Fall 12-2021

Comparison Between CPU and GPU for Parallel Implementation for a Neural Network Model Using Tensorflow and a Big Dataset

Intisar Alkaabwi
University of Maine, intisar.alkaabawi@maine.edu

COMPARISON BETWEEN CPU AND GPU FOR PARALLEL IMPLEMENTATION FOR A NEURAL NETWORK MODEL USING TENSORFLOW AND A BIG DATASET

By

Intisar Alkaabawi

B.S. The University of Mustansiriyah, Iraq, 2006

A THESIS

Requirements for the Degree of

Master of Science

(in Computer Engineering)

The Graduate School

The University of Maine

December 2021

Advisory Committee:

Bruce Segee, Henry R. and Grace V. Butler, Professor, Electrical and Computer Engineering, Advisor

Vince Weaver, Associate Professor, Electrical and Computer Engineering Department

Chaofan Chen, Assistant Professor, Computer Science Department

COMPARISON BETWEEN CPU AND GPU FOR PARALLEL IMPLEMENTATION FOR A NEURAL NETWORK MODEL USING TENSORFLOW AND A BIG DATASET

By Intisar Alkaabawi
Thesis Advisor: Dr. Bruce Segee

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Engineering)
December 2021

ABSTRACT

Machine learning is a rapidly growing field that has become more common of late. Because of the demanding computational usage of machine learning, this field has many dimensions needing research. TensorFlow has been developed to deal with and analyze neural networks computation. In particular, TensorFlow is often used in one of the machine learning branches and is called deep learning. This work discusses the performance of a deep learning model to train a very large dataset with TensorFlow. It compares performance when the run happens on CPUs and on GPUs regarding the run time and speed. The run time is an important factor for deep learning projects. The goal is to find the most efficient machine and platform to run the neural networks computation. TensorFlow provides all the resources and operations that are needed to process the neural

networks computations. TensorFlow has two versions 1.0 and 2.0. This work uses TensorFlow 2.0 which is easier to code, faster to build the models, and faster for training time. Also, TensorFlow 2.0 has the methods used to distribute the run on multi-CPUs and multi-GPUs which use the strategy scope to run the model in parallel. The hardware utilized for this work consist of two clusters referred to as BlueMoon CPU and DeepGreen GPU. These clusters were developed by the University of Vermont for artificial intelligence research. The results show the performance of running the model for training a large dataset that becomes better each time the number of processors increases. The speedup is the highest when training a large batch size of samples with a higher number of processors. When more devices (GPU) have been added to these processors to run the model the performance becomes faster especially for the larger batch sizes. When the model runs on GPUs it requires a CPU to make the computation processing complete. Reducing the CPU number that is used to distribute the data on multi- GPUs makes the speedup higher than using more CPUs that are responsible to distribute the data on the same number of GPUs. The results contain the comparison run time of the classification model for the same dataset when run on a different machine with or without accelerator and using two different TensorFlow versions. The comparison results show using the TensorFlow 2.0 is better than when using the TensorFlow 1.0. Running the model with accelerator achieves more speedup and running the model on the clusters with both TensorFlow versions obtains higher performance.

DEDICATION

This thesis is dedicated to my loving parents, who have loved me unconditionally. Also, a special feeling of gratitude is to my husband and my adorable children, who have been a continuous source of support and encouragement during the challenges of graduate school and life.

ACKNOWLEDGEMENTS

I wish to express my sincere thankfulness to my supervisor, Professor Bruce Segee, for his support and help. I appreciate the help from Andrea Elledge, Operations and Outreach Administrator, Kerime Toksu, Research Computing Engineer, and the other team members of Vermont Advanced Computing Core for helping me by creating an account and answering all my questions about how to use the clusters. Thanks to Margaret Clancey, a close family friend, for the continuous support. Thanks to my parents, sisters, and brothers, they did not forget me with their prayers. Thank you to all the others who helped make this work a success.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE: INTRODUCTION AND MOTIVATION

Machine learning is a relatively new and growing field whereby computers can "learn" from big data sets by building a nonlinear model. The model is somewhat analogous to the processing of neurons in living creatures (Wang, 2020). In machine learning, a newer branch called deep learning has become more commonly used in research and other applications. Deep Learning is a branch of machine learning based on artificial neural networks with representation learning. Deep Learning is a type of software that simulates the network of neurons in the human brain. It is called deep learning because it uses deep and many layers of neural networks (Hans-D *et al.*, 2017). Deep learning uses neural networks to learn from a dataset and give predictions for previously unseen data (Goldsborough, 2016). Although deep learning is used in many fields such as speech recognition, medical imaging, etc., the size of the data and the computation required present significant challenges. The computations require a quite powerful platform to compute and find the results from this process (Lopes *et al.*, 2010). Big data generally has multiple processing dimensions. Other challenges in machine learning computation include storage, data management, and data processing. Depending on these challenges and dimensions, it is important to use good tools and the right architecture to analyze, compute and implement the artificial neural network (Adiyoso, Widiarto, *et al.*, 2018). There are many tools and architectures that can be used to analyze and implement big data processing, such as CPU clusters and GPU clusters. TensorFlow is a good platform or tool to handle big data processing (Abadi *et al*., 2016).

The deep learning training process typically has millions of parameters that are learned during the training (Hassan et al., 2014). Interestingly, high-performance is becoming more advanced, which increases the capability of the computation and gives the researchers more opportunities to pursue

deep learning in many different fields (Lopes et al., 2010). It is important to choose the correct computational method depending on the dataset used (especially for image classification datasets) and the specific hardware used to implement any specific application (Mustafa et al., 2019). The efficient use of a graphical processing unit GPU with a parallel distribution method to train deep learning models on the big datasets is a very good way to accelerate the time of training (Adie, 2018).

This thesis focuses on comparing the running time for a parallel implementation for the machine learning model using TensorFlow API (TensorFlow.org, 2015) and a big dataset on multiple GPUs and multiple CPUs.

TensorFlow is an end-to-end open-source platform for machine learning computation. It has a flexible environment of tools and many libraries to enable the researchers to build their models. TensorFlow can be a good way to compute many algorithms in deep neural networks and a wide area of computer science and other fields (TensorFlow.org, 2015). The GPU system already has the parallel computation framework and programming model that work with C/C++ languages like CUDA (Lopes, 2010). TensorFlow is a machine learning framework that can be used with systems ranging from small machines such as phones up to large-scale distributed systems of hundreds of machines containing thousands of computational devices. The computational devices supported include GPU as well as CPU. TensorFlow scales to very big sizes of a dataset (TensorFlow.org, 2015).

The TensorFlow computation can be visualized as a graph that is composed of an interconnected set of nodes (Goldsborough, 2016). The graph represents the dataflow of the computation. The graph has extensions that allow some kinds of nodes to maintain and update the persistent states

for looping and branching control structures (Abadi *et al.*, 2016). The edges between the nodes have so called control dependencies. These control dependencies require that the source node must finish executing before the destination node starts executing for the same data.

The math operations in TensorFlow have names to represent the computation we want to implement in the model (e.g., “add”, “multiply”, etc.) to make it easier to deal with tensors in multi-dimensional arrays (Abadi et al., 2016).

Figure 1.1 shows a TensorFlow graph that performs computation on the inputs using nodes that represent the operations and a TensorFlow graph using edges that describe tensors that get transferred between the nodes. At the end of the computation, the ReLu function is used to produce the final result. ReLU is a type of activation function. ReLU stands for the rectified linear unit, which is a linear unit for all positive values, and zero for all negative values. It is defined as y = max (0, x) in Mathematics. ReLU is the most commonly used activation function in neural networks, ReLU is usually a good first choice (Danqing Liu, 2017).

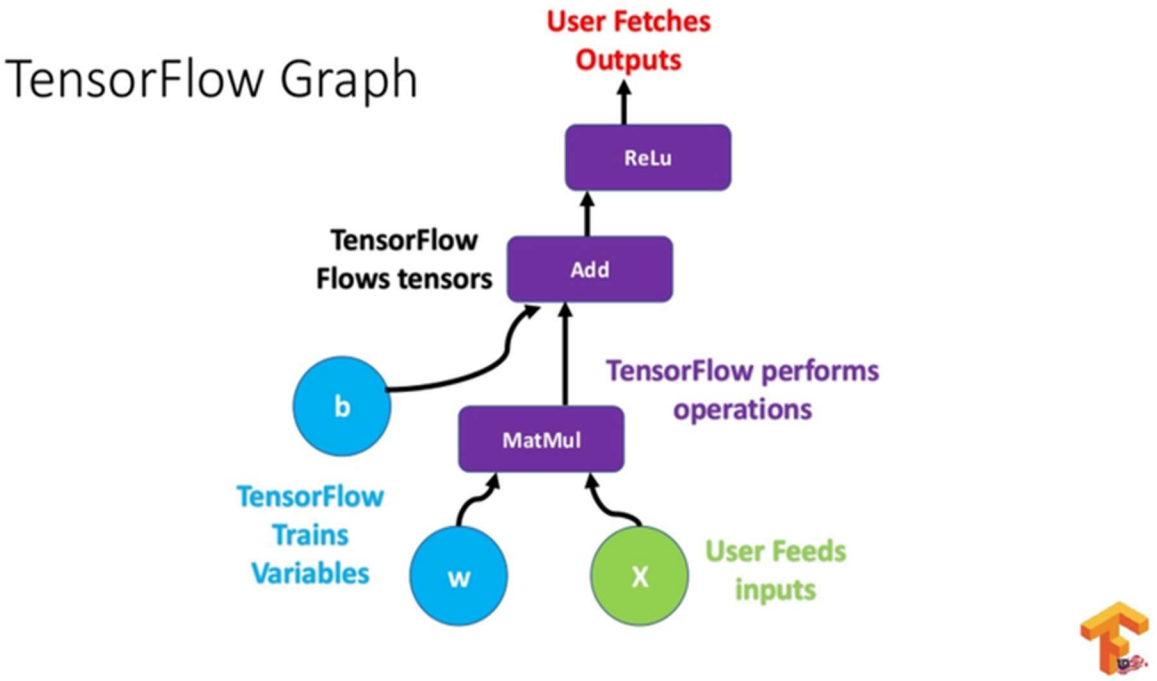

Figure 1-1 A directed graph is describing a TensorFlow computation.

To execute the computations in TensorFlow, shown in Figure 1.1, the Graph is considered as a Session. w, x, and b are tensors over the edges of this graph. MatMul is an operation over the tensors w and x; after that, Add is called and adds the result of the previous operator with b. At the end of the computation, the ReLu or any one of the other activation functions is used to produce the wanted result. (Ravi Ranjan Singh, 2020).

TensorFlow works as follows (see Figure 1.2): First, a Session is created to write arguments. Next, the Run interface is called to compute the output that we want to find. In machine learning, the parameters of the model are stored in tensors held in variables and updated as part of Run for the

model. By using Sessions and Variables, and then calling Run; each node can be updated. Beyond that, training is accomplished by calling Run thousands or millions of times (Goldsborough, 2016).

```
1.  graph = tf.Graph()
2.  with graph.as_default() as g:
3.      w1 = tf.Variable(np.array([1,2], dtype = np.float32))
4.      w2 = tf.Variable(np.array([2,2], dtype = np.float32))
5.
6.      wx = tf.multiply(w1, w2)
7.      initialize = tf.global_variables_initializer()
8.
9.  with tf.Session(graph=graph) as sess:
10.     sess.run(initialize)
11.     wx_v= sess.run([wx])
12.   print(wx_v)
```

Figure 1-2 An example of creating a Session to write arguments and calling the Run interface to compute the wanted output (tutorialexample.com, 2020).

Google Brain released the first version of TensorFlow 1.0 in November 2015 (Aurélien, 2017). At that time, TensorFlow was often seen by beginners and experts alike as exhausting code because TensorFlow logic code was vastly different from that of other libraries. At the same time, there were many high-level packages such as Pytorch and Keras that became more popular than TensorFlow. TensorFlow and Keras were both open-source libraries for artificial intelligence applications in 2017. Later, Keras was integrated into TensorFlow, and even after this integration, TensorFlow was losing popularity. After that, Google released the version of TensorFlow 2.0 on September 30th, 2019, and it has been the point of focus for the machine learning and data science

experts since then. TensorFlow quickly became the most popular open-source machine learning library worldwide (Singh et al., 2020) (TensorFlow.org, 2015).

TensorFlow 2.0 is an updated version of TensorFlow that has been released with a focus on ease of use, simple syntax writing, and higher productivity for developers. The development of deep learning applications became easier with TensorFlow 2.0 because of the updated features such as tight integration of Keras, default eager execution, and Python function execution to make the working of developing applications more familiar to python users (Silaparasetty, 2020). Also, the TensorFlow team modified the Application Programming Interface (API) for the TensorFlow 2.0. All operations that are used internally can be exported, such as variables and checkpoints. TF1.0 is demanded to build manually an abstract syntax tree (the graph) together by making (tf.*) calls. Then manually compile the graph by passing a set of output tensors and input tensors to a session.run call. TF2 executes eagerly which means it can immediately execute Tensors and Operations. TF2 makes graphs and sessions feel like implementation details. There is no need to do a session.run call in TensorFlow 2.0 because it is deleting all graphs - sessions code and writing TensorFlow code like a simple Python code. All these made it easy for users to get started with TensorFlow. The experts and even the beginners can execute the deep learning models with this latest version of TensorFlow (Singh et al., 2020) (TensorFlow.org, 2015).

An example of the difference in code syntax between TensorFlow 1.0 and TensorFlow 2.0 is shown below in Figure (1-3).

TensorFlow 1.0 code	Simplified TensorFlow 2.0 code

# Graph generation 	
tf_a = tf.placeholder(dtype=tf.float32)
tf_b = tf.placeholder(dtype=tf.float32)
tf_c = tf.add(tf_a, tf.math.multiply(tf_b, 2.0))
 	
# Execution 	
with tf.Session() as sess:
  c = sess.run(tf_c, feed_dict={tf_a: 5.0, tf_b: 2.0})
  print(c)
	a = tf.constant(5.0)
	b = tf.constant(3.0)
	c = tf_a + (tf_b * 2.0)
	print(c.numpy())

Figure 1-3 An example of the difference in code syntax between TensorFlow 1.0 and TensorFlow 2.0 (stackoverflow.com, 2020).

This thesis investigates the performance of a deep learning model on different hardware configurations. The model was a classification model for a big dataset of images. The interface TensorFlow was used to build the model and to distribute the model to run it in parallel on a specific hardware. Then, there is a discussion of the results of the running time for the parallelized model for the various hardware tested. After the discussion of the results, the comparison is done for running the model on Multi GPUs and Multi CPUs with TensorFlow and a large dataset. The goal of the comparison is to find the best and fastest hardware to run deep learning or machine learning models, especially when a large dataset is used with the model. The work is organized as an introductory chapter and then four additional chapters. Chapter Two discusses the related work for this thesis and the contribution of this work. Chapter Three talks about the details of the important steps for this work and the experimental setup of hardwares and softwares that have been used to accomplish the work. Chapter Four shows the results that were gotten from finishing the experiments. The results consist of plots and tables for recorded running time and for the calculated speedup to measure the performance. Chapter Five includes the conclusions and future recommendations.

CHAPTER TWO: RELATED WORK

2.1 Parallel Implementation of Neural Network Models on GPUS:

Ryan J. Meuth, Donald C. Wunsch II University of Missouri-Rolla showed that the performance of implementations of neural network algorithms could be higher or lower on graphics processing units as compared to a CPU. They showed that it is important to know the strengths and limitations of the GPU and to develop algorithms targeted to the GPU in a way that exploits GPU capabilities (Meuth et al., 2007).

2.2 Parallel Computing Using TensorFlow for Artificial Intelligence Models on GPU:

Adie et al. showed that using TensorFlow with GPUs could speed up the running time of the image processing models. Also, using TensorFlow could simplify the code significantly because of standard libraries that handle typical operations. Using TensorFlow to build a model could also simplify migration between CPUs or GPUs since the same code can be compiled for either. (Adie, 2018).

2.3 The Evaluation of The TensorFlow Model on GPU And CPU.

K V Sai Sundar et al. showed a comparison of training time by using GPUs and CPUs to evaluate the training time of the TensorFlow model. They noticed higher performance with an increasing number of iterations and/or deeper networks for GPUs vs. CPUs. Although sometimes it is possible to use an existing partially trained model, at other times, the training begins from scratch. K V Sai Sundar et al. noticed that because of the high number of training iterations required when training from scratch, it could be important to use GPUs to train the model from scratch because the difference in speed can have a large impact on training time. Using the GPUs gave better

performance when completing the computations for these numbers of iteration. (Sundar et al., 2018).

The following figure (Figure 2.1) shows the procedure of the computation of deep neural networks to find the output classification by extracting features from inputs to arrive at probabilities for various classes through softmax function "The softmax function is used in various multiclass classification methods, such as multinomial logistic regression" (en.wikipedia.org, 2021) or any other classifier functions " The activation function is an integral part of a neural network. The activation function is used in the final layer of a neural network-based classifier. Without an activation function, a neural network is a simple linear regression model. This means the activation function gives non-linearity to the neural network." (Shipra Saxena, 2021).

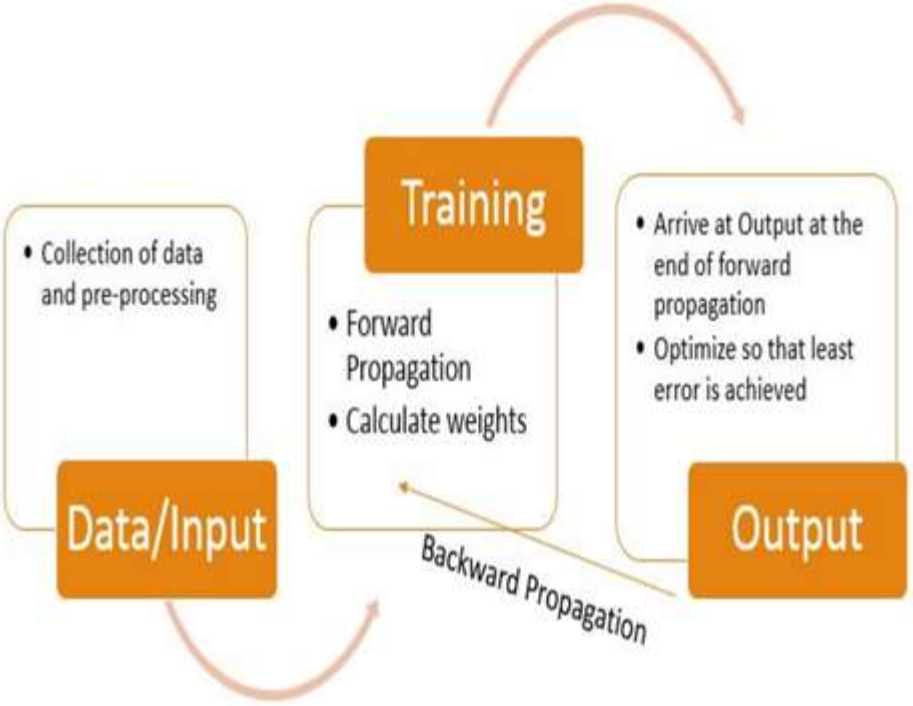

Figure 2-1 The procedure of computation of neural networks (Sundar et al., 2018).

2.4 The Performance Comparison of TensorFlow when Using CPU And GPU:

Eric Lind and Ävelin Patnigoso compared the performance of TensorFlow when using CPU and GPU. They noticed that the performance of TensorFlow depends significantly on the CPU for a small-size dataset. Also, they found it is more important to use a graphic processing unit (GPU) when training a large-size dataset. Furthermore, the CPU and GPU comparison results were enough to show that when a complex neural network is being trained with TensorFlow, the performance is enhanced by using GPU parallelizing power. (Lind et al., 2019).

2.5 Clustering Computation of A Big Dataset Using Multi CPUs And Multi GPUs:

Widiarto Adiyoso et al. investigated the clustering of big data using multi-CPUs and multi-GPUs. They analyzed the performance when using Spark as a baseline for multi-CPU big data clustering and TensorFlow as a baseline for multi-GPU big data clustering. Spark is a framework that is used for distributing the processing of the models. Spark uses the MapReduce framework to solve problems related to big data and its processing (MA Raza, 2020). The experiment showed that TensorFlow performs in a range of five to twelve times faster in computation than Spark. TensorFlow supports CPU and GPU computation. Ideally, a high-end GPU is required to fully utilize the performance of TensorFlow. So the recommended implementation of TensorFlow with a high-end GPU contained a high-performance computing (HPC) node. (Adiyoso et al., 2018).

The experiment showed that the bigger the cluster, the greater performance speedup can be achieved by TensorFlow so that it can be a good package for big dataset computation research.

The following figure (Figure 2.2) shows the three important steps in the framework for big data management. The three steps are: obtain the dataset from sources, cluster the dataset using the clustering methods, and lastly having the clusters as the application of the processed dataset.

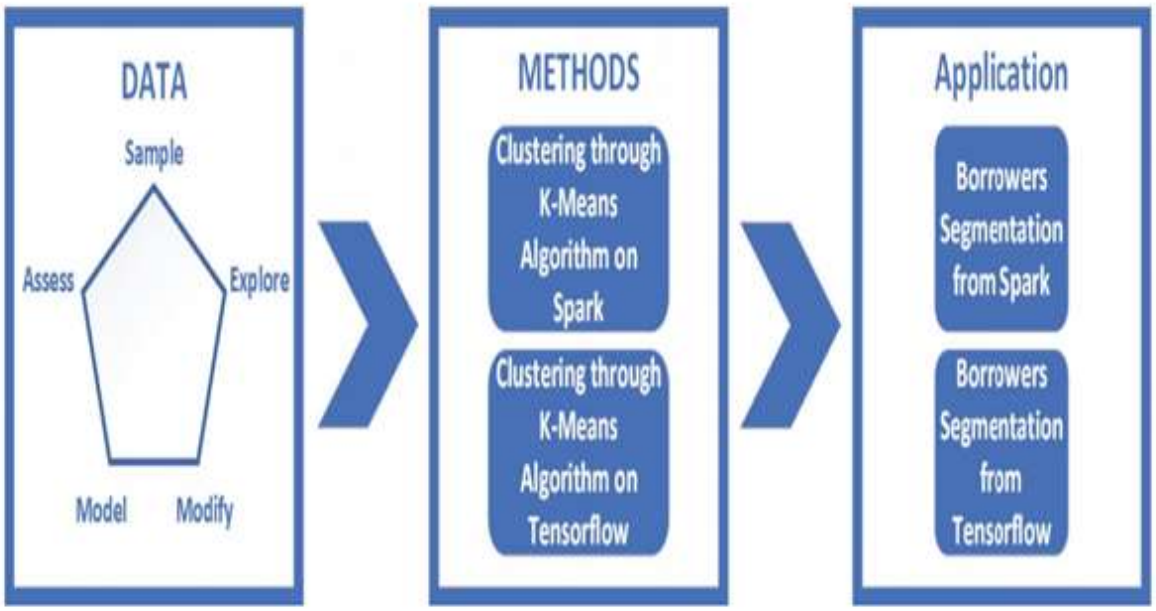

Figure 2-2 Big data clustering by using Spark method for (CPU-cluster) and TensorFlow method for (GPU-cluster) (Adiyoso et al., 2018).

2.6 Contribution of This Thesis:

This work was inspired by the previous work mentioned. It builds on the previous work in several ways. A goal of this work is to do a comparison between running the machine learning model for a big dataset by using TensorFlow distribution for parallelizing the computation on GPU clusters vs. TensorFlow multi-workers for parallelizing the computation on CPU clusters. This work also uses TensorFlow 2.0, which is faster and easier with coding than TensorFlow 1.0.

CHAPTER THREE TENSORFLOW FOR MULTI GPUS AND MULTI CPUS

3.1 Introduction

Machine learning and deep learning architectures often have millions of parameters to be learned during the training. When training a model from scratch on a specific dataset, one needs to look at the performance of the architectures on this dataset (Sharma, 2014). It is also important to analyze the hardware conditions that affect the performance of the Deep Learning computation (Lopes, 2010).

This work focuses on three things:

3.1.1 Model for training on the large dataset:

Training on a large dataset takes a very long time to achieve a high percentage of accuracy. Training may take weeks of computer time, especially when using older versions of the TensorFlow and Keras packages. This work focused on making a comparison between a distributed training model on CPU versus GPU. This model classifies an image dataset by extracting the features (kaggle.com/shayanriyaz, 2019), learning about various classes using the Conv2D model for training, and giving the probability of the outputs through the SoftMax function (Sundar et al.,2018). The mini-batch size was used for the distribution of the data (Wang, 2020). Strategy scoop provided by TensorFlow was used for the distribution of the model. Strategy scoop is part of TensorFlow 2.0 Distribution strategies. It helps distribute training across multiple GPUs or multiple machines with minimal code changes (TensorFlow.org, 2015).

Figure 3.1 represents the model distribution of the modern Deep Learning systems that train deep neural networks and the data distribution, which also can be executed in parallel. They both can be suitable for multi-threaded execution on CPUs and multi-stream models on GPUs (Wang, 2020).

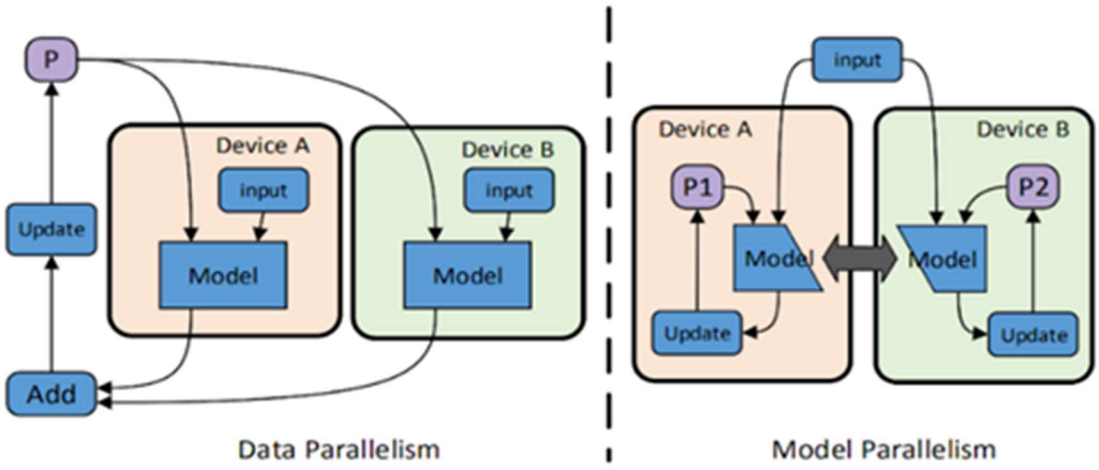

Figure 3-1 Data distribution and model distribution (Wang, 2020).

3.1.2 CPU vs. GPU:

TensorFlow was the framework used to distribute the implementation of the classification model with a few changes for the code (Abadi M, 2016).

The model was run in parallel on multiple cores or multiple GPU devices. Thus, the runs depended on how many cores or GPU devices were requested from the clusters and they would be suitable for parallel implementation (Adie, 2018).

The multiple CPUs would make the training faster, but there is one constraint when the training happens on multiple GPUs. Choosing the batch size of the training data depends on the memory size of the GPUs. So, the memory size of the GPUs should be considered before training the model (Sundar et al., 2018).

3.1.3 The framework TensorFlow:

This work uses TensorFlow libraries for deep learning to train the model and for TensorFlow distribution to achieve parallel model computation. TensorFlow is a deep learning platform that Google developed, and the Python programming language supports it. It is an open-source library, commonly used in deep learning implementations, and supports distributed computations on CPU and GPU clusters. It also can do the automatic parallel computation code for the easy way of multiple threading (Geron Aurelien, 2017).

In order to use GPU processing, TensorFlow needs the CUDA drivers to be installed. Other software requirements to be provided include the CUDA toolkit, CUPTI, and cuDNN SDK so TensorFlow can run on single or multiple GPUs (TensorFlow.org, 2015).

3.2 Experimental Setup

Computational resources at the University of Vermont were used for the runs described in this thesis. It used a cluster called DeepGreen for the GPU runs, and a cluster called Bluemoon for the CPU runs. A necessary first step was creating an account for these systems without being a student or faculty member at the University of Vermont. We worked closely with staff at the University of Vermont Advanced Computing Core to work through these issues.

3.2.1 Hardware:

The hardware used in this project consists of a laptop connected to the Internet. This is used to login into two types of clusters. The first one is called DeepGreen see Figure 3.2, which is a highly parallel cluster commissioned in Summer 2019, that is the most recent artificial intelligence supercomputer in the University of Vermont with 80 GPUs capable of over eight petaflops of mixed-precision calculations based on the NVIDIA Tesla V100 architecture. Its hybrid design can accelerate high throughput artificial intelligence and machine learning computations, and its widespread parallelism can model new and transformative research pipelines (VACC, 2019). The hardware components of this cluster are:

1. 10 GPU nodes (Penguin Relion XE4118GTS) each with:
    - 2 Intel(R) Xeon (R) Gold 6130 CPU @ 2.10GHz (2x 16 cores, 22M cache).
    - 768GB RAM (256GB for GPFS pagepool).
    - 8 NVIDIA Tesla V100s with 32GB RAM.
    - 4 2-lane HDR (100Gb/s, so 400Gb/s/node) Infiniband links to QM8700 switch.
    - NVMe nodes, each with 64TB NVMe devices (8x8TB), replicated to provide a 64TB / gpfs3 filesystem.
2. Mellanox QM8700 switch running at HDR speed.

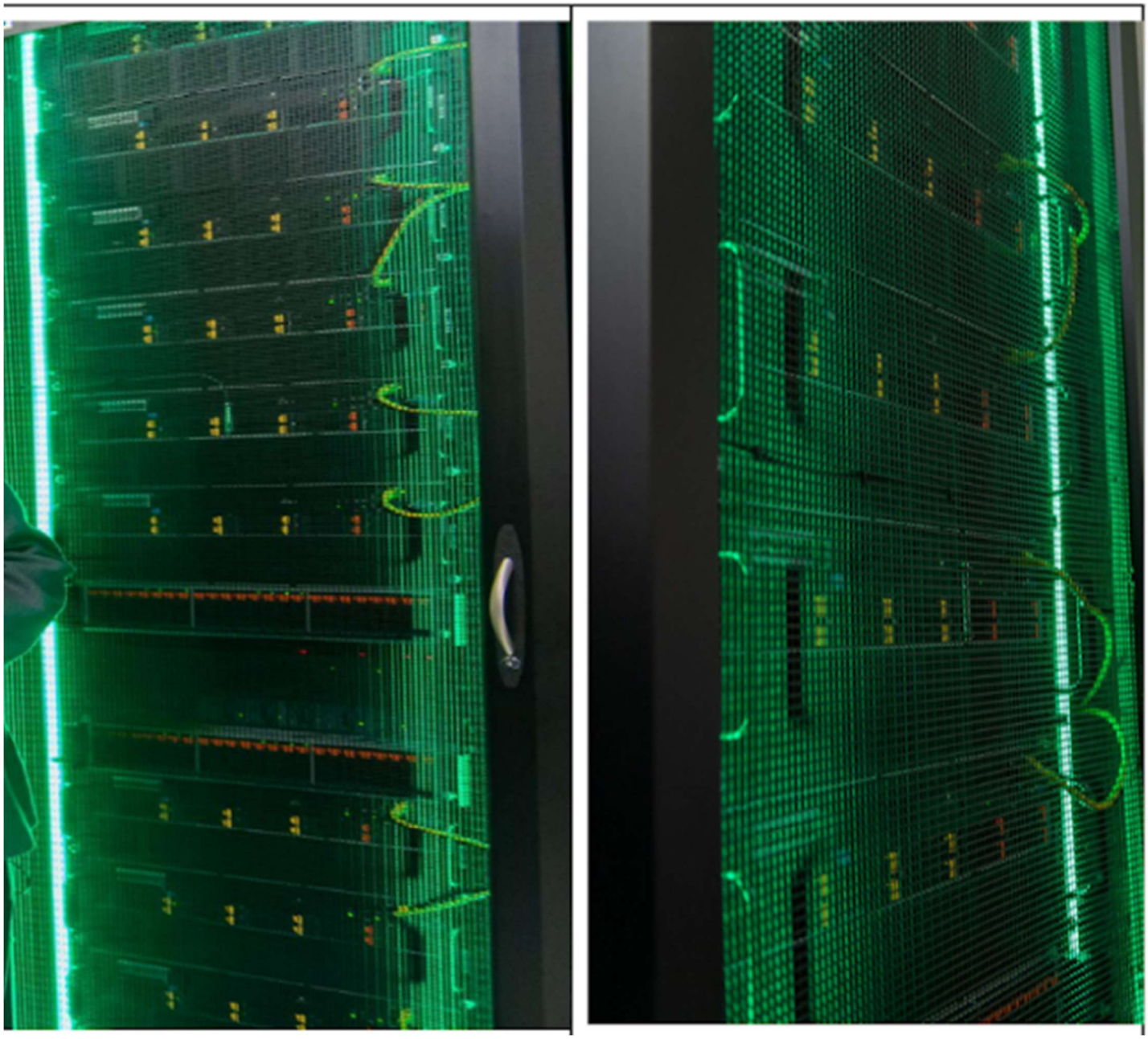

Figure 3-2 DeepGreen cluster

Additionally, this research uses a second cluster called BlueMoon see figure 3.3, which is a 300 node, 4004 core, high-performance computing system modeled after national supercomputing centers (VACC, 2019); it supports large-scale computation, large memory

systems, low-latency networking for MPI loads, and high-performance for parallel files (VACC, 2019). This cluster is also in the University of Vermont for parallel computation. The hardware components of this cluster are:

1. Thirty-two dual-processor, 12-core (Intel E5-2650 v4) Dell PowerEdge R430 nodes.

2. Eight dual-processor, 12-core (Intel E5-2650 v4) Dell PowerEdge R430 nodes.

3. Nine dual-processors, 20 cores (Intel 6230), PowerEdge R440.

4. Three dual-processor, 10-core (Intel E5-2650 v3) Dell PowerEdge R630 nodes.

5. 130 dual-processor, 6-core (Intel X5650) IBM dx360m3 nodes.

6. Infiniband: 8 dual-processor, 160-core (Intel E5-2650 v3) Dell PowerEdge R630 nodes.

7. Infiniband: 32 dual-processor, 640-core (Intel E5-2650 v3) Dell PowerEdge R630 nodes.

8. Infiniband: 22 dual-processor, 252-core (Intel E5-2630) IBM dx360m4 nodes.

9. 2 dual-processor, 12-core (Intel E5-2650 v4) Dell R730, with 1TB

10. 1 dual-processor, 8-core (Intel E7-8837) IBM x3690 x5, with 512GB

11. 2 dual-processor, 12-core (Intel E5-2650 v4) Dell R730 GPU nodes, each with 2 NVidia Tesla P100 GPUs. (Each GPU has 3584 CUDA cores and 16GB RAM).

12. 2 Flash-storage GPFS Metadata nodes (IBM x3655s, 10G Ethernet connected).

13. 2 I/O nodes (Dell R430s, 10G ethernet connected) along with 2 I/O nodes (IBM x3655s, 10G ethernet connected) connected to:

- 1 IBM DS4800 providing 260 terabytes of raw storage to GPFS (roughly 197TB usable)
- 1 IBM DS4700 providing 104 terabytes of raw storage (roughly 76TB usable)
- 1 IBM DCS3850 providing 240 terabytes of raw storage to GPFS (roughly 164TB usable)
- 1 Dell MD3460 providing 357.5 terabytes of raw storage to GPFS (roughly 260.5TB usable), and 43 terabytes of the solid-state disk to GPFS (for fast random-access data and metadata, roughly 27.5 TB usable)
- 1 IBM V3700 providing ten terabytes of the solid-state disk to GPFS (for fast random-access data and metadata)

Figure 3-3 BlueMoon cluster

3.2.2 Software:

The clusters have the following software:

3.2.2.1 DeepGreen Software:

- Operating System: RedHat Enterprise Linux 7 (64-bit) with the GNU compilers (gcc, f77).
- Resources Manager: Slurm v19.
- Package Manager: Spack v0.11

3.2.2.2 Bluemoon Software:

- Operating System: RedHat Enterprise Linux 7 (64-bit) with the GNU compilers (gcc, f77).
- Resources Manager: Slurm v20.
- Package Manager: Spack v0.11.

3.2.2.3 MiniConda:

Miniconda is a free minimal installer for Conda. It is a small, bootstrap version of Anaconda that includes Conda. Miniconda has the core Python language, a package manager tool (Conda), and the packages the Conda and Python depend on (Anaconda.com, 2017). "Anaconda is an open-source python distribution. It is purpose-built for such applications as machine learning, data science, and large-scale data processing" (linuxnetmag.com, 2021). Conda is an environment manager and installer for the Conda packages on the platform (linuxnetmag.com, 2021). Miniconda offers all the benefits that can be obtained, like using the Anaconda with minimal space requirements. We downloaded the Linux 64 version with Python 3.9 on a personal laptop that has the Windows operating system; then, it was moved to account on the cluster by using the WinScp application (winscp.net, 2014). After MiniConda was moved to the directory, it got installed on the cluster, which is working with Linux.

The MiniConda program can read and use the Python files that have the extension (.py). However, MiniConda cannot read or use the Python files that have an extension (.ipynb). This means that it can only read Python files, not the Jupyter NoteBook Python files. This is another difference between the MiniConda and Anaconda.

3.2.2.4 Use SSH client to connect to the cluster:

To be able to log in to the cluster from the Windows computer, the SSH client application terminal was needed. The SSH is used to write the commands for login into the cluster see figure (3.5). After logging in, the command line continued in the same terminal by using a text editor. Nano text editor was used for writing the Python files and also for writing the batch files. The batch files are Slurm scripts for submitting the job to be running in the cluster (VACC, 2019).

3.2.2.5 Install (WinSCP):

The WinSCP application is installed on a computer with Windows operating system. WinSCP requires the same login credentials as the cluster. WinSCP is used to move files from the local computer to the remote supercomputer's directory see Figure (3.4) (winscp.net, 2014). These files are important for this work to be moved from and into the cluster. These files were moved from the computer into the clusters: the MiniConda and the dataset are used on the cluster. Finally, the data and results from this work were moved from the clusters into the local laptop computer.

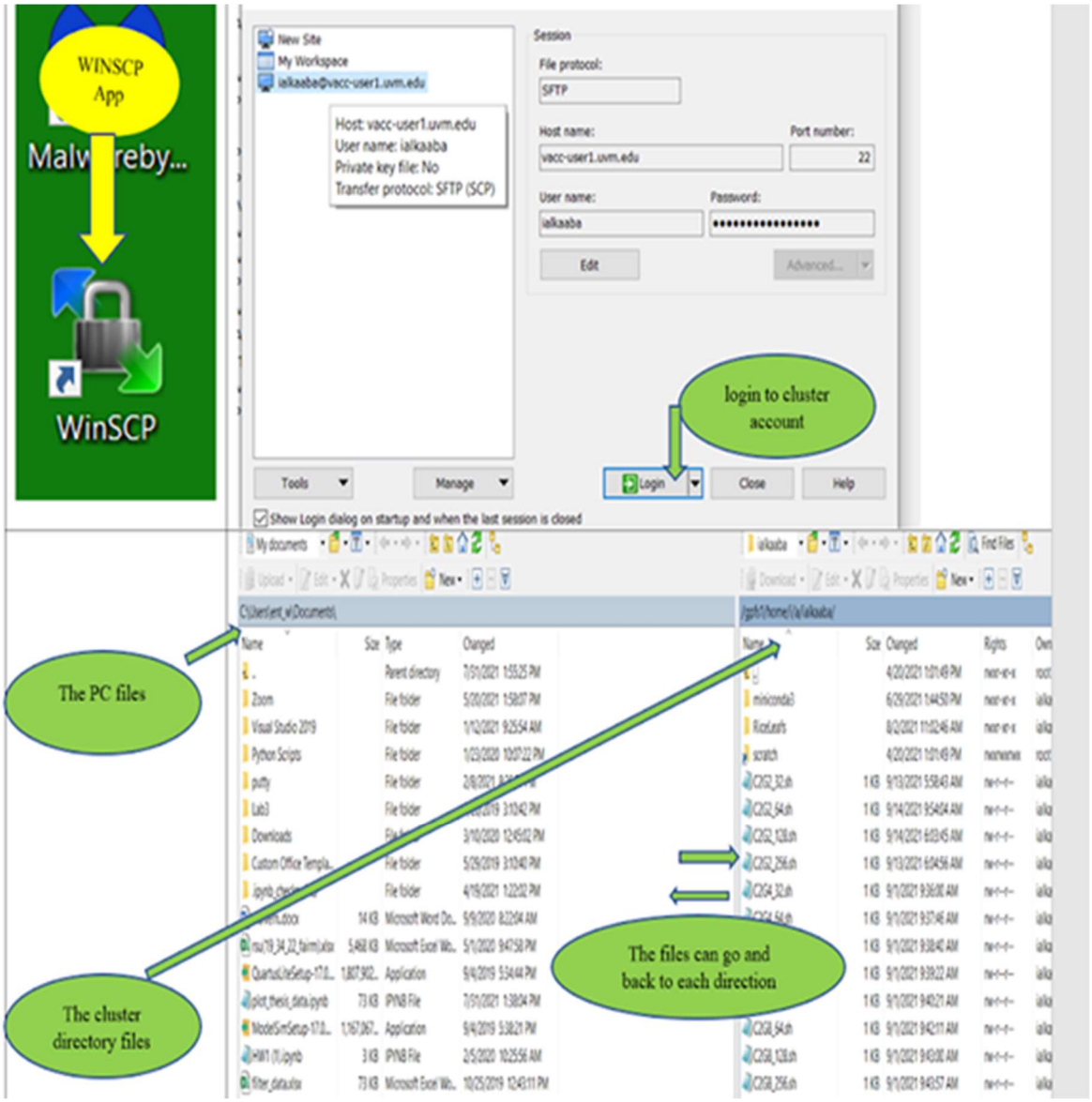

Figure 3-4 Using the WinScp application to move files forth and back between the local computer to the remote supercomputer's directory

3.2.2.6 Create three Environments of TensorFlow:

When the MiniConda software package was installed on the cluster directory, it had the base environment that pointed to the original Python software package. It was necessary to create two additional TensorFlow environments. One environment is the TensorFlow environment to do the computation on the CPU cluster. The other environment is the TensorFlow-GPU environment to do the computation on the GPU cluster.

All three environments were needed for the packages and Python libraries to be installed before running any job.

The following figure (Figure 3.5) shows a screenshot of the SSH terminal and how to use the University of Vermont account to log in to the clusters; first, the login has to be to BlueMoon (CPU-cluster) by using the command contains Vermont account, and all three environments could be used in this cluster by activating any one of them to work with. The login to DeepGreen (GPU_cluster) can happen after the log into BlueMoon (CPU-cluster) by using different command contains the name of the cluster and directory name that is used in the BlueMoon (CPU-cluster). After this, any TensorFlow environment can be activated.

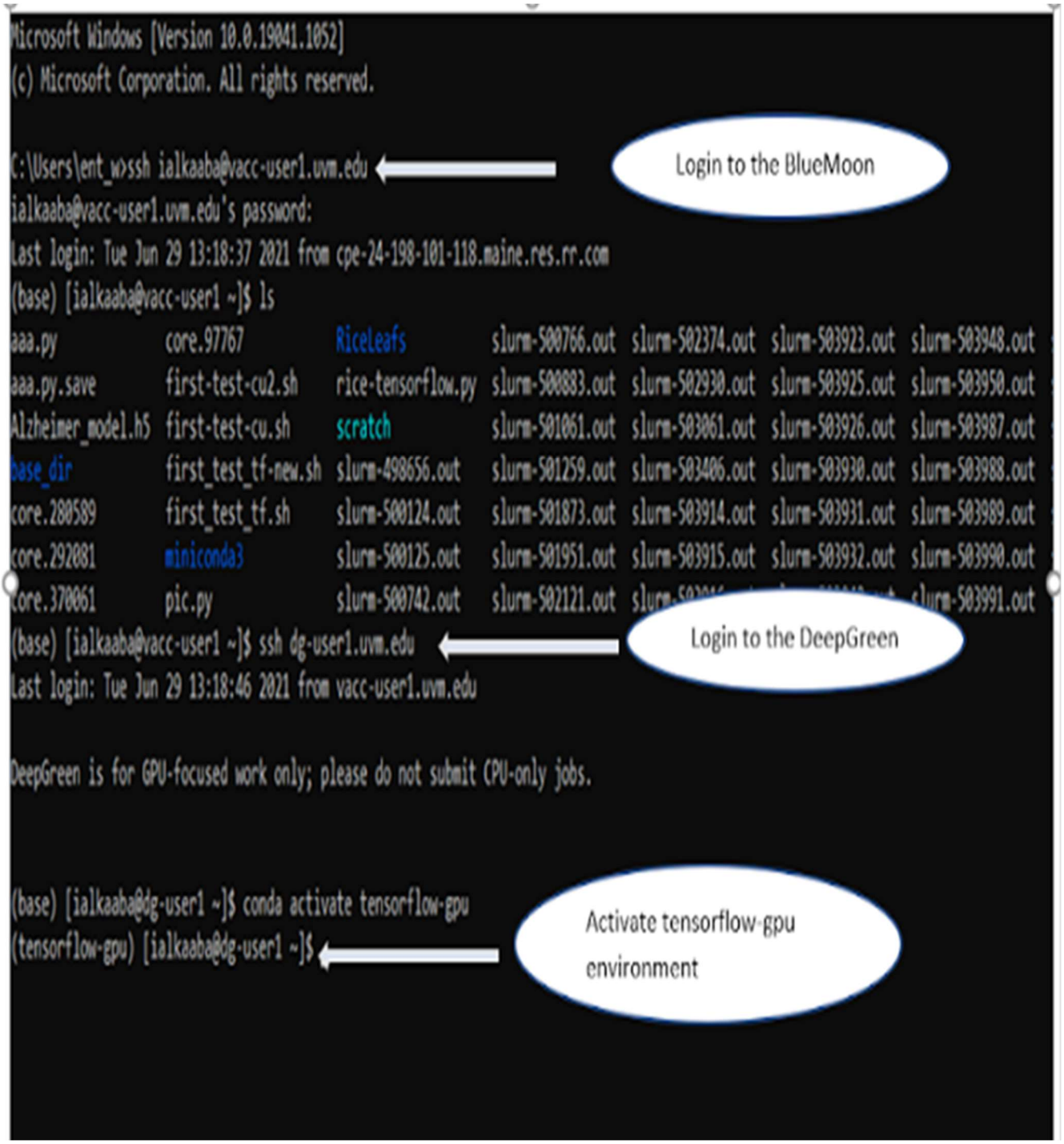

Figure 3-5 Using the SSH application terminal to log in the cluster (CPU-cluster) and (GPU-cluster) and shows how to change TensorFlow environments

3.2.2.7 Using a Slurm script to submit a job:

The batch system which is used in both clusters is Slurm (VACC, 2019). Jobs are submitted by running Slurm scripts that specify the necessary parameters for the system. It is a free and open-source scheduler to submit the jobs that need to be run on clusters. The job script can be created using any text editor, and for this work, the Nano editor was used for writing the Slurm commands. The commands used in each cluster are mostly the same, but there are a few differences in requesting the resources for running the jobs; the Slurm script for running jobs on DeepGreen (GPU) cluster requires to request the numbers of GPU and requires adding the CUDA path to this script. These numbers of GPUs can be used to parallelize the computation of the job and the CUDA path to enable the computation to be performed by GPUs.

Slurm script for GPU_cluster (DeepGreen)	Slurm script for CPU_cluster (BlueMoon)
#!/bin/bash #SBATCH --partition=dggpu # Request nodes #SBATCH --nodes=1 # Request some processor cores #SBATCH --ntasks=4 #SBATCH --mem=12G #SBATCH --time=01:00 # use YOUR job name #SBATCH --job-name=first_test_tf # use YOUR email address #SBATCH --mail-user=intisar.alkaabawi@maine.edu #SBATCH --mail-type=ALL # loads CUDA export PATH=${PATH}:/gpfs3/arch/x86_64-rhel7/cuda-10.0/bin export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/gpfs3/arch/x86_64-rhel7/cuda-10.0/lib64 # provide YOUR python filename python test2.py	#!/bin/bash #SBATCH --partition=bluemoon #SBATCH --nodes=1 #SBATCH --ntasks=2 #SBATCH --mem=12G #SBATCH --time=01:00 # use YOUR job name #SBATCH --job-name=job_name # use YOUR email address #SBATCH --mail-user=ktoksu@uvm.edu #SBATCH --mail-type=ALL # provide YOUR python filename python test2.py

Figure 3-6 Sample slurm script

3.2.2.8 Download the dataset and move it to the cluster:

The dataset used in this thesis is Rice Leaf's dataset. The dataset was found on the Kaggle website. The size of Rice Leaf's dataset is 7.49 GB of images; the dataset has two directories called train directory and validation directory. Inside each directory are four categories they are BrownSpot,

Healthy, Hispa, and LeafBlast these names denote that the rice leaf either is healthy or has one of the other three diseases. The total images for the training set are 2684 files, and the total images for the validation set are 671 files. The machine learning model for this dataset is doing the classification model for the images of the rice leaves. The output for this classification is one of these four categories. Figure 3.7 shows some representative images from the dataset.

First, the dataset was downloaded to the Windows computer and extracted. The time for download and extract over one hour. Then the dataset was moved to the cluster directory by using the WinScp application so it could be used in the model to do the classification. The time for moving the data to the cluster directory took almost two hours.

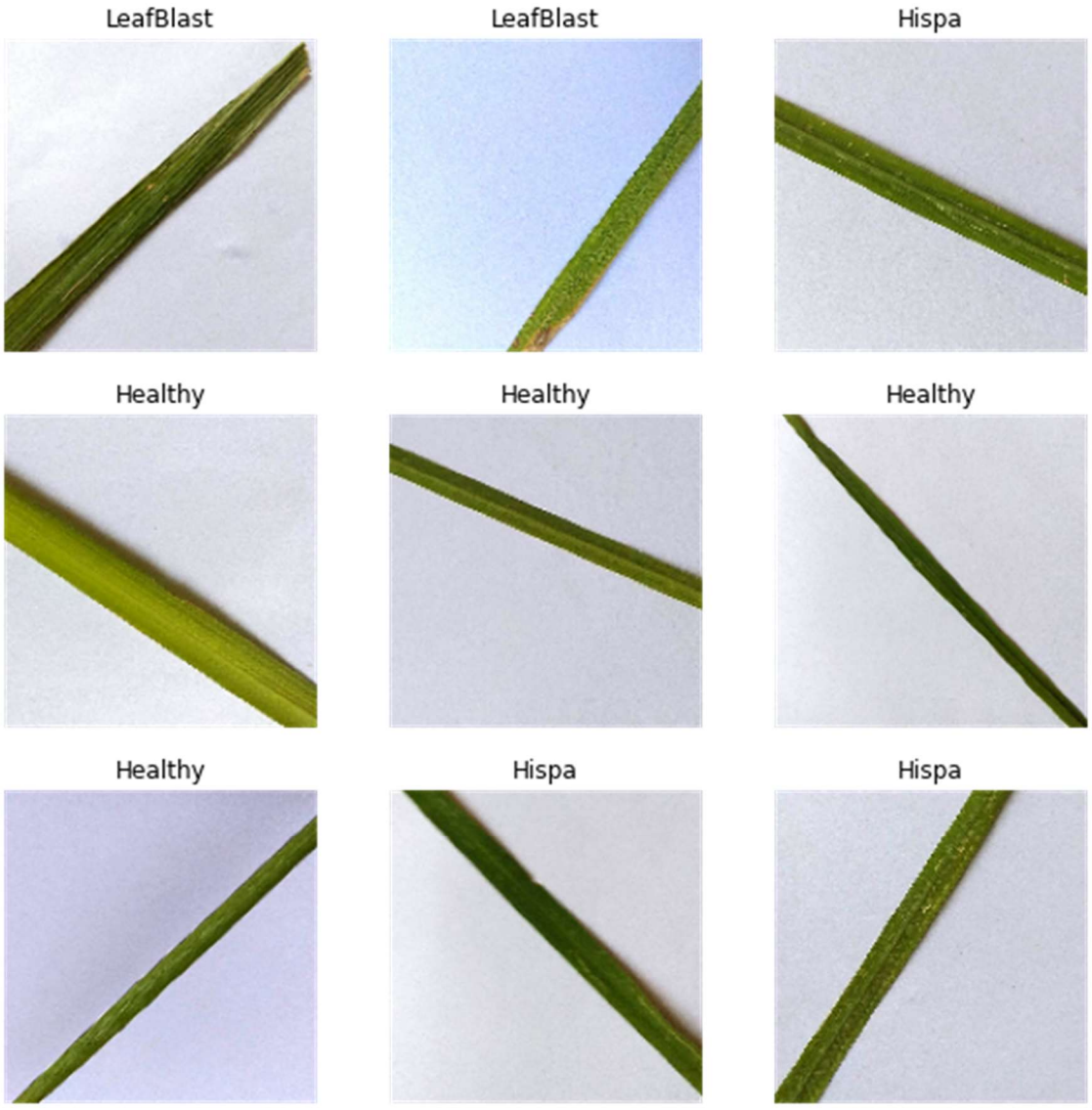

Figure 3-7 The Rice Leaves classification for four categories

3.2.2.9 Download the Anaconda and Jupyter Notebook on a PC:

A personal computer was used to download the Anaconda software and TensorFlow environment. Jupyter NoteBook in this environment was used for plotting the data that was gotten from training the model and viewing the images of the dataset. Using this environment was necessary because the cluster has a terminal interface, and it does not show any images.

3.3 Conclusion

All hardware and software mentioned above were used to achieve the work. The results for the run time were recorded by the clusters when the slurm scripts were submitted to run the code. The results were sent to the email address that was added to the slurm scripts. After collecting these results, they were saved in excel sheets so they could be used for plotting or calculating the speedup. The other results of running the model such as the accuracy function and loss function for both training and test datasets.

The loss function is a measurement of the difference between the actual output and the predicted output. The loss value is a number indicating how bad or good the model's prediction was on the trained example. The model's prediction is good as long as the loss is small; otherwise, the model's prediction is worse the larger the loss (developers.google.com, 2020). The accuracy and loss functions for both training and test datasets were obtained through the epochs of training the model and were saved in a csv file. One epoch has been completed when the algorithm has seen all samples in the dataset (androidkt.com, 2021).

The csv files were stored in the directory of the user's account on the cluster, and the files can be moved from the directory into the laptop by using the WinScp application. When the csv files were moved to the laptop, the Jupyter Notebook was used to run a python code to show plots of the accuracy and loss functions for training and test datasets.

CHAPTER FOUR: RESULTS

In this chapter, results from running the neural network models on the clusters are presented. These results include the time to run the model when it was running on different numbers of processors for a different number of batch sizes for the image dataset, and the time for implement the model when it was running on different numbers of processors with different numbers of Graphics Processing Units for the same batch sizes for the image dataset. Batch size is a term used in machine learning and refers to the number of samples that will be passed through to the network at one time (androidkt.com, 2021).

The results are presented in tables of running times, tables of speedup results, plots of time comparisons and speedup, and plots for the model classification results, which are the accuracy and loss values of training dataset and validation dataset.

4.1 Execution time on multiple CPUs with multiple batch sizes:

In this section, the execution time results are presented. The execution time is the time measured by using TensorFlow multi worker distribution for training the neural network model. The results consist of plots of the comparison of execution time on different numbers of CPUs. For each number of CPUs (two, four, and eight), the results contain running the model with a different number of batch sizes of the dataset. Time is measured in seconds in all plots.

Figure 4-1The Execution Time of multiple CPUs for four batch sizes

Figure 4.1 above shows the time for running the neural network model on the CPUs cluster. The time is measured in seconds. Four batch sizes were used to run the model with three different numbers of CPUs on the cluster, two, four, and eight. One can see that each time the number of CPUs increases, the performance time is faster. The performance is a little faster for the same batch size when the CPUs number increases. But when the batch size is large, like 256, the performance is a lot faster when the CPUs number is increased. When the batch size is too large, it may not fit in the memory of the computer request used for the training.

4.2 Execution time on MULTIPLE CPUs with multiple GPUs for the same batch sizes:

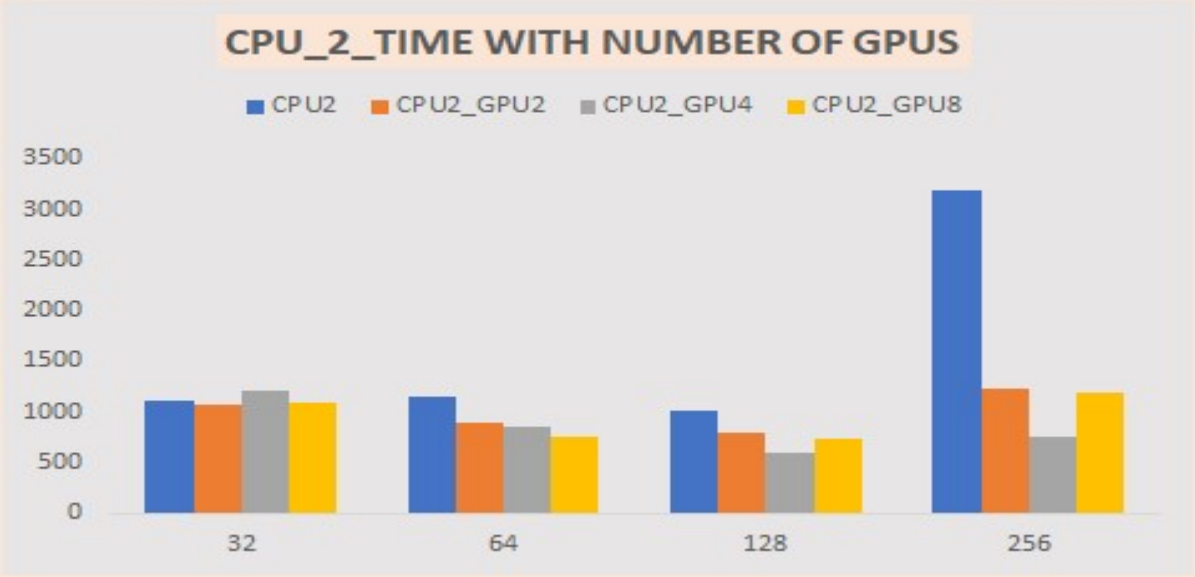

Figure 4-2 The Execution Time of two CPUs with multiple numbers of GPUs for four batch sizes

Figure 4.2 shows the time taken to run the same model while requesting two processors with three different numbers of GPUs on the cluster. For the 32-batch size, the performance was almost the same with or without requesting the GPUs. Each time the calculation of requesting more devices made the performance faster, but at the same time, the distribution time made the run time slower. So, when the batch size is small, the performance does not become so much faster because of the time spent on sharing the data on many devices. The performance is better when increasing the number of GPUs every time the batch size becomes larger because the batch size is too large to fit in the memory when the distribution happens on a few devices. For a large batch size, more memory location is needed to fit all the samples from the training dataset.

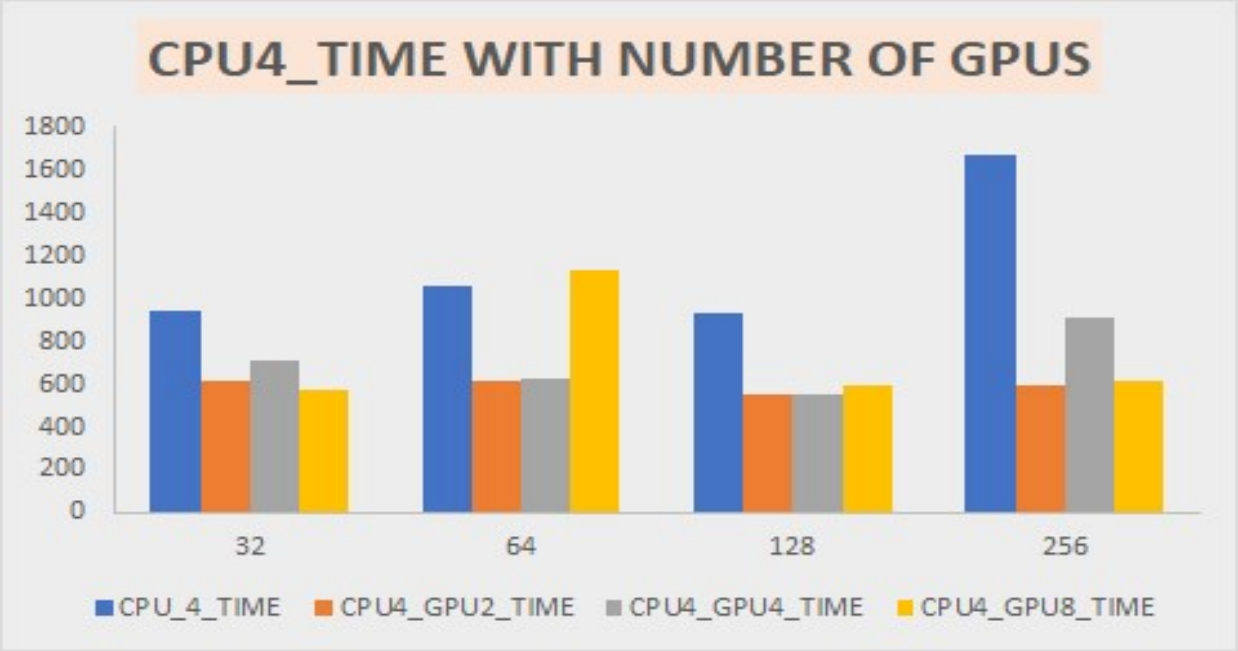

Figure 4-3 The Execution Time of four CPUs with multiple numbers of GPUs for four batch sizes

Figure 4.3 shows the time taken to run the model while requesting four processors with three different numbers of GPUs on the cluster. For all the batch sizes, the performance was faster when requesting the GPUs with four processors. We can see the time difference when running the model with just the four processors and when running the model with GPUs beside the four processors. But there is some slow performance when increasing the number of GPUs that were requested from the cluster. The slower performance is repeatable; that is, it was observed in multiple runs. It is not completely clear why the GPU4 times did not follow the general trend of the others.

The performance is better when increasing the number of GPUs every time the batch size becomes bigger because the bigger batch size needs more memory locations, so more devices mean more memory available.

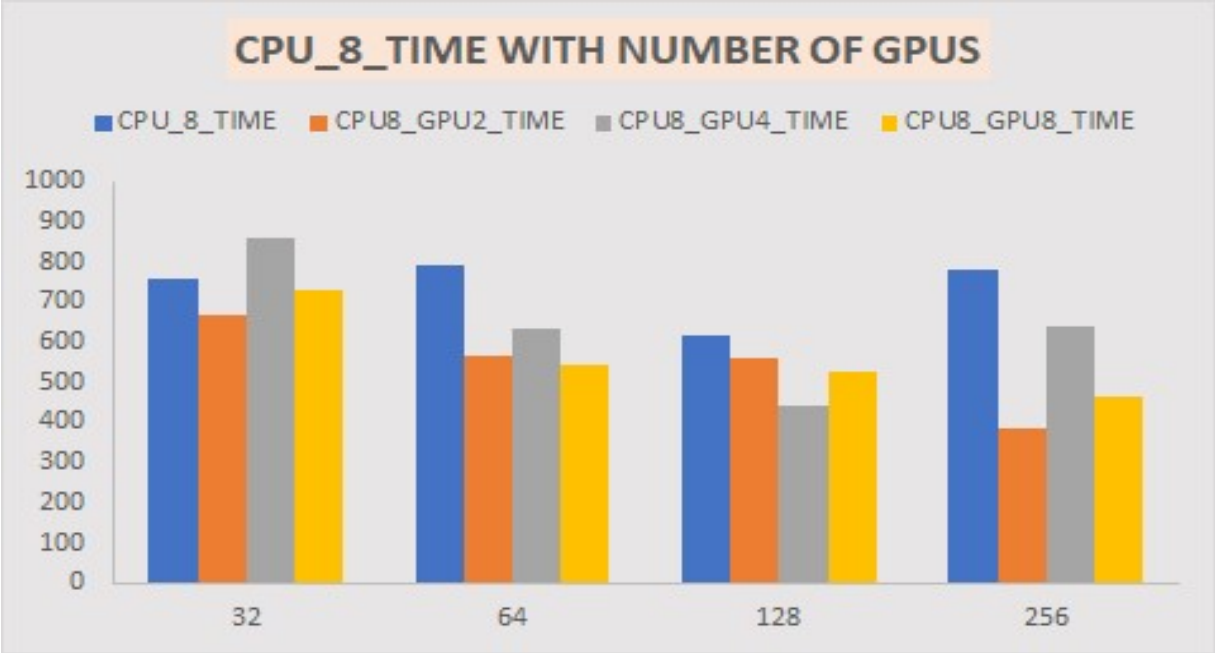

Figure 4-4 The Execution Time of eight CPUs with multiple numbers of GPUs for four batch sizes

Figure 4.4 shows the time spent to run the model when requesting eight processors. These eight processors were running the model each time, requesting a different number of GPUs on the cluster, two, four, and eight. Four different numbers of batch sizes were used to run the model with these three numbers of GPU and eight processors. In general, the performance was faster when requesting the GPUs with eight processors. The performance is unlike for all four batch sizes when increasing the number of GPUs. For the large batch size, more devices provide more memory to make all training samples fit at the same time to do a faster computation of the model distributed on many devices that work in parallel, so the computation time is faster than the time spent on the distribution and communication between the devices and the purpose of this work is to find a less run time to make the performance better. The bulk of the processing is done on the GPU devices,

and thus one would expect performance to improve as more GPUs are added. Also, the function of the CPU(s) in these calculations is to distribute data to the GPU devices and collect the results. Having multiple CPUs can make this process faster. In running this code, there was no ability to control which CPUs in a multi-core processor or which GPUs in a multi-GPU machine were used. This could account for some of the variability seen.

4.3 Speedup for multiple CPUs and CPUs with multiple GPUs for the same batch sizes:

This section contains plots of the speedup calculation for training the parallel model on different types and numbers of hardware. The speedup is calculated as the execution time when the model runs before increasing the number of devices divided by the execution time when the model runs after increasing the number of devices (Gelenbe, 1988). The speedup plots can help to make the comparison between the different hardware. The comparison leads to choosing which hardware would be faster to train the neural networks model.

Figure 4-5 The Speedup of the distribution of multiple CPUs for four batch sizes

Figure 4.5 shows the speedup of the model distribution on multiple CPUs for four different numbers of batch sizes. The speedup is a little higher for the small batch sizes when the CPUs number is increased but, when the batch size is big, like 256, the speedup is much higher when the CPUs number is increased for more distribution. Also, the speedup is higher every time the batch size is bigger when the running happens on the same number of processors and when fewer processors are requested to run the model. In another way, when the batch size is 256, or bigger and more processors are requested, the speedup becomes much higher because fewer processors have the lack the memory to compute a large number of samples at the same time on a few processors. The bigger batch size needs distribution on more processors for higher speedup.

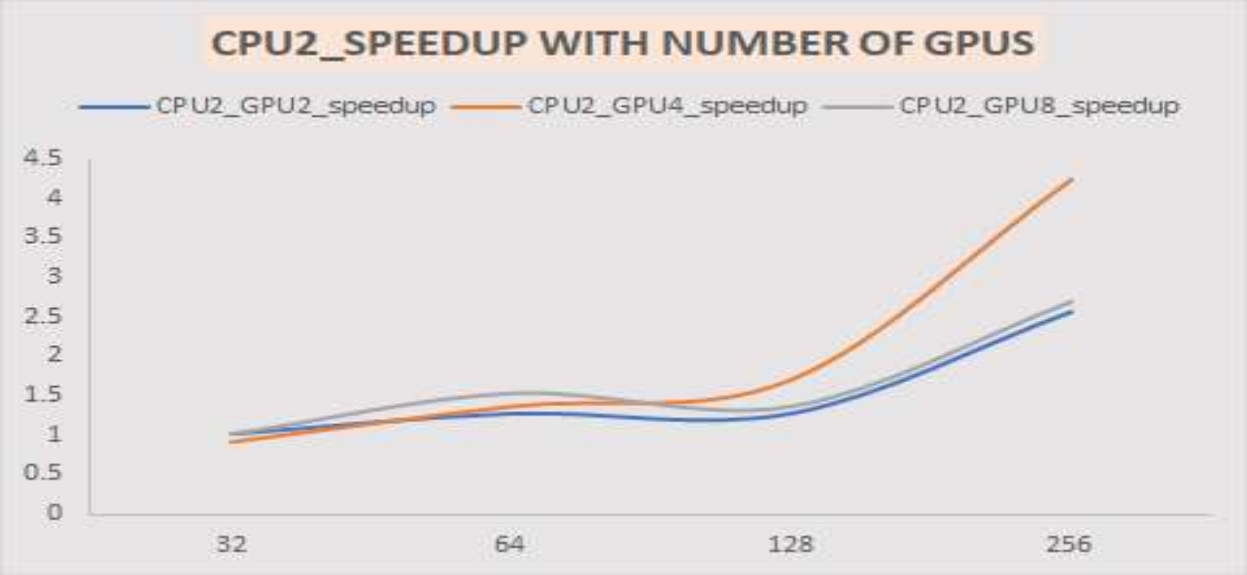

Figure 4-6 The Speedup of the distribution of two CPUs with multiple numbers of GPUs for four batch sizes

Figure 4.6 showed the speedup when the model was running on two CPUs then requesting three different numbers of GPUs for four batch sizes. We can see the results are close to the previous figure. The speedup is higher every time the batch size is bigger when the running happens on the same number of processors. Also, the speedup is generally higher when more GPUs are requested to run the model. Similarly, using a bigger batch size when requesting more GPUs to distribute the model generally leads to higher speedup.

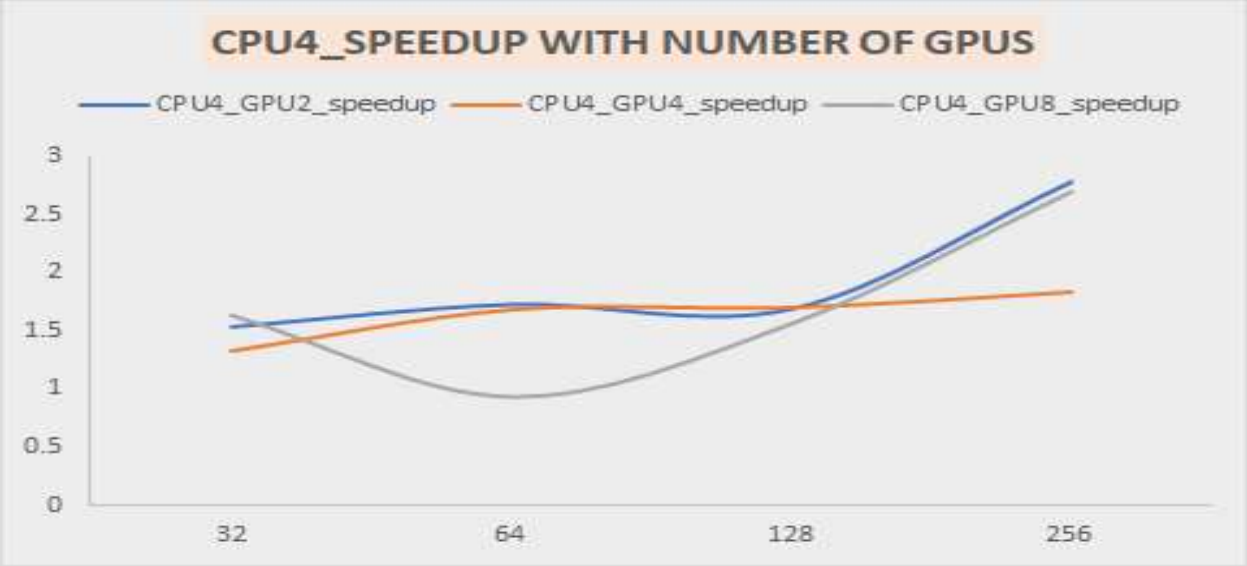

Figure 4-7 The Speedup of the distribution of four CPUs with multiple numbers of GPUs for four batch sizes

Figure 4.7 showed the speedup when the model was running on four CPUs, each run with a different number of GPUs, for four batch sizes. When four GPUs were requested, the speedup was a little higher and the running time with four GPUs was close to the running time without the four GPUs. On the other hand, When two and eight GPUs were requested the speedup was much higher than the speedup without requesting the GPUs. As was mentioned previously, the behavior when using four GPUs did not fit the expected trend. We can see the performance for running the model with requesting eight GPUs, and the batch size was 64 is the same performance for running the model without requesting eight GPUs. This was verified with multiple runs.

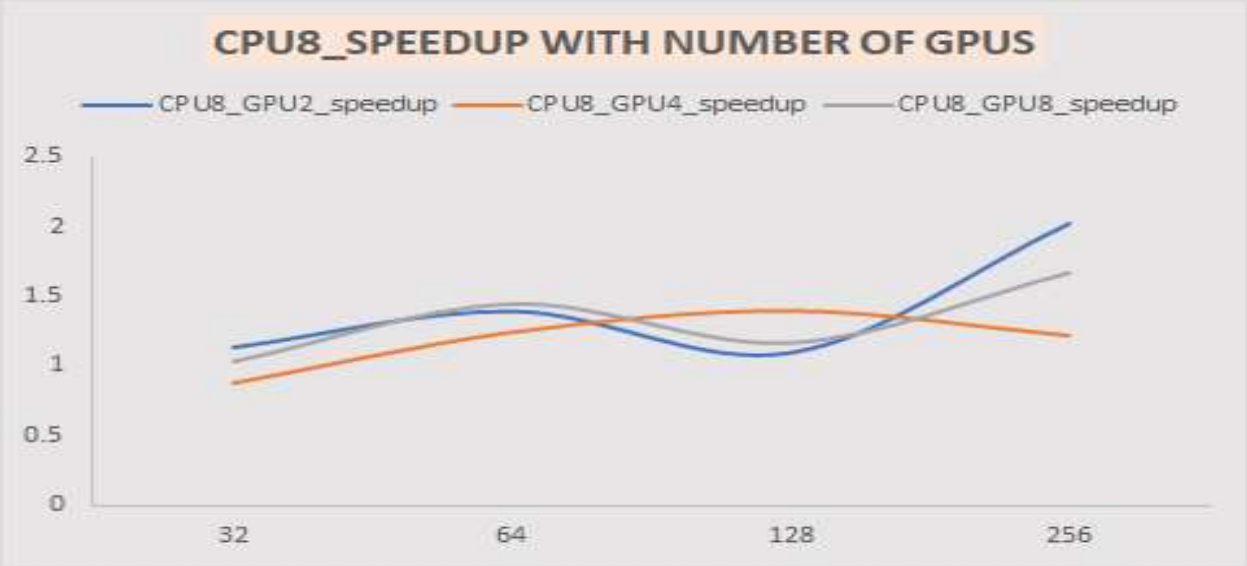

Figure 4-8 The Speedup of the distribution of eight CPUs with multiple numbers of GPUs for four batch sizes

Figure 4.8 showed the speedup when the model was run on eight CPUs each run with a different number of GPUs for four batch sizes. Running the model on 8 CPUs was relatively fast compared to the GPUs, and so the speedup values were relatively close to 1 for all numbers of GPUs except for large batch sizes when the GPUs outperformed the CPUs.

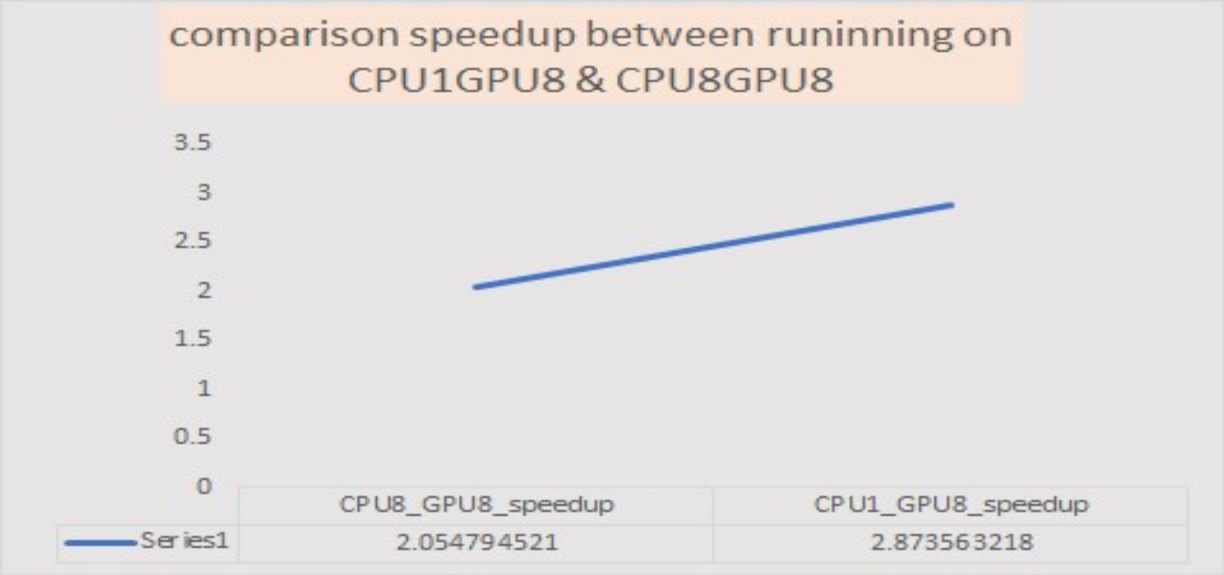

Figure 4-9 The speedup is calculated as the execution time when the model runs eight CPUs divided by the execution time when the model runs on eight CPUs with eight GPUs for the first value in this figure. The speedup is calculated as the execution time when the

Figure 4.9 shows the comparison speedup between when the model was running on eight CPUs with eight GPUs for one batch size and when the model was running on one CPU with eight GPUs for the same batch size. Figure 4.8 showed that the speedup was higher when eight GPUs were requested than the speedup without requesting the GPUs. But The enhancement is better when the model runs on one CPU with eight GPUs. Although more distribution should make the speedup higher, in this case, the time spent on the distribution, or the processor can run a smaller number of devices at a higher clock rate than more number of devices makes the run time slower. So, reducing the number of processors with increasing the number of GPUs is the best result that is obtained from running this model on the clusters.

4.4 Results of Accuracy and loss functions for training and validation datasets:

The results of this section are shown in Figure 4.10 and Figure 4.11. Figure 4.10 shows the accuracy function values for the training dataset and the validation dataset through the epoch steps of building the parallel model. The accuracy is a method for measuring a classification model's performance. It is the percentage count of predictions where the predicted value equals the true value. In figure 4.10, the accuracy for the training dataset comes out to 0.98 (98 correct predictions out of 100 total examples), which means the classifier model is doing great. While the accuracy for the validation dataset comes out to 0.70, that means the classifier model is still a good accuracy percentage, and the model makes a good prediction.

Figure 4.11 shows the loss function values for the training dataset and the validation dataset through the same epoch steps. The categorical cross entropy loss function is used to compute loss values between true labels and predicted labels for this model because it's mainly used for multiclass classification problems. This function evaluates how good our model is performing by comparing what the model is predicting with the actual output value. Loss function is the value of the difference between the actual output and the predicted output. It is the count of the probabilities or uncertainty of a prediction based on how much the prediction varies from the true value. The model's prediction is good as long as the loss is less; otherwise, the model's prediction becomes worse when the loss is greater. If output-pred is very far off from real output, the Loss value will be very high. As seen in figure 4.11, the loss curve of the training dataset is going very low, which means the prediction for the training is very good. The loss curve of the validation dataset starts with high values, then during the epochs continue running the curve becomes low until it reaches

almost close to the curve of loss values of the training dataset, which means the prediction for the validation is becomes better through increasing the epochs, and the model is learning.

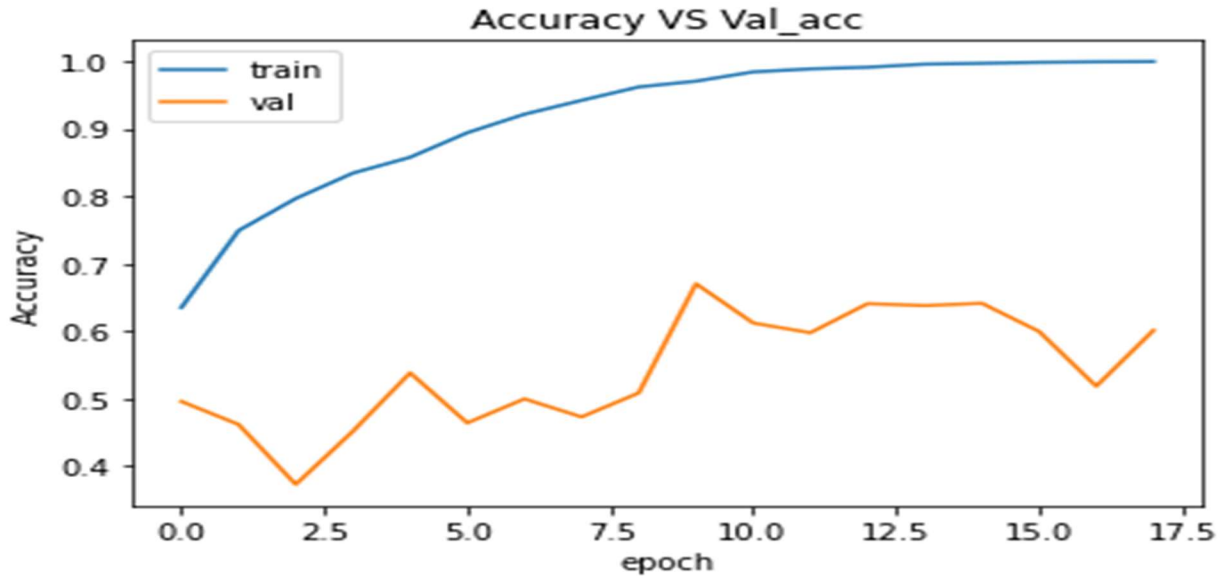

Figure 4-10 The accuracy function values for the training dataset vs. the validation dataset

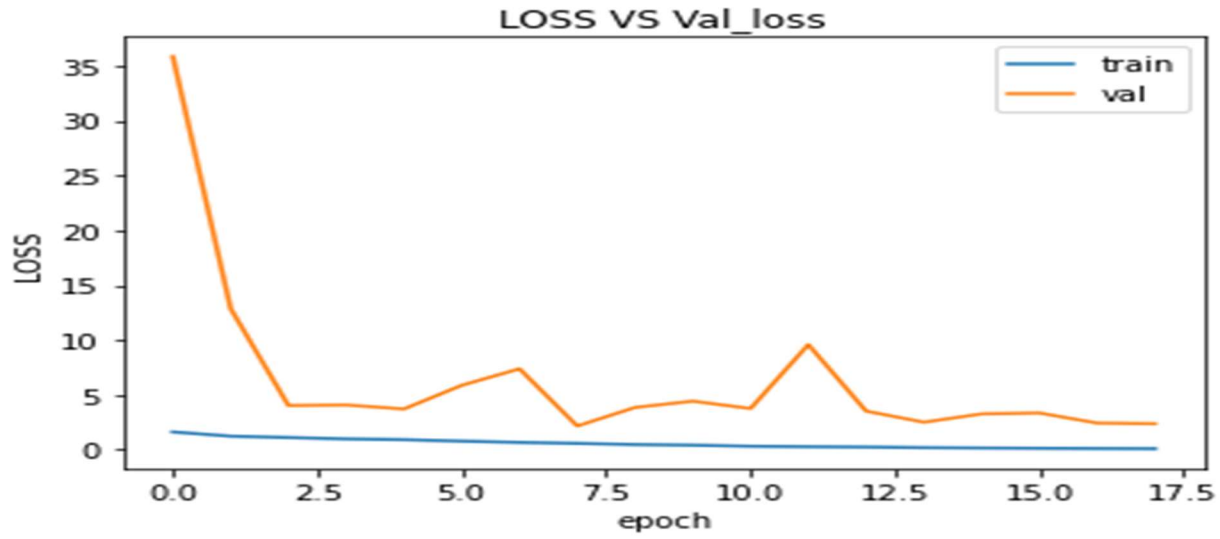

Figure 4-11 The loss function values for the training dataset vs. the validation dataset

4.5 Results of running time and Speedup for the same dataset on different devices:

This section discusses the running time for different models which run on different devices. All these models are to classify the same dataset. There are six values of running time in this section. The first value is the time to run a model using the TF 1.0 on a laptop with no accelerator. The second value is the time to run a model using the TF 1.0 on a laptop with a GPU accelerator. The third value is the time to run a model using the TF 1.0 on a GPU cluster. The fourth value is the time to run a model using the TF 2.0 on a GPU cluster. The fifth value is the time to run a model using the TF 1.0 on a TPU cluster. The third value is the time to run a model using the TF 2.0 on a TPU cluster. Tensor Processing Unit (TPU) is an application-specific integrated circuit to accelerate the AI calculations and algorithm (TensorFlow projects). TPUs are usually Cloud platform workers, but they can be a smaller version of the chip (TensorFlow.org, 2015).

See figure (4.12) shows the values of run time measured by seconds. Figure (4.13) shows the calculated speedup depending on the speedup is the old run time divided by the new run time. The first value in figure 4.12 is considered as the old runtime, and other values are considered as the new runtime to calculate the speed up for them see figure 4.13. Figure 4.14 and figure 4.15 show the plotting chart of the comparison time of the values in figure 4.12 and the comparison speedup of the values in figure 4.13.

TF_1.0_NO_Acc elerator_ laptop	TF_1.0_YES_Ac celerator_ laptop	TF_1.0_GPU _cluster	TF_2.0_GPU _cluster	TF_1.0_TPU _cluster	TF_2.0_TPU _cluster
27466.7	23591	1272.7	528	6	4.1

Figure 4-12 The time results when running the model for training the same dataset on different devices

speedup_TF1.0_YES_Accelerator_laptop	speedup_TF1.0_GPU_cluster	speedup_TF2.0_GPU_cluster	speedup_TF1.0_TPU_cluster	speedup_TF2.0_TPU_cluster
1.164287228180	21.581441030879	52.020265151515	4577.78333333	6699.195121951

Figure 4-13 The speedup results when running the model for training the same dataset on different devices

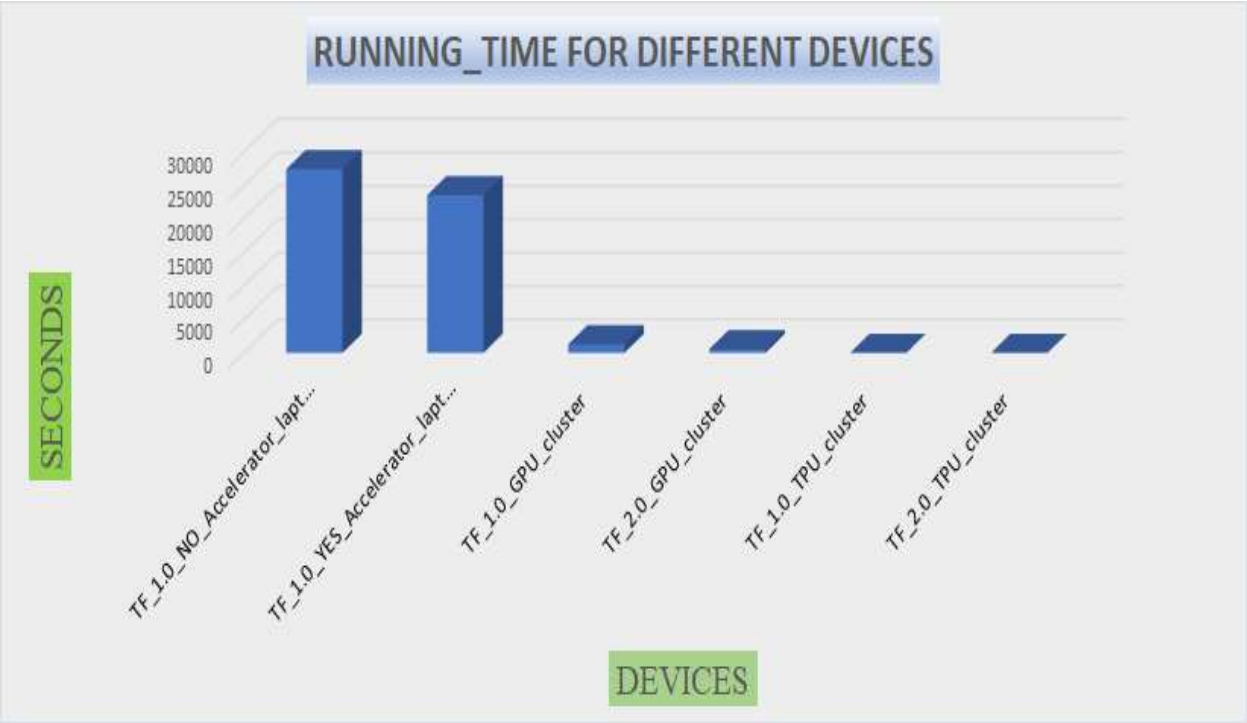

Figure 4-14 Plot shows the run time comparison when running the model on different devices

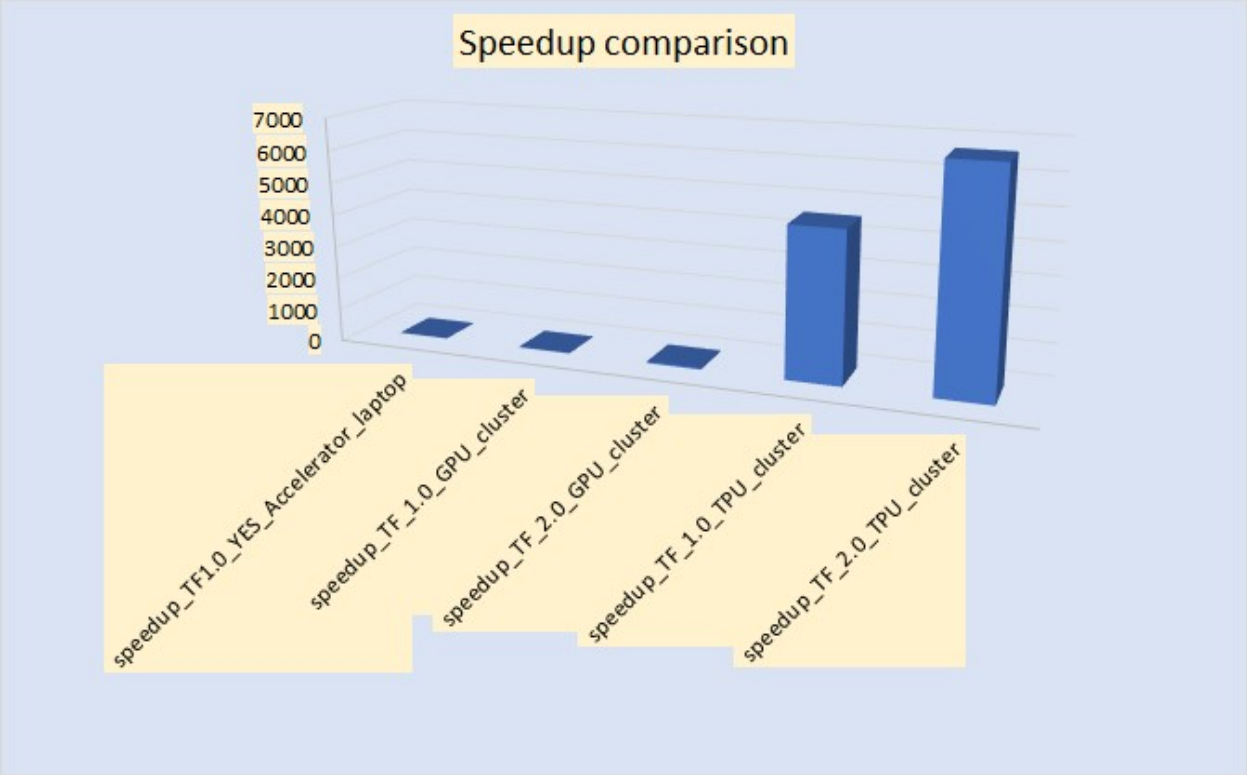

Figure 4-15 Plotting the speedup comparison when running the model on different devices

4.6 Summary:

This section provides an overall summary of the results of this work. As seen at the beginning of this chapter, when distributing the model on multiple processors on the cluster, the performance was generally faster. A larger batch size with multiple processors improves the performance even more. Besides the fast performance on multiprocessors, adding increasing numbers of GPUs for more distribution made the performance even better. There were several slow results when more GPUs were added, which was unexpected but repeatable. The accuracy was high, and the loss value was low for the training dataset and a reasonable percentage for the validation dataset. The timing results with TensorFlow 2.0 are much better when the model runs on CPU clusters and/or GPU clusters than when the model runs on a laptop with or without an accelerator. The results are

also much better than can be obtained with TensorFlow 1.0. Also, the results of the performance are the best when the model runs on Tensor Processing Unit (TPU) clusters with TensorFlow 1.0 and TensorFlow 2.0.

CHAPTER FIVE: CONCLUSION AND FUTURE WORK

5.1 Conclusion:

The performance of TensorFlow definitely improved when the model was distributed and run on multiple processing units and/or GPUs. The parallelizing capabilities of TensorFlow, in general, become better for the model of a neural network of classification images with a large dataset. This makes using GPUs with TensorFlow well worth the effort in most cases as faster performance will result. When training a model with multiple devices, increasing the batch size uses more computing power and thus lowers overall running time. However, the benefits of more GPUs become more significant when a large batch size is used, i.e., with more computation, the GPUs contribute more.

The selection of the right device can make the speed up of the calculation time higher, which is important for machine learning and neural networks computing.

Using the GPUs becomes not important when a small dataset has been trained with a less complex neural networks model because there is not a big difference in the run time with or without using the GPU. So, using the CPU is better in this case for a higher response of the calculation with less cost. Otherwise, for a large dataset with a more complex neural networks model, the calculation requires a parallel computation to achieve more speed up, and the higher parallel requirements should be calculated by the GPU. As seen in the results chapter, the use of parallel computing when using TensorFlow and GPUs can speed up the computational process time of image classification.

The results using the TensorFlow 2.0 framework shows that the framework can simplify the code with its built-in function, which made it easier for interested users to work with Tensorflow to build machine learning models. Also, Tensorflow can ease the code syntax of parallel methods

without changing the code when switching from CPU to GPU. Changing the distribution syntax is what is required for switching from CPU to GPU parallel method, and all the rest of the code is still the same.

5.2 Future Work:

We find machine learning a very interesting and demanding field for future work. Future work may include running a machine learning model on Tensor Processing Units (TPUs) that are available in Google cloud clusters. These work on the TensorFlow platform that is provided by the MATLAB Programming language and has the ability to build a machine learning model. Additionally, working on more comparisons between CPU clusters and GPU clusters in other regards could be of interest. For example, it would be of interest to see the effects of running a neural network model regarding not just the run time but also memory allocation for CPU vs. GPU. Other differences in resource allocation could also be studied.

REFERENCES:

[1] Abadi, Martín, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467* (2016). http://download.tensorflow.org/paper/whitepaper2015.pdf.

[2] Adie, Heronimus Tresy Renata, and Ignatius Aldi Pradana. "Parallel computing accelerated image inpainting using gpu cuda, theano, and tensorflow." *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)*. IEEE, 2018. https://ieeexplore-ieee-org.wv-o-ursus-proxy02.ursus.maine.edu/document/8534858.

[3] Adiyoso, Widiarto, et al. "Time performance analysis of multi-CPU and multi-GPU in big data clustering computation." *2018 international workshop on big data and information security (IWBIS)*. IEEE, 2018. https://ieeexplore-ieee-org.wv-o-ursus-proxy02.ursus.maine.edu/document/8471715.

[4] Aurélien, Géron. "Hands-on machine learning with scikit-learn & tensorflow." Geron Aurelien (2017). https://upload.houchangtech.com/pdf/Hands-on_Machine_Learning.pdf.

[5] Cuomo, Salvatore, et al. "Parallel implementation of a machine learning algorithm on GPU." *International Journal of Parallel Programming* 46.5 (2018): 923-942. https://www.researchgate.net/publication/322149433_Parallel_Implementation_of_a_Machine_Learning_Algorithm_on_GPU.

[6] Goldsborough, Peter. "A tour of TensorFlow." *arXiv preprint arXiv:1610.01178* (2016). https://export.arxiv.org/pdf/1610.01178.

[7] Hasan, Shafaatunnur, Siti Mariyam Shamsuddin, and Noel Lopes. "Machine learning big data framework and analytics for big data problems." *Int. J. Advance Soft Compu. Appl* 6.2 (2014). https://www.researchgate.net/publication/272356868_Machine_Learning_Big_Data_Framework_and_Analytics_for_Big_Data_Problems.

[8] Jia, Zhihao, et al. "Exploring hidden dimensions in parallelizing convolutional neural networks." *arXiv preprint arXiv:1802.04924* (2018). https://arxiv.org/pdf/1802.04924v1.pdf.

[9] Lind, Eric, and Ävelin Pantigoso Velasquez. "A Performance Comparison between CPU and GPU in TensorFlow." (2019). https://www.diva-portal.org/smash/get/diva2:1354858/FULLTEXT01.pdf.

[10] Lopes, Noel, Bernardete Ribeiro, and Ricardo Quintas. "GPUMLib: a new library to combine machine learning algorithms with graphics processing units." *2010 10th International Conference on Hybrid Intelligent Systems*. IEEE, 2010. https://www.researchgate.net/publication/224182227_GPUMLib_A_new_Library_to_combine_Machine_Learning_algorithms_with_Graphics_Processing_Units.

[11] Meuth, Ryan J., and Donald C. Wunsch. "A survey of neural computation on graphics processing hardware." *2007 IEEE 22nd international symposium on intelligent control*. IEEE, 2007.
https://web.archive.org/web/20200318115134/https://scholarsmine.mst.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=2404&context=ele_comeng_facwork.
[12] Mustafa, Eslam M., Mohamed A. Elshafey, and Mohamed M. Fouad. "Enhancing the performance of CNN-based blind image steganalysis approach using multi-GPU TESLA P100." *IOP Conference Series: Materials Science and Engineering*. Vol. 610. No. 1. IOP Publishing, 2019. https://www.researchgate.net/publication/344140896_Enhancing_CNN-based_Image_Steganalysis_on_GPUs.

[13] Sharma, Chetan. "Big data analytics using neural networks." (2014). https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1366&context=etd_projects.

[14] Silaparasetty, Nikita. "Introducing Tensorflow 2.0." *Machine Learning Concepts with Python and the Jupyter Notebook Environment*. Apress, Berkeley, CA, 2020. 191-213. Web Resource
ONLINE, UM Orono Electronic Resource.

[15] Singh, Pramod, and Avinash Manure. "Introduction to TensorFlow 2.0." *Learn TensorFlow 2.0*. Apress, Berkeley, CA, 2020. 1-24. https://link.springer.com/chapter/10.1007/978-1-4842-5558-2_1.

[16] Sundar, KV Sai, et al. "Evaluating training time of Inception-v3 and Resnet-50,101 models using TensorFlow across CPU and GPU." *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE, 2018.
https://www.researchgate.net/publication/328458615_Evaluating_Training_Time_of_Inception-v3_and_Resnet-50101_Models_using_TensorFlow_across_CPU_and_GPU.

[17] Wang, Minjie. *Flexible and Efficient Systems for Training Emerging Deep Neural Networks*. Diss. New York University, 2020.
https://www.proquest.com/docview/2393632009?https://www.library.umaine.edu/auth/EZProxy/test/authej.asp?url=accountid=14583&accountid=14583.

[18] TensorFlow.org. Retrieved November 10, 2015, https://www.tensorflow.org.

[19] VACC "Vermont Advanced Computing Core at University of Vermont" 2019 https://vacckb.helpline.w3.uvm.edu/vacc/kb/.

[20] Anaconda, 2017, https://docs.conda.io/en/latest/miniconda.html.

[21] kaggle.com/shayanriyaz, 2019, https://www.kaggle.com/shayanriyaz/riceleafs.

[22] winscp.net. Retrieved 21 November 2014, https://winscp.net/eng/docs/introduction.

[23] Hans-D, Wehle. "Machine Learning, Deep Learning, and AI: What's the Difference." (2017). https://www.researchgate.net/publication/318900216_Machine_Learning_Deep_Learning_and_AI_What%27s_the_Difference.

[24] medium.com, Ravi Ranjan Singh, Mar 15, 2020. https://medium.com/analytics-vidhya/tensorflow-tutorial-a-beginners-guide-to-tensorflow-part-2-5d1219a8ba5c

[25] medium.com, Danqing Liu, Nov 30, 2017. https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7.

[26] tutorialexample.com, by admin, May 2, 2020. https://www.tutorialexample.com/understand-tensorflow-sess-run-a-beginner-introduction-tensorflow-tutorial/.

[27] en.wikipedia.org, last edited 24 June 2021. https://en.wikipedia.org/wiki/Softmax_function#Statistical_mechanics

[28] analyticsvidhya.com, Shipra Saxena, April 5, 2021. https://www.analyticsvidhya.com/blog/2021/04/introduction-to-softmax-for-neural-network/.

[29] developers.google.com, 2020. https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss.

[30] androidkt.com, 2021. https://androidkt.com/batch-size-step-iteration-epoch-neural-network/.

[31] linuxnetmag.com, 2021. https://linuxnetmag.com/miniconda-vs-anaconda/.

[32] stackoverflow.com, 2020. https://stackoverflow.com/questions/59112527/primer-on-tensorflow-and-keras-the-past-tf1-the-present-tf2#:~:text=Differences%20between%20TF1%20and%20TF2%20TF1%20requires%20a,the%20full%20graph%20defined%20before%20starting%20the%20computations.

[33] MA Raza, Sep 11, 2020. https://towardsdatascience.com/machine-learning-with-spark-f1dbc1363986.

[34] Gelenbe, Erol. *Multiprocessor speed-up, Amdahl's Law, and the Activity Set Model of parallel program behavior*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1988, https://www.researchgate.net/publication/24286816_Multiprocessor_speed-up_Amdahl%27s_Law_and_the_Activity_Set_Model_of_parallel_program_behavior.

APPENDIX A: TENSORFLOW CODE OF THE NEURAL NETWORK MODEL

1. Muli Workers distribution for multi-CPUs:

This section contains the code of one part of this work. It has a multi workers distribution method (tf.distribute.MultiWorkerMirroredStrategy) that is provided by TensorFlow to distribute and run the model on multi CPUs. The code consists of importing the necessary libraries at the first step, then loading the dataset to be read inside the code; after that, preparing the dataset and splitting it into a training dataset and validation dataset. Set up the input pipeline by choosing the batch size and data generator so the dataset can be distributed to multi-devices to make the model run in parallel. The code contains two functions, one for building the dense layers, then the second one for building the model, and it is calling the first function inside it to complete the model building. Then, staringt to call the model and run it by using a strategy scope that works with the multi workers distribution method on TensorFlow. Furthermore, starting to train the model and saving the results on a CSV file to be able to plot them after moving the file to the PC.

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Lambda, Dense, Flatten,GlobalAveragePooling2D,BatchNormalization,Dropout,Activation
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
import numpy as np
from glob import glob
import matplotlib.pyplot as plt
import os
import json
import sys
```

```
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
os.environ.pop('TF_CONFIG', None)
if '.' not in sys.path:
  sys.path.insert(0, '.')
import tensorflow as tf
print(tf.__version__)



train_path = '/gpfs1/home/i/a/ialkaaba/RiceLeafs/RiceLeafs/train/'
val_path = '/gpfs1/home/i/a/ialkaaba/RiceLeafs/RiceLeafs/validation/'

data_dir = os.path.join(os.path.dirname('/gpfs1/home/i/a/ialkaaba/'),
'RiceLeafs/RiceLeafs')

train_dir = os.path.join(data_dir, 'train')
train_BrownSpot_dir = os.path.join(train_dir, 'BrownSpot')
train_Healthy_dir = os.path.join(train_dir, 'Healthy')
train_Hispa_dir = os.path.join(train_dir, 'Hispa')
train_LeafBlast_dir = os.path.join(train_dir, 'LeafBlast')


validation_dir = os.path.join(data_dir, 'validation')
validation_BrownSpot_dir = os.path.join(validation_dir, 'BrownSpot')
validation_Healthy_dir = os.path.join(validation_dir, 'Healthy')
validation_Hispa_dir = os.path.join(validation_dir, 'Hispa')
validation_LeafBlast_dir = os.path.join(validation_dir, 'LeafBlast')

train_BrownSpot_names = os.listdir(train_BrownSpot_dir)
print('train_BrownSpot',train_BrownSpot_names[:10])

train_Healthy_names =  os.listdir(train_Healthy_dir)
print('train_Healthy', train_Healthy_names[:10])

train_Hispa_names = os.listdir(train_Hispa_dir)
print('train_Hispa', train_Hispa_names[:10])

train_LeafBlast_names =  os.listdir(train_LeafBlast_dir)
print('train_LeafBlast', train_LeafBlast_names[:10])

## Image Count

import time
import os
from os.path import exists

def count(dir, counter=0):
```

```
    "returns number of files in dir and subdirs"
    for pack in os.walk(dir):
        for f in pack[2]:
            counter += 1
    return dir + " : " + str(counter) + " files"

print('total images for training :', count(train_dir))
print('total images for validation :', count(validation_dir))


## Feature Engineering
def one_hot_label(image, label):
    label = tf.one_hot(label, NUM_CLASSES)
    return image, label

from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.preprocessing.image import ImageDataGenerator


def rice_dataset(BATCH_SIZE):
    IMAGE_SIZE = (150, 150)
    training_generator = image_dataset_from_directory(
        train_dir,
        validation_split = 0.2,
        subset = 'training',
        seed = 220,
        image_size = IMAGE_SIZE,
        batch_size = BATCH_SIZE,
      )

    validation_generator = image_dataset_from_directory(
        validation_dir,
        validation_split = 0.3,
        subset = 'validation',
        seed = 220,
        batch_size = BATCH_SIZE,
        image_size = IMAGE_SIZE,
      )


    class_names = os.listdir(train_dir)

    print(class_names)
    training_generator.class_names = class_names
    validation_generator.class_names = class_names

    NUM_CLASSES = len(class_names)
```

```
    train_ds = training_generator.map(one_hot_label, num_parallel_calls=AUTOTUNE)
    val_ds = validation_generator.map(one_hot_label, num_parallel_calls=AUTOTUNE)
    train_ds = train_ds.cache().prefetch(buffer_size = AUTOTUNE)
    val_ds = val_ds.cache().prefetch(buffer_size = AUTOTUNE)
    return train_ds,val_ds



class_names = os.listdir(train_dir)

print(class_names)
NUM_CLASSES = len(class_names)
    return block
def dense_block(units,dropout_rate):
    block = tf.keras.Sequential([
        tf.keras.layers.Dense(units,activation = 'relu'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dropout(dropout_rate)
    ])
    return block

## Build the Model
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32,(3,3),activation = 'relu',input_shape =
(*IMAGE_SIZE,3)),
        tf.keras.layers.MaxPooling2D(2,2),
#       tf.keras.layers.Conv2D(64,(3,3),activation = 'relu'),
#       tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Conv2D(128,(3,3),activation = 'relu'),
        tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Flatten(),
        dense_block(512,0.2),
        dense_block(128,0.3),
        tf.keras.layers.Dense(4,activation = 'softmax')

    ],    name = 'Conv2D_Model')

    return model
tf_config = {
    'cluster': {
        'worker': ['localhost:12345', 'localhost:23456']
    },
    'task': {'type': 'worker', 'index': 0}
}

json.dumps(tf_config)
```

```
os.environ['GREETINGS'] = 'Hello TensorFlow!'
num_workers = len(tf_config['cluster']['worker'])

strategy = tf.distribute.MultiWorkerMirroredStrategy()
## Data Generators
AUTOTUNE = tf.data.experimental.AUTOTUNE
EPOCHS = 100
IMAGE_SIZE = (150, 150)
BATCH_SIZE = 32 * num_workers    ## the batch size can be changed
train_ds,val_ds = rice_dataset(BATCH_SIZE)

with strategy.scope():
   model = build_model()

   METRICS = [tf.keras.metrics.AUC(name='auc')]

   model.compile(
   optimizer = 'adam',
   loss = tf.losses.CategoricalCrossentropy(),
   metrics = METRICS
   )
## Training the Model
def exponential_decay(lr0,s):
   def exponential_decay_fn(epoch):
      return lr0 * 0.1 **(epoch/s)
   return exponential_decay_fn

exponential_decay_fn = exponential_decay(0.001,20)

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)

checkpoint_cb = tf.keras.callbacks.ModelCheckpoint('CPU_cluster_model.h5',
                                    save_best_only = True)

early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience = 10,
                                     restore_best_weights = True)

CPU_32 = model.fit(
   train_ds,
   validation_data=val_ds,
   callbacks=[checkpoint_cb, early_stopping_cb, lr_scheduler],
   epochs=EPOCHS
)
import pandas as pd
# convert the history.history dict to a pandas DataFrame:
```

```
hist_df = pd.DataFrame(CPU_32.history)
# or save to csv:
hist_csv_file = 'CPU_32.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
```

2. Distribution for multi-CPUs and multi-GPUs:

This code is exactly the same as the previous code except for the distribution method. This code uses distribution training with Keras to distribute the model on multi GPUs (tf.distribute.MirroredStrategy). The difference is with setting the input pipeline. When choosing the batch size multiply it by the number of GPUs that used to run the model instead of multiplying it by the number of workers. This method also uses strategy scope to run the model in parallel.

```
 import tensorflow as tf
from tensorflow.keras.layers import Input, Lambda, Dense, Flatten,GlobalAveragePooling2D,BatchNormalization,Dropout,Activation
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
import numpy as np
from glob import glob
import matplotlib.pyplot as plt
import os
import json
import sys

print(tf.__version__)



train_path = '/gpfs1/home/i/a/ialkaaba/RiceLeafs/RiceLeafs/train/'
val_path = '/gpfs1/home/i/a/ialkaaba/RiceLeafs/RiceLeafs/validation/'

data_dir = os.path.join(os.path.dirname('/gpfs1/home/i/a/ialkaaba/'), 'RiceLeafs/RiceLeafs')

train_dir = os.path.join(data_dir, 'train')
```

```
train_BrownSpot_dir = os.path.join(train_dir, 'BrownSpot')
train_Healthy_dir = os.path.join(train_dir, 'Healthy')
train_Hispa_dir = os.path.join(train_dir, 'Hispa')
train_LeafBlast_dir = os.path.join(train_dir, 'LeafBlast')


validation_dir = os.path.join(data_dir, 'validation')
validation_BrownSpot_dir = os.path.join(validation_dir, 'BrownSpot')
validation_Healthy_dir = os.path.join(validation_dir, 'Healthy')
validation_Hispa_dir = os.path.join(validation_dir, 'Hispa')
validation_LeafBlast_dir = os.path.join(validation_dir, 'LeafBlast')

train_BrownSpot_names = os.listdir(train_BrownSpot_dir)
print('train_BrownSpot',train_BrownSpot_names[:10])

train_Healthy_names =  os.listdir(train_Healthy_dir)
print('train_Healthy', train_Healthy_names[:10])

train_Hispa_names = os.listdir(train_Hispa_dir)
print('train_Hispa', train_Hispa_names[:10])

train_LeafBlast_names =  os.listdir(train_LeafBlast_dir)
print('train_LeafBlast', train_LeafBlast_names[:10])

## Image Count

import time
import os
from os.path import exists

def count(dir, counter=0):
    "returns number of files in dir and subdirs"
    for pack in os.walk(dir):
        for f in pack[2]:
            counter += 1
    return dir + " : " + str(counter) + " files"

print('total images for training :', count(train_dir))
print('total images for validation :', count(validation_dir))


## Feature Engineering
def one_hot_label(image, label):
    label = tf.one_hot(label, NUM_CLASSES)
    return image, label

from tensorflow.keras.preprocessing import image_dataset_from_directory
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator


def rice_dataset(BATCH_SIZE):
    IMAGE_SIZE = (150, 150)
    training_generator = image_dataset_from_directory(
        train_dir,
        validation_split = 0.2,
        subset = 'training',
        seed = 220,
        image_size = IMAGE_SIZE,
        batch_size = BATCH_SIZE,
      )

    validation_generator = image_dataset_from_directory(
        validation_dir,
        validation_split = 0.3,
        subset = 'validation',
        seed = 220,
        batch_size = BATCH_SIZE,
        image_size = IMAGE_SIZE,
       )


    class_names = os.listdir(train_dir)

    print(class_names)
    training_generator.class_names = class_names
    validation_generator.class_names = class_names

    NUM_CLASSES = len(class_names)
    train_ds = training_generator.map(one_hot_label, num_parallel_calls=AUTOTUNE)
    val_ds = validation_generator.map(one_hot_label, num_parallel_calls=AUTOTUNE)
    train_ds = train_ds.cache().prefetch(buffer_size = AUTOTUNE)
    val_ds = val_ds.cache().prefetch(buffer_size = AUTOTUNE)
    return train_ds,val_ds



class_names = os.listdir(train_dir)

print(class_names)
NUM_CLASSES = len(class_names)

def dense_block(units,dropout_rate):
    block = tf.keras.Sequential([
        tf.keras.layers.Dense(units,activation = 'relu'),
```

```
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dropout(dropout_rate)
    ])
    return block

## Build the Model
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32,(3,3),activation = 'relu',input_shape = (*IMAGE_SIZE,3)),
        tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Conv2D(64,(3,3),activation = 'relu'),
        tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Conv2D(128,(3,3),activation = 'relu'),
        tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Flatten(),
        dense_block(512,0.2),
        dense_block(128,0.3),
        tf.keras.layers.Dense(4,activation = 'softmax')

    ],    name = 'Conv2D_Model')

    return model

strategy = tf.distribute.MirroredStrategy()
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))

## Set up the input pipeline

AUTOTUNE = tf.data.experimental.AUTOTUNE
EPOCHS = 100
IMAGE_SIZE = (150, 150)
BATCH_SIZE = 32 * strategy.num_replicas_in_sync   ## batch size can be changed
train_ds,val_ds = rice_dataset(BATCH_SIZE)
with strategy.scope():
    model = build_model()

    METRICS = [tf.keras.metrics.AUC(name='auc')]

    model.compile(
    optimizer = 'adam',
    loss = tf.losses.CategoricalCrossentropy(),
    metrics = METRICS
    )
## Training the Model
def exponential_decay(lr0,s):
```

```
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 **(epoch/s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(0.001,20)

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)

checkpoint_cb = tf.keras.callbacks.ModelCheckpoint('GPU_cluster_model.h5',
                                                   save_best_only = True)

early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience = 10,
                                                     restore_best_weights = True)

GPU_32 = model.fit(
    train_ds,
    validation_data=val_ds,
    callbacks=[checkpoint_cb, early_stopping_cb, lr_scheduler],
    epochs=EPOCHS
)


import pandas as pd


# convert the history.history dict to a pandas DataFrame:
hist_df = pd.DataFrame(GPU_32.history)


# or save to csv:
hist_csv_file = 'GPU_32.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
```

3. The code for plotting the data:

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv('C:\\Users\\ent_w\\Desktop\\thesis\\CPU_32.csv') ## here we
can          ## change the file to plot any file from running a different model.

df[['epoch', 'auc', 'val_auc']].plot(
    x='epoch',
    xlabel='epoch',
    ylabel='Accuracy',
```

```
    title='Accuracy VS Val_acc for CPU_32'
 )
plt.legend(['train', 'val'])
plt.show()
df[['epoch', 'loss', 'val_loss']].plot(
    x='epoch',
    xlabel='epoch',
    ylabel='LOSS',
    title='LOSS VS Val_loss for CPU_32'
)
plt.legend(['train', 'val'])
plt.show()
```

The code for SLURM Script:

a. CPU cluster:

The slurm script has been created using the Nano text editor. slurm is the batch system used for the Bluemoon cluster. The Slurm commands are several lines starting with **#SBATCH**. These Slurm commands provide the job setup information used by Slurm, including resource requests. Slurm commands begin with **#SBATCH** then followed by the commands to be executed, the "executable section." However, the hash sign **(#)** followed by a space is a "comment." It is often the comments are written above commands as explanations of the executable commands below. These comments are ignored (not processed as commands). The slurm commands were included to run the job are:

1. Specifying a partition is bluemoon.
2. Nodes, Tasks, and Cores (CPUs)
3. Memory
4. .Walltime is the maximum amount of time your job will run. It's the runtime of your job.
5. The job name is used as part of the name of the slurm script file.

6. Email Address to receive all mail types the begin time, the fail, and the end time for running the job.
7. The executable section of slurm script comes after all previous lines and tells what slurm job should run, which is the python file name.

After the job script is written, we can submit the job on the cluster terminal using the sbatch command with slurm filename **(sbatch filename.sh).**

```
#!/bin/bash
#SBATCH --partition=bluemoon
#SBATCH --nodes=1
#SBATCH --ntasks=1 ## the number of processors that we requested
#SBATCH --mem=30G
#SBATCH --time=07:00:00
#SBATCH --job-name=C2G2_128 ##the name of slurm file that contain .sh
#SBATCH --mail-user=intisar.alkaabawi@maine.edu
#SBATCH --mail-type=ALL
# provide YOUR python filename
python CPU_32.py  ## the name of python that we want to run it
```

b. GPU cluster:

The slurm script for the DeepGreen cluster (GPU) is exactly the same as the previous script for the BlueMoon cluster (CPU). In addition to the previous script the DeepGreen slurm script requires the user to load the CUDA path and export the CUDA library so the script is able to run the job on GPUs.

```
#!/bin/bash
#SBATCH --partition=dggpu
#SBATCH --nodes=1
#SBATCH --ntasks=8 ## the number of processors that we requested
#SBATCH --gres=gpu:4 ## the number of GPUs that we requested
#SBATCH --mem=30G
#SBATCH --time=01:00:00
#SBATCH --job-name=GPU8_256  ##name of slurm file that contain .sh

#SBATCH --mail-user=intisar.alkaabawi@maine.edu
```

```
#SBATCH --mail-type=ALL
# loads CUDA
export PATH=${PATH}:/gpfs3/arch/x86_64-rhel7/cuda-10.0/bin
export     LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/gpfs3/arch/x86_64-
rhel7/cuda-10.0/lib64
# provide YOUR python filename
python GPU_256.py ## the name of python that we want to run it
```

BIOGRAPHY OF THE AUTHOR

Intisar Al-Kaabawi was born in Baghdad, Iraq, on April 28th, 1984. She obtained a bachelor's degree in Computer Science at Mustansiriyah University, 2006. In addition, she was a teaching assistant at the University of Maine and a research assistant at the same University. Intisar is a candidate for the Master of Science degree in Computer Engineering from the University of Maine in December 2021.