GPU based Handwritten Digit Recognition using Artificial Neural Networks

Udit Gupta
M.Tech CDS
14454
uditgupta@iisc.ac.in

Shivansh Srivastava
M.Tech CDS
14521
shivansh2524@gmail.com

***Abstract*—The current methods for training a Neural network(NN)[1] apply the same approach for all layers. However, this can prove to be suboptimal as we move deeper into the network as the size of layers, in general, decrease. Through this project we have shown how we can decrease the training time of a NN by exploiting the fact that last layers have fewer parameters and hence the "optimal" strategy that worked for initial layers can still be improved. We show the results for a 3 layer NN(Input, Hidden and output layer) in which our contribution is the optimization of last 2 layers.**

I. Introduction

Neural Networks have proven to be useful for the task of pattern recognition in various domains such as speech data and images. Neural network architecture consists of several layers which are essentially a set of parameters which use output of previous layer as input and feeds the calculated output to the next layer. Neural Networks follow a supervised learning framework and are trained using sample data points. The network is fed the data points on which the network generates the output. Based on generated outputs, error in prediction is propagated back to the network parameters. These propagated errors are then used to update the parameters accordingly in a way to minimize the value of the loss function. This is known as backpropagation of errors and forms a major part of training a neural network. Training a neural network incorporates the task of feedforward of inputs through all the layers and then backpropagation of the errors through all the layers along with parameter updates. These tasks essentially correspond to a matrix vector multiplication which makes these individual tasks computationally expensive.

Our work is centred towards the implementation of a Neural Network architecture which consists of an input layer, a hidden layer and an output layer. As a part of this implementation, the execution is further optimized in terms of better use of shared memory in the final layer along which reduction in kernel calls that are made for each iteration of the algorithm. We have trained the network for the task of handwritten digit recognition using MNIST dataset [1] which consists of 60,000 human labeled handwritten digits and labels. The labels are integer values from 0 to 9.

The network architecture that we are optimizing is popularly known as simplenet and is used in several domains such as image classification, parameter tuning and ensemble models.

[1]For this project we use Multilayer Perceptron form of NN. The terms MLP and NN have been used interchangeably in this report.

II. Related Work

In the last decade, there has been tremendous use of GPGPUs to build efficient machine learning algorithms and harness inherent data parallelism. There have been several attempts at parallelizing artificial neural networks. A MLP(MultiLayer Perceptron) based text detection algorithm was implemented by Jang et al. [2] for solving the problem of text detection using OpenMP and CUDA. In their CUDA implementation each multiplication in a single layer was handled by a single thread. They have proposed implementation of neural networks on both GPU and multi-core CPU and reported a speed up of 15 times compared to implementation using only CPU, and about 4 times compared to implementation on only GPU without OpenMP.

A similar approach is used by Huqqani et al. [3] using a single hidden layer network for face recognition using OpenMP and CUDA. Their results show that CUDA implementation outperforms OpenMP in terms of execution time when number of hidden layer neurons are more than 64. The reported speedup for GPU implementation was 22 times as compared single threaded CPU application.

Another similar implementation of MultiLayer Perceptron with single hidden layer is done by Brito et al. [4] for handwritten digit recognition problem. In all the above mentioned implementations, the parallelization is performed at each layer by allocating each output neuron to one thread block and all the threads within a threadblock perform computations for generating output of the given neuron.
The **base paper** for this project is [4]

The implementation details for kernel calls reported by [4] are shown in Fig 1.

In the implementation depicted in [4], all the layers are treated in a similar way to generate outputs and the whole execution algorithm is divided into 5 kernel calls. To the best of our knowledge, there is no work that exploits the fact that final layers in a MultiLayer Perceptron generally use small number of weights and hence all the computations pertaining to the feedforward and backpropogation of final layer can be performed using only small amount of memory. The focus of this paper is to exploit the low memory requirement of final layer in an artificial neural network. Also, we have tried

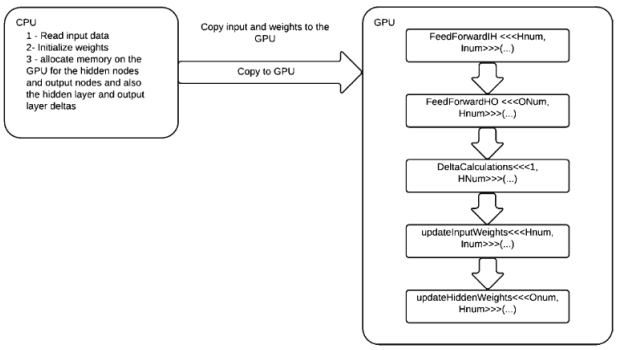

Fig. 1. Implementation methodology currently prevalent [4]

to accomplish optimization over number of kernel calls for computation of variables for final layer.

III. Contributions and Improvements over Baseline

1) The approach proposed by Brito et al. [4] performs 5 kernel calls. In our approach we perform only 3 kernel calls, hence reducing the overhead.
2) Generally the last 2 layers have less number of nodes and hence the parameter updates are less as compared to initial layers. Therefore, we can do the computations for last 2 layers within shared memory of a single threadblock.

IV. Methodology

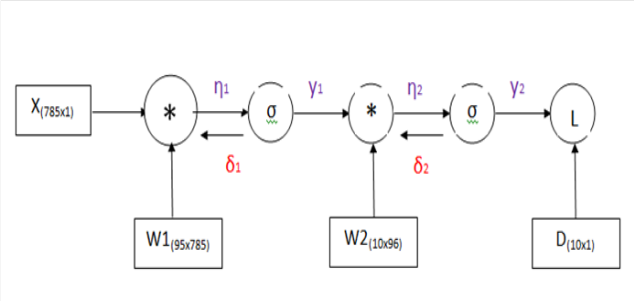

Fig. 2. Computational Graph Representation. Purple colored variables are calculated during Feedforward and red colored variables are calculated during Backpropagation

V. Implementation Outline

Memory allocation on GPU

1) Input data: $X \in \mathbb{R}^{785}$
2) Input label: $d \in \mathbb{R}^{10}$
3) Weights between hidden and input layer: $w_{hi} \in \mathbb{R}^{95*785}$
4) Weights between output and hidden layer: $w_{oh} \in \mathbb{R}^{10*96}$
5) Output of hidden layer: $y1 \in \mathbb{R}^{95}$
6) Error of hidden layer(for backprop): $\delta 1 \in \mathbb{R}^{95}$

Note: We are working in augmented feature space, hence the bias for input and hidden layer is included in the weights w_{hi} and w_{oh}.

FEED FORWARD

For all layers:

$$\eta_j^\ell = \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1}$$
$$y_j^\ell = f(\eta_j^\ell)$$

Fig. 3. Equations used for feed forward

BACKPROPOGATION

For Final Layer:

$$\delta_j^L = (y_j^L - d_j^s) f'(\eta_j^L)$$

Hidden layers:

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} w_{js}^\ell \right) f'(\eta_j^\ell)$$

W updates:

$$w_{ij}^\ell(t+1) = w_{ij}^\ell(t) - \lambda \delta_j^{\ell+1} y_i^\ell$$

Fig. 4. Equations used for Backpropogation

Mem copy to device: Following values are copied to global memory of GPU before the start of algorithm.

1) X 2) d 3) w_{hi} 4) w_{oh}

Note:w_{hi} and w_{oh} have been initialized to random values in the interval [-0.25, 0.25].

We work in single precision floating point arithmetic, hence all elements are stored as 4 byte floats.

VI. Proposed Approach

A. Feedforward 1 (*Feed1*)

1) The computations pertaining to a hidden node are handled by a threadblock, by doing so we ensure coalesced memory access. Therefore, in kernel 'Feed1' (refer Fig 6) we launch 95 threadblocks and 785 threads in each threadblock.
2) Within a threadblock, each thread performs a multiplication between an input node and the corresponding weight in w_{hi}. This step is inherently parallel in a Neural network.
3) The reduce sum operation is done in shared memory and at last, thread #0 of each threadblock writes the result (after applying sigmoid activation function) to the global memory. (Refer Fig 5)

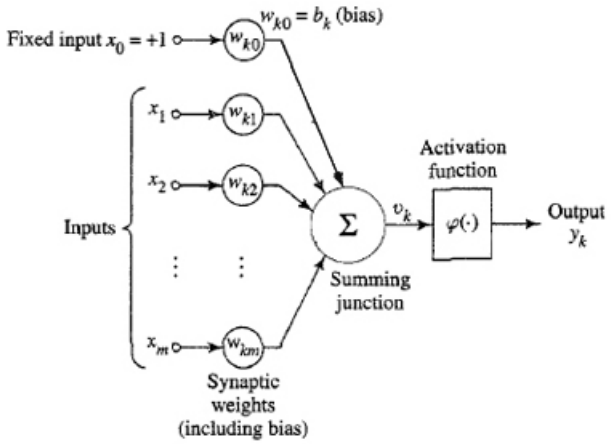

Fig. 5. Illustration of computations in hiddenlayer during Feed1 [5]

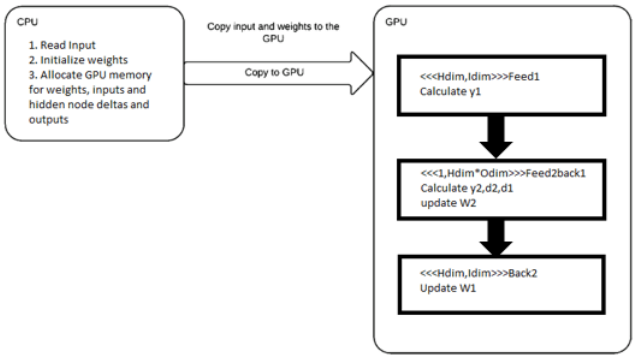

Fig. 6. Flowchart of our implementation

B. Feedforward 2, Backprop 1, Weight updation (***Feed2Back1***)

1) The array w_{oh} is of size 960. We do 1 multiplication per thread and hence all multiplications can be done in one threadblock.
2) This approach allows us to make maximum use of shared memory . Also, we are not moving back and forth from global memory thereby reducing the access times.
3) The reducesum operation has to be performed in back-propagation stage as well, but this results in strided (uncoalesced) global memory access. To overcome this issue, we copy all 960 elements of the weight array into shared memory and work with it in Feedforward 2 as well as in Back-propagation 1.
4) The sequence of computations is:
   a) Multiply y1 (shared mem) and w_{oh} (shared mem) and perform reducesum to obtain y2 in shared memory.
   b) Use y2 (shared mem) and d (coalesced global memory) to compute $\delta 2$ in shared memory.
   c) Use $\delta 2$ (shared mem), w_{oh} (shared mem) and y1 (shared mem) to calculate $\delta 1$. Do reducesum operation in shared memory and write $\delta 1$ to global memory.
   d) Use $\delta 2$ (shared mem) and y1 (shared mem) to update w_{oh} in global memory.

C. Back-propagation 2 (***Back2***)

1) This kernel is used to update w_{hi}.
2) We fetch $\delta 1$ and the current Input X from global memory (coalesced access) and apply the stochastic gradient descent update rule to update w_{hi}.

Our approach in Fig 6. makes fewer kernel calls than the baseline as shown in Fig 1.

VII. Experiments and Results

A. Experiment Setup

GPU Model: Tesla K40c
Compute capability: 3.5

We have made comparisons of our proposed approach with the baseline paper as well as Keras [6] which is a state of the art library that implements neural networks (Tensorflow backend).

B. Results

Performance Comparisons

Metric	Our Implementation	Keras-GPU	Keras-CPU
Time per Epoch(sec.)	4.95	17.5	82
Accuracy (in %)after 1 epoch	80.2	85.7	85.4
Speedup	1	3.53	16.56

Note: The baseline implementation in [4] uses 64 nodes in the hidden layer. Their runtime is 5.51 seconds and they achieve an accuracy of 92%. This cannot be directly compared to our implementation as we use 96 nodes in the hidden layer and a simpler update rule. In terms of runtime, our approach is superior as we use more hidden nodes but still achieve lesser runtime than [4]

Profiling Results (each kernel)

Metric	Feed1	Feed2Back1	Back2
Occupancy(in %)	78	94	78
Execution time per epoch (in sec.)	0.941	0.461	0.508

Note: The above readings were taken using the Nvidia Visual Profiler.

C. Occupancy Graphs

The following graphs were generated using Cuda Occupancy calculator [7] released by Nvidia.

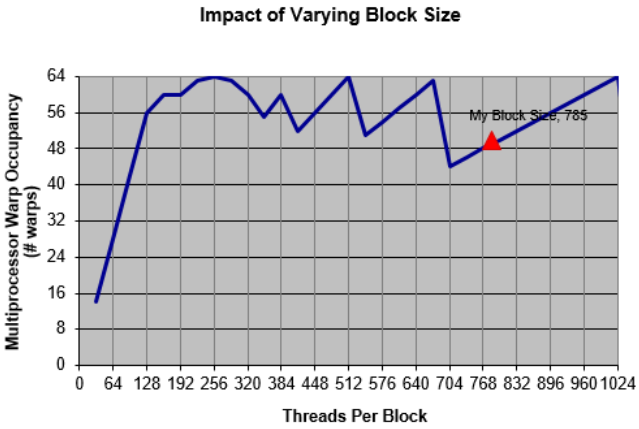

Fig. 7. Occupancy graph for Feed1

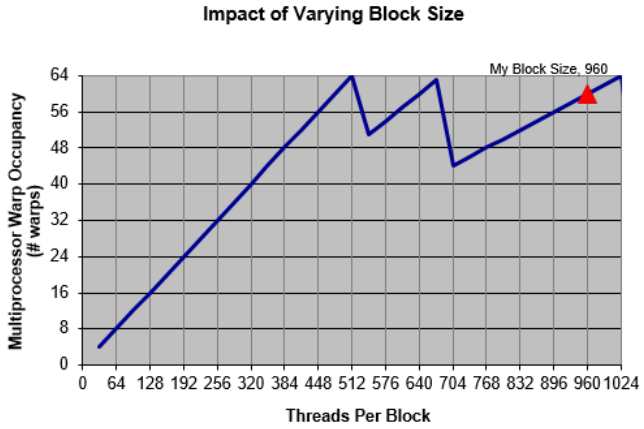

Fig. 8. Occupancy graph for Feed2Back1

The occupancy graph in Fig 8 corresponds to the kernel 'Feed2Back1' where we use maximum use of shared memory and it has an occupancy of 94%, also the time taken for execution of this kernel is lesser than other 2 kernels as reported earlier.

VIII. Conclusions

1) The previous approach [4] was using the same concept of parallelization throughout all the layers in a Neural Network. In our work we found that not all layers have the same size and shape. Initial layers have more parameters and the complexity decreases as we reach the last layer. We exploited this general property and were able to show promising results in terms of speedup.
2) This project demonstrates that we can decrease the training time of a Neural Network by exploiting the fact the last 2 layers have less parameters and hence can be completely computed in shared memory of a single threadblock. This has two advantages:
   a) Shared memory is much faster than global memory, hence decreases access times.
   b) In [4] they had to make multiple kernel calls so as to ensure coalesced memory access. Our approach goes around this problem as we are using shared memory and coalescing is not an issue. Hence, we were able to compute parameters using a single kernel call instead of three.

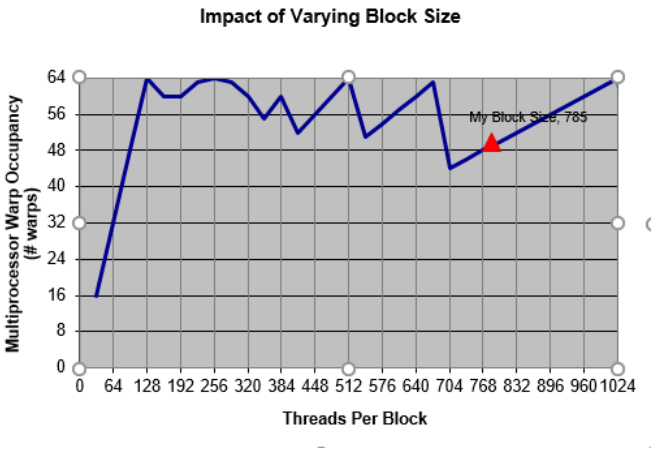

Fig. 9. Occupancy graph for Back2

IX. Future Work

Our work was primarily focused towards decreasing the training time of a neural network. We have adopted a conservative approach in terms of learning algorithm for the network. We believe that a better learning algorithm can be implemented within this framework without much impact on the training time. Specifically following modifications can be made:

1) tanh activation function instead of sigmoid activation function (this is straightforward).
2) Categorical cross-entropy loss instead of mean squared error loss.
3) Stochastic gradient descent with momentum instead of plain SGD.

References

[1] C. C. Yann LeCun and C. Burges, "Mnist handwritten digit database." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[2] H. Jang, A. Park, and K. Jung, "Neural network implementation using cuda and openmp," in *Computing: Techniques and Applications, 2008. DICTA'08. Digital Image*. IEEE, 2008, pp. 155–161.

[3] A. A. Huqqani, E. Schikuta, S. Ye, and P. Chen, "Multicore and gpu parallelization of neural networks for face recognition," *Procedia Computer Science*, vol. 18, pp. 349–358, 2013.

[4] R. Brito, S. Fong, K. Cho, W. Song, R. Wong, S. Mohammed, and J. Fiaidhi, "Gpu-enabled back-propagation artificial neural network for digit recognition in parallel," *The Journal of Supercomputing*, vol. 72, no. 10, pp. 3868–3886, 2016.

[5] C. Sergiu, "Financial predictor via neural network," *Retrieved from Http Ilwww. codeproiect. com*, 2011.

[6] F. Chollet *et al.*, "Keras," 2015.

[7] C. NVIDIA, "Gpu occupancy calculator," *CUDA SDK*, 2010.