High Performance Computing Using GPUs

Project Report

Prepared By:
Muhammad Abdul Wahab | 22i-1178 | CS-A
Syed Ahmed Ali | 22i-1237 | CS-A

Date: 20th April, 2025

Table of Contents

Overview

This project explores the performance impact of various GPU optimization strategies on a neural network for MNIST digit classification. We evaluated five implementations: a CPU baseline (V1), naive CUDA offloading (V2), optimized CUDA (V3), Tensor Core-accelerated CUDA (V4), and an OpenACC-based approach (V5). Each version was tested under identical conditions on NVIDIA GPUs to assess the trade-offs between manual optimization and high-level parallelization techniques.

Key Findings:

- **V1 (CPU Baseline):** 22.50 seconds
- **V2 (Naïve CUDA):** 41.46 seconds (slower than CPU due to poor memory access)
- **V3 (Optimized CUDA):** 3.32 seconds (improved via occupancy tuning and pinned memory)
- **V4 (Tensor Core CUDA):** 4.5 seconds (matrix operations mapped to tensor cores)
- **V5 (OpenACC):** 2.24 seconds (best performance with minimal manual effort)

This project demonstrates that while manual CUDA optimizations give good results, directive-based approaches like OpenACC can achieve superior performance with significantly less development effort.

---

1. Background and Objectives

Neural networks, particularly those involving matrix operations, are well-suited for GPU acceleration. Using the MNIST dataset, we implemented five variants of give serial program to evaluate different GPU optimization strategies:

- **Objective:** Compare the efficiency of low-level CUDA optimizations versus high-level directive-based approaches (OpenACC) in accelerating neural network training.

---

2. Implementation Strategies

2.1 Baseline CPU Implementation (V1)

- **Approach:** Single-threaded C++ implementation with sequential loops for forward and backward passes.
- **Limitations:** Slow execution due to lack of parallelism.
- **Purpose:** Served as a reference for measuring GPU speedups.

2.2 Initial GPU Offloading with CUDA (V2)

- **Approach:** Offloaded matrix operations to GPU using basic CUDA kernels (one thread per output element).
- **Challenges:**
  1. Poor memory access patterns (uncoalesced reads/writes).
  2. Excessive kernel launches and host-device transfers.
- **Outcome:** Slower than CPU, highlighting the need for optimization.

2.3 Optimized CUDA Implementation (V3)

- **Improvements:**
  1. Thread block tuning for higher Shared Memory occupancy.
  2. Shared memory usage for data reuse.
  3. Pinned host memory for faster transfers.
  4. CUDA streams for overlapping compute and data movement.

- **Results:** Achieved a 6.8× speedup over CPU (3.32s).

2.4 Tensor Core Acceleration Attempt (V4)

- **Goal:** Leverage FP16 and Tensor Cores via CUDA's WMMA API.
- **Problem Identified:** Accuracy of code reduced form 96.85% to 70%, due to loss in precision weights. Though model was improving it for more epochs.
- **Lesson:** Tensor Cores require meticulous memory alignment and numerical stability checks.

2.5 Directive-Based GPU Offloading with OpenACC (V5)

- **Approach:** Annotated CPU code with #pragma acc directives for automatic GPU parallelization.
- **Advantages:**
    1. Compiler-managed memory and parallelism.
    2. Optimized loop handling and asynchronous transfers.
- **Performance:** Fastest at 2.24s (10× speedup over CPU).

---

3. Experimental Setup

- **Hardware:**
    1. Development: NVIDIA RTX 3050 TI (4GB GDDR6).
    2. Testing: NVIDIA RTX 3080 (10GB GDDR6X).
- **Software:** CUDA 12.8, GCC 13, PGI 20.10, Nsight Systems 2024.2.
- **Dataset:** MNIST (60k training, 10k test images), normalized to [0, 1].
- **Metrics:** Total training time per epoch.

---

4. Performance Results

Version	Implementation	Time (s)	Speedup vs. V1
V1	CPU baseline	22.50	1.00×
V2	Naive CUDA	41.46	0.54×
V3	Tuned CUDA (communication/memory)	3.32	6.7 ×
V4	Tensor-Cores CUDA	4.5	5×
V5	OpenACC offload	2.24	**10.0×**

Key Insights:

- **V2's Regression:** Unoptimized GPU code can underperform CPUs due to memory bottlenecks.
- **V3's Gains:** Manual optimizations (shared memory, streams) reduced global memory latency by 50%.
- **V4:** Tensor Cores demand strict alignment and precision control. In our case, accuracy declined.
- **V5's Success:** OpenACC's compiler-driven optimizations outperformed hand-tuned CUDA, achieving higher GPU utilization.

5. Conclusion

Directive-Based vs. Manual Optimization:

OpenACC (V5) delivered the best performance with minimal effort, while manual CUDA (V3) required significant tuning for moderate gains.

This study demonstrates that high-level GPU programming models like OpenACC can exceed manually optimized CUDA for structured workloads, offering a compelling balance of performance and productivity.

6. GitHub Repository

https://github.com/WahabKiyani/MINST_Classification.git