An Artificial Neural Network for MNIST Digits Using CUDA C

Merlin Carson, Victor Heredia, Sudheer Nair, Son Vu
Department of Computer Science
Portland State University, Portland, OR, USA
Email: {mpc6, victor32, sudheer, vson}@pdx.edu
code repository: NeuralNet_CUDAC

***Abstract*—Many automation tasks require computer vision techniques for image recognition. As Graphical Processing Units have become more cost and power efficient, artificial neural networks have proven to be a more versatile and accurate tool for these types of algorithms than traditional implementations. One of the most prevalent and useful data sets for this type of research is the MNIST handwritten digits.**

Using the C++ standard library, along with NVIDIA's CUDA C, we've developed a fast and accurate artificial neural network for the classification of numeric digits. We demonstrate our classifier's ability to recognize handwritten digits significantly faster and at a similar accuracy to human level performance. We also compare the speedup of our code when run on a GPU vs a CPU.

***Index Terms*—neural network, multi-layer perceptron, MNIST, GPU, accelerated computing, machine learning**

I. Introduction

Handwritten digit classification has been a long researched topic in computer vision applications. A significant amount of traditional algorithms have been developed for the task, but many struggle to handle the variety of inconsistencies between each human's writing style. In 1989, researcher Yann LeCun released a paper [1] demonstrating the usefulness of artificial neural networks for this task. Shortly after the US Postal Service and many banking services began using this type of technology for the automation of recognizing zip codes and numbers on cheques.

Since the majority of mathematics in a neural network's training and application are linear algebra operations, they are highly parallelizable. To investigate the advantages of using Graphical Processing Units (GPUs) for this task, we've developed code for an artificial neural network using a combination of the C++ standard library and NVIDIA's CUDA C.

Using the MNIST handwritten digits data set, research has shown that humans are capable of recognizing the digits with $\sim 99.8\%$ accuracy. However, when they are not given time to study each digit, just quickly shown them, human level performance drops to $\sim 98\%$. Our results show that our implementation not only matches this level of performance, without a significant amount of hypertuning, but is also capable of classifying a single digit in ~ 0.0002 seconds, an impossible feat for a human.

II. Experimental Design

A. Data

The MNIST data set consists of 70,000 28x28 grayscale images of handwritten digits [0,9] and is freely available in a variety of formats. For our project we used a comma seperated value (CSV) format of the training and test data [2], [3] downloaded from Joseph Chet Redmon's site pjreddie.com. We developed a CSV loader from scratch using the C++ 98 standard library and loaded the data into a custom data structure. The training data consists of 60,000 examples and is randomly split into a training set and a validation set, 50,000 and 10,000 examples respectively. The test set consists of another 10,000 examples. The data is normalized, scaled to [0,1], by dividing all features by the maximum grayscale value of 255. This helps normalize the signal through the neural network, keeping it near the center of the activation function where the gradient is strongest. Thus, resulting in better learning, smaller weights, and greater generalization for new examples.

B. Training

During the training phase, the training set is passed through the neural network model in batch sized chunks. This is considered the forward pass and results in a real value for each of the neurons in the output layer. These output values are used to calculate an error between the target output, given the input, and the actual output of the network. This error is backpropagated through the network to update it and is thus called the backward pass.

After the entire training set has been processed, the validation set is passed through the network. An error is determined for the validation process, but the network is not updated. This is how the out-of-sample error is approximated, an estimate of how well the model will perform on examples not used during the training process. This is important to not only determine how well training is going, but to determine when overfitting begins to occur. If a network is overly complex for the problem it is used for or is trained for too long, it will begin to memorize the training data and will not generalize well when predicting with new examples. This is what is referred to as overfitting. The network reaches its best performance when the error on the validation set is minimized. At some point past this, the validation error will begin to increase despite the training error continuing to decrease. This is a sign that the network is overfitting to the training data and at which point training needs to have stopped.

The above process is repeated for some amount of iterations, referred to as epochs. Despite the validation set not being used to update the network, it still has influence on when training should be stopped, thus is considered a biased data set. Therefore, a third data set, the test set, is used to more fairly evaluate the true out-of-sample error for the model once training is complete.

C. Forward Pass

The forward pass consists of two major operations, determining the signal at each neuron in a layer and passing that signal through a non-linearity. This is performed on a layer by layer basis. The signal, denoted z, to an individual neuron is a linear combination of weighted inputs:

$$z = \sum_{i=1}^{num\_inputs} w_{ji} x_i, \tag{1}$$

where j is the jth neuron in the hidden layer. The signal at each neuron is then passed through a non-linear function, σ, thus resulting in its output. While there are a variety of non-linear functions used in artificial neural networks, we have chosen to use the traditional sigmoid activation function:

$$\sigma = \frac{1}{1+e^{-z}}. \tag{2}$$

The signal for the neurons in the output layer is a linear combination of the outputs of the hidden layer and their associated weights to the neurons in the output layer:

$$z = \sum_{i=1}^{hidden\_size} w_{ki} h_i, \tag{3}$$

where k is the kth neuron in the output layer and h_i is the ith neuron in the hidden layer. These signals are also passed through the sigmoid activation function, Eq. (2), resulting in the output values for the network.

D. Backward Pass

In the backward pass, the network is trained and the "learning" takes place. The backward pass is only done on the training data set. In the backward pass we perform the backpropagation algorithm and update the weights based on the calculated error, in preparation for a better result on the next forward pass. By calculating the error and updating the weights the network will start to improve with every update. In order for the network to be able to accomplish this we will need to calculate the error, and update the weights.

E. Error

The error is a function that measures the inaccuracies of the network. It does this by measuring the difference between the calculated output and the expected values, otherwise known as the label of the data. When that data set comes with a label, we call that supervised learning, as it is known what the expected output should be. In our experiment the labels are the integer values representing the digit being processed through the network. When calculating the error, it is possible for the prediction to be larger than the expected output. Likewise it is also possible that the prediction is smaller than the expected output. If we try to take the average of these errors it is possible for positive and negative errors to cancel each other out. Thus, the network would then not make meaningful adjustments. To avoid this we can just measure the distance of the prediction to the expected output. There are many different cost functions that help calculate the error of a neural network. For our experiment we decided to used the mean square error, where we square the difference and get the mean error:

$$C = \frac{1}{2} \sum_{k=1}^{output\_size} (t^k - a^k)^2 \tag{4}$$

We can use the error as an indicator of how well the network is learning. A large error tells us the network performed poorly and needs major tuning. As the objective is to minimize the error, we can use some calculus to help reduce the error. The gradient is the direction of greatest change in the positive direction. As we are trying to minimize the error, we can negatize the gradient to point us in the direction of greatest negative change, helping us reduce the error the fastest.

As this is the start of the backward pass, we have to start by measuring the error from the output layer and work our way back to the input layer. Brace yourself, this is where the math gets bumpy, but we will try to take it one step at a time. In the following derivation k represents a neuron from the output layer, z is the signal to a neural, the output from Eq. (3), and a is the output from the non-linear function Eq. (2), the sigmoid activation.

$$\begin{aligned}
\delta_k &= \frac{\delta C}{\delta z} \\
\delta_k &= \frac{\delta C}{\delta a}\frac{\delta a}{\delta z} \\
\delta_k &= \frac{\delta C}{\delta a}\frac{\delta(\sigma(z))}{\delta z} \\
\delta_k &= \frac{\delta C}{\delta a}(\sigma'(z)) \\
\delta_k &= \frac{\delta C}{\delta a}(\sigma(z))(1-\sigma(z)) \\
\delta_k &= \frac{\delta(\frac{1}{2}\sum(t^k-a^k)^2)}{\delta a}(\sigma(z))(1-\sigma(z)) \\
\delta_k &= (t^k-a^k)(\sigma(z))(1-\sigma(z))
\end{aligned}$$

$$\delta_k = (t^k - a^k)(a)(1-a) \tag{5}$$

Thanks to the calculus chain rule, we have been able to derive the output error, Eq. (5). The error for the other layers in the network don't have a loss function or an expected target they can measure against. As we are trying to perfect the output values from the network, we need the other layers to help in reducing the error we just derived, the output error. This can be done by using the output error to help the other layers correct the weights connecting the layers. In other words, we need to define the error from the previous layer in terms of the error from the current layer. Brace yourself, here we go again. All previously describe variable maintain their meaning. The new guys, l represents the current hidden layer calculating its error, and j represents a neuron from hidden layer l:

$$\delta_j = \frac{\delta C}{\delta z^l}$$
$$\delta_j = \sum_k \left[\frac{\delta C}{\delta z_j^{l+1}} \frac{\delta z_j^{l+1}}{\delta z_j^l} \right]$$
$$\delta_j = \sum_k \left[\delta_k^{l+1} \frac{\delta z_j^{l+1}}{\delta z_j^l} \right]$$
$$\delta_j = \sum_k \left[\delta_k^{l+1} \frac{\delta \left(w_{kj}^{l+1} \sigma(z_j^l) \right)}{\delta z_j^l} \right]$$
$$\delta_j = \sum_k \left[\delta_k^{l+1} \left(w_{kj}^{l+1} \sigma'(z_j^l) \right) \right]$$
$$\delta_j = \sigma'(z_j^l) \sum_k \left[\delta_k^{l+1} \left(w_{kj}^{l+1} \right) \right]$$
$$\delta_j = \sigma(z_j^l)(1 - \sigma(z_j^l)) \sum_k \left[\delta_k^{l+1} \left(w_{kj}^{l+1} \right) \right]$$
$$\delta_j = (a_j)(1 - a_j) \sum_k \left[\delta_k^{l+1} \left(w_{kj}^{l+1} \right) \right] \quad (6)$$

We have now derived the error for layer l in terms of the error from layer $l + 1$, Eq. (6). As we can see the error from the layer in front is used to help define the error for the current layer. This is how the error is propagated backwards through the network.

F. Weights

The arithmetic in the forward pass uses the weights to calculate the signals at the endpoint of the connecting edges. In other words the weight values determine the error of the network. This means the weights determine the error on the forward pass, and the error helps tune the weights on the backwards pass to help reduce the error on the next pass.

It should be stated that reducing the error does not mean reducing the value of the weights. Some weights might be too low, contributing to the magnitude of the error. The error in conjunction with the learning rate helps tune the weights. This means that the error will help lower the value of weights that are too high and raise those that are too low.

The degree to which the weights change is also directly correlated with the degree of the error. A large error will result in a large change in the weights, helping the network learn that much faster. Updating the weights connecting to the output layer is slightly different than updating the other weights. This difference is due to the hidden error using the output error in its calculation. This time we will use B to represent the total batch size, b represents a single batch, η is the learning rate, t represents the number of times the backpropagation algorithm has been calculated in the current epoch, h is the activation from neuron j, calculated in the forward pass for the hidden layer connecting to the output layer, all other variable remain the same as before.

$$\Delta w_{kj}^t = \eta \sum_b^B \delta_k^b h_j^b$$
$$\Delta w_{kj}^t = \eta \delta_k h_j$$

$$w_{kj}^t = w_{kj}^t + \Delta w_{kj}^t \quad (7)$$

As can be seen in the derivation of Eq. (7), we were summing the products of two elements for the entire batch size. That is the definition of the dot product in every sense. We will see the same thing when we update the weights connecting the input layer to the hidden layer. As before j represents a neuron from the hidden layer, the new variables this time are i, which represents a node from the input layer, and x which represent the signal from the i^{th} input feature (pixel value).

$$\Delta w_{ji}^t = \eta \sum_b^B \delta_j^b x_i^b$$
$$\Delta w_{ji}^t = \eta \delta_j x_i$$

$$w_{ji}^t = w_{ji}^t + \Delta w_{ji}^t \quad (8)$$

Thus, with Eq. (8), we've defined the weight update for the edges that connect the inputs to the hidden layer. As stated earlier, each layer has a set of weights that will need to be tuned to get the network to make better predictions in the forward pass. With the math we just covered we can literally see the error propagate backwards through the network making the necessary adjustments.

Now that we outlined the process lets take a look at how our network performed in our experiment. Looking at Fig. 1, the Model Error, we can see the error is the highest after the first forward pass. If we were to use our friend Calculus to help us determine the rate of change on the error in that figure, we can clearly see the error is being lowered significantly in first epochs compared to later epochs. This makes sense as the network is not tuned at all in the beginning. Thanks to the cost function the weights are being tuned significantly early on, helping to refine the error. After the twentieth epoch the error starts to settle. Looking at Fig. 2, the Model Accuracy, we can see that shortly after epoch 20 the accuracy starts to settle down as well. We can see that thanks to the error, the network is able to adjust the weights and result in a higher accuracy. The magic, I mean the learning, happens in the backward pass thanks to the backpropagation algorithm.

G. Momentum

In our experiment we also implemented a momentum hyper-parameter, α. Momentum helps a network by helping the error overcome local minimums and continue to improve. Neural networks, given enough time, might be able to achieve this on their own, but momentum helps nudge them in the right direction, avoiding that additional time. As we can see in Eq. (7) and Eq. (8), the update in weights is scaled using the learning rate and the error that was propagated backwards to the layer and the signal at the current neuron.

Now what if the delta was so small that it really didn't change the weight a useful amount? The network would continue to make similar predictions on the subsequent forward passes, resulting in a similar error, in turn providing marginal deltas for the weights again. This is where the momentum comes to help. By adding the coefficient α to the previous change in weight we will be able to alter the current value of the weight, and get out of that rut, allowing the network the possibility to continue learning in a similar direction that it had in the previous weight update. As stated earlier, given enough time, the network might be able to accumulate enough minor changes and progress, but that is not guaranteed. Lets take a look at our new and improved weight update equations:

$$w_{kj}^{t} = w_{kj}^{t} + \alpha \Delta w_{kj}^{t-1} \tag{9}$$

$$w_{ji}^{t} = w_{ji}^{t} + \alpha \Delta w_{ji}^{t-1} \tag{10}$$

H. Kernels

We have implemented several CUDA kernels for obtaining performance benefits due to the parallel nature of these operations. We did not get a chance to optimize this further by trying out tiling. For each of these kernels we used a $BLOCK\_SIZE = 32$. We could potentially further experiment with a different set of kernel parameters to fine tune the performance. These kernels were built based on those presented in *Programming Massively Parallel Processors* [4].

1) Sigmoid Activation: Each thread calculates the sigmoid activation function (σ), Eq. (2), for the respective neuron, given Y the signal matrix obtained after multiplying the weight matrix by the input matrix.

```
dim3 gridDim((int)ceil((float)numCols /
    BLOCK_WIDTH), (int)ceil((float)numRows /
    BLOCK_WIDTH), 1);
dim3 blockDim(BLOCK_WIDTH, BLOCK_WIDTH, 1);

sigmoid(z) = 1 / (1 + exp(-z));

int row = blockIdx.y * blockDim.y +
    threadIdx.y;
int col = blockIdx.x * blockDim.x +
    threadIdx.x;

int idx = col + row * numCols;
float z = d_Z[idx];
d_Y[idx] = sigmoid(z);
```

2) Element Multiplication: For element multiplication each thread again corresponds to each of the element in the target matrix. If P is the resultant matrix and M and N are the input matrices:

```
int idx = col + row * num_PCols;
d_P[idx] = d_M[idx] * d_N[idx];
```

3) Transpose: The way in which each of the elements are accessed is similar except that the rows and cols need to be swapped in the resultant matrix. If N is the transposed matrix and M the original matrix:

```
int n_idx = colN + rowN * num_NCols;
int m_idx = colM + rowM * num_MCols;
d_N[n_idx] = d_M[m_idx];
```

4) Dot Product: In this case we have each thread corresponding to each of the elements of the resultant matrix P. This thread then accesses the respective elements of the input matrices M and N.

```
if (row < num_PRows && col < num_NCols) {
   float pVal = 0.0;

   // Each thread computes one element of the
       block
   int i, j;
   for (i = 0, j = 0; i < num_NRows && j <
       num_MCols; i++, j++) {
      int m_idx = j + row * num_MCols;
      int n_idx = col + i * num_NCols;
      pVal += d_M[m_idx] * d_N[n_idx];
   }

   d_P[row * num_PCols + col] = pVal;
}
```

5) MSE: This kernel is used to calculate the mean squared error (MSE) between two matrices. The index calculations are slightly different in this kernel compared to the other kernels. In this case $batchId$ is used as an index to access the expected output vector (T). this would give us the expected output for each of the labels, 1 for the neuron that represents the target value and 0s elsewhere. We then synchronize the threads since we have to sum all the values together and the kernel must guarantee all other threads have finished there intermediate calculations for each example in the batch.

```
dim3 gridDim((int)ceil((float)batchSize /
    BLOCK_WIDTH), 1, 1);
dim3 blockDim(BLOCK_WIDTH, 1, 1);

int batchId = blockIdx.x * blockDim.x +
    threadIdx.x;
int t_idx = d_T[batchId];

for (int j = 0; j < numLabels; j++) {
   int o_idx = j + batchId * numLabels;

   if (t_idx == j) {
      // If this is the same as the expected
         output
      float diff = 1 - d_O[o_idx];
      err += diff * diff;
   }
   else {
      float diff = d_O[o_idx];
      err += diff * diff;
   }
}
d_sampleSquareErr[batchId] = err;

__syncthreads();

// Calculate the square error for the batch

// Need only one thread to do this
if (batchId == 0) {
   for (int i = 0; i < batchSize; i++) {
      *batchLoss += d_sampleSquareErr[i];
   }
   *batchLoss /= (float)2;
   *batchLoss /= (float)batchSize;
}
```

6) Batch Prediction: This kernel is used for generating the output predictions for a batch. The digit that is predicted is the argmax of the output neurons. This is parallelized by each example represented in the batch.

```
dim3 gridDim((int)ceil((float)activation_size
    / BLOCK_WIDTH), (int)ceil((float) b_size
    / BLOCK_WIDTH), 1);
dim3 blockDim(BLOCK_WIDTH, BLOCK_WIDTH, 1);

for(int i = 1; i < output_size; ++i)
{
   int idx = i + row*output_size;
   if(out_activations[idx] > maxValue)
   {
      maxValue = out_activations[idx];
      counter = i;
   }
}
batch[row] = counter;
```

7) Output Error: This kernel is used for calculating error at the output layer for the backward propagation, as seen in Eq. (5).

```
dim3
    dimGrid(ceil((float)BATCH_SIZE/BLOCK_WIDTH),
    ceil((float)NUM_LABELS/BLOCK_WIDTH), 1);
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);

if(t[r] == c)
   d_error[index] = d_out_layer[index] * (1 -
       d_out_layer[index]) * (1 -
       d_out_layer[index]);
else
   d_error[index] = d_out_layer[index] * (1 -
       d_out_layer[index]) * (0 -
       d_out_layer[index]);
```

8) Hidden Error: This kernel is used for calculating error at the hidden layers during the backward propagation, as seen in Eq. (6).

```
dim3
    dimGrid(ceil((float)HIDDEN_SIZE/BLOCK_WIDTH),
    ceil((float)BATCH_SIZE/BLOCK_WIDTH), 1);
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);

int index = r*Cols + c;
d_error[index] = d_hidden_layer[index] * (1 -
    d_hidden_layer[index]) * (d_dotP[index]);
```

9) Update Weights: This kernel is used for updating weights during the backward propagation, as seen in Eq. (9) and Eq. (10).

```
int index = r*Cols + c;
d_delta_weights[index] = eta *
    d_dotP[index]/BATCH_SIZE + alpha *
    d_delta_weights[index];
d_w[index] += d_delta_weights[index];
```

10) Scalar Multiplication: This kernel is used for multiplying a matrix with a scalar.

```
M[r*Cols + c] *= scalar;
```

11) Unit Tests: We also implemented host versions of all these kernels and wrote unit tests that ran calculations using these host functions and compared them against the kernel functions. This helped us in testing out the kernel functions before they were actually used in forward and backward propagation. This also allowed us to compare the performance of the network on the CPU vs. the GPU.

III. Results

The mean square error (Fig. 1) and accuracy of the trained MNIST model (Fig. 2) is recorded after each epoch to determine how well the training is doing and if any parameters need tuning. The error, accuracy, and confusion matrix seen in Fig. 1, Fig. 2, and Fig. 3 respectively uses a batch size of 1. The network was trained using a learning rate of 0.1 and 300 hidden nodes in the single hidden layer.

As seen from Fig. 1, as the epoch increases both the training and validation errors start decreasing and levels out. The validation error did not increase after leveling out, indicating the model is not over-fitted after running for 120 epochs. From Fig. 2, we can also see that as the epoch for the training increases, the accuracy also starts to improve and levels out at $\sim 98.12\%$. Since the model gives accuracy similar to what a human can do without over-fitting, we believe the trained network has good performance for the MNIST digits.

After confirming our code provides an accurate model for MNIST digits, we also wanted to compare execution speed for training the model between CPU and GPU. For this goal we used the host code, written for our unit tests, and the CUDA

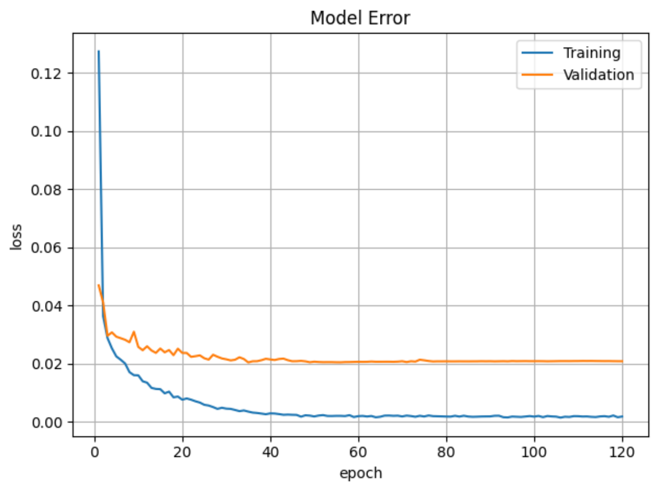

Fig. 1: Training and validation loss for each epoch during the training phase.

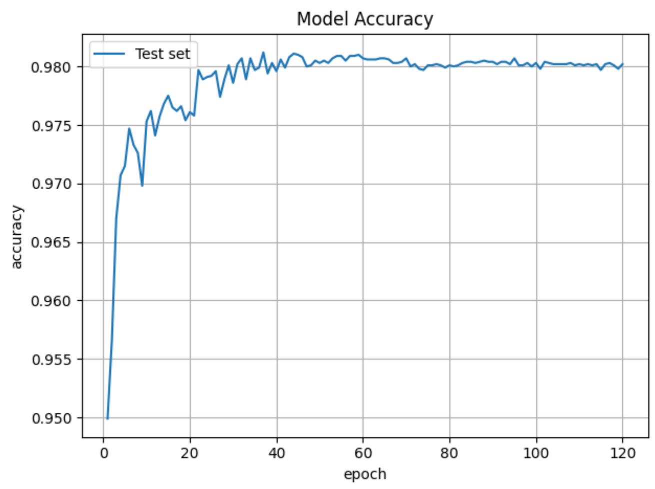

Fig. 2: Test set accuracy for each epoch during the training phase.

TABLE I: Processing time on different devices using different batch sizes.

Device Type	Time Per Epoch		
	Batch Size 1	Batch Size 10	Batch Size 64
CPU	$252s$	$110s$	$100s$
GPU	$267s$	$28.2s$	$7.07s$

kernels for both forward pass and backward pass so that we can train the network using either the CPU or the GPU.

Table I shows the execution time of a single epoch using different batch sizes for both CPU and GPU. CPU executes 1 epoch slightly quicker than GPU with a batch size of 1 because of the overhead of copying data back and forth between GPU and and CPU when using CUDA global memory. We see a significant decrease in execution time for the GPU as the batch size increases, while only a modest decrease in execution time on the CPU. As we increase the batch size, more data can be executed in parallel which is why we see such a significant

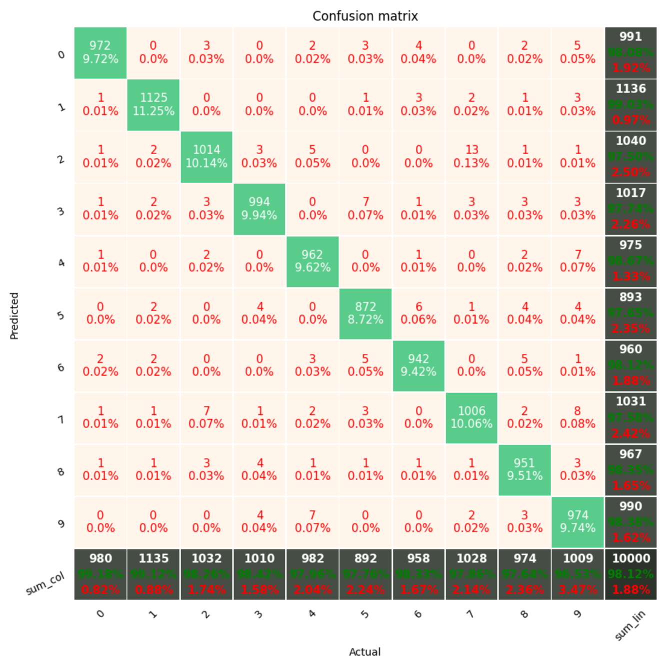

Fig. 3: Confusion matrix of predictions for epoch with the highest accuracy.

decrease in execution time when we use the GPU for training the network. When the batch size increases, the CPU cannot take advantage of parallelization, which is why we don't see as nearly a significant decrease in execution time as with the GPU. However, with batch processing, weights are only updated once per-batch, thus despite the CPU sequentially processing the batch, the backward pass is processed $\frac{1}{batch\_size}$ as often, which is why there is still a decrease in execution time for the CPU.

A confusion matrix is a table used to visualize the performance of the trained model on each class (digits [0,9]). The rows represents the instances of the predicted classification, while the columns represent the target classification. The confusion matrix in Fig. 3 shows the performance of the trained model using a batch size of 1 with a hidden layer containing 300 neurons. We can see that the trained model gets most of the classifications correct by the huge counts along the diagonal of the matrix, where the predictions intersect the targets. We can also see that the model has the most mistakes (13) for a predicted classification of 2 with a target classification of 7. This makes sense since a handwritten 2 is very similar to 7, even to the human eye. The confusion matrix also gives the precise accuracy of the trained model at the bottom right with a value of $\sim 98.12\%$.

IV. Conclusion

We have found that with a batch size of one performance time between the sequential processing on the CPU is nearly identical to the parallel processing on the GPU, due to the overhead of moving data to and from the GPUs global memory. However, as the batch size increases a significant more amount of data can be processed concurrently, taking advantage of parallelization on the GPU, resulting in a seemingly exponential increase in performance. Therefore, we have found that programming an artificial neural network from scratch using CUDA C can significantly boost the performance over a CPU bound implementation. Our neural network model is capable of matching human level performance when classifying MNIST digits and can predict at a significantly higher rate.

V. Future Work

Given the nature of machine learning algorithms, artificial neural networks have a significant amount of tunable parameters which have a direct affect on the accuracy and efficiency of training and testing. These parameters are known as hyperparameters and determine the complexity of the model. Our network is parameterized by the number of neurons in the hidden layer, the number of examples (batch size) that is passed through the network between each weight update, the learning rate η, the momentum coeffecient α and the number of epochs to train for. Ideally a grid search of these hyperparameters would be performed to determine the best possible performance that can be obtained. However, to fully train our network to accurately classify the MNIST digits takes several hours, and with our time constraint for this project we were only able to try a very small subset of parameter value combinations. Future work would involve a more in-depth search for these hyperparameters.

To get the best performance results for our artificial neural network code, we created kernels for nearly all mathematically operations. However, given the significant number of kernels we implemented, there was not enough time to go through and optimize each one. Future work would focus on implementing more efficient version of the kernels, including tiled matrix operations, and better utilization of shared memory and memory coalescing.

Acknowledgement

We would like to thank Dr. Karavanic for her instruction on programming device kernels using CUDA C. We would also like to thank professor Anthony Rhodes for the use of his slides for the implementation details of the multi-layer perceptron more commonly referred to as an artificial neural network.

References

[1] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, “Handwritten digit recognition with a back-propagation network,” in *NIPS*, 1989.

[2] N. I. of Standards and Technology. MNIST handwritten digits training set in CSV format. [Online]. Available: http://www.pjreddie.com/media/files/mnist_train.csv

[3] ——. MNIST handwritten digits testing set in CSV format. [Online]. Available: http://www.pjreddie.com/media/files/mnist_test.csv

[4] W. mei W. Hwu and D. B. Kirk, *Programming Massively Parallel Processors, 3rd Edition*. Morgan Kaufmann, 2016.