VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Course Code: CSE4001

Course name: Parallel and Distributed Computing

Slot: C2

Prof. MURUGAN K

FINAL REPORT

TOPIC: Parallelization of CPU and GPU for Image Processing

Group Members:

Vidhi Agarwal – 18BCE2190

Shaifali Choudhary – 18BCE2242

Saurabh Kumar Jha – 18BCE0197

Guneshwar Singh Manhas – 18BCE0960

CONTENTS

ABSTRACT

Image classification algorithms such as Convolutional Neural Network used for classifying huge image datasets takes a lot of time to perform convolution operations, thus increasing the computational demand of image processing. Compared to CPU, Graphics Processing Unit (GPU) is a good way to accelerate the processing of the images. Parallelizing multiple CPU cores is also another way to process the images faster. Increasing the system memory (RAM) can also decrease the computational time of image processing. Comparing the architecture of CPU and GPU, the former consists of a few cores optimized for sequential processing whereas the later has thousands of relatively simple cores clocked at approx. 1Ghz. The aim of this project is to compare the performance of parallelized CPUs and a GPU. Python's Ray library is being used to parallelize multicore CPUs. The benchmark image classification algorithm used in this project is Convolutional Neural Network. The dataset used in this project is Plant Disease Image Dataset. Our results show that the GPU implementation achieves 80% speedup compared to the CPU implementation.

Keywords – Convolutional Neural Network, parallel computing, speedup.

PROBLEM STATEMENT

It has always been cumbersome to process real time images. Studies showed that there can be two ways to analyse this. One hand is about central processing unit (CPU) and the other is about Graphics Processing Unit (GPU). To obtain highest possible performance they have to be used at the same time. This project will compare the performance of CPU and GPU for real time image processing. The main drawback of Python's Multiprocessing module is that it cannot be used for handling large numeric data. It cannot be used in Deep Learning Frameworks such as Keras as it decreases the accuracy of the models.

INTRODUCTION

In recent years, parallel computing and soft computing has become a rapidly evolving field of study. The demand for parallel processing in increasing day by day. There are various software tools and libraries by which we can parallelize our programs. For example, we have OPENMP in c++ for parallel computing. OPENMP supports FORTRAN, C and C++. It is basically an Application Programming Interface for shared Memory Model programming. Python has its separate parallel processing module named Multiprocessing. Multiprocessing module enables to spawn multiple processes, allowing programmer to fully leverage the computing power of multiple processors.

The main drawback of Python's Multiprocessing module is that it cannot be used for handling large numeric data. It cannot be used in Deep Learning Frameworks such as Keras as it decreases the accuracy of the models. Shared variables cannot be used in the Multiprocessing Module. Python also has a Parallel and Distributed computing framework called Ray. Ray can be used for developing emerging AI applications such as image classification, face recognition etc. Parallelizing multiple cores of CPU using Ray can also increase the speedup of the model significantly. The benchmark image classification algorithm used in this project is Convolutional Neural Network. The dataset used in this project is Plant

Disease Image dataset containing around 30000 images. The system is configured with 16 GB RAM with 4 CPU Cores and Tesla P100 GPU. This project compares the performance of 2-core, 3-core and 4-core parallelized CPUs with GPU

A. COMPARISON OF CPU AND GPU ARCHITECTURE

A Graphics Processing Unit (GPU) is mostly used in hardware devices to support applications which require heavy graphics processing such as 3-D modelling, designing, etc. GPUs are mostly used in gaming PCs to accelerate gaming graphics processing. Nowadays, GPGPU (General Purpose Graphics Processing Unit) are used to accelerate computational graphics processing workloads. The main purpose of GPU is to project textured polygons onto the screen in a fiercely competitive consumer-facing industry. The design goals of GPU were to increase the throughput and not focus on single threads. GPUs basically hide memory latency through parallelism. They let the programmer deal with raw storage hierarchy. The most important design goal for GPU is to avoid high frequency clock speed. The GPU consists of multiple independent CPU cores. The original design goals of CPU were to make single threads faster, reduce latency through large caches and use prediction and speculation for the instruction stream to guard against the branches in the instruction stream.

Modern CPU-Style core design emphasizes individual thread performance. Each core of the CPU executes scalar or vector operations whereas each GPU core only executes vector instructions. CPU uses Single Instruction Multiple Data (SIMD) parallelism through ILP and vector execution units whereas GPU uses Single Instruction Multiple Data (SIMD) parallel execution of all operations. The GPU cores are designed and engineered to switch quickly between threads to recover stalls. A GPU has multiple cores and each core has one or more wide SIMD vector units. These wide SIMD vector units execute one instruction stream and also has a pool of shared memory. Each core of the GPU shares a registry file shared privately among all the ALUs. Each core quickly switches thread blocks to hide memory latency. Modern GPUs combine multiple wide vector processing cores with local and global-shared memory.

B. PERFORMANCE COMPARISON OF CPU AND GPU

We all need to understand that the latency optimization takes place in CPUs while GPUs are bandwidth optimized. For Deep Learning, GPUs hide latency via thread parallelism thus offering high bandwidth. This is the reason why GPUs are widely used in training deep learning models. Compared to CPUs, GPUs cannot have more memory capacity. 24GB of RAM is known to be the highest amount of memory that a GPU can hold whereas CPUs can have RAM upto 1TB. The main advantage of using GPUs in Deep Learning is that they can perform matrix operation faster than CPUs. Deep Learning largely comprises of large matrix operations. Researchers say that the GPUs can gain speedup up to 60% more than that of CPUs.

C. PARALLELIZING MULTIPLE CPU CORES

To parallelize multiple cores of CPU, the OS has to be a multiprocessing OS. Multiprocessing OS uses two

or more CPU cores to run a single program. Each core works on different parts of the same task, or on two or more different tasks. They are used for high speed computations and to increase the power of the computer as the execution takes place in parallel. Python's Multiprocessing module can be used to parallelize all the CPU cores present in the machine. Multiprocessing module uses Pool property to parallelize CPUs. The pool distributes the workload of the program to the available processors using First In First Out (FIFO) scheduling. But this module cannot be used with Deep Learning frameworks such as Keras as these frameworks are not multiprocessing safe.

Python has a parallel and distributed framework called Ray which exclusively used for building AI applications by parallelizing the CPU cores. Ray automatically detects the number of CPU and GPU cores present in the system.

LITERATURE REVIEW

S. No	Title	Author	Journal name and Date	Key Concepts	Advantages, Disadvantages, Future Enhancement
GPU vs CPU COMPUTING					
1	A Comparative Evaluation of The GPU Vs. The CPU For Parallelization of Evolutionary Algorithms Through Multiple Independent Runs	Anna Syberfeldt and Tom Ekblom	International Journal of Computer Science & Information Technology (IJCSIT) Vol 9, No 3, June 2017	The GPU platform is widely adopted to implement parallelization. The paper focuses on the performance of the GPU in comparison to that of CPU in the context of multiple nheritance.	A number of experiments can be used to infer and analyze the efficiency of GPU versus that of CPU. In future, the project can be further expanded with proper comparison on sequential versus parallel computing in GPU and compare the same with CPU.

2	Parallel implementation of machine learning-based many-body potentials on CPU and GPU	Yaoguang Zhai, Nathaniel Danandeh, Zhenye Tan, Sicun Gao, Francesco Paesani, Andreas W. Götz	SC'18, Nov 2018, Dallas, Texas, USA 2018	[1] Models of Machine Learning can be used efficiently to accurately simulate many-body extension of the total energy.	[1] OpenMP parallel implementation as well as CUDA implementations can fasten the CPU and GPU pro cesses respectively.
3	Computing Performance Benchmarks among CPU, GPU, and FPGA	Christopher Robert Cullinan, Timothy Richard Frattesi, Christopher Michael Wyant Worcester Polytechnic Institute	E-project-0302 12-123508 2015	[1] The high performamce computing, as it is growing, needs to be shifted from CPU to GPU.	[1] The performances can be monitored in CPU as well as GPUs by having same application run on them.
IMAGE CLASSIFICATION AND PROCESSING USING MACHINE LEARNING					
4	CNN-RNN: A Unified Framework for Multi-label Image Classification	Jiang Wang Yi Yang Junhua Mao Zhiheng Huang Chang Huang Wei Xu Baidu Research University of California at Los Angles Facebook Speech Horizon Robotics	IEEE Transactions on Image Processing, Vol. 24, No. 12, December 2015	. The paper proposes a framework combined of advantages of image embedding and label models by using CNN and RNN. . The proposed model can focus on different image regions when predicting labels but small objects still poses a challenge.	[1] In future the work can be extended to not only predict labels but also predict segmentation.

5	Deep Recurrent Neural Networks for Hyperspectral Image Classification	Student Member, IEEE, Pedram Ghamisi, Member, IEEE, and Xiao Xiang Zhu, Senior Member, IEEE	IEEE Transactions on Geoscience And Remote Sensing, Vol. 55, No. 7, July 2017	[1] The work proposes an RSS model for HSIs classification.	[1] The proposed model considers the intrinsic sequential data structure of a hyperspectral pixel for the first time, representing a novel methodology for better understanding, modelling, and processing of hyperspectral data.
6	Spectral–Spatial Feature Extraction for Hyperspectral Image Classification: A Dimension Reduction and Deep Learning Approach	Wenzhi Zhao and Shihong Du	IEEE Xplore 2015	[1] The work proposesa method for classification of hyperspectral images which are used for urban mapping.	[1] The method helps to overcome the singularity and improvise the classification accuracies; the technique also helps to overcome the unpleasantness in spatial filtering but it is difficult to interpret the significance of CNN is future form.
7	The Effectiveness of Data Augmentation in Image Classification using Deep Learning	Jason Wang, Luis Perez Stanford University 450 Serra Mall	Stanford University 13 December, 2017	[1] The research shows various methods to increase the accuracy of classification tasks.	[1] It proposes the combination of traditional augmentation techniques followed by neural augmentation further improves the classification strengths.
8	Hyperspectral Image Classification Using Deep Pixel- Pair Features	Wei Li, Member, IEEE, Guodong Wu, Student Member, IEEE, Fan Zhang, Member, IEEE, and Qian Du, Senior	IEEE Transactions on Geoscience And Remote Sensing, Vol. 55, No. 2	[1] In this paper a framework based on deep PFFs for CNN based classification.	[1] The model is to exploit the similarity between the pixels and ensure sufficient amount of data input to learn. The surrounding pixels with the testing pixel to fit in the pixel- pair

		Member, IEEE	February 2017		model.
IMAGE CLASSIFIER USING TENSORFLOW					
9	CNN-RNN: A Unified Framework for Multi-label Image Classification	Jiang Wang Yi Yang Junhua Mao Zhiheng Huang Chang Huang Wei Xu Baidu Research University of California at Los Angles Facebook Speech Horizon Robotics	IEEE Transactions on Image Processing, Vol. 24, No. 12, December 2015	. The paper proposes a framework combined of advantages of image embedding and label models by using CNN and RNN. . The proposed model can focus on different image regions when predicting labels but small objects still poses a challenge.	[1] In future the work can be extended to not only predict labels but also predict segmentation.
10	Deep Recurrent Neural Networks for Hyperspectral	Student Member, IEEE, Pedram Ghamisi, Member, IEEE, and Xiao	IEEE Transactions on Geoscience And	[1] The work proposes an RSS model for HSIs classification.	[1] The proposed model considers the intrinsic sequential data structure of a hyperspectral pixel for the

	Image Classification	Xiang Zhu, Senior Member, IEEE	Remote Sensing, Vol. 55, No. 7, July 2017		first time, representing anovel methodology for better understanding, modelling, and processing of hyperspectral data.
11	Spectral–Spatial Feature Extraction for Hyperspectral Image Classification: A Dimension Reduction and Deep Learning Approach	Wenzhi Zhao and Shihong Du	IEEE Xplore 2015	[1] The work proposesa method for classification of hyperspectral images which are used for urban mapping.	[1] The method helps to overcome the singularity and improvise the classification accuracies; the technique also helps to overcome the unpleasantness in spatial filtering but it is difficult to interpret the significance of CNN is future form.
12	The Effectiveness of Data Augmentation in Image Classification using Deep Learning	Jason Wang, Luis Perez Stanford University 450 Serra Mall	Stanford University 13 December, 2017	[1] The research shows various methods to increase the accuracy of classification tasks.	[1] It proposes the combination of traditional augmentation techniques followed by neural augmentation further improves the classification strengths.
13	Hyperspectral Image Classification Using Deep Pixel-Pair Features	Wei Li, Member, IEEE, Guodong Wu, Student Member, IEEE, Fan Zhang, Member, IEEE, and Qian Du, Senior	IEEE Transactions on Geoscience And Remote Sensing, Vol. 55, No. 2	[1] In this paper a framework based on deep PFFs for CNN based classification.	[1] The model is to exploit the similarity between the pixels and ensure sufficient amount of data input to learn. The surrounding pixels with the testing pixel to fit in the pixel- pair

		Member, IEEE	February 2017		model.
SERIAL PROGRAMMING IN CPU AND GPU					
14	A Microbenchmark to study GPU performance models	Vasily Volkov	PPoPP 2018 Proceedings of the 23rd ACM SIGPLAN Symposium 2018	[1] To model the performance, many solutions were proposed for makinga stable, functional basic microarchitectural features of GPUs of NVIDIA.	[1] There are many superior methods for estimating performance of GPs on the basis of their classical work.
15	CPU-GPU Processing	Zulfiqar Ali Memon, Fahad Samad, Zafar Rehman Awan, Abdul Aziz	CPU-GPU Processing September 2017	[1] To compare and consider the performance, the architectures is also taken into account.	[2] To achieve quite high performance from CPU and GPU processors, a lot of steps are taken in the light of architectural advancement in design as well as the computation techniques.

16	Performance Modeling and analysis of Heterogeneous Lattice Boltzmann Simulations on CPU-GPU clusters	Christian Feichtinger, Johannes Habich, Harald Kostler, Ulrich Rude, Takayuki Aoki	Parallel Computing 46, 1-13 2015	[1] When a system has multiple GPUs, the domain partitioning method is used to distribute computational load to multiple GPUs.	[1] Optimized and faster execution and performance can be obtained by creating multiple GPUs working as one-dimensional, or two- dimensional or three-dimensional domain partitioning. The performance increases by 3 to 4 times.
PARALLELIZING GPU USING CUDA					

17	Study of Parallel Image Processing with the Implementation of vHGW Algorithm using CUDA on NVIDIA'S GPU Framework	Sanjay Saxena, Shiru Sharma, Neeraj Sharma	Proceedings of the World Congress on Engineering 2017 Vol I WCE 2017, July 5-7, 2017, London, U.K.	The paper tells us about the parallel image processing using CUDA (Computer Unified Device Architecture) which is an environment provided by NVIDIA for extremely large and complex parallel processing techniques. The computing mechanism in GPU is done by CUDA. This research paper also tells us about the importance of parallel computation in image processing algorithms used for medical purposes.	It helps to encapsulate the complexities of the GPU computations from the developers or programmers. The vHGW (van Herk/Gill-Werman morphology) algorithm is mainly used for extracting a particular or identifying a particular section in an image.
18	Performance of Medical Image Processing Algorithms Implemented in CUDA running on GPU based Machine.	Kalaiselvi Thiruvenkadam, Sriramakrishnan Padmanaban, Somasundaram Karuppanagounder	I.J. Intelligent Systems and Applications, 2018, 1, 58-68 Published Online January	This paper is mostly based on evaluation of algorithms for analyzing and processing the images for the medical purposes. The paper focuses on using parallel processing on GPU with CUDA. This paper also discusses in detail some of the medical image processing operations like the operations on an MRI scanned image in which the image of the	. The paper talks about the algorithms that will run in CUDA for adaptive filtering, anisotropic diffusion, bilateral filtering, non-local means (NLM) filtering, K-Means segmentation and feature extraction. . The paper states that up to 3-300 times more speedup is achieved when GPU processor with CUDA is used instead of a CPU processor in PPT model as well as PST model.

			2018 in MECS		

				3D human head is taken by capturing several 2D pictures.	
19	Digital image processing using parallel computing based on CUDA technology	I P Skirnevskiy, A V Pustovit, M O Abdrashitova	International Conference on Information Technologies in Business and Industry 2016	This paper describes the advantage of using GPU (Graphical Processing Unit) for image processing algorithms and tasks involving huge amounts of data processing. The paper compares the performance of the system by applying some algorithms in a GPU environment and without a GPU environment.	The paper states the ways of using parallel processing using CUDA for the analysis of these huge amounts ofdata. The research paper also states several other applications of parallel computing using CUDA such as algorithms for removal of image noise.
PARALLELIZING CPU USING RAY					
20	Ray: A Distributed Framework For Emerging AI Applications	Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol	Proceedings of the 13th USENIX Symposium on Operating Systems Design and lamentationion (OSDI '18) October 8-10, 2018	[1] The tasks done using TensorFlow, PyTorch or MXNet can be parallelized in CPU for faster performance.	[1] The scaling beyond 1.8 million can be achieved per seconds for several difficult strengthening learning applications. Ray gives a strong combination of flexibility, performance and ease of use for the development of future AI applications.

PROPOSED WORK

The novelty of our project is in the fact that we have shown the time comparison of the processing in both Central Processing Unit and the Graphics Processing Unit.

The Image Classification algorithm of Convolutional Neural Network has a huge dataset which takes a lot of time to perform the said analysis. Thus, the process of parallelizing CPU core and using GPU gives us a lot of computation power. Note that a serial processor would provide an accuracy of a mere 80%.

Thus, our project has provided with an adequate comparison of the performance of the parallel CPU and GPU computing.

SOFTWARE USED

Languages: Python

Software's: Anaconda Jupyter Notebook Kaggle

Libraries:

- NumPy
- Cv2
- Pickle
- Os
- Skylearn
- Keras
- Matplotlib
- Ray
- Time

ALGORITHM

--Import the necessary package-parallise using ray.init(num_cpus=4)

-start = time.time()

-use CNN model to process the image

-Divide the given dataset into train and test

-model.summary() for getting summary of model

-opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS) for optimising the model

-ray.get(output) for getting the output

-end=time.time()

-.format(end-start) for printing time

FRAMEWORK MODEL

A. APPROACH

The following approach is being followed to compare the performance of parallelized CPUs and GPUs.

- Downloading the required dataset for our model. In this case, it is Plant disease image dataset. The dataset is taken from Kaggle's dataset library.
- Designing the architecture for Convolutional Neural Network.
- Preprocessing all the images of the dataset.
- Dividing the given dataset into training set and testing set.
- Parallelizing CPU cores with Ray and training the model. Varying the number of CPU cores
- Using GPU to train the model
- Observing the execution time for all the epochs for GPU and CPU computing
- Observing the accuracy for CPU and GPU computing
- Analysis of results by plotting the appropriate graphs

B. ARCHITECTURE OF CONVOLUTIONAL NEURAL NETWORK

Deep learning is a subdomain of machine learning which is based on neural networks. The architecture of neural network is based on the anatomy of the human neuron. That's the reason these algorithms are called neural networks. Convolutional neural network is a type of neural network which is used mostly in visual imagery. As the name suggests, this neural network uses convolution operation to process images. Convolution operation of two functions f and g is defined as the integral product of these two functions after one is reversed and shifted. The CNN consists of multiple hidden layers which are bounded by an input layer and an output layer.

These hidden layers are basically a series of convolution operations. The common operations present in the hidden layer are Convolution, ReLu, Max pooling, Flattening and Full connection

Step 1: Convolution layer

In this layer, the feature/filter is moved to each potential position on the image. Firstly, the features and image are lined up. Then each image pixel is multiplied by the corresponding feature pixel. Then these values are added and the sum is divided by the total number of pixels in the feature/filter.

Step 2: ReLU Layer

In this layer all the negative values from the filtered images are replaced with zeros. We do this to avoid summing up of the values up to zero. ReLU (Rectified Linear Unit) function activates the node when the input is greater than a certain threshold value, otherwise it returns 0. The input becomes linear to dependent variable when it increases after certain threshold.

Step 3: Pooling Layer

Shrinking of image stack takes place in this layer. First we choose a window size (usually 2 or 3). Then we pick up a stride (usually 2) to traverse the whole image. Then the window is walked across the filtered images. From each window, the maximum value is found and stored.

Step 4: Flattening Layer

In this step the 2-D matrix obtained from the pooling layer is converted into a 1-D matrix (vector) so that it can be fed into the fully connected layer

Step 5: Stacking the layers

In order to increase the accuracy of the model, we can repeat the stack of above layers before feeding the vectors into the fully connected layer.

Step 6: Fully connected layer

The actual classification happens in this layer. This is the final layer of CNN. In this layer the filtered and shrinked images are put into a single list. This layer connects every node of one layer to every node of the other layer. The vector obtained from flattening layer is passed through layer to predict the class.

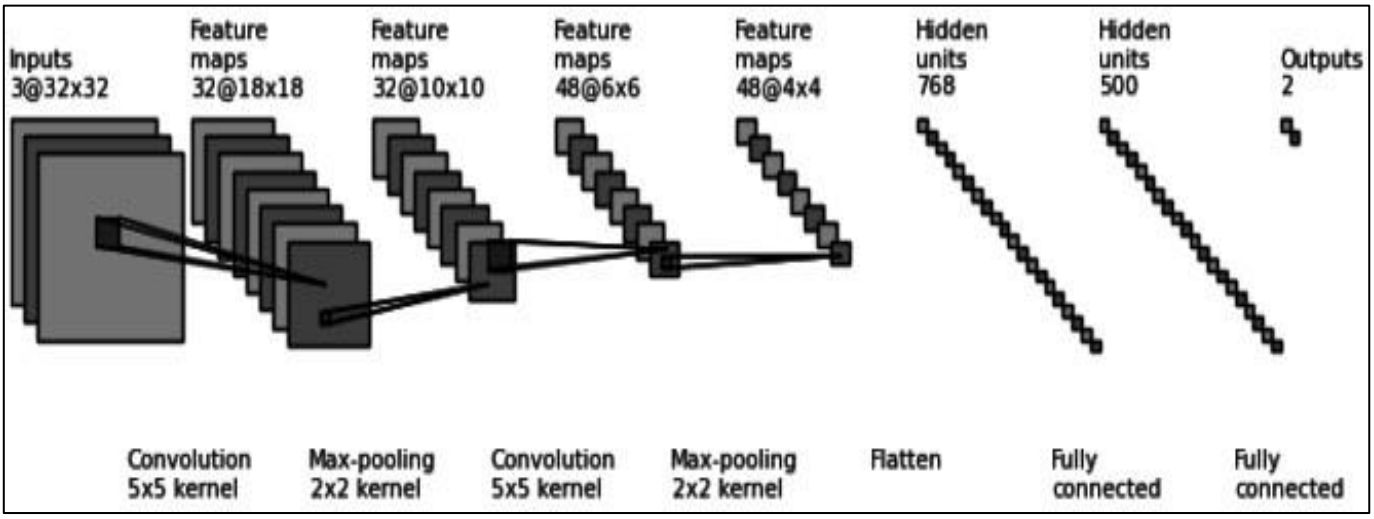


Fig: Architecture of CNN

C. PARALLELIZING CPUs USING RAY

Python has a parallel and distributed module called Ray which is used to build scalable AI applications. Ray helps in easily parallelizing python applications. Ray is an open source project built by Machine Learning department of University of California, Berkeley. OpenMP, Python Multiprocessing and ZeroMQ are low-level primitive parallel programming libraries whereas Ray is a high-level parallel and distributed framework. Ray automatically detects the number of CPU and GPU cores present in the hardware system. The API ray.init() initializes the ray context and the number of workers of the cluster. We can also specify the number of resources such as number of CPUs and GPUs. By default Ray initializes number of CPUs as the number of CPU cores present in the system. We can overwrite the number of CPUs and GPUs using the following ray.init(num_cpus=2, num_gpus=1). The decorator @ray.remote(num_cpus=3) is called before a function. This decorator assigns the resources to the function for the computation. The API ray.get() returns the output of the function and the Ids of the objects of the function.

D. ABOUT THE DATASET

The dataset used in this project is Plant Disease Detection image dataset. The same is available on Kaggle Data Science Platform. This dataset contains the images of leaf of the plant having a particular disease. The dataset is divided into following 15 categories:

- Pepper_bell_Bacterial_spot
- Pepper_bell_healthy
- Potato_Early_blight
- Potato_healthy,
- Tomato_Spider_mites_Two_spotted_spider
- Potato_Late_blight,
- Tomato_Bacterial_spot, Tomato_Early_blight
- Tomato_healthy, Tomato_Late_blight,
- Tomato_Leaf_mold, Tomato_Septoria_leaf_spot
- Tomato_target_spot,
- Tomato_Tomato_mosaic,_virus
- Tomato_Tomato_Yellow_Leaf_Curl_Virus.

CODE READABILITY

Main Code:

```
import numpy as np
import pickle
import cv2
from os import listdir
from sklearn.preprocessing import LabelBinarizer   #for converting into binary form
from keras.models import Sequential                #for Sequential model generation
from keras.layers.normalization import BatchNormalization   #for normalizing
from keras.layers.convolutional import Conv2D              #for convolution layer
from keras.layers.convolutional import MaxPooling2D        #for max pooling layer
from keras.layers.core import Activation, Flatten, Dropout, Dense     #for dense , flatten,activation
from keras import backend as K
from keras.preprocessing.image import ImageDataGenerator   #for generating image data
from keras.optimizers import Adam     #for optimizing our model
from keras.preprocessing import image
from keras.preprocessing.image import img_to_array
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt      #for plotting graph
!pip install setproctitle
!pip install redis

import ray
ray.init(ignore_reinit_error=True)

import time
print("Timer started\n")
print("-------------------------------------------------------------------------------------")
```

```
start = time.time()

EPOCHS = 5  #for specifying number of epochs
INIT_LR = 1e-3
BS = 32
default_image_size = tuple((256, 256))
image_size = 0
directory_root = '../input/plantvillage/'
width=256
height=256
depth=3

@ray.remote(num_cpus=4)
def train_fit4():
    def convert_image_to_array(image_dir): # for skimming through the folder
        try:
            image = cv2.imread(image_dir)
            if image is not None :
                image = cv2.resize(image, default_image_size)
                return img_to_array(image)
            else :
                return np.array([])
        except Exception as e:
            print(f"Error : {e}")
            return None

image_list, label_list = [], []
try:
    print("[INFO] Loading images ...")
    root_dir = listdir(directory_root)
```

```
    for directory in root_dir :
        # remove .DS_Store from list
        if directory == ".DS_Store" :     #for removing metadata
            root_dir.remove(directory)
            # for removing metadata
    for plant_folder in root_dir :
        plant_disease_folder_list = listdir(f"{directory_root}/{plant_folder}")
        for disease_folder in plant_disease_folder_list :
            # remove .DS_Store from list
            if disease_folder == ".DS_Store" :
                plant_disease_folder_list.remove(disease_folder)


        for plant_disease_folder in plant_disease_folder_list:
            print(f"[INFO] Processing {plant_disease_folder} ...")
            plant_disease_image_list =
listdir(f"{directory_root}/{plant_folder}/{plant_disease_folder}/")
            for single_plant_disease_image in plant_disease_image_list :
                if single_plant_disease_image == ".DS_Store" :
                    plant_disease_image_list.remove(single_plant_disease_image)


            for image in plant_disease_image_list[:200]:
                image_directory =
f"{directory_root}/{plant_folder}/{plant_disease_folder}/{image}"
                if image_directory.endswith(".jpg") == True or image_directory.endswith(".JPG")
== True:
                    image_list.append(convert_image_to_array(image_directory))
                    label_list.append(plant_disease_folder)
                    print("[INFO] Image loading completed")
except Exception as e:
    print(f"Error : {e}")
```

```
image_size = len(image_list)
label_binarizer = LabelBinarizer()
image_labels = label_binarizer.fit_transform(label_list)
pickle.dump(label_binarizer,open('label_transform.pkl', 'wb'))
n_classes = len(label_binarizer.classes_) # for generating classes

print(label_binarizer.classes_)

np_image_list = np.array(image_list, dtype=np.float16) / 225.0

print("[INFO] Spliting data to train, test")
x_train, x_test, y_train, y_test = train_test_split(np_image_list, image_labels, test_size=0.2, random_state = 42)

aug = ImageDataGenerator(
    rotation_range=25, width_shift_range=0.1,
    height_shift_range=0.1, shear_range=0.2,
    zoom_range=0.2,horizontal_flip=True,
    fill_mode="nearest")
def create_and_fit_model():
    width=256
    height=256
    depth=3
    EPOCHS=2
model = Sequential()
inputShape = (height, width, depth)
chanDim = -1
if K.image_data_format() == "channels_first":
    inputShape = (depth, height, width)
    chanDim = 1
```

```
model.add(Conv2D(32, (3, 3), padding="same",input_shape=inputShape))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation(“softmax"))

model.summary()
```

```
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS) # optimizing the model
model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["accuracy"])
model.fit_generator(
    aug.flow(x_train, y_train, batch_size=BS),
    validation_data=(x_test, y_test),
    steps_per_epoch=len(x_train) //BS,
    epochs=EPOCHS, verbose=1
)
print("[INFO] Calculating model accuracy")
scores = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {scores[1]*100}")

create_and_fit_model()
output = train_fit4.remote() # for calling the function to be parallelized using ray
ray.get(output)
print("----------------------------------------------------------------------------------------------------")
end = time.time()
print("Time elapsed is {}".format(end-start)) # print time taken
```

Graph Code:

```
[ ] import matplotlib.pyplot as plt
    import numpy as np
    import math
```

```
[ ] labels1 = ["Epoch 1", "Epoch 2", "Epoch 3", "Epoch 4", "Epoch 5"]
    values_gpu = [50,46,47,46,46]
    values_cpu_4 = [577,569,562, 572,565]
    values_cpu_3 = [620, 610, 615, 612, 612]
    values_serial = [538, 548, 537, 535, 538]
```

```
[ ] index = np.arange(len(labels1))
```

```
[ ] def plot_bar_x_gpu():
      #this is for plotting purpose
      index = np.arange(len(labels1))
      low=min(values_gpu)
      high = max(values_gpu)
      plt.figure(figsize=(7,7))
      plt.ylim([math.ceil(low-0.5*(high-low)), math.ceil(high+0.5*(high-low)) ])
      plt.bar(index, values_gpu, color='blue')
      plt.xlabel('Epoch', fontweight='bold', fontsize=13)
      plt. ylabel('Time Elapsed in seconds' , fontweight='bold', fontsize=13)
      plt.xticks (index, labels1, fontsize=10, rotation=30)
      plt.title('GPU computing performance analysis', fontsize=18)
      plt.show()

    def plot_bar_x_cpu4():
      index = np.arange(len(labels1))
      low = min(values_cpu_4)
      high = max(values_cpu_4)
      plt.figure(figsize=(7,7))
      plt.ylim( [math.ceil(low-0.5*(high-low)), math.ceil(high+0.5*(high-low))])
      plt.bar(index, values_cpu_4, color='green')
      plt.xlabel('Epoch', fontweight='bold', fontsize=13)
      plt.ylabel('Time Elapsed in seconds', fontweight='bold', fontsize=13)
      plt.xticks(index, labels1, fontsize=13, rotation=30)
      plt.title('Performance analysis of 4-CPU Ray Parallelization', fontsize=18)
      plt.show()

    def plot_bar_x_cpu3():
      index = np.arange(len(labels1))
      low = min(values_cpu_3)
      high = max(values_cpu_3)
      plt.figure(figsize=(7,7))
      plt.ylim( [math.ceil(low-0.5*(high-low)), math.ceil(high+0.5*(high-low))])
      plt.bar(index, values_cpu_3, color='red')
      plt.xlabel('Epoch', fontweight='bold', fontsize=13)
      plt.ylabel('Time Elapsed in seconds', fontweight='bold', fontsize=13)
```

```
[ ]   plt.ylim( [math.ceil(low-0.5*(high-low)), math.ceil(high+0.5*(high-low))])
      plt.bar(index, values_cpu_3, color='red')
      plt.xlabel('Epoch', fontweight='bold', fontsize=13)
      plt.ylabel('Time Elapsed in seconds', fontweight='bold', fontsize=13)
      plt.xticks(index, labels1, fontsize=13, rotation=30)
      plt.title('Performance analysis of 3-CPU Ray Parallelization', fontsize=18)
      plt.show()

    def serial():
      index = np.arange(len(labels1))
      low = min(values_serial)
      high = max(values_serial)
      plt.figure(figsize=(7,7))
      plt.ylim( [math.ceil(low-0.5*(high-low)), math.ceil(high+0.5*(high-low))])
      plt.bar(index, values_serial, color='orange')
      plt.xlabel('Epoch', fontweight='bold', fontsize=13)
      plt.ylabel('Time Elapsed in seconds', fontweight='bold', fontsize=13)
      plt.xticks(index, labels1, fontsize=13, rotation=30)
      plt.title('Performance analysis of Serial Python', fontsize=18)
      plt.show()
```

```
[ ]  def multi_bar():
       barWidth = 0.20
       r1 = np.arange(len(labels1))
       r2 = [x + barWidth for x in r1]
       r3 = [x + barWidth for x in r2]
       r4 = [x + barWidth for x in r3]
       plt. figure(figsize=(20,7))
       plt.bar(r1, values_gpu, color='blue', width=barWidth, edgecolor='white', label='GPU')
       plt.bar(r2, values_cpu_4, color='green', width=barWidth, edgecolor='white', label='4-CPU')
       plt.bar(r3, values_cpu_3, color='red', width=barWidth, edgecolor='white', label='3-CPU')
       plt.bar(r4, values_serial, color='orange', width=barWidth, edgecolor='white', label='Serial Python')
       plt.xlabel('Epochs', fontweight='bold', fontsize=15)
       plt.ylabel('Time elapsed in seconds', fontweight='bold', fontsize=15)
       plt.xticks([r + barWidth for r in range(len(labels1))], labels1)
       plt.title('Comparison of performance of GPU, 3-CPU, 4-CPU and Serial Python', fontsize=18)
       plt.legend()
       plt.show()
```

```
[ ]  def accuracy():
       labels=['GPU', '4-CPU', '3-CPU', 'Serial python']
       values = [94.67, 93.23, 92.32, 88.75]
       index = np.arange(len(labels))
       colors=['blue', 'green','red', 'orange']
       plt.figure(figsize=(7,7))
       low = min(values)
       high= max(values)
       plt.ylim([math.ceil(low-0.5*(high-low)), math.ceil(high+0.5*(high-low))])
       plt.bar(index, values, color=colors)
       plt.xlabel('Processor', fontsize=13)
       plt.ylabel('Model Accuracy', fontsize=13)
       plt.xticks (index, labels, fontsize=10, rotation=30)
       plt.title('Comparison of accuracy', fontsize=18)
       plt.show()
```

NOVELTY OF THE WORK

The novelty of our project is in the fact that we have shown the time comparison of the processing in both Central Processing Unit and the Graphics Processing Unit. As is evident by the outputs and the result analysis, the accuracy of the computation increases when we go from a serial python, to 3-CPU, then 4-CPU and then finally to the GPU, which shows an accuracy of above 95%.

The Image Classification algorithm of Convolutional Neural Network has a huge dataset which takes a lot of time to perform the said analysis. Thus, the process of parallelizing CPU core and using GPU gives us a lot of computation power. Note that a serial processor would provide an accuracy of a mere 80%.

Thus, our project has provided with an adequate comparison of the performance of the parallel CPU and GPU computing.

EXPERIMENTAL RESULTS

To compare the performance of GPU with CPU, we have configured the following setup.

For CPU: Intel Core i5 7th generation with 16GB RAM and 2GB Nvidia 940MX 2GB graphics card.

For GPU: Intel Core i5 7th generation with 16GB RAM and Tesla P100 graphics card.

The dataset was split into training set and testing set with 80% of the images in the training set. To record the first observation, we observed the execution time and the accuracy of the model after training the same for serial python. We observed the execution time for each epoch. The accuracy was observed to be 88.75%.

Serializing:

After serializing the process, we got an accuracy of 88.59%.

Following were the execution time of all the epochs.

Epoch sequence number	Execution time (in seconds)
1	538
2	540
3	537
4	535
5	538

Parallelizing 3 CPU cores:

After parallelizing 3 CPU cores using ray.init(num_cpus=3), we got an accuracy of 92.32%.

Following were the execution time of all the epochs.

Epoch sequence number	Execution time (in seconds)
1	620
2	610
3	615
4	612
5	612

Parallelizing 4 CPU cores:

After parallelizing 4 CPU cores using ray.init(num_cpus=4), we got an accuracy of 93.23%.

Following were the execution time of all the epochs.

Epoch sequence number	Execution time (in seconds)
1	577
2	569
3	562
4	572
5	565

Parallelizing GPU:

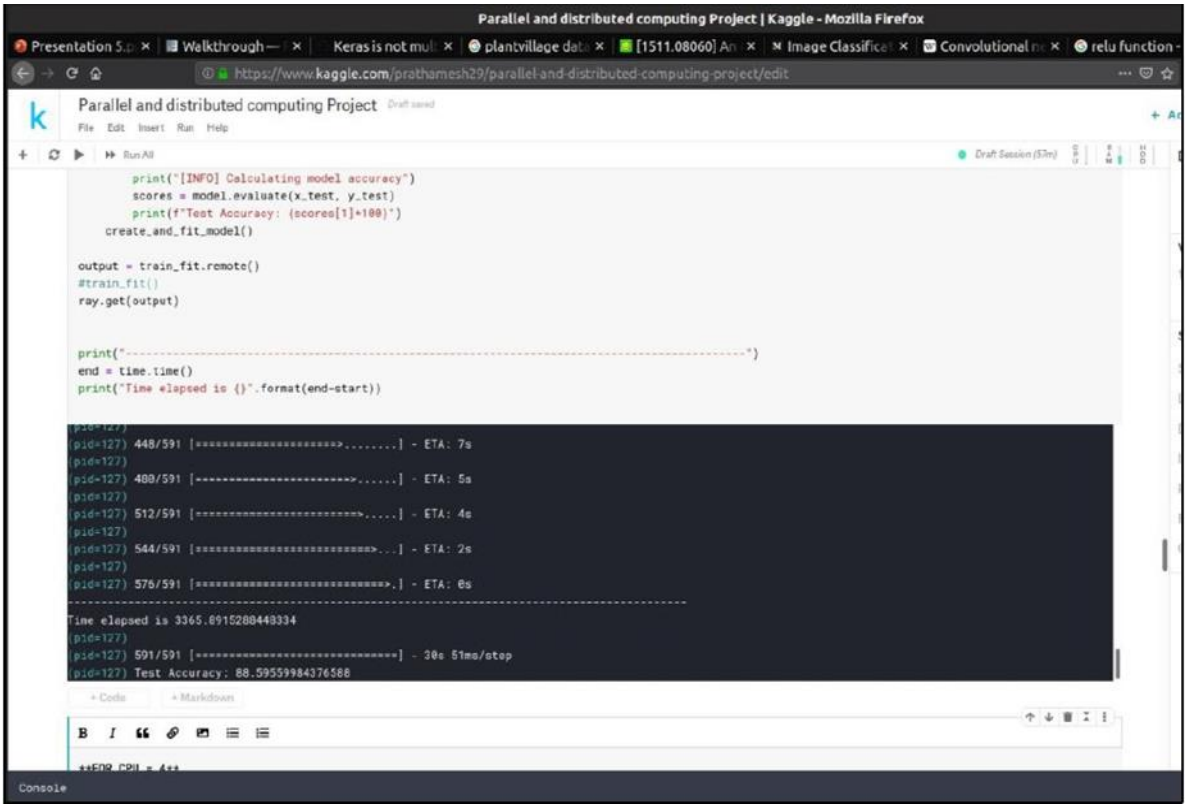
After parallelizing GPU using ray.init, we got an accuracy of 94.95%.

Following were the execution time of all the epochs.

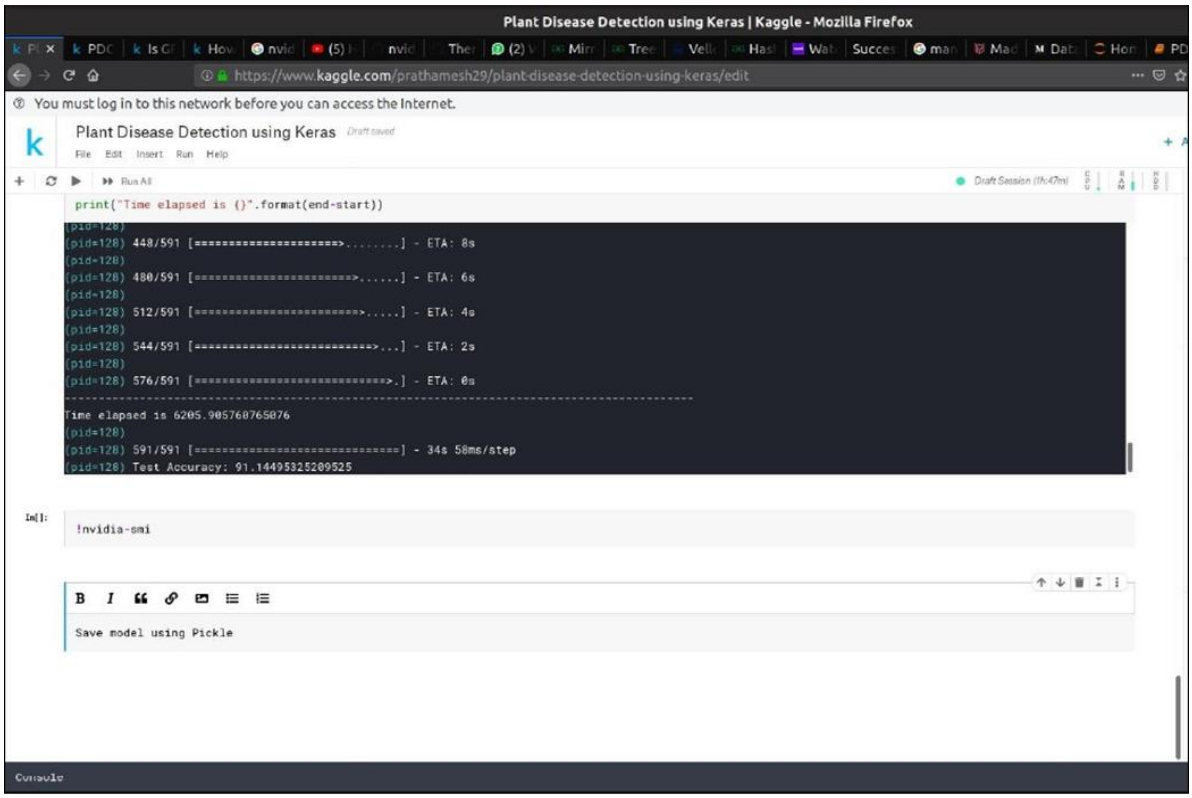
Epoch sequence number	Execution time (in seconds)
1	50
2	46
3	47
4	46
5	46

RESULTS AND DISCUSSION

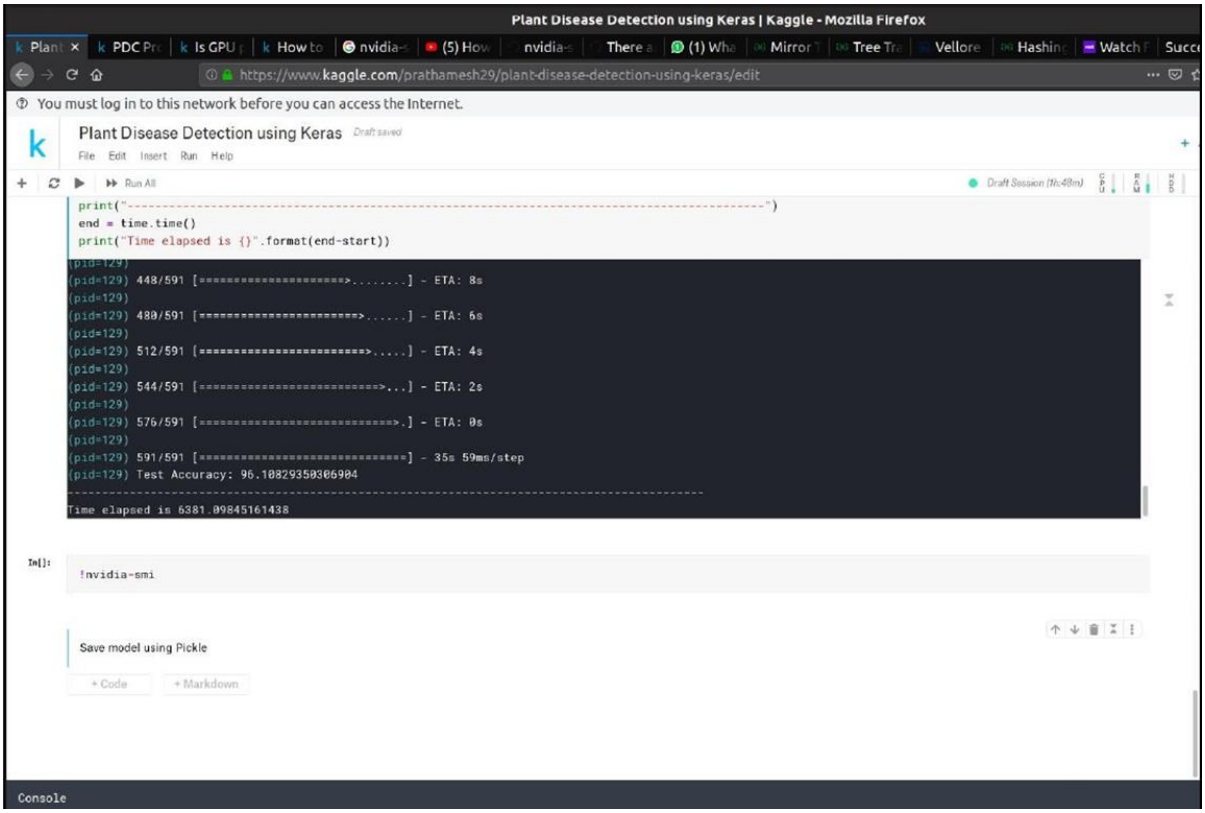
SERIAL PYTHON



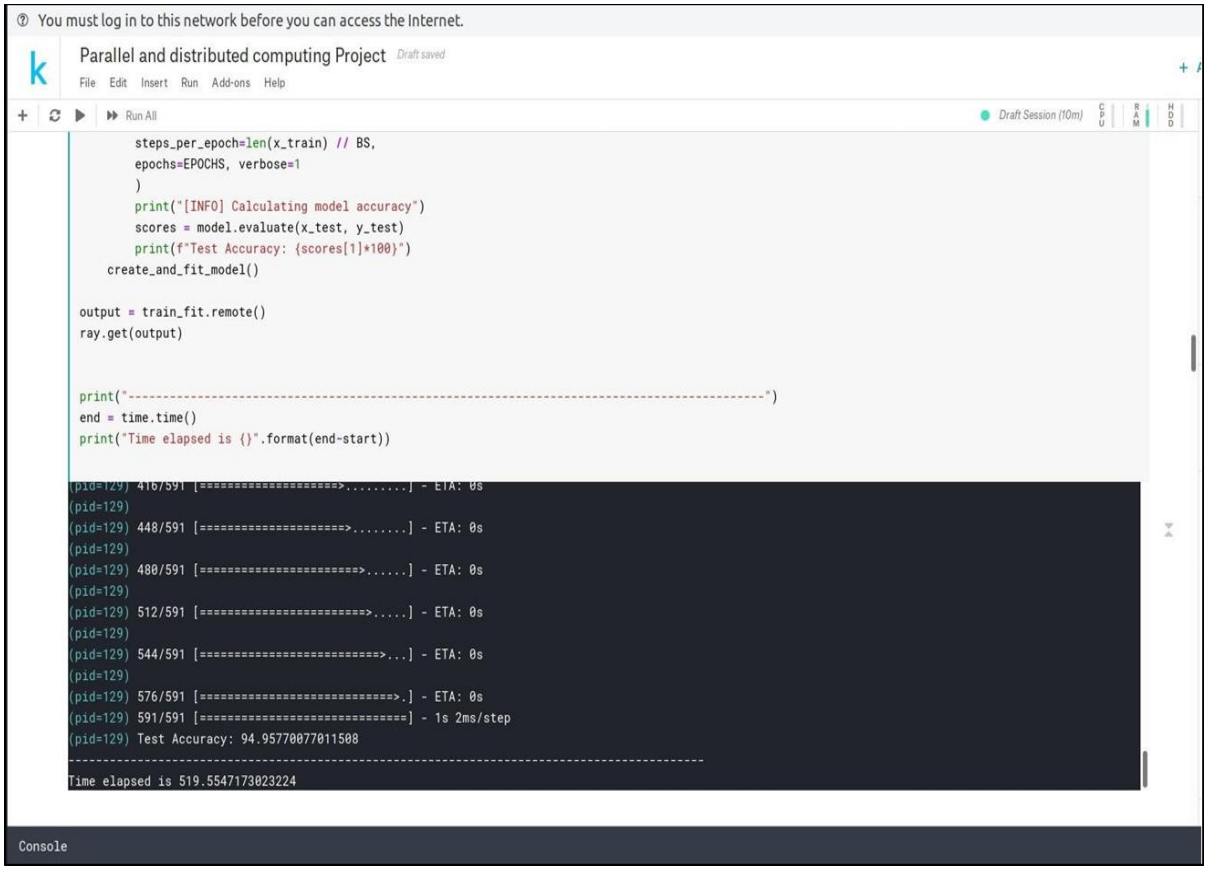
3 CORE CPU



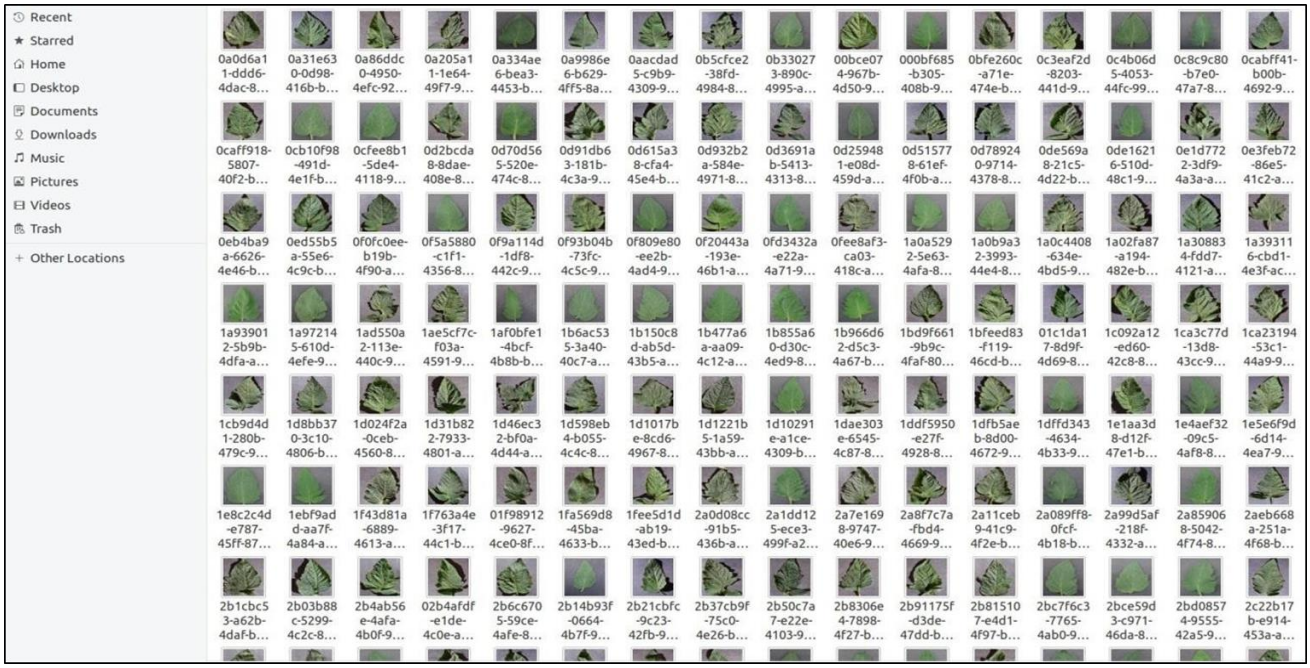
4 CORE CPU



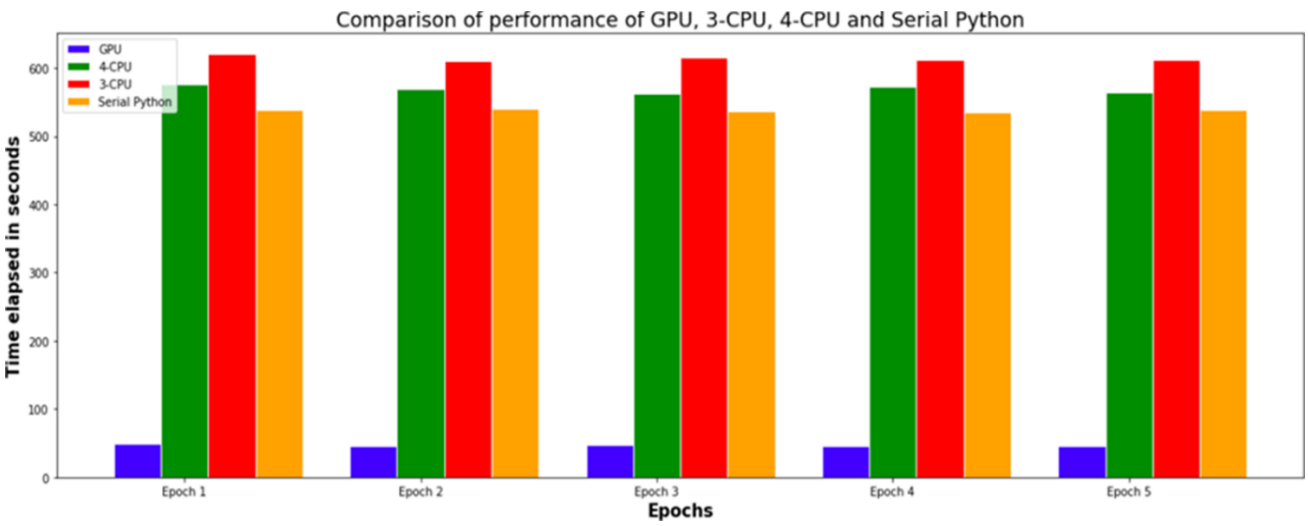
GPU



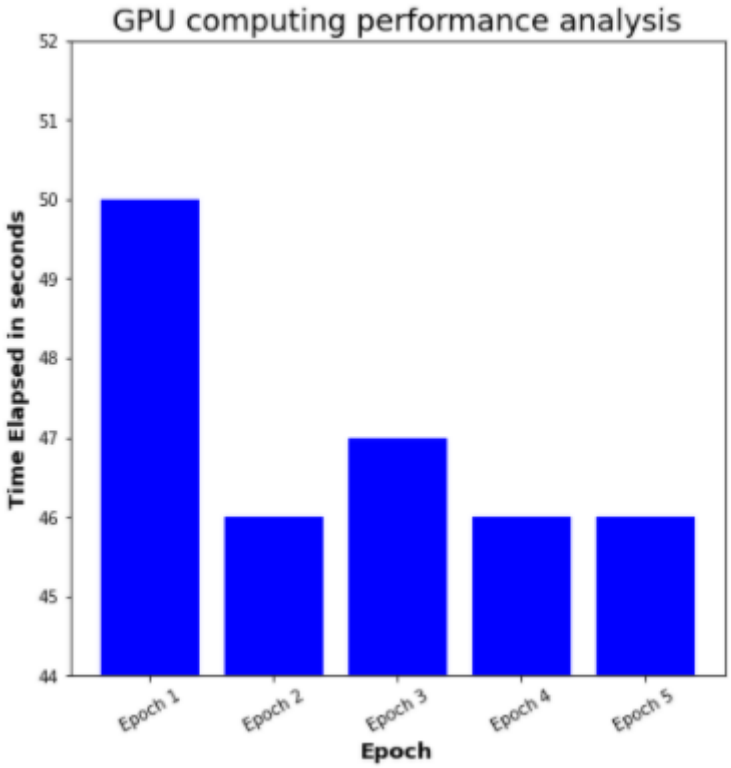
POTATO EARLY BLIGHT



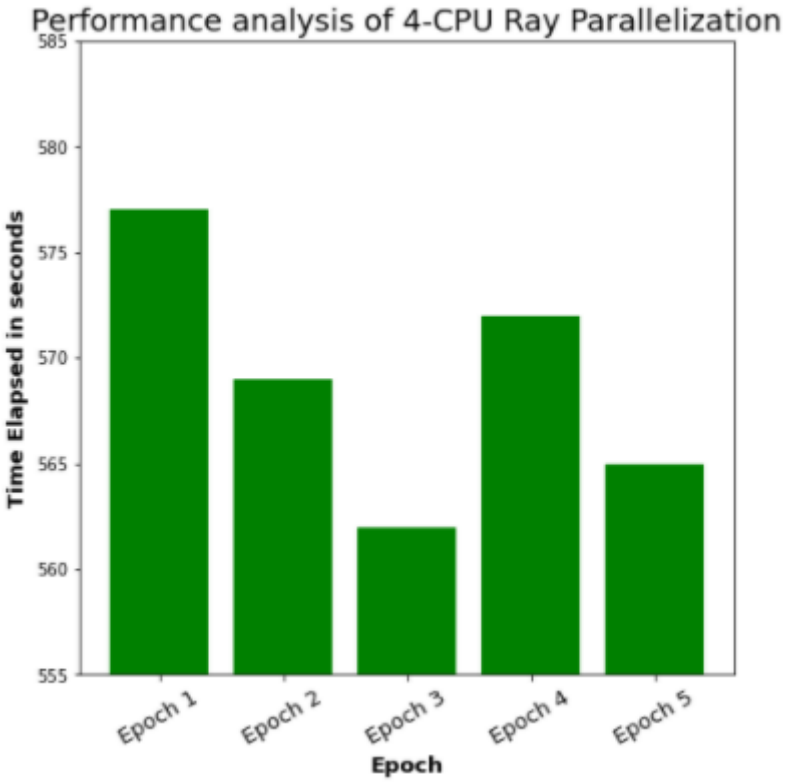
PERFORMANCE COMPARISON



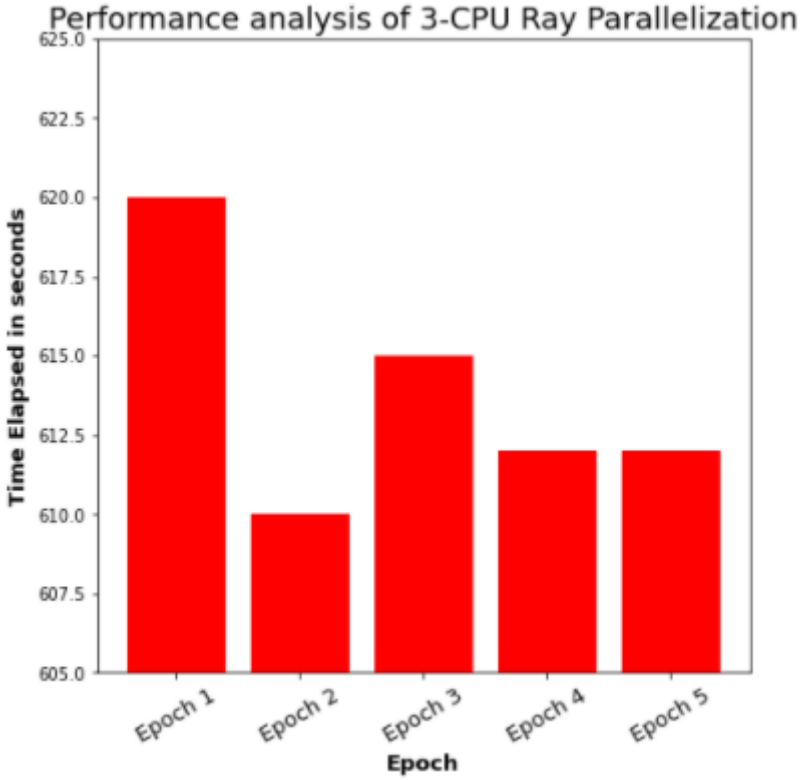
GPU COMPUTING PERFORMANCE ANALYSIS



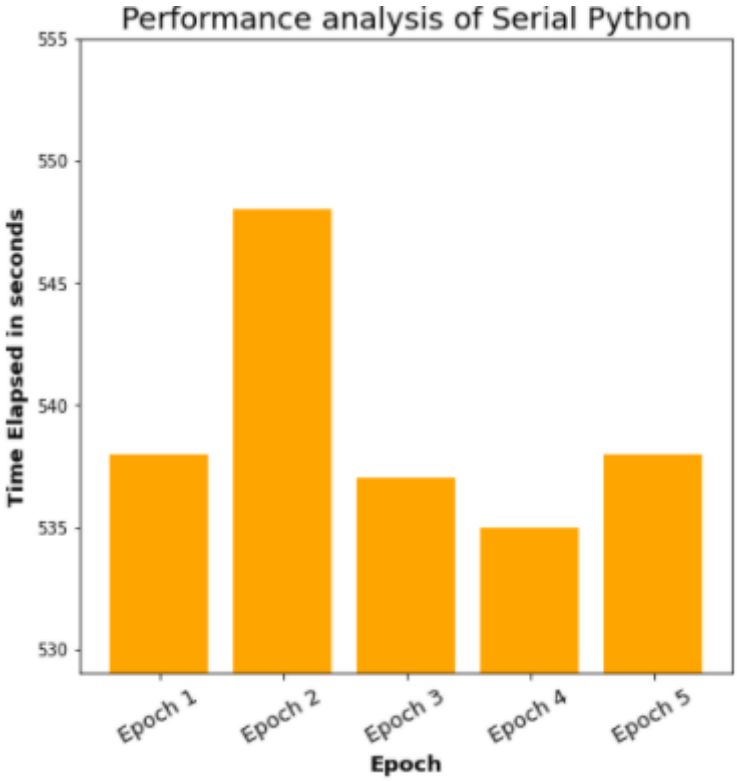
4-CPU COMPUTING PERFORMANCE ANALYSIS



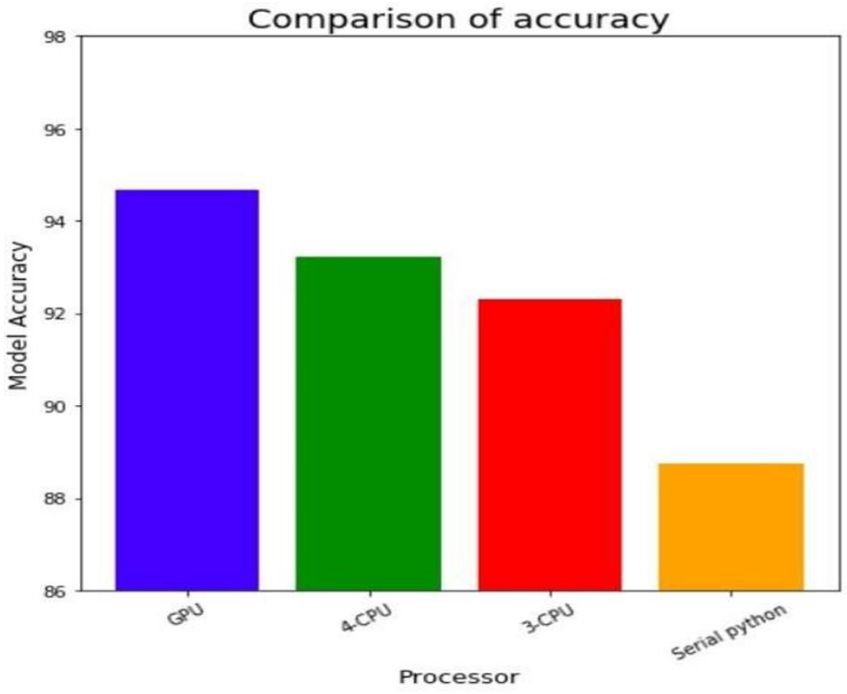
2-CPU COMPUTING PERFORMANCE ANALYSIS



SERIAL COMPUTING PERFORMANCE ANALYSIS



COMPARISON OF ACCURACY



CONCLUSION

Training an image classification Convolutional Neural Network on GPU not only decreased the computation time but also increased the accuracy of the model compared to parallelized CPUs. Our results show that the GPU implementation achieves 80% speedup compared to the CPU implementation. Most of the existing studies show that the GPU implementation of deep learning model such as the CNN can yield significant speedup compared to CPUs. Due to highly parallel architecture, it helps GPU compute vector operations, matrix operations and image processing faster than CPU. The latency is overridden by the number of small cores present in the GPU. In summary, the results of the experiment showed that the computation time taken by the GPU about 80% less than the time taken by CPU to train the model. CPU is very efficient in handling small data compared to GPU. A traditional CPU might outperform GPU in some aspects, so we should not blindly choose for any computational applications.

REFERENCES

1. “Image Processing Techniques for Detection of Leaf Disease” by Arti N. Rathod, Bhavesh Tanawal, Vatsal Shah.

2. “An Introduction to Convolutional Neural Networks” by Keiron Teilo O'Shea.

3. “Performance Analysis of GPU V/S CPU for Image Processing Applications” by B. N. Manjunatha Reddy, Dr. Shanthala S. , Dr. B. R. VijayaKumar

4. “A Comparative evaluation of the GPU vs. the CPU for Parallelization of Evaluation Algorithms through multiple independent runs” by Anna Syberfeldt and Tom Ekblom.

5. “Ray: A Distributed Framework for Emerging AI Applications” by Philipp Moritz , Robert Nishihara

6. “Image Processing Application on Graphics processors” by Chouchene Marwa, Bahri Haythem, Sayadi Fatma Ezahra, Atri Mohamed

7. “Real-Time Image Processing Applications on Multicore CPUs and GPGPU” by R. Samet, O.F. Bay, S. Aydın, S. Tural, A. Bayram