Benchmarking image processing tasks using Apache Spark

Syed Obaid Dawarki[1] *, **Abhishek Saini**[1] *

[1]University of Washington, Seattle

Abstract

In this project presents a benchmarking study of image processing on Apache Spark. The study focuses on the performance of image processing algorithms on Apache Spark, and compares the run time as the number of workers are increased in the cluster. The datasets used in the study includes COCO dataset. The algorithms tested include gray scale conversion of RGB images, followed by taking Laplace transform of gray scale images, data augmentation and CNN (ResNet-18). The results of the study show that Apache Spark is a suitable platform for image processing and makes the computations very fast.

Introduction

Spark is a distributed computing platforms (Zaharia et al. 2010). It is used to process and analyze large data sets and can be used in a variety of applications, such as machine learning, stream processing, graph processing, and image processing. It is based on the MapReduce model and can be used to process large data sets in a parallel and distributed manner.

One of the most popular cloud environments for Spark is Microsoft Azure Databricks which is an analytics platform designed to make working with data simple and easy. It provides a unified data analytics platform, composed of Apache Spark, that enables us to be more productive when working with data. The platform includes all of the tools we need to be successful with data, including a powerful and easy-to-use data processing engine, a rich set of libraries for data analysis and machine learning, a simple and efficient way to manage data, and a collaborative environment that makes working with data easy and fast.

Image Processing

Image processing is a subset of computer vision that deals with the manipulation and analysis of digital images. It is used in a variety of fields, from medicine to manufacturing, and has seen a surge in popularity in recent years due to the rapid advances in machine learning and artificial intelligence.

Image processing is often used to improve the quality of digital images, remove blur or noise, or extract useful information from images. With the rise of deep learning, image processing has become an important part of the data generation process for many machine learning applications. For example, image processing can be used to pre-process images before training a convolutional neural network (CNN).

Most image processing tasks involve working on one single image at a time. Doing such a computation on a single CPU can be very time-consuming since the dataset size in modern deep learning applications can be huge. For example, the ImageNet dataset has around 14 million images. So using multiple CPUs in parallel can help speed things up—and that's where Spark comes into the picture. By using Spark, we can process images in parallel on a cluster of machines, which can help us achieve much better performance. Additionally, Spark also provides a Python interface called PySpark making it easy to implement.

Evaluated Systems

The system we are evaluating is Spark. There are multiple ways to install and setup Spark. One can install Spark in a standalone format on a single instance or on multiple instances of a self-hosted cluster. The easiest way to install Spark quickly on a cluster is to use a cloud service platform like AWS or Azure. We will be using Microsoft Azure Databricks, which is a managed Spark platform that provides all of the tools we need to analyze data at scale.

Architecture

Apache Spark architecture is composed of three main components:

- **Driver**: The driver is the main process that runs the user application and is responsible for creating the SparkContext, which is the entry point for the application. The driver also communicates with the cluster manager to request resources for the application.
- **Cluster Manager**: The cluster manager is responsible for allocating resources to the application and managing the cluster. It can be either a standalone cluster manager such as Apache Mesos or YARN, or a cloud-based cluster manager such as Amazon EC2 or Google Compute Engine.

*These authors contributed equally.

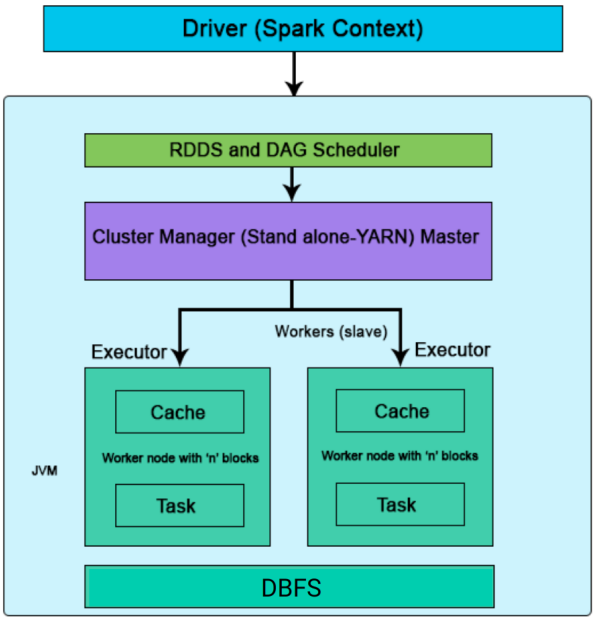

Figure 1: Standalone mode of Apache Spark Architecture (Pedamkar 2022)

- **Executors**: The executors are the processes that run the actual tasks of the application. They are responsible for executing the user code and returning the results to the driver. The executors are managed by the cluster manager and can be deployed on multiple machines.

Next, we introduce some data storage abstractions that we will use for this project.

Databricks File System
Databricks File System (DBFS) is a distributed file system mounted into a Databricks workspace and available on Databricks clusters. It provides a platform for storing and sharing data within a Databricks workspace. The DBFS root is the default storage location for an Azure Databricks workspace, provisioned as part of workspace creation in the cloud account containing the Azure Databricks workspace. We can access DBFS root as if it were a local file system. It also provides a platform for sharing data between users and teams. DBFS is integrated with Apache Spark, allowing users to access data stored in DBFS using Spark APIs.

Resilient Distributed Dataset
A Resilient Distributed Dataset (RDD) is a fundamental data structure of Apache Spark, a distributed computing framework. It is a collection of elements partitioned across the nodes of a cluster that can be operated on in parallel. RDDs are immutable, meaning that once created, their values cannot be changed. This makes them fault-tolerant, as any failed operation can be re-executed on a different node without affecting the original data.

RDDs are created by transforming existing datasets, such as text files, databases, or other RDDs. This transformation process is known as a transformation operation. Common transformation operations include map, filter, and reduce. These operations allow users to manipulate data in a distributed manner, allowing for faster processing of large datasets. Further, we can partition our data across different machines and set the number of partitions that we want to work with in our application.

Problem Statements & Methods

Problem statement

We are evaluating Spark on Image processing tasks to understand the effect of the increase in the number of workers on the run time to perform image processing tasks. We consider three types of task of increasing complexity and evaluate performance of Spark on each of these three types of task.

Dataset used

We will use the COCOVal-2014 dataset for all our experiments. COCO is a large-scale object detection, segmentation, and captioning dataset (Lin et al. 2014). It consists of 40504 images of different dimensions. We will only use the images and not the labels since we are only interested in image processing tasks. The size of the compressed images is 6 GB. We followed the steps that included creating a compute node in Azure Databricks, downloading the compressed data locally on the compute node and unzipping the files onto the DBFS (Databricks File System) so as to load the data into the Spark environment.

Initially we tried loading the data from the compute node itself but we realized that the data was not accessible to the worker nodes. The recommended way was to load the data into the DBFS first.

Methods

As a part of our experimentation and methods, our focus is on three main types of image processing tasks:

- Simple transformation containing grayscale followed by Laplacain transform.
- Pipeline containing a series of transformations
- Feature extraction using a convolutional neural network

Our approach involves reading the image as a binary file in a spark data frame with image paths as one of the columns. Following that we convert our spark data frame into RDD and using the map function we apply the image transformation technique to each image. The various image transformation techniques that are included in our experiments are mentioned below:

- **Transformation:** This step consists of the grayscale conversion of an image followed by Laplacian transform of gray scale image. Grayscaleis the process of converting an image from a color format to a grayscale format. The most common grayscale conversion is the conversion of an RGB image to a grayscale image. Next, we perform the Laplacian transform on this grayscale image. We use the cv2 library to perform this task and note down the run time as we increase the number of workers in our cluster (Bradski 2000).

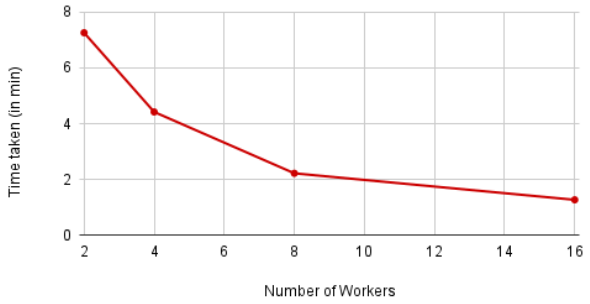

Figure 2: Time vs number of Workers - Gray Scale plus Laplace Transform

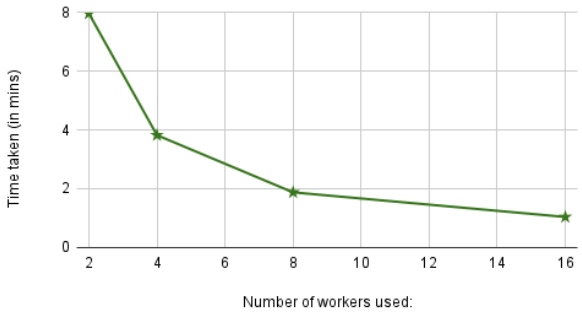

Figure 3: Time vs number of Workers - Data Augmentation

- **Data Augmentation Pipeline:** This pipeline containes a set of transformations that are common for input preprocessing of images for downstream ML tasks or as image augmentation for deep learning. We implement a series of such transformations using the albumenations library.(Buslaev et al. 2018) Albumentations is a Python library designed specifically for image augmentation. The library contains more than 70 different augmentations to generate new training samples from the existing data. The pipeline we have used for a hypothetical image augmentation process which includes - Resizing of image, Rotation of image, Flipping of an image, Cropping, Gaussian Blur, RGB Shift, ISO Noise, Random Brightness, Random Contrast and HueSaturationValue
- **Feature extraction using a CNN:** These are pretrained on large datasets like ImageNet are rich feature extractors. These features are usually extracted from the layers before output heads and can be used for dimensionality reduction for downstream ML applications. We use a pretrained Resnet-18 downloaded from the PyTorch library as the CNN for this step (Paszke et al. 2019).
  Additionally, deep learning libraries also support batching multiple inferences in a single call to the model. This is usually more efficient than doing inference one image at a time due to the inherent parallelization support offered. Further, we also experiment with the number of partitions since partitions allow data to be stored across multiple machines and can thus decrease the overall execution time.
  Each of the above tasks was benchmarked on 2, 4, 8, and 16 workers, and the execution times were recorded. Our code is available online here https://github.com/obaiddawarki/Scalable_Systems_Project

Results

The result for the first experiment - Grayscale and Laplacian transformation is shown in Figure 2. Increasing the number of workers reduced the execution time of all the tasks. Also, the reduction in run time wasn't linear which suggests that there is a point at which the additional workers do not significantly improve the run time. This trend can also be seen in the second and third experiments in Figure 3 and Figure

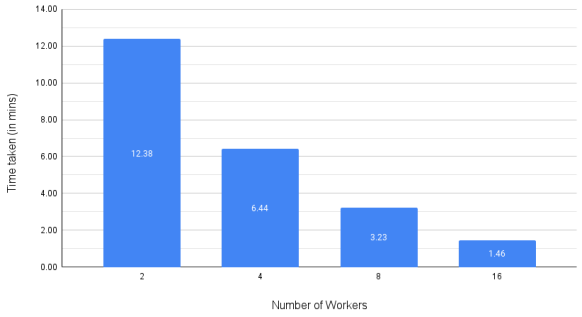

Figure 4: Time taken(in mins) vs no. of workers - number of partitions = 4 x no. of cores, batchsize=16

4 where the run time is highest when only two workers were used and then it decreases with the addition of more workers until it reaches 16 workers where the run time is the lowest.

For the third experiment, on feature extraction using CNN, as seen in Figure 5, the performance initially improves as we increase the batch size from 4 to 16 but the performance slowly starts to degrade and at batch size = 256 we could have gotten better performance using fewer nodes (albeit for optimal batch size and number of partitions). This suggests the importance of selecting the correct batch size for this task. Similarly, the performance degrades signifi-

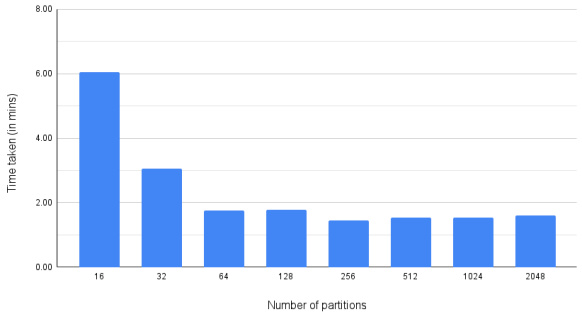

Figure 5: Time taken(in mins) vs no. of partitions - number of workers = 16 (64 cores), batchsize=16

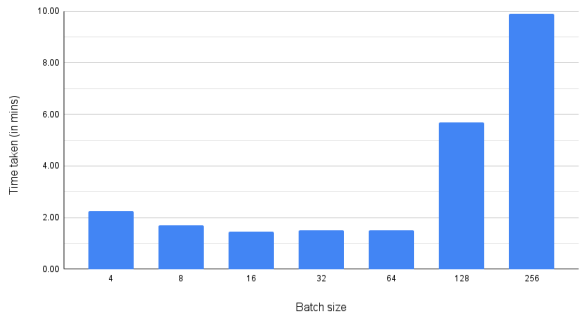

Figure 6: Time taken(in mins) vs batchsize
- number of workers = 16 (64 cores), no. of partitions =256

cantly if the RDD isn't partitioned across sufficient nodes as seen in Figure 6.

Conclusion

With the help of image processing tasks which included grayscale conversion followed by Laplacian, data augmentation, and feature extraction, we explored the potential of Apache Spark for distributed image processing tasks. We demonstrate that Spark can be used to parallelize image processing tasks, allowing for faster image processing. Also with the increase in the number of workers, we observed that the run-time execution decreases. Furthermore, we discussed the importance of getting the correct batch size and number of partitions for optimal performance for deep learning feature extraction/inference. Our results show that Spark can be a powerful tool for distributed image processing tasks, providing faster and more efficient results.

Our code is available online here https://github.com/obaiddawarki/Scalable_Systems_Project

References

Bradski, G. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Buslaev, A. V.; Parinov, A.; Khvedchenya, E.; Iglovikov, V. I.; and Kalinin, A. A. 2018. Albumentations: fast and flexible image augmentations. *CoRR*, abs/1809.06839.

Lin, T.-Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; and Zitnick, C. L. 2014. Microsoft COCO: Common Objects in Context. In Fleet, D.; Pajdla, T.; Schiele, B.; and Tuytelaars, T., eds., *Computer Vision – ECCV 2014*, 740–755. Cham: Springer International Publishing. ISBN 978-3-319-10602-1.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Köpf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY, USA: Curran Associates Inc.

Pedamkar, P. 2022.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; and Stoica, I. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 10. USA: USENIX Association.