Classification Using A Parallelized Multilayer Perceptron

Gamini Udawat
Prableen Kaur
Nikki Kyllonen
Shruti Verma

1. INTRODUCTION

Artificial Neural Networks have generated a lot of excitement in Machine Learning research and industry, thanks to many breakthrough results in speech recognition, computer vision and text processing. The most basic form of an Artificial Neural Network is the Multilayer Perceptron(MLP).

Speed has always been the Achilles heel of Artificial Neural Networks. For many problems, MLPs are too slow to provide viable solutions. However, these algorithms are highly parallelizable and modern-day graphics processing units (GPUs) can offer tremendous speed gains at relatively low cost.

An MLP consists of, at least, three layers of nodes: an input layer, a hidden layer and an output layer (Fig. 1). Except for the input nodes, each node is a neuron that uses a nonlinear activation function. The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives inputs from some other nodes, or from an external source and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function $f()$ to the weighted sum of its inputs. The function $f()$ is an activation function and this introduces nonlinearity into the output of a neuron. This is important because most real world data is non linear and we want neurons to learn these non linear representations. There are several activation functions such as, Sigmoid, ReLu, etcetera.

The input nodes provide information from the outside world to the network and are together referred to as the Input Layer. No computation is performed in any of the Input nodes - they just pass on the information to the hidden nodes. The Hidden nodes have no direct connection with the outside world (hence the name "hidden"). They perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a "Hidden Layer". While an MLP will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers. The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world. MLP utilizes a supervised learning technique called backpropagation for training. We make use of the stochastic gradient descent algorithm to implement the backpropogation for training.

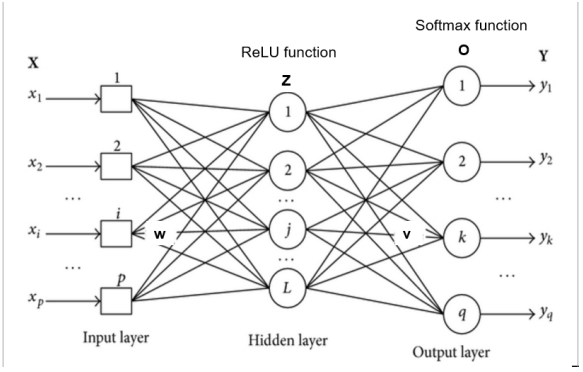

Figure 1: Multilayer Perceptron for Classification

For our project, we implement a MLP to perform classification. We made use of MNIST data-set: a data-set of handwritten digits that consists of 60,000 rows where each row is a 28x28 image. We compare the performance of both, the sequential implementation and the parallel implementation of the algorithm.

2. DESIGN OVERVIEW

2.1 Structure of the Data Set

We are classifying the digits in the MNIST dataset into one of the ten digits (0,1,2,...9) using the Multilayer Perceptron described above. The MNIST dataset consists of 60,000 images of 784 features each. This is stored in form of a matrix of 60,000 rows, each row corresponding to an individual image. The matrix consists of 784 columns, where the columns corresponds to the features of the image. Therefore, each row of the matrix is an individual image with its corresponding 784 features. The aim of this project is to train the multilayer perceptron model based on the image features to classify images into ten classes or labels. These 10 classes correspond to the digits 0,1,2,.....9.

2.2 Structure of the Multilayer Perceptron Model

In the Multilayer Perceptron model we use for classification there are three layers. The first is the input layer, which consists of 784 nodes corresponding to the 784 features for

each image. The input layer has weights **w** associated with it. The middle layer is the hidden layer which consists of 10 hidden nodes. The hidden layer has weights **v** associated with it. The third layer is the output layer which has 10 nodes, corresponding to the 10 classes that the input can be classified into.

2.3 Data Flow - Forward Propagation

An image **x** is fed into the multilayer perceptron at the input layer. Each feature of the image corresponds to each node of the input layer. The input is then multiplied by the weights (**w**) corresponding to the input layer and is passed through a non linear activation function. The non linear activation function that is used in this model is called ReLU. ReLU stand for Rectified Linear Unit and is defined as:

$$f(x) = \max(0, x)$$

The output generated by the input layer after being passed through the ReLU function is called **z**. This output **z** is then passed through the hidden layer, where it is multiplied by the corresponding weights (**v**) of the hidden layer. Then, the result is passed through a Softmax function. Softmax function is regularly used in multiclass classification in neural networks to map the non-normalized output to probability distribution over predicted output classes. Essentially, before applying Softmax function, some vectors could be negative, greater than one or may not sum up to 1, but after applying the Softmax function each element is in interval $[0, 1]$ and $\sum_i x_i = 1$. The Softmax function is defined by

$$\sum \mathbf{y}_i = \frac{e_i^y}{\sum e_i^y}$$

.

The output generated by the hidden layer after being passed through the Softmax layer is the final predicted output **y**.

2.4 Data Flow - Backpropagation

The error between the true labels and the predicted labels is backpropagated to update the weights of the hidden layer (v) and the input layer (w). Backpropagation basically uses Stochastic Gradient Descent Algorithm to calculate the gradient which is used to further update the weights of the hidden and input layers. The update equations used are described in detail later in the Backpropagation section of the paper.

2.5 Iterations

The process of iteration is used to find the set of optimum weights in the Neural Network by reducing the error repeatedly at each iteration. The forward propagation and back propagation process described above is repeated for each image in the data-set. It is repeated 60,000 times for the MNIST data set used in the experiment. Following this, the process is repeated over the entire data-set for EPOCH number of times to reduce the error and optimize the weight values. In this experiment the value of EPOCH was set to be 10.

3. IMPLEMENTATION

3.1 Overall Flow and Design

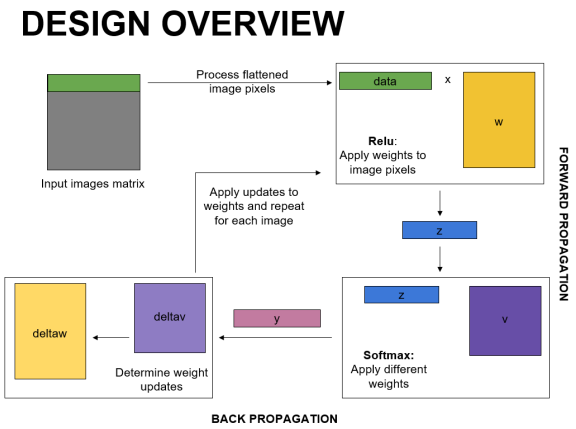

Figure 2: Design Overview of Multilayer Perceptron used for Classification

Our design has minimal Device to Host and Host to Device transfers. After the initial transferring of the Host data loaded from our data files, we initialize four more float arrays. However, these float arrays are allocated and filled directly into the Device global memory, therefore eliminating the need for more Host to Device transfers. Once all of these arrays have been loaded into device global memory, our algorithm begins operating upon this data. Once it is completed, the final computed values are then transferred from Device global memory to the Host for accuracy checking.

All of the data used within our program are one-dimensional arrays of floats, even the matrices. Upon input, each matrix is flattened into a one-dimensional array holding the original row values consecutively in memory.

3.2 Matrix Multiplication Kernels

The first two kernels that operate on our device data are vector-matrix multiplication kernels. The multiplication algorithm itself is the same, however different functions are applied to the resulting vector values, therefore two kernels were needed. The first kernel uses the Relu function.

As described earlier, before either of these functions are applied, a parallelized vector-matrix multiplication is computed. The parameters passed into each function are: an input vector of floats, a matrix of floats, and an output vector of floats. For each kernel, a single thread is assigned to each element of the input matrix. Each thread then computes the product between its matrix element and the input vector element at the corresponding row value. This multiplication is shown below in Figure 3. Initially, these products were then directly summed into the global output vector using an *atomicAdd*(). However, to reduce the number of collisions and to reduce the number of global memory reads and writes, a local output vector was added to shared memory. With this shared memory vector, each thread in a block uses an *atomicAdd*() to create a vector of partial sums. These partial sums are then atomically added to the global output vector by the first N threads, with N being the length of the output vector. Another optimization applied to both of the kernels is thread coalescing. Since the matrices are flattened, this coalescing is easily achieved using one-dimensional thread blocks that cover the entire length

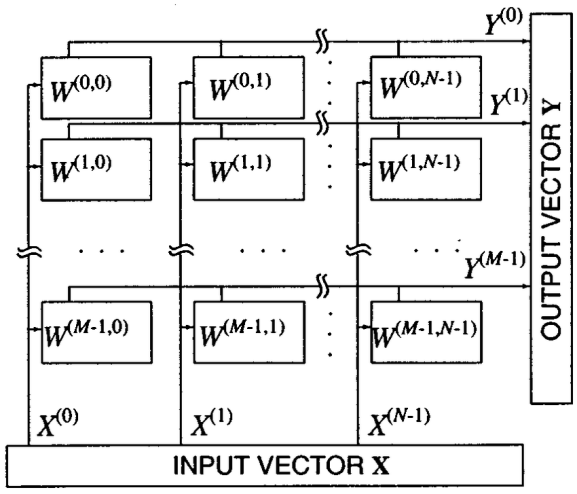

Figure 3: Graphical representation of parallel vector matrix multiplication. In this example, the vector X is applied to two-dimensional matrix W to produce the resultant vector Y. [2]

of the matrix.

For the Relu kernel ($ReluMulKernel()$), the inner dimension of the matrix matches the length of the input vector, therefore no transposition is needed. The product sums in the output vector are then immediately passed through the Relu function, which, as shown above, removes all negative values and sets them to zero. However, this procedure requires a way to determine which index value is negative. In our implementation, an if-statement is used. This lead to the Relu kernel having added thread divergence as only some threads will need to change their corresponding values.

For the Softmax kernel ($SoftmaxMulKernel()$), the inner dimensions of the matrix and the input vector do not match, therefore the matrix must be transposed. In order to avoid waiting for the entire matrix to be transposed, however, the kernel accounts for this by adjusting which input vector index each thread accesses. Instead of the corresponding row index, each thread uses the column index of its matrix value to access the correct input index. Once these products are completed and the global output vector is updated, the kernel then processes each global value by passing it through the Softmax function, as shown above. In order to calculate the Softmax, each thread first atomically adds to a local float value stored in shared memory so that the sum of the exponential values can be determined. Since it is known that the output vector is much smaller than $BLOCKSIZE$, there is no need for this float value to be in the device global memory. This summing operation requires an additional *syncthreads*() call as each thread waits for all the others to finish adding before moving on.

One drawback of this approach is the need for a *syncthreads*() call after each *atomicAdd*() to ensure that each vector, local and global, holds all of the values from each of the necessary threads. Another drawback is the large volume of dropped out threads after the first *atomicAdd*() to the local vector. After that initial add, the only active threads are the first N threads, with N being the number of elements in the output vector. For both kernels, N is less than the size of a warp, therefore there will be warp divergence upon execution of both multiplication kernels.

3.3 Vector Operation Kernels

Additionally, our algorithm requires two vector operations: component-wise addition and subtraction. Although each of these operations are more accurately matrix addition and subtraction, since all matrices used in this algorithm are flattened, the operation is essentially a vector operation. As with the matrix multiplications, a single thread is assigned to each matrix element, however instead of to an input matrix, each thread is assigned to an index of the output matrix. Therefore, each thread accesses the corresponding values in both input matrices and applies the necessary operation. Initially, both addition and subtraction kernels acted in the same way, with one thread per output element. However, using the knowledge that for the vector addition kernel, the input matrix sizes were of a reasonably large size, thread coarsening was applied to increase the performance of the kernel. A $COARSENESS$ factor was introduced that dictated how many output elements a single thread computed. The $COARSENESS$ value used in the final submission was a value of four. With more time, it was hoped that this value could be adjusted to possibly further optimize the performance. Another performance boosting attribute that both vector kernels had however, was thread coalescing. In the subtraction kernel, this was done in the same way as with the matrix multiplications, by using consecutive threads to access consecutive matrix elements. However, with the $COARSENESS$ factor in the addition kernel, more manipulation was needed to achieve thread coalescing. Since the grid size of the addition kernel is reduced by a factor equal to the $COARSENESS$ value, to achieve thread coalescing as each thread operates on multiple indices, the kernel has two options: shift by block size or by grid size. Our implementation chose to shift by grid size so that for each pass through the loop within the kernel, the i_{th} grid size portion of values is calculated.

3.4 Back-propagation Kernels

The back propagation kernel are responsible for propagating the error from the output layer y back to the inputs. In our experimental setup it is done in two steps. First the error is back propagated from the output layer y to the hidden layer and the weights of hidden layer are appropriately updated. Following this, the error is propagated to the input layer through the middle and the weights of the input layer are updated accordingly.

3.4.1 Backpropagation Kernel 1

The BackPropMulKernel1 updates the weights v of the hidden layer using the following update relation:

$$\Delta \mathbf{v}_i = \eta \sum_t (r_i^t - y_i^t)\mathbf{z}$$

where $\Delta \mathbf{v}$ is the the amount by which the weights of the hidden layer are updated. r^t is the actual label, y^t is the predicted label and z_h^t is the output of the first layer after applying the non linear ReLU function.t represents each image or single row of the data set consisting of 784 features. i represents one of the possible classes that the input can be classified into. There are 10 possible classes/labels (from 0 to 9) that the MNIST dataset can be classified into.

The update of weights v is performed in parallel by par-

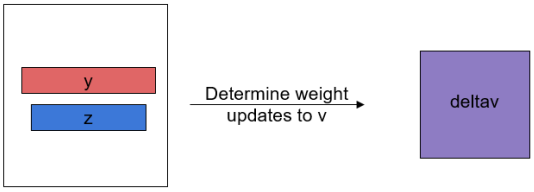

Figure 4: Updating the weights v of hidden layer, where y is the predicted output and z is the output of the input layer after applying ReLU activation function.

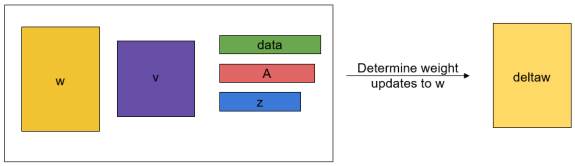

Figure 5: Updating the weights w of the input layer using w, v, data, A and z. w are the initial set of weights of the input layer, v are the weights of the hidden layer, data is one input image, A is the difference between the predicted label and actual label and z is the output of the input layer after applying ReLU activation function.

alleling with respect to hidden units in the hidden layer. Therefore, one thread is assigned for each hidden unit. Consequently, the above weight update turns into a matrix multiplication.

3.4.2 Backpropagation Kernel 2

The BackPropMulKernel2 updates the weights w of the input layer using the following update relation.

$$\Delta \mathbf{w}_h = \eta(\sum_i (r_i^t - y_i^t)v_{ih})z_h(1 - z_h)\mathbf{x}^t$$

In the above relation η is the parameter to tune stochastic gradient descent algorithm. h represents one of hidden units, it goes from 1 to H where H are the total number of hidden units in the hidden layer. z_h is the output of the input layer after applying ReLU for one hidden layer unit. $\mathbf{x}$ is the input consisting of one image of 784 features. r_i^t,y_i^t,η,i and t are the same as described above for Back-propagation Kernel 1.

The update of weight w is performed in parallel in BackPropMuxKernel1. The kernel is parallelized with respect to hidden units in the hidden layer. Therefore, one thread is assigned to process computation associated with each hidden unit. The above update of Δw is performed as parallel matrix multiplication.

4. VERIFICATION

4.1 Testing Procedure

To test individual kernels against CPU implementation of the respective kernels, we created dummy data and compared both the CPU and GPU implementation. To make sure parallel implementation of multilayer perceptron is robust and correct, the weights and biases at each layer were

```
[udawa004@ece-gpulab04 APP-Project]$ ./neuralnet train_da
    Timing 'Sequential Code' started
The problem started with sequential coding
Reading inputs...
Successs!!
Initializing variables
Successs!!
MLP Training
After MLP call
Accuracy: 0.098717
        GetTimeOfDay Time (for 1 iterations) = 82.03
        Clock Time        (for 1 iterations) = 81.96
    Timing 'Sequential Code' ended
    Timing 'getAccuracy' started
        GetTimeOfDay Time (for 1 iterations) = 90.357
        Clock Time        (for 1 iterations) = 90.38
    Timing 'getAccuracy' ended
ACCURACY: 0.098717
```

Figure 6: Performance metric

compared against the working code implemented in Matlab. The comparison of accuracy of sequential and parallel code is used as an acceptance test.

4.2 Acceptance Test

Softmax function at the last layer of the model generated lot of NaNs due to overflow/underflow error. Due to this, accuracy for both GPU and CPU was compromised. Hence we tested the equality of the accuracy generated by GPU and CPU with Nans as the acceptance test for our implementation.

5. PERFORMANCE

The inherent nature of the problem was sequential. Despite having a big data set, we were not able to parallelize it, since we were following stochastic gradient descent algorithm to train the model, where each data was dependent on the weights and bias adjusted by the previous data. Because of this, we were not able to achieve our performance goals. If we had more time we would have implemented batch gradient descent to achieve maximum parallelism. There was no speedx between CPU and GPU implementation. The GPU time measure consisted of 4 cudaMalloc and 4 cudaMemset, epochs*noOfImages*7 kernel calls and epochs*noOfImages*7 cudaDeviceSynchronize beside the normal kernel function calls. This was one of the reason our GPU time was more than CPU in the figure 3.

6. CHALLENGES

During the course of this project we faced numerous challenges at every step of the way. The very first challenge we faced was the issue of having NaN's in our output matrices in both the sequential and parallel implementations. the root cause of this issue is the softmax function used at the output layer:

$$\sum \mathbf{y}_i = \frac{e_i^y}{\sum e_i^y}$$

In order to solve this issue we first recognized that the input data matrix consisted of RGB values of pixels ranging from 0-255 that may have caused the the softmax function to have a really high value which in turn was causing an overflow, thus pushing the output to NaN. Our first step was to normalize our input data so that the input pixels are all in a range of 0 and 1. This solved the issue on MATLAB,

however it did not make a difference in our C implementation of both the sequential and parallel program. Next, we converted the float data types to double. Followed by comparing outputs at every stage of the algorithm with the MATLAB implementation.

We then came across the safe softmax implementation that avoids the NaN issue:

$$\sum \mathbf{y}_i = \frac{e^{y_i} - max(Y)}{\sum e^{y_i - max(Y)}}$$

However, this did not seem to solve the issue either. We believe that given more time we would be able to find a solution to this problem.

7. CONCLUSIONS

In conclusion, we were successfully able to implement the Multilayer Perceptron algorithm in both sequential and parallel form in C and CUDA respectively; with both giving us an equal accuracy, despite the compromise in accuracy due to the challenges mentioned above.

Thus, we can say that due to the inherently sequential components of stochastic gradient descent, the problem could not benefit much from parallelization.Given the time, we would have liked to implement the a batch gradient descent algorithm that could take advantage of a higher number of parallel computations or implement a Multilayer Perceptron that has a higher number of hidden units and hidden layers that would lead to high number of parallel computations as well. We would have also liked to use a larger dataset to see how the performance improves with teh size of the data. Further, making use of some more optimized methods to improve performance of the computations could also be implemented, such as, sparse matrix multiplications and constant memory for storage.

8. REFERENCES

[1] E. Alpaydin. *Introduction to Machine Learning.* The MIT Press, 2014.

[2] R. Genov and G. Cauwenberghs. Charge-mode parallel architecture for vector-matrix multiplication. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 48:930 – 936, 11 2001.

[1] [2]