2020

Parallelization of Neural Network

PROJECT WORK

KIRIT KHADE

1. Problem Statement

This section describes the background, problem statement and the scope of HPC in this project. This section also discusses the dataset used, and the background knowledge required for understanding underlying concepts of Neural Networks.

1.1 Background

Neural Networks are programming codes that broadly take inspiration from biological neural network comprising animal brain, which enables a computer to learn from observed data. A neural network contains layers with interconnected nodes, and each node is a mathematical function that is trained to identify the underlying relationship in the dependent(output) and independent(input) variables in a dataset. An illustration of the underlying input & output data from a neural network trained to recognise hand written numbers can be seen in the Figure 1. Furthermore, serial and parallel implementation of a 3-layered Neural Network will be done as a part of this project. The performance of the network across various implementation techniques will be discussed and the performance based on different resources (nodes and threads) will be compared.

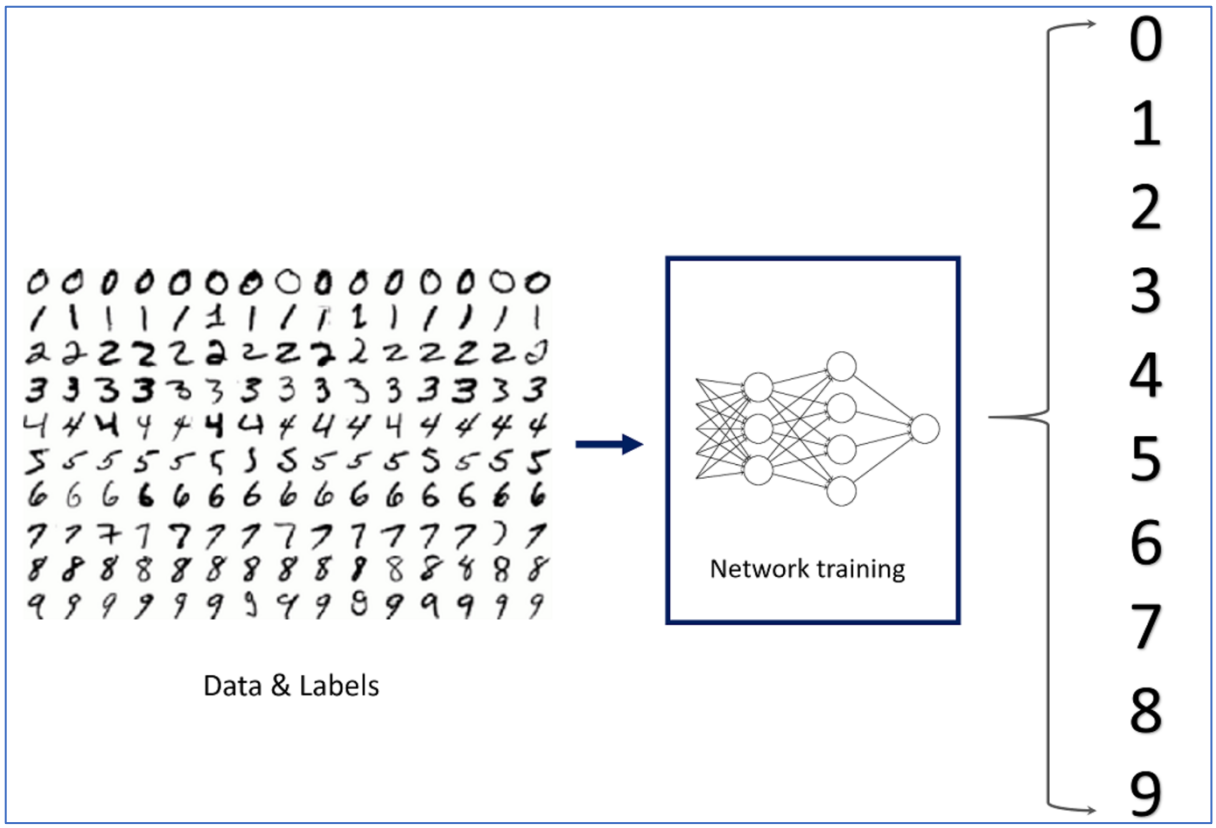


Figure 1: Neural Network input and output

1.2 Problem Statement

Parallelising the Neural Network computations to reduce the processing time

Since 1950s when neural networks were first conceptualised, continuous improvement in computation algorithms, computing power and quantity and quality of data has enabled the neural networks have revolutionized technology. Various cutting-edge applications such as Face Recognition, speech-to-text conversion would not have been possible without the

advent of neural networks. Conceptually the neural network has been around since 1960; the current growth in this technology is possible due to the advent of high-performance computing (HPC) (EXXACT, 2018). Depending on the amount of data and the algorithm used to train a neural network, the time to train a neural network model can vary from hours to weeks. High-performance computing (HCP) is used improve the training efficiency and to reduce training time. HPC parallelizes the various computations and iterations performed to train and test a Neural Network (Lim, 2019). In this project, the aim is to parallelise the program to reduce the total processing time required to train and test the Neural Network.

1.3 Scope of HCP

With the increase in the size of layers, the computations and run-time of the neural network increase. In this project, HPC can help save time in processing in the following way:

1. Neural Network training involves many matrix multiplication calculations; these operations can be parallelised.
2. Iterations in Neural Network calculation are often run in mini-batches. The training data is generally split into small batches in each iteration to calculate cumulative average error for the model. These mini-batches can be parallised to achieve run-time gains.

1.4 Dataset Details

For this project, the Neural Network will be trained on the MNIST dataset. The MNIST dataset is a database of hand-written digits, where the input is an image of a hand-written number and the output is the computerised label of the digit in the image. A sample image from the dataset can be seen in Figure 2.There are a total of 60,000 training samples and 10,000 test samples. Each input image is square grid of 28 * 28 pixels in size. Each image has an output label lying between 0 and 9. The aim of the neural network model is to predict the label of the image.

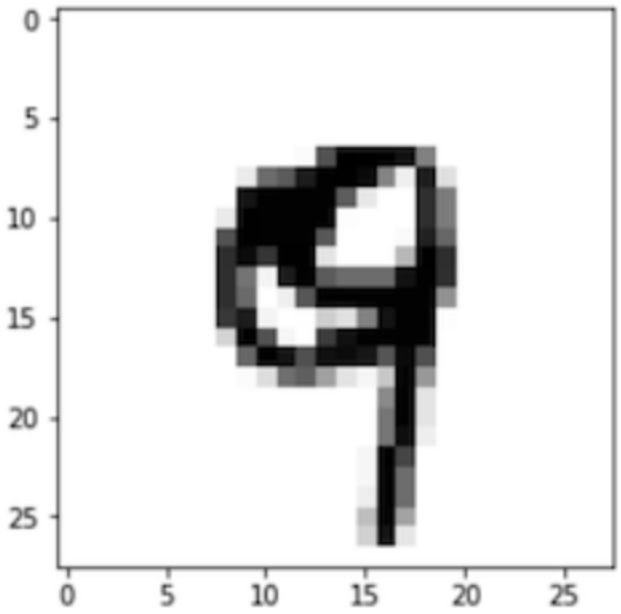


Figure 2: Sample MNIST data

1.5. Neural Network

In a Neural network, the first layer is the input layer that collects data from the 28*28 image and the last layer is the output layer. The output layer has a classification or output signal to which input patterns will be mapped. All the other layers in between the input and the output layers are known as the hidden layers. There could be multiple hidden layers in a neural network. The hidden layers help in identifying patterns and features in input data that will help the input data to map to output data (Chen, 2020).

A simple neural network can be seen as illustrated in Figure 3. In this neural network (Nielsen, 2019):

1. ***Input***: x_1 , x_2 , x_3 are the inputs of the neural network
2. ***Weight***: w_1 , w_2 , w_3 are the weights of the neural network; the weights are the real number expressing the relative importance of each input to the final output
3. ***Bias***: 'b' is the bias, that shifts the activation function by a constant
4. ***Activation function***: These are the mathematical function that determines the output of a neural network. A necessary condition for activation function is that they should be differentiable. In this example, the activation used is a 'Sigmoid' function.
   Mathematically, the sigmoid activation function($S(x)$) is defined as:
   $$S(x) = \frac{1}{(1 + e^{-(\sum_i x_i * w_i + b)})}$$
5. ***Output***: The output of the neural network is given by
   Output = S($\sum_i x_i * w_i + b$)

Neural network are often trained over many iterations. The termination of neural network training occurs when model stops to learn. A model can be said to have stopped learning when subsequent iterations stop influencing the weights and biases of neurons. There are many essential asspects of neural network traning, like Forward & Backward propogation, mini-batch and update step, which will be discussed in the following selctions.

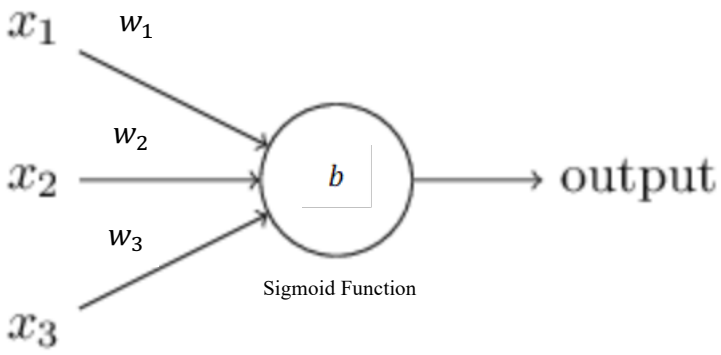


Figure 3: Simple Neural Network

1.5.1 Forward Propagation

In forward propagation, each layer and node process the data coming in it according to the weights, bias, and the mathematical function defined for it till that point. Considering that all the mathematical functions are sigmoid. The forward propogation is shown in **Error! Reference source not found.Error! Reference source not found.**

Input layer:

Since no input data comes in the input layer, there is no forward propagation defined in it.

Hidden layer:

$$hidden_j = \ sigmoid(\sum_{i=0}^{n} wh_{ji} * \ x_i \ + bh_j \ )$$

Where, $hidden_j$ is the output of hidden node j

wh_{ji} is the $weight\ between\ j^{th}\ hidden\ node\ and\ i^{th}\ input\ node$

x_i is the $i^{th}\ input\ node$

n is the number of inputs

bh_j is the $bias\ of\ \ j^{th}\ hidden\ node$

Output layer:

$$output_k = sigmoid(\sum_{j=0}^{m} wo_{kj} * \ hidden_j \ + bo_k)$$

Where, $output_k$ is the output node k

wo_{kj} is the $weight\ between\ k^{th}\ output\ node\ and\ j^{th}\ hidden\ node$

h_j is the $output\ of\ j^{th}\ hidden\ node$

m is the number of hidden layers (15)

bo_k is the $bias\ of\ \ k^{th}\ output\ node$

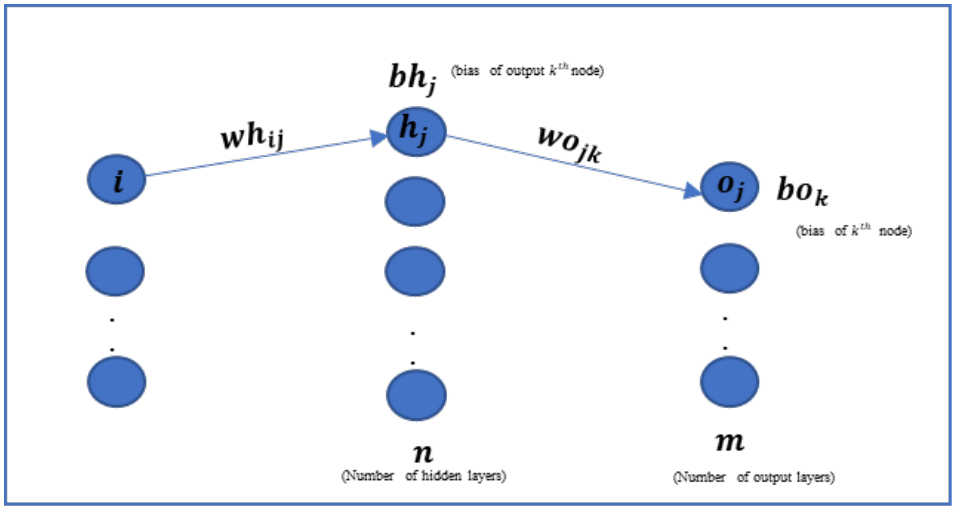


Figure 4: Forward Propogation

1.5.2 Back-Propagation

Neural network learns by repeatedly fine-tunning the weights and bias so as to minimise the model error or model loss. This fine-tunning is done by backpropagation algorithm that feeds the model loss backward. The model loss is defined by the difference between predicted output and the actual output (or the label of the image). In each iteration, the model loss is calculated and transferred back to each node, resulting in change of the bias and weights. (Al-Masri, 2019). The back propogation is shown in Figure 5

1. ***Error in each node:*** *Error or* loss in each layer is calculated by:

$$delta_{(l-1)j} = w_{ij} * delta_{(l)i} * f'(z)$$

Where,

$delta_{li}$ *is the error in node* i *of layer* l

$f'(z)$ *is the derivative of output of layer* $(l-1)$

$delta_{(l-1)j}$ and $f'(z)$ belong to the same layer

w_{ij} *is the weight between node* i *in* l^{th} *layer and node* j *in* $(l-1)^{th}$ *layer*

2. ***Error in each weight:*** Error in each weight is calculated by:

$$deltaweight_{(l-1)(l)ij} = z_{(l-1)j} * delta_{(l)i}$$

Where,

$delta_i$ *is the error in node* i *of layer* l

$z_{(l-1)j}$ *is the output from* $l-1$ *layer and node* j

$deltaweight_{(l-1)(l)ij}$ *is weight error in* i *node of* l *layer vs* j *node of* $l-1$ *layer*

3. ***Derivative of a sigmoid function:***

$$sigmoid'(z) = sigmoid(z) * (1 - sigmoid(z))$$

This can also be written as, $F'(x) = x * (1-x)$; where $x = sigmoid(z)$

Applying the above formulas in the 3-layers network:

1. ***Output Layer***:

a. $deltaoutput_k = (preidcted\ output_k - actual\ output_k) * F'(x)$;

Where $x = sigmoid(preidcted\ output_k)$

$deltaoutput_k$ *is the error in* k^{th} *node in the output layer*

This is because there is no layer beyond the output layer, hence no weight. The delta is calculated as the difference between predicted and actual output; which is also known as model loss

b. $deltaweight_{kj} = (delta_k) * hidden_j$

Where, $deltaweight_{kj}$ *is the weight between* k^{th} *node in output layer and* j^{th} *node in hidden layer*

k *denotes the* k^{th} *node of output layer,*

and , j *denotes the* j^{th} *node of hidden layer*

2. Hidden Layer:

a. $deltahidden_j = (deltaoutput_k) * F'(x) * weight_{kj}$;

Where, $x = sigmoid(hidden_j)$

$hidden_j$ is the output of hidden node j

$deltaoutput_k$ *is the error in output* $k - node$,

$weight_{kj}$ *is the weight between* k *node of output and* j *node of hidden layer*

b. $deltaweight_{ji} = (deltahidden_j) * input_i$

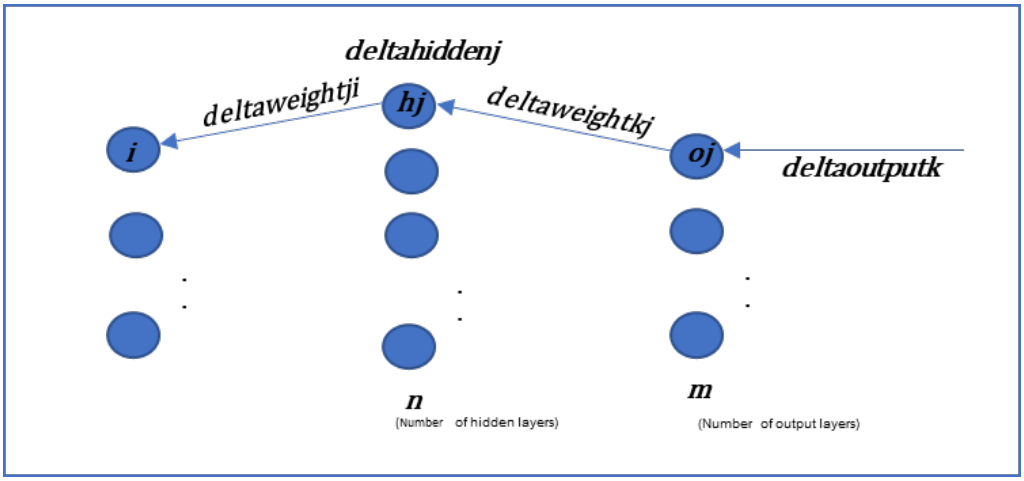


Figure 5: Back Propogation

Visually, it could be seen as:

https://gfycat.com/comfortableornerykangaroo-three-blue-one-brown-machine-learning

1.5.3 Mini batch

Mini-batch splits the training dataset into small-batches that are used to calculate the model error and update the model coefficients of the entire mini-batch at a time. (Brownlee, 2017)

1.5.4 Update of weights and bias

Weights are updated after each mini-batch is run. For the implementations, the size of the mini-batch is set to 10. For each datapoint in the mini-batch:

1. Forward propagation is calculated.
2. Backward propagation is calculated
3. All the errors from backward propagations from all the 10 datapoints are added
4. The weights and bias are updates in the following manner:
   a. ***Bias***: $bias_{li} = bias_{li} - lr * (sum\ of\ delta_{li})/(\ size\ of\ mini\ batch)$
   b. ***Weights***: $weight_{lij} = weight_{lij} - lr\ * (sum\ of\ deltaweight_{lij})/$ $(\ size\ of\ mini\ batch)$

Where, $bias_{li}$ *is the* i^{th} *bias of layer* l^{th}

$weight_{lij}$ *is the weight between* i^{th} *node and* j^{th}*node in* l^{th} *and* $(l-1)^{th}$ *layer*

$delta_{li}$ is the loss in i^{th} *node of* l^{th} *layer*

1.5.5 Termination of Neural Network

The termination of the nueral network is done when the neural network stop learning. This means that neural network training is termintated, when the accuracy of neural network on the test data stops to increase and weights and biases of all the neurons in different layers have reached a convergance.

1.5.6 Algorithms Complexity and execution time

It is evident that with the increase in the number of layers and number of nodes in each layer, the complexity of the algorithm and computational requirement will increase. This is because, with the increase in the number of layers or nodes, the size of the weights/bias/input/output/hidden arrays will increase. This will result in an increase in algorithm complexity and execution time. Execution time is also affected by the number of epochs used for training the neural network. As generally is observed in model training, a greater number of epochs and layers yield better accuracy for the model.

2. Serial Implementation

This section gives a description of the serial implementation of the neural network model. The section dwells into the program design, model details and explanation of implementation. We will also cover thes changes made in the serial implementation post Milestone-1. In the end, the section will also cover details about code verification.

2.1 Program Design

This program was implemented in C++ 11 version. University of Queensland's Getafix cluster infrastructure was used to train and test the model. This cluster has 512GB of RAM and 9 Nvidia Tesla P100 Graphical Processing Units.

2.2 Changes post Milestone -1

The following changes where implemented post Miletone-1:

1. The code was migrated from C language to C++
2. A 3-layer neural network code was finalised, as opposed to many codes with various layers in Milestone-1
3. User defined Input argument was added to decide the number of nodes in hidden layer
4. The number of epoch for the analysis was set to 1. This was because:
    a. Neural network trains by updating the weights and bias post each epoch, hence every epoch needs to run in serial. This meant that all the epoch have to run in serial and can not be paralised. This means that runtime of 10 epoches will be approximetly equal to 10 x (runtime of one epoch) regardless of the implementation technique.
    b. Hence, the parallel optimization was done at epoch level and runtimes for single epoch were analysed and reported for every implementation
5. The code was updated to contain more functions as compared to the version in Miletone-1. This was done for the following reasons:
    a. To remove redundancy in the code
    b. To understand the profiling of the code, which was not done in Milestone-1
    c. To implement the parallelization algorithm on these functions directly
6. Conversation of 2D array to 1D array: With user defined number of nodes in hidden layer, to improve the interpretability and acceptance of array by functions, 2D array wsa converted into 1D array. Another reason was, at many places same functions was used for hidden and output layer (these layers were variable). Hence, I decided to convert the 2D array to 1D after taking an inspiration from assignment 1

2.3 Model Details

The neural network in this project was built with the following specifications:

1. Each node in the neural network was a sigmoid function. This is because, with a small change in weights and bias, sigmoid function generates small change in output. This property of sigmoid function is essential because neural network training involves with change in weights and bias to reduce the model error. Sigmoid function makes sure that the change in weights and bias is proportionally reflected in the output.
2. Model loss is defined by the difference between the predicted output and the actual output.

3. The learning rate is set to 0.01 for all models. The learning rate is the tuning parameters used to determine the change in weights and bias with respect to the model loss.
4. The number of epochs was set to 1. Epoch is the number of passes of the entire training data to train the neural net model.
5. ***Input layer:*** The image of size 28 * 28 was flattened to an array of 784. Therefore, size of the input array was 784* 1. Corresponding to that, the number of nodes in input layer were 784.
6. ***Hidden Layer:*** There was one hidden layer with user defined number of nodes. The number of nodes could vary in the model. For illustration purpose, the number of nodes in the hidden layers was set to 15.
7. ***Output Layer:*** Number of nodes in the output layer was 10. Hence, the output was an array of size 10 * 1 with probabilistic likelihood of a given image being each number between 0 to 9. Since in this case, output is infact known to be a unique label between 0 to 9, the highest likelihood in the array was set to 1 and all the rest of the labels were set to zero.
8. ***Weights:*** Weights express the relative importance of input to that layer
    a. ***Weights between input and hidden layer:*** For each input, weight is needed for mapping between input and hidden layer. So, the size of array was size of input layer * size of hidden layer i.e. (784 * 15). For ease of understanding, we took transpose of this matrix.
    $$wh_{ji} = weight\ between\ j^{th}\ hidden\ node\ and\ i^{th}\ input\ node$$
    b. ***Weights between hidden and output layer:*** The size of this array was size of hidden layer * size of output layer i.e. (15 * 10). For ease of understanding, we took transpose of this matrix.
    $$wo_{kj} = weight\ between\ k^{th}\ output\ node\ and\ j^{th}\ hidden\ node$$
9. ***Bias***: For each node in hidden and output layer, there was a bias.
    a. ***Bias of hidden layer:*** A bias array of size of hidden layer i.e. (15 * 1)
    $$bh_j = bias\ of\ \ j^{th}\ hidden\ node$$
    b. ***Bias of output layer:*** A bias array of size of output layer i.e. (10 * 1)
    $$bo_j = bias\ of\ \ k^{th}\ output\ node$$
10. All the nodes were fully connected and nodes had sigmoid activation function. The neural network can be seen in Figure 6.

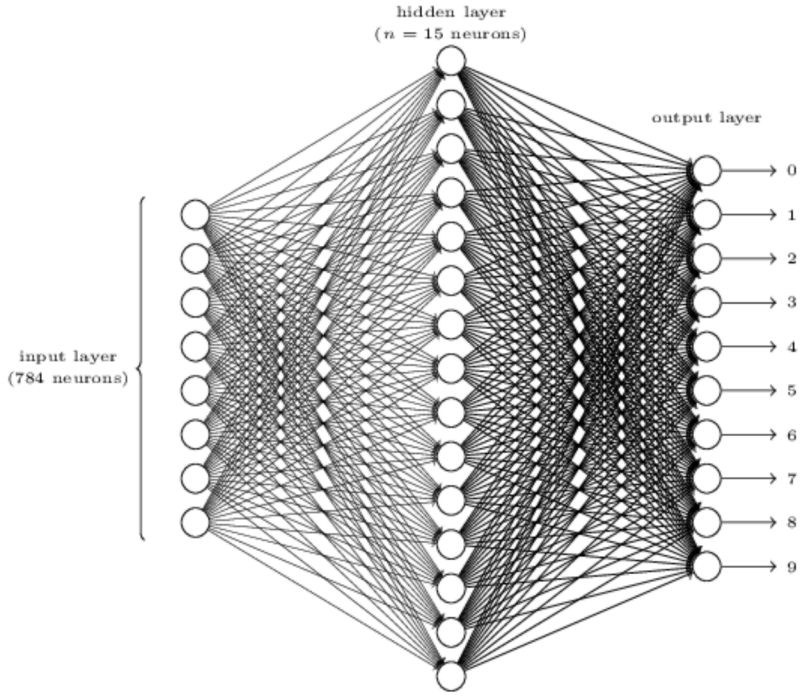


Figure 6: Neural Network

2.4 Algorithm Flow chart

1. Data was obtained from the official MNIST dataset: here
2. Weights and bias were initialised using sample normal distribution. The code for this in C++ was obtained from this location. This helps to prevent the neural network from being stuck in local minima during backpropagation.
3. Processes followed in each epoch:
    a. Iterate through the training dataset
    b. Generate mini-batch of size 10
    c. In each mini-batch:
        i. Obtain the forward propagation results
        ii. Obtain the backward propagation results
        iii. Sum the backward propagation from all the 10 datapoints
    d. Update the weights and bias post each mini-batch
4. After all the epochs were run (or even in between), model loss was calculated on test data.
5. The model, training per epoch and testing at relevant epochs can be seen in
6. Figure 7 ,
7. Figure 8 ,
8.
9. Figure 9
10. The algorithum process flow can be seen in Figure 10

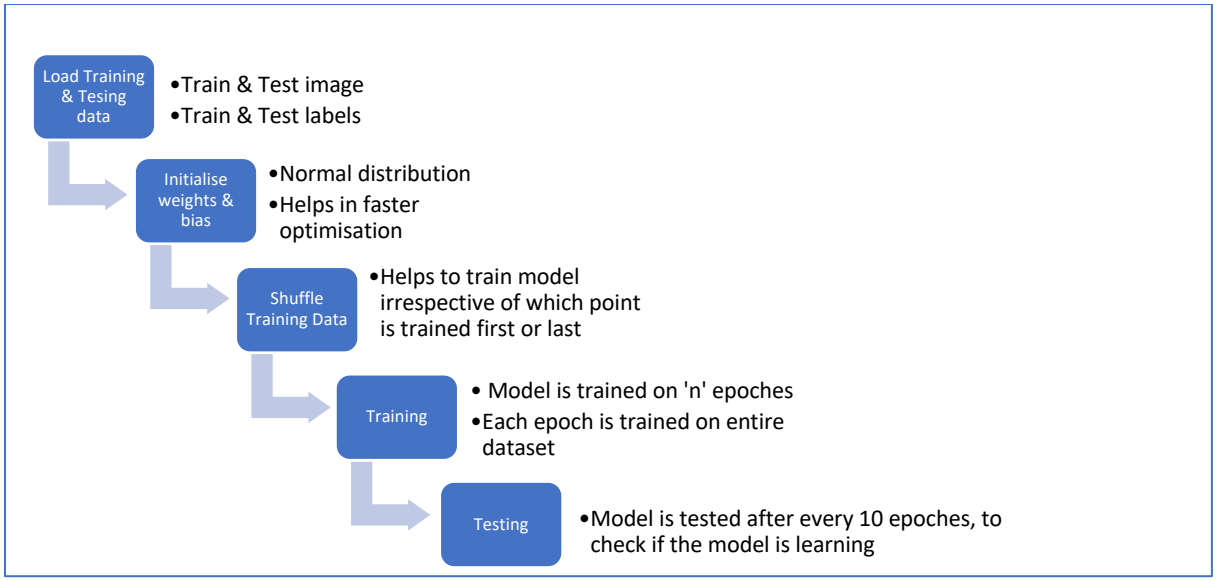


Figure 7: Process Flow

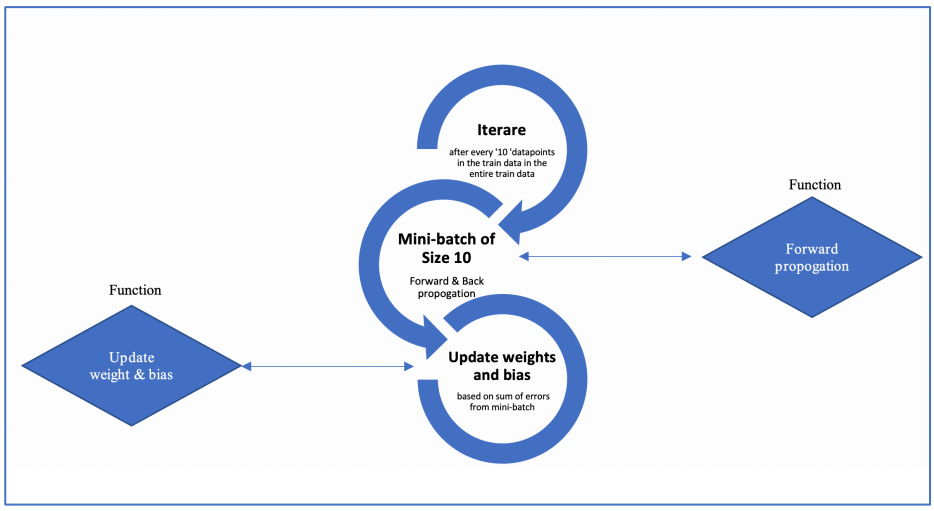


Figure 8: Training process at each epoch

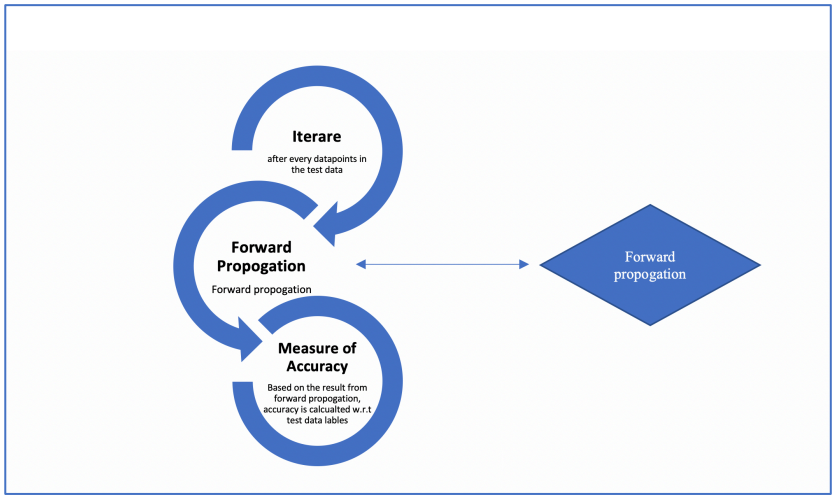


Figure 9: Testing per epoch

Algorithm 1 Neural Network

```
procedure NEURAL NETWORK(3 layer)
    Inputs: TestImage, TrainImage, TestLabel, TrainLabel                ▷ Input variables
    weight_1 ← normaldistribution                                       ▷ Initialization
    weight_2 ← normaldistribution
    bias_1 ← normaldistribution
    bias_2 ← normaldistribution
    hiddenlayer ← 0
    outputlayer ← 0
    epochs ← 1
    rows_train ← 60,000
    rows_test ← 10,000
    lr ← 0.01

    for epoch in epochs do
        for iteration ≤ row_train do iteration + 10                     ▷ Iterate over entire Train-data
            δweight_1 ← 0
            δweight_2 ← 0
            δbias_1 ← 0
            δbias_2 ← 0
            for index ≤ 10 do                                           ▷ Mini-batch
                Get TrainLabel_index, TrainImage_index
                inputdata ← TrainImage_index
                δbias_2 ← 0
                hiddenlayer ← forwardpropogation(weight_1, bias_1, inputdata)
                outputlayer ← forwardpropogation(weight_2, bias_2, hiddenlayer)
                δoutput+ ← backpropogation(outputlayer, TrainLabel)
                δweight_2+ ← backpropogation(δoutput, hiddenlayer)
                δhidden+ ← backpropogation(weight_2, outputlayer)
                δweight_1+ ← backpropogation(TrainLabel,δhidden)
            endfor
            weight_1 ← weight_1 − δweight_1 * lr/10                     ▷ Update the weights and bias
            weight_2 ← weight_2 − δweight_2 * lr/10
            bias_1 ← bias_1 − δbias_1 * lr/10
            bias_2 ← bias_2 − δbias_2 * lr/10
        endfor
        if epoch ÷ 10 == 0 then
            for iteration ≤ row_test do                                 ▷ Iterate over entire Test-data
                Get TestLabel_index, TestImage_index
                inputdata ← testImage_index
                hiddenlayer ← forwardpropogation(weight_1, bias_1, inputdata)
                outputlayer ← forwardpropogation(weight_2, bias_2, hiddenlayer)
                accuracy ← argmax(outputlayer) vs TestLabel
            endfor
```

Figure 10: Serial Algorithum

2.5 Profiling and optimsisation

Optimization:

The optimization of the serial code across numerous optimization levels can be seen in the Figure 11. It can be observed that as the level of optimization increases, the run time decreases.

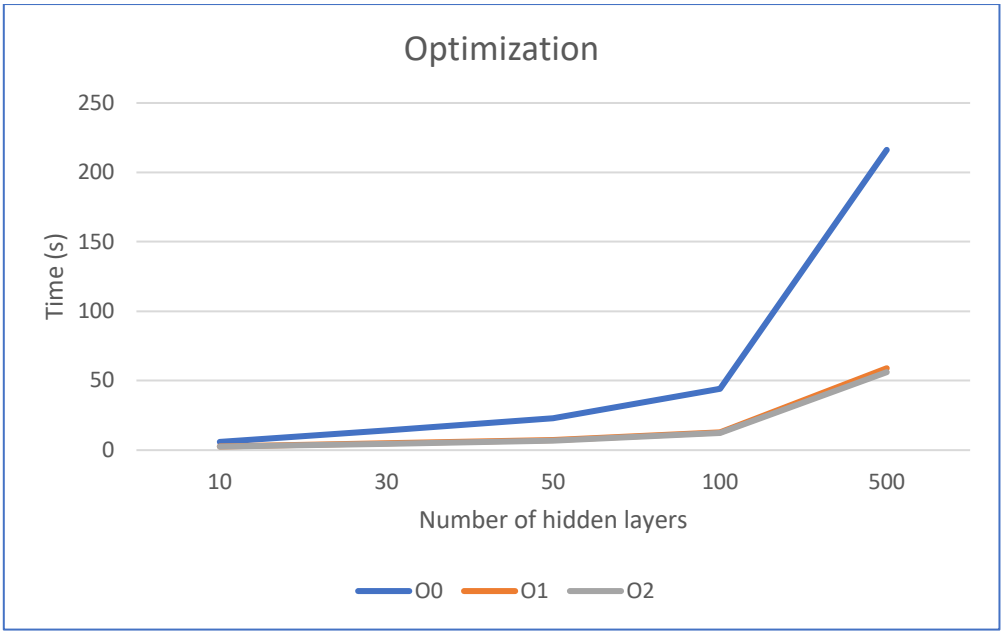


Figure 11: Optimization of Serial Code

Profiling:

The profiling information of the serial implementation can be seen in Figure 12 & Figure 13. It can be observed that, the main time consuming areas of the program are:

1. Forward pass
2. Main function execution
3. Updation of weights and bias
4. Sigmoind and Differenciation of Sigmoid also had many calls to it, but it hardly took much time.

During parallisation, these functions were the main focus to be improved upon.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 48.98     19.07    19.07   140000     0.14     0.14  forwardpass(int, int, double*, double*, double*, double*)
 43.73     36.09    17.03                             main
  4.01     37.65     1.56    12000     0.13     0.13  changewb(int, int, double*, double*, double*, double*)
  1.83     38.36     0.71    12000     0.06     0.06  reseteweights(int, int, double*)
  1.08     38.78     0.42        1   420.38   420.38  ReadMNIST(int, int, double**)
  0.33     38.91     0.13        1   130.12   130.12  ReadMNISTTest(int, int, double**)
  0.10     38.95     0.04  7700000     0.00     0.00  sigmoid(double)
  0.03     38.96     0.01  6600000     0.00     0.00  dSigmoid(double)
  0.00     38.96     0.00    79510     0.00     0.00  sample_normal_distribution()
  0.00     38.96     0.00    12000     0.00     0.00  resetbias(int, double*)
  0.00     38.96     0.00       16     0.00     0.00  ReverseInt(int)
  0.00     38.96     0.00        2     0.00     0.00  intialisebias(int, double*)
  0.00     38.96     0.00        2     0.00     0.00  intialiseweights(int, int, double*)
  0.00     38.96     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z10ReverseInti
  0.00     38.96     0.00        1     0.00     0.00  ReadMNISTLables(int, int, int**)
  0.00     38.96     0.00        1     0.00     0.00  ReadMNISTTestLables(int, int, int**)
  0.00     38.96     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00     38.96     0.00        1     0.00     0.00  shuffle(int*, unsigned long)

 %         the percentage of the total running time of the
time       program used by this function.
```

Figure 12: Profiling output

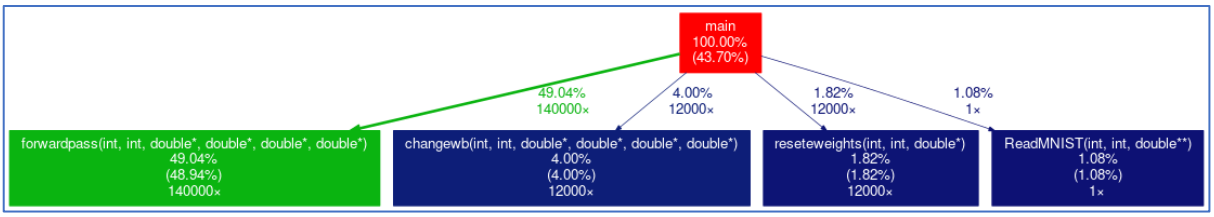


Figure 13: Profiling plot

2.6 Verification Procedure

To verify the results, the following test were done:

1. ***Verify the calculations with the python code*** *:* The inspiration for this code has been drawn from the neural network explanation of (Nielsen, 2019). Given a defined set of weights, bias, input, and output; the calculations of forward propagation, backward prorogation, and update weights should match in the current program and in the python code.
2. ***Verification of accuracy with iteration:*** The accuracy of the code with the python counterpart can be verified by initialising neural network of size 784*15*10 with weights and bias as 0.
    a. The results can be seen in below images.
    b. Both python and C++ code reach ~1130/10,000 accuracy in after a few iterations
    c. It should be noted that the accuracy of the neural network in python is more significant than that in C++. This is because the number of decimal places that C can handle is 14, while python can handle 28 decimal places. Due to this reason, the neural network in python is much more accurate as compared to C with the same number of epochs.
3. ***1st principle (sense) checks:***
    a. The number of epochs run are 100
    b. All the elements of the training data should be accessed in each iteration
    c. After each epoch, the training data should be shuffled
    d. In a dataset, both input and output datapoints should be shuffled together
    e. Weights or bias are not dramatically increasing from one epoch to another
    f. During testing, the correct and incorrect labels are correctly identified.

3. Parallel Implementation

This section details the parallesation of the serial neural network implementation from the previous chaper. This section also summaries the CUDA & Hybrid implementation strategies for the serial implementation above.

3.1 Parallelisation strategies for Neural Network

From the
Figure 13 it can be seen that the program spends 49% of the time in forward pass, while the other 4% of the time in updating the weights and bias after each mini-batch is run. And so, these two areas of calculation provides an opputunity for parallesation. These functions mainly include matrix multiplication, summation or substraction, hence on parallesing these, the runtime of the code will be reduced.

3.2 CUDA implementation

CUDA is a properitery platform by NVIDIA. The following steps where taken for CUDA implementation:

1. Identify the the sections of the code that need to be parallesied:
    a. Forward Propogation: This function processes the data coming in each layer, as defined in [
    b. 1.5.1 Forward Propagation]
    c. Update weights & bias: This function updates the weights and bias, post each minibatch, as defined in [1.5.4 Update of weights and bias]
2. Identify the variables that needs to be sent to the Device(from host) for calculation. Also for an optimal calcualation, identify how frequently does the variable needs to be sent to the device, depending on their frequency of updation.
    a. Forward propogation:
        i. Weights/ bias from all the layers:
            1. Sending to device: Once intialised, the weights/bias can be sent to the device. So, this will be once per program run
            2. Sending to host: The weights/bias don't need to be copied back to the host, as they are not being used in any calculation in the host
        ii. Input data
            1. Sending to device: Needs to be sent to the device for every mini-batch run
            2. Sending to host: The input data don't need to be copied back to the host, as they are not being used in any calculation in the host
        iii. Output from all the layers
            1. Sending to device: Since they are calculated only in the device, they don't need to be sent from host
            2. Sending to host: The output/hidden need to be copied back to the host for calculation of backpropogation after each mini-batch run
    b. Update weights & bias:
        i. Weights/bias from all the layers: Same as above

ii. weights/bias error from all the layers:
    1. Sending to device: Needs to be sent to the device for every iteration run post-minibatch run
    2. Sending to host: The weights eror don't need to be copied back to the host, as they are not being used in any calculation in the host

Variable Name	Host to Device Frequency	Device to Host Frequency
Weights/ bias from all the layers	Sent once post initilisation	-
Inputdata	Sent at each mini-batch during testing and training	-
Hidden layer output/ Output layer output	-	Sent at each mini-batch during testing and training
Backpropogation update	At each iteration	-

3. Based on the points in Part-2, all the relevant merrory allocation is done in the device (GPU)
4. Only the output from forward propogation needs to be copied back to the host post their calculation in forward propogation function
5. Launch the relevant global kernel on the GPU device:
    a. Forward propogation
    b. Updatation of weight and bias
6. Clear the memory. Memory cleared for all the variables post processing
7. Global kernel: Call the global CUDA kernel using the maximum threads and blocks possible.

3.3. CUDA Algorithum

According to the explanation in the above section, the algorithum for CUDA implementation can be seen in Figure 14. Please note, cuda device is clreaded as and when required by the variables.

Algorithm 1 Neural Network

```
procedure NEURAL NETWORK(3 layer)
    Inputs: TestImage, TrainImage, TestLabel, TrainLabel          ▷ Input variables
    weight_1 ← normaldistribution                                 ▷ Initialization
    weight_2 ← normaldistribution
    bias_1 ← normaldistribution
    bias_2 ← normaldistribution
    hiddenlayer ← 0
    outputlayer ← 0
    epochs ← 1
    rows_train ← 60, 000
    rows_test ← 10, 000
    lr ← 0.01

    weight_1 →GPU DEVICE                                          ▷ Send to GPU Device
    weight_2 →GPU DEVICE
    bias_1 →GPU DEVICE
    bias_2 →GPU DEVICE

    for epoch in epochs do
        for iteration ≤ row_train do iteration + 10               ▷ Iterate over entire Train-data
            δweight_1 ← 0
            δweight_2 ← 0
            δbias_1 ← 0
            δbias_2 ← 0
            for index ≤ 10 do                                     ▷ Mini-batch
                Get TrainLabel_index, TrainImage_index
                inputdata ← TrainImage_index
                inputdata →GPU DEVICE
                δbias_2 ← 0
    ▷ Process Forward Propagation in GPU
                hiddenlayer ← forwardpropogation(weight_1, bias_1, inputdata)
                outputlayer ← forwardpropogation(weight_2, bias_2, hiddenlayer)
    ▷ Send to HOST from GPU
                hiddenlayer →HOST
                outputlayer →HOST
                δoutput+ ← backpropogation(outputlayer, TrainLabel)
                δweight_2+ ← backpropogation(δoutput, hiddenlayer)
                δhidden+ ← backpropogation(weight_2, outputlayer)
                δweight_1+ ← backpropogation(TrainLabel, δhidden)
            endfor
    ▷ Send to GPU Device
            δweight_1 →GPU DEVICE
            δweight_2 →GPU DEVICE
            δhiddenlayer →GPU DEVICE
            δoutputlayer →GPU DEVICE                              ▷ Process Update weight  Bias in GPU
            weight_1 ← weight_1 − δweight_1 * lr/10               ▷ Update the weights and bias
            weight_2 ← weight_2 − δweight_2 * lr/10
            bias_1 ← bias_1 − δbias_1 * lr/10
            bias_2 ← bias_2 − δbias_2 * lr/10
        endfor
        if epoch ÷ 10 == 0  then
            for iteration ≤ row_test do                           ▷ Iterate over entire Test-data
                Get TestLabel_index, TestImage_index
                inputdata ← testImage_index
                inputdata →GPU DEVICE
    ▷ Process Forward Propagation in GPU
                hiddenlayer ← forwardpropogation(weight_1, bias_1, inputdata)
                outputlayer ← forwardpropogation(weight_2, bias_2, hiddenlayer)
    ▷ Send to HOST from GPU
                hiddenlayer →HOST
                outputlayer →HOST
                accuracy ← argmax(outputlayer) vs TestLabel
            endfor
```


Figure 144: Cuda Algorithum

3.4. Hybrid Algorithum

To perform the hybrid implementation, analysis was done to ascentain which part of the code can be more parallised. The aim was to parallise a part of the 'main' function, since it is the second most time consuming section after the 'Forward Propogation'. Every epoch in a neural network is always serial, also, every iteration in an epoch is serial. However, mini-batch can possibly be paralised using OpenMP. This means that the various training set in the Mini-batch can be paralysed. However it should be noted that the backpropogation calculations needs to be marked as "critical" as they need to happen exactly 10 times. Another important thing to note here is,

"#pragma omp simd" was used instead of "#pragma omp parallel for". This is because "#pragma omp simd" threads on a single CPU, while "#pragma omp parallel for" threads on multicore processor. Since the GPU code is already threading on the multicore processor, I needed the mini-batch to process only on one thread.

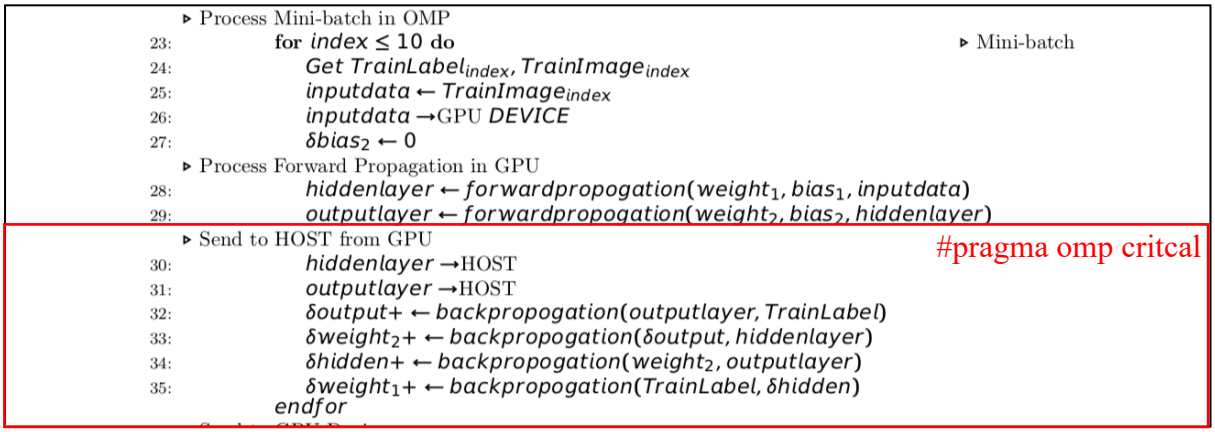


Figure 15: Hybrid Implementation

3.2.4 Verification Procedure

Similar verifiction process was done as specified in [

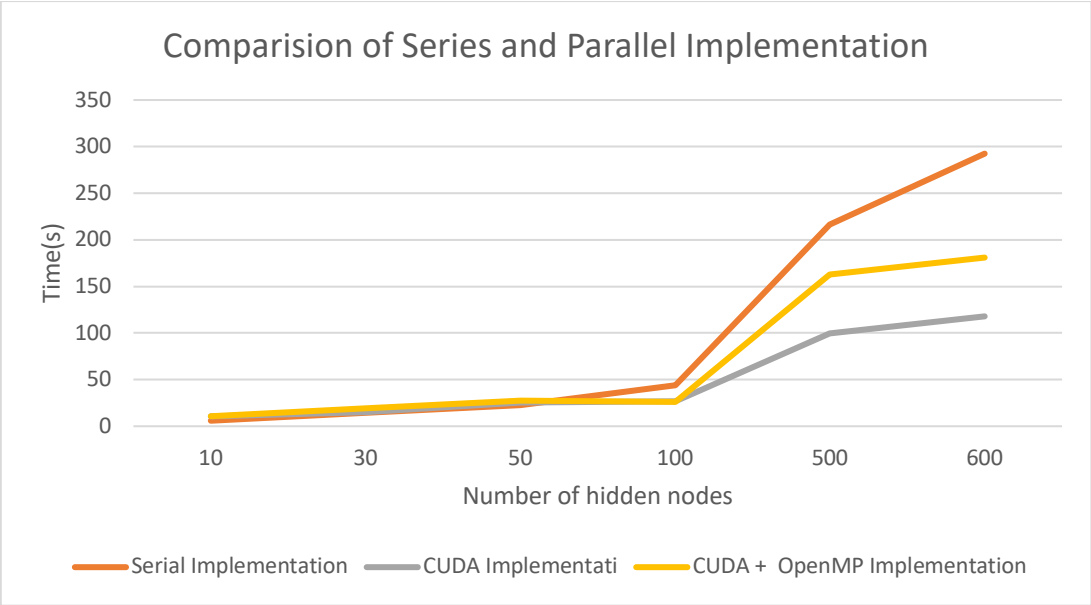
2.6 Verification Procedure]

4. Result Comparison

4.1. Serial vs Parallel

The following observations can be made:

1. For the hidden node size = 10 till 50, the serial implementation is less time consuming as compared to CUDA or hybrid implementation
2. As the size of the matrix increases beyond 50, the time taken by CUDA or hybrid implementation decrease considerably
3. This is happening because CUDA takes time to transfer the data from host to device. For small matrix size, this is not very efficient and consumes additional time over serial implementation.



4.2 Amdahl's Law & Gustafson's law

Nodes	CPU	GPU	Hidden Node = 100	Hidden Node = 500
1	1	1	28.32	99.45
2	1	1	28.03	98.98
1	1	2	25.65	96.74
2	1	2	25.79	96.82
2	1	3	26.75	94.94
1	1	3	26.6	94.74

1. Does Parallelizing the code on N processors reduces run time by a factor of N?: As can be seen from the above table, the runtime does not reduce by a factor of N; however, it does reduce by a significant amount as the number of CPU's/Nodes increases.

2. ***Amdahl's law***: If a fraction p of a program can be parallelised, then with a large number of processors the unparallelizable fraction (1 − p) dominates. The maximum theoretical speedup is 1/(1 − p):
    a. In the current code, the parallelism is done only on a few kernels and the rest of the code is run as is. This suggests that hypothetically even if on paralleling the kernels takes 0 seconds to run, the runtime of the code will depend on the process that was not parallelised.
    b. As seen in the table above, no matter how many nodes and CPU are being deployed, the timing of the code does not decrease by a lot beyond a certain computation resource. This suggests that there is only a part of code that is parallelised which results in reduction of the run time, but one cannot reduce the timing by a lot post a certain point.
3. ***Gustafson's law***: This law says that increase of problem size for large machines can retain scalability with respect to the number of processors. As seen in the above table, the processing time for 500 nodes decreases more as compared to that in 100 nodes (relatively).

4.3. Conclusion

Neural network are generally trained thousands of times over huge training datasets. Even a slight decrease in time of one epoch, helps to reduce the runtime by a lot. In this project, a serial code was parallelised using CUDA and OpenMP capabilities. The parallel code could reduce the runtime by up to 50%. When compared with default optimizer, parallel code came out to be slower than the default optimizer in the compiler. However, it is always better to implement an optimization algorithum as the defult optimizer may remove or rearrage part of your code which might result in a huge change in the expected result.