MNIST Digit Classification using a GPU-accelerated Neural Network

Jonathan Yin, Oliver Ye

Introduction

Although neural networks were first introduced in the 1950s, their widespread impact has only been realized in the past decade. This relatively recent revolution in deep learning has come from an exponential growth in available data as well as significant gains in computing power, as opposed to any particular algorithmic advances. Without specialized hardware like GPUs, training these massive models would be entirely infeasible. In this paper, we explore the use of GPUs to accelerate the training of a simple neural network and compare it against the performance from training on a CPU. We evaluate performance by measuring the training time of models tasked with classifying handwritten digits from the MNIST dataset. While many parallelization techniques exist for accelerating neural networks, we focus on parallelizing the computation in the forward pass stage and backpropagation stage. Since such parallelization techniques are standard for large deep-learning frameworks like TensorFlow and PyTorch, we expect to see notable improvements in runtime performance, especially as our model size increases.

Background on Neural Networks

At a high level, neural networks consist of layers that perform computations passed on to the next layer. Each layer has weights associated with it, which are used to compute the outputs for that layer. The output for a single node of a layer can be understood as a weighted sum of the layer inputs and the weights associated with that output node. In aggregate, the output for an entire layer can be seen as a matrix multiplication between the weight matrix and the input

matrix. In addition, activation functions are applied on top of the outputs from a layer to provide some non-linearity, which allows neural networks to learn more complicated functions.

Training a neural network is largely broken down into two phases: forward propagation and backpropagation. In the forward propagation phase, input data is passed into the network and each layer computes the input for the next layer until a final output is computed. In the backpropagation phase, we compute the gradients of the error function with respect to the weights of the network. This is done backward, layer by layer, starting with the output layer. Like in forward propagation, much of the gradient computation can be represented as matrix multiplication. After the gradients for each layer are computed, we apply gradient descent to update the associated weights.

Matrix operations are responsible for the majority of computations when training a neural network. Thus, by parallelizing this computation on a GPU, a significant portion of the computation time will be reduced, resulting in very significant speed-ups.

Neural Network to Classify MNIST Digits:

The MNIST data set consists of 60,000 grayscale 28×28 pixel images of handwritten digits. Each image has an associated label, indicating the correct digit. In our model, we pass in the image data as a flattened vector of length 784. To classify the digit, we output a length 10 vector representing each digit's predicted probability. For simplicity, we use a 2-layer network, consisting of a hidden layer of size variable size and an output layer of size 10. We apply a ReLU activation on the hidden layer and a softmax activation for the output layer. For our loss function, we use cross-entropy. We train our model using 37800 training images and evaluate our prediction accuracy using 4200 images.

In our experiments, we measure the training time of each model for 5 epochs. We vary the size of the hidden layer to investigate how the size of the model influences runtime and speedup from using a GPU. We also vary the batch size, which is how much training data is fed into the model each time before performing a weight update based on the error of the batch. We hold all other variables except the learning rate constant. However, note that the learning rate has no impact on the training time, only on model convergence. We also record the trained model accuracy on the test data to verify that the models are indeed learning. In terms of evaluating performance, we are only interested in the training time rather than test accuracy. We expect that using larger hidden layer sizes and larger batch sizes correspond to worse test accuracy. This is because we only make 5 passes through the training data, so a larger model will be less trained and a larger batch size results in fewer weight updates. In real-world applications, we would simply train for a greater number of epochs to improve the performance of larger models. We do not increase epochs here for the sake of comparability in terms of training time.

Serial Implementation:

We implemented the simple neural network described above using C++. We have two linear layers (which represent the two-layered Neural Network described above), one ReLU layer for thresholding, and one softmax layer for normalizing probabilities between 0 and 1. We then backpropagate through all the layers adjusting the weights based on errors. Here, we measure the un-accelerated performance by running on a CPU. We gather data for hidden layer sizes 32, 64, 128, and 256 and batch sizes 32, 64, and 128.

Batch size 32				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	53.7502	101.854	298.495	641.182
Forward propagation time (s)	25.547	48.2244	142.924	305.355
Backpropagation time (s)	28.1886	53.6147	155.556	335.795
Test accuracy	0.924141	0.913168	0.903149	0.883826

Batch size 64				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	52.4897	102.788	292.312	640.129
Forward propagation time (s)	24.8217	48.7706	140.632	305.177
Backpropagation time (s)	27.6457	53.9888	151.656	334.922
Test accuracy	0.921994	0.918177	0.908397	0.890983

Batch size 128				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	52.5417	117.589	294.443	640.037
Forward propagation time (s)	24.8378	57.6774	141.233	305.181
Backpropagation time (s)	27.689	59.8883	153.194	334.829
Test accuracy	0.926527	0.919132	0.905773	0.884303

As expected, larger hidden layers result in greater training time. We observed roughly a 2x increase in training time when the hidden layer size increased by a factor of 2. In terms of time spent computing the forward propagation and time spent on backpropagation, we see that those times also increased by a factor of 2 when the hidden layer size increased. The forward propagation time is roughly around the same as the backpropagation time. However, as the hidden layers scale up, the backpropagation time begins to scale up as well. Furthermore, we see a slightly bigger jump from 128 nodes → 256 nodes in training time. This could possibly be due to some computation that does not scale the same way when the hidden layer size increases, as we see similar types of scaling in further implementations. As a sanity check, we see that all test accuracies are all around 88%, with some slight variation. Increasing the batch size also did not affect the training time, which was expected. In the serial version, the same amount of computation is done within 5 epochs, independent of the batch size. The batch size simply reorganizes the chunk size of the work. We do not expect this to be the case in the parallel version of the code, however.

Naive Parallel Implementation:

After verifying that the serial implementation worked, we parallelized the matrix computations for forward and backward propagation. For the sake of comparison of performance but also a fast iteration of code, we decided to implement a relatively naive parallelization scheme for this version of the Neural Network. The two main features we made use of from CUDA were

1. The Unified Memory Model, which allowed us to place arrays in a unified address space that was accessible from CPU and GPU space

2. Only L1 cache usage with no managed shared memory among blocks.

Thus, we traded code complexity for performance for our first implementation of parallel CUDA code. And further, in this report we'll look to optimize these two features out of the naive implementation of code.

Similar to the serial testing, we gathered data for hidden sizes of 32, 64, 128, 256 sizes and batch sizes of 32, 64, and 128. We expected similar levels of scaling across nodes. The multiples of 32 make it conducive to CUDA warps. Throughout all parallel implementations, we used a 32x32 block, and whatever necessary grid was needed for different neural network operations.

Batch size 32				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	4.08057	4.20267	4.70656	6.62628
Forward propagation time (s)	1.85931	1.85233	1.96686	2.29949
Backpropagation time (s)	1.97359	2.11107	2.50106	4.06278
Test accuracy	0.926765	0.911498	0.902433	0.889313

Batch size 64				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	2.16029	2.26497	2.82669	4.60949
Forward propagation time (s)	0.978487	0.987802	1.0069	1.45947
Backpropagation time (s)	1.04824	1.14375	1.68536	3.00794

Test accuracy	0.92605	0.920086	0.877604	0.859375

Batch size 128				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	1.13883	1.4589	2.08266	3.7738
Forward propagation time (s)	0.506824	0.530228	0.663619	1.11407
Backpropagation time (s)	0.555735	0.850573	1.34303	2.58632
Test accuracy	0.89607	0.866951	0.848485	0.763494

We observed a decrease in training time by a factor of 13 for the smallest model and batch size and a factor of over 150 for the largest model with the largest batch size. Specifically, for nodes 32, 64, and 128, we observe that there is very little increase in total training time, which differs from the serial scaling of hidden layers. We suspect this is because for greater hidden layers, we allocate bigger grids which allows us to utilize more CUDA streaming processors. Perhaps we lose this performance boost when going from 128 → 256 nodes because now we're bottlenecked by the total CUDA streaming processors available to us and more warps need to be scheduled concurrently now by the instruction dispatcher. Otherwise, we observe similar training accuracies as well as similar breakdown between forward propagation and backwards propagation. Increasing batch size also decreases training time by a factor of roughly 2, which is expected because each time we increase the batch size, twice the amount of data is being processed through the model in parallel. In contrast, in the serial version, larger batch sizes would still have to be processed serially, so no speed-up gains would be observed.

Parallel Optimization #1: Unified Memory Optimization:

The first aspect of the naive implementation that we attempted to optimize was memory usage. Specifically, after researching online, we found that people were able to find speed ups from moving from unified memory (cudaMallocManaged) to managed memory (cudaMalloc & cudaMemCpy). We maintained the same testing structure from before and found that there was no speed up despite swapping from unified memory to managed memory.

Batch size 32				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	3.86961	3.99109	4.63106	5.50236
Forward propagation time (s)	1.87352	1.87325	2.06445	2.26276
Backpropagation time (s)	1.97332	2.0951	2.54402	3.21667
Test accuracy	0.924141	0.857347	0.794609	0.72376

Batch size 64				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	2.0507	2.01787	2.3057	2.84771
Forward propagation time (s)	0.994761	0.971128	1.01114	1.14148
Backpropagation time (s)	1.04451	1.01875	1.28303	1.69464
Test accuracy	0.923187	0.881629	0.72822	0.544508

Batch size 128

Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	1.09844	1.01638	2.01367	2.31638
Forward propagation time (s)	0.538116	0.478708	0.675377	0.756874
Backpropagation time (s)	0.554768	0.531796	1.33255	1.55173
Test accuracy	0.841856	0.779119	0.699337	0.316051

Overall, we see a slight improvement in training time ~10% performance boost. However, we also got a stark decrease in training accuracy as the hidden layers scaled. This is perhaps due to the fact that when using unified memory, for particular arrays we initialized the values to 0 in the CPU address space. However, as we moved to device memory, any undefined reference that we may not have expected would hit memory that may not have been initialized to 0, hence throwing off the model.

The reason we didn't receive any speed up was because the CUDA unified memory model relies on lazy allocation of pages between CPU memory space and GPU memory space. Furthermore, for any data, only one page of data was kept either in CPU memory space or the GPU memory space at one time. This corrected our previous understanding of unified memory space, where we assumed that two copies of where existed in both CPU and GPU memory spaces — thus maintaining both copies whether it was with write-through or write-back calls would incur a cost performance.

For our parallel implementation, between kernel launches everything remained within GPU code. Even after all the code ran, the only relevant information that we needed back in the CPU memory space were the weights of the linear layers. Thus, the Memcpy from GPU code back to CPU code would only happen when we extracted the weights out of the GPU when

running the Neural Network test with CPU code. Thus, no additional performance of writing back to the CPU was incurred during the actual training process hence leading to no additional performance from this optimization.

Parallel Optimization #2: Shared Memory and Tiling:

The approach for this optimization was very similar to Task 2 in Pset 4. In fact, since linear forwarding was just a matrix multiplication, we were able to re-use the tiling code from Pset a4 for that function. This was able to improve our forward propagation time. We also tinkered with using similar methods for parallelizing the softmax, ReLU, and backpropagation methods. But for reliable results, we decided only to run tests on the tiling method on the forward propagation of our neural network.

Batch size 32				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	3.18578	3.14709	3.48614	4.1886
Forward propagation time (s)	0.842789	0.841156	0.909431	0.879057
Backpropagation time (s)	2.04593	2.00311	2.26113	3.05697
Test accuracy	0.924141	0.912929	0.903149	0.88645

Batch size 64				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	1.43178	1.50934	2.01402	3.3077
Forward propagation time (s)	0.378923	0.39256	0.410312	0.583677

Backpropagation time (s)	0.929922	0.986995	1.47011	2.58314
Test accuracy	0.927242	0.913884	0.886364	0.858191

Batch size 128				
Hidden layer size	32 nodes	64 nodes	128 nodes	256 nodes
Training time (s)	0.947403	1.01155	1.64613	2.89612
Forward propagation time (s)	0.240763	0.20776	0.288108	0.408085
Backpropagation time (s)	0.612559	0.720308	1.27084	2.40406
Test accuracy	0.895833	0.877604	0.84517	0.776042

From the tiling method, we see a great decrease in the forward propagation time, going from 1.85s to 0.84s which is more than a 50% decrease in time for the smallest model and batch size. For the largest model and batch size, our time decreased from 1.11s to 0.41s, which is also more than 50% decrease in time. Compared to the serial training time for the largest model and batch size, we observe a nearly 220 times speed up. The reasoning behind this is identical to our previous problem set where by using shared memory we are able to explicitly utilize the L1 cache for common rows and column elements.

Although not shown above, our next step would be to get reliable results for applying the tiling method on backwards propagation. For our simple neural network, this boils down to being able to apply a similar GPU matrix multiplication tiling operation to the transpose of matrices. Furthermore, we also tinkered with parallelizing the softmax and ReLU layers but there wasn't too big of a timing difference. Thus we concluded that the main bottlenecks existed in the linear

layers. For forward propagation it was the linear_update_forward() method (that we tiled above) and for back propagation it would be the linear_update_gpu() method (that we didn't tile thus leading to much greater back propagation times above)

Discusssion:

Since many neural network operations involve matrix multiplication, being able to effectively and efficiently utilize the GPU becomes imperative in speeding up training time. As evident from the results above, we were able to achieve speedups on the order of 2 magnitudes. The scale-up factor also seemed relatively linear for smaller models and smaller batch sizes. While performance did take a hit for larger models and larger batch sizes, this could easily be remedied by training for more time.

Overall, there are still numerous places where we can improve the parallelization of the neural network. First and foremost, more parallel optimization could still be made to the activation layers and backpropagation process. Moreover, throughout the test runs, we only vary the hidden layer sizes and batch sizes. For the future, we would consider varying the number epochs as well as block sizes to see the effects on performance. Specifically, by varying epochs, we're hoping that we would be able to investigate the accuracy vs. training time trade offs for larger vs. smaller models. By varying block sizes, we would experiment whether smaller block sizes would lead to better performances by combining more tasks within a single warp in CUDA — hence better utilizing the GPU.

File Structure Overview:

Overview:

- The Serial directory contains Serial code
- The GPU directory contains the naive GPU implementation
- The GPU_mem directory contains the first unified memory optimizations
- The GPU_opt directory contains the second shared_memory optimization with tiling.

Results Files:

- NN_CPU-final.out: Output for Serial code
- NN_GPU-final.out: Output for Parallel naive code
- NN_GPU_mem-final.out: Output for Parallel code optimization #1
- NN_GPU_opt-final.out: Output for Parallel code optimization #2

Scripts:

- Run_cpu.sh: Runs serial cpu code in Serial dir
- Run_gpu.sh: Runs corresponding parallel code in either Parallel, Parallel_mem, or Parallel_opt

Code:

- Load_mnist.cpp: Loads MNIST data into file
- serial.cpp/gpu.cu: Contains the main logic for NN forward and backwards propagation
- Train.txt data.

Note: All of our code is up on this github

How to run: sbatch run_gpu.sh or run_cpu.sh. .

References:

The serial version of the code for the neural network was largely inspired by this tutorial. Although we viewed their parallel code for reference, we quickly realized their implementation was relatively naive. By utilizing shared memory and tiling techniques we learned from class we were able to acheive significantly better performance. Research into unified memory spaces and shared memory were largely based on online forums and research and informed the direction and decision behind what optimizations to adopt.