

并行计算大作业课程报告

方驰正 PB21000163

2024 年 6 月 2 日

1 实验目的

在本次实验中，我们使用 MNIST 手写数字数据集，训练了一个 mlp 自动感知机，并使用其预测了测试集，将结果提交至 kaggle 上的手写数字识别项目。分别实现了 cpu 串行版本和 gpu 并行版本，并对 gpu 并行版本进行优化，最终达到了 96.1% 的准确率，以及相比于 cpu 版本的 9.5 倍的加速比。

2 实验环境

- 操作系统: WSL2 Ubuntu 22.04
- 编程语言: C++, cuda
- 编译器: g++ 11.4.0
- cuda 版本: 12.4
- GPU: NVIDIA GeForce RTX 4060 Laptop

3 算法设计和实现

以下为主要的文件目录结构:

```
/
├── /data
│   ├── train.csv
│   └── test.csv
├── /output
│   └── submission.csv
├── main.cpp
├── mnist.h
├── mnist.cpp
├── matrix.h
├── matrix.cpp
└── Makefile
```

可以看到,核心代码主要分为三个部分,分别是 mnist.h 和 mnist.cpp, matrix.h 和 matrix.cpp, 以及 main.cpp。其中 mnist.h 和 mnist.cpp 主要负责数据的读取和预处理, matrix.h 和 matrix.cpp

主要负责矩阵运算，main.cpp 主要负责模型的训练和预测。下面我们分别介绍这三个部分的实现。

3.1 mnist

这一部分为数据的读取和预处理部分。我们使用了 mnist 数据集，其中 train.csv 为训练集，test.csv 为测试集。

具体地，在 mnist.h 中的声明如下：

```
1 struct mnist_data {
2     unsigned char label;
3     unsigned char a[28 * 28];
4 };
5 std::vector<mnist_data> input(std::string filename, bool is_train);
6 void output(std::string filename, std::vector<int> data);
```

可以看到，我们定义了一个 mnist_data 结构体，其中包含了一个 label 和一个 28*28 的数组，表示该数据的标签和特征。input 函数用于读取数据，output 函数用于输出数据。

3.2 matrix

```
1 struct Mat {
2     int n, m;
3     double *a;
4     double &operator()(int i, int j) const;
5     void random_init(int n, int m, double loc, double scale); //随机初始化
6     void zero_init(int n, int m);
7     Mat();
8     Mat(Mat &&_);
9     Mat(const Mat &_);
10    Mat operator=(Mat &&_);
11    Mat operator=(const Mat &_);
12    ~Mat();
13    Mat operator*(const Mat &_) const;
14    Mat operator*(const double &_) const;
15    Mat operator+(const Mat &_) const;
16    Mat operator-(const Mat &_) const;
17    Mat relu() const; //激活函数
18    Mat relu_() const; //激活函数的导数
19    Mat softmax() const; // softmax函数
20    Mat softmax_() const; // softmax函数的导数
21    Mat T() const; //转置
22    double sum() const;
23
24    Mat mult(const Mat &_) const; //矩阵对应元素相乘
25 };
26
27 double Loss(const Mat &y, const Mat &y_hat);
28 double Accuracy(const Mat &y, const Mat &y_hat);
```

在这里，我们定义了一个 Mat 结构体，用来表示一个矩阵。同时，我们定义了一系列在 mlp 训练与预测中需要用到的函数，如矩阵乘法、矩阵加法、矩阵减法、激活函数、softmax 函数、损失函数、准确率等。

3.3 mlp

这是整个模型的核心部分，我们使用了一个三层的 mlp 模型，分别为输入层、隐藏层和输出层。其中，输入层包含 784 个神经元，隐藏层包含 256 个神经元，输出层包含 10 个神经元。我们使用了 relu 作为激活函数，使用 softmax 作为输出层的激活函数。同时，我们使用梯度下降方法进行反向传播。在训练时，我们使用了交叉熵作为损失函数。主要相关代码如下：

```

1 struct MLP {
2     Mat W1, W2, b1, b2;
3     double lr;
4
5     void forward(const Mat &input, Mat &z1, Mat &a1,
6                 Mat &z2, Mat &a2) {
7         z1 = input * W1 + b1;
8         a1 = z1.relu();
9         z2 = a1 * W2 + b2;
10        a2 = z2.softmax();
11    }
12
13    void backward(Mat &input, Mat &z1, Mat &a1, Mat &z2,
14                 Mat &a2, Mat &label) {
15        Mat delta2 = (a2 - label).mult(z2.softmax_());
16        Mat delta1 = (delta2 * W2.T()).mult(z1.relu_());
17
18        Mat dW2 = a1.T() * delta2;
19        Mat dW1 = input.T() * delta1;
20
21        W1 = W1 - dW1 * lr;
22        W2 = W2 - dW2 * lr;
23        b1 = b1 - delta1 * lr;
24        b2 = b2 - delta2 * lr;
25    }
26
27    std::pair<double, double>
28    step(Mat &input, Mat &label,
29         bool train = true) {
30        Mat z1, a1, z2, a2;
31        forward(input, z1, a1, z2, a2);
32
33        double loss = Loss(a2, label);
34        double accuracy = Accuracy(a2, label);
35
36        if (train) backward(input, z1, a1, z2, a2, label);
37        return std::make_pair(loss, accuracy);
38    }
39 } net;

```

可以看到，我们定义了一个 MLP 结构体，其中包含了 W1、W2、b1、b2 和 lr 等参数。forward 函数用于前向传播，backward 函数用于反向传播，step 函数用于一次训练迭代。在训练时，我们使用了随机梯度下降法。

```

1 void train(std::vector<mnist_data> &data, int epoch) {
2     net.init(784, 256, 10, 0.01);
3     ans = net;
4     double best_accuracy = 0;
5     double t = clock();
6     for (int i = 0; i < epoch; i++) {

```

```

7     std::vector<double> loss, accuracy;
8     int t = 0;
9     for (auto &d : data) {
10         Mat input, label;
11         input.zero_init(1, 784);
12         for (int i = 0; i < 784; i++)
13             input.a[i] = d.a[i] / 255.0;
14
15         label.zero_init(1, 10);
16         label.a[d.label] = 1;
17         auto res = net.step(input, label);
18         loss.push_back(res.first);
19         accuracy.push_back(res.second);
20     }
21     double average_accuracy =
22         std::accumulate(accuracy.begin(),
23                         accuracy.end(), 0.0)
24         / accuracy.size();
25     if (average_accuracy > best_accuracy) {
26         best_accuracy = average_accuracy;
27         ans = net;
28     }
29 }
30 }
31
32 std::vector<int> test(std::vector<mnist_data> &data) {
33     std::vector<int> res;
34     for (auto &d : data) {
35         Mat input;
36         input.zero_init(1, 784);
37         for (int i = 0; i < 784; i++)
38             input.a[i] = d.a[i] / 255.0;
39
40         Mat z1, a1, z2, a2;
41         ans.forward(input, z1, a1, z2, a2);
42
43         int predict = 0;
44         for (int i = 0; i < 10; i++) {
45             if (a2.a[i] > a2.a[predict]) predict = i;
46         }
47         res.push_back(predict);
48     }
49     return res;
50 }

```

如上为训练以及测试部分的代码。

可以看到，在训练时，我们将数据归一化到 $[0, 1]$ 之间，并进行若干个 epoch 的训练。取平均准确率最高的模型作为最终的模型。为了简单起见，我们在测试时没有使用交叉验证，而是直接使用了训练集的数据进行测试。

在测试时，我们将数据归一化到 $[0, 1]$ 之间，然后使用训练好的模型进行预测。最终将预测结果输出到 submission.csv 文件中。

3.4 串行实验结果

编译并运行后，输出如下：

```

1 epoch:0 average_acc:0.861167 time used:128.734s
2 epoch:1 average_acc:0.925143 time used:257.458s
3 epoch:2 average_acc:0.941952 time used:384.517s
4 epoch:3 average_acc:0.952071 time used:510.324s
5 epoch:4 average_acc:0.959524 time used:635.946s
6 epoch:5 average_acc:0.964476 time used:770.660s
7 epoch:6 average_acc:0.968905 time used:899.398s
8 epoch:7 average_acc:0.972143 time used:1024.27s
9 epoch:8 average_acc:0.975429 time used:1150.56s
10 epoch:9 average_acc:0.978310 time used:1279.45s

```

可以看到，经过 10 个 epoch 的训练，最终的准确率为 97.8%，尚未过拟合。但是训练时间较长，达到了 23 分钟。将输出的测试数据提交至 kaggle 上，得到的准确率如下：

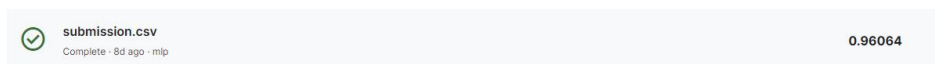


图 1: 串行版本的 kaggle 提交结果

由于训练时间较长，我们尝试并行计算。

4 并行实现和优化

可以看到，串行版本的程序的运行时间还是比较长，因此我们尝试使用 cuda，在 gpu 上进行并行计算。由于主要的计算部分都在 matrix.cpp 中，因此我们重点的优化部分也在这里。这里，我们以矩阵乘法为例，介绍我们的优化方法。

4.1 编写核函数

```

1 void Mat::zero_init(int N, int M) {
2     n = N, m = M;
3     a = new double[n * m];
4     memset(a, 0, sizeof(double) * n * m);
5 }
6 Mat::~Mat() {
7     if (a) delete[] a;
8 }
9 Mat Mat::operator*(const double &_) const {
10     Mat res;
11     res.zero_init(n, m);
12     for (int i = 0; i < n; i++)
13         for (int j = 0; j < m; j++)
14             res(i, j) = (*this)(i, j) * _;
15     return res;
16 }

```

这是最原始的矩阵乘法的实现，我们可以看到，我们首先在栈空间上申请了一个新的矩阵 res，对其进行矩阵乘法运算，返回后调用析构函数释放内存。我们使用 cuda 优化后的代码如下：

```

1 void Mat::zero_init(int N, int M) {
2     n = N, m = M;
3     cudaMalloc(&a, sizeof(double) * n * m);
4     cudaMemset(a, 0, sizeof(double) * n * m);
5 }

```

```

6 Mat::~Mat() {
7     if (a) cudaFree(a);
8 }
9 __global__ void mult_mat_kernel(double *a, double *b,
10                                double *c, int n, int m,
11                                int k) {
12     int i = blockIdx.x * blockDim.x + threadIdx.x;
13     if (i < n * k) {
14         int x = i / k, y = i % k;
15         c[i] = 0;
16         for (int j = 0; j < m; j++)
17             c[i] += a[x * m + j] * b[j * k + y];
18     }
19 }
20 Mat Mat::operator*(const Mat &_) const {
21     assert(m == _.n);
22     Mat res;
23     res.zero_init(n, _.m);
24     mult_mat_kernel<<<(n * _.m + 1023) / 1024, 1024>>>(
25         a, _.a, res.a, n, m, _.m);
26     return res;
27 }

```

可以看到，我们仅仅是将核心的计算部分改为了 cuda 的核函数。将这份代码编译训练 1 个 epoch 后，得到如下结果：

```
1 epoch:0 average_acc:0.856976 time used:71.9752s
```

可以看到，相比于 cpu 版本，速度提升了将近一倍。然而我们发现，cuda 版本的速度并没有达到我们的预期，因此我们尝试进一步优化。使用 NVIDIA 的 nsight 进行性能分析，得到如下结果：

Time	% Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
49.3%	13.831 s	367270	37.658 µs	2.276 µs	519 ns	5.015 ms	72.053 µs	cudaFree
29.8%	8.362 s	367286	22.766 µs	2.098 µs	655 ns	90.375 ms	173.375 µs	cudaMalloc
8.9%	2.499 s	146928	17.006 µs	14.235 µs	2.666 µs	3.362 ms	18.237 µs	cudaMemcpy
6.3%	1.760 s	283322	6.212 µs	3.844 µs	1.415 µs	555.861 µs	7.989 µs	cudaMemset
5.6%	1.579 s	262334	6.017 µs	4.416 µs	1.941 µs	513.023 µs	7.905 µs	cudaLaunchKernel
0.0%	509 ns	1	509 ns	509 ns	509 ns	509 ns	0 ns	cuModuleGetLoadingMode

图 2: cuda 性能分析 1

可以看到，cudaFree 占了 49.3% 的时间，cudaMalloc 占了 29.8% 的时间，cudaMemcpy 占了 8.9% 的时间，远远超过了 cudaLaunchKernel 的时间。因此我们尝试优化这几部分。

4.2 优化内存分配

观察代码可以发现，我们在每次调用矩阵相关函数时，都需要在栈空间上分配 cuda 指针，申请 cuda 内存，又在函数结束时调用析构函数释放 cuda 内存。这样的操作会导致频繁的 cudaMalloc 和 cudaFree，从而导致性能下降。因此，我们尝试将每个中间结果以 static 的形式保存在全局变量中，避免重复的 cudaMalloc 和 cudaFree。具体代码如下：

```

1 void forward(const Mat &input, Mat &z1, Mat &a1,
2              Mat &z2, Mat &a2) {
3     static Mat input_mult_W1(input.n, W1.m);
4     Mult_mat(input, W1, input_mult_W1);
5     Add_mat(input_mult_W1, b1, z1);
6     Relu(z1, a1);
7
8     static Mat a1_mult_W2(a1.n, W2.m);

```

```

9     Mult_mat(a1, W2, a1_mult_W2);
10    Add_mat(a1_mult_W2, b2, z2);
11    Softmax(z2, a2);
12 }
13 void Mult_mat(const Mat &a, const Mat &b, Mat &c) {
14     assert(a.m == b.n);
15     assert(a.n == c.n);
16     assert(b.m == c.m);
17     mult_mat_kernel<<<(a.n * b.m + 1023) / 1024, 1024>>>(
18         a.a, b.a, c.a, a.n, a.m, b.m);
19 }

```

这样，我们优化掉了绝大部分的 `cudaMalloc` 和 `cudaFree`，从而提升了性能。测试结果如下：

```

1 epoch:0 average_acc:0.857357 time used:18.3029s
2 epoch:1 average_acc:0.923952 time used:36.164s
3 epoch:2 average_acc:0.940238 time used:53.9217s
4 epoch:3 average_acc:0.951690 time used:71.6025s
5 epoch:4 average_acc:0.958976 time used:89.4938s
6 epoch:5 average_acc:0.964762 time used:107.319s
7 epoch:6 average_acc:0.969524 time used:125.155s
8 epoch:7 average_acc:0.973071 time used:143.015s
9 epoch:8 average_acc:0.976024 time used:161.011s
10 epoch:9 average_acc:0.978238 time used:179.375s

```

可以看到，相较于 `cpu` 版本已经达到了约 7 倍的加速比。我们继续使用 NVIDIA 的 `nsight` 进行性能分析，得到如下结果：

Time	% Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
52.6%	18.519 s	210328	88.048 μ s	44.510 μ s	4.419 μ s	4.677 ms	83.696 μ s	cudaMemcpy
30.0%	10.550 s	1705052	6.187 μ s	4.135 μ s	1.738 μ s	856.352 μ s	9.105 μ s	cudaLaunchKernel
10.1%	3.557 s	63150	56.320 μ s	54.958 μ s	889 ns	2.677 ms	33.338 μ s	cudaDeviceSynchronize
5.1%	1.785 s	210349	8.487 μ s	4.128 μ s	1.175 μ s	103.581 ms	253.498 μ s	cudaMalloc
2.3%	796.677 ms	147171	5.413 μ s	4.503 μ s	1.435 μ s	493.235 μ s	6.206 μ s	cudaMemset
0.0%	679.054 μ s	20	33.952 μ s	13.653 μ s	1.629 μ s	217.123 μ s	53.151 μ s	cudaFree
0.0%	635 ns	1	635 ns	635 ns	635 ns	635 ns	0 ns	cuModuleGetLoadingMode

图 3: cuda 性能分析 2

可以看出，`cudaMalloc` 和 `cudaFree` 的时间大大减少，而 `cudaMemcpy` 占了 52.6% 的运行时间。因此我们尝试继续优化 `cudaMemcpy`。

4.3 优化 cudaMemcpy

使用 `nsight`，我们能发现主要的 `cudaMemcpy` 操作都是在计算 Accuracy 和 Loss 时，将 `cuda` 内存中的数据拷贝到主机内存中。我们以计算 Accuracy 为例，介绍我们的优化方法。

```

1 __global__ void Accuracy_kernel(double *a, double *b, int n,
2                                 int *c) {
3     int Maxa = 0, Maxb = 0;
4     for (int i = 0; i < n; i++) {
5         if (a[i] > a[Maxa]) Maxa = i;
6         if (b[i] > b[Maxb]) Maxb = i;
7     }
8     *c = Maxa == Maxb;
9 }
10
11 double Accuracy(const Mat &a, const Mat &b) {
12     // a,b are 1*n matrix
13

```

```

14     static int *c;
15     cudaMalloc(&c, sizeof(int));
16     Accuracy_kernel<<<1, 1>>>(a.a, b.a, a.m, c);
17     int host_c;
18     cudaMemcpy(&host_c, c, sizeof(int),
19               cudaMemcpyDeviceToHost);
20
21     return host_c;
22 }

```

我们可以看到，由于我们需要将一个 int 类型的数据拷贝到主机中，因此使用了 cudaMemcpy。这样的操作会导致性能下降。因此，我们将所有的 Accuracy 和 Loss 的计算都放在 cuda 内存中进行，并在主程序中使用 cuda 数组来保存这些数据，直到需要计算 Average Accuracy 时，再将数据拷贝到主机内存中。这样，我们就避免了频繁的 cudaMemcpy 操作，从而提升了性能。输出结果如下：

```

1 epoch:0 average_acc:0.856595 time used:13.6053s
2 epoch:1 average_acc:0.923310 time used:27.1446s
3 epoch:2 average_acc:0.940500 time used:40.6845s
4 epoch:3 average_acc:0.951762 time used:54.2125s
5 epoch:4 average_acc:0.959762 time used:67.7621s
6 epoch:5 average_acc:0.965143 time used:81.2944s
7 epoch:6 average_acc:0.969619 time used:94.8306s
8 epoch:7 average_acc:0.972905 time used:108.369s
9 epoch:8 average_acc:0.976071 time used:121.902s
10 epoch:9 average_acc:0.978238 time used:135.449s

```

可以看到，相较于 cpu 版本，我们的 cuda 版本已经达到了约 9.5 倍的加速比。运用 nsight 进行性能分析，得到如下结果：

Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
87.3%	18.562 s	1588896	11.682 µs	4.109 µs	1.736 µs	6.044 ms	32.512 µs	cudaLaunchKernel
8.3%	1.772 s	84030	21.088 µs	16.559 µs	4.522 µs	13.532 ms	48.684 µs	cudaMemcpy
2.5%	521.959 ms	84051	6.210 µs	2.675 µs	1.198 µs	80.836 ms	280.063 µs	cudaMalloc
2.0%	417.221 ms	84021	4.965 µs	4.460 µs	1.408 µs	453.336 µs	5.271 µs	cudaMemset
0.0%	960.366 µs	20	48.018 µs	11.001 µs	1.766 µs	267.351 µs	82.311 µs	cudaFree
0.0%	3.869 µs	1	3.869 µs	3.869 µs	3.869 µs	3.869 µs	0 ns	cudaDeviceSynchronize
0.0%	414 ns	1	414 ns	414 ns	414 ns	414 ns	0 ns	cuModuleGetLoadingMode

图 4: cuda 性能分析 3

可以看到，主要的时间开销都在 cudaLaunchKernel 上，占据了总运行时间的 87.3%。因此，接下来的优化方向应该是优化时间开销大的核函数，如矩阵乘法。不过由于时间限制，暂时没有进一步优化。我们将测试结果提交至 kaggle 上，得到如下结果：


 submission.csv Complete · 6d ago · 1	0.96135
--	----------------

图 5: 并行版本的 kaggle 提交结果

可以看到我们的并行方法在达到 9.5 倍加速比的同时，准确率也有略微提升，达到了 96.135%。

5 总结

在本次实验中，我们使用了 mnist 数据集，训练了一个 mlp 自动感知机，并使用其预测了测试集，将结果提交至 kaggle 上的手写数字识别项目。分别实现了 cpu 串行版本和 gpu 并行版本，

并对 gpu 并行版本进行优化，最终达到了 96.1% 的准确率，以及相比于 cpu 版本的 9.5 倍的加速比。

通过本次实验，我学会了如何使用 cuda 进行并行计算，以及如何使用 nsight 调试并优化 cuda 程序。