The Multi Layer Perceptron

Using parallelization for faster training times

Introduction

In this presentation we will cover the following :

- Brief review of a multi layer perceptron.
- Model parallelism and data parallelism.
- The MNIST dataset.
- Our serial, CUDA and MPI implementations and results.
- Scaling the implementations.
- Notes on use of resources.

The Multilayer Perceptron

- Also known as a fully connected feed forward neural network.
- Information processing technique inspired by the human brain.
- Consists of layers of artificial neurons. Each neuron’s input is the sum of the output of all the previous neurons times a weight.
- Training times increase as the size of the network and training data increases.

Parallelization techniques

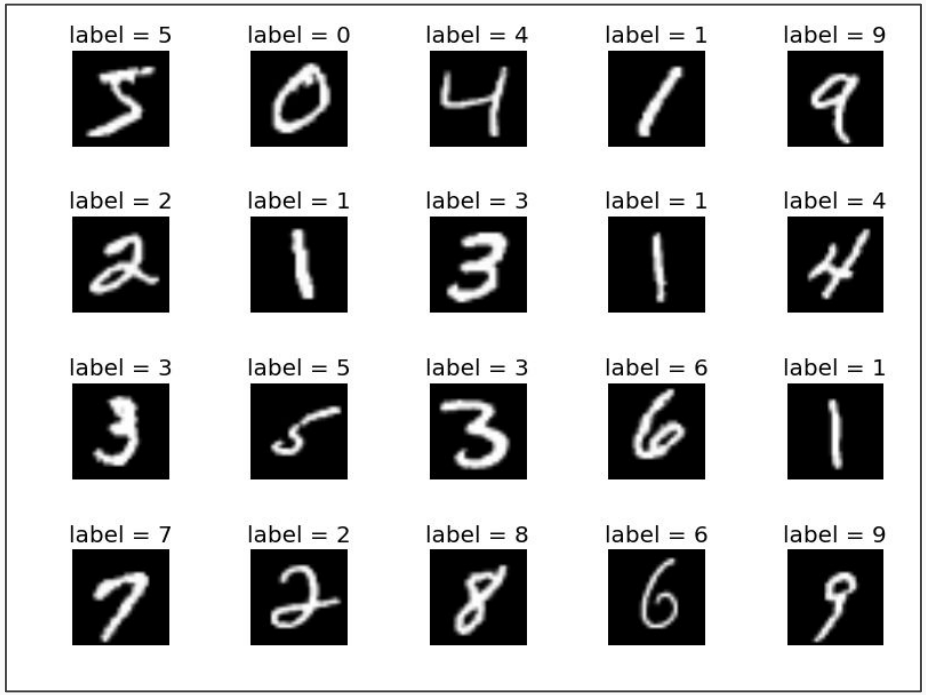
1. Model parallelization
2. Data parallelization

Model Parallelization

Each worker is responsible for processing a part of a single model. In the context of a neural network each neuron's output could be computed by a worker since there is no data-dependency on other workers. There is still dependency between layers. For a neuron to compute it's input it requires the previous layer to have computed it's output.

Data Parallelization

Each worker has the same model but sees different parts of the data. We could divide the training set by the number of workers available and assign each worker their own subset of training data. We need to combine the learned parameters from each worker somehow.



MNIST Dataset

- Handwritten digits.
- 70000 images.
- 28 x 28 pixels.
- 1 Byte precision.

Our implementation

Serial

- No parallelization.
- Layer based structure.
- Weights are flattened into a 1d array.
- Floating point numbers.

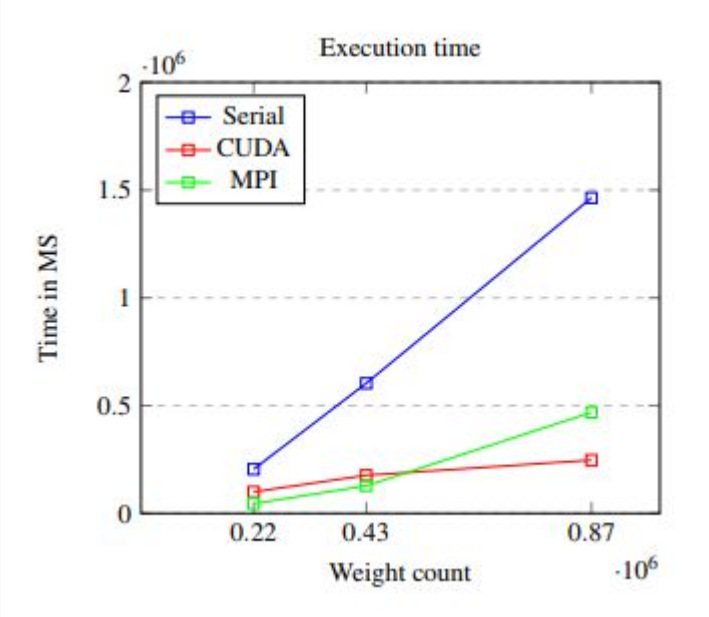
Cuda

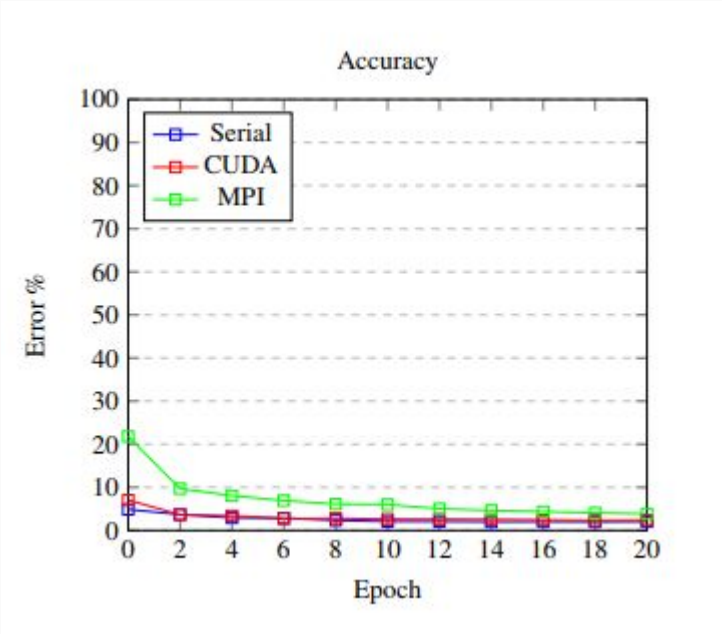
- Model parallelization.
- Each layer has a forward and backward kernel.
- Kernels are queued while they execute to reduce kernel launch overhead.

MPI

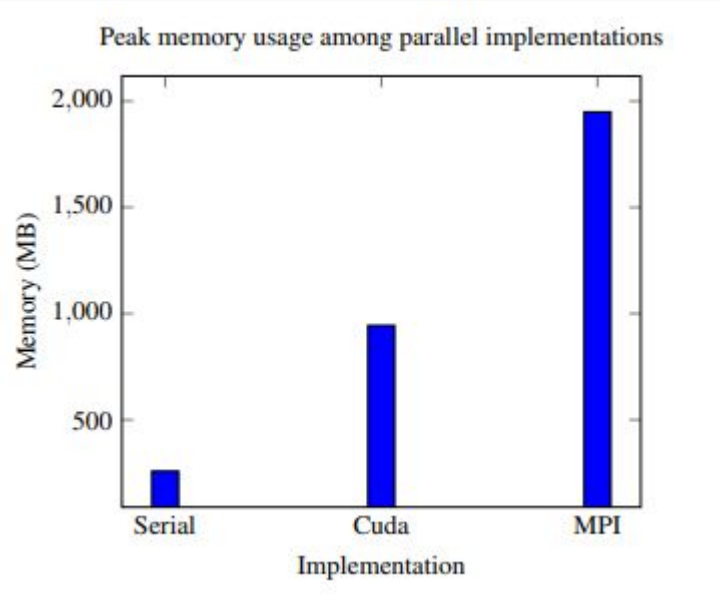
- Data parallelization.
- Each node has its own network.
- Each node has entire dataset.
- Train for n/(process count)
- Parameter averaging at the end of an epoch.

Results


Execution time
$\cdot 10^6$
2
1.5
1
0.5
0
Time in MS
Serial
CUDA
MPI
0.22
0.43
0.87
Weight count
$\cdot 10^6$


Accuracy
100
90
80
70
60
50
40
30
20
10
0
Error %
Serial
CUDA
MPI
0
2
4
6
8
10
12
14
16
18
20
Epoch

Results cont..



Resource use

Serial

- Only uses one CPU core.
- CPU memory access is highly optimized with L1 L2 and L3 caches, and delivers impressive performance.

Cuda

- Uses 100% of the GPU multiprocessor's capacity.
- The CPU is only responsible for invoking kernel launches and loading data.
- CPU is mostly idle. This can be improved.

MPI

- Each core of a node's CPU runs the a worker which has its own network it trains.
- The node's GPU is not used. This is an opportunity for future work..

Scaling up

Serial

- For the serial implementation to train quicker one can only improve the underlying hardware. A faster CPU would yield quicker training times.
- Faster and larger cache memory would help as well.

Cuda

- The cuda implementation shines when the model is very large.
- Multiple GPUs can be used since memory is likely to be the bottleneck on larger datasets. Imagenet 2012 paper.

MPI

- The MPI implementation is promising in that one can add more nodes to improve training time.
- At some point however the batch size will be so small that the cost of communicating the parameter updates overshadow compute gains.

Conclusion

Parallel neural networks :

- Trains faster. This allows for rapid experimentation which is something that is missing from current deep learning research.
- Converge slower when parameters are averaged. Thus we need to train for more cycles over the same data points (more epochs). Results from the gradient update being less refined.

Thanks!

Email:

kobusvdwalt9@gmail.com

Project on Github:

https://github.com/Kobusvdwalt/hpc_term