    CHECK(cudaMemcpy(d\_M, M, rows\*cols\*sizeof(float), cudaMemcpyHostToDevice));

    CHECK(cudaMemcpy(d\_N, N, rows\*cols\*sizeof(float), cudaMemcpyHostToDevice));

    kernel\_matrix\_add\_with\_coefficient<<<(rows\*cols-1)/1024+1, 1024>>>(d\_M, d\_N, d\_S, coefficientM, coefficientN);

    CHECK(cudaDeviceSynchronize());

    CHECK(cudaMemcpy(S, d\_S, rows\*cols\*sizeof(float), cudaMemcpyDeviceToHost));

    CHECK(cudaFree(d\_M));

    CHECK(cudaFree(d\_N));

    CHECK(cudaFree(d\_S));

}

// softmax加速 （一次读入一个10\*100的小块到shared memory中，减少求每一列的exp函数值的和时的stride式访问）

\_\_global\_\_ void kernel\_softmax(float \*activations, int number\_of\_samples)

{

    \_\_shared\_\_ float sdata[SDATA\_SIZE\_SOFTMAX\_R][SDATA\_SIZE\_SOFTMAX\_C];

    \_\_shared\_\_ float sums[SDATA\_SIZE\_SOFTMAX\_C];

    sdata[threadIdx.y][threadIdx.x] = activations[threadIdx.y \* number\_of\_samples + blockIdx.x \* SDATA\_SIZE\_SOFTMAX\_C + threadIdx.x];

    \_\_syncthreads();

    if(threadIdx.y == 0){

        float temp = 0;

        for(int i=0; i < SDATA\_SIZE\_SOFTMAX\_R; i++){

            temp += exp(sdata[i][threadIdx.x]);

        }

        sums[threadIdx.x] = temp;

    }

    \_\_syncthreads();

    activations[threadIdx.y \* number\_of\_samples + blockIdx.x \* SDATA\_SIZE\_SOFTMAX\_C + threadIdx.x] = exp(sdata[threadIdx.y][threadIdx.x]) / sums[threadIdx.x];

}

void gpu\_softmax(float \*activations, int number\_of\_samples)

{

    float \*d\_activations = NULL;

    CHECK(cudaMalloc((void \*\*)&d\_activations, 10 \* number\_of\_samples \* sizeof(float)));

    CHECK(cudaMemcpy(d\_activations, activations, 10 \* number\_of\_samples \* sizeof(float), cudaMemcpyHostToDevice));

    dim3 block(100, 10);

    dim3 grid((number\_of\_samples-1)/100+1);

    kernel\_softmax<<<grid, block>>>(d\_activations, number\_of\_samples);

    CHECK(cudaMemcpy(activations, d\_activations, 10 \* number\_of\_samples \* sizeof(float), cudaMemcpyDeviceToHost));

    CHECK(cudaFree(d\_activations));

}

// delta计算加速

\_\_global\_\_ void kernel\_delta(float \*delta, float \*activations, int \*labels, int number\_of\_samples)

{

    int r = threadIdx.x;

    int c = blockIdx.x;

    if(r == labels[c])

        delta[r \* number\_of\_samples + c] = activations[r \* number\_of\_samples + c] - 1;

    else

        delta[r \* number\_of\_samples + c] = activations[r \* number\_of\_samples + c];

}

void gpu\_compute\_delta(float \*delta, float \*activations, int \*labels, int number\_of\_samples)

{

    float \*d\_activations = NULL;

    float \*d\_delta = NULL;

    int \*d\_labels = NULL;

    CHECK(cudaMalloc((void \*\*)&d\_activations, 10 \* number\_of\_samples \* sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_delta, 10 \* number\_of\_samples \* sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_labels, number\_of\_samples \* sizeof(int)));

    CHECK(cudaMemcpy(d\_activations, activations, 10 \* number\_of\_samples \* sizeof(float), cudaMemcpyHostToDevice));

    CHECK(cudaMemcpy(d\_labels, labels, number\_of\_samples \* sizeof(int), cudaMemcpyHostToDevice));

    kernel\_delta<<<number\_of\_samples, 10>>>(d\_delta, d\_activations, d\_labels, number\_of\_samples);

    CHECK(cudaMemcpy(delta, d\_delta, 10 \* number\_of\_samples \* sizeof(float), cudaMemcpyDeviceToHost));

    CHECK(cudaFree(d\_activations));

    CHECK(cudaFree(d\_delta));

    CHECK(cudaFree(d\_labels));

}

// 计算accuracy（利用原子操作atomicAdd）

\_\_global\_\_ void kernel\_accuracy(float \*activations, int \*labels, int number\_of\_samples, int \*correct)

{

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    int res = 0;

    float max = activations[i];

    for(int k = 0; k < 10; k++){

        if(activations[k \* number\_of\_samples + i] > max)

        {

            max = activations[k \* number\_of\_samples + i];

            res = k;

        }

    }

    if(res == labels[i]) atomicAdd(correct, 1);

}

float gpu\_compute\_accuracy(float \*activations, int \*labels, int number\_of\_samples)

{

    float \*d\_activations = NULL;

    int \*d\_labels = NULL;

    int \*correct = NULL;

    int \*d\_correct = NULL;

    int co = 0;

    correct = (int \*)malloc(sizeof(int));

    CHECK(cudaMalloc((void \*\*)&d\_correct, sizeof(int)));

    CHECK(cudaMalloc((void \*\*)&d\_activations, 10 \* number\_of\_samples \* sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_labels, number\_of\_samples \* sizeof(int)));

    CHECK(cudaMemcpy(d\_activations, activations, 10 \* number\_of\_samples \* sizeof(float), cudaMemcpyHostToDevice));

    CHECK(cudaMemcpy(d\_labels, labels, number\_of\_samples \* sizeof(int), cudaMemcpyHostToDevice));

    CHECK(cudaMemset(d\_correct, 0, sizeof(int)));

    kernel\_accuracy<<<(number\_of\_samples-1)/1024+1, 1024>>>(d\_activations, d\_labels, number\_of\_samples, d\_correct);

    CHECK(cudaMemcpy(correct, d\_correct, sizeof(int), cudaMemcpyDeviceToHost));

    co = \*correct;

    CHECK(cudaFree(d\_activations));

    CHECK(cudaFree(d\_labels));

    CHECK(cudaFree(d\_correct));

    free(correct);

    return co / (float)number\_of\_samples;

}

// 计算loss（利用原子操作atomicAdd）

\_\_global\_\_ void kernel\_loss(float \*activations, int \*labels, int number\_of\_samples, float \*loss)

{

    int i = blockIdx.x;

    int k = threadIdx.x;

    if(labels[i] == k)

        atomicAdd(loss, log(activations[k \* number\_of\_samples + i]));

}

float gpu\_compute\_loss(float \*activations, int \*labels, int number\_of\_samples)

{

    float \*d\_activations = NULL;

    int \*d\_labels = NULL;

    float \*loss = NULL;

    float \*d\_loss = NULL;

    float res = 0;

    loss = (float \*)malloc(sizeof(float));

    CHECK(cudaMalloc((void \*\*)&d\_activations, 10 \* number\_of\_samples \* sizeof(float)));

    CHECK(cudaMalloc((void \*\*)&d\_labels, number\_of\_samples \* sizeof(int)));

    CHECK(cudaMalloc((void \*\*)&d\_loss, sizeof(float)));

    CHECK(cudaMemset(d\_loss, 0, sizeof(float)));

    CHECK(cudaMemcpy(d\_activations, activations, 10 \* number\_of\_samples \* sizeof(float), cudaMemcpyHostToDevice));

    CHECK(cudaMemcpy(d\_labels, labels, number\_of\_samples \* sizeof(int), cudaMemcpyHostToDevice));

    kernel\_loss<<<number\_of\_samples, 10>>>(d\_activations, d\_labels, number\_of\_samples, d\_loss);

    CHECK(cudaMemcpy(loss, d\_loss, sizeof(float), cudaMemcpyDeviceToHost));

    res = \*loss / (- number\_of\_samples);

    CHECK(cudaFree(d\_activations));

    CHECK(cudaFree(d\_labels));

    CHECK(cudaFree(d\_loss));

    free(loss);

    return res;

}

int main(int argc, char \*argv[]) {

    const char \* train\_images\_file = "train-images-idx3-ubyte";

    const char \* train\_labels\_file = "train-labels-idx1-ubyte";

    const char \* test\_images\_file = "t10k-images-idx3-ubyte";

    const char \* test\_labels\_file = "t10k-labels-idx1-ubyte";

    float \*data\_train, \*data\_test;

    int \*labels\_train, \*labels\_test;

    int number\_of\_samples\_train, number\_of\_samples\_test, rows, columns;

    /\*

    \* \* \* \* Load training data  \* \* \* \*

    \* data\_train: float, 60000x784, each row represents a data sample \*

    \* labels\_train: int, 60000, data labels, [1,2,3,4,5,...] \*

    \* number\_of\_samples\_train: 60000 \*

    \* rows: 28, number of pixel rows in an image; columns: 28, number of pixel columns in an image \*

    \*/

    get\_dataset(train\_images\_file, train\_labels\_file, &data\_train, &labels\_train, &number\_of\_samples\_train, &rows, &columns);

    scale\_pixels(data\_train, number\_of\_samples\_train \* rows \* columns);

    printf("Training dataset: [%d %d %d] \n\n", number\_of\_samples\_train, rows, columns);

    /\*

    \* \* \* \* Load test data  \* \* \* \*

    \* data\_test: float, 10000x784, each row represents a data sample \*

    \* labels\_test: int, 10000, data labels, [1,2,3,4,5,...] \*

    \* number\_of\_samples\_test: 10000 \*

    \* rows: 28, number of pixel rows in an image; columns: 28, number of pixel columns in an image \*

    \*/

    get\_dataset(test\_images\_file, test\_labels\_file, &data\_test, &labels\_test, &number\_of\_samples\_test, &rows, &columns);

    scale\_pixels(data\_test, number\_of\_samples\_test \* rows \* columns);

    printf("\n Test dataset: [%d %d %d] \n", number\_of\_samples\_test, rows, columns);

    /\*

    \* Model initialization \*

    \* output = softmax(W\*input) \*

    \* W:10x784, input:784xn, output:10xn\*

    \*/

    int W\_rows = 10;

    int W\_columns = 784;

    float\* W = (float \*)malloc(W\_rows\*W\_columns\*sizeof(float));

    weight\_initialization(W, W\_rows, W\_columns);

    /\*

    \* Training data, activation and gradient buffers \*

    \*/

    float\* activations\_train = (float \*)malloc(W\_rows \* number\_of\_samples\_train \* sizeof(float));

    float\* data\_transposed\_train = (float \*)malloc(rows \* columns \* number\_of\_samples\_train \* sizeof(float));

    float \*delta = (float \*)malloc(W\_rows\*number\_of\_samples\_train\*sizeof(float));

    float \*W\_grad = (float \*)malloc(W\_rows\*W\_columns\*sizeof(float));

    /\*

    \* Test data and activation buffers \*

    \*/

    float\* activations\_test = (float \*)malloc(W\_rows \* number\_of\_samples\_test \* sizeof(float));

    float\* data\_transposed\_test = (float \*)malloc(rows \* columns \* number\_of\_samples\_test \* sizeof(float));

    /\*

    \* Data sample visualization \*

    \*/

    printf("label: %d\n", labels\_train[18]);

    for(int i=0; i<rows; i++)

    {

        for(int j=0; j<columns; j++)

        {

            printf("%s ", \*(data\_train+i\*columns+j + 18\*rows\*columns)>0?"#":" ");

        }

        printf("\n");

    }

    // 把data\_train的所有值限制到 0 ~ 1 之间，防止后面部分值超出float型表示范围

    for(int r = 0; r < number\_of\_samples\_train; r++)

    {

        for(int c = 0; c < rows\*columns; c++)

        {

            data\_train[rows\*columns \* r + c] /=225;

        }

    }

    // 把data\_test的所有值限制到 0 ~ 1 之间，防止后面部分值超出float型表示范围

    for(int r = 0; r < number\_of\_samples\_test; r++)

    {

        for(int c = 0; c < rows\*columns; c++)

        {

            data\_test[rows\*columns \* r + c] /=225;

        }

    }

    /\*

    \* data: [n,784], one image per row \*

    \* data\_transposed\_train | data\_transposed\_test: [784,n], one image per column\*

    \*/

    matrix\_transpose(data\_train, data\_transposed\_train, number\_of\_samples\_train, rows\*columns);

    matrix\_transpose(data\_test, data\_transposed\_test, number\_of\_samples\_test, rows\*columns);

    /\*

    \* Training loop \*

    \*/

    int epoch\_num = 1000;

    float learning\_rate = 0.5;

    float loss\_train, acc\_train, loss\_test, acc\_test;

    double time\_begin;

    for(int epoch=0; epoch<epoch\_num; epoch++)

    {

        time\_begin = cpuSecond();

        /\*

        \* Forward on training set \*

        \* data: [n,784], one image per row \*

        \* W:[10,784], data\_transposed\_train:[784,n], activations\_train: [10,n] \*

        \*/

        // Todo 1

        // activations\_train = W \* data\_transposed\_train

        gpu\_matrix\_multiply(W, data\_transposed\_train, activations\_train, W\_rows, W\_columns, W\_columns, number\_of\_samples\_train);

        /\*

        \* softmax normalization on activations \*

        \* activations: [10,n] \*

        \*/

        // Todo 2

        // softmax(activations\_train)

        gpu\_softmax(activations\_train, number\_of\_samples\_train);

        // Todo 3

        // loss on training set

        loss\_train = gpu\_compute\_loss(activations\_train, labels\_train, number\_of\_samples\_train);

        // Todo 4

        // accuracy on training set

        acc\_train = gpu\_compute\_accuracy(activations\_train, labels\_train, number\_of\_samples\_train);

        // Todo 5

        // [Test] Forward on test set

        // activations\_test = W \* data\_transposed\_test

        gpu\_matrix\_multiply(W, data\_transposed\_test, activations\_test, W\_rows, W\_columns, W\_columns, number\_of\_samples\_test);

        // Todo 6

        // [Test] Softmax on activations\_test

        gpu\_softmax(activations\_test, number\_of\_samples\_test);

        // Todo 7

        // [Test] loss on test set

        loss\_test = gpu\_compute\_loss(activations\_test, labels\_test, number\_of\_samples\_test);

        // Todo 8

        // [Test] accuracy on test set

        acc\_test = gpu\_compute\_accuracy(activations\_test, labels\_test, number\_of\_samples\_test);

        // Reset gradients

        memset(delta, 0, W\_rows\*number\_of\_samples\_train\*sizeof(float));

        memset(W\_grad, 0, W\_rows\*W\_columns\*sizeof(float));

        // Todo 9

        // Compute delta

        // delta [10,n] = activations [10,n] - y [10,n]; (y[labels\_train\_set, :]=1)

        gpu\_compute\_delta(delta, activations\_train, labels\_train, number\_of\_samples\_train);

        // Todo 10

        // W\_grad [10,784] = delta [10,n] \* images\_train\_set [n,784]

        gpu\_matrix\_multiply(delta, data\_train, W\_grad, W\_rows, number\_of\_samples\_train, number\_of\_samples\_train, W\_columns);

        // Todo 11

        // Update, W = W - alpha \* W\_grad;

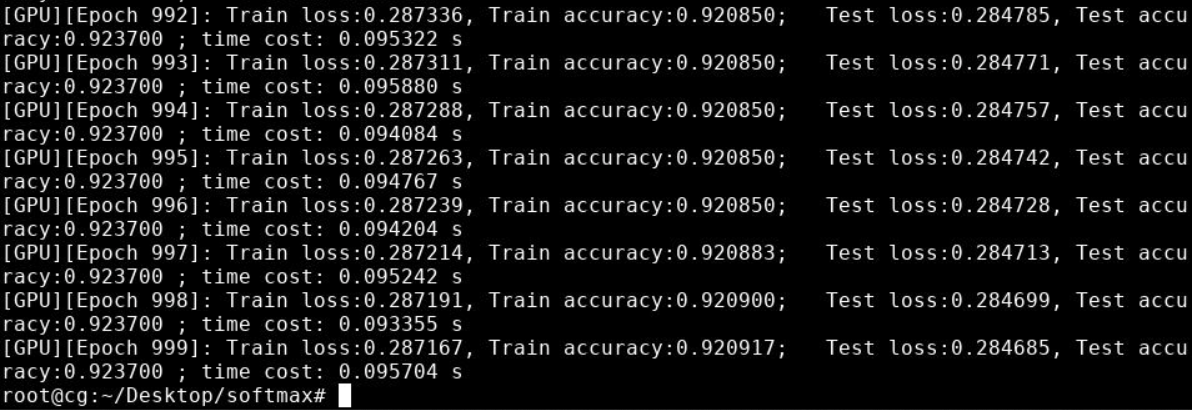
        gpu\_matrix\_add\_with\_coefficient(W, W\_grad, W, 1.0, - learning\_rate, W\_rows, W\_columns);

        printf("[GPU][Epoch %d]: Train loss:%f, Train accuracy:%f;   Test loss:%f, Test accuracy:%f ; time cost: %lf s\n", epoch, loss\_train, acc\_train, loss\_test, acc\_test, cpuSecond()-time\_begin);

    }

    return 0;

}

**实验结果：**

由上图结果可见，在使用了《GPU平台编程设计》中的方法对原来的方法进行改进加速后（具体为：矩阵乘法加速、矩阵带系数的加法加速、softmax加速、delta计算加速、利用原子操作atomicAdd计算accuracy、利用原子操作atomicAdd计算loss），一个epoch耗费时间明显大大减少，只需要0.094秒左右，不到两分钟就完成了1000个epoch，完成了训练和推理。

另外，可见softmax分类器分类效果较好，对MNIST手写体数字识别问题准确率达到了训练集上92.09%、测试集上92.37%的准确率。训练集上损失函数值下降到0.287，测试集上损失函数值下降到0.285，收敛表现也很好。

**注：**本代码数据预处理部分的矩阵转置也可以使用之前实验中的消除bank conflicts的使用shared memory的连续存储访问的矩阵转置来加速。代码如下：

\_\_global\_\_ void kernel\_transpose\_per\_element\_tiled\_no\_bank\_conflicts(DTYPE \*input, DTYPE \*output, int num\_rows, int num\_cols)

{

    int col\_idx = blockIdx.x \* blockDim.x + threadIdx.x;

    int row\_idx = blockIdx.y \* blockDim.y + threadIdx.y;

    int input\_width = num\_cols;

    int output\_width = num\_rows;

    \_\_shared\_\_ DTYPE tile[BLOCK\_SIZE+1][BLOCK\_SIZE];

    int thy = threadIdx.y + ( (threadIdx.x + threadIdx.y < BLOCK\_SIZE) ? 0 : 1 );

    int thx = (threadIdx.x + threadIdx.y) % BLOCK\_SIZE;

    tile[thy][thx] = input[row\_idx\*input\_width+col\_idx];

    \_\_syncthreads();

    // compute the target block index

    int blockIdx\_x\_target = blockIdx.y;

    int blockIdx\_y\_target = blockIdx.x;

    int col\_idx\_target = blockIdx\_x\_target \* blockDim.y + threadIdx.x;

    int row\_idx\_target = blockIdx\_y\_target \* blockDim.x + threadIdx.y;

    int thx\_ = threadIdx.x + ( (threadIdx.x + threadIdx.y < BLOCK\_SIZE) ? 0 : 1 );

    int thy\_ = (threadIdx.x + threadIdx.y) % BLOCK\_SIZE;

    output[row\_idx\_target\*output\_width+col\_idx\_target] = tile[thx\_][thy\_];

}

由于此非题目要求，不在训练和推理过程中，而在数据预处理过程中，不需要多次反复执行，只需执行两次，对总的训练过程时间影响很小。且此方法在实验9中已经实现，此处只需稍加修改即可。故不再赘述。