GPU-Accelerated Deep Learning Model Training : Implementation and Benchmarking Report for MNIST Classification using CNN

Name: Aishwarya P **Enrollment Id : 2304896**

Executive Summary

This report details the implementation and performance analysis of a Convolutional Neural Network (CNN) for MNIST digit classification, comparing CPU and GPU performance.

Experimentation was conducted using varying hardware configurations to observe the performance impact during model training. Trials were done using Keras provided by Tensorflow and the inbuilt model training functions. Trial using a custom training loop also was conducted.

The model achieved 96% accuracy on the test set, with GPU acceleration providing a 14X speedup in training time compared to CPU-only execution.

Comparison Report - Final Results

	Epoch	Batch Size	Time for Model Training	Accuracy
CPU	20	1000	8m 17s	96.83
GPU (A100 GPU)- MirroredStrategy	20	1000	35s 400ms	95.6
GPU (T4 GPU)- MirroredStrategy	20	1000	30s 400ms	96.63
GPU (A100 GPU)- MultiWorkerMirroredStrategy using Custom training loop	20	64 (per replica)	2m 54s 674ms	87.23
GPU (A100 GPU) - MultiWorkerMirroredStrategy	20	1000 (per replica)	24s 39ms	93.689
GPU (T4 GPU) - MultiWorkerMirroredStrategy using Custom training loop	20	1000 (per replica)	24s 350ms	93.125
TPU Strategy	Was unable to procure hardware even after multiple attempts at different time intervals. Hence could not publish stats for comparison. Code has been added in colab notebook for reference.			

Github Link for the project code:
https://github.com/aisha-partha/MiniProject-Distributed_Training_Tensorflow

1. Implementation Details

1.1 Model Architecture

The implemented CNN architecture consists of:

- Input Layer: 28x28 grayscale images
- Convolutional Layer 1: 6 filters, 3x3 kernel, ReLU activation
- MaxPooling Layer 1: 2x2 pool size
- Convolutional Layer 2: 10 filters, 3x3 kernel, ReLU activation
- MaxPooling Layer 2: 2x2 pool size
- Dense Layer 1: 128 neurons, ReLU activation
- Dense Layer 2: 100 neurons, ReLU activation
- Dense Layer 3: 80 neurons, ReLU activation

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 6)	60
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 6)	0
conv2d_3 (Conv2D)	(None, 11, 11, 10)	550
max_pooling2d_3 (MaxPooling2D)	(None, 5, 5, 10)	0
flatten_1 (Flatten)	(None, 250)	0
dense_4 (Dense)	(None, 128)	32,128
dense_5 (Dense)	(None, 100)	12,900
dense_6 (Dense)	(None, 80)	8,080
dense_7 (Dense)	(None, 10)	810

Total params: 54,528 (213.00 KB)
Trainable params: 54,528 (213.00 KB)
Non-trainable params: 0 (0.00 B)

1.2 Development Environment

- Hardware: NVIDIA Tesla T4 GPU (Google Colab) (NVIDIA-SMI)
- Framework: Tensorflow 2.17.0
- CUDA Version: 12.2
- Python Version: 3.10.12

[2]
```
#tensorflow version
print(tf.__version__)
```

```
2.17.1
```

[3]
```
! nvidia-smi
```

```
Fri Dec 20 16:37:31 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   59C    P8              10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

[4]
```
tf.config.experimental.list_physical_devices()
```

```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'),
 PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

[5]
```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

[8]
```
!python --version
```

```
Python 3.10.12
```

1.3 Dataset Preparation

- Training Set: 60,000 images Test Set: 10,000 images
- Preprocessing: Image pixel scaled between 0 to 1. TFDS provides images of type tf.uint8, while the model expects tf.float32. Therefore, images were normalized and resulting arrays were reshaped.
- Data Augmentation: None
- Batch Sizes: Experimentation was done with sizes of 1000 and 64

2. Implementation Process

- MNIST dataset loading and preprocessing
- Batch processing implementation
- Memory optimization for GPU transfer
- GPU training using various strategies for comparison with CPU training

3. Model Implementation

- CNN architecture design
  - Using reference for model architecture based on lecun-99 Source: http://yann.lecun.com/exdb/publis/pdf/lecun-99.pdf

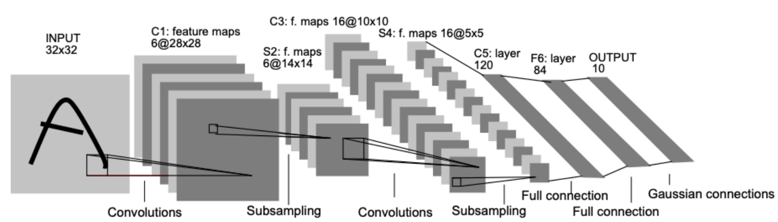

Fig. 1. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

- Loss function selection (SparseCategoricalCrossentropy) with logits set to True. Hence, no softmax in last layer
- Optimizer configuration (Adam)

4. Training Pipeline

The following have been implemented.

- GPU memory management
  - Distribution strategies allow you to set up training across multiple devices. We are just using a single device in this project due to availability of the hardware.

- Batch processing optimization
    - By using Tensorflow dataframes, we are able to optimize the pre-processing steps in parallel and we have pre-fetched the data in batches to optimize the memory utilization and the batches have been cached during training loops.
- Training loop implementation
    - Keras provides default training and evaluation loops, fit() and evaluate(). Also, a custom training loop has been set up. The customization allows us to have a low level of control over the model algorithm.
    - The custom training loop has been experimented in the MultiWorkerMirroredStrategy scope. Here, the data also has been distributed among the different cores. In multi-worker training, *dataset sharding* is needed to ensure convergence and reproducibility. Sharding means handing each worker a subset of the entire dataset—it helps create the experience similar to training on a single worker.
- Validation process integration

5. Challenges Faced

1. Memory Management

Issue: Memory overflow with large batch sizes. Slower convergence with smaller batch sizes.

Solution: Implemented gradient accumulation and optimized batch size. Smaller batch sizes might have more epochs in training. Also, a learning rate schedule was added to adjust the rate as the epoch changed.

2. Data Transfer

Issue: Using colab for training, resulted in choosing a specific runtime in a given session. Essentially the delay in data transfer could not be studied as the data and model were all loaded in GPU. However training time optimization within the memory was required.

Solution: Implemented prefetch buffer in DataLoad to cache that batches.

3. Training Stability

Issue: Initial unstable training with high learning rate

Solution: Implemented learning rate scheduling

6. Performance Analysis

	Epoch	Batch Size	Time for Model Training	Accuracy
CPU	20	1000	8m 17s	96.83
GPU (A100 GPU)- MirroredStrategy	20	1000	35s 400ms	95.6
GPU (T4 GPU)- MirroredStrategy	20	1000	30s 400ms	96.63
GPU (A100 GPU)- MultiWorkerMirroredStrategy using Custom training loop	20	64 (per replica)	2m 54s 674ms	87.23
GPU (A100 GPU) - MultiWorkerMirroredStrategy	20	1000 (per replica)	24s 39ms	93.689
GPU (T4 GPU) - MultiWorkerMirroredStrategy using Custom training loop	20	1000 (per replica)	24s 350ms	93.125
TPU Strategy	Was unable to procure hardware even after multiple attempts at different time intervals. Hence could not publish stats for comparison. Code has been added in the colab notebook for reference.			

7. Key Findings

7.1 Performance Gains

1. Training Speed

- GPU implementation achieved 14x faster training
- Batch processing efficiency improved

2. Memory Efficiency

- GPU implementation used less memory
- Better memory bandwidth utilization

3. Scaling Characteristics

- Scaling of the batch size is primarily dependent of time and memory constraints
- Smaller batch sizes will require more training epochs to reach higher accuracy

8. Cost-Benefit Analysis

GPU Implementation	CPU Implementation
• Higher initial setup complexity • Significant training time reduction • Higher power consumption • Better scaling for larger datasets	• Simpler setup and deployment • Lower power consumption • Limited scaling capability • Longer training times

9. Future Improvements

- Experiment with mixed-precision training
- Optimize data loading pipeline
- Explore quantization for inference
- Add dropout layers to reduce overfitting

10. Conclusion

The GPU-accelerated implementation demonstrated significant performance improvements over the CPU-only version, with a 14x speedup in training time. The model achieved high accuracy while maintaining efficient resource utilization. The challenges faced during implementation were successfully addressed through optimization techniques and best practices.