IT477 GPU Project Report

Submitted To: Prof. Bhaskar Chaudhary

Riddhi Thakker(201601124) Purva Mhasakar(201601082),
Anery Patel (201601077), Kirtana Phatnani (201601210)

November 10, 2019

1 Context

1.1 Problem Description

Time consuming computations involved in the training phase of an artificial neural network limits the number and type of problems we can leverage the network for. Thus, we aim to parallelize an artificial neural network (ANN)[4] using CUDA for the hand written digit recognition task. We use MNIST dataset containing 60,000 training samples and 10,000 test samples to train the network. We harness the power of GPU by applying parallelization techniques for an efficient and fast implementation of ANNs.

1.2 Algorithm Complexity

Serial Complexity of recognizing handwritten digits task: Assuming we have e number of epochs, t number of training examples, n number of input features per example and w number of neurons, the approximate time complexity is $O(e \times t \times n \times w)$. Considering the worst case scenario, we get $O(n^4)$ for forward propagation in the training phase while the backward propagation takes $O(n^5)$.

1.3 Profiling information

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 64.06   4922.36  4922.36  5586993     0.88     0.88  backward_prop()
 35.72   7666.71  2744.35  5586993     0.49     0.49  forward_prop()
  0.26   7686.44    19.73    60000     0.33   128.21  training_process()
```

```
  0.08   7692.51     6.07 771005034     0.00     0.00  sigmoid_func(double)
  0.00   7692.88     0.37    60000     0.01     0.01  take_input()
  0.00   7693.22     0.34  5706993     0.00     0.00  error()
  0.00   7693.56     0.34      601     0.57     0.57  write_matrix(std::string)
  0.00   7693.57     0.01        1    10.01    10.01  initialize()
  0.00   7693.57     0.00        2     0.00     0.00  std::operator|(std::_Ios_Openmode,
std::_Ios_Openmode)
  0.00   7693.57     0.00        1     0.00     0.00  _GLOBAL__sub_I_start
  0.00   7693.57     0.00        1     0.00     0.00  __static_initialization_
and_destruction_0(int, int)
  0.00   7693.57     0.00        1     0.00     0.00  details()


granularity: each sample hit covers 2 byte(s) for 0.00% of 7693.57 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00 7693.57                 main [1]
               19.73 7673.11   60000/60000       training_process() [2]
                0.37    0.00   60000/60000       take_input() [6]
                0.34    0.00     601/601         write_matrix(std::string) [8]
                0.01    0.00       1/1           initialize() [9]
                0.01    0.00  120000/5706993     error() [7]
                0.00    0.00       2/2
std::operator|(std::_Ios_Openmode,std::_Ios_Openmode) [16]
                0.00    0.00       1/1           details() [19]
-----------------------------------------------
               19.73 7673.11   60000/60000       main [1]
[2]    100.0   19.73 7673.11   60000         training_process() [2]
             4922.36    0.00 5586993/5586993     backward_prop() [3]
             2744.35    6.07 5586993/5586993     forward_prop() [4]
                0.33    0.00 5586993/5706993     error() [7]
-----------------------------------------------
             4922.36    0.00 5586993/5586993     training_process() [2]
[3]     64.0 4922.36    0.00 5586993         backward_prop() [3]
-----------------------------------------------
             2744.35    6.07 5586993/5586993     training_process() [2]
[4]     35.7 2744.35    6.07 5586993         forward_prop() [4]
                6.07    0.00 771005034/771005034     sigmoid_func(double) [5]
-----------------------------------------------
                6.07    0.00 771005034/771005034     forward_prop() [4]
[5]      0.1    6.07    0.00 771005034         sigmoid_func(double) [5]
-----------------------------------------------
                0.37    0.00   60000/60000       main [1]
[6]      0.0    0.37    0.00   60000         take_input() [6]
-----------------------------------------------
```

```
                0.01    0.00  120000/5706993     main [1]
                0.33    0.00 5586993/5706993     training_process() [2]
[7]      0.0    0.34    0.00 5706993         error() [7]
-----------------------------------------------
                0.34    0.00     601/601         main [1]
[8]      0.0    0.34    0.00     601         write_matrix(std::string) [8]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[9]      0.0    0.01    0.00       1         initialize() [9]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[16]     0.0    0.00    0.00       2         std::operator|(std::_Ios_Openmode,
std::_Ios_Openmode) [16]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [25]
[17]     0.0    0.00    0.00       1         _GLOBAL__sub_I_start [17]
                0.00    0.00       1/1           __static_initialization_
and_destruction_0(int, int) [18]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I_start [17]
[18]     0.0    0.00    0.00       1         __static_initialization_
and_destruction_0(int, int) [18]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[19]     0.0    0.00    0.00       1         details() [19]
-----------------------------------------------

Index by function name

  [17] _GLOBAL__sub_I_start    [8] write_matrix(std::string) [19] details()
   [9] initialize()            [3] backward_prop()      [6] take_input()
   [4] forward_prop()            [2] training_process()
[5] sigmoid_func(double)
   [7] error()         [18] __static_initialization_and_destruction_0(int, int)
[16] std::operator|(std::_Ios_Openmode, std::_Ios_Openmode)
```

We can observe from the profiling information of the neural network implementation that the maximum time for execution is consumed by the backward propagation and forward propagation in the training phase. Hence, we can devise some optimization strategy to specifically improve backward propagation and forward propagation performance in the code.

1.4 Theoretical speedup

From the profiling, we can infer that the time consumed by the entire algorithm is concentrated by the training phase of the network, that is forward propagation and backward propagation. Now, the computations of the activations of the neurons in same layer do not dependent upon one another, hence their calculations can be parallelized whereas the activations and weights of the different layers depend upon one another, basically there exists flow dependency among the layers, therefore, it is not possible to parallelize this part of ANN.
From the Amdahl's Law[1], suppose P are the number of processors, σ be the part of the code that has to be sequential, ϕ be the part of the code that can be parallelized and K be the overhead for synchronizations, kernel launch and extra computations for scheduling, control divergence, etc.

$$\text{Theoretical Speedup} = \frac{\sigma + \phi}{\sigma + \frac{\phi}{P} + K}$$

For 128 number of neurons in the hidden layer, we get the following information after profiling: ϕ is the total time to execute parallel code. In our case, it sums up to backwardprop() time plus forwardprop() time per epoch, per layer. Since layer wise and epoch wise computations are sequential, we divide the total parallel time by number of epochs and layers, assuming uniform distribution. Thus, ϕ is $(4922.36 + 2744.35)/(2 * 512)$ where the number of epochs are 512 and the number of layers are 2. Thus ϕ is equal to 7.487. Similarly, σ is equal to (220.355)/(2*512) = 0.2152. The number of processors are 1536. Using this information, the theoretical speedup is nearly 35.

1.5 Optimization Strategy

1. Dependency
   The multiplication of weights and input features for each neuron is independent of computations for other neurons in the same layer. Thus, we can leverage the benefit of independent computations of same layer to optimize our code. However, we cannot parallelise layer by layer computations since the flow is sequential.

2. Locality of Reference
   Global memory access plays a huge role in determining the throughput and latency. There is no usage of previous layer weights and inputs for computing next layer matrix multiplication. Thus, shared memory has no role in layer wise computations. However, we can use shared memory while performing matrix multiplication for the same layer.

3. CGMA Ratio (Compute to Global Memory Access)
   The CGMA[2] ratio is 1. It is calculated as follows: Assume the number of training samples are t, the number of epochs are e, the number of input features are n, the number of neurons in hidden layer are n_1, the number of output classes are n_2. We also add bias after each layer computation

which will be equal to the number of neurons in that layer. Thus, the number of computations equals $t \times e \times (n \times 2n_1 + n_1 \times 2n_2)$. Every time, a computation is performed, the weights, biases and inputs are accessed. Thus, the global memory accesses equals the number of computations. We can reduce the number of global memory accesses since we are using shared memory, thus we can optimise the CGMA ratio.

4. Thread Organisation model
   Given the hardware properties, we can make the best use of the given number of threads and blocks that can be launched at a time in GPU. By prudent or complete usage of all the available blocks, we can achieve peak performance corresponding to the hardware specifications.

1.6 Problems faced in parallelization and possible solutions

The main problem here is the size of the MNIST dataset, the image that is used is $28x28$, which is not a multiple of 32. This would mainly lead to control divergence. Moreover, the size of the data changes at each layer of the neural network, hence thread organisation cannot be specific to one layer, it has to be more of generalized for the entire neural network. The reuse of data is much higher, as it depends upon the number of epochs used (here they are 512), so the global memory access would be much more. Moreover, every data in the entire neural network gets modified after every call to kernel and at every epoch, and hence, the use of the constant memory is not possible in ANN. The usage of shared memory will be limited to the matrix multiplication for a single layer. Thus, the global memory accesses will decrease with the use of shared memory.

Now, parallelization cannot be done where there exists flow dependency, so we have execute that part of code serially. Hence, we have to launch kernel for (number of layers - 1) times each for forward propagation and back propagation.

Here, we are considering 3 layers.

The other issue was that the neural network has be trained for 60000 such images. After each training the values of the data will change, and hence each image will get different neural network. If we consider only one image for the parallel algorithm of ANN, then the resources will not be used effectively (example: 784*128 data), which is very less and we have to launch kernels for all the images i.e. 60000 * 4 (2+2 for forward propagation + back propagation). Hence, we decide to process a batch of 10000 images together, and after that the individual results will be combined through softmax method to get the resultant neural network after training. This resultant neural network will be the input of next batch of images. This will ensure the usage of resources.

2 Observations and Analysis

2.1 Comparison of Serial & Parallel Code

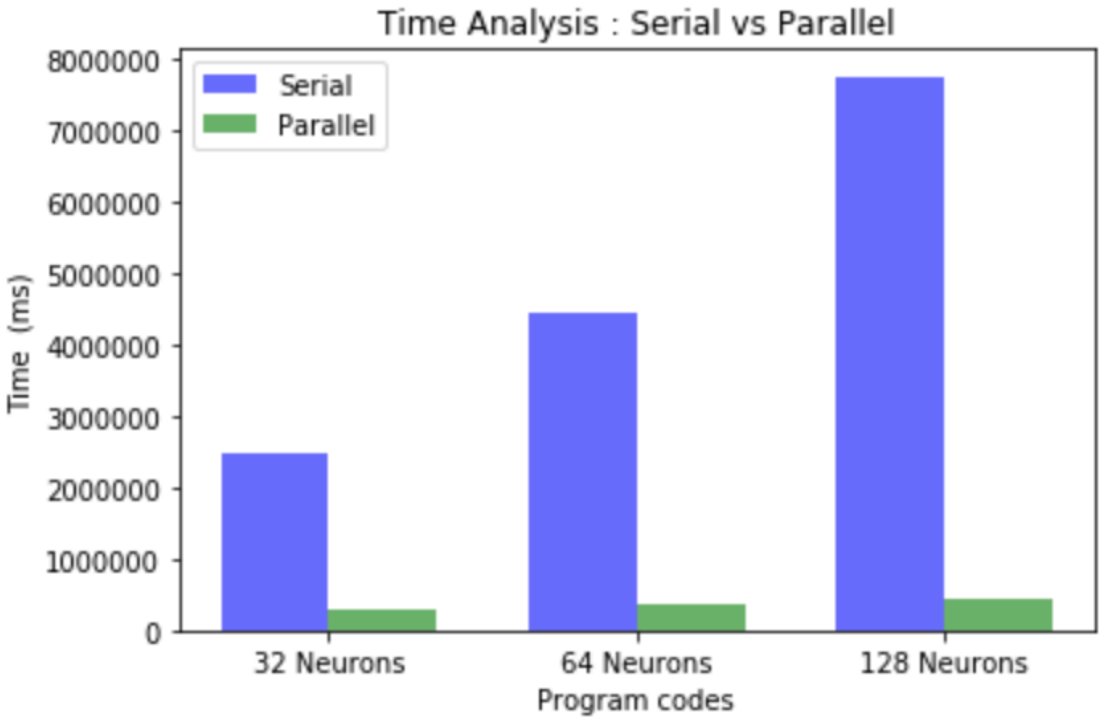

Figure 1: Layer Size VS Time for serial and parallel

The problem size for 32 neurons is approximately 2^{18}, that of 64 neurons is 2_{19} and 128 is 2^{20}. As the problem size increases, the time taken by the serial code will also increase significantly. But the time taken by the parallel code as we observe does not increase as the serial algorithm. Until the resources get exhausted, the increase in the parallel algorithm would be less.

2.2 Speedup Curve

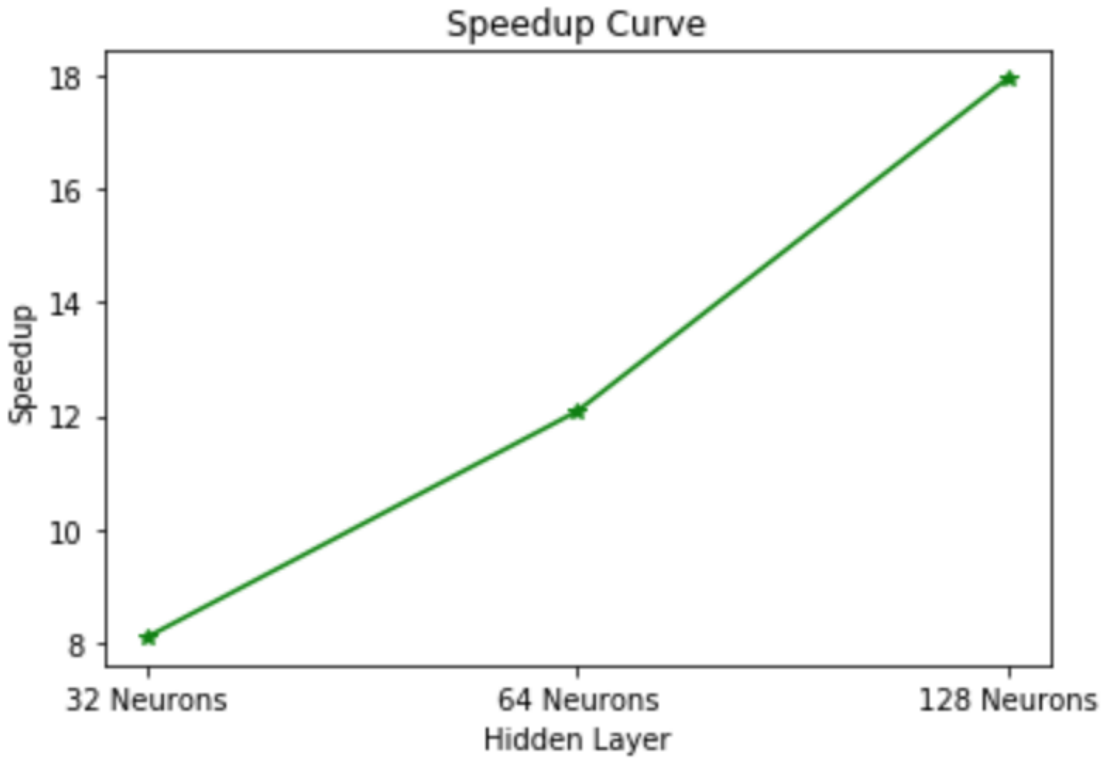

Figure 2: Layer Size vs Time for Parallel code

As the number of neurons n increases in the hidden layer, the dimension of the weight matrix varies as per n and hence, we obtain an hike in the number of computations for the Neural Network as shown in figure 1. We observe the serial code shows an exponential increase in computational time where as due to parallel programming we obtain a the increase in computational time is at a lesser rate for parallel code.

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

As the serial time increases the with the number of neurons and the parallel time remains almost constant we obtain the speedup increasing with n.

2.3 Effect of Block size on Speedup Curve

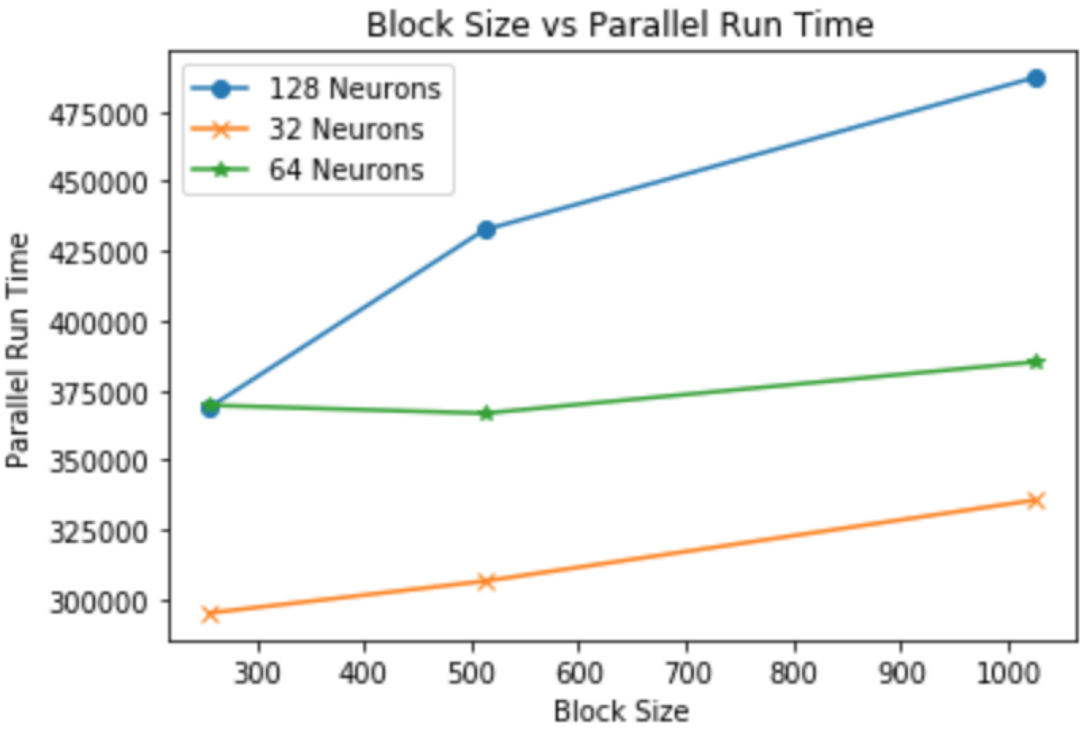

Figure 3: Block Size vs Parallel time for different layer sizes

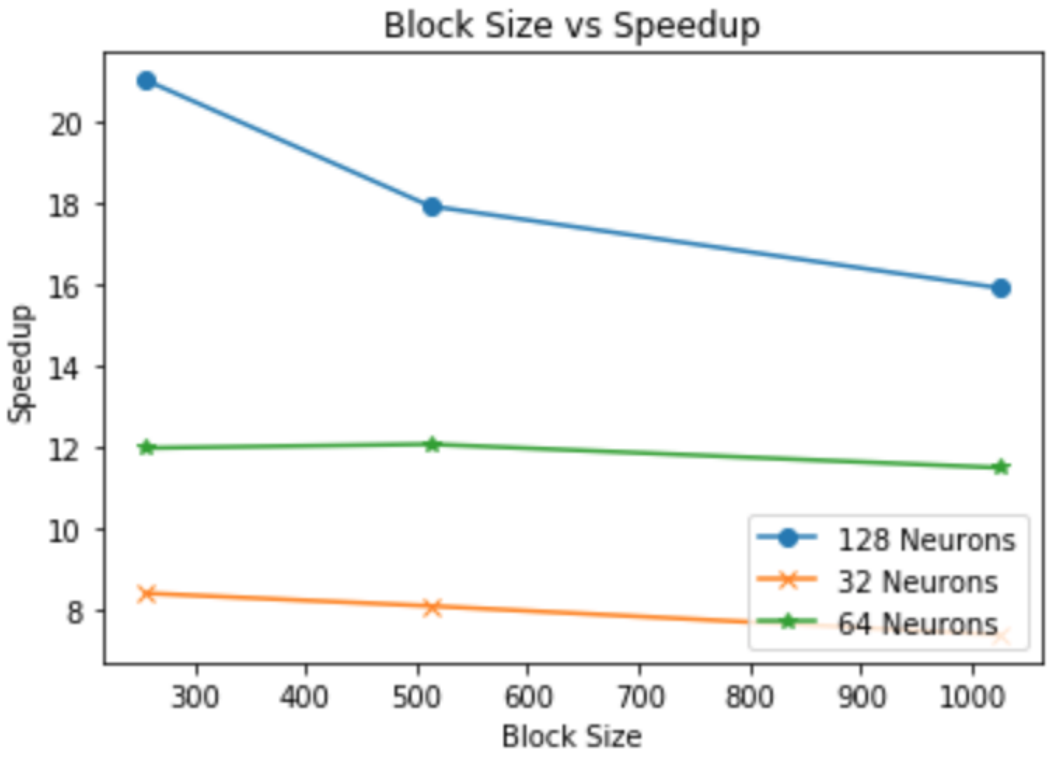

Figure 4: Block Size vs Speedup for different layer sizes

We use the block sizes 256,512 and 1024. As per the hardware properties, we can launch maximum 1024 threads at a time thus we do not increment beyond 1024 blocksize. We do not launch more than 1024 blocks because that will cause the processor to launch the blocks in phases, which in equivalent to running blocks serially. That will incur overhead costs in terms of time. Thus, using blocksize greater than 1024 is a bad choice for performance of the code.

2.4 No. of floating point operations

There are total 6 operations per thread in computation of sigmoid function, 6*batch_size operations per thread for softmax calculations and for calculation derivation and delta operation for the back propagation approximately n operations per thread.

2.5 Control Divergence

Here, in order to prevent the control divergence, we have to keep the number of neurons in each layer except the input layer in multiples of 32. If it is not

a multiple of two, the computational time increases due to increase in wasteful wraps.

2.6 References

1. https://www.sciencedirect.com topics computer-science amdahls-law
2. https://www2.cs.siu.edu mengxia
3. docs.nvidia.com cuda cuda-memcheck
4. https://skymind.ai wiki neural-network