Parallelizing fully connected networks

Sreekar Garlapati
150050065
CSE Dept.
IIT Bombay
sreekar@cse.iitb.ac.in

Sai Teja Gutta
150050072
CSE Dept.
IIT Bombay
saitejag@cse.iitb.ac.in

Sai Prakash Reddy
150050075
CSE Dept.
IIT Bombay
saiprakash@cse.iitb.ac.in

Yashasvi Sriram
150050080
CSE Dept.
IIT Bombay
yashsriram@cse.iitb.ac.in

***Abstract*—We paralleize Fully Connected Neural Networks using CUDA, openMP and openBLAS approaches. We show that CUDA approach outperforms all other approaches by a significant margin. We show that CUDA approach takes almost constant time for a large range of problem sizes. We also illustrate the strengths and weakenesses of a CPU and a GPU in the process.**

I. Introduction

Deep learning is a rapidly growing area of Machine Learning, of which neural networks are the basic building blocks. Neural networks are good examples of systems that can be significantly parallelized. In fact one of the reasons for recent explosion in research in the area is due to the advances in the field of parallel processing. Neural networks are made up of layers, each containing a bunch of neurons which are basic computation units in the neural networks. Fully connected, Convolution and Recurrent layers are popular ones among many others. In this paper we consider a fully connected network viz. network which only contains fully connected layers and try to parallelize it in different approaches.

II. Background

A fully connected layer can be modelled as a weight matrix W of shape MxN and a bias matrix b of shape 1XN where M is size of input and N is size of output to the layer. Given an input matrix i of shape 1xM we have output o given by $o = i * W + b$ where * is matrix multiplication and + is matrix addition.

A fully connected neural network is simply a sequence of such weight and bias matrices in sequence i.e. output of first layer is fed as input to second layer. This way input data is propagated forward in the network. This step is called Forward pass. After getting the final predicted output we compare that with true output and compute a loss function which decreases in value as predicted outputs get close to true outputs. The loss function is minimized by updating weights and biases of matrix using Stochastic Gradient Descent algorithm.

III. Concerns

A. Correctness

Theoretically a neural network is said to be correct or functional if it converges. To test for correctness empirically we can see if the accuracy increases with number of epochs trained. Figure 1 shows accuracy as a function of time.

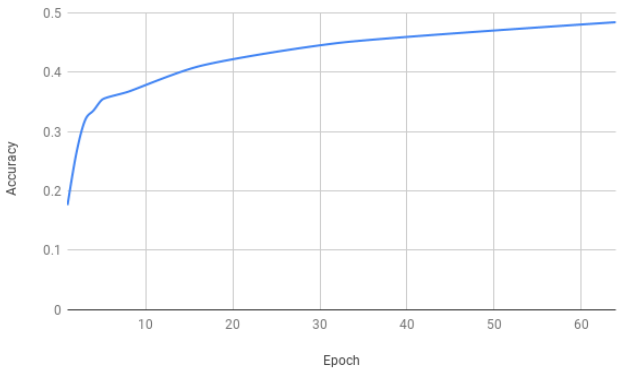

Fig. 1. Accuracy vs epoch

B. Memory

Valgrid's memcheck tool is used to profile memory usage. It is ensured that there were no memory leaks or other memory problems.

C. Reproducibility

Reproducibility refers to be able to reproduce same results obtained in serial algorithm. This is one of the major concerns while designing a parallel algorithm. In neural networks there can be several sources of randomness, random weights initialization and stochasticity in gradient descent being the main ones. However the randomness can also be reproduced by carefully setting random seeds.

IV. Task and dataset

We choose digit image classification as the task. Our task is a multi-class single-label classification and we use categorical cross-entropy as our loss function. We use a subset of the popular MNIST dataset for training and testing. We have 7490 train samples and 2510 test samples.

V. CPU Implementation

We use a matrix class which uses STL 1D vector of floats to store the values of a matrix. We implemented forward pass and back propagation steps of training in on CPU. At first we used a 2D vector of vector of floats which performed significantly worse. Double memory dereferencing and lack of good caching were main reasons for it. This forms a baseline to

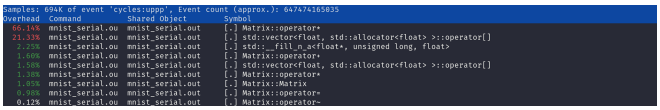

Fig. 2. CPU time profile

calculate speedups for other approaches. Figure 2 is a snapshot of time profile of CPU implementation. Matrix multiplication among others was the most time taking operation.

VI. CUDA APPROACH

CUDA is a natural approach to parallelize matrix multiplication operations. So we start with that. There were some roadblocks in this approach and a discussion of those constitute this section.

A. Naive matrix multiplication kernel

As matrix multiplication was the most time taking operation we tried to parallelize it by writing a CUDA kernel for it. But in contrast to our expectations this version performed significantly worse than CPU baseline. After profiling it with nvprof we found that cudaMemCpy was the dominant operation. The reason for this was because each time matrix multiplication was called two whole matrices were memcopied to device operated on and them memcopied back to host.

B. GPU serial baseline

To alleviate this memcopy situation we changed the STL vector which holds matrix values to thrust::device_vector, in other words we used device memory for allocating matrix values. In addition to this kernels for all matrix operations were implemented. To establish a new baseline in this version all parallelization was removed and the GPU was used as if it were another CPU.

C. GPU with parallelized kernels

After keeping a copy of GPU serial baseline all major time taking kernels were parallelized using CUDA threads and blocks. CUDA's nvprof was used as a feedback to decide which operation's kernel to parallelize. Kernel for matrix multiplication, addition, sigmoid, derivate of sigmoid, transpose were among the top few.

D. GPU with mini-batch kernels

After parallelizing kernels we found that mini-batch creation was taking the most amount of time. This was because each mini-batch is a matrix, whose values are allocated on device but the training data are loaded onto host. So while setting mini-batch matrix values from train data, the data had to be transferred to device from host many times. To prevent this we first collect training data into thrust::host_vector instead of STL vector and then simply copy entire training data to device using thrust = operator. Then we use a kernel to set mini-batch matrix values from device copy of training data. This is better than mini-batch data transfer because in the latter case, number of data transfers equals number of epochs times time number of data transfer in the former case.

E. GPU with compund kernels

Even after all this the GPU was lagging behind CPU which was far from expected. After a lot of code scanning we discovered the problem. In training and testing steps like forward pass include a bunch of matrix operations in sequence. Even though individual matrix operations were parallelized the whole operation was not per se. In other words for each operation a new kernel was launched and destroyed, which caused unnecessary creation and destruction of CUDA threads. To fix this all we had to do was to write a combined kernel for the whole operation.

For example, the forward pass during test data label prediction is given by the following equation.

$$out = sigmoid(in^{\mathrm{T}} * W + b)^{\mathrm{T}}$$

Instead of using separate kernels for transpose, matrix multiplication and addition we set matrix out values in a single kernel given in, W, b as input. Other compound operations for which kernels were written are as follows

$$out = (in^{\mathrm{T}} * W + b)^{\mathrm{T}}$$
$$temp = sigmoid(in_{\mathrm{i}})^{'} @ (W_{\mathrm{i+1}} * delta)$$
$$W_{\mathrm{i}} = W_{\mathrm{i}} + out_{\mathrm{i}} * delta^{\mathrm{T}} * learningRate;$$

where @ is element wise matrix multiplication. After implementing these kernels CUDA approach performed significantly better than all other approaches.

VII. OPENMP APPROACH

A. openMP

We used openMp to parallelize the martix operators. We saw a significant speedup compared to the CPU serial version due to the parallelization of matrix multiplication operation. The speedup due to OpenMP increased with increasing number of threads.

B. openBLAS

Matrix multiplication accounted for 60% of the time in CPU serial execution. We used an open source BLAS implementation called (surprise) openBLAS for the parallelizing matrix multiplication. This serves as a good baseline to test our openMP approach's efficiency.

VIII. SETUP

All the experiments in this paper was conducted on a linux machine with following specifications. Table VIII

Processor	i7-7700HQ
Processor clock rate	3GHz
Host memory	8GB
Device	GTX 1060
Device memory	6GB
Device clock rate	1.67 GHz
Max threads per block	1024

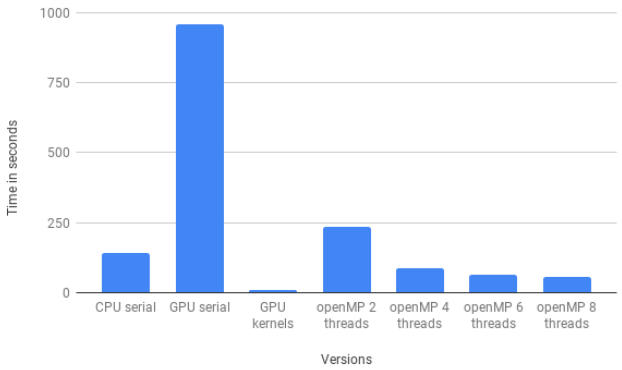

Fig. 3. Time taken for different versions

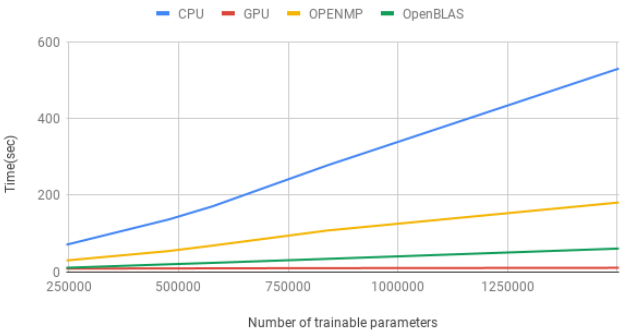

Fig. 4. Time taken for different versions

IX. Experiments and analysis

A. Version analysis

We compared time taken for a specific model to train with different versions of library. We use a network with 4 layers with 512, 128, 64, 32 number of nodes. We use 16 sized mini-batches, learning rate of 0.01 and train it for 2 epochs. Figure 3 shows times taken for different versions of library for training. We can see that GPU serial performs much worse than all others. This illustrates that the single core performance of CPU is much higher when compared to GPU. But in task involving parallel processing eventhough GPU has lower single core performance it outperforms CPU due to sheer number of cores.

B. N vs speedup analysis

Figure 4 shows times taken for all versions (except GPU serial due to sheer difference) for different problem sizes. We can see that CUDA approach beats all other approaches by significant margin. OpenMP is better than CPU but still lags behind openBLAS implementation. openBLAS comes close to CUDA approach but still takes more time as problem size increases unlike CUDA which takes almost same time for a large range of problem sizes.

X. Conclusion

We compare performances of different implementations of neural network viz. CPU-serial, GPU-serial, GPU-kernels, opemMP and openBLAS. We see that CUDA kernels outperform every other approach by significant margin and have almost constant runtime for a large range of problem sizes. We learn that thinking parallely is foundationally different from serial thining which was well illustrated when we implemented compound kerenels. We learn the strengths and weakenesses of both CPU and GPU and how they can be used together to overcome each other's shortcomings.

XI. References

https://abhyvyth.github.io/parallel-project/
http://www.cplusplus.com/reference/cstdlib/rand/
https://codeyarns.com/2011/02/16/cuda-dim3/
https://stackoverflow.com/questions/15669841/cuda-hello-world-printf-not-working-even-with-arch-sm-20
https://stackoverflow.com/questions/12164235/cuda-kernel-doesnt-launch https://stackoverflow.com/questions/12373940/difference-between-global-and-device-functions
https://stackoverflow.com/questions/44574692/odd-behavior-of-cudamemcpyasync-1-cudamemcpykind-makes-no-difference-2-copy
https://stackoverflow.com/questions/14539867/how-to-display-a-progress-indicator-in-pure-c-c-cout-printf
https://stackoverflow.com/questions/14221763/stdrandom-shuffle-produces-same-result-each-time

XII. Code usage

The project has the following structure

- data - contains mnist data
- demos - contains main files and Makefile
- lib - contains implementations of Matrix and FullyConnectedNetwork clases

Go to demos directory. To make all demos run **make** To compile a specific [demo].cpp

- **make [demo].out**