Application of GPUs in Training A Perceptron on CIFAR-10 Dataset

Applied GPU Programming 2020 Final Report

Chang Fu
KTH Royal Institute of Technology
changf@kth.se

Chao Xiong
KTH Royal Institute of Technology
cxiong@kth.se

Our target project grade is +1, and our target final grade is A since we have done all bonus parts. The Github repository to this project is *here*.

1. Introduction

Machine learning has undoubtedly changed the ways of our lives nowadays, and is being applied to various fields. In order to achieve better performance, neural networks go deeper and deeper. As a result, training deep neural networks acquires more and more computational resources, and this is where GPUs show their power.

Training neural networks involves a lot of matrix computations which can be performed in parallel, and GPUs are designed as throughput-oriented architecture, which means they are able to process multiple computations simultaneously. The application of GPUs contributes to fast development of machine learning and makes it possible for many complex models.

The simplest neural network dates back 1957 when the perceptron [Rosenblatt 1957] was invented by Frank Rosenblatt at the Cornell Aeronautical Laboratory. The perceptron is a simple linear classifier with only one hidden layer, and it is trained using gradient descent algorithm. In this project, we are going to investigate how can GPUs help training the perceptron and how they are used in minibatch gradient descent algorithm.

The dataset we use in this project is the famous CIFAR-10 dataset, which is a collection of images with handed labels. The whole dataset contains 60,000 32x32 color images in 10 different classes, with 6,000 images in each class. We are going to train our perceptron on this dataset using both a CPU and a GPU and investigate the differences.

2. Mathematical Background

The problem we want to solve using the perceptron is image classification, where given an input x, the perceptron should output the correct label y for this input. Suppose the dimension of input space x is d, and the classifier outputs a vector of probabilities p with dimension K, then we have

$$s = Wx + b$$

$$p = \text{softmax}(s)$$

where the matrix W has size $K \times d$ and the vector b has size $K \times 1$. Function *softmax* is defined as

$$\text{softmax} = \frac{\exp(s)}{1^T \exp(s)}$$

The final label for the input corresponds to the label with highest probability:

$$y = \arg\max_{1 \le k \le K} \{p_1, \ldots, p_k\}$$

The whole procedure of predicting is shown in Figure 1.

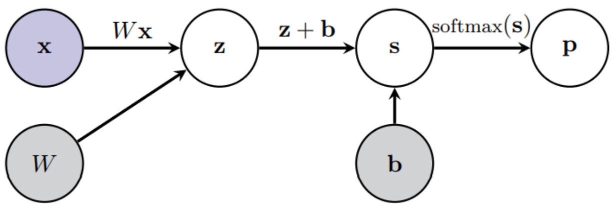

Figure 1. Computational graph of classification

The parameters W and b of our classifier should be learned during training. Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ denote all training data, where $x_i \in \mathbb{R}^d$ and $y_i \in \{1, \dots, K\}$. The loss function we use is cross-entropy loss, and we add a penalty term on W for regularization. The final cost function $\mathcal{J}$ is

$$\mathcal{J}(\mathcal{D}, \lambda, W, b) = \frac{1}{\mathcal{D}} \sum_{(x,y)\in\mathcal{D}} \mathrm{l}_{\text{cross}}(x, y, W, b) + \lambda \sum_{i,j} W_{ij}^2$$

where

$$\mathrm{l}_{\text{cross}}(x, y, W, b) = -\log(p_y)$$

and our final target is to minimize the cost function. The graph for the cost function is shown in Figure 2.

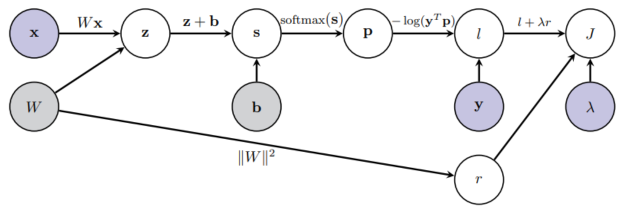

Figure 2. Computational graph for the cost function

We use mini-batch gradient descent algorithm to solve this optimization problem, and our parameters are updated according to the following rules:

$$W^{(t+1)} = W^{(t)} - \eta \frac{\partial \mathcal{J}}{\partial W^{(t)}}$$

$$b^{(t+1)} = b^{(t)} - \eta \frac{\partial \mathcal{J}}{\partial b^{(t)}}$$

where η is the learning rate. The gradients are computed as follows:

$$\frac{\partial \mathcal{J}}{\partial W^{(t)}} = \frac{1}{N}(P - Y)X^T + 2\lambda W$$

$$\frac{\partial \mathcal{J}}{\partial b^{(t)}} = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)$$

where N is the batch size, and P is the probability matrix of batch X evaluated in feed forward process.

3. Implementation

In this section, we are going to introduce different aspects of our implementation, and talk about how we process data, manage memory, create kernels and synchronize all threads during our implementation.

3.1 Data Pre-processing

The CIFAR-10 dataset is designed for Python at first, which can be loaded easily using *pickle* module. In order to load the data in C++, we find a header on the Internet which helps load the CIFAR-10 data into any C++ STL. We then perform data normalization to change all values into the same scale. Weights are also initialized by Gaussian distribution with standard deviation 0.01 and mean 0.

3.2 Memory Management

The language we use is C++, and it is very likely to rely on some libraries and the C++ STL to implement matrix operation. However, these libraries are all unusable in the kernel, and they do not have a multi-thread version either. In this project, we use the most basic one dimension array to represent matrices, and write our own code to operate matrices. Although it is a little bit trivial, we gain the flexibility of manipulating details of all operations.

Memories residing on the GPU mainly consist of three parts: training and validation data, weights and gradients. Instead of using *cudaMalloc()* which requires *cudaMemcpy()* afterwards, we use *cudaMallocmanaged()* in our code, which is suitable for memories accessed by both CPU and GPU. Other memories are some intermediate variables which will be freed soon after they have been used, and we use *cudaMalloc()* to allocate space for them. All memories are initialized using *cudaMemset()* (except training and validation data), and they are all freed using *cudaFree()* after their usage.

We also declare some global variables for special usage. For example, in function *softmax()*, we need to calculate the sum of the exponential of a row, and we need a global variable to store the sum. Note we use *atomicAdd()* here since many threads may perform the addition at the same time, and we need to ensure the atomic of the operation.

```
__device__ double sum[BLOCK_DIM];
for (int j = index_x; j < DIM_OUTPUT; j += stride_x){
    ...
    atomicAdd(&sum[index_y], exponential);
}
```

3.3 Kernels

In total we have three kernels in our code, and they are assigned into the default stream since we need to perform them sequentially.

The first kernel is *computeGredient()*, which takes in a batch of training data and computes the gradients of W and b respectively. The second kernel is *updateWeight()*, which updates the weights given the calculated gradient. The last one is *computeCost()*, which computes the value of cost function given current weights. This kernel is not necessary while training, but it gives us some information about the training process.

Most of the parallelization in the code is done as follows:

```
int index_x = blockIdx.x * blockDim.x + threadIdx.x;
int index_y = blockIdx.y * blockDim.y + threadIdx.y;
int stride_x = blockDim.x * gridDim.x;
int stride_y = blockDim.y * gridDim.y;

for (int i = index_y; i < DIM_INPUT; i +=stride_y) {
for (int j = index_x; j < DIM_OUTPUT; j += stride_x){
    int index = i * DIM_OUTPUT + j;
    w[index] -= LEARNING_RATE * gradient->
        grad_W[index];
}
}
```

Since the most common operation is matrix operation, we use a two-dimensional block for each kernel, and we iterate the block on the matrix in two directions. Variables *index_x* and *index_y* indicate the position of the current thread in the block, and variables *stride_x* and *stride_y* indicate the strides in two directions respectively. We then iterate the block using two *for* loops, and variable *index* indicates the index of the element we are operating in the given matrix.

3.4 Thread Synchronization

Unlike writing code on the CPU, we have to deal with synchronization while writing CUDA code on the GPU. Since we have multiple operations on a single matrix, we need to ensure the completion of the current operation when we proceed to the next one.

One way of doing this is to create a kernel for each operation, where we go to the next kernel after we finish the current one. In this project, we take advantage of function *__syncthreads()* to synchronize all operations. However, this function is a block level signal, and this is the reason why we set the grid dimension to one. Following code is an example showing how we use *__syncthreads()* to synchronize different operations in *computeGradient()*.

```
gpu_evaluate(x, w, b, N, p);
__syncthreads();

... // some other operations
__syncthreads();


gpu_dot(x, p, DIM_INPUT, N, DIM_OUTPUT, gradient
    ->grad_W, true);
__syncthreads();
```

Synchronizing the device and the host is rather simple. What we do is to call function *cudaDeviceSynchronize()* every time we initialize a kernel. This will ensure we go to the following host code after finishing executing the device code.

4. Experimental Setup

The CPU version of our code is written in C++, while the GPU version is written in CUDA. We compile our GPU code using the following command:

```
nvcc -arch=sm_61 perceptron_gpu.cu -o
    perceptron_gpu.out
```

The compiler version is 10.2, and the GPU we use is GTX1050, whose architecture is sm_61. Notice that the compilation fails on lower architectures, since we use some features only supported in architectures higher than sm_60.

5. Results

In this section, we will validate our GPU program, and compare the training time between CPU code and GPU code varying the block size of kernels.

5.1 Validation

The simplest way of validating the correctness of our GPU program is to compare the training process and the accuracy on the test set. In all experiments, we use 5,000 training images, 2,000 validation images and 1,000 test images. Configuration of hyperparameters is shown in Table 1:

Hyperparameter	Value
Epoch	40
Batch size	100
Learning rate	0.001
Lambda	0.3

Table 1. Hyperparameters configuration

Figure 3 shows the training process on the CPU, and Figure 4 shows the training process on the GPU with block size (10, 100). We can see they share some common traits such as continuous descending costs and higher validation cost than training cost. The average accuracy of CPU version on test set is 0.346, and the accuracy of GPU version is 0.342. These all validate the correctness of our code.

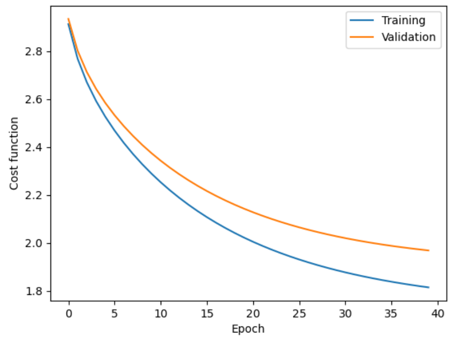

Figure 3. Training on the CPU

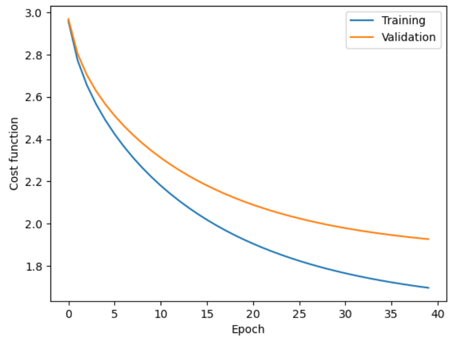

Figure 4. Training on the GPU

5.2 Performance

In this section we investigate the performance improvement when training with GPU in two scenarios: with cost computation and without cost computation. We will increase the block size from (10, 10) to (10, 100) and find out the influence on their performance.

We run each configuration five times and take the average of the training time. Figure 5 shows the performance of our code in different scenarios.

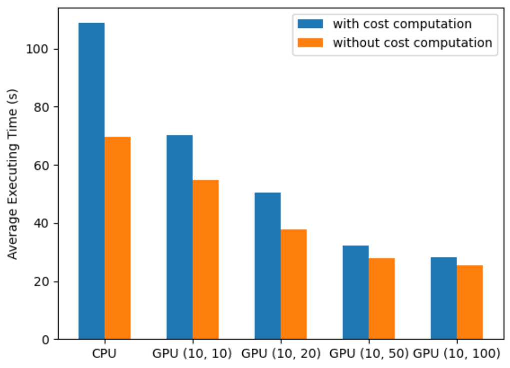

Figure 5. Performance of CPU code and GPU code varying block size in two scenarios

In both scenarios when we train with and without cost computation, we observe performance improvement when we adopt parallelism. However, the improvement does not reach our expectation that the GPU code runs ten times quicker than the CPU code. When the block size is (10, 10), the GPU code has 100 threads to perform the same task, which in our expectation should run at least ten times faster than the CPU code which has only one thread. But the result is that the GPU code only runs roughly 1.5 times quicker, and the overhead is much bigger than we expect. The reason for this does not lie in idle threads, since the batch size is 100, and each thread has tasks to do. Two obvious reasons for this are the memory transfer overhead between the CPU and the GPU and the synchronization among threads. Another hypothesis is that threads in the GPU run slower than the single thread in the CPU.

When we increase the block size, we gain further performance improvement. Still, we do not get the improvement we expect when we double the number of threads in the block. Instead the improvement gets smaller and smaller when the number of threads increases. At last, we only have 10% improvement when we increase the block size from (10, 50) to (10, 100). We can be more certain that the overhead such as memory transfer becomes a bottleneck of our application when the block size is big enough.

We also measure the accuracy on the test set under different configurations, which is shown in Table 2.

Configuration	With cost	Without cost
CPU	0.3460	0.3472
GPU (10, 10)	0.3466	0.3432
GPU (10, 20)	0.3432	0.3408
GPU (10, 50)	0.3400	0.3442
GPU (10, 100)	0.3422	0.3412

Table 2. Accuracy on the test set under different configurations

The accuracy remains roughly the same and they are not affected by configurations, and the slight differences are caused by weight initialization.

6. Conclusion

In this project, we investigate performance improvement of training a perceptron using GPU, and how the block size affects the performance. We find out that the improvement of increasing threads is limited, and the performance is restricted by other overheads such as memory transfer.

Further improvements can be done in two aspects. On the one hand, we can optimize the efficiency of the GPU code, where we can use better strategies for synchronizing all threads. On the other hand, we need to decrease the overhead of the program, for example, use shared memory to speed up the memory access.

References

F. Rosenblatt. *The Perceptron: A Perceiving and Recognizing Automaton.* Cornell Aeronautical Laboratory, 1957.