GPU Programming Assignment 1 - Report

Course: GPU Programming
Assignment: Train a 3-layer Neural Network (NN) for certain accuracy suitable for any classification problem
Github Link: https://github.com/akula-rajesh/gpu-programming-assignment-1

Team Members

Name	Roll Number
Akula Rajesh	M25AI1048
Pradeep Annepu	M25AI1109
Anirudh Reddy Aligireddy	M25AI1131
V Amarendra Chakravarthi	M25AI1082

Train a 3-layer Neural Network (NN) for certain accuracy suitable for any classification problem

1. Introduction to 3-Layer Neural Networks and Classification

A 3-layer Neural Network typically comprises an input layer, one hidden layer, and an output layer. This architecture is a foundational model in deep learning, capable of learning complex non-linear relationships in data. For demonstrating and benchmarking such a network, the MNIST handwritten digit classification dataset is an excellent choice. MNIST consists of 28×28 grayscale images of handwritten digits (0-9), representing a multi-class classification task. Its simplicity and well-defined nature make it ideal for illustrating core neural network concepts and performance characteristics.

2. Neural Network Architecture

The 3-layer Neural Network designed for MNIST classification has the following structure:

• **Input Layer**: Each MNIST image is 28×28 pixels. These are flattened into a 1D vector of 784 features. Thus, the input layer consists of 784 neurons.

• **Hidden Layer**: This intermediate layer processes the input features. For this model, a dense layer with 128 neurons is used. The Rectified Linear Unit (ReLU) activation function is applied to the output of this layer, given by $f(x)=\max(0,x)$. ReLU is chosen for its computational efficiency and its ability to mitigate the vanishing gradient problem.

• **Output Layer**: This layer produces the classification probabilities for each digit. With 10 possible digits (0-9), the output layer has 10 neurons. The softmax activation function is given by:

$$\text{softmax}(z)*j = *frac{e^{z_j}}{*sum*{k=1}^K e^{z_k}}, \quad j=1,\dots,K$$

It converts the raw outputs into a probability distribution over the 10 classes, where each probability is between 0 and 1, and their sum is 1.

The network is trained using the Adam optimizer, a popular adaptive learning rate optimization algorithm. The sparse categorical cross-entropy loss function is employed, suitable for multi-class classification with integer labels. Model performance is evaluated using accuracy as the primary metric.

---

3. Dataset Description – MNIST

The **MNIST** (Modified National Institute of Standards and Technology) dataset is a standard benchmark for handwritten-digit classification.
It contains grayscale images of digits **0 to 9**, each of size **28×28 pixels**.
For training efficiency, every image is **flattened to 784 features** and **normalized to [0, 1]**.

Property	Description
Total Samples	60,000 train + 10,000 test
Classes	10 (Digits 0–9)
Image Size	28×28 (784 features after flattening)
Type	Multi-class classification
Normalization	Pixel values scaled to [0, 1]

These preprocessing steps are exactly what our **NumPy**, **Keras**, and **CuPy** implementations perform before model training.

---

4. Implementation Approaches and Performance Analysis

To analyze speed and accuracy trade-offs, we implemented the same network in three ways:

4.1 Optimized Framework Implementation (Keras/TensorFlow on CPU)

- Built using the **TensorFlow Keras Sequential API**.
- Uses highly optimized C/C++ and BLAS backends.
- Trained with **Adam optimizer** for faster convergence.
- Serves as the **optimized CPU reference implementation**.

4.2 From-Scratch NumPy Implementation (CPU)

- Manual forward and backward propagation using NumPy matrix operations.
- Trained with **SGD** and fixed learning rate.
- Provides a **baseline** for understanding computational cost on CPU.

4.3 From-Scratch CuPy Implementation (GPU)

- GPU-accelerated version of the NumPy model using **CuPy arrays and CUDA**.
- All matrix multiplications, activations, and gradients run on GPU.

- TensorFlow was pinned to CPU to avoid GPU conflicts.
- Executed on **NVIDIA T4 GPU** in Google Colab.

---

5. Experimental Setup

Parameter	Value
Dataset	MNIST
Epochs	10
Batch Size	128
Learning Rate	0.01
Optimizer	SGD (NumPy/CuPy), Adam (Keras)
Hidden Layer	128 neurons with ReLU
Hardware	Intel Xeon CPU + NVIDIA T4 GPU (Colab)

---

6. Measured Results

Implementation	Training Time (s)	Test Accuracy
CuPy (GPU)	**5.55**	**93.0%**
NumPy (CPU)	7.79	97.2%
Keras (CPU)	25.12	97.3%

---

7. Speed-Up Analysis

Comparison	Speed-Up
GPU vs NumPy (CPU)	**1.40× faster**
GPU vs Keras (CPU)	**4.53× faster**

Although the GPU version uses simple SGD and slightly lower accuracy,
it demonstrates a clear training time improvement and shows how parallel GPU execution reduces computation time even for a small network.

---

8. Discussion and Insights

- **Speed:** GPU parallelism significantly reduces training time by processing matrix operations concurrently.
- **Accuracy:** The slight drop in CuPy accuracy (~93%) is due to the fixed SGD learning rate compared to Keras' adaptive Adam.
- **Framework Overhead:** Keras CPU training is slower due to graph execution and Python overhead.

- **Scalability:** The GPU advantage grows with larger datasets and deeper architectures.

---

9. Conclusion

Training a 3-layer Neural Network for classification, such as on the MNIST dataset, highlights significant performance differences between highly optimized deep learning frameworks (like Keras/TensorFlow) and custom NumPy implementations. The frameworks provide substantial speed-ups (typically 15-20x or more) by leveraging optimized C/C++ backends and BLAS libraries, minimizing Python overhead.

Regarding specific optimizations:

• **Tiling** is a crucial technique for optimizing matrix multiplication by improving cache locality. While essential for performance, it is transparently handled by the underlying BLAS libraries used by frameworks, making manual implementation in Python generally less efficient than using np.dot.

• **Neighbor-based computations** (stencil operations) are not directly relevant to the fully connected layers of a 3-layer MLP. However, they are a core concept in Convolutional Neural Networks, where they enable efficient local feature extraction in grid-like data such as images.