

基于神经网络的手写数字识别的 CUDA 优化

姓名：李亚杰

学号：SA18219058

一、实验平台

硬件平台：CPU：i7-8750h 2.2GHz，RAM：8GB；GPU：GTX1050TI，RAM：4GB

软件平台：win10 家庭版+Visual Studio2015+Cuda8.0

二、实验目的

用 CUDA 实现 GPU 版本的手写数字识别的神经网络的训练，并与 CPU 训练的时间进行对比，加快神经网络训练的速度，提高开发效率。

三、实验原理及内容

实验所使用的数据集为 MNIST 数据集，数据集中每张图片是大小为 28*28 的灰度图，数字图像从 0 到 9，数据集内容如下表 1 所示，每个图像的标签采用 one-hot 的编码形式。

表 1 数据集

数据集	大小
训练集	60000
训练集标签	60000
测试集	10000
测试集标签	10000

神经网络模型：输入层 784 个节点，隐藏层 200 个节点，输出层 10 个节点。如下图 1.1 所示：

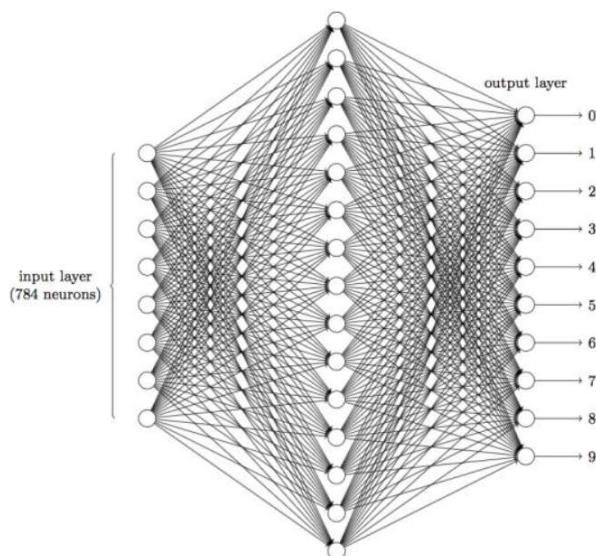


图 1.1：三层神经网络

设神经网络的输入层为 $x = [x_1, x_2, \dots, x_{784}]^T$ ，输出层的输出为 $y = [y_1, y_2, \dots, y_{10}]^T$ ，输入

层与隐含层的链接权值为 $W_{ih} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,784} \\ \vdots & \ddots & \vdots \\ w_{200,1} & \cdots & w_{200,784} \end{bmatrix}$ ，数据的真实标签为 $t = [t_1, t_2, \dots, t_{10}]$

隐含层与输出层的连接权值为 $W_{ho} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,200} \\ \vdots & \ddots & \vdots \\ w_{10,1} & \cdots & w_{10,200} \end{bmatrix}$ ，网络中各个节点所使用的激活函

数为 sigmoid 激活函数。根据公式(1.0)可以计算每个数据 x 的网络输出 y 。

$$y = \text{sigmoid}(W_{ho} \text{sigmoid}(W_{ih}x)) \quad (\text{公式 1.0})$$

采用的网络训练算法为随机梯度下降法来更新网络的权值, 为了减小复杂度, 在计算反向误差时使用原始权值进行误差传播。输出层误差 $E_{out} = t - y$; 隐藏层误差 $E_{hid} = W_{ho}^T E_{out}$ 。学习率为 $lr=0.1$, 所得到的权值更新公式如下公式 (1.1) 和公式 (1.2)。

$$W_{ho} += lr * (E_{out} * y * (1 - y)) y_{hidden}^T \quad \text{公式 (1.1)}$$

$$W_{ih} += lr * (E_{hid} * y_{hidden} * (1 - y_{hidden})) x^T \quad \text{公式 (1.2)}$$

我们从上述公式可以看到训练网络时存在大量的矩阵运算, 可以对其进行并行化处理, 加快训练速度。

四、优化过程及结果

使用 C 语言进行神经网络训练的串行代码运行结果如下图 4.1 所示, 训练时间为 80.7910s, 测试时间为 6.9270s, 正确率保持在 0.94 左右。训练的时间还是挺久的, 需要对其进行使用 GPU 优化, 在不损失精度的条件下, 减小训练时间。

```
train time used:80.791054s
currentCount= 9416 total_count=10000
the total correct is 0.941600
test time used:6.927020s
predict time used:-0.009508s
```

图 4.1 CPU 运行结果

优化方案 1:

GPU 优化方式: 因为网络层与层之间是依赖关系, 因此层间是串行运算, 同一层内的节点之间无相互作用, 可以并行运算。每执行一次训练使用一个样本数据, 即数据传输每次只传输一个样本数据。训练数据共执行 60000 个样本数据, 因此有 60000 次的样本拷贝到设备。代码中涉及矩阵运算的使用 cublas 库 `cublasSgemv` 来进行并行化出处理。

主要训练代码如下所示:

```
for (int i = 0; i < TRAIN_SIZE; i++)
{
    getData(fp, Input, Target);
    cudaMemcpy(d_Input, Input, sizeof(float)*INPUTNODES, cudaMemcpyHostToDevice);
    cudaMemcpy(d_Target, Target, sizeof(float)*OUTPUTNODES,
        cudaMemcpyHostToDevice);
    train(d_Input, d_Hidden, d_Output, d_Target, d_Wih, d_Who, output_error, hidden_error);
}

void train(float *d_Input, float *d_Hidden, float *d_Output, float *d_Target, float
*d_Wih, float *d_Who, float *output_error, float *hidden_error) {
    cublasHandle_t handle = 0;
    cublasCreate(&handle);
```

```

float alpha = 1.0;
float beta = 0.0;
cublasSgemv(handle, CUBLAS_OP_N, HIDDENNODES, INPUTNODES, &alpha, d_Wih,
HIDDENNODES, d_Input, 1,&beta, d_Hidden, 1);
Sigmod_kernel << <1, HIDDENNODES >> > (d_Hidden, HIDDENNODES);
cublasSgemv(handle, CUBLAS_OP_N, OUTPUTNODES, HIDDENNODES, &alpha, d_Who,
OUTPUTNODES, d_Hidden, 1, &beta, d_Output, 1);
Sigmod_kernel << <1, OUTPUTNODES >> > (d_Output, OUTPUTNODES);
VecSub_kernel << <1, OUTPUTNODES >> > (d_Target, d_Output, output_error,
OUTPUTNODES);
cublasSgemv(handle, CUBLAS_OP_T, OUTPUTNODES, HIDDENNODES, &alpha, d_Who,
OUTPUTNODES, output_error, 1, &beta, hidden_error, 1);
argument1_kernel << <1, OUTPUTNODES >> > (output_error, d_Output, OUTPUTNODES);
dim3 block(16, 16);
dim3 grid((HIDDENNODES + block.x - 1) / block.x, (OUTPUTNODES + block.y - 1) /
block.y);
updateW << <grid,block >> > (output_error, d_Hidden, d_Who, OUTPUTNODES,
HIDDENNODES);
argument1_kernel << <1, HIDDENNODES >> > (hidden_error, d_Hidden, HIDDENNODES);
grid.x = (INPUTNODES + block.x - 1) / block.x;
grid.y = (HIDDENNODES + block.y - 1) / block.y;
updateW << <grid, block >> > (hidden_error, d_Input, d_Wih, HIDDENNODES,
INPUTNODES);
cublasDestroy(handle);
}

```

运行结果如图 4.1，我们可以看到时间减小了一半。

```

E:\cuda_prj\myGPU\myNeural\x64\Release\myNeural.exe starting reduction at device 0: GeForce GTX 1050 Ti
Training .....
  training over ,time used 44.717522s
Testing .....
  test over,time used 6.447538s
Consequention:
  currectCount= 9430 total_count=10000
  the total accuracy is 0.943000

```

图 4.1

我们使用 nvprof 来查看时间的损耗情况，如图 4.2。可以看到在训练中花费时间最多的为 cudaMemcpy 和 updateW 核函数。可以对这两个进行优化。

Time(%)	Time	Calls	Avg	Min	Max	Name
55.37%	2.77001s	120000	23.083us	1.3120us	45.536us	updateW(float*, float*, float*, unsig
25.82%	1.29177s	140000	9.2260us	2.8480us	16.096us	void gemv2N_kernel_val<float, float, i
						int=4, int=4, int=1>(float, float, cublasGemm2Params_v2<float, float, float>)
5.61%	280.83ms	200002	1.4040us	543ns	97.247us	[CUDA memcpy HtoD]
4.75%	237.52ms	140000	1.6960us	1.4070us	15.584us	Sigmod_kernel(float*, int)
3.13%	159.02ms	60000	2.6500us	2.4640us	15.071us	void gemv2T_kernel_val<float, float, i
						int=2, int=2, bool=0>(int, int, float, float const *, int, float const *, int, float, float*, int)
3.01%	150.62ms	120000	1.2550us	927ns	15.968us	argument1_kernel(float*, float*, int)
2.09%	104.76ms	60000	1.7450us	1.6630us	9.0560us	VecSub_kernel(float*, float*, float*,
0.16%	7.9477ms	10000	794ns	767ns	1.0560us	[CUDA memcpy DtoH]
==28476== API calls:						
Time(%)	Time	Calls	Avg	Min	Max	Name
35.77%	19.2449s	280008	68.729us	464ns	1.05010s	cudaFree
35.33%	19.0068s	210008	90.505us	6.0290us	793.49ms	cudaMalloc
15.69%	8.44464s	210002	40.212us	19.014us	65.462ms	cudaMemcpy
6.43%	3.45996s	640000	5.4060us	3.2460us	5.0848ms	cudaLaunch
2.57%	1.38058s	140000	9.8610us	8.3470us	484.17us	cudaThreadSynchronize
1.32%	711.64ms	1120000	635ns	0ns	2.1468ms	cudaEventCreateWithFlags

图 4.2

优化方案 2:

训练过程中有大量时间用于数据传输,为了减小主机和设备的数据交互,因此将全部数据集的数据一次全部拷贝到设备中,再进行训练。代码如下:

```
unsigned int offset_input = 0;
unsigned int offset_output = 0;
for (int i = 0; i < TRAIN_SIZE; i++)
{
    offset_input = i*INPUTNODES;
    offset_output = i*OUTPUTNODES;
    getData(fp, Input, Target);
    CHECK(cudaMemcpy(&d_Input[offset_input], Input, sizeof(float)*INPUTNODES,
        cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(&d_Target[offset_output], Target, sizeof(float)*OUTPUTNODES,
        cudaMemcpyHostToDevice));
}
//训练
train(d_Input, d_Hidden, d_Output, d_Target, d_Wih, d_Who, output_error,
hidden_error);
```

运行结果如图 4.3, 我们可以看到训练时间减小到 12.5s, 比上次加快了 3.75 倍, 比 CPU 串行代码加快了 6.5 倍。

```
E:\cuda_prj\myGPU\myNeural\x64\Release\myNeural.exe starting reduction at device 0: GeForce GTX 1050 Ti
Training .....
training over ,time used 12.506361s
Testing .....
test over,time used 1.374262s
Consequention:
correctCount= 9432 total_count=10000
the total accuracy is 0.943200
```

图 4.3

我们使用 nvprof 来查看时间的损耗情况,如图 4.4。从图中可以看到 cudaMemcpy 时间降低了, updateW 核函数第时间还很高,接下来优化 updateW 核函数。

Time(%)	Time	Calls	Avg	Min	Max	Name
56.10%	2.34982s	120000	19.581us	1.0560us	45.311us	updateW(float*, float*, float*, unsigned i
18.64%	780.92ms	140000	5.5780us	2.2390us	19.839us	void gemv2N_kernel_val<float, float, float,
						int=4, int=4, int=1>(float, float, cublasGemmV2Params_v2<float, float, float>)
9.77%	409.37ms	10000	40.937us	36.173us	245.33us	[CUDA memcpy HtoH]
4.19%	175.31ms	140000	1.2520us	1.0870us	20.703us	Sigmoid_kernel(float*, int)
3.80%	159.24ms	130004	1.2240us	544ns	97.278us	[CUDA memcpy HtoD]
3.06%	128.11ms	120000	1.0670us	800ns	20.160us	argument1_kernel(float*, float*, int)
2.46%	102.90ms	60000	1.7140us	1.6310us	19.712us	void gemv2T_kernel_val<float, float, float,
						int=2, int=2, bool=0>(int, int, float, float const *, int, float const *, int, float, float*, int)
1.76%	73.650ms	60000	1.2270us	1.1520us	20.096us	VecSub_kernel(float*, float*, float*, int)
0.22%	9.0381ms	10000	903ns	799ns	17.856us	[CUDA memcpy DtoD]
0.00%	60.735us	1	60.735us	60.735us	60.735us	[CUDA memcpy DtoH]

==24516== API calls:

Time(%)	Time	Calls	Avg	Min	Max	Name
50.12%	5.89273s	150005	39.283us	5.5650us	22.988ms	cudaMemcpy
34.05%	4.00255s	640000	6.2530us	3.2460us	4.7583ms	cudaLaunch
5.79%	680.80ms	14	48.628ms	6.4920us	677.51ms	cudaMalloc
4.47%	525.35ms	16	32.834ms	1.8550us	515.64ms	cudaFree
3.90%	458.50ms	2500000	183ns	0ns	1.0129ms	cudaSetupArgument
1.23%	145.05ms	640000	226ns	0ns	576.93us	cudaConfigureCall

图 4.4

优化方案 3:

updateW 核函数主要用于每次训练时来更新连接权值,代码如下。该核函数的不同 block 和 grid 的配置会有不同的影响,我们尝试了 4 种配置:block(16, 16)、block(4, 8)、block(8, 8)、block(8, 16)。

并进行比较。

```
__global__ void updateW(float *a, float *b, float *C, unsigned int row, unsigned int col)
{
    float lr = LR;
    unsigned int i = threadIdx.y + blockDim.y*blockIdx.y;
    unsigned int j = threadIdx.x + blockDim.x*blockIdx.x;
    if (i < row && j < col)
    {
        C[i+ j*row] += lr*a[i] * b[j]; //权值是列优先存储
    }
}
```

结果如下表 4.1 所示：从表中我们可以看到最好的 block 配置为 (8, 8), 训练时间减小到 10.56s, 比上次优化减小了近 2s。

表 4.1 不同 block 运行对比

block	训练时间	updateW 运行时间
16, 16	12.5s	2.359s
4, 8	10.91s	1.360s
8, 8	10.56s	1.232s
8, 16	10.62s	1.235s

我们使用 nvprof 来查看时间的损耗情况，如图 4.5。我们可以看到在 API 调用中 cudaMemcpy 共调用了 150005 次，总时间达 5.8s，而从主机到设备端传输数据却只有 159ms。调用 cudaMemcpy API 花费了太多时间。

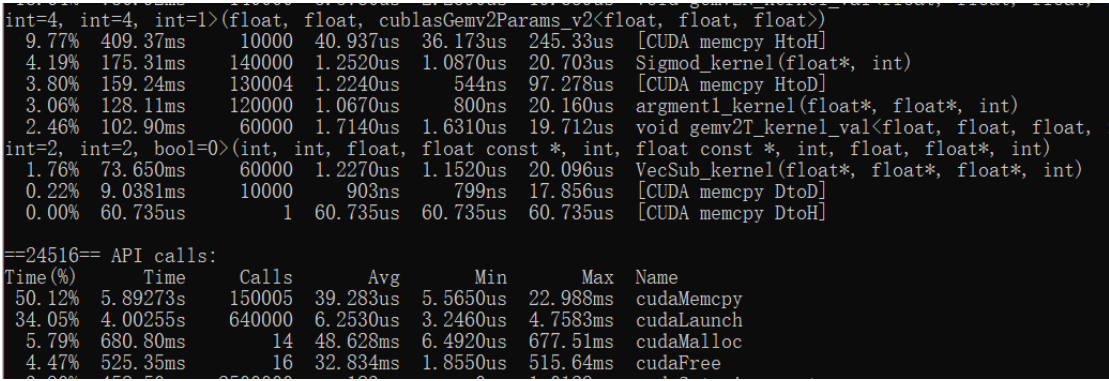


图 4.5

优化方案 4:

在传全部数据到设备时采用只调用一次 cudaMemcpy API 的方式，将所有数据传到显存中。代码如下；

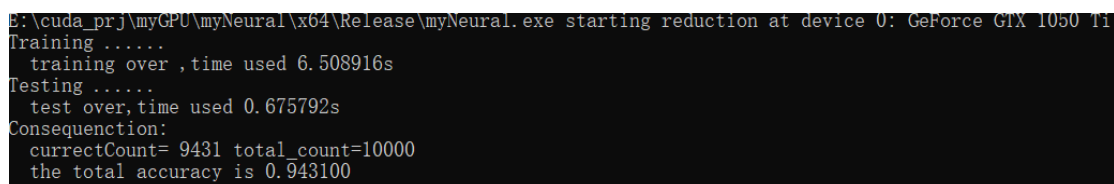
```
unsigned int offset_input = 0;
unsigned int offset_output = 0;
for (int i = 0; i < TRAIN_SIZE; i++)
{
    offset_input = i*INPUTNODES;
    offset_output = i*OUTPUTNODES;
    getData(fp, Input+ offset_input, all_Target+ offset_output);
}
```

```

}
CHECK(cudaMemcpy(d_Input, Input, sizeof(float)*INPUTNODES*TRAIN_SIZE,
cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_Target, all_Target, sizeof(float)*OUTPUTNODES*TRAIN_SIZE,
cudaMemcpyHostToDevice));
//训练
train(d_Input, d_Hidden, d_Output, d_Target, d_Wih, d_Who, output_error,
hidden_error);

```

运行结果如图 4.6, 时间减小到 6.5s, 比上次运行时间减小了 4.6s。



```

E:\cuda_prj\myGPU\myNeural\x64\Release\myNeural.exe starting reduction at device 0: GeForce GTX 1050 Ti
Training .....
training over ,time used 6.508916s
Testing .....
test over,time used 0.675792s
Consequention:
correctCount= 9431 total_count=10000
the total accuracy is 0.943100

```

图 4.6

优化方案 5:

我们发现 updateW 核函数虽然减小到 1.23s, 但仍然是所有核函数中运行时间最多的, 我们在 cublas 库中找到了替代的库函数 cublasSger(), 它用于计算 $A = \alpha xy^T + A$ 表达式。代码更改如下:

```

void train(float *d_Input, float *d_Hidden, float *d_Output, float *d_Target, float
*d_Wih, float *d_Who, float *output_error, float *hidden_error)
{
    cublasHandle_t handle = 0;
    cublasCreate(&handle);
    float alpha = 1.0;
    float beta = 0.0;
    float alpha2 = LR;
    unsigned int offset_input = 0;
    unsigned int offset_output = 0;
    for (int i = 0; i < TRAIN_SIZE; i++)
    {
        offset_input = i*INPUTNODES;
        offset_output = i*OUTPUTNODES;
        //前向传播
        cublasSgemv(handle, CUBLAS_OP_N, HIDDENNODES, INPUTNODES, &alpha, d_Wih,
HIDDENNODES, d_Input+ offset_input, 1, &beta, d_Hidden, 1);
        Sigmoid_kernel << <1, HIDDENNODES >> > (d_Hidden, HIDDENNODES);
        cublasSgemv(handle, CUBLAS_OP_N, OUTPUTNODES, HIDDENNODES, &alpha, d_Who,
OUTPUTNODES, d_Hidden, 1, &beta, d_Output, 1);
        Sigmoid_kernel << <1, OUTPUTNODES >> > (d_Output, OUTPUTNODES);
        //误差反向传播,
        VecSub_kernel << <1, OUTPUTNODES >> > (d_Target+ offset_output, d_Output,
output_error, OUTPUTNODES);
        cublasSgemv(handle, CUBLAS_OP_T, OUTPUTNODES, HIDDENNODES, &alpha, d_Who,
OUTPUTNODES, output_error, 1, &beta, hidden_error, 1);
    }
}

```

```

//更新权值
argument1_kernel << <1, OUTPUTNODES >> > (output_error, d_Output, OUTPUTNODES);
cublasSger(handle, OUTPUTNODES, HIDDENNODES, &alpha2, output_error, 1,
d_Hidden, 1, d_Who, OUTPUTNODES);
argument1_kernel << <1, HIDDENNODES >> > (hidden_error, d_Hidden, HIDDENNODES);
cublasSger(handle, HIDDENNODES, INPUTNODES, &alpha2, hidden_error, 1, d_Input
+ offset_input, 1, d_Wih, HIDDENNODES);
}
cublasDestroy(handle);
}

```

运行结果如图 5.7 所示，时间减小到 5.67s, 比上次减小了近 1s。

```

E:\cuda_prj\myGPU\myNeural\x64\Release\myNeural.exe starting reduction at device 0: GeForce GTX 1050 Ti
Training .....
training over ,time used 5.671744s
Testing .....
test over,time used 0.650036s
Consequention:
currentCount= 9405 total_count=10000
the total accuracy is 0.940500

```

图 5.7

五、实验小结

表 5.1 各优化结果对比

优化方案	训练时间	总体正确率	相对 CPU 加速比
CPU 串行	80.791s	0.9416	1.0
优化方案 1	44.715s	0.9430	1.986
优化方案 2	12.506s	0.9432	6.460
优化方案 3	10.560s	0.9432	7.651
优化方案 4	6.509s	0.9431	12.412
优化方案 5	5.672s	0.9405	14.244

从表 5.1 可以看出，最终在 GPU 上优化结果比 cpu 串行程序快了 14 倍左右。在进行 cuda 优化时，考虑到大部分时间都是在进行 cpu 与 gpu 间的数据交互，我们要减小这种数据交互，以此来减小时间开销；同时要充分利用 GPU 的计算资源，增大 block 的数目，增加每个 SM 常驻线程快的数目来最大化并行性，提高运算速度。