Parallelization of Image Classification using Convolutional Neural Networks

Manahil Aamir (24441), Humera Raheel (24446), Noor us Sabah (25173)

Department of Computer Science, Institute of Business Administration

CSE 467: Parallel and Distributed Computing

Sir Muhammad Zain Uddin

Introduction

Over the past few years there has been a tremendous advancement in computer vision with regard to the extraction, analysis and comprehension of image data. In this evolution Convolutional Neural Networks (CNNs) are at the forefront as a key technique for tasks like object detection, recognition, and image classification. However, given the large volume and complexity of images and with the growing number of datasets it has become necessary to increase the computational effectiveness of current approaches.

The goal of this research project is to parallelize image classification algorithms to meet this demand, which will reduce training time and streamline processing.

In order to uncover new possibilities for fast and scalable image classification, we aim to take advantage of parallel computing. Digital imaging is widely used in many areas, such as healthcare and driverless cars which shows the importance of quick and accurate picture analysis. Although sequential processing techniques have proven effective, they are often limited by the volume and complexity of modern image datasets. Parallelization is a useful technique for resolving this problem by distributing computational tasks among multiple processors.

The current computing environment has become very advanced with a large volume of data pouring in. Therefore, parallelization has become extremely important to satisfy the increasing need for faster and more effective data processing. It is possible to utilize the combined power of multiple processors or computational resources to perform calculations in parallel for image classification algorithms. Processing large amounts of image data quickly is one of the promises of this approach which can ultimately result in real-time performance and improved efficiency overall.

Image classification is one area where parallelization is especially necessary as the size of datasets and complexity of models is increasing at a very fast rate. Traditional sequential algorithms cannot handle these kinds of large-scale operations, which leads to poor performance and longer processing times. Therefore, by dividing the computational load among several processors, parallelization offers a practical solution that facilitates the economical use of available resources and reduces processing time.

The main aim of this research is to develop parallelizing sequential image classifications, which are now commonly applied. By using parallelization techniques, we want to accomplish the following objectives:

1. Advanced computational efficiency: Parallelization will grant more power and the opportunity to use multiple processors in sequence, which in turn will reduce the training period and boost performance considerably. This enhancement will be of good help to researchers and practitioners in the efficient processing of large datasets and in the speeding of their model iterations.

2. Scalability: The code written in parallel will possess the ability to optimally utilize available computing resources, making it flexible in the majority of situations to undertake complex tasks such as the classification of large datasets. However, this adaptability is surely crucial for the improvement of the system so it can easily meet the needs of such a changing image classification application in the near future.

3. Enhanced Accuracy: Through the reduction of training time, we are able to experiment with different model architectures, hyperparameters, and optimization techniques, which in turn will lead to an increase in the classification accuracy. Among that capacity is the ability to model more efficiently and check many more possibilities, which will generate more robust and correct models.

4. Real-time Performance: Alongside the parallelization ability, the system will be able to respond promptly, and it will suit systems that need real-time decision-making. This feature becomes vital in areas where very fast and accurate image classification is needed, for example, in driverless cars, surveillance systems, and medical imaging.

5. Reduction of Training Time: Using the parallel operation, we can considerably reduce the sum of the computational burdens of training CNN algorithms for image classification tasks. We aim to compress the training period with the help of the underlined capacities of modern technology and hardware, which will lead to the models transitioning from the convergency point quickly and easily.

6. Utilization of Modern Computing Hardware: With the utilization of the latest hardware configurations, i.e., GPUs and TPUs, we can intensify the training process of the model, and there will be an accuracy improvement in the association process. We anticipate that we can get the highest efficiency and throughput of the CNN model by using parallel processing on this specialized hardware.

We suggest the use of distributed computing to implement the parallelization of the image classification methods. Through this method, the computational task is segmented in such a way that individual processors can run multiple jobs simultaneously. The phases that outline our suggested approach are as follows:

1. Data Partitioning: By dividing the image dataset into smaller subsets, each processor will have the liberty to work on one particular region on its own. Partitioning will be managed in such a way that the images are distributed in a balanced manner among the processors. A number of ways, like replication and data sharding, are examples of how one can handle a large load without downtime.

2. Parallel Model Training: After the partitioning of the entire data set, each processor will pick and train an individual copy of the CNN model using the selected subset of images. Weight adjustments, compliances, and forward and backward propagation must all be carried out simultaneously for this task. These tasks can be done in parallel by using parallel computing frameworks such as TensorFlow or PyTorch.

3. Communication and Synchronization: The synchronization and communication between processors are extremely important for exchanging model parameters, weight updates, and guaranteeing consistent training of all the processes, so shared communication will be an important point. The message-passing interfaces and the parameter server are just two of the many techniques that can be applied to achieve this goal.

4. Model Fusion: The models will be trained within each processor separately, after which the fusion process, which includes the joining of learned features and weights from each processor, will be done. At the end of this development, indeed, an optimized model that can function excellently for targeted image

categorization will be generated. Ensemble learning or model averaging techniques can be applied to combine the results of different models.

5. Evaluation: Utilize a set of measures that comprise accuracy, precision, recall, and F1-score and analyze the scalability of the model through parallelized CNN. For measuring the difference achievable in performance via parallelization, reported results should be compared with those of the baseline model, which was trained on a single GPU or a CPU.

Literature Review

Parallelization of convolutional neural networks (CNNs) is being done in order to enhance the effectiveness and efficiency of a variety of tasks such as object detection, image categorization, and structural damage assessment. This is an important area of research. In this literature review, we analyze a number of important past studies that studied novel methods for CNN parallelization by using different hardware architectures and optimization methodologies.

Girshick et al. (2014) discussed the steps taken by a large CNN model in order to compute the features, which are then classified with class-specific linear SVMs. Through the application of high-capacity CNNs on bottom-up region recommendations, this technology has greatly raised the object identification performance, which in turn has made object segmentation and localization much easier. Besides, the paper also discusses a training mode for large CNNs in cases where there is a lack of labeled training data, where pre-training the CNN on a single core is more efficient and quicker than training it on multiple cores. (1).

In order to enhance image classification, Yan et al. (2015) developed the Hierarchical Deep Convolutional Neural Network (HD-CNN), which is a scalable and parallelizable design. HD-CNN uses coarse category classifiers for the simple classes and fine category classifiers for the harder ones, thus it gets the two-level hierarchy. For large datasets like CIFAR100 and ImageNet, HD-CNN manages scalability by using component-wise pretraining and global fine-tuning. (2)

Additionally, researchers have been using synchronous Stochastic Gradient Descent (SGD) techniques to parallelize CNN training across several CPU units. The challenge of finding a middle ground between convergence precision and batch size is still present. The conventional methods that we are familiar with have scaling problems and they usually use small batch sizes to make sure that the convergence is precise.

New studies have provided the answer to this complex problem, and the work of Sunwoo Lee et al. (3) is a good example of it. The paper outlines a flexible way to change the learning rate and mini-batch size on the go, thus enabling the parallelization of the tasks while maintaining the accuracy levels comparable to those of the smaller batch sizes.

Martinez et al. in their thorough research, examined the intricate connections that take place between the high-performance GPUs and convolutional neural networks (CNNs). Their results showed that GPUs are used for CNN inference jobs in a variety of fields. They gave useful information that could be used to make deep learning applications in an environmentally friendly way by pointing out the patterns of energy consumption. [6]

A more efficient method for employing GPUs to parallelize CNN inference is put forward by the authors, among them Bonvallet. The strategy works directly with the input data and puts the stress on the coalesced memory accesses. The experiments demonstrate that the results are significant when compared to the present ones. Real-time applications such as autonomous cars and image processing are based on this kind of work. (4)

In the training of convolutional neural networks (CNN) on heterogeneous CPU/GPU architectures, Marques, Falcao, and Alexandre proposed a new method of parallelization. They claim that the existing methods do not fully utilize the parallelization abilities CNNs, and different hardware configurations offer. Consequently, their technique is a new type of model parallelism which concentrates on the distribution of the convolutional layer that usually requires 60-90% of the total processing time. It proves effectiveness without losing classification accuracy and at the same time, it shortens the training time drastically. To illustrate this, the CIFAR-10 dataset with a CNN that has two convolutional layers can experience speedups of 3. 28× and 2. 45x were attained with the help of four CPUs and three GPUs, respectively. (10)

Convolutional Neural Networks (CNNs) are sped up by NVIDIA GPUs by utilizing parallelism and efficient cluster communication. Among the many, the Heterogeneous Computing Group at Baidu, for instance, widened the CNN designs to 36 servers, each with four NVIDIA Tesla K40m GPUs. They applied data parallelism techniques such as butterfly synchronization and lazy updating, which are the cornerstones of computation and communication overlap, and model-data parallelism. A huge cut in training time without losing classification performance is what this method shows on how NVIDIA GPUs can accelerate the large-scale CNNs. [11]

To address the problems of scalable and quicker image classification methods, Hernández et al. (2018) proposed parallelizing the artificial visual cortex (AVC) model for object classification and using CUDA for GPU acceleration. The process was divided into phases, where the production of the descriptor vectors was done with the images being changed at each level. The goal of parallelization was to use the GPU's huge parallelism to compute the functions inside the stages concurrently, which consequently led to the speedup of up to 90 times faster than the original system. [16]

The authors, Hailesellasie and Hasan (2017), in their study of FPGA platforms, proposed an efficient convolutional layer implementation of the convolutional neural networks (CNNs) for the classification of images. Through the authors' innovative design, asynchronous memory access is enabled which in turn dramatically cuts the computing time by means of distributing the input images through an array of pseudo parallel memories. This strategy enables the fast execution of the convolutional operations by doing the computer work and the resource management at the same time. The proposed architecture deeply diminishes the number of clock cycles required for convolution by distributing the input data over several memories and reading them at the same time. [15]

The "Speculative Backpropagation for CNN Parallel Training" method was the brainchild of Park and Suh [1], who came up with it in the pursuit of better image recognition using the latest CNN parallel training techniques. Through speculative backpropagation and tackling the sequentiality of gradient descent, this technique tries to develop simultaneous training. This study is significant in machine learning and computer science since it presents a new approach to training CNNs, which can be of great use in jobs that require image recognition, thus improving their effectiveness and efficiency. By utilizing the speculative backpropagation approach, the sequential property of the gradient descent algorithm is demolished; hence, faster and more sophisticated CNN training procedures are developed. The Park and Suh approach can be a game-changer for CNN training and, at the same time, will be a step forward in the development of image recognition technologies. The ongoing research on the described approach might advance the performance of CNNs in diverse fields like computer vision and pattern recognition (5).

Ren et al. (2017) introduced a new CNN design that is a combination of a Back-Propagation Neural Network (BPNN) and parallel CNNs. The main goal of this architecture is to enhance feature extraction and to simplify computation by using parallel CNNs with different layer levels. From LeNet5 to the modern architectures like AlexNet, ZFNet, GoogLeNet, VGGNet and ResNet, it illustrates the development of CNN. (8)

The Bi-layered Parallel Training (BPT-CNN) model was created by Chen and his colleagues to speed up the training of huge convolutional neural networks (CNNs) in distributed computing settings. The BPT-CNN, a two-pronged parallelization technique, is the solution to this problem. First of all, it employs outer-layer parallelism to train several CNN subnetworks on different portions of data; hence, it is able to tackle the issues of task imbalance, the synchronization of the tasks, and the transmission of the data. Besides, the result of the computation steps and the local weight training can be done by different machines due to task parallelism, which is inner-layer parallelism, and the process of training subnetworks one by one on each machine is sped up. Hence, the architecture is designed in such a way that it combines techniques like Asynchronous Global Weight update (AGWU) and Incremental Data Partitioning and Allocation (IDPA) in order to achieve workload balance and have fewer synchronization overheads. (7)

The design of BranchyNet by Teerapittayanon et al. (2016) is a product of the initiatives to accelerate the Convolutional Neural Network (CNN) inference. This enables some test samples to leave the network early through extra-side branch classifiers, which, therefore, reduces the problem of the higher latency and energy consumption in deep networks. Through this ground-breaking architecture, the fact that features that have been acquired at the previous layers of a deep network are often sufficient for the classification of a lot of data points is utilized. As for most of the samples, BranchyNet decreases the inference runtime and the energy consumption by stopping the samples with high-confidence predictions early on. (9).

The paper "Parallel convolutional neural network toward high efficiency and robust structural damage identification" recommended the use of a parallel convolutional neural network (P-CNN) for the efficient and reliable identification of structural damage. The computer vision algorithm in this work was used to extract the multidimensional features by combining 1D-CNN and 2D-CNN. The P-CNN model aims to enhance the current structural damage recognition methods' accuracy and efficiency through the use of parallel processing in cases where noise-contaminated data is present. The efficient feature extraction from multiple dimensions is enabled by the parallel implementation, which in turn increases the model's potential to identify and classify structural deterioration. (12)

The researchers, Huang et al. (2022), present a new method to the ship image classification (called "Fine-Grained Ship Classification by Combining CNN and Swin Transformer") that is the combination of CNN and Swin Transformer. The study shows how important it is to consider a number of scale parameters when classifying and detecting ship images. It is the first instance of the transformer architecture with self- attention being used for categorization and detection. In order to successfully extract the features from ship photos, a parallel network which contains a CNN and a transformer is used in the suggested technique. Through the combination of CNN and transformer architectures for feature extraction and classification, this parallel approach boosts the efficiency and accuracy of ship classification tasks. [13]

A parallel deep convolutional neural network (PDCNN) architecture was the outcome of Abd-Ellah et al. (2019) for the purpose of the detection and classification of gliomas from brain MRI data. Hence, the PDCNN structure is composed of two parallel roads, local and global, and their common task is to extract both local and global features from the input images. Seven convolutional layers with very small filter sizes constitute the local path; on the other hand, the larger filter sizes form the global path. Each convolutional layer's output is either reduced by downsampling using the max pooling layers or the pathways are combined before they get to the final output layer. The parallel architecture makes feature extraction and classification possible, which in turn also reduces the processing time and improves the glioma detection accuracy. (14)

This literature analysis wraps up with an emphasis on the pathbreaking effects of parallelization methods on convolutional neural networks (CNNs) in computer vision research. By means of parallel computing, researchers have set up distinctive architectures that are adapted to certain workloads, optimized training and inference procedures, and solved scaling problems. Parallelization has become the key tool for the enhancement of CNN efficiency in different fields of application, from the development of efficient GPUs to the design of cutting-edge parallel CNN architectures for structural damage identification and fine-grained ship classification systems. Parallel computing will create more innovations and speed up the development of computer vision and deep learning.

Methodology

Github Link

https://github.com/Manahil-Aamir/Parallelization-of-Image-Classification-using-Convolutional-Neural-Networks

Sequential Implementation

Architecture and Logic

- Data Preprocessing:

For our image classification problem, we chose CIFAR-10 dataset, which is famous for image classification related AI models. Total number of classes in this data are 4 and it is a data of 50,000 training and 10,000 test images. This dataset is available, and ca be loaded from TensorFlow. Initially, the shape of this data is examined along, and the classification labels are also stored in a 1D array.

- Model Definition:

- **MiniXception CNN Model:** For extraction of features from images to divide in ddifferent categories, MiniXception CNN model is used.

- **Architecture Components:** Convolutional layers, batch normalization and ReLU activation are used to define entry, middle, and exit flows.

- **Global Average Pooling:** For feature aggregation in the end,global average pooling has been used.

- Training:

  - **Model Compilation:** Adam Optimizer is a common algorithm that is used to compile this model. It is mostly used with classification problems like in our case.

  - **Training Process:** To ensure maximum accuracy, the model and best performance, the model has been trained for 10 epochs. Along with the training process, the weights of the model are shifted to ensure minimum loss.

  - **Evaluation:** Lastly, a few images form test data are used to see results of the model's performance on different data.

Key Functions/Modules:

1. **TensorFlow/Keras:** These libraries aid in neural network training, thus are necessary to train the MiniXception model.

2. **Matplotlib:** Used on the side to plot graphs.

3. **NumPy:** Essential library that is used for handling array related purposes.

Explanation of Algorithms/Techniques:

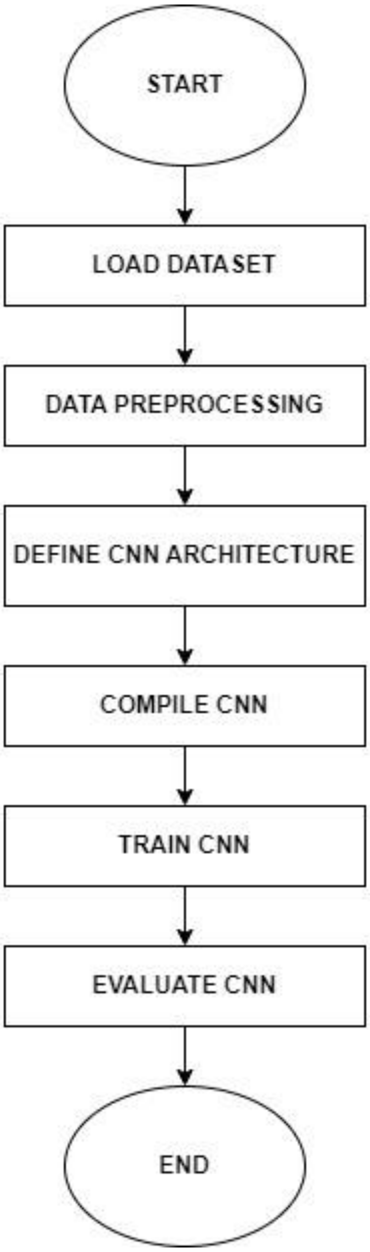
1. **Convolutional Neural Networks (CNNs):** Used for image detection purposes.

2. **MiniXception Model:** As explained previously, this model is used in extracting the essential features from the images to aid classification.

3. **Sparse Categorical Cross-Entropy Loss:** This is a popular loss function that is used in problems relating to classification. It works by measuring the difference between original and predicted values.

4. **Adam Optimizer:** Main reason for using this is to shift the model's weights during training to improve efficiency.

Flowchart



Parallel Implementation

We used two different approaches in order to tackle the problem of parallelizing the image classification using CNN.

Code Using Pytorch, CUDA and Multithreading

Architecture and Logic

- Data Preprocessing: The constants including the directory for saving the preprocessed data (PREPROCESSED_DATA_DIR) are established. It is declared that the directory for the preprocessed data exists. os. madeirs() is used to give this particular result. The code downloads the famous CIFAR-10 through the TensorFlow's datasets module beforehand. Thus, the data is normalized by normalization of the pixel values. Thus, it is performed by the selection of values for them in the range of (0, 1). After the labels are generated, the labels are subsequently transformed to one-hot encoding. The previously processed data is finally stored on the disk by np. savez_compressed().

- PyTorch Implementation: In this code, the MiniXception model is implemented following the standards of the PyTorch library. It is quite similar to the TensorFlow/Keras implementation's architecture. The distinction is that, that PyTorch implements nn.Conv2d, nn.BatchNorm2d, and nn.ReLU modules unlike TensorFlow/Keras implementation of MiniXception. TensorFlow/Keras implementation uses convolutional layers, batch normalisation, and ReLU activation.

- Training: Transforms are defined using torchvision. They are applied to resize images to 32x32, convert them to tensors and to normalize pixel values. CIFAR-10 training dataset is loaded using torchvision.datasets.CIFAR10 with specified transformations. Training data is loaded into batches using torch.utils.data.DataLoader with multi-threading enabled (num_workers=4). This setup kick starts the model, then shifts it to GPU, if available, then sets it up with the Adam optimizer and cross-entropy loss. Then, the training loop kicks in, goes through each epoch and handles batches of training data. Moreover, for every batch, the model gets trained, and we calculate and print out the loss and accuracy metrics.

Key Functions/Modules:

1. OS: This verifies the directory of preprocessed data.

2. Torch, Torch.nn, and Torch.optim : The neural network is defined and trained by these.

3. Torchvision.transforms and Torchvision.datasets:Used to load the CIFAR-10 dataset.

4. Torch.utils.data.DataLoader: utilized to build a multi-threaded data loader that loads batches of data to train the neural network

5. Torch summary.summarizes: provides information about the MiniXception model.

Explanation of Algorithms/Techniques:

1. Convolutional Neural Networks (CNNs):CNNs were used in this code to extract hierarchial features from the pixel values of the input images. To make this process of detecting features better, MiniXception is incorporated.MiniXception makes the process better by utilizing batch normalization, residual connections, and depthwise separable convolutions.

2. MiniXception Model: Pytorch implementation of MiniXception model consists of ReLU activation functions, batch normalization and most importantly, convolutional layers. The flows in the model are defined as follows: The entering flow has two convolutional layers, and the middle block has four residual blocks. Each block is again made of convolutional layers, ReLU and batch normalization. Lastly, the exit flow too has convolutional layers but with max pooling. As for feature aggregation ie when features get combined together, Global Average pooling is applied. Then, the fully connected layer classifies the images.

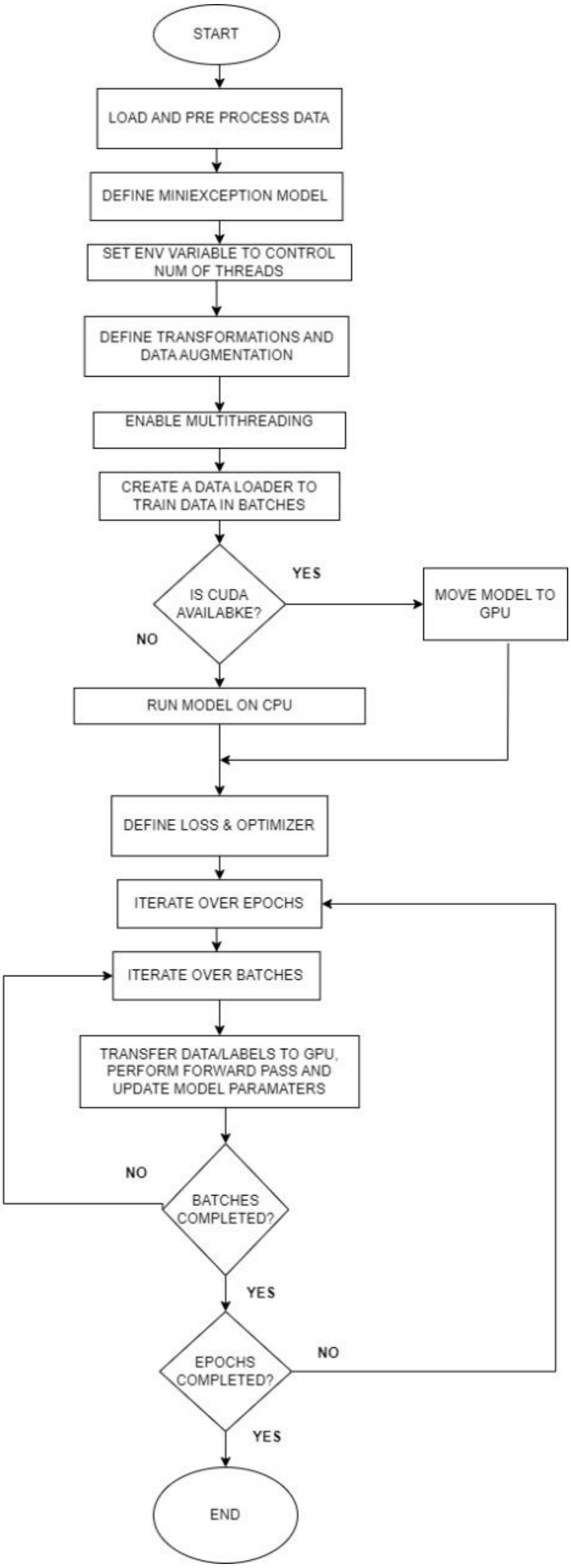
3. CUDA: Pytorch is easily integrated with NVIDIA CUDA,to utilize GPUs to train the neural networks.This process is is much faster compared to texecution only on the CPU. Taking advantage of this parallel processing power by using CUDA, parallel computations performed lead to faster training and better model performance.This is a massive benefit to large datasets in deep learning models.

4. DataLoader Multi-threading: Simultaneous preprocessing and data loading is performed by several threada using PyTorch's DataLoader. This cuts down on time and improves training efficiency

by making greater use of the computational resources available. It also simultaneously fetches data and preprocesses it between training to minimize the idle time between the training processes.This is particularly common when using large data sets as in the case with our code that uses the CIFAR 10 data set.Hence, the overall process results in faster training.

5. Cross-Entropy Loss: This is a frequently used loss function used for problems like our, ie Classification problems. It quantifies how different the original distribution of class labels is by the predicted one using probability distribution. This is used by the PyTorch model to differentiate between the unique classes provided in the dataset and then produce accurate predictions by maximising cross-entropy loss.

Flowchart


START
LOAD AND PRE PROCESS DATA
DEFINE MINIEXCEPTION MODEL
SET ENV VARIABLE TO CONTROL NUM OF THREADS
DEFINE TRANSFORMATIONS AND DATA AUGMENTATION
ENABLE MULTITHREADING
CREATE A DATA LOADER TO TRAIN DATA IN BATCHES
IS CUDA AVAILABKE?
YES
MOVE MODEL TO GPU
NO
RUN MODEL ON CPU
DEFINE LOSS & OPTIMIZER
ITERATE OVER EPOCHS
ITERATE OVER BATCHES
TRANSFER DATA/LABELS TO GPU, PERFORM FORWARD PASS AND UPDATE MODEL PARAMATERS
NO
BATCHES COMPLETED?
YES
EPOCHS COMPLETED?
NO
YES
END

Code using Dask Arrays and TensorFlow Mirrored Strategy on GPUs.

Architecture and Logic

- Dataset Loading and Preprocessing: Firstly, the commonly known dataset; CIFAR-10 dataset is loaded using TensorFlow's datasets module. Then, images have been normalized such that they have values between 0 and 1. Then, some sort of processing is also done on the class labels.

- Model Architecture: The CNN model used uses a a variant of the Xception architecture; MiniXception. It consists of three different flows that include entry flow, middle flow, and exit flow. All of these flows involve mechanisms of separable convolutions and batch normalization. Also, they use the function of Activation. This architecture is designed such that it facilitates efficient feature extraction. This efficiency helps in keeping the computational cost minimum.

- Training: The model is compiled using Adam optimizer and sparse categorical cross-entropy loss function. Dask arrays are used to distribute the data across multiple workers. This is how training is parallelized. Training is performed on GPU subject to its availability otherwise it’s done on CPU.

- Evaluation: Lastly, the model is evaluated on the test dataset. Further, it is checked on basis of its predictions.

Key Functions/Modules:

1. TensorFlow and Keras: These are used for loading datasets. Also, their main function in code involved defining the model, and then training the CNN model.

2. Dask: It is a well-known method that is utilized for parallelizing the training process by distributing the data across multiple workers.

3. Matplotlib: It is mainly used for providing visualizations of sample images and their corresponding labels.

4. NumPy: Its task is to perform numerical operations and to manipulate the array.

5. Scikit-learn: It is employed for label encoding and splitting the dataset into training and validation sets.

Explanation of Algorithms/Techniques:

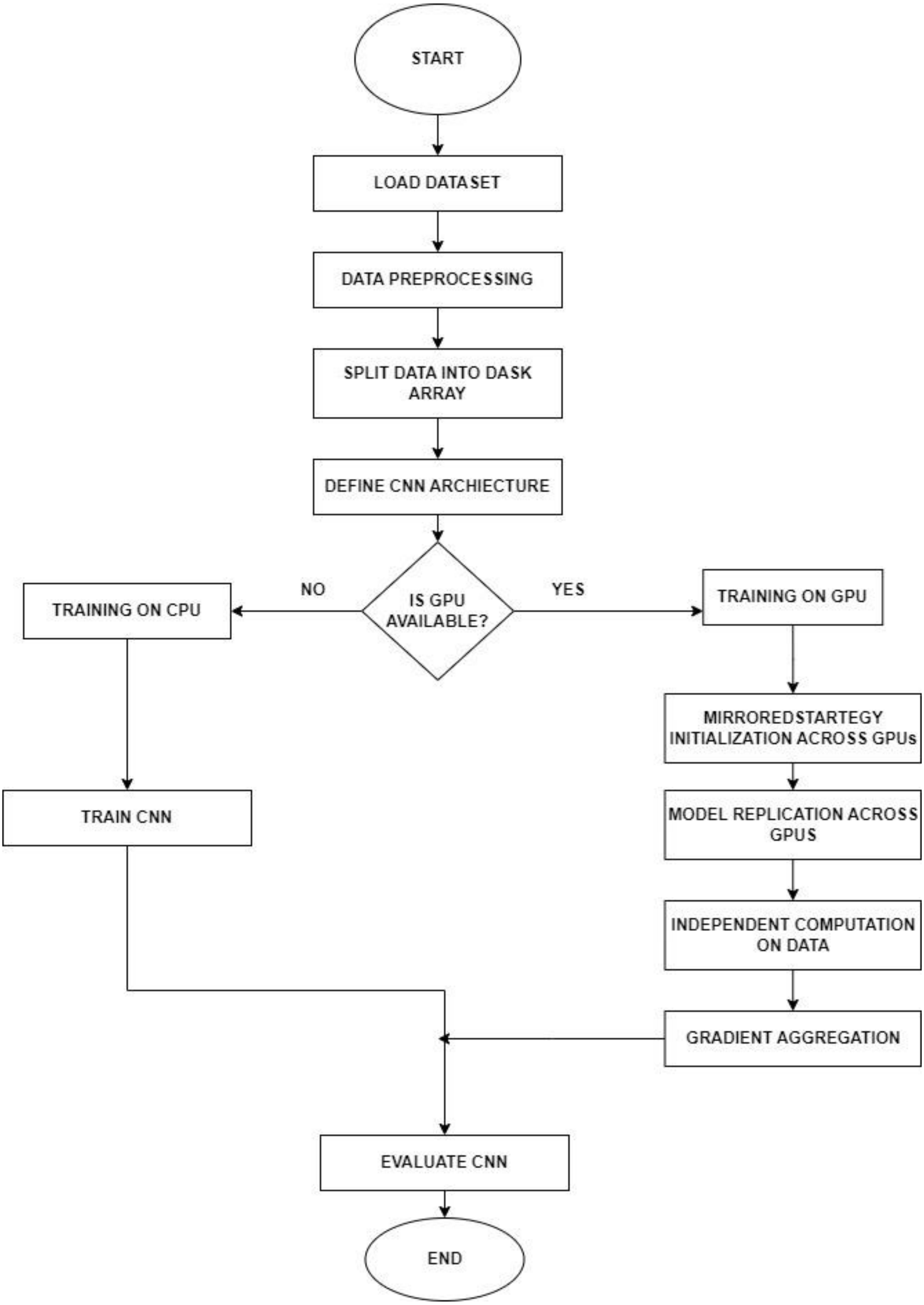
1. Convolutional Neural Networks (CNNs): Our code revolves around CNN. CNNs are predominantly used for image classification because they excel in learning the hierarchical representations of features directly from pixel values. They use the Xception architecture. One of the methods from that architecture that we have employed is MiniXception. MiniXception model includes the methods/functions of depthwise separable convolutions, batch normalization, and residual connections. It uses these techniques to achieve efficiency in feature extraction and parameter reduction.

2. Data Parallelism using Dask Arrays: Dask arrays are utilized because they aid in achieving parallelism by parallelizing the training process across several workers. Also, it proves an ideal option for large datasets, as they partition large datasets into smaller chunks and then apply a distributed computation strategy.

3. GPU Acceleration: Before training on the model, the code first checks for the availability of GPU. If GPU isn't available, then it trains on the CPU. We have implemented GPUs in our code because they greatly accelerate parallelism. They leverage thousands of cores to perform computations in parallel. This allows for the simultaneous processing of multiple data points or operations. This in turn greatly speeds up training as compared to traditional CPUs.

4. MirroredStrategy for Distributed Training: The MirroredStrategy method of TensorFlow is used for synchronous training data distribution across multiple GPUs. The condition here is, that all GPUs must reside on a single machine. The MirroredStrategy then instils data parallelism, by assigning a portion of the data to each GPU. Using this technique leads to faster convergence and improvement in training efficiency.

Flowchart

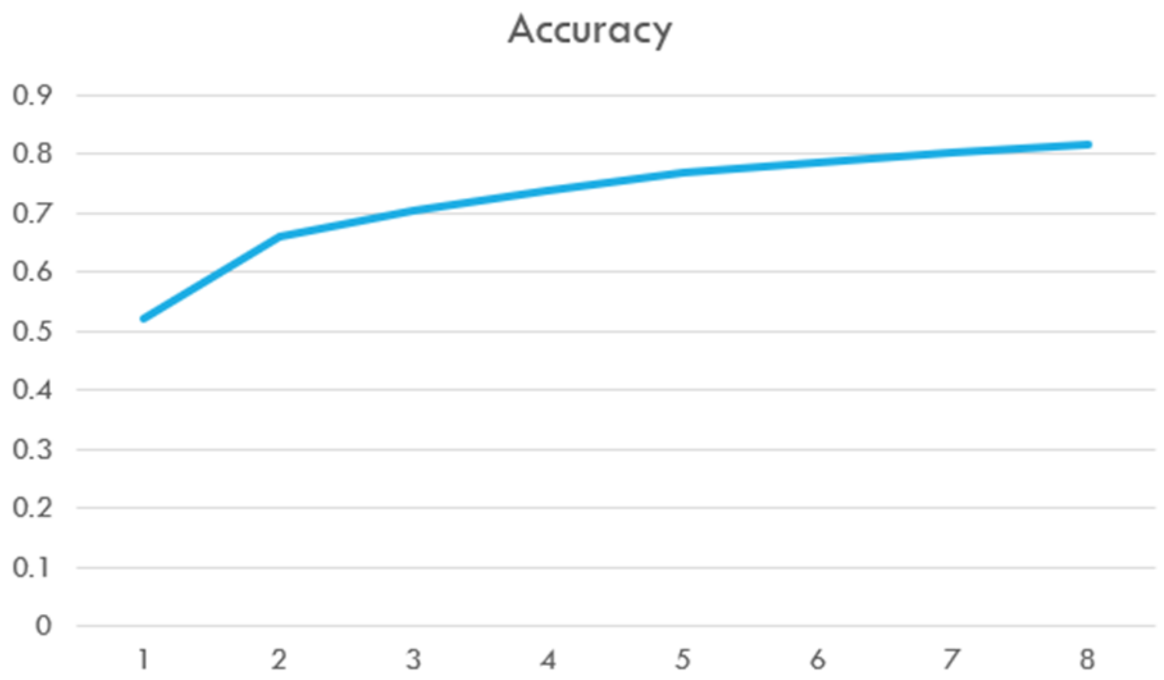

START
LOAD DATASET
DATA PREPROCESSING
SPLIT DATA INTO DASK ARRAY
DEFINE CNN ARCHIECTURE
IS GPU AVAILABLE?
NO
YES
TRAINING ON CPU
TRAIN CNN
TRAINING ON GPU
MIRROREDSTARTEGY INITIALIZATION ACROSS GPUs
MODEL REPLICATION ACROSS GPUS
INDEPENDENT COMPUTATION ON DATA
GRADIENT AGGREGATION
EVALUATE CNN
END

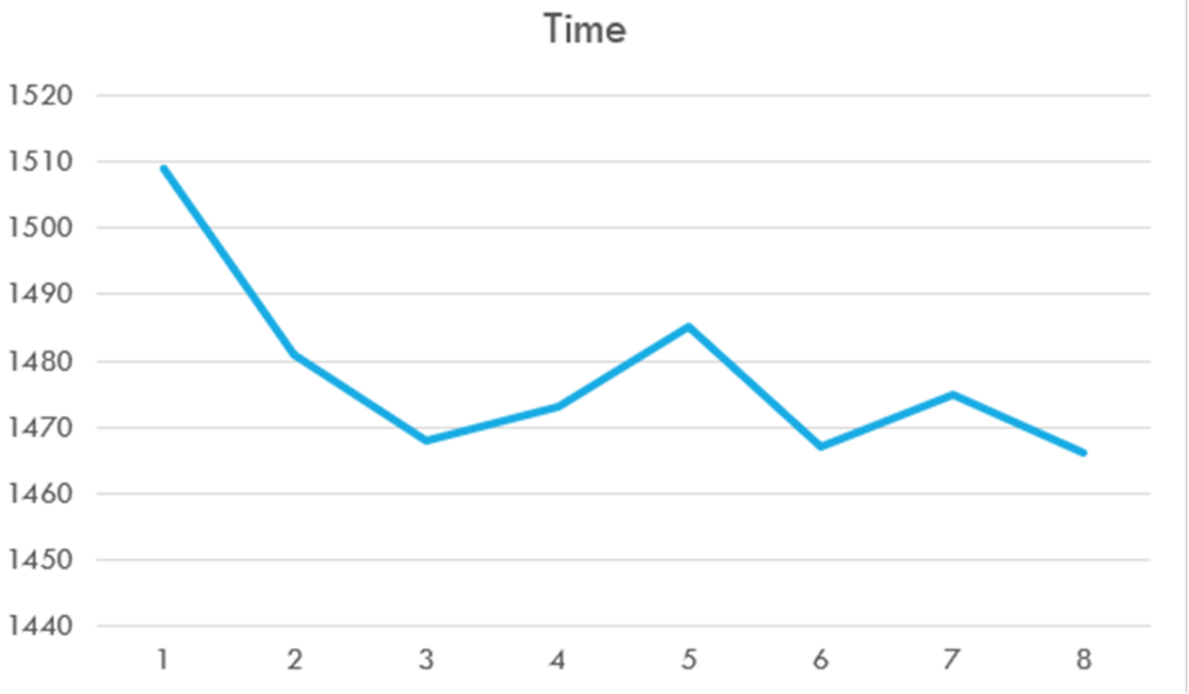
Results

Table 1 shows training performance of a basic sequential Convolutional Neural Network (CNN) implementation for image classification.

Table 1 - Sequential

Epoch	Accuracy	Time (in seconds)
1	0.5215	1509
2	0.6588	1481
3	0.7058	1468
4	0.7371	1473
5	0.7671	1485
6	0.7856	1467
7	0.8018	1475
8	0.8156	1466


Accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
4
5
6
7
8


Time
1520
1510
1500
1490
1480
1470
1460
1450
1440
1
2
3
4
5
6
7
8

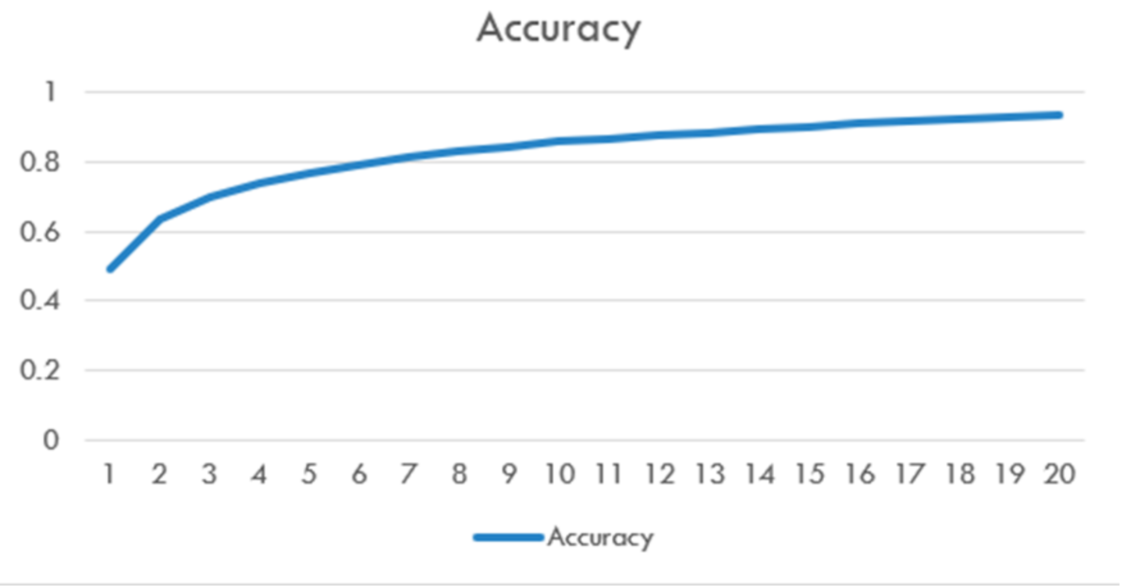
Cm

Table 2 shows the training progress of a PyTorch model using CUDA acceleration and multithreading.

Table 2 - Using PyTorch, CUDA and Multithreading

Epoch	**Accuracy**	**Time** (in seconds)
1	0.4918	36.80
2	0.6375	33.76
3	0.7009	34.05
4	0.7407	35.27
5	0.7666	33.57
6	0.7932	34.12
7	0.8132	33.37
8	0.8298	33.84
9	0.845	33.53
10	0.8578	33.50
11	0.8671	34.13

12	0.8751	33.33
13	0.8856	34.05
14	0.8943	33.89
15	0.9017	33.87
16	0.9094	33.46
17	0.917	33.46
18	0.9225	33.64
19	0.9299	33.35
20	0.9334	33.76



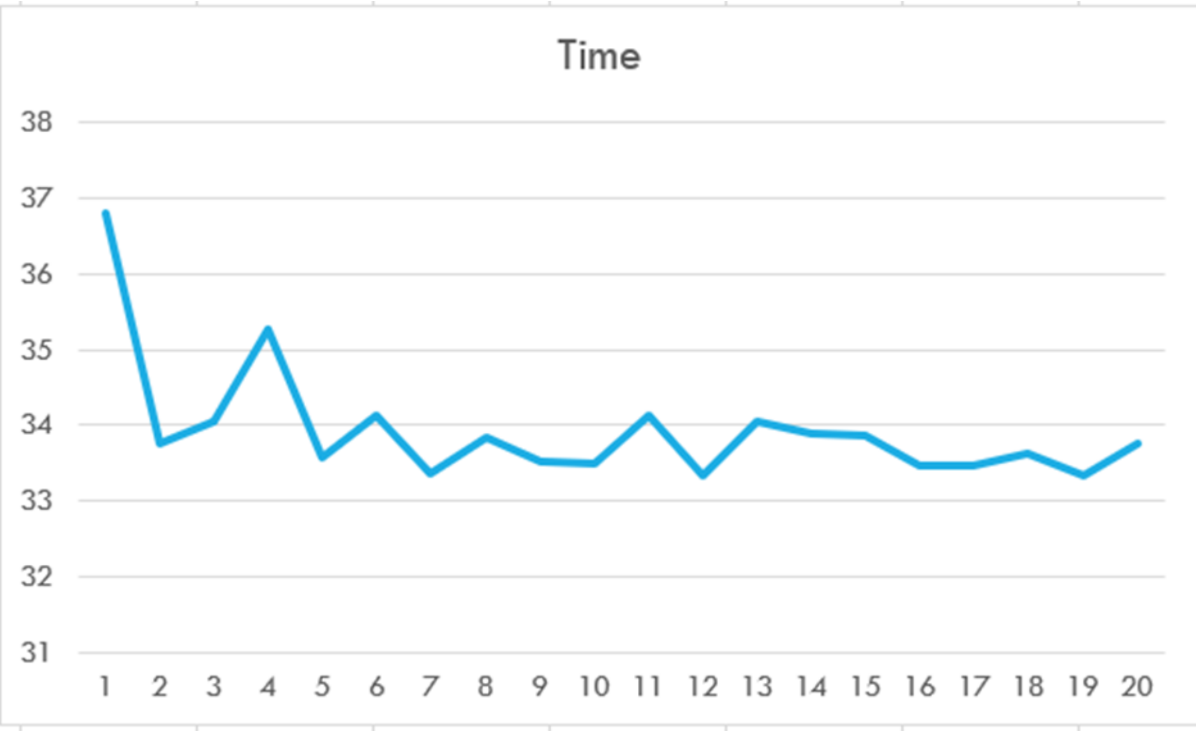


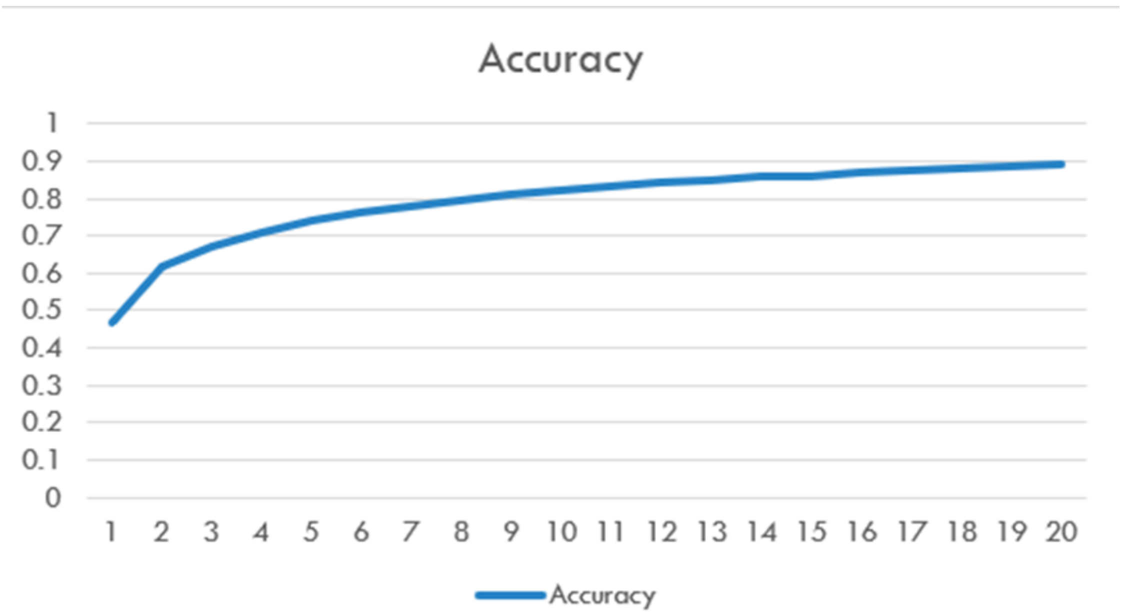
Table 3 shows the training performance of a parallel Convolutional Neural Network (CNN) utilizing GPU acceleration, MirroredStrategy, and Dask Arrays.

Table 3 - Using GPU acceleration, Dask Arrays & Mirrored Strategy

Epoch	Accuracy	Time (in seconds)
1	0.4662	48
2	0.6161	36

3	0.6724	36
4	0.7108	35
5	0.7404	36
6	0.7627	36
7	0.7784	35
8	0.7969	36
9	0.8102	35
10	0.8222	36
11	0.8309	37
12	0.8411	35
13	0.8488	36
14	0.8566	36
15	0.8615	35
16	0.8686	36
17	0.8763	36

18	0.8804	36
19	0.8841	36
20	0.8915	36



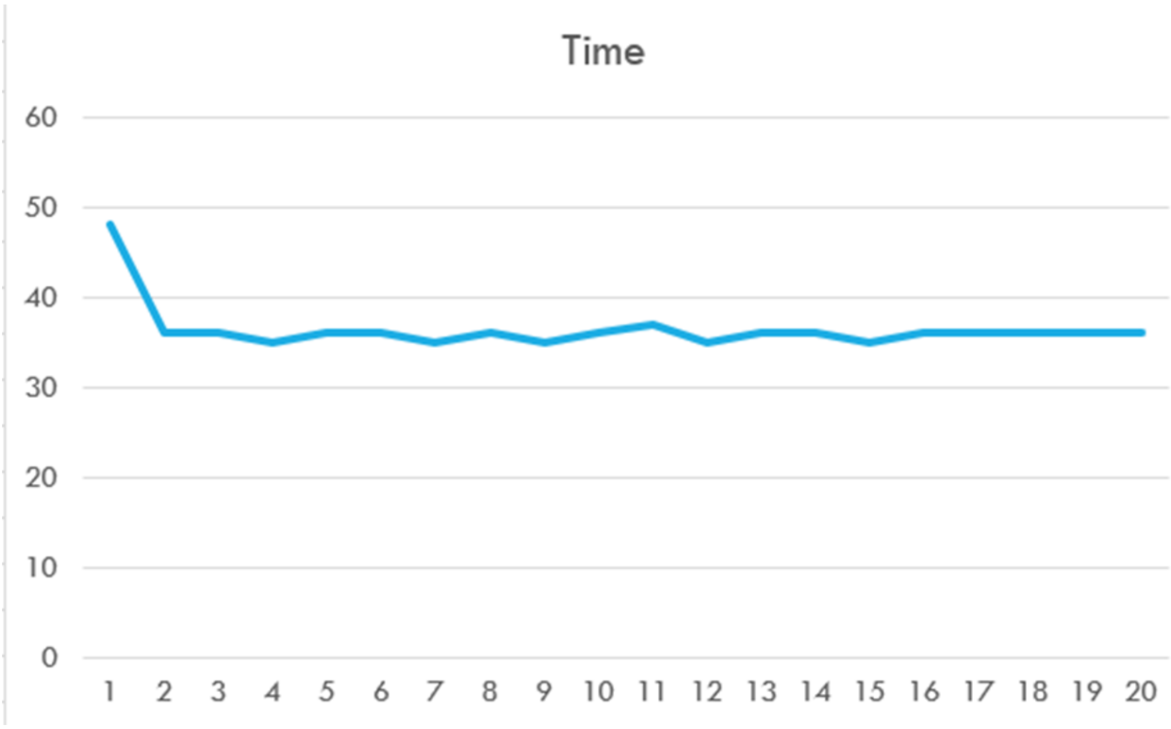


Table 4 shows the overall comparison of all approaches with respect to accuracy. It compares the accuracy of image classification models across epochs: the sequential model achieves the highest accuracy, followed by the model utilizing GPU, MirroredStrategy, and Dask Arrays, with the PyTorch model using CUDA and multithreading displaying slightly lower accuracy.

Table 4 - Accuracy Comparison

Epochs	**Sequential**	**GPU, MirroredStrategy and Dask Arrays**	**Using Pytorch, CUDA and Multithreading**
1	0.5215	0.4662	0.4918
2	0.6588	0.6161	0.6375
3	0.7058	0.6724	0.7009
4	0.7371	0.7108	0.7407
5	0.7671	0.7404	0.7666
6	0.7856	0.7627	0.7932
7	0.8018	0.7784	0.8132
8	0.8156	0.7969	0.8298

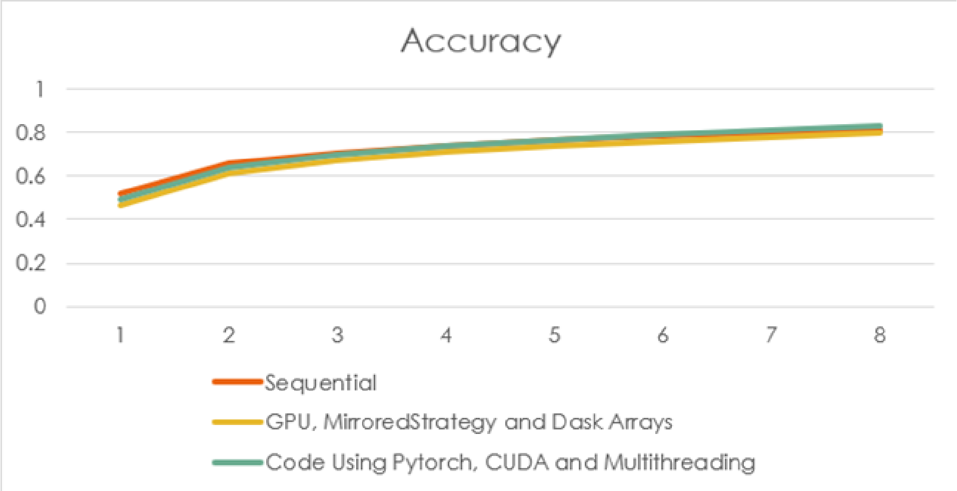
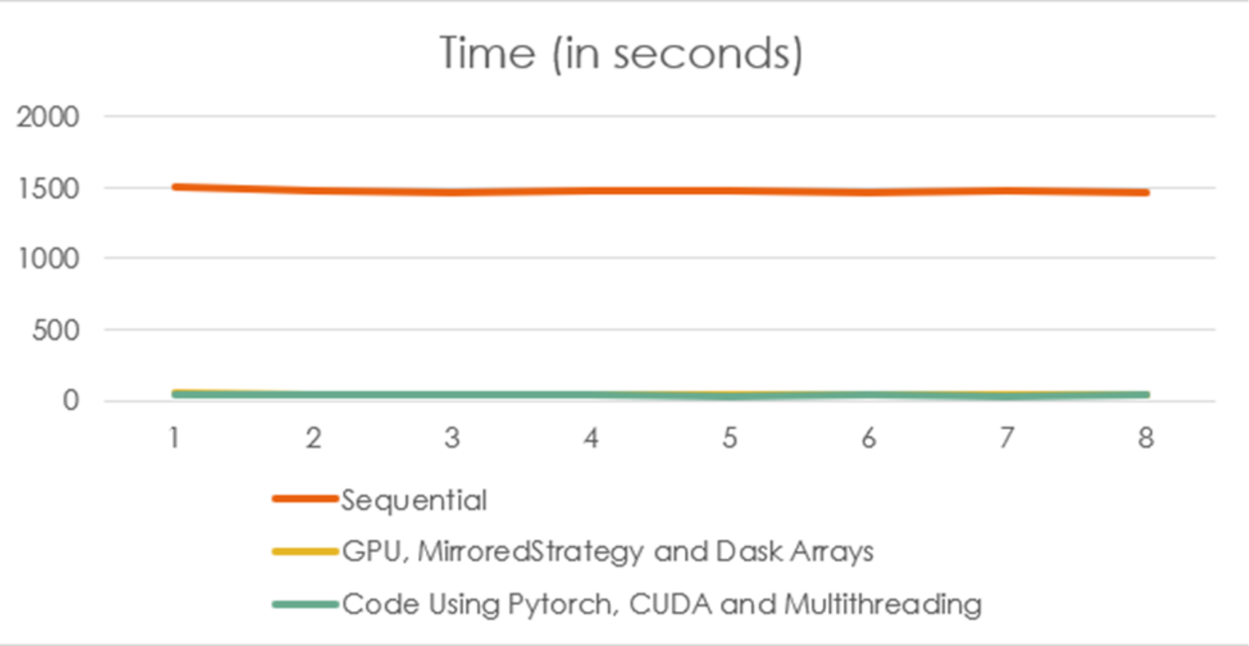


Table 5 shows the training time (in seconds) per epoch for different implementations: the sequential model takes the longest time, followed by the PyTorch model with CUDA and multithreading, while the GPU, MirroredStrategy, and Dask Arrays implementation is the fastest.

Table 5 - Time Comparison

Epochs	**Sequential**	**GPU, MirroredStrategy and Dask Arrays**	**Using Pytorch, CUDA and Multithreading**
1	1509	48	36.80
2	1481	36	33.76
3	1468	36	34.05
4	1473	35	35.27
5	1485	36	33.57

6	1467	36	34.12
7	1475	35	33.37
8	1466	36	33.84



Conclusion

The application of cutting-edge tactics like GPU acceleration with MirroredStrategy and Dask Arrays, as well as PyTorch's CUDA integration with multithreading, has significantly shortened the training time without the loss of the accuracy of the sequential implementation. These techniques rely on the parallelism and the distributed computing to speed up the training and so the algorithms can converge faster thus giving the possibility to be scaled.

The case of GPU acceleration with MirroredStrategy and Dask Arrays demonstrates that the training time is considerably shortened because of the parallel processing ability of GPUs. Through the use of MirroredStrategy, the model can share the workload among several GPUs, thus realizing the computations at the same time, which in turn, the time of each epoch is reduced. Besides, Dask Arrays are the key to the process of distributed computing, which means the training data is divided and processed at the same time on different workers. This vectorization of the training tasks is the major reason why the training time is drastically reduced, as shown by the shorter epoch times in the parallel version than in the sequential one.

Similarly, in the PyTorch implementation using CUDA and multithreading, the training process is fastened by the use of the high-speed GPUs. CUDA is the tool that allows the program to be executed on the GPU, where it can perform operations on thousands of cores at the same time. Through the change of training workload from the CPU to the GPU, the model gets faster matrix multiplications and other operations, so in the end, the training becomes quicker. Besides this, the application of multithreading in the data loading process combines the fetching and preprocessing of batches of training data at the same time, thus reducing the idle time between training iterations.

Moreover, the evaluation revealed that the training time was reduced significantly by these forthcoming techniques, but the accuracy of the trained models did not decrease and they got the same as the sequential implementation. Thus it shows the greatness of parallelism and distributed computing in the fastening of the training process without the loss of the model's performance. Generally, the use of GPU acceleration with MirroredStrategy and Dask Arrays, together with PyTorch's CUDA integration with multithreading, makes a persuasive option for the training of deep learning models in an efficient and effective way, which is the reason why they are the most preferred choice for large-scale training tasks.

Future Work

- Implement advanced parallelization techniques like incorporating pipeline or distributing computations across different nodes.
- The current time being taken is still too much. To make the process more efficient, work on communication overhead by processes such as reduced synchronization barriers, and batching

communication operations effectively. Techniques used could be model compression, quantization, and sparsification. They will significantly impact the large overhead.

- Another feature that can be experimented is utilizing different optimizing algorithms like learning rate schedules and using regularization techniques to make the process faster

- Leverage mixed precision training to speed up training further, especially when using NVIDIA GPUs with Tensor Cores. This technique uses both 16-bit and 32-bit floating-point representations during training, which can reduce memory usage and improve performance. PyTorch provides support for mixed precision training through the **torch.cuda.amp** module.

- Experiment with cloud-based services or distributed computing platforms by training on a larger data than CIFAR10 and compare the differennces in time taken for both the approachs.

- Explore distributed training frameworks such as TensorFlow's **tf.distribute** and PyTorch's **torch.distributed** for training data on different machines and observing differences in the time taken to get better comparing values.

- Implement distributed training using PyTorch's popular **torch.nn.parallel.DistributedDataParallel** module to train the model across multiple machines or GPUs in a distributed environment.

- Profile the code to identify performance bottlenecks and optimize critical sections using tools like TensorFlow Profiler, PyTorch Profiler, or NVIDIA's Nsight Systems (very commonly used).

- Moreover, an advanced level optimization could also be explored like using GPU-aware optimizations eg kernel fusion, memory layout optimization. These would result in maximum utilization of GPUs efficiency which could make the process a lot more efficient.

References

1. Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* https://doi.org/10.1109/cvpr.2014.81

2. Yan, Z., Zhang, H., Piramuthu, R., Jagadeesh, V., DeCoste, D., Wei, D., ... Yu, Y. (2015). HD-CNN: Hierarchical deep convolutional neural networks for large scale visual recognition. *Proceedings of the IEEE International Conference on Computer Vision.* https://doi.org/10.1109/iccv.2015.314

3. Lee, S., Kang, Q., Madireddy, S., Balaprakash, P., Agrawal, A., Choudhary, A., ... Liao, W. (2019). Improving scalability of parallel CNN training by adjusting mini-batch size at run-time. *Proceedings of the IEEE International Conference on Big Data.* https://doi.org/10.1109/BigData47090.2019.9006550

4. Bonvallet, R., Maureira, C., Fernández, C., Arce, P., & nete., A. (2014). A feed forward neural network in CUDA for a financial application. *Proceedings of the Brazilian Congress on Computational Mechanics.* https://doi.org/10.5151/mecengwccm2012-1991

5. Park, J., & Suh, Y. (2020). Speculative Backpropagation for CNN Parallel Training. *IEEE Access.* https://doi.org/10.1109/ACCESS.2020.3040849

6. Yao, L., et al. (2020). Evaluating and analyzing the energy efficiency of CNN inference on high-performance GPU. *Concurrency and Computation Practice and Experience.* https://doi.org/10.1002/cpe.6064

7. Chen, J., Li, K., Bilal, K., Zhou, X., Li, K., & Yu, P. S. (2019). A bi-layered parallel training architecture for large-scale convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, *30*(5), 965-976. https://doi.org/10.1109/tpds.2018.2877359

8. Ren, S., He, K., Girshick, R., & Sun, J. (2017). Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 39*(6), 1137-1149. https://doi.org/10.1109/tpami.2016.2577031

9. Teerapittayanon, S., McDanel, B., & Kung, H. (2016). BranchyNet: Fast inference via early exiting from deep neural networks. *Proceedings of the International Conference on Pattern Recognition.* https://doi.org/10.1109/icpr.2016.7900006

10. Marques, J., Falcao, G., & Alexandre, L. A. (2017). Distributed learning of CNNs on heterogeneous CPU/GPU architectures. https://doi.org/10.48550/arXiv.1712.02546

11. Zhang, Q., Zhang, M., Chen, T., Sun, Z., Ma, Y., & Yu, B. (2018). Recent advances in convolutional neural network acceleration. *Neurocomputing.* https://doi.org/10.1016/j.neucom.2018.09.038

12. Author(s). (2023). Parallel convolutional neural network toward high efficiency and robust structural damage identification. *Structural Health Monitoring.* https://doi:10.1177/14759217231158786.

13. Huang, et al. (2022). Fine-Grained Ship Classification by Combining CNN and Swin Transformer. *Remote Sensing.* https://doi:10.3390/rs14133087

14. Abd-Ellah, M. K., Awad, A. I., Hamed, H. F. A., & Khalaf, A. A. M. (2019). Parallel Deep CNN Structure for Glioma Detection and Classification via Brain MRI Images. *2019 31st International Conference on Microelectronics (ICM), 1-6.* https://doi:10.1109/icm48031.2019.9021872

15. Hailesellasie, M., & Hasan, S. R. (2017). A fast FPGA-based deep convolutional neural network using pseudo parallel memories. *2017 IEEE International Symposium on Circuits and Systems (ISCAS).* https://doi:10.1109/iscas.2017.8050317

16. Hernandez, D., Olague, G., Hernández, B., & Clemente, E. (2017). CUDA-based parallelization of a bio-inspired model for fast object classification. *Neural Computing and Applications*, *30*(10), 3007-3018. https://doi.org/10.1007/s00521-017-2873-3