Parallelizing Neural Networks

HPSC Project Presentation

Problem statement

- Deep learning is a rapidly growing area of Machine Learning
- Neural networks are buildblocks
- The process of training neural networks consists of forward pass and back propagation steps
- In backprop uses gradient descent to update parameters and thus learn!

Problem statement

- Neural nets have a great potential to be parallelized
- In this project we consider the simplest type of neural net called Fully Connected Network
- A Fully connected network contains only so called fully connected layers in which each node takes input from all nodes of previous layer

Dataset

- We use MNIST dataset where input is a 28x28 grayscale image of a handwritten digit
- The task is to classify digit i.e. to determine which one it is from 0 - 9
- This is a multi-class single-label kind of classification
- Categorical cross entropy loss is used

Basic Abstraction

- The forward pass and back prop can be seen (computationally) as Matrix multiplications and additions.
- Therefore a Matrix naturally becomes a basic abstraction

Serial approach (Baseline)

Serial approach

- A matrix class to hold values and other metadata like number of rows and number of columns
- In the baseline (or serial) version uses a std::vector<float> to store the matrix in ROW-MAJOR order
- At first we used vector<vector<float>>, which took relatively a lot more time to run
- This was one of the first optimizations in the process

Serial approach

- A FullyConnectedNetwork class holds the information about number of layers and number of nodes in each layer
- User can add layers one by one
- Network has to be compiled by calling compile() function in which the actual matrices representing weights and biases of neural network are allocated and initialized
- A fit() function which takes a list of inputs and corresponding list of outputs from training data uses (mini-batch stochastic) gradient descent to update weights of network

Serial approach

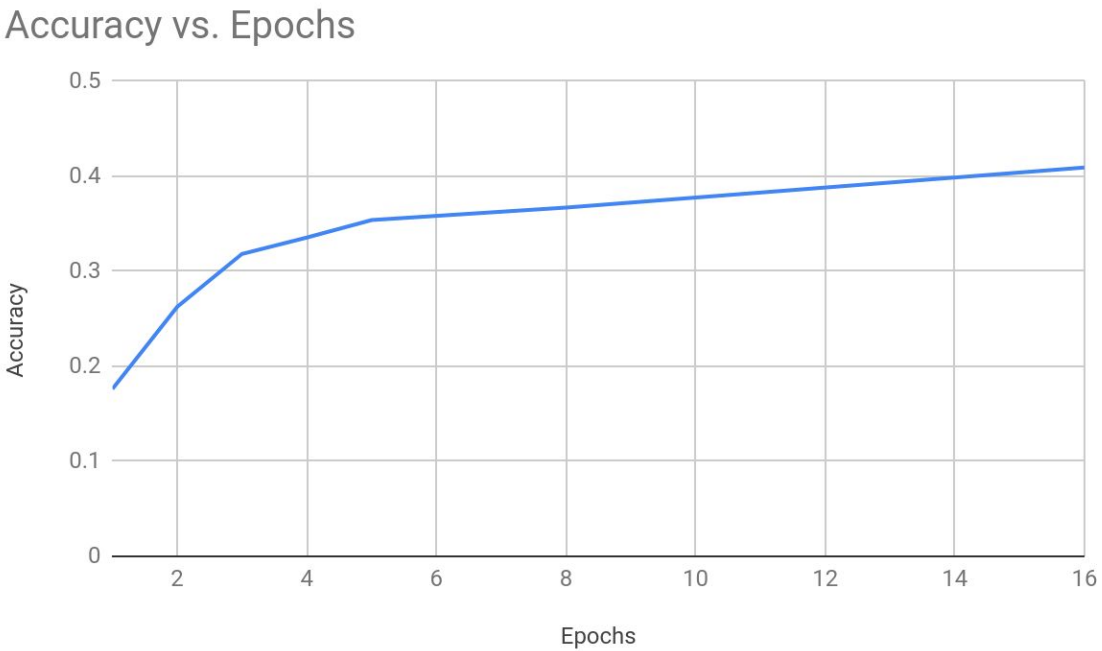
- The training and testing data is read, the network is configured and compiled
- Then batches of fixed size of samples and their labels are made from training data by randomly picking without replacement from the whole dataset
- A fit() is called on each mini-batch
- After training for a certain number of epochs the network with weights is tested on testing dataset to get accuracy
- Accuracy = total predicted correctly / total in testset

Correctness concerns

- Checking correctness in neural network is not a very defined task
- But we can test for whether the accuracy increases with number of epochs trained

Next figure shows accuracy as a function of epochs for a fully connected network with 4 layers of number of nodes 512, 128, 64, 32 with 16 mini-batch size and 0.01 learning rate

Correctness



Reproducibility concerns

- Parallelizing a task can only be justified if the parallelized system produces exact same result as the original one
- With neural networks which contain inherent randomness due to initialization of weights and stochastic gradient descent this can be a major problem
- But fortunately randomness in C can be controlled by setting a random seed at the start of program

Memory concerns

- All versions are checked with valgrind’s memcheck tool for memory leaks and other bad stuff
- No such leaks were detected

CUDA approach

CUDA approach

- The cuda approach had a lot of iterations
- Matrix multiplication took the most amount of time in CPU serial version
- So we tried to parallelize it first

CUDA approach - Iter 0

- Used a CUDA kernel to parallelize matrix multiplication
- At this point the matrices were allocated in the host and mem copied to device whenever matrix multiplication was needed (similar to HW3 statement)
- To our surprise, this actually increased runtime quite a lot!
- After profiling it with nvprof we found that reason was the frequent memcpys of whole matrices across host and device

CUDA approach - Iter 1

- To alleviate the frequent memcpys we allocated all matix data on device itself using thrust::device_vector
- This choice also means that any operation on matrix had to be done on device itself as if it is done on host it would require memcpy b/w host and device
- In this version we chose to NOT parallelize matrix multiplication
- This was all operations are done on device and all operations are serially done
- This establishes a GPU-serial baseline (||er to CPU-serial)

CUDA approach - Iter 2

- The obvious improvement from GPU-serial version is to use blocks and threads in kernels and parallelize functions
- Using nvprof as a feedback we parallelized *****, **copy constructor**, **+**, **multiply by scalar** functions

CUDA approach - Iter 2

- After using parallelism in kernels the most time taking operation was the mini-batch creation
- This process requires to stochastically select a fixed number of samples from training data and make appropriate input and output matrix objects

CUDA approach - Iter 2

- However the training data is loaded as a thrust::host_vector and setting matrices (which at this version are allocated in device) using host_vector causes memcpys
- To resolve this first all training data was copied to device and two kernels were created to set values for input and output matrices of that batch from the whole training data
- This gave a speedup of 3.96 wrt Iter 1.

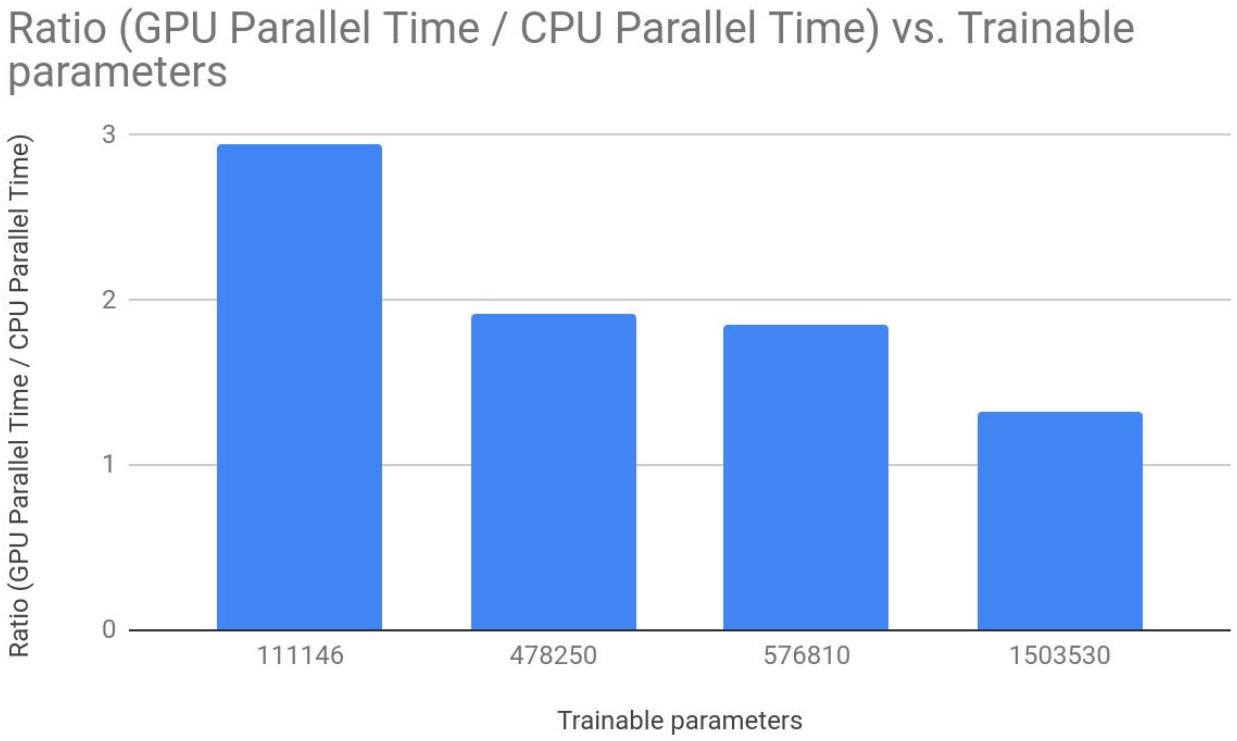
CUDA approach - Iter 3

- In this version we try and replace our hand coded kernels with cuBLAS and thrust routines
- This speedup achieved is 1.04 wrt Iter 3.

CUDA approach

CUDA seems to have a lot of parallel overhead in terms of memcpys. The gain by parallelization is generally surpassed by the parallel overhead for small problem sizes. For large problem sizes this gap becomes small.

The next figure shows the CPU-serial vs GPU-parallel for various problem sizes


Ratio (GPU Parallel Time / CPU Parallel Time) vs. Trainable parameters
Ratio (GPU Parallel Time / CPU Parallel Time)
3
2
1
0
111146
478250
576810
1503530
Trainable parameters

OpenMP approach

Openmp approach

- Matrix Multiplication takes most time in the CPU serial code
- We used openmp parallelization on Matrix Multiplication
- This gives a relative speedup which increases with the number of threads

Summary

Runtime comparisons

We used a fully connected network with 4 layers of 512, 128, 64, 32 number of nodes. The problem size can be taken as number of trainable parameters = 478250

The following graph summarizes average time taken for 2 epochs with mini-batch size 16 and learning rate 0.01

Runtime comparison