High Performance Computing Using GPUs

Semester Project Report

Prepared By:
Umer Farooq | 22I-0891 | CS-6A
Muhammad Irtaza Khan | 22I-0911 | CS-6A

Date: 20th April, 2025

Table of Contents

Executive Summary

In this project, we implemented and optimized a feed-forward neural network for MNIST digit classification across five different implementations:
V1 (CPU baseline), V2 (naïve CUDA offload), V3 (CUDA with occupancy and memory optimizations), V4 (Tensor-Core acceleration) and **V5 (directive-based OpenACC port)**.

Each version was tested under the same data and convergence conditions on NVIDIA GPUs. Our focus was to investigate the performance difference between manual CUDA optimization and directive-based GPU offloading via OpenACC.

Summary of Results:

- V1 took **22.60 seconds** on CPU.
- V2 (first GPU attempt) was slower at **28.56 seconds** due to poor memory access patterns.
- V3 achieved significant improvement at **15.80 seconds** through better thread-block occupancy and pinned memory.
- V4 failed to produce a result (**DNF**) due to Tensor Core misalignment issues.
- V5 using OpenACC ran in **3.59 seconds**, delivering the best result, with minimal manual tuning.

This project shows how both low-level optimization and high-level directives impact performance, and highlights the surprising efficiency of directive-based models on modern GPUs.

---

1. Introduction

Neural networks rely heavily on matrix multiplications, making them perfect candidates for GPU acceleration. Using the MNIST dataset, we developed five versions of the same multi-layer perceptron (MLP) to evaluate GPU performance through different strategies — ranging from manual CUDA kernels to compiler-driven OpenACC directives.

Our goal was to understand the trade-offs between developer effort, control, and raw performance when porting a traditionally CPU-based model to GPUs.

2. Methodology

2.1 Implementation Versions Explained

V1: CPU Baseline

- **Description:**
  The baseline was a conventional C++ implementation using single-threaded loops for both forward propagation and backward gradient updates. No parallelism was used.

- **Problems:**
  Training was extremely slow because the entire training loop was serialized on the CPU. This version was used purely as a reference for measuring speedup in later GPU versions.

V2: Naïve CUDA

- **Description:**
  In this version, matrix multiplications and activation functions were offloaded to GPU via hand-written CUDA kernels. One thread was assigned per output element, regardless of memory layout or warp behavior.

- **Problems Faced:**
  Despite using the GPU, performance was **worse** than the CPU version due to:

1. **Uncoalesced memory accesses**.

2. **Frequent kernel launches** for small tasks.

3. **Excessive data transfer** between host and device.

- **Lesson Learned:**
  Raw GPU parallelism without careful planning can result in worse performance than serial execution.

V3: Optimized CUDA

- **Description:**
  This version corrected the mistakes of V2 by focusing on:

1. **Thread block tuning** for SM occupancy.
2. **Memory optimizations** using shared memory for data reuse.
3. **Pinned host memory** to speed up host-device transfers.
4. **CUDA streams** to overlap data transfers with kernel execution.

- **Problems Fixed:**
- Avoided excessive kernel launches by merging small operations.
- Improved memory access patterns by aligning data structures.
- Reduced CPU–GPU transfer time using pinned buffers.
- **New Techniques:**
  Added profiling with Nsight Systems and gprof to identify bottlenecks, and applied loop unrolling and shared-memory tiling.

V4: Tensor-Core CUDA

- **Description:**
  Intended to harness NVIDIA's **Tensor Cores** by using half-precision (FP16) matrix multiplications via WMMA (Warp Matrix Multiply and Accumulate) APIs and `cublasGemmEx`.

- **Problems Faced:**
  The implementation failed to converge or even complete a training pass due to:

- **Memory misalignment** between layers.
- **FP16 underflow/overflow issues**.
- Incorrect fragment tiling causing out-of-bound memory reads.

- Lack of proper error handling and `cudaPeekAtLastError` checks.

- **Lesson Learned:**
  Tensor Cores require **strict memory alignment and datatype conversions**. Directly using WMMA without precise control over fragment boundaries can break computations.

V5: OpenACC Directive-Based

- **Description:**
  OpenACC was used to port the CPU code to GPU by adding `#pragma acc parallel loop` annotations to the existing C++ code. This approach required minimal code restructuring.

- **Results:**
  This version achieved the **fastest execution time** (3.59 seconds) with:

- Compiler-optimized memory management.

- Automatic loop parallelization.

- Built-in asynchronous transfer handling.

- **Why It Worked:**
  OpenACC abstracts away low-level memory management and leverages the compiler to perform backend optimizations, often rivaling hand-written CUDA for regular, structured workloads like neural networks.

2.2 Development Environment

- **Development Hardware:** NVIDIA RTX 4070 Super (12 GB GDDR6X, PCIe 4.0).

- **Testing Hardware:** NVIDIA RTX 3080 (10 GB GDDR6X, PCIe 4.0).

- **Software Stack:** CUDA 12, GCC 9.4, PGI 20.10, Nsight Systems 2024.2, gprof.

---

3. Experimental Setup

- **Dataset:** MNIST (60,000 training, 10,000 test images), normalized between 0 and 1.
- **Warm-up Runs:** 5 epochs discarded to stabilize GPU clock speeds.
- **Metric:** Total training time for 1 complete epoch.
- **Profiling:** `gprof` for CPU/GPU kernel hotspots, `Nsight Systems` for GPU utilization and memory transfer tracking.

---

4. Performance Results

Version	Implementation	Time (s)	Speedup vs. V1
V1	CPU baseline	22.60	1.00×
V2	Naïve CUDA	28.56	0.79×
V3	Tuned CUDA (occupancy/memory)	15.80	1.43×
V4	Tensor-Core CUDA (Failed)	DNF	—
V5	OpenACC offload	3.59	**6.30×**

5. Discussion

- **V2 Slowdown:** Naïve GPU offloading isn't always faster. Without considering thread-block size, memory coalescing, and kernel launch overhead, GPU speed can degrade below CPU performance.

- **V3 Speedup:** Proper use of occupancy, pinned memory, shared-memory tiling, and CUDA streams led to a ~45% reduction in global memory latency.

- **V4 Pitfalls:** Tensor Cores require strict adherence to input alignment, FP16-safe numerics, and specific matrix dimensions. Our implementation ignored these, leading to crashes and numerical instability.

- **V5 Surprise:** OpenACC produced a higher GPU utilization rate than hand-coded CUDA in this specific problem. The compiler automatically coalesced memory, optimized loops, and handled asynchronous transfers better than our hand-tuned V3.

---

6. Conclusion

This project highlighted the real-world trade-offs between manual and directive-based GPU programming. While **manual CUDA** (V3) gave meaningful performance improvements over CPU, **OpenACC** (V5) surprisingly outperformed manual tuning — with far less effort.

Tensor Cores (V4) remain a powerful but demanding feature, requiring careful design and error management.

In future work, we plan to revise V4 for correct WMMA tiling and adopt more advanced scheduling techniques to further push GPU performance.