WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A & COMS7045A Term Project

Kobus Van der Walt, 2219677, Coms Hons

April 14, 2019

1 Introduction

1.1 Multi Layer Perceptron

The Multi Layer Perceptron (MLP) is an information processing technique that was inspired by the human brain's neurons and synapses. The MLP consists of multiple layers of neurons, with every neuron in a layer being fully connected with all the neurons in the following layer. The input of every layer is fed forward to the following layer until the output is produced at the final layer. A MLP can thus be described as a fully connected feed forward neural network. A neuron in the MLP is an artificial construct that aims to imitate the biological neuron of the human brain. It receives one or more inputs, each input being multiplied by a weight. This weight multiplication emulates synapses of variable strength found in the human brain.

The input of a neuron is the sum of the output from the previous layer's neurons multiplied by each respective weight. This is then fed through what is known as an activation function. Activation functions play an important role in neural networks which is to introduce non linearity. The strength of a MLP comes from being able to learn a hierarchical interpretation of a problem domain. If a neural network does not have non linearity it is equivalent to simply having a single layer multi layer perceptron, and as such there is no hierarchical understanding. Thus we need non linear activation functions such as the sigmoid, tanh or ReLU to understand complex and noisy problem domains.

In the biological neuron the synapses of neurons deemed useful are strengthened and the synapses deemed not useful are weakened. In the human brain this process is managed through a chemical process. We can emulate this process through a technique called back propagation. Back propagation is the core of modern machine learning and has led to numerous improvements in the field of AI. The basic idea of back propagation is to determine how much the weight of a particular connection should be increased or decreased based on how much it contributed to the final prediction error. This is done through propagating the error backwards. First we start with the output layer and determine the error between each output node and the expected output from the label. Next we take the partial derivative of this error with respect to each output node's input. Finally we take the derivative of the error with respect to the input of each neuron. We can use this back propagated error to update the weights and biases, and then we can follow the whole process again with the preceding layer. A more in depth explanation is discussed by [3].

The input layer receives the input from some external source. This can be pixel data from an image, waveform samples from an audio file, temperature readings from a table, or any other discrete data point. The hidden layers form a hierarchy of understanding about the dataset and determines which neurons should fire in order to produce the correct output. The output layer is where the result of the neural network is presented.

Since the input of a layer can be described as the dot product between 2 matrices, matrix 1 being the input and matrix 2 being the weights, it is obvious that there is parallelism opportunities. Each neuron in a layer can compute it's input independently from the other neurons in the layer. This implies high parallel opportunity. A parallel bottleneck is the fact that a layer can not compute it's result without the preceding layer having finished its computation.

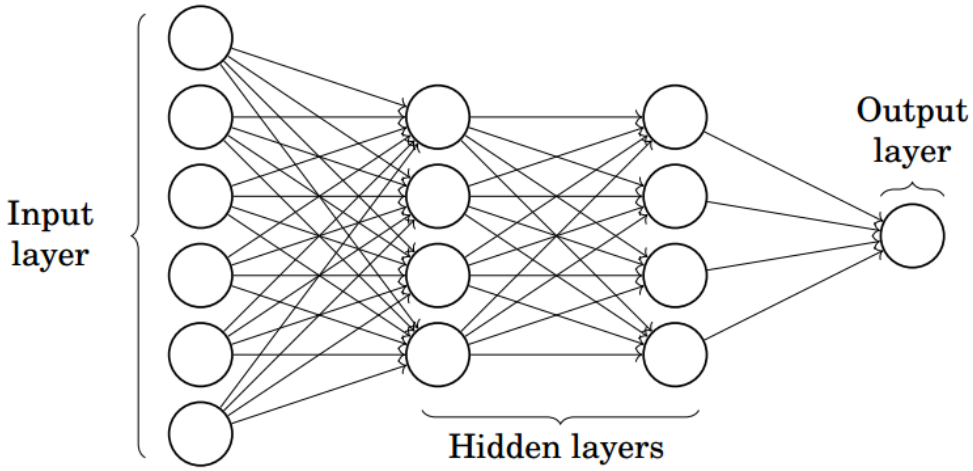

Figure 1: A Multi Layer Perceptron with 2 hidden layers.

1.2 MNIST Dataset

The MNIST dataset is comprised of 70000 hand written digits and is the go to datset for experimenting with machine learning techniques. The dataset is split into 60 000 training samples and 10 000 test samples. Each samples is 28*28 pixels in size which results in a 784 input vector for our neural network.

Figure 2: Examples of images in the MNIST dataset. The labels are 4, 0, 9, and 2 respectively.

1.3 MPI

MPI is short for Message Passing Interface which is a standard developed by researchers and academics for use in High Performance Computing tasks. There are multiple implementations of the standard such as Mpich for Linux and on Windows one can use MSMPI. MPI provides a collection of primitive functions which aim to make the development of parallel applications simpler. There are simple building blocks such as blocking Send and Receive as well as more complex functions such as scatter, gather, allreduce and more.

1.4 CUDA

The Graphics Processing Unit was originally designed to perform expensive graphics rendering tasks in video games, and over time has become a core part of the modern computer. More recently there has been a focus on utilizing the massively parallel nature of the GPU to perform general purpose computing tasks. NVIDIA developed an API called CUDA to enable developers to take advantage of the GPU's power through familiar languages such as C++ and Fortran. With CUDA most developers can execute their instructions on the GPU in a relatively easy manner, in contrast to the requirement of advanced and specialized knowledge to use APIs like OpenGL and Direct3D.

The processing flow in CUDA can be described as :

1. Copy data from the host(CPU/RAM) to the device (GPU).
2. Initiate the compute kernel from the host.
3. The GPU's multiprocessors execute the kernel in parallel.
4. Once GPU computation is complete, the results are copied back the host.

The execution of a CUDA kernel can be controlled by specifying the number of threads, the block size, and the grid size. A thread is the logical component which executes the kernel. Each grid contains a number of blocks limited to 65 535 per dimension and each block contains a number of threads with a limit of 1024 on most cards. On a hardware level threads are contained in a warp, which is a collection threads (typically 32) that execute the same instruction in parallel. This warp design allows for both parallel and pipelined behaviour. This means that if there is execution divergence within warps they are partitioned based on the instructions needed and executed in serial.

The CUDA memory model contains several different types of memory, each with their own strengths and weaknesses. Firstly there is Global Memory which can be read and written to by every thread. The host can copy data between this Global Memory and the main memory on the CPU side. Local memory is an abstraction of Global Memory which can only be accessed by the thread which allocated it. On modern GPU architectures local memory is cached to improve performance but when there is a larger data requirement than the registers or cache can accommodate, local memory is stored off chip alongside Global Memory. This means that the the kernel should be carefully designed to work within the intended hardware's performance characteristics.

The next memory construct is known as Shared Memory which can be used by each thread within the same block. Shared Memory is allocated when the kernel is launched and as such is not dynamic within the execution of the kernel. The host is also not able to directly copy data into shared memory. The advantage of shared memory is that it is located on the multiprocessor chip which means the latency is much lower than that of Global Memory. The capacity of Shared Memory is around 64KB depending on the processor used.

Constant Memory is a cache optimized piece of memory that is allocated by the host and is readable by every thread, but does not allow any thread to write to it. Constant Memory, like Shared Memory, has a relatively small capacity of around 64KB. Texture Memory is also a cached optimized piece of memory but allows for much larger storage capacity than Constant Memory. It is allocated by host for use by the device and all threads can read texture memory, but not write to it. Texture Memory tries to place neighbouring pieces of data physically close to each other on the chip. This results in reduced memory traffic.

2 Methodology

2.1 The Network

For our tests we made use of a neural network with multiple configurations. We decided to run the training process for 20 epochs since we found that the network achieves an accuracy rate within 2 % of the eventual best accuracy within 15-20 epochs. This is corroborated by [1] who worked on MNIST as well.

For the activation of the final layer in our neural network we utilized the sigmoid function. The sigmoid function has the benefit of normalizing our output to a value between 0 and 1 while ensuring that, no matter how small, there will always be a derivative to reduce the error.

Sigmoid :

$$f(x) = (1 + e^{-x})^{-1} \tag{1}$$

The sigmoid function was one of the early activation functions researchers used when developing the MLP, and as such it isn't perfect. The sigmoid function suffers from the vanishing gradient problem. This occurs when the errors that are back propagated get progressively smaller as we move away from the final layer, until they can no longer be stored in a floating point number and then "vanish". A non saturating non linearity was noted by [2] and is subsequently referred to as a Rectified Linear Unit (ReLU). With a ReLU the derivative is 1 which results in a parameter update that does not get progressively smaller like the sigmoid function. This results in a 25% improvement in convergence according to [2]. The ReLU is also computationally simple and thus improves the performance of the forward and backward passes.

ReLU:

$$f(x) = \begin{cases} 0 \text{ for } x \leq 0 \\ \text{x for } x \geq 0 \end{cases} \tag{2}$$

For the loss function we used the Mean Square Error since it is conceptually and computationally simple.

MSE Loss :

$$MSE = \frac{1}{n} \sum (Y_i - \hat{Y}_i)^2 \tag{3}$$

Initialization : All biases were initialized with a value between 0 and 0.01. The weights of the sigmoid layer were initialized with a random value between -1 and 1. The weights of the ReLU layer were initialized with a value between 0 and 0.01. We did not experiment much with initialized but found that for the sigmoid layer it is optimal if the approximate mean of the initialed weights is 0. If weights are initialized with means which are higher lower there is a risk of experiencing NaN errors. For ReLU layers we have a small positive value since we would like all neurons to have some error back propagated at the start. We employed a leaky ReLU which prevents neurons from ever truly "dying" but proper initialization still leads to quicker convergence.

The dataset's images and labels are converted to floating point numbers. The network's weights and biases are stored as floating point numbers as well. This allows us to halve the size of our network and decrease the time of a single forward and backward pass, without suffering any degradation in the accuracy of the network.

The labels in our dataset are transformed into "one hot encoding" when loaded. For example the label of a image that resembles the digit 7 would be "7", and the one hot encoded version would be "000000100". This is done to ensure that larger numbers don't receive a larger error simply because of their numerical meaning. Each digit should be considered a symbol with the same amount of error being backpropegated if a mistaken prediction is made. The consequence of this is a network with 10 neurons in the output layer.

For our training process we used Stochastic Gradient Descent which achieves remarkably good results while being conceptually simple.

We found that our networks with a lot of neurons had a divergence in the test and train error. This is known as overfitting and is addressed by introducing some regularization mechanism. We opted for a technique known as dropout where neurons are randomly disabled with a probability p and the output of the active neurons are multiplied by $1-p$.

In all our implementations we flattened the memory access for the weights. Traditionally weights would be stored in a 2 dimensional array with each dimension representing the input or output of the layer. Our flattened version was a 1 dimensional array which stacks the weights for each output neuron next to each other sequentially. For example a network with 5 input neurons and 3 output neurons 000001111122222. This scheme has a number of benefits. Firstly in the MPI implementation we don't have to copy the parameter updates of each network in a complex way. We can simple use MPI's reduction methods. Secondly the memory allocation on CUDA is simpler since we don't have to first allocate an array of pointers, with each pointer pointing to a different array. From a performance point of view, this memory layout reduces the chance of a cache miss since memory used together is packed together. It should be noted that for back propagation data is sparse but gains are achieved in the updating of weights.

2.1.1 Serial Implementation

The benchmark implementation of our MLP was a serial implementation running on the CPU. This version provided the core structure which the parallel implementations would build on. The network is comprised of several classes which represent layers. This structure provides an extendable framework for building a neural network. We implemented 5 types of layers and due to the structure we could easily add additional layers such as a convolutional or recurrent layer. We can also easily change the order or layers so we believe this approach to building neural networks is scalable and robust from a software development perspective.

The layers we implemented were :

- Sigmoid : This layer applies the sigmoid activation function to a set of input values and presents the result in a set of output values.
- Relu : This layer applies the relu activation function to a set of input values and presents the result in a set of output values.
- Dense : This layer has a configurable number of inputs and outputs which can be different. We use this layer to compute the input to some activation function. We can also scale the network down or up the deeper we go by setting the number of inputs and outputs in this layer.
- MSELoss : This layer compares the predicted output with the known true label (expected output) and determines an error for each output neuron. We also use this layer to keep track of our collective error to ensure the network is actually learning.
- Dropout : This layer simply applies the dropout technique to a set of input values and presents the result in a set of output values.

2.1.2 Cuda Implementation

The MNIST dataset is occupies around 256mb of RAM when using one-hot encoding for the labels and floating point storage. We decided to store the entire training and testing set on the global memory of the GPU. This eliminates the need to constantly communicate between the GPU and RAM on the host side or even worse the HDD.

Our fist attempt at a parallel MLP was based on data parallelism. We developed a CUDA kernel which would perform the entire training step for a sample in a single CUDA thread. This approach involved initializing 8192 MLP networks. Each with their own set of weights and biases. Every thread would be assigned to a network and perform it's forward and backward pass. In this approach we only updated the weights after every minibatch of 5 training examples. This

was done in attempt to increase performance since updating the weights does take some time. After each minibatch the parameters needed to be averaged. This would ideally be done by a second kernel without copying back the parameters to main memory on the host side. We did not implement this parameter averaging since it was apparent at this point that this approach had some significant challenges which need to be overcome first.

The biggest challenge we encountered was the large amount of memory that needed to be allocated, moved, read and written to. This alone is a significant drawback since memory access is the usual bottleneck in a system. This is the reason computers introduced several levels of cache. Consider a network with an input layer of 784 neurons, a hidden layer of 64 inputs, and an output layer of 10 neurons. The size of this very simple network can be calculated as (784 * 64 + 64 * 10) for the weights and (64 + 10) for the biases which totals 50890 parameters. Each parameter requires 4 byte of memory in the case of a floating point number meaning the network requires 203560 bytes or 0.20356 mb to be stored. Each parameter also needs memory to store it's error from the back propagation step. This doubles the required memory to 0.40712 mb. A network with a hidden layer of 512 requires 3.2564 mb which allows you to train 1842 networks in parallel if the GPU has 6gb of memory.

The second challenge was in testing the kernel to determine which part was the bottleneck. We suspect there were cases of branching execution which prevents the kernel from benefiting from the warp parellization structure of the GPU. We also suspect that a lot of memory ended up in global memory which could be stored in registers throughout a particular pass. But since the implementation of a whole network within one kernel was complex to debug we decided to rather implement a conceptually simpler model.

Because of the challenges discussed above we did not pursue this approach to completion but we did compare the performance of the forward pass to that of the serial implementation and found a rough increase of 10% on the full 60 000 sample training set with 8192 parallel networks and a size of 784-64-10.

The second CUDA parallelization implementation was a model parallelization approach where we developed a CUDA kernel for each of the layer classes and used the framework developed in the serial implementation. For the Dense layer we had a kernel for the initialization of the weights and biases which made use of curand to sample random values from a uniform distribution. We then had a kernel for the forward and backward passes as well as the weight update pass. The MSELoss layer had only one kernel for calculating the appropriate error for each output neuron. The sigmoid, ReLU and dropout layers all had 2 kernels. One for the forward pass, and one for the backward pass which calculates the error derivative. This kernel per layer approach does not incur a significant overhead since the kernel invocations are all put into a stream which allows the GPU to get the next kernel invocation ready for execution while it is execute the current kernel.

In this implementation we assigned the computation of each neuron to a single CUDA thread. The result of this is the fewer neurons there are in a layer, the smaller the benefit gained from parallelism. This is especially true in the last layers of this dataset which is only 10. We do however suspect that autoencoder structures, such as those used in semantic segmentation, will observe greater parallel gain.

Our biggest bottleneck in this model parallelization approach was the use of global memory which resides off chip in a Graphics Processing Chip. To achieve greater performance we ensured coalesced memory access. We attempted to use shared memory since it is lighting fast but did not find an appropriate place to make use of it. We implemented shared memory in the forward and backward kernels of the Dense layer but did not find a significant speed up in performance. Shared memory was used to store the input in the forward pass, and the backward derivative in the backward pass. We found that the overhead of first reading the input into shared memory from global memory, then synchronizing the threads performing the load, only left us with a 3% increase in performance at most. As the number of neurons grow within the input layer this performance increase will grow as well.

2.1.3 MPI Implementation

The third implementation utilized MPI to develop a data parallelism approach. The first part of this implementation loads the dataset into the memory of the MPI process with rank equal to 0. This process is considered the master node in our implementation. After the master node has loaded the dataset it communicates the dataset to the other nodes through the use of the MPI broadcast functionality. Each training sample is sent as a sperate message. In total 140 000 messages are used to transfer the dataset from the master node to a process. After every node has a copy of the full dataset, a network is created on each node and the parameters are initialized according to that networks rank to ensure as wide coverage of the parameter space as possible.

The train loop can now commence. Each node trains their local network on $trainCount/nodeCount$ randomly selected samples from the entire dataset. Then the master node tests it's network against the test set for us to determine whether or not the network is improving. After the test is performed all the network parameters are summed into an array on the master node and distributed to the other nodes. We made use of MPI's all reduce utility to achieve this. After this step each node sets their appropriate biases and weights to this global sum divided by the number of nodes. And as such we have a distributed parameter averaging operation.

3 Experiment

3.1 Hardware

The tests were performed on a machine with the following specifications :

Machine Specifications	
Operating system	Microsoft Windows 10
Processor	Intel Core i7-7700HQ CPU @ 2.8GHz
Memory	16 GB DDR4 (2,666MHz)
Graphics Processor	Nvidia GeForce GTX 1060 (6GB GDDR5)

3.2 Focus

The focus of this experiment revolved around 4 key metrics. The accuracy of the final trained model (measured by the error rate). The time it took for the model to converge and achieve this accuracy (measured in epochs). The amount of memory used by the training process. The total time the training took to complete.

3.3 Results and Discussion

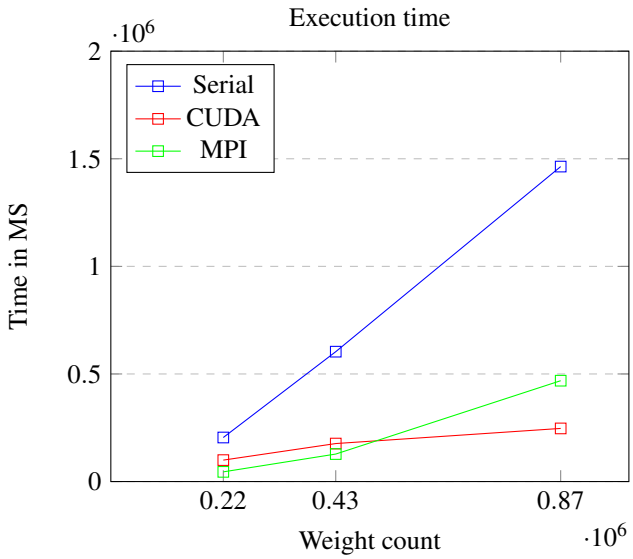

Figure 3: This graph illustrates the average time an epoch took to train. An epoch consists of randomly sampling 60 000 training samples and performing both the forward and backward passes as well as the weight updates. We can see in this graph that the serial implementation is the slowest of the implementations. The MPI implementation outperforms CUDA when the parameter count is low but we can see an inflection point with around half a million parameters where it becomes clear that CUDA scales better. The fact that MPI provides the best performance when parameter counts are low is likely due to the L1 L2 and L3 cache of the CPU. This cache memory is faster than the global memory on the GPU and thus allows for quicker training.

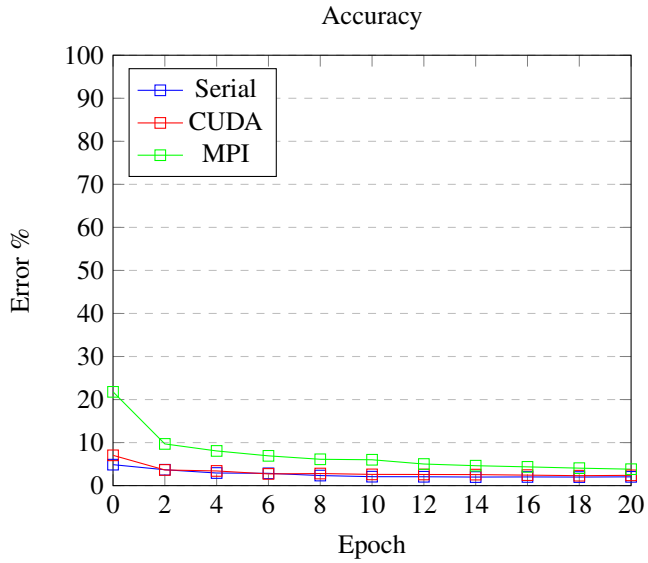

Figure 4: This graph shows the error rate of the test step while training. We can see that the serial implementation converges quicker and achieves the lowest final error rate of 1.9. Cuda is the second best performer with a rate of convergence comparable to serial and a final error rate of 2.31. MPI converges slower and achieves a final rate of 3.82. We suspect that MPI higher optimal error rate within 20 epochs might be due to the parameter averaging technique introducing sub optimal gradient steps. We trained the MPI implementation again for 50 epochs and then a score of 2.61 was achieved.

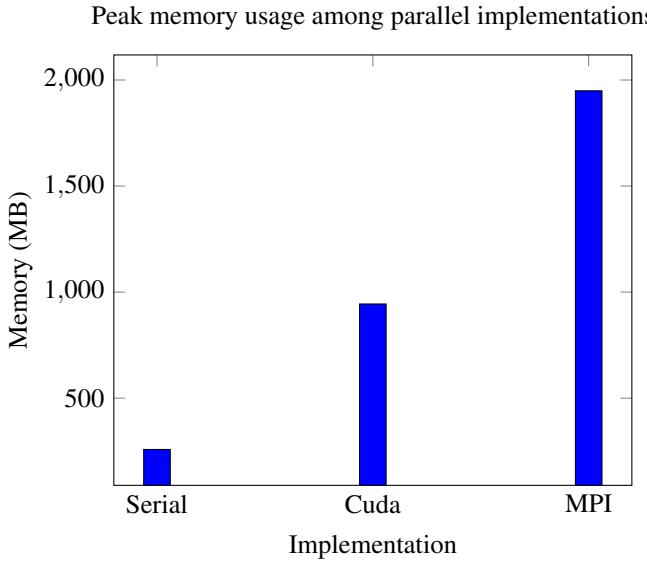

Figure 5: This chart shows the memory requirements of each solution. Serial is the obvious winner and the bulk of the serial memory requirements is simply RAM to store the training data. The runner up is CUDA which requires the same amount of memory in RAM but introduces an overhead which is used by the SDK when data is transfered between host and device. MPI requries the most memory when run on a single computer since each node needs to store it's own copy of the training data and a neural network. This is not a problem when the nodes are run on a cluster where each machine has it's own share of memory.

4 Conclusion

We have shown that it is possible to use the GPU and high performance computing tools such as the MPI standard to speed up the training of a traditional Multi Layer Perceptron. We discussed the strengths and weaknesses of each of 3 MLP implementations. We showed execution time, memory usage, convergence rate and accuracy of each implementation. In future we would like to look at combining the CUDA and MPI solutions into a single solution where several nodes utilize their own GPUs to train a model and then perform parameter averaging. Finally we think that applying HPC techniques to more machine learning problems is likely to unlock more advances in the field and that it's a worthwhile pursuit.

References

[1] DC Ciresan, U Meier, LM Gambardella, and J Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. corr abs/1003.0358 (2010).

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[3] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.