Parallelizing Artificial Neural Networks*

Syed Zain Raza
Stevens Institute of Technology
Department of Computer Science
Jersey City, USA
sraza3@stevens.edu

***Abstract*—In this project I parallelized artificial neural networks using different techniques from the field of parallel programming. Artificial neural networks just like any other machine learning task, consists of training and testing tasks. However, the training time for neural networks takes a lot of time especially in case when the dataset is large. The motivation to parallelize also comes from this fact and goal is to decrease the overall training time while keeping testing accuracy same. I have used MNIST dataset in this project, which consists of sixty thousand images of all digits and is used to train neural networks to act as a digit recognizer. In this report, I have performed two different techniques to parallelize the processing of neural networks: (a) One is to process each image one at a time. My results show that this technique is only good enough to achieve a small increase in performance. (b)The other technique I used is batch processing that is to process a batch of images instead of only processing one image at a time. This technique provides more promising results and succeeds in achieving a far less training time.**

I. Introduction

Artificial Neural Networks can be defined as a combination of different neurons connected with each other through different layers. A simple structure of a neural network consists of three layers namely input, hidden and output layers. As already mentioned, training and testing are two parts of a machine learning task. In a supervised machine learning task, with each input there is also a label. These labels are used in training process and helps the network to learn the details of inputs. Training further consists of two passes named as forward and backward pass.

In a forward pass, different neurons in the three layers of neural network are responsible for various things. Input layer consists of multiple neurons which takes different inputs, their weights and pass it over to the hidden layer. Hidden layer neurons are responsible to multiply each of these inputs with their respective weights, taking summation and adding a bias value. These results are then passed through an activation function. I used sigmoid activation function which convert given values to values in the range[0,1]. After activation each hidden neuron then passes its values to the output layer which also multiplies it with their randomized weights, adds a bias value and generate outputs. The outputs of the output layer are then stored in a vector.

After this, we use a loss function also known as cost function to know how much far our predicted value is from the actual value or label. Label is also stored in a vector which is of the same shape as of the output layer. Different loss functions are used in different machine learning tasks, in this task I used a mean squared error function.

Each output value from the output vector is subtracted from the values from label vector, squared and then added with each other. This gives us a cost value. As loss function is a measure of difference of our prediction and actual values, we want to minimize the cost values.

To achieve this, an algorithm known as gradient descent is used. In a backward pass, this algorithm takes derivatives of the loss function, calculate values and use them to update weights and biases of previous layers. A derivative calculates the slope which helps to move towards the global minimum as we want to minimize the cost values.

We continue to do this process of forward and backward passes until out cost value is equal to a very small. This assures that our predicted value is as close to the actual.

Now moving on to the testing process, the final model that is the weights and biases saved in the training are used to perform only the forward pass on unseen examples which gives each input a predict value. To calculated the accuracy of our model, we then divide correctly predicted inputs with the total number of inputs.

II. Why Parallelize?

As we have seen in the previous section, that the training process of a neural network consists of a large number of calculations which results in consuming a lot of time. Sequential programs written for neural networks take around numerous hours to complete the training process.

For example, in a simple case of my dataset each input image was of 28*28 size. This 28*28 matrix consisted of grey scale pixel values from 0 to 1. With anything near to 0 being more dark and near to 1 being more bright. Because of the image size being 28*28, the input layer have 784 neurons. Each of these neurons then were connected to each hidden layer neuron. There are 128 hidden neuron in the hidden layer so the weight matrix becomes 784*128. The output layer has 10 neurons so its weight matrix is 128*10. Other than this when we also consider biases, we have 128 biases in hidden layer and 10 biases in the output layer. It means network has to handle with a total of 101,770 values.

For this reason, we can use the power of GPU's to speedup the process of training while maintaining the performance of the network. Both forward and backward pass of neural network is parallelizable.

During parallelization of both passes, one of the bottlenecks is that at one time only one layer of the neural network can be parallelized. This is due to the reason that the next layer is waiting for the outputs of the previous layer. But still a lot of improvement in the training time can be achieved using parallel programming and some other techniques as I will mention in the later sections.

III. CPU Version

I first implemented a CPU version of neural network in C++. This was a simple version to be taken as a baseline while parallelizing the neural networks.

As already described, my input layer has 784 neurons, hidden layer has 128 neurons and output layer has 10 neurons. I noted the timings of both forward and backward passes. A forward pass for the CPU version took around 0.31 ms and backward pass took around 0.68 ms. In total, the training process was about 1.7 hours.

For testing, I used the model on the unseen test examples and got an accuracy of 94

The implementation was done for both forward and backward passes. As already said in the introduction part, I used sigmoid activation function in the forward pass, in the hidden layers and also in the output layers.

IV. Methods

I used two main methods during this project. (a) Single Image Iteration. (b) Batch Processing - Batch of Images Iteration. I will explain them one by one in the following paragraphs.

A. Single Image Iteration

In single image iteration, I only used one image as an input. By iteration, I mean one forward and backward pass. For this method, the input is processed as single image of 28*28 size. Input neurons are same as 784. I also kept the hidden and output neurons as 128 and 10.

B. Batch of Images Iteration

In this part, I used a batch of images instead of processing only one image at a time. How much images will be taken at a time depends on the batch size we chose before starting the program. After trying different batch sizes, I selected ”32” as the correct batch size as it was giving me the desired accuracy within least time. For this method, I also changed the number of input and hidden neurons.

As now the input is of 32 images with each 28*28 size, I kept the input neurons as 25088. I also changed the hidden neurons to 768. It is not a special number, I randomly guessed different number of neurons but it was giving me the correct and best possible results.

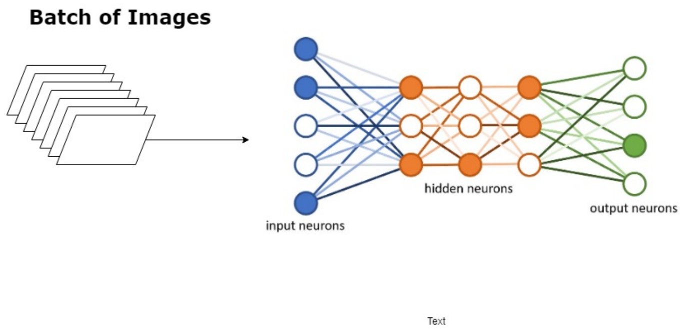

Fig. 1. Batch Iteration!

V. GPU Versions

Before starting to explain the all the GPU versions I implemented and tested, I would first like to mention the baselines that were set for all of these versions.

In all of these versions except the batch processing part, I had 784 input neurons, 128 hidden neurons and 10 output neurons. The epochs was set as 50 meaning there will be 50 iteration of forward and backward passes. The test accuracy for all of these versions were above 90

A. GPU Version 1 - Naive

The first version of GPU that I implemented was very simple and naive. I had a total of three kernels, two for the forward pass, and one for the backward pass. One of the forward pass kernel was responsible for multiplying weights with input matrices, taking their summation and adding a bias. The other kernel of forward pass was for applying the activation function.

The backward pass kernel, was responsible for calculating the derivatives, applying the gradient descent algorithm and computing new weights for the layers. These new weights were then updated in the host.

As far as work load of threads is concerned, each thread was responsible for calculating one output value in the forward pass. Multiplying weights with input matrices is same as multiplying two matrices and thread was calculating one output of the output matrix. In the backward pass, each thread was responsible for calculating new weights, but only first thread was given the work of computing weighted sum of multiplying previous weights with error gradients.

I used a total of 1024 threads and one block for this implementation. I was getting an accuracy of above 90

B. GPU Version 2 - Naive

This version was also the same as the previous version, the only thing I changed in this version was to use multiple blocks instead of only using one block. I tried two designs, one was to set 1024 blocks and other was to set blocks same as the number of hidden neurons for the hidden layer pass and same as the number of output neurons in the output layer pass.

I was still getting an accuracy of above 90

C. *GPU Version 3*

For this implementation, I removed the sigmoid activation kernel. I noticed that I am calling all three kernels two times for each image. Instead of doing this, I can just apply the activation within one single forward kernel in that way I will have to call only two kernels two times for each image. So i changed the implementation a bit, and applied the activation in the same forward pass kernel.

The reason I am saying that I have to call each kernel two times is because there are two layers after the input layer. So one kernel does the job of hidden layer and one computes the values for the output layer.

Another improvement for this implementation was to use shared memory, as in my naive implementation I was not using any shared memory. I used shared memory in the forward pass for computing output matrix for each weight, input and also for their summation. For the backward pass, I used shared memory to save computed weighted sum which was then used in the gradient descent algorithm.

This change in implementation, resulted in a significant increase in performance as I was still getting above ninety percent accuracy for only 1.6 hours which was better than CPU version.

One point to remember is that this implementation still had only one block for each kernel and 1024 threads. Using multiple blocks for this implementation was also taking more time than the CPU version.

D. *GPU Version 4*

In this version, I tried two different enhancements. Until now, I was only using 1D blocks and threads. To further increase the performance of forward pass, I tried using tiled implementation with 2D blocks and threads as it normally amplifies the performance for matrix multiplication. I implemented it both with shared memory version and without shared memory.

But the time taken during the training time for both forward and backward pass was further increased instead of decreasing. The total time taken was increased to 3 and 2.6 hours which was far more than the CPU version.

GPU Version 5 - Batch

In all of the previous implementations, I was using only one image at a time. For this version I changed my implementation to batch processing by taking multiple images at a time and improving the overall time taken during training. As already said for one image only 1 block was enough, but to further increase occupancy I had to try to this implementation as it was giving me an opportunity to use an increased number of blocks available in my GPU.

Almost everything was same for this version, the only big change was to process a huge amount of data at once. The forward and backward passes almost remained the same with only a increment of a variable batch size which determined the number of images processed at once. Another change in design was now I was using blocks same as the number of neurons in the layer which is being processed at that time.

This version significantly, increased the performance meaning decreasing the training time as compared to other versions. The accuracy was still above ninety percent as was the case with the previous versions.

Further details of all these versions can be seen in the Results section of this report.

VI. Use of Resources

I had a Tesla M-60 GPU on my instance with compute capability 5.2. The maximum number of threads allowed per block are 1024 and the maximum number of threads allowed per SM is 2048. There are total of 16 SM's on this GPU. If i used a block of 1024 threads then at one time I can only have 2 blocks per SM because of the 2048 threads limit. If each SM can have 2 blocks so in total there can be 32 blocks running at one given time. This means a total of 32,768 threads. As already stated I used blocks equal to the number of neurons in specific layers of batch processing implementation.

For the single iteration forward pass, my kernel was using 14 registers, 16384 bytes of shared memory per block, 356 bytes of constant memory and 20 bytes of compiler generated constant memory in bank 2. For the backward pass of single iteration, my kernel was using 31 registers, 4 bytes of shared memory per block and 380 bytes of constant memory.

For the batch iteration, only difference was the use of shared memory. Forward pass had 2048 shared memory per each block with total blocks equal to neurons and backward pass had same 4 bytes of shared memory per block.

VII. Occupancy

In terms of occupancy I was achieving 100 percent occupancy on the calculator. As the resources already mentioned in the previous section, I used 1024 threads per block, maximum 16384 shared memory per block and maximum 31 registers per block which was giving me following results on the occupancy calculator.

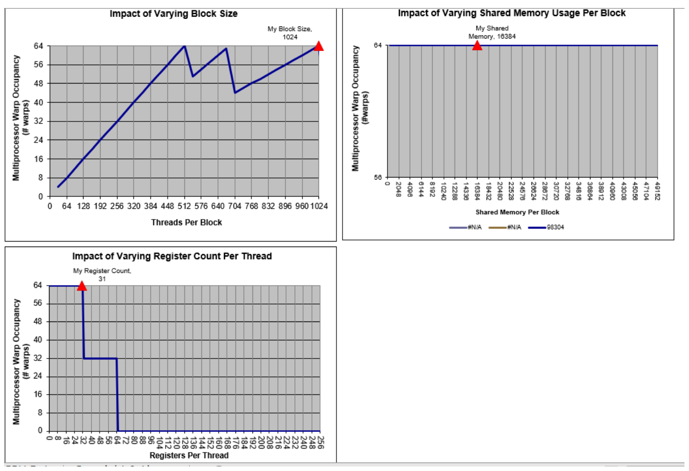

Fig. 2. Impacts on the Occupancy Calculator !

VIII. Results

The results I obtained for different versions that I implemented can be shown in the table.

Forward and Backward passes times are in milliseconds and total time in hours.

GPU shared with single image iteration was giving the best performance and showed an increase of about 1.1x. Batch iteration for batch of images was taking far less time which resulted in an improvement of about 2.4x as compared to the CPU version.

Fig 3 and Fig 4 shows this.

As you can see in the table and also in Fig 3 that both forward and backward pass of batch iteration is taking a long time but the overall time taken in training is less now. This is because of the reason that when iterating on only one image I had to call both kernels two times for each image and total images are sixty thousand. But in the case of batch iteration, 32 images are processed at once, so both kernels have to be called 3750 times which is the number obtained by dividing sixty thousand by the batch size(32).

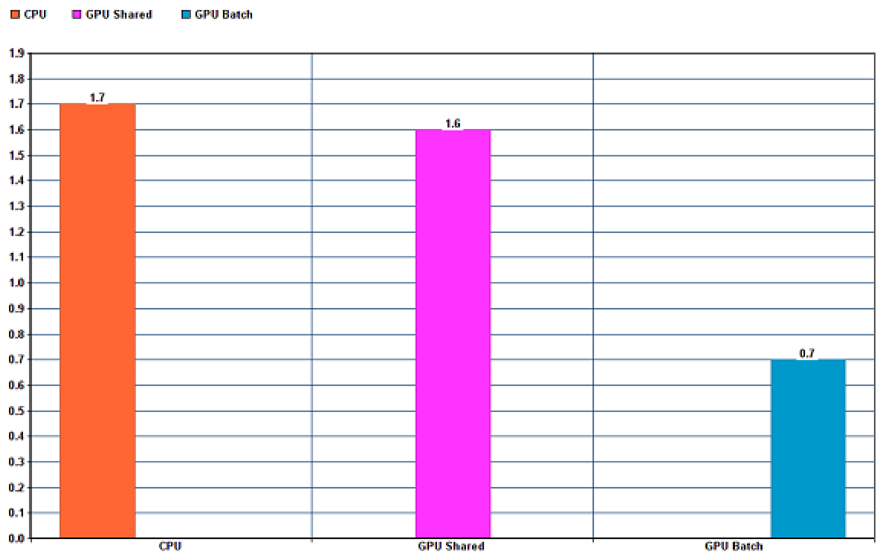

Fig. 4. Comparison of times taken during training by different versions !

Experiment	Forward Pass	Backward Pass	Total Time
CPU Version	0.31 ms	0.68 ms	1.7 hours
GPU Naive	0.22 ms	0.92 ms	1.9 hours
GPU Naive with multiple blocks	0.25 ms	1.10 ms	2.2 hours
GPU shared	0.17 ms	0.80 ms	1.6 hours
GPU tiled without shared	0.30 ms	1.5 ms	3 hours
GPU tiled with shared	0.28 ms	1.3 ms	2.6 hours
Batch Iteration	1.80 ms	4.1 ms	0.7 hours

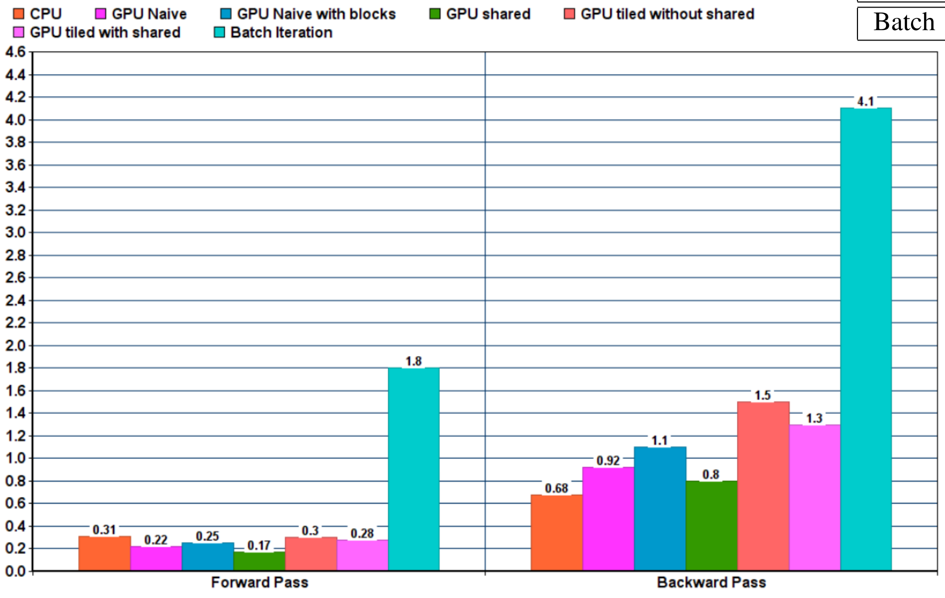

Fig. 3. Comparison of forward and backward passes time taken during training by different versions !

IX. Future Improvements

The versions that I implemented in this project can be improved in a number of ways. There are a lot of papers published in this area with multiple tweaks to decrease training time of large datasets for neural networks.

Few of the things that I noticed in my code which could not be improved due to time limitations. One is the updation of weights in the backward pass as it can be seen in the graphs that the backward is taking much longer time as compared to the forward pass which is probably due to this reason of not updating weights within the kernel or maybe writing a new kernel for updating weights. We can also utilize different machine learning methods to improve performance, one example can be of mini batch stochastic gradient descent algorithm which also works on the same principle as of batch iteration but the only difference is the calculation of gradients. It says that instead of calculating all the gradients we can randomly sample data and calculate gradient only of this sample. This also gives the same results as that gradient descent in terms of test accuracy as model is able to update weights correctly by only using a random sample with less time.

Another improvement can be by using multiple GPU's in that way large datasets can be distributed to all these different GPU's which will eventually result in way less training time.

X. Conclusion

In this report, I explained my different implementations for parallelizing neural networks. As it can be seen from the results that the best implementation in single image iteration was GPU shared which only used one block and shared memory for calculating outputs, summation of all outputs and then applying activation.

For batch iteration, it gave far better results but still there is room for improvement as my backward pass is still taking a lot time which it should not. Accuracy baseline was above ninety percent but it can also be improved using other machine learning techniques such as momentum in gradient descent or stochastic gradient descent.

To conclude, I would say this this project taught me a lot of things related to parallel programming. How each problem is different and having a solid baseline is very important. One can only parallelize an implementation if s/he knows how to implement a sequential version. Moving forward I will take all these lessons with me.

XI. References

Following are the links which were used as references during the project:

References

[1] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011, 2011.

[2] Noriyuki Fujimoto (2008) Faster Matrix-Vector Multiplication on GeForce 8800GTX, Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), LSPP-402, pp. 1–8.

[3] Rajat Raina, Anand Madhavan, and Andrew Y. Ng (2009) Large-scale deep unsupervised learning using graphics processors, Proceedings of the 26th Annual International Conference on Machine Learning, ICML'09, ACM.

[4] Kyoung-Su Oh and Keechul Jung (2004) GPU implementation of neural networks Pattern Recognition, 37(6):1311-1314.

[5] Honghoon Jang, Anjin Park, and Keechul Jung (2008) Neural Network Implementation using CUDA and OpenMP, Proceedings of the 2008 Digital Image Computing: Techniques and Applications, pp 155–161.

[6] Geoffrey E. Hinton, Alex Krizhevsky, "Image Net Classification with Deep Convolutional Neural Networks , vol. I,, NIPS , 2012, pp. 1–15.

[7] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlun, Ronak Singhal, and Pradeep Dubey (2010) Debunking the 100× GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, Proceedings of the 37th annual international symposium on Computer architecture, ISCA'10, ACM

[8] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In NIPS, 2010.

[9] Chen, Xie, et al. "Pipelined Back-Propagation for Context-Dependent Deep Neural Networks."

[10] D. Nguyen, B. Widrow, Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights, in: Proceedings of the international joint conference on neural networks, Vol. 3, Washington, 1990, pp. 21–26.