CUDA implementation of a 3-layer Neural Network

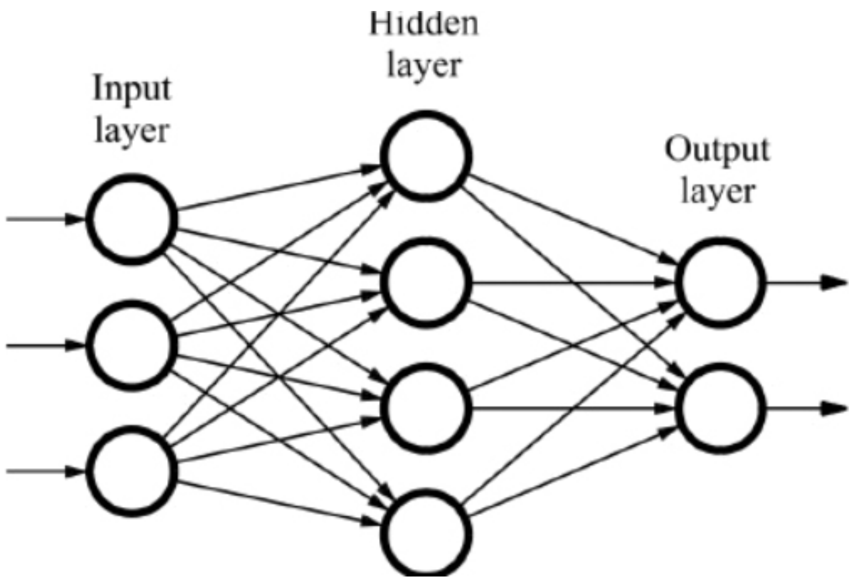
Trained using back propagation algorithm on MNIST

Sreepathy Jayanand
CED17I038

Introduction

Neural network:

A Neural network is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called *edges*. Neurons and edges typically have a *weight* that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.



CUDA :

Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface model created by Nvidia. The CUDA platform is designed to work with programming languages such as C, C++, Fortran.

System Specifications CPU (Google Colab):

- Intel(R) Xeon(R) CPU
- Clock speed - 2.20GHz
- Physical Cores - 1
- Threads / Core - 2
- Logical Cores - 2
- L3 cache size - 56.32 MB
- Main memory - 12.43 GB (DDR4)

System Specifications GPU (Google Colab):

- Nvidia K80
- Memory - 16280 MB (GDDR5)
- Clock speed - 1.59GHz
- Cores - 4992 cuda cores

Softwares:

- C++
- Cuda version 9.2
- Bash scripts
- Python3 extension for calculation of overall execution time

I/O Specification:

The inputs for training the network were taken from the standard MNIST handwritten dataset which contains about 60k images, each of 28 x 28 pixels in size and 10 digits showing the confidence of prediction of each output.

Working principle of the algorithm:

Each input image is taken and passed through the neural network “epoch” number of times and every time the weights are updated using the back - propagation, based on the error generated by comparing result with actual output. The error is back propagated and the weights updated proportional to the error gradient with respect to the weight.

Initially the image which is a 28 x 28 matrix is converted to a 784 x 1 vector.
Forward and backward propagation is performed using vector - matrix multiplication (since we are training a single input at a time, we don’t need more than 1-dimensional vector to represent the image in the batch (batch size is 1) and a 2-dimensional array to store the weights matrix).

Project specifications:

No of inputs: 60,000
No of nodes in the input layer (n1) : 784 (Each image is 28 x 28 pixels)
No of nodes in the hidden layers (n2) : 128
No of nodes in the output layer (n3) : 10 (For each label)
Epsilon: 0.001
Momentum: 0.9 (To avoid getting stuck at a local minima)
Epochs: 512
Activation function: Sigmoid
Error: Square error
After the network is updated, the weights are updated in the saved weights file, and for testing this file is loaded.

Functions used:

- Input(), write_output() - For reading input images and storing saved weights as output
- Square_error() - To calculate the error
- Init() - To initialise the weights array and other parameters
- Perceptron() - To perform the forward propagation using vector - vector multiplication
- Back_propagation() - To perform the weight vector updations using the error terms generated from the layers in front of it.

- Learning_process() - Combination of perception() and back_propagation(), called once each per epoch

Inferences from profiling:

The table given below gives us an understanding of how much time is spent on each of the functions.

Function Name	Percentage of time taken
back_propagation()	73.08
perceptron()	26.41
init_array()	0.42
square_error()	0.10
learning_process	0.10

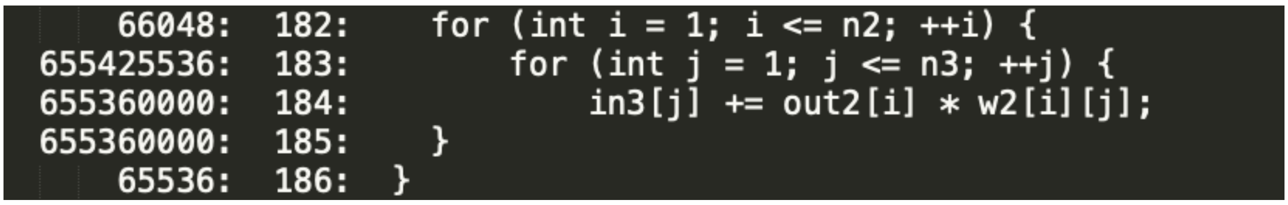
The table given below gives us an understanding of different branching statistics.

Category	Statistics
Branch rate	0.0590
Branch mis-prediction rate	3.84E-05
Branch mis-prediction ratio	0.0007
Instructions per branch	16.93

From this table we can infer that the back_propagation() is the bottleneck and and optimization of that function will help us more in the improvement of the program compared to the other functions

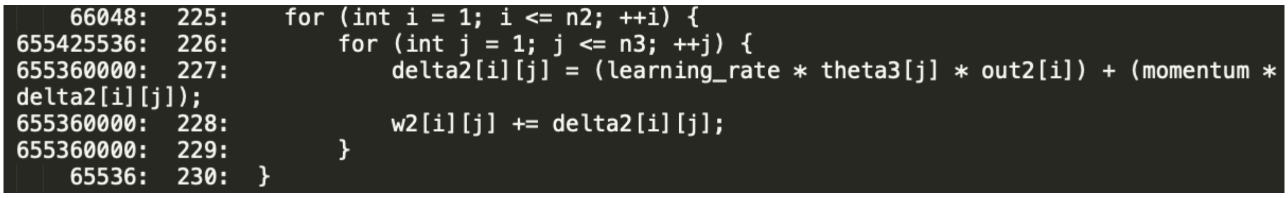
The following 2 images give us more of an idea of the number of iterations in each of perceptron() function's bottleneck part and the back_propagation() function's bottleneck part.

Perceptron():



```
    66048:  182:    for (int i = 1; i <= n2; ++i) {
655425536:  183:        for (int j = 1; j <= n3; ++j) {
655360000:  184:            in3[j] += out2[i] * w2[i][j];
655360000:  185:    }
    65536:  186:  }
```

Back_propagation():



```
    66048:  225:    for (int i = 1; i <= n2; ++i) {
655425536:  226:        for (int j = 1; j <= n3; ++j) {
655360000:  227:            delta2[i][j] = (learning_rate * theta3[j] * out2[i]) + (momentum *
delta2[i][j]);
655360000:  228:            w2[i][j] += delta2[i][j];
655360000:  229:        }
    65536:  230:  }
```

Here we can see that both the bottlenecks in both the functions execute for the same amount of iterations but the intensity of the calculations within the back_propagation() function is bigger than the perceptron() function. And according to the profiling table mentioned earlier the intensity of the number of calculations of the back_propagation() function is 2.8 times that of the perceptron() function.

CODE BALANCE:

This gives an idea about the ratio of the actual calculations performed to the amount of data transfer the program performs.

Function Name	No of loads	No of stores	No of FP operations
init_array()	6	2	6
perceptron()	23	8	5
back_propagation()	2	3	11
learning_process()	0	7	0
total	31	20	22

Code Balance (CB) = (No of loads + No of stores) / No of floating point operations
= (31 + 20) / 22 = 2.31

Implementation details:

The training for the neural network is usually done serially with the errors being updated one after the other and the images taken one after the other, rather than paralleley, all at once which doesn't produce a well trained network.
This creates an inherent anti - parallel nature in the code.
Since in the context of high performance computing parallel codes are preferred, the following adjustments have been made to ensure parallelism in the code:

- The no of nodes in the hidden layer has been increased to 10000
- Epochs have been restricted to only 10
- And have only trained 5 set of images

These adjustments will reduce the accuracy significantly but will give us a better understanding of the effect of the number of threads vs the execution time.

Regions Parallelised:

- The initialisation of the weights of the network.
- Forward propagation from the 1st layer to the 2nd layer.
- Calculation of the activation function of the 2nd layer.
- Forward propagation from the 2nd layer to the 3rd layer.
- Calculation of the activation function of the 3rd layer.
- Calculation of the error terms.
- Back propagation from the 3rd layer to the 2nd.
- Back propagation from the 2nd layer to the 1st.

Strategy:

The overall view for the strategy is each thread takes care of a particular number of nodes in each layer and calculates the values necessary for it.

To elaborate,

Let's consider the **init_array()** function which initialises the weights between the input layer and the hidden layer. There are n1 nodes in the input layer and n2 layers in the hidden layer.

Let the block size = bs which is also the number of threads which run the program.

Thread 0 takes care of initialising the weights coming to nodes 1, bs + 1, 2 * bs + 1, 3 * bs + 1 and so on, in the hidden layer.

Thread 1 takes care of initialising the weights coming to nodes 2, bs + 2, 2 * bs + 2, 3 * bs + 2 and so on, in the hidden layer.

In general thread t takes care if initialising the weights coming to nodes t + 1, bs + t + 1, 2 * bs + t + 1 and so on, in the hidden layer.

Synchronisation of threads is done.

For the initialisation of weights between the hidden layer and the output layer,

Thread 0 takes care of weights leaving from nodes 1, bs + 1, 2 * bs + 1, 3 * bs + 1 and so on, in the hidden layer.

Thread 1 takes care of weights leaving from nodes 2, bs + 2, 2 * bs + 2, 3 * bs + 2 and so on, in the hidden layer.

In general thread t takes care of weights leaving from nodes t + 1, bs + t + 1, 2 * bs + t + 1 and so on, in the hidden layer.

Synchronisation of threads is done.

Coming to the **perceptron()** function,

The first part is the initialisation of in2[], in3[] which is the input to layers 2 and 3 and have to be initialised to 0. For in3[] since it is very much smaller compared in2[] in our case, initialisation of in3[] is taken care by thread 0 always while the rest of the threads wait (implemented using **__syncthreads()).** For the initialisations of in2[] which is a very big array, each thread takes care of a particular set of indices,

To elaborate,

Thread 0 takes care of initialising in2[1], in2[bs + 1], in2[2 * bs + 1], in2[3 * bs + 1] and so on.
Thread 1 takes care of initialising in2[2], in2[bs + 2], in2[2 * bs + 2], in2[3 * bs + 2] and so on.
In general thread t takes care of initialising in2[t + 1], in2[bs + t + 1], in2[2 * bs + t + 1], in2[3 * bs + t + 1] and so on

Synchronisation of threads is done.

Now coming to the calculation of output for hidden layer computed by multiplying the weight matrix w1[][] with the output from layer 1 out1[].
Here each thread takes care of producing the output for a particular set of nodes in the hidden layer.

To elaborate,
Thread 0 takes care of output of nodes 1, bs + 1, 2 * bs + 1, 3 * bs + 1 and so on, in the hidden layer.
Thread 0 takes care of output of nodes 2, bs + 2, 2 * bs + 2, 3 * bs + 2 and so on, in the hidden layer.
In general thread t takes care of output of nodes t + 1, bs + t + 1, 2 * bs + t + 1 and so on, in the hidden layer.

Synchronisation of threads is done.

After all this the final output from the hidden layer is obtained by calculating the sigmoid function for each node. This is also done in the same way as stated above.

Synchronisation of threads is done.

Coming to the calculation of output for the final layer, since the number of elements in the hidden layer is more and it is optimal to parallelise based on the nodes in the hidden layer, and partial output for the output nodes will be updated to the output layer by all the threads simultaneously there is a chance of a race condition occurring. To prevent this a reduction technique is used.
To elaborate,

A partial matrix multiplication array is introduced (part_mat_mul[bs][n3]). What this does is saves the partial output created by thread t in part_mat_mul[t][n3].
After all the threads finish calculation of the partial answers, thread 0 combines all the partial answers as follows,
∑part_mat_mul[t][n3] for all t, and stores it in in3[n3]. This will give us the final in3[] array.

Synchronisation of threads is done.
After this the sigmoid of the this layer is calculated by thread 0 as the number of nodes here is small.

Synchronisation of threads is done.

Coming to the **back_propagation() function,**
The first part is the calculation of theta3[], which the error * sigmoid derivative obtained from the output layer. This number of nodes in this layer is small and hence thread 0 takes care of that calculation.
The next part is the calculation of theta2[], which is, the output of output layer * sigmoid derivative from the hidden layer. Each thread takes care of the calculation of theta for a set of particular nodes in the hidden layer.
To elaborate,
Thread 0 takes care of calculation of theta of nodes 1, bs + 1, 2 * bs + 1, 3 * bs + 1 and so on, in the hidden layer.
Thread 0 takes care of calculation of theta of nodes 2, bs + 2, 2 * bs + 2, 3 * bs + 2 and so on, in the hidden layer.
In general thread t takes care of calculation of theta of nodes of nodes t + 1, bs + t + 1, 2 * bs + t + 1 and so on, in the hidden layer.

Synchronisation of threads is done.

Coming to the updation of weights w2[i][j], using delta2[i][j]
Each thread t takes care of the calculation of a particular set of delta[2][] elements.
To elaborate,
Thread 0 takes care of calculation of delta2[1][], delta2[bs + 1][], delta2[2 * bs + 1][] and so on..

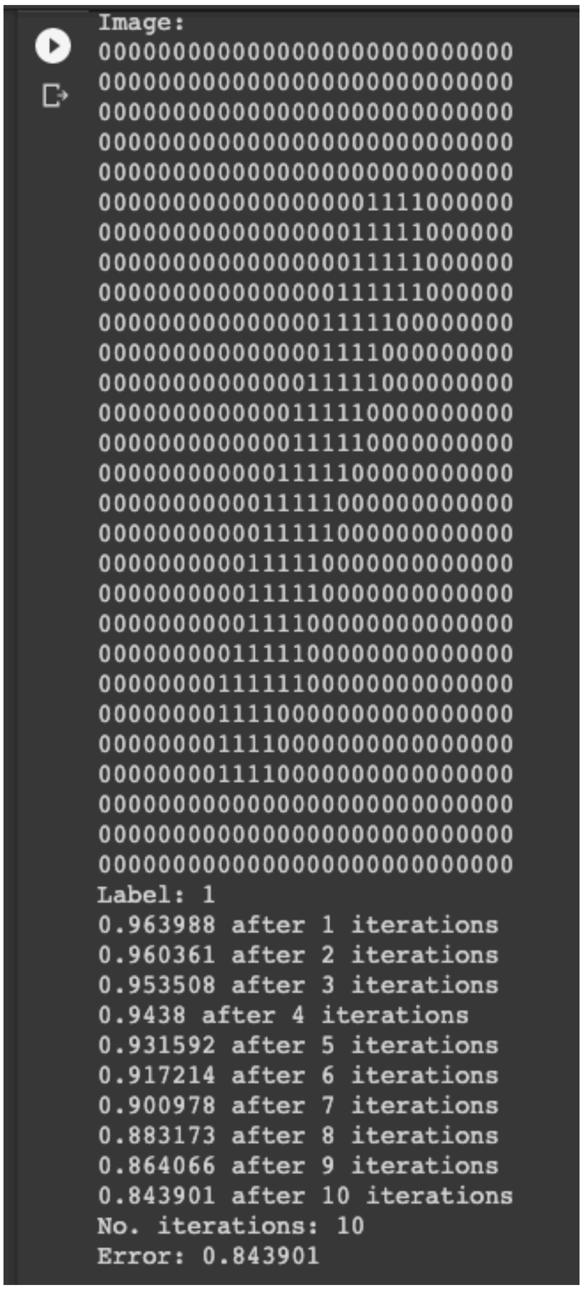
Thread 1 takes care of calculation of delta2[2][], delta2[bs + 2][], delta2[2 * bs + 2][] and so on..

In general thread t takes care of calculation of delta2[t + 1][], delta2[bs + t + 1][], delta2[2 * bs + t + 1][] and so on..

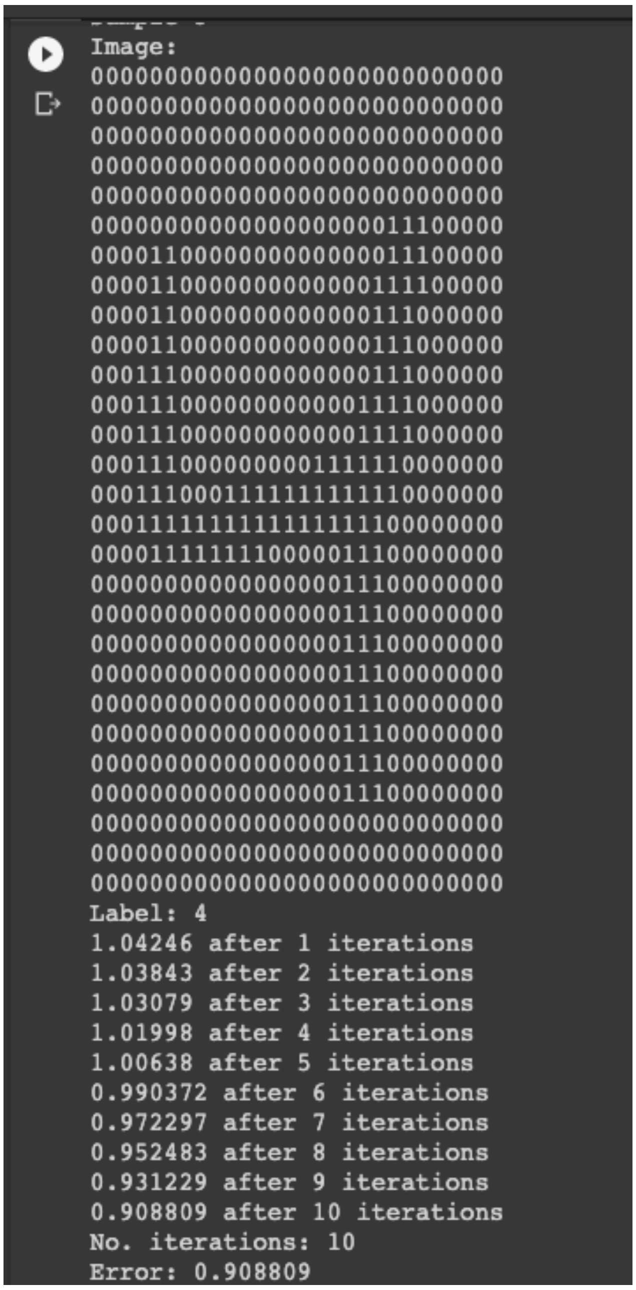
Synchronisation of threads is done.

A similar approach is followed for the updation of weights w1[i][j], using delta1[i][j],
This completes the back propagation part as well.

Outputs:

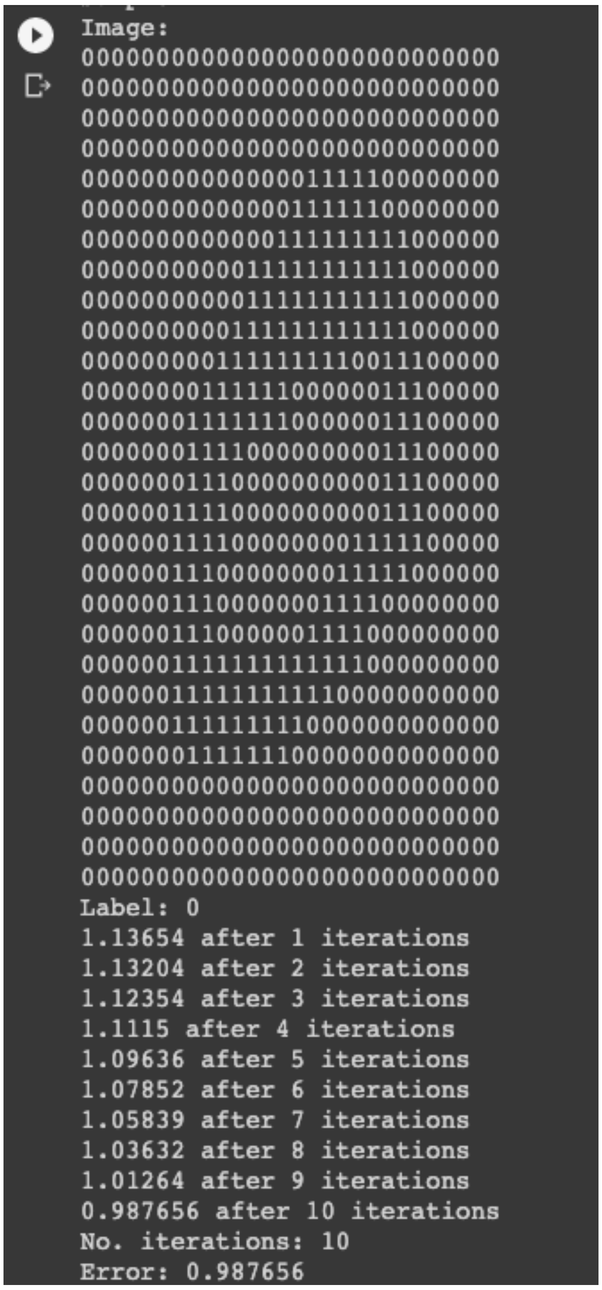


```
Image:
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000001111000000
0000000000000000011111000000
0000000000000000011111000000
0000000000000000111111000000
0000000000000001111100000000
0000000000000001111000000000
0000000000000011111000000000
0000000000000111110000000000
0000000000000111110000000000
0000000000001111100000000000
0000000000011111000000000000
0000000000011111000000000000
0000000000111110000000000000
0000000000111110000000000000
0000000000111100000000000000
0000000001111100000000000000
0000000011111100000000000000
0000000011110000000000000000
0000000011110000000000000000
0000000011110000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
Label: 1
0.963988 after 1 iterations
0.960361 after 2 iterations
0.953508 after 3 iterations
0.9438 after 4 iterations
0.931592 after 5 iterations
0.917214 after 6 iterations
0.900978 after 7 iterations
0.883173 after 8 iterations
0.864066 after 9 iterations
0.843901 after 10 iterations
No. iterations: 10
Error: 0.843901
```

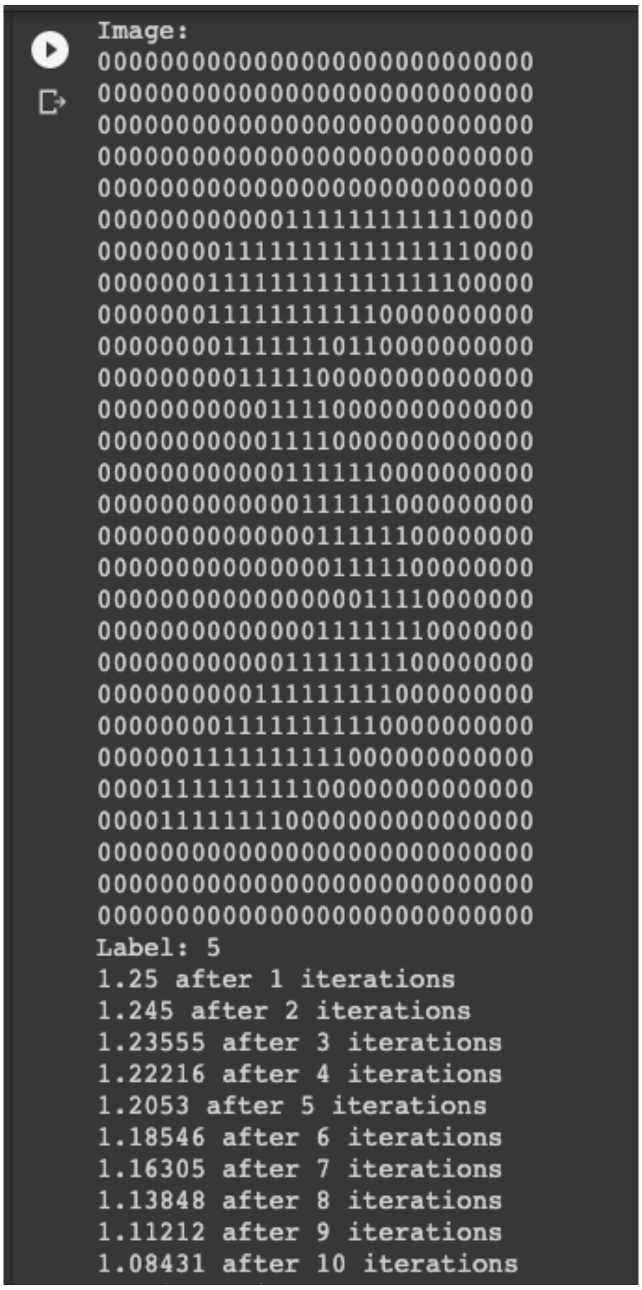


```
Image:
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000011100000
0000110000000000000011100000
0000110000000000000111100000
0000110000000000000111000000
0000110000000000000111000000
0001110000000000000111000000
0001110000000000001111000000
0001110000000000001111000000
0001110000000001111110000000
0001110001111111111110000000
0001111111111111111100000000
0000111111100000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000111000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
Label: 4
1.04246 after 1 iterations
1.03843 after 2 iterations
1.03079 after 3 iterations
1.01998 after 4 iterations
1.00638 after 5 iterations
0.990372 after 6 iterations
0.972297 after 7 iterations
0.952483 after 8 iterations
0.931229 after 9 iterations
0.908809 after 10 iterations
No. iterations: 10
Error: 0.908809
```

Here we can see the steady decrease in the error as we travel down the gradient.



```
Image:
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000001111100000000
0000000000000011111100000000
0000000000000111111111000000
0000000000011111111111000000
0000000000011111111111000000
0000000000111111111111000000
0000000001111111110011100000
0000000011111100000011100000
0000000111111100000011100000
0000000111100000000011100000
0000000111000000000011100000
0000001111000000000011100000
0000001111000000001111100000
0000001110000000011111000000
0000001110000000111100000000
0000001110000001111000000000
0000001111111111111000000000
0000001111111111110000000000
0000001111111111000000000000
0000000111111100000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
Label: 0
1.13654 after 1 iterations
1.13204 after 2 iterations
1.12354 after 3 iterations
1.1115 after 4 iterations
1.09636 after 5 iterations
1.07852 after 6 iterations
1.05839 after 7 iterations
1.03632 after 8 iterations
1.01264 after 9 iterations
0.987656 after 10 iterations
No. iterations: 10
Error: 0.987656
```



```
Image:
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000001111111111110000
0000000011111111111111110000
0000000111111111111111100000
0000000111111111110000000000
0000000011111110110000000000
0000000001111100000000000000
0000000000011110000000000000
0000000000011110000000000000
0000000000001111110000000000
0000000000000111111000000000
0000000000000011111100000000
0000000000000001111100000000
0000000000000000011110000000
0000000000000011111110000000
0000000000001111111100000000
0000000000111111111000000000
0000000011111111110000000000
0000001111111111100000000000
0000111111111110000000000000
0000111111110000000000000000
0000000000000000000000000000
0000000000000000000000000000
0000000000000000000000000000
Label: 5
1.25 after 1 iterations
1.245 after 2 iterations
1.23555 after 3 iterations
1.22216 after 4 iterations
1.2053 after 5 iterations
1.18546 after 6 iterations
1.16305 after 7 iterations
1.13848 after 8 iterations
1.11212 after 9 iterations
1.08431 after 10 iterations
```

Again, here we can see the steady decrease in the error as we travel down the gradient.

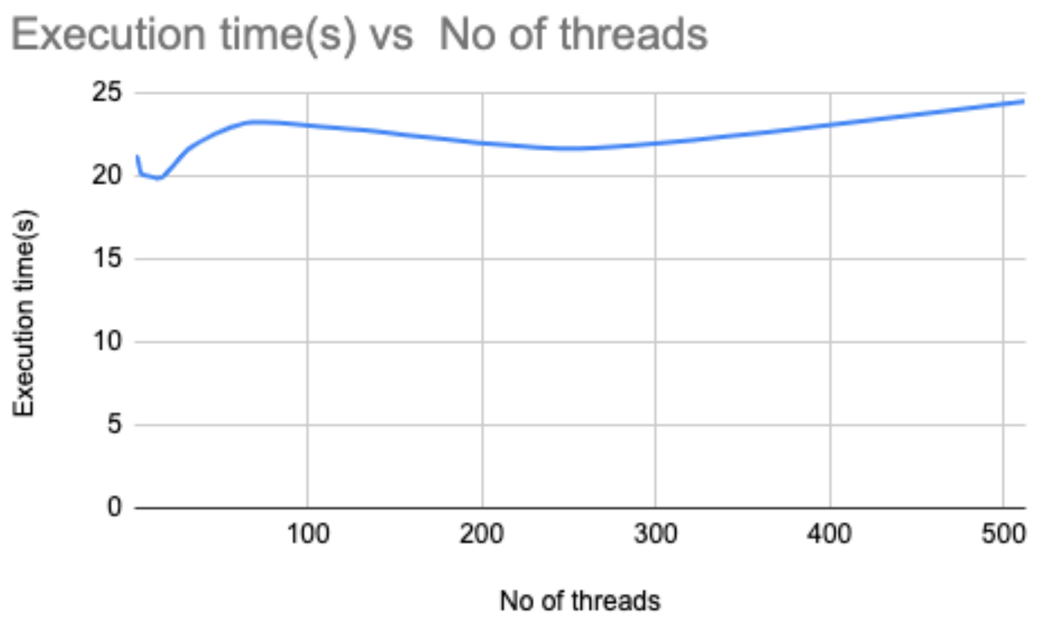
Table (Number of threads vs. Execution time) :

No of threads	Execution time (s)
1	21.21
2	21.11
4	20.20
8	20.04
16	19.93
32	21.70
64	23.23
128	22.83
256	21.69
512	24.55

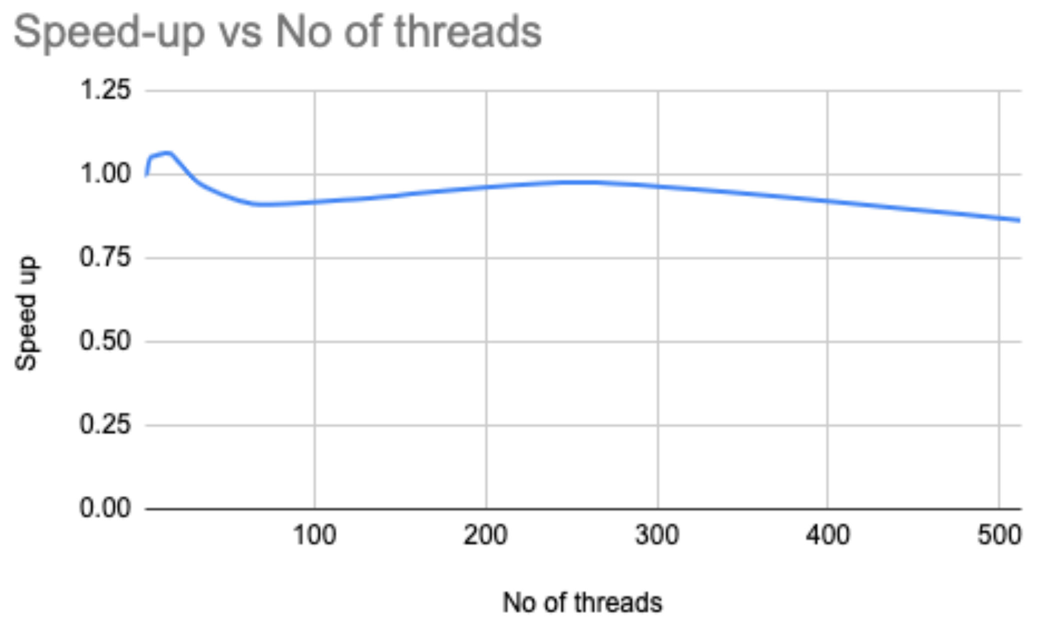
Analysis of the performance for different number of threads:

We can see that initially the performance increases as the number of threads increases and it gradually looses its performance. This represents a very positively skewed inverted bell curve. This is mainly due to the **__syncthreads()** functions being called multiple times within the program which ensures the correctness of the program. For the program to give the correct output it is necessary that the calculations be done layer by layer, serially. So we are only able to parallelise the calculations within each layer thus making the code mostly serial in nature.

Plots:



Execution time(s) vs No of threads
Execution time(s)
25
20
15
10
5
0
100
200
300
400
500
No of threads



Speed-up vs No of threads
Speed up
1.25
1.00
0.75
0.50
0.25
0.00
100
200
300
400
500
No of threads

Performance analysis:

According to Amdahl's law,

Speed up (S.U)= 1/((1 - p) + p / N), where 'p' is the parellelization fraction.

Hence parellelization fraction = (N - (N / S.U)) / (N - 1)

According to the obtained output the most efficient thread count = 16

Speed up = 21.21 / 19.93 = 1.064

Correspondingly the parallelisation fraction = (16 - (16 / 1.064)) / (16 - 1) = 6.41%

Conclusion:

This implementation of a 3 layer neural network executed in a NVIDIA K40 GPU, with the improvements after parallelising the code can achieve a speedup of 1.064 times the original speed with a parallelisation fraction of 6.4%. The main reason for not being able to improve the speedup even with higher threads is because of an inherent serial nature within the code which has to be taken care of to ensure the correctness of the neural network.