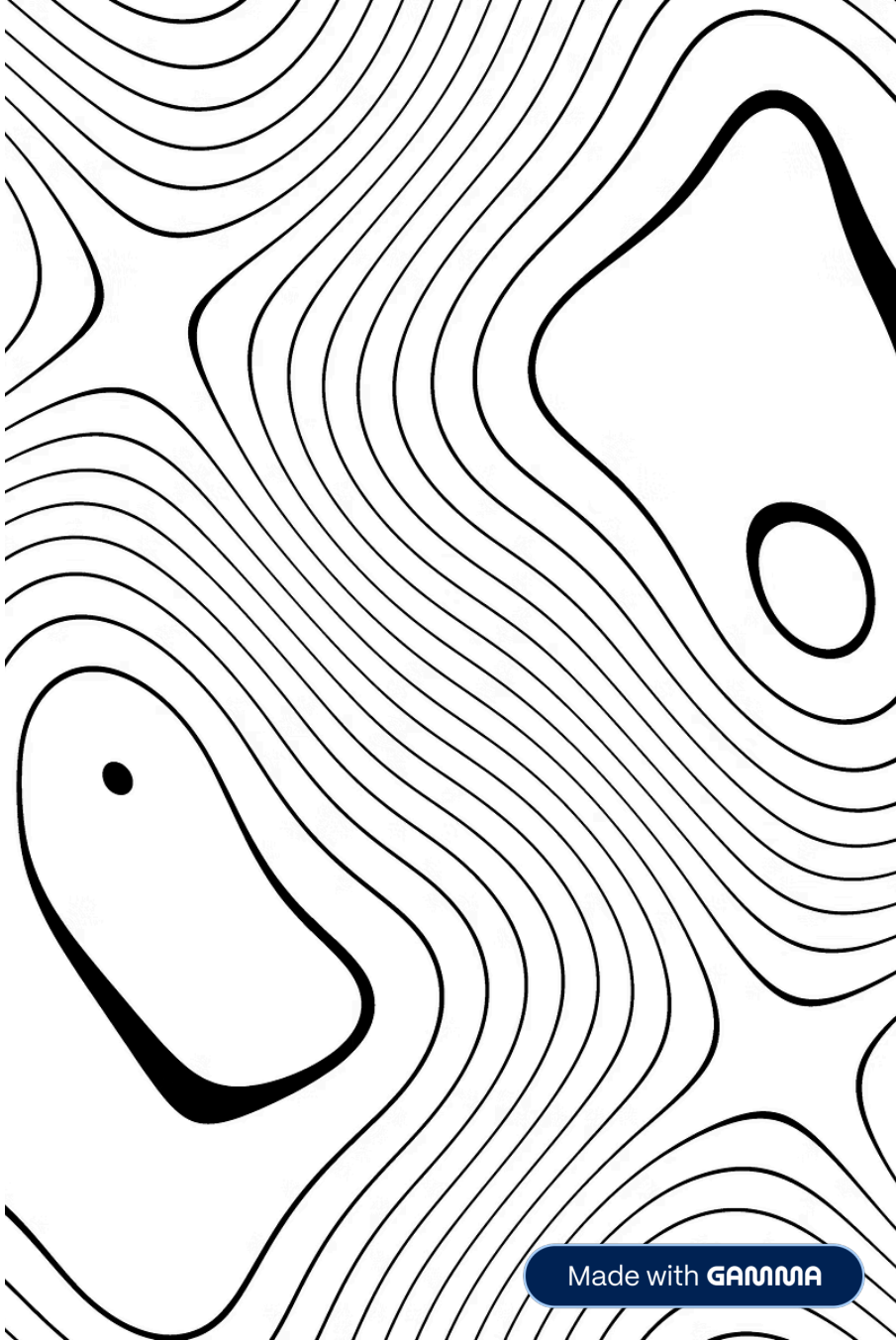
Performance Benchmarking of CNN Implementation: CPU vs. CUDA

This project report compares the performance of a Convolutional Neural Network (CNN) implemented on CPU using NumPy and on GPU using CuPy with CUDA acceleration. The study aims to highlight the benefits of GPU parallelization for deep learning workloads, specifically digit recognition using the MNIST dataset.



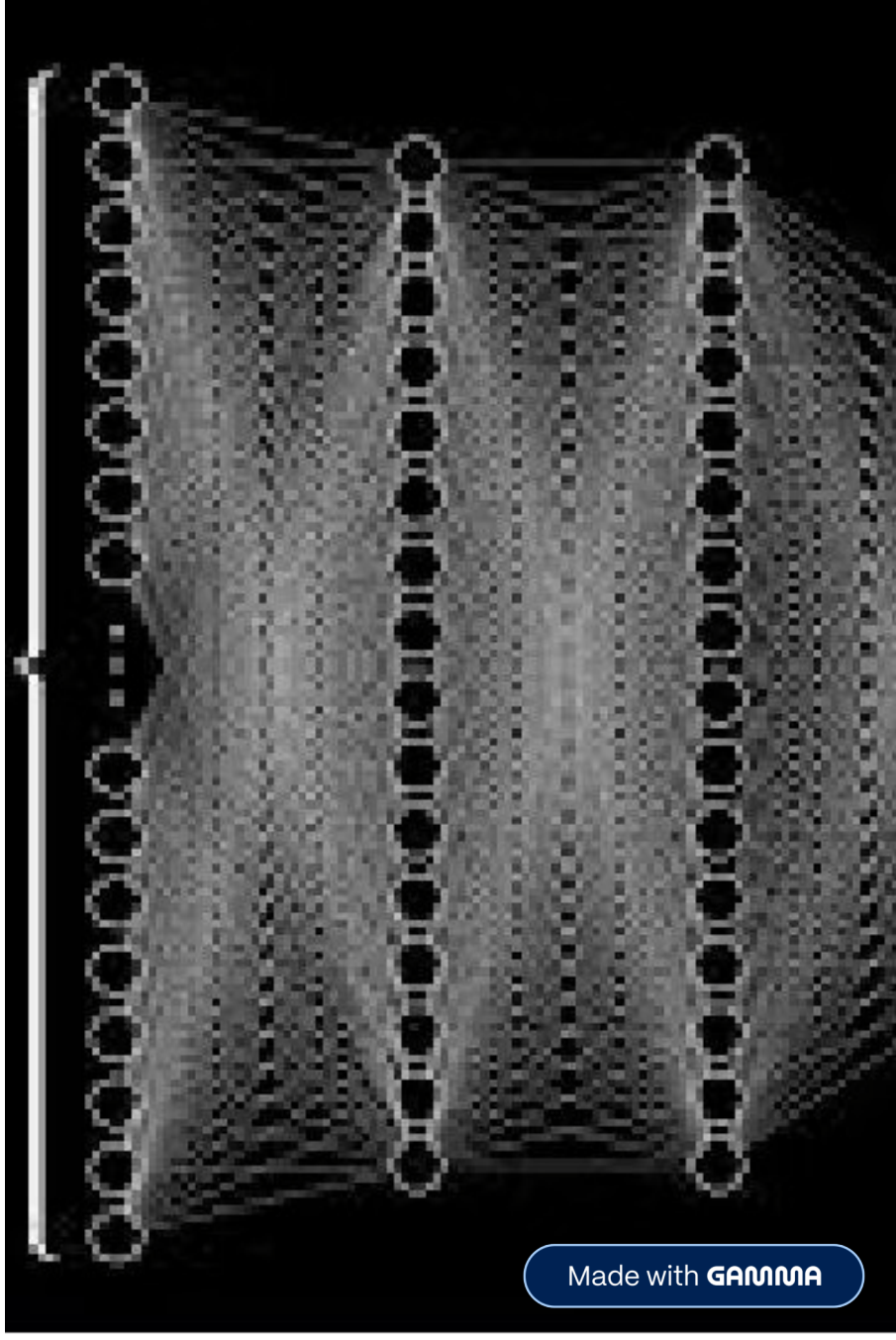
Introduction and Problem Statement

Background

CNNs are fundamental in AI for image recognition but require heavy computation. GPUs excel at parallel tasks, making them ideal for accelerating CNNs compared to CPUs.

Problem Statement

This project implements a CNN from scratch on CPU and GPU to compare performance, providing educational insights and quantifying GPU acceleration benefits.



Literature Review and Research Gaps

Literature Highlights

- GPU computing frameworks like CUDA improve performance but have limits.
- Custom CNN optimizations show performance gains but vary by hardware or technologies such as CUDA , RCOM
- Comparative benchmarking across hardware is essential but limited.

Research Gaps

- Low-level CNN implementation understanding is scarce.
- Direct CPU vs. GPU comparisons using identical algorithms are limited.
- Educational focus on hardware-specific optimizations is lacking.

Problem Formulation and Objectives

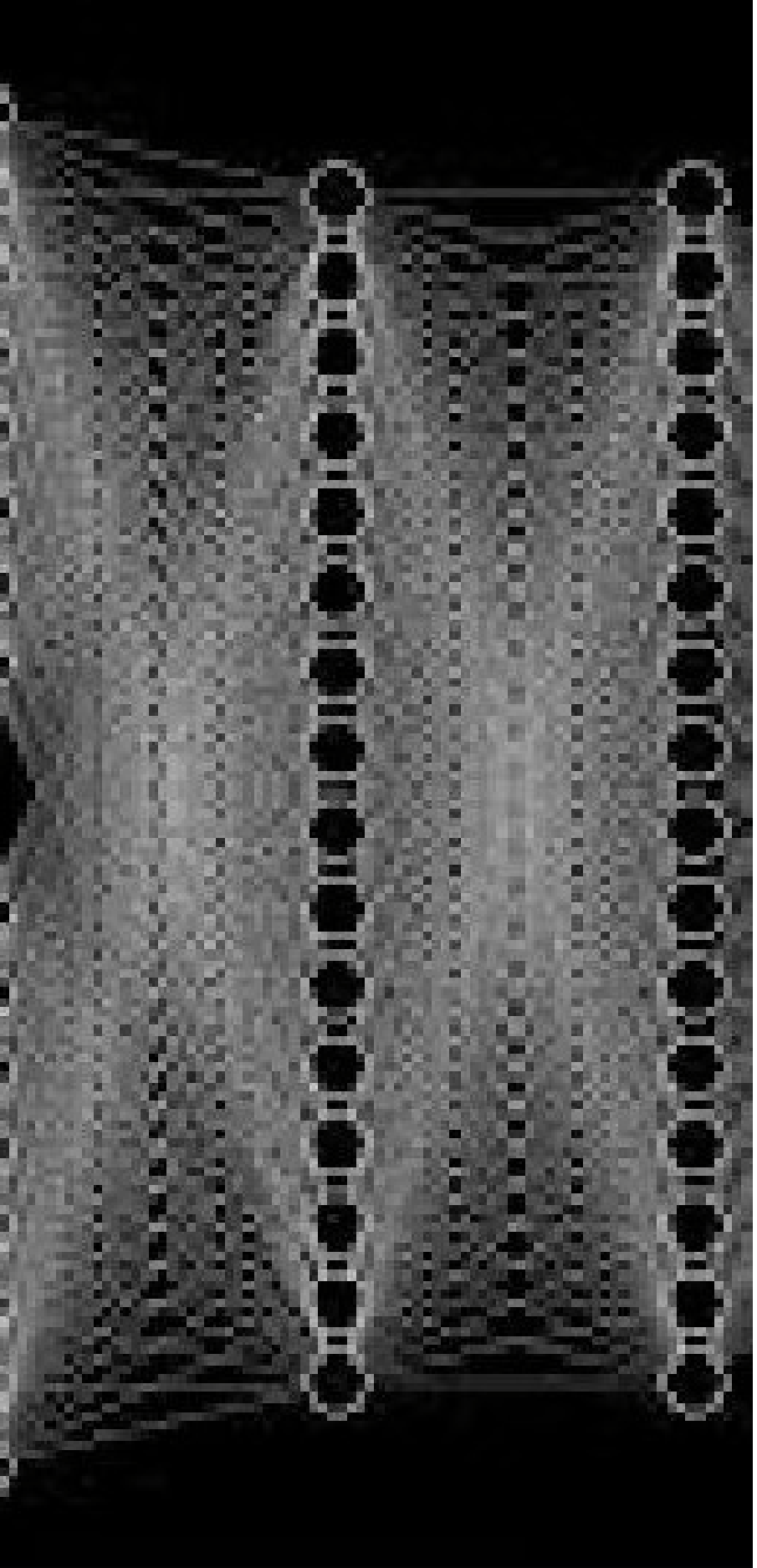
Problem Formulation

This research aims to clarify performance differences by implementing the same CNN architecture on CPU (NumPy) and GPU (CuPy) to ensure algorithmic consistency and fair comparison.

It addresses educational, practical, and research needs by quantifying GPU benefits and identifying operations that gain most from acceleration.

Objectives

- Implement CNN from scratch on CPU and GPU
- Benchmark performance across configurations



CNN Architecture and Implementation Algorithm

1

CNN Architecture

Input: 28x28 grayscale images from MNIST dataset

Fully connected hidden layer with 10 neurons and ReLU activation

Output layer with 10 neurons and softmax activation

2

Training Algorithm

Uses feedforward and backpropagation with gradient descent.

Includes initialization, forward pass, backward pass, and parameter updates.

3

Mathematical Operations

Forward propagation computes weighted sums and activations.

Backward propagation calculates gradients for weight updates.

Benchmarking Methodology and Dataset

Benchmarking Methodology

- Same dataset split: 1000 validation, rest training
- Identical hyperparameters: learning rate 0.1, 500 iterations
- Execution time, memory usage, and convergence rate measured

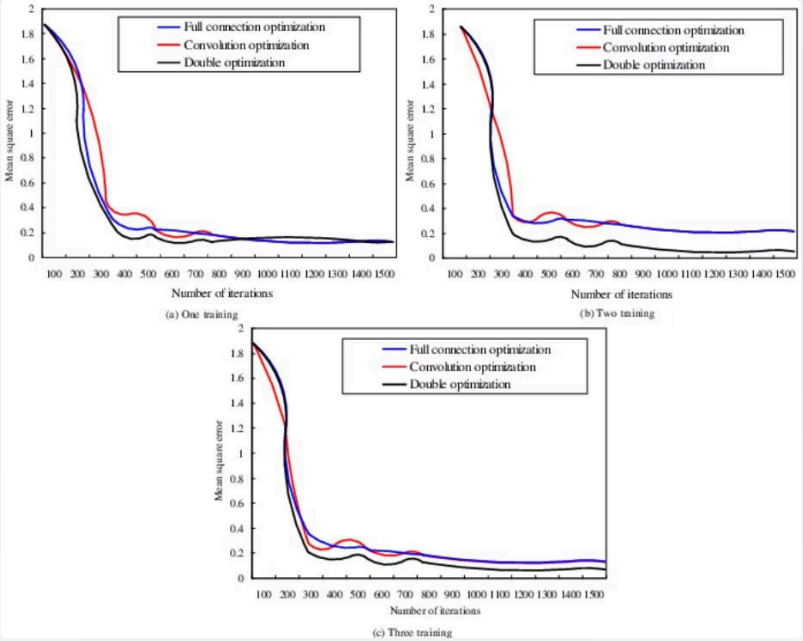
Dataset Details

MNIST dataset: 28x28 grayscale images of digits 0-9

59,000 training samples, 1,000 validation samples

Data normalized and shuffled for balanced training

Performance Results and Operation-Level Analysis



Metric	CPU (NumPy)	GPU (CuPy)	Speedup
Total Training Time	42.3 s	15.1 s	2.8x
Average Time per Iteration	84.6 ms	30.2 ms	2.8x
Forward Pass Time	36.7 ms	11.9 ms	3.1x
Backward Pass Time	47.9 ms	18.3 ms	2.6x

Matrix multiplication shows the highest speedup (6.3x), while activation functions have modest gains due to lower computational intensity.