

PARALLELIZING DEEP NEURAL NETWORKS USING MPI AND GPU COMPUTING

**A Project Report submitted in partial fulfillment of the requirements for the
award of the degree of**

**BACHELOR OF
TECHNOLOGY**

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

VAMSI KRISHNA VAJJA 121910312014

AIDHITHA 121910312029

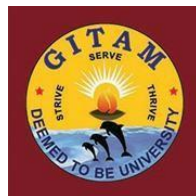
YASHWANTH 121910312041

BINDU MARRI 121910312042

Under the esteemed guidance of

Mrs.Saraswathi Pedada

ASSISTANT PROFESSOR,GST



**DEPARTMENT OF COMPUTER SCIENCE
& ENGINEERING**

GITAM

(Deemed to be university)

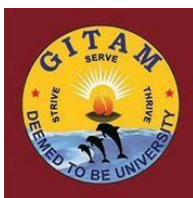
VISAKHAPATNAM

OCTOBER 2022

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING GITAM SCHOOL OF TECHNOLOGY**

GITAM

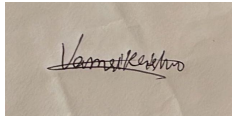
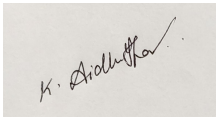
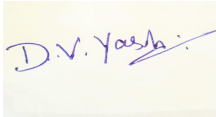
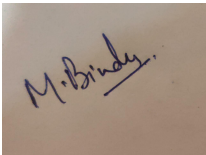
(Deemed to be University)



DECLARATION

We, hereby declare that the project report entitled “**PARALLELIZING DEEP NEURAL NETWORKS USING MPI AND GPU COMPUTING**” is an original work done in the Department of Computer Science and Engineering, GITAM School of Technology, GITAM (Deemed to be University) submitted in partial fulfillment of the requirements for the award of the degree of B.Tech. in Computer Science and Engineering. The work has not been submitted to any other college or University for the award of any degree or diploma.

Date:19-10-22

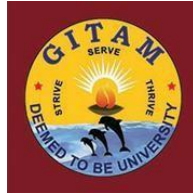
| Registration No(s). | Name(s) | Signature(s) |
|---------------------|---------------------|---|
| 121910312014 | VAMSI KRISHNA VAJJA |  |
| 121910312029 | AIDHITHA K |  |
| 121910312041 | YASHWANTH |  |
| 121910312042 | BINDU MARRI |  |

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM

(Deemed to be University)



CERTIFICATE

This is to certify that the project report entitled “**PARALLELIZING DEEP NEURAL NETWORKS USING MPI AND GPU COMPUTING**” is a bona fide record of work carried out by **VAMSI KRISHNA VAJJA(121910312014), AIDHITHA KONATHALA(121910312029), YASHWANTH(121910312041),** **BINDU MARRI(121910312042)** students submitted in partial fulfillment of requirements for the award of degree of Bachelors of Technology in Computer Science and Engineering.

Project Guide

Head of the Department

Mrs.Saraswathi Pedada

Dr. R.SIREESHA

ASSISTANT PROFESSOR

PROFESSOR

CSE,GST

CSE,GST

GITAM

GITAM

ACKNOWLEDGMENT

We would like to thank our project guide **Mrs.Saraswathi Pedada**, Assistant Professor, Department of CSE for her stimulating guidance and profuse assistance. We shall always cherish our association for her guidance, encouragement and valuable suggestions throughout the progress of this work. We consider it a great privilege to work under her guidance and constant support.

We also express our gratitude to the project reviewers **Mrs.K.N.SOUJANYA**, Assistant Professor and **Mr.P.Sanyasi Naidu**, Associate Professor, Department of CSE, GITAM (Deemed to be University) for their valuable suggestions and guidance in the completion of our project.

We consider it as a privilege to express our deepest gratitude to **Dr.R.SIREESHA**, Head of the Department, Computer Science And Engineering for her valuable suggestions and constant motivation that greatly helped us to successfully complete this project.

Our sincere thanks to **Prof.Ch. VIJAY SHEKAR**, Principal, GITAM Institute of Technology, GITAM (Deemed to be University) for inspiring us to learn new technologies and tools.

Finally, we deem it a great pleasure to thank one and all that helped us directly and indirectly throughout this project.

| | |
|------------------------|---------------------|
| VAMSI KRISHNA V | 121910312014 |
| AIDHITHA K | 121910312029 |
| YASHWANTH | 121910312041 |
| BINDU M | 121910312042 |

TABLE OF CONTENTS

| | | |
|----|---------------------------------------|-------|
| 1. | Abstract | 1 |
| 2. | Introduction | 2-3 |
| 3. | Literature Review | 4 |
| 4. | Problem Identification and Objectives | 5 |
| 5. | System Methodology | 6-7 |
| 6. | Overview of Technologies | 8-12 |
| 7. | Implementation | |
| | 7.1 Coding | 13-18 |
| | 7.2 Results | 19-23 |
| 8. | Conclusion and Future Scope | 24 |
| 9. | References | 25 |

1. ABSTRACT

Recent years have seen an exponential growth in the field of machine learning, which has led to some state-of-the-art achievements in computer vision, natural language processing, etc. Models are trained in machine learning. There are many algorithms to train the models. While working with large datasets, it will take a lot of time to train the model. So to reduce the time consumed by the model while we are training it, the concept of parallelizing deep neural networks using MPI and GPU computing is used. In this project, MPI and GPU processing are used to create a neural network model in Python. This project's objective is to develop a sequential and parallel neural network model utilizing MPI and GPU and to assess how much faster training time and performance are as a result. Research has been done on data-based parallelism, model-based parallelism and implemented data-based parallelism as two ways to parallelize neural network models.

Keywords: Data parallelism, Model Parallelism, MPI, GPU, Parallel Computing, Deep Neural networks, Deep learning, Parallelism , Multi Processors.

2. INTRODUCTION

This development can be attributed to deep learning/deep neural nets, also known as deep neural network-based models. Deep learning models are very good at learning both high-level and low-level representations of the data (such as images, voices, languages, etc.), but this comes at the cost of a long training period and staggering computational requirements. This is principally caused by the fact that the foundation of any deep learning model is the recurrent learning of parameters throughout the whole parameter space. Additionally, this method only uses large matrix multiplications as operations. It necessitates lots of time and effort to compute these large matrices. As a result, GPU computing has recently offered a viable choice that might speed up the entire learning/training process. This study explores fundamental parallel deep neural network designs using the Fashion MNIST dataset. I developed two iterations of this model in Python: a sequential iteration and a parallel iteration.

Traditionally, in contexts with only one processor, machine learning has been used. In these settings, Significant inefficiencies inside this processing of models, from training via identification through distance and error calculation and beyond, can be driven upon with algorithmic bottlenecks. To be equipped to attain a utilization economy. An architecture incorporating shared memory in absence of parallel processing or execution necessitates executing a lot of usage on a lot of different data sets after completing independent tasks. Thus, paralleled programming is difficult from my perspective. When you employ data parallelism, each thread is fed a separate part of the same data while still using the same data overall. While using the same data for each thread while using model parallelism, the model is divided among the threads. The weights of the net are evenly distributed among the threads through model parallelism.

Multiple calculations or processes can be conducted simultaneously in parallel computing. Large problems can frequently be divided into fairly manageable proportions dealt with together at once. Bit-level parallelism, instruction-level parallelism, data parallelism, and assignment multithreading are still only a couple of minor parallel computing approaches. The use of polyphony in high-performance computing, which has long been popular, has grown more popular, despite the inherent limitations preventing frequency scaling. Since power consumption (and therefore heat generation) by computers has been a problem in recent years, parallel computing has emerged as the dominant paradigm in computer design, primarily in the form of multi-core processors.

In this project, MPI and GPU processing are utilized to create a simple neural network model in Python. This project's objective is to develop a sequential and parallel neural network model utilizing MPI and GPU and to assess how quickly training progresses and how well performance improves. I have researched data-based parallelism, model based parallelism, and implemented data-based parallelism as two ways to parallelize neural network models data-based parallelism, model-based parallelism, and implemented database parallelism. Algorithms are simply deployed in several processors in parallel processing. This generally relates to distributed processing since a typical machine learning algorithm actually requires doing multiple calculations (tasks) on different data sets.

Datasets were taken from kaggle, we have taken the Fashion MNIST dataset to work with the data and observe the output after performing parallelization on the dataset.

3. LITERATURE REVIEW

It is challenging to develop and implement efficient, presumably correct parallel neural network processing. The variety and massive scale data size have represented a critical test to build an adaptable and high-performance execution of deep learning neural networks. Due to its low cost and incredible parallel processing capability, the graphical processing unit, or GPU, has developed into a crucial component of high-performance computing (HPC). Utilizing the MPI framework to circulate deep neural networks is a decent decision since productivity and scalability have been demonstrated in modern practice. Accomplishing effective parallel algorithms for the GPU is definitely not an unimportant task. There are a few specialized technical restrictions that should be fulfilled to accomplish the expected outcome. By understanding the GPU architecture and its huge parallelism programming model, one can subdue a large number of the specialized limitations found along the way, design better GPU algorithms for computational physics problems and accomplish speedups that can arrive at up to two significant degrees of magnitude when contrasted with sequential implementations. Parallelizing deep neural networks using MPI and GPU ought to be simplicity of arranging, and is completely versatile and doesn't rely upon the technology. As a matter of fact, it permits moving helpful data to the correspondence library, which can be utilized to further develop equipment explicit MPI libraries.

4. PROBLEM IDENTIFICATION AND OBJECTIVES

The Problem: Whenever we use a single processor, the time taken by the processor to complete any kind of operations is more as the calculations and operations are done in a sequential manner. As the process needs to be completed in a sequential mode, the processor performs operations one after the other operations. This is what happens in a single CPU unit.

The main problems with sequential computing are

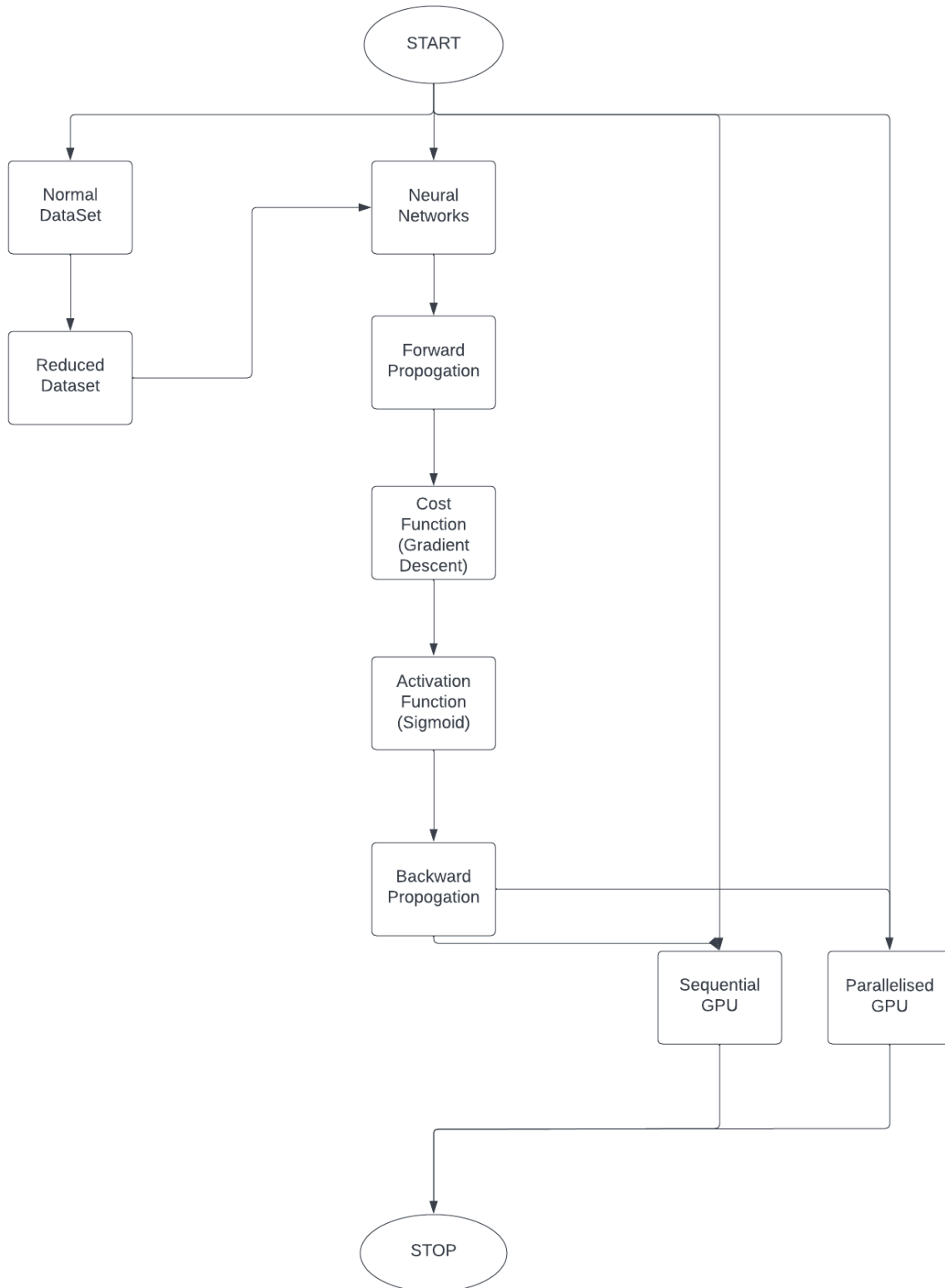
1. It performs badly and the workload is high when there is only one processor.
2. It requires more time to complete the whole process.
3. Slowly but surely, each instruction is implemented in order.

Proposed Solution: By parallelizing the deep neural networks using MPI and GPU computing, we can reduce the time taken by the processors to give the output. This is possible by constructing a 3-layered neural network and distributing the works to different threads and combining all of them at the very end. This is done to:

1. Increase the performance and reduce the workload of the processor as multiple processors are working simultaneously.
2. Perform all the instructions parallelly.
3. Reduce the time consumed to complete the whole.

5. SYSTEM METHODOLOGY

BASIC FLOWCHART



Initially, a dataset(Fashion MNIST) was acquired from kaggle. It contains grayscale images. It contains 10 different classes like shirts, pants, shorts , shoes,etc. The size of the dataset was 28x28 initially and it has been reduced to 20x20 by pooling.

After the dataset size is reduced to 20x20 size, A 3 layered neural network is built and the dataset is given to the neural network's input. As soon as the input is given to the neural network, the forward propagation takes place and the data is moved to the layer of output via the hidden layer. Activation function sigmoid is used in the hidden layers to perform the calculations.

After the output is acquired we find the error and perform backward propagation to change the weights. We performed data parallelism in this neural network. It speeds up the process

We are making some progress, but it seems to be minor; this may be because the dataset we utilized was too tiny to really utilize parallel processing. The fact that we can now quickly scale-out the computing power from one computer to the GPU cluster, however, is what's actually significant. This method can speed up the learning process significantly because GPU computing is still one of the best technologies for deep learning.

In the end, we compare the time consumed by the sequential model and parallel model to see which works faster. Overview of Technologies

Neural networks:

Neural Networks can simply be defined as a set of algorithms that are basically modeled with reference to the human brain, and are designed to identify patterns and solve any problems. It uses a set of functions to understand and interpret the given data. The neural networking system is inspired by the human neural or nervous system and the way the neurons in a human system function to understand and interpret data from the environment.

Neural Networks are useful for clustering and classifying. Both clustering and classifying categorize objects into various classes based on their features. They are quite similar to each other with a minute basic difference. In case of classifying, the input labels are predefined and are assigned to input instances according to their individual properties but in case of clustering those predefined labels are absent so clustering just groups the input instances based on the similar features they possess without any class labels

Gradient descent:

A major or critical part of machine and deep learning is refining and improving the algorithm used for the optimization of the models at their core. Gradient descent is the easiest optimization algorithm to understand and process and thus implement. It is used to find the local minimum of a function since it is a first-order iterative algorithm, that is it helps us find the lowest point from the data provided. It aims to minimize errors in algorithms between the actual output and the expected output

1. Batch Gradient descent:

In Batch gradient descent(BGD) when a training dataset is given, it determines the inaccuracy in each depiction present in the given dataset, and only after all the training datasets The model is updated and deployed to do it after being evaluated.

Batch gradient descent has several advantages. It has great efficiency in terms of computing and also produces a stable rate of error gradient. Some disadvantages it possesses are that it requires an entire dataset to be available to the algorithm in memory.

2. Stochastic Gradient Descent:

It is a type of gradient descent where only 1 training process takes place at a time. Hence, the parameters are updated after every single iteration where only one example has been executed or processed. This process is faster than batch gradient descent, comparatively.

The only disadvantage Stochastic Gradient descent has is that even when the training datasets are larger in number, it still processes one at a time which can be an additional burden for the system as it would produce a large number of iterations.

3. Mini Batch gradient descent

Mini Batch gradient descent method is a fusion of BGD and SGD method and thus works faster than both. In this gradient descent method even when the training datasets are larger in number it is processed in batches of these training datasets all at a time. So it works for training sets that are both larger and lesser in number in terms of the number of iterations.

Forward propagation:

In Forward Propagation, as the name suggests, initial input data is uploaded in a forward direction through a network. As we know there are several layers present in a neural networking system and this input data uploaded is accepted by each hidden layer which then processes it according to the activation function (an activation function in a neural network is a function that gives a small output for any given small input) which is then passed onto the successive layers.

To generate some output, the input data should be given in a forward direction. The data should never be processed or flow in a reverse direction as it would never help us generate an output. This type of arrangement is known as a feed-forward network.

Backward Propagation:

Backward propagation could be called one of the most important concepts in neural networking as its main job is to classify the data given in the best possible way

In Forward propagation, the process moves from the input layer to the output layer in the neural networking system, but in the case of Backward propagation, the process moves backward from the output layer toward the input layer.

Backward propagation is usually preferred to reach the minimized loss function. It is considered to be an important process for improving the accuracy of certain processes in machine learning.

Working of backward propagation:

Backward propagation is a simple, easy and efficient way of programming. It does not necessarily require any knowledge about the network to be processed and it is quite a flexible method and works well.

Data Parallelism

Data Parallelism is the process of using the same model or dataset for every thread, that is for every chain of a neural network, but we give different parts of the data to every thread.

First, some copies of a model are created and released and then the data is broken into pieces and distributed to the corresponding devices. It finally groups all the results in the backward propagation step.

Model Parallelism

Model parallelism could be called the process of using the same data set or model for every single thread but here we divide the model among the threads.

It divides or breaks a model into layers across many cores, recreating the same model for all the training cores. So basically we need to clearly specify the neural networking layers and the immediate outputs or results so the corresponding cores.

MPI (Message Passing Interface)

Message Passing Interface acts as a communication protocol(a set of rules or instructions that must be followed for the exchange of information) in parallel programming. It is explicitly used so that the applications in a neural network can run in parallel across separate devices but are also connected by a network.

MPI usually just runs the same code on various processors which then communicate with each other. It contains and sometimes provides operations that are used to evenly distribute the workload among various processes in a cluster network.

GPU (Graphical Processing Unit)

The Graphical Processing Units are designed to carry out tasks that require graphical orientation. They are efficiently optimized to train deep learning models since they are useful in processing multiple processes simultaneously due to the presence of a larger number of cores. Also, the memory bandwidth in GPU is most suitable to handle large amounts of data for computations in deep learning.

GPUs perform multiple computations or processes simultaneously which in turn helps with the distribution of various training processes and also speeds up the various operations. Additionally, in this we can gather multiple cores that use lesser resources to function without wasting power and also work efficiently.

6. Implementation

This section consists of coding or code outline for various files.

Importing required modules, Initializing neural network structure and loading datasets

This part of the code initializes the structure of the neural network and loads the datasets to perform model training using MPI and GPU.

```
1  import functools
2  import numpy as np
3  import math
4  import os
5  import scipy.io as sio
6  import time
7  from mpi4py import MPI
8
9  comm = MPI.COMM_WORLD
10
11
12  Input_layer_size = 400
13  Hidden_layer_size = 25
14  Output_layer_size = 10
15
16
17  Matrix_dot = np.dot
18
19
20  def convert_memory_ordering_f2c(array):
21      if np.isfortran(array) is True:
22          return np.ascontiguousarray(array)
23      else:
24          return array
25
26
27  def load_training_data(training_file='mnistdata.mat'):
28      training_data = sio.loadmat(training_file)
29      inputs = training_data['X'].astype('f8')
30      inputs = convert_memory_ordering_f2c(inputs)
31      labels = training_data['y'].reshape(training_data['y'].shape[0])
32      labels = convert_memory_ordering_f2c(labels)
33      return (inputs, labels)
34
35
36  def load_weights(weight_file='mnistweights.mat'):
37      weights = sio.loadmat(weight_file)
38      theta1 = convert_memory_ordering_f2c(weights['Theta1'].astype('f8')) # size: 25 entries, each has 401 numbers
39      theta2 = convert_memory_ordering_f2c(weights['Theta2'].astype('f8')) # size: 10 entries, each has 26 numbers
40      return (theta1, theta2)
41
42
43  def rand_init_weights(size_in, size_out):
44      epsilon_init = 0.12
45      return np.random.rand(size_out, 1 + size_in) * 2 * epsilon_init - epsilon_init
46
```


Activation function(Sigmoid)

A sigmoid is a variety of mathematical functions that has an identifiable S-shaped curve. The logistic function, the hyperbolic tangent, and the arctangent are some extensively encountered sigmoid functions.

```
46
47
48 def sigmoid(z):
49     return 1.0 / (1 + pow(math.e, -z))
50
51
52 def sigmoid_gradient(z):
53     return sigmoid(z) * (1 - sigmoid(z))
54
55
```

Cost function(Construction of neural network, forward propagation and backward propagation)

- **Construction of neural network**

A three layer network has been used which has two hidden layers.

```
def cost_function(theta1, theta2, input_layer_size, hidden_layer_size, output_layer_size, inputs, labels, regular=0):
    input_layer = np.insert(inputs, 0, 1, axis=1) # add bias, 5000x401

    time_start = time.time()
    hidden_layer = Matrix_dot(input_layer, theta1.T)
    hidden_layer = sigmoid(hidden_layer)
    hidden_layer = np.insert(hidden_layer, 0, 1, axis=1) # add bias, 5000x26
    time_end = time.time()
    if comm.rank == 0:
        print('\tconstruction: hidden layer dot costs {} secs'.format(time_end - time_start))

    time_start = time.time()
    output_layer = Matrix_dot(hidden_layer, theta2.T) # 5000x10
    output_layer = sigmoid(output_layer)
    time_end = time.time()
    if comm.rank == 0:
        print('\tconstruction: output layer dot costs {} secs'.format(time_end - time_start))
```

- **Forward propagation**

Forward propagation is the method of evaluating and accumulating intermediate parameters (including outputs) for a neural network in the sorted position from the source to the destination layer (also known as forward pass). Now, let's step-by-step walk through how a neural network with a single hidden layer works.

```

88
89 # forward propagation: calculate cost
90 time_start = time.time()
91 cost = 0.0
92 for training_index in range(len(inputs)):
93     outputs = [0] * output_layer_size
94     outputs[labels[training_index]-1] = 1
95
96     for k in range(output_layer_size):
97         error = -outputs[k] * math.log(output_layer[training_index][k]) - (1 - outputs[k]) * math.log(1 - output_layer[training_index][k])
98         cost += error
99 cost /= len(inputs)
100 time_end = time.time()
101 if comm.rank == 0:
102     print('\tforward prop: costs {} secs'.format(time_end - time_start))
103

```

• Backward propagation

Backpropagation is the way of calculating the gradient of neural network parameters. The technique basically traverses the network from the output to the input layer in reverse order through using chain rule from calculus. The strategy keeps any intermediate variables (partial derivatives) involved in calculating the gradient with respect to a few parameters.

```

104 # back propagation: calculate gradients
105 time_start = time.time()
106 theta1_grad = np.zeros_like(theta1) # 25x401
107 theta2_grad = np.zeros_like(theta2) # 10x26
108 for index in range(len(inputs)):
109     # transform label y[i] from a number to a vector.
110     outputs = np.zeros((1, output_layer_size)) # (1,10)
111     outputs[0][labels[index]-1] = 1
112
113     # calculate delta3
114     delta3 = (output_layer[index] - outputs).T # (10,1)
115
116     # calculate delta2
117     z2 = Matrix_dot(theta1, input_layer[index:index+1].T) # (25,401) x (401,1)
118     z2 = np.insert(z2, 0, 1, axis=0) # add bias, (26,1)
119     delta2 = np.multiply(
120         Matrix_dot(theta2.T, delta3), # (26,10) x (10,1)
121         sigmoid_gradient(z2) # (26,1)
122     )
123     delta2 = delta2[1:]
124     theta1_grad += Matrix_dot(delta2, input_layer[index:index+1])
125     # (10,26) = (10,1) x (1,26)
126     theta2_grad += Matrix_dot(delta3, hidden_layer[index:index+1])
127 theta1_grad /= len(inputs)
128 theta2_grad /= len(inputs)
129 time_end = time.time()
130 if comm.rank == 0:
131     print('\tback prop: costs {} secs'.format(time_end - time_start))
132
133 return cost, (theta1_grad, theta2_grad)
134

```

Gradient Descent

Gradient descent seems to be an iterative first-order optimization technique for discovering the lower bound or optimum of a given function (GD). This method is often used in machine learning (ML) and deep learning (DL) to shrink a cost/loss function (e.g. in a linear regression). Due to its importance and flexibility of use, this procedure is frequently taught at the beginning of substantially all machine learning courses.

```
def gradient_descent(inputs, labels, learningrate=0.8, iteration=50):
    """
    @return cost and trained model (weights).
    """
    if Distributed is True:
        if comm.rank == 0:
            theta1 = rand_init_weights(Input_layer_size, Hidden_layer_size)
            theta2 = rand_init_weights(Hidden_layer_size, Output_layer_size)
        else:
            theta1 = np.zeros((Hidden_layer_size, Input_layer_size + 1))
            theta2 = np.zeros((Output_layer_size, Hidden_layer_size + 1))
        comm.Barrier()
        if comm.rank == 0:
            time_bcast_start = time.time()
            comm.Bcast([theta1, MPI.DOUBLE])
            comm.Barrier()
            comm.Bcast([theta2, MPI.DOUBLE])
            if comm.rank == 0:
                time_bcast_end = time.time()
                print('\tBcast theta1 and theta2 uses {} secs.'.format(time_bcast_end - time_bcast_start))
    else:
        theta1 = rand_init_weights(Input_layer_size, Hidden_layer_size)
        theta2 = rand_init_weights(Hidden_layer_size, Output_layer_size)

    cost = 0.0
    for i in range(iteration):
        time_iter_start = time.time()

        if Distributed is True:
            # Scatter training data and labels.
            sliced_inputs = np.asarray(np.split(inputs, comm.size))
            sliced_labels = np.asarray(np.split(labels, comm.size))
            inputs_buf = np.zeros((int(len(inputs)/comm.size), Input_layer_size))
            labels_buf = np.zeros((int(len(labels)/comm.size), dtype='uint8'))
```

```

comm.Barrier()
if comm.rank == 0:
    time_scatter_start = time.time()
comm.Scatter(sliced_inputs, inputs_buf)
if comm.rank == 0:
    time_scatter_end = time.time()
    print('\tScatter inputs uses {} secs.'.format(time_scatter_end - time_scatter_start))

comm.Barrier()
if comm.rank == 0:
    time_scatter_start = time.time()
comm.Scatter(sliced_labels, labels_buf)
if comm.rank == 0:
    time_scatter_end = time.time()
    print('\tScatter labels uses {} secs.'.format(time_scatter_end - time_scatter_start))

# Calculate distributed costs and gradients of this iteration
# by cost function.
comm.Barrier()
cost, (theta1_grad, theta2_grad) = cost_function(theta1, theta2,
    Input_layer_size, Hidden_layer_size, Output_layer_size,
    inputs_buf, labels_buf, regular=0)

# Gather distributed costs and gradients.
comm.Barrier()
cost_buf = [0] * comm.size
try:
    cost_buf = comm.gather(cost)
    cost = sum(cost_buf) / len(cost_buf)
except TypeError as e:
    print('[{0}] {1}'.format(comm.rank, e))

theta1_grad_buf = np.asarray([np.zeros_like(theta1_grad)] * comm.size)
comm.Barrier()
if comm.rank == 0:
    time_gather_start = time.time()
comm.Gather(theta1_grad, theta1_grad_buf)
if comm.rank == 0:
    time_gather_end = time.time()
    print('\tGather theta1 uses {} secs.'.format(time_gather_end - time_gather_start))
comm.Barrier()
theta1_grad = functools.reduce(np.add, theta1_grad_buf) / comm.size

theta2_grad_buf = np.asarray([np.zeros_like(theta2_grad)] * comm.size)
comm.Barrier()
if comm.rank == 0:
    time_gather_start = time.time()
comm.Gather(theta2_grad, theta2_grad_buf)
if comm.rank == 0:
    time_gather_end = time.time()
    print('\tGather theta2 uses {} secs.'.format(time_gather_end - time_gather_start))
comm.Barrier()
theta2_grad = functools.reduce(np.add, theta2_grad_buf) / comm.size
else:
    cost, (theta1_grad, theta2_grad) = cost_function(theta1, theta2,

```

```

        Input_layer_size, Hidden_layer_size, Output_layer_size,
        inputs, labels, regular=0)

    theta1 -= learningrate * theta1_grad
    theta2 -= learningrate * theta2_grad

    if Distributed is True:
        # Sync-up weights for distributed worknodes.
        comm.Bcast([theta1, MPI.DOUBLE])
        comm.Bcast([theta2, MPI.DOUBLE])
        comm.Barrier()

    time_iter_end = time.time()
    if comm.rank == 0:
        print('Iteration {0} (learning rate {2}, iteration {3}), cost: {1}, time: {4}'.format(
            i+1, cost, learningrate, iteration, time_iter_end - time_iter_start)
        )
    return cost, (theta1, theta2)

```

Main Function

All the function calls are made from this main function

```

1  if __name__ == '__main__':
2
3      inputs, labels = load_training_data()
4      model = train(inputs, labels, learningrate=0.1, iteration=60)
5
6      outputs = predict(model, inputs)
7      correct_prediction = 0
8      for i, predict in enumerate(outputs):
9          if predict == labels[i][0]:
10             correct_prediction += 1
11     precision = float(correct_prediction) / len(labels)
12     print('precision: {}'.format(precision))

```

7.1 Results

Sequential Code Output

```
(vamsi@Vamsis-MacBook-Air Parallel-Deep-Learning-in-Python % python3 mnist-nn.py
Iteration 1 (learning rate 0.1, iteration 60), cost: 7.4984315165802355
Iteration 2 (learning rate 0.1, iteration 60), cost: 6.203170746377222
Iteration 3 (learning rate 0.1, iteration 60), cost: 5.347060114573348
Iteration 4 (learning rate 0.1, iteration 60), cost: 4.7502019136897795
Iteration 5 (learning rate 0.1, iteration 60), cost: 4.325932893158747
Iteration 6 (learning rate 0.1, iteration 60), cost: 4.023092747592217
Iteration 7 (learning rate 0.1, iteration 60), cost: 3.8069981654496003
Iteration 8 (learning rate 0.1, iteration 60), cost: 3.6527847193103
Iteration 9 (learning rate 0.1, iteration 60), cost: 3.542505480066043
Iteration 10 (learning rate 0.1, iteration 60), cost: 3.4633302029886806
Iteration 11 (learning rate 0.1, iteration 60), cost: 3.406179756125658
Iteration 12 (learning rate 0.1, iteration 60), cost: 3.3646679099971655
Iteration 13 (learning rate 0.1, iteration 60), cost: 3.33430942059134
Iteration 14 (learning rate 0.1, iteration 60), cost: 3.3119481234996004
Iteration 15 (learning rate 0.1, iteration 60), cost: 3.295353841679875
Iteration 16 (learning rate 0.1, iteration 60), cost: 3.282942110388112
Iteration 17 (learning rate 0.1, iteration 60), cost: 3.2735803993828894
Iteration 18 (learning rate 0.1, iteration 60), cost: 3.2664541833539142
Iteration 19 (learning rate 0.1, iteration 60), cost: 3.260974093585416
Iteration 20 (learning rate 0.1, iteration 60), cost: 3.256711221210622
Iteration 21 (learning rate 0.1, iteration 60), cost: 3.25335175547364
Iteration 22 (learning rate 0.1, iteration 60), cost: 3.250664963836423
Iteration 23 (learning rate 0.1, iteration 60), cost: 3.2484804345062197
Iteration 24 (learning rate 0.1, iteration 60), cost: 3.2466717931420153
Iteration 25 (learning rate 0.1, iteration 60), cost: 3.2451449769602765
Iteration 26 (learning rate 0.1, iteration 60), cost: 3.243829739672562
Iteration 27 (learning rate 0.1, iteration 60), cost: 3.2426734625673506
Iteration 28 (learning rate 0.1, iteration 60), cost: 3.2416366224719657
Iteration 29 (learning rate 0.1, iteration 60), cost: 3.240689457431851
Iteration 30 (learning rate 0.1, iteration 60), cost: 3.2398095031302336
Iteration 31 (learning rate 0.1, iteration 60), cost: 3.238979765658492
Iteration 32 (learning rate 0.1, iteration 60), cost: 3.238187361577276
Iteration 33 (learning rate 0.1, iteration 60), cost: 3.237422502620512
Iteration 34 (learning rate 0.1, iteration 60), cost: 3.236677735592795
Iteration 35 (learning rate 0.1, iteration 60), cost: 3.2359473719003606
Iteration 36 (learning rate 0.1, iteration 60), cost: 3.2352270584514606
Iteration 37 (learning rate 0.1, iteration 60), cost: 3.2345134542502665
Iteration 38 (learning rate 0.1, iteration 60), cost: 3.2338039862151065
Iteration 39 (learning rate 0.1, iteration 60), cost: 3.233096664519065
Iteration 40 (learning rate 0.1, iteration 60), cost: 3.2323899427428477
Iteration 41 (learning rate 0.1, iteration 60), cost: 3.231682611827897
Iteration 42 (learning rate 0.1, iteration 60), cost: 3.2309737195661006
Iteration 43 (learning rate 0.1, iteration 60), cost: 3.2302625094115336
Iteration 44 (learning rate 0.1, iteration 60), cost: 3.229548373932553
Iteration 45 (learning rate 0.1, iteration 60), cost: 3.228830819368697
Iteration 46 (learning rate 0.1, iteration 60), cost: 3.2281094386208578
Iteration 47 (learning rate 0.1, iteration 60), cost: 3.227383890652918
Iteration 48 (learning rate 0.1, iteration 60), cost: 3.226653884770207
Iteration 49 (learning rate 0.1, iteration 60), cost: 3.225919168612846
Iteration 50 (learning rate 0.1, iteration 60), cost: 3.2251795189791723
Iteration 51 (learning rate 0.1, iteration 60), cost: 3.224434734807824
Iteration 52 (learning rate 0.1, iteration 60), cost: 3.2236846318071115
Iteration 53 (learning rate 0.1, iteration 60), cost: 3.2229290383423885
Iteration 54 (learning rate 0.1, iteration 60), cost: 3.2221677922842753
Iteration 55 (learning rate 0.1, iteration 60), cost: 3.2214007385920596
Iteration 56 (learning rate 0.1, iteration 60), cost: 3.2206277274588624
Iteration 57 (learning rate 0.1, iteration 60), cost: 3.2198486128876835
Iteration 58 (learning rate 0.1, iteration 60), cost: 3.2190632515963915
Iteration 59 (learning rate 0.1, iteration 60), cost: 3.218271502175592
Iteration 60 (learning rate 0.1, iteration 60), cost: 3.2174732244407127
precision: 0.3578
```

GPU mode without parallelism:

```
GPU mode
theano expression creation costs 0.00930476188659668 secs
Parallelism: no
    Bcast theta1 and theta2 uses 4.553794860839844e-05 secs.
    Scatter inputs uses 0.0017404556274414062 secs.
    Scatter labels uses 1.33514404296875e-05 secs.
    construction: hidden layer dot costs 0.017520904541015625 secs
    construction: output layer dot costs 0.005150794982910156 secs
    forward prop: costs 0.13963890075683594 secs
    back prop: costs 0.9486510753631592 secs
    Gather theta1 uses 0.00027370452880859375 secs.
    Gather theta2 uses 1.3589859008789062e-05 secs.
Iteration 1 (learning rate 0.1, iteration 60), cost: 6.545538332227445, time: 1.1277730464935303
    Scatter inputs uses 0.0017504692077636719 secs.
    Scatter labels uses 1.0013580322265625e-05 secs.
    construction: hidden layer dot costs 0.0174405574798584 secs
    construction: output layer dot costs 0.0050241947174072266 secs
    forward prop: costs 0.1349499225616455 secs
    back prop: costs 0.9307937622070312 secs
    Gather theta1 uses 2.193450927734375e-05 secs.
    Gather theta2 uses 1.239776611328125e-05 secs.
Iteration 2 (learning rate 0.1, iteration 60), cost: 5.570359414428981, time: 1.1057555675506592
    Scatter inputs uses 0.002033710479736328 secs.
    Scatter labels uses 1.1682510375976562e-05 secs.
    construction: hidden layer dot costs 0.019628524780273438 secs
    construction: output layer dot costs 0.004518032073974609 secs
    forward prop: costs 0.1388859748840332 secs
    back prop: costs 0.9215435981750488 secs
    Gather theta1 uses 2.2411346435546875e-05 secs.
    Gather theta2 uses 1.0728836059570312e-05 secs.
Iteration 3 (learning rate 0.1, iteration 60), cost: 4.9019502930163235, time: 1.1001486778259277
    Scatter inputs uses 0.0017066001892089844 secs.
    Scatter labels uses 9.775161743164062e-06 secs.
    construction: hidden layer dot costs 0.0176239013671875 secs
    ✓ 1m 8s    completed at 20:33
```


Iteration 56 (learning rate 0.1, iteration 60), cost: 3.2173922426143697, time: 1.152275562286377
Gather theta1 uses 2.0503997802734375e-05 secs.
Gather theta2 uses 1.1444091796875e-05 secs.
Scatter inputs uses 0.0016858577728271484 secs.
Scatter labels uses 0.00032448768615722656 secs.
construction: hidden layer dot costs 0.018312692642211914 secs
construction: output layer dot costs 0.0050699710845947266 secs
forward prop: costs 0.14115118980407715 secs
back prop: costs 0.9490973949432373 secs
Gather theta1 uses 1.6450881958007812e-05 secs.
Gather theta2 uses 9.775161743164062e-06 secs.
Iteration 57 (learning rate 0.1, iteration 60), cost: 3.2165876077619515, time: 1.1293275356292725
Scatter inputs uses 0.0016353130340576172 secs.
Scatter labels uses 1.0728836059570312e-05 secs.
construction: hidden layer dot costs 0.01937556266784668 secs
construction: output layer dot costs 0.005156993865966797 secs
forward prop: costs 0.14019083976745605 secs
back prop: costs 0.9719610214233398 secs
Gather theta1 uses 1.8596649169921875e-05 secs.
Gather theta2 uses 4.76837158203125e-06 secs.
Iteration 58 (learning rate 0.1, iteration 60), cost: 3.215775706240834, time: 1.1493456363677979
Scatter inputs uses 0.001980304718017578 secs.
Scatter labels uses 1.811981201171875e-05 secs.
construction: hidden layer dot costs 0.018647432327270508 secs
construction: output layer dot costs 0.005421876907348633 secs
forward prop: costs 0.13241124153137207 secs
back prop: costs 0.9658398628234863 secs
Gather theta1 uses 0.00030493736267089844 secs.
Gather theta2 uses 1.0728836059570312e-05 secs.
Iteration 59 (learning rate 0.1, iteration 60), cost: 3.2149563923933266, time: 1.1390509605407715
Scatter inputs uses 0.0016622543334960938 secs.
Scatter labels uses 1.0728836059570312e-05 secs.
construction: hidden layer dot costs 0.022619009017944336 secs
construction: output layer dot costs 0.0046918392181396484 secs
forward prop: costs 0.1390852928161621 secs
back prop: costs 0.9768922328948975 secs
Gather theta1 uses 1.5020370483398438e-05 secs.
Gather theta2 uses 9.298324584960938e-06 secs.
Iteration 60 (learning rate 0.1, iteration 60), cost: 3.214129521186516, time: 1.1641230583190918
precision: 0.2996

✓ 1m 8s completed at 20:33

GPU mode with parallelism:

```
↳ CPU mode
Parallelism: yes
  construction: hidden layer dot costs 0.01697707176208496 secs
  construction: output layer dot costs 0.005660533905029297 secs
  forward prop: costs 0.14137625694274902 secs
  back prop: costs 0.6564035415649414 secs
Iteration 1 (learning rate 0.1, iteration 60), cost: 6.703611349510278, time: 0.8254992961883545
  construction: hidden layer dot costs 0.01684403419494629 secs
  construction: output layer dot costs 0.0046384334564208984 secs
  forward prop: costs 0.1373276710510254 secs
  back prop: costs 0.6897101402282715 secs
Iteration 2 (learning rate 0.1, iteration 60), cost: 5.684853247480495, time: 0.855006217956543
  construction: hidden layer dot costs 0.017127275466918945 secs
  construction: output layer dot costs 0.004649639129638672 secs
  forward prop: costs 0.14248228073120117 secs
  back prop: costs 0.7027475833892822 secs
Iteration 3 (learning rate 0.1, iteration 60), cost: 4.984062076713913, time: 0.8739066123962402
  construction: hidden layer dot costs 0.017966032028198242 secs
  construction: output layer dot costs 0.005415678024291992 secs
  forward prop: costs 0.14230632781982422 secs
  back prop: costs 0.6813180446624756 secs
Iteration 4 (learning rate 0.1, iteration 60), cost: 4.488327254955315, time: 0.8523626327514648
  construction: hidden layer dot costs 0.0170438289642334 secs
  construction: output layer dot costs 0.004861593246459961 secs
  forward prop: costs 0.1426382064819336 secs
  back prop: costs 0.6994366645812988 secs
Iteration 5 (learning rate 0.1, iteration 60), cost: 4.135381933080387, time: 0.8707549571990967
  construction: hidden layer dot costs 0.018872499465942383 secs
  construction: output layer dot costs 0.0046617984771728516 secs
  forward prop: costs 0.14195942878723145 secs
  back prop: costs 0.6765162944793701 secs
Iteration 6 (learning rate 0.1, iteration 60), cost: 3.8842630105507143, time: 0.848135232925415
  construction: hidden layer dot costs 0.016937971115112305 secs
  construction: output layer dot costs 0.005563497543334961 secs
  forward prop: costs 0.15550565719604492 secs
  back prop: costs 0.6665811538696289 secs
✓ 50s    completed at 20:39
```

```
Iteration 53 (learning rate 0.1, iteration 60), cost: 3.216470319138767, time: 0.8265223503112793
  construction: hidden layer dot costs 0.016994476318359375 secs
  construction: output layer dot costs 0.005446672439575195 secs
  forward prop: costs 0.14976787567138672 secs
  back prop: costs 0.6644794940948486 secs
Iteration 54 (learning rate 0.1, iteration 60), cost: 3.215632028588958, time: 0.8424324989318848
  construction: hidden layer dot costs 0.0171201229095459 secs
  construction: output layer dot costs 0.005633115768432617 secs
  forward prop: costs 0.15030646324157715 secs
  back prop: costs 0.698336124420166 secs
Iteration 55 (learning rate 0.1, iteration 60), cost: 3.2147874375896466, time: 0.8768243789672852
  construction: hidden layer dot costs 0.01639580726623535 secs
  construction: output layer dot costs 0.004687786102294922 secs
  forward prop: costs 0.13793635368347168 secs
  back prop: costs 0.729034423828125 secs
Iteration 56 (learning rate 0.1, iteration 60), cost: 3.213936409631522, time: 0.8939335346221924
  construction: hidden layer dot costs 0.016814708709716797 secs
  construction: output layer dot costs 0.005507946014404297 secs
  forward prop: costs 0.13576912879943848 secs
  back prop: costs 0.6824018955230713 secs
Iteration 57 (learning rate 0.1, iteration 60), cost: 3.2130788120939417, time: 0.8455753326416016
  construction: hidden layer dot costs 0.017169475555419922 secs
  construction: output layer dot costs 0.005397796630859375 secs
  forward prop: costs 0.13457155227661133 secs
  back prop: costs 0.6858577728271484 secs
Iteration 58 (learning rate 0.1, iteration 60), cost: 3.2122145153582857, time: 0.8482482433319092
  construction: hidden layer dot costs 0.016921043395996094 secs
  construction: output layer dot costs 0.005540609359741211 secs
  forward prop: costs 0.14007353782653809 secs
  back prop: costs 0.6677303314208984 secs
Iteration 59 (learning rate 0.1, iteration 60), cost: 3.2113433921276227, time: 0.8360872268676758
  construction: hidden layer dot costs 0.01661229133605957 secs
  construction: output layer dot costs 0.005562782287597656 secs
  forward prop: costs 0.14515447616577148 secs
  back prop: costs 0.7302095890045166 secs
Iteration 60 (learning rate 0.1, iteration 60), cost: 3.210465316905576, time: 0.9031646251678467
precision: 0.362
```

7. CONCLUSION AND FUTURE SCOPE

In all CPU vs. GPU experiments, the parallel model is speeding up more than the sequential approach. On CPU-based computing, we are seeing a speedup of around 30%, and on GPU-based computing, we are seeing a speedup of more than 30%. When you distribute your burden and parallelize your learning/training process, this is to be expected. If we utilize a large dataset or have a more complex/deep model, I think we can accelerate things considerably. Using CUDA allows GPU computing to be utilized considerably more. The transition from serial to parallel computing has greatly altered the computation graph. By using multiple core CPUs, Mostly in the domain of parallel computing, tech behemoth Intel has already taken the first steps.. Future computer technology will undergo a positive transformation thanks to parallelised processing. Parallelising the computing process plays a stronger function in assisting us in maintaining our connection to one another as the world becomes even more interconnected than before. It is especially more essential as networks, distributed systems, and computers with several processors get quicker.

8. REFERENCES

Research Papers:

- M. Garland et al., "Parallel Computing Experiences with CUDA," in IEEE Micro, vol. 28, no. 4, pp. 13-27, July-Aug. 2008, doi: 10.1109/MM.2008.57.
- Navarro, C., Hitschfeld-Kahler, N., & Mateu, L. (2014). A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. Communications in Computational Physics, 15(2), 285-329. doi:10.4208/cicp.110113.010813a
- S. W. Keckler, W. J. Dally, B. Khailany, M. Garland and D. Glasco, "GPUs and the Future of Parallel Computing," in IEEE Micro, vol. 31, no. 5, pp. 7-17, Sept.-Oct. 2011, doi: 10.1109/MM.2011.89.
- Alerstam E, Svensson T, Andersson-Engels S. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. Journal of biomedical optics. 2008 Nov;13(6):060504.
- Berger ED, McKinley KS, Blumofe RD, Wilson PR. Hoard: A scalable memory allocator for multithreaded applications. ACM Sigplan Notices. 2000 Nov 1;35(11):117-28.
- Nickolls J, Dally WJ. The GPU computing era. IEEE micro. 2010 Apr 12;30(2):56-69.
- Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J. and Karp, S., 2008. Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep, 15, p.181.
- Collange, Sylvain, David Defour, and Yao Zhang. "Dynamic detection of uniform and affine vectors in GPGPU computations." In European Conference on Parallel Processing, pp. 46-55. Springer, Berlin, Heidelberg, 2009.

Plagiarism

PARALLELIZING DEEP NN USING MPI AND GPU COMPUTING

ORIGINALITY REPORT

5%

SIMILARITY INDEX

3%

INTERNET SOURCES

1%

PUBLICATIONS

4%

STUDENT PAPERS

PRIMARY SOURCES

1

Submitted to Asia Pacific Institute of
Information Technology

Student Paper

1%

2

en.wikipedia.org

Internet Source

1%

3

Submitted to Anglia Ruskin University

Student Paper

1%

4

Submitted to Houston Community College

Student Paper

1%

5

www.dezyre.com

Internet Source

1%

6

www.wovo.org

Internet Source

<1%

7

repositorio.yachaytech.edu.ec

Internet Source

<1%

8

Managerial Auditing Journal, Volume 9, Issue
3 (2006-09-19)

Publication

<1%

9

library.samdu.uz

Internet Source

<1%

Exclude quotes Off

Exclude bibliography Off

Exclude matches

< 7 words