MNIST Digit Classification: Neural Network Optimization Analysis

Team: Aisha Siddiqa, Hania Aamer, Ayesha Ejaz

Section: B

Contents

3. Version 1: Sequential CPU Baseline

- **3.1. Implementation Summary**

    - Standard C implementation using nested loops for matrix operations. Dynamic memory allocation for weights/biases. Sequential execution of forward pass, backpropagation, and weight updates.
- **3.2. Workflow & Execution**
    - Load data -> Initialize network -> Loop through 3 epochs -> For each sample: Forward -> Loss -> Backward -> Update -> Evaluate on test set. Compiled and run via `make`, profiled with `gprof`.

- **3.3. Performance Metrics (V1)**
    - **Total Training Time:** 42.391s (Average ~14.133/epoch)
    - **Test Accuracy:** 97.15%
- **3.4. Bottleneck Identification (V1)**
    - `gprof` profiling confirmed the CPU-bound nature: `forward` (~61.5%) and `backward` (~36.1%) functions accounted for over 97% of runtime due to loop-based matrix math. The core issue was the **lack of parallelism**.

---

4. Version 2: Naive GPU Implementation

- **4.1. Optimization Goal & Strategy**
    - **Objective:** Leverage GPU parallelism via CUDA for initial speedup.
    - **Approach:** Ported core computations (`forward`, `backward`) to CUDA kernels. Managed memory explicitly on host and device. Maintained per-sample processing logic.
- **4.2. Implementation Summary**
    - CUDA C/C++ implementation. Kernels written for matrix multiplication, activations, and gradient steps. Used standard `cudaMalloc` and `cudaMemcpy` for memory management.
- **4.3. Workflow & Execution**
    - Load data (Host) -> Init weights (Host) & Copy to Device -> Loop Epochs -> For each sample: **Copy Input/Label Host->Device** -> Launch Kernels (Forward, Backward, Update) -> **Synchronize** -> Copy necessary results Device->Host -> Evaluate (likely similar per-sample transfers).
- **4.4. Performance Metrics (V2)**
    - **Total Training Time: 36.470** (~12.8133/epoch)
    - **Test Accuracy: 96.70%** (0.45% change from V1)
    - **Speedup vs V1: ~1.162x**
- **4.5. Identified Bottlenecks (V2)**

- **Data Transfer Overhead:** Profiling (e.g., with Nsight Systems) revealed significant time spent in `cudaMemcpy` operations. Transferring each sample individually created a severe bottleneck, limiting the benefits of GPU computation.
- **GPU Underutilization:** Processing single samples failed to saturate the GPU's parallel processing capabilities.
- **Kernel Launch Overhead:** Repeatedly launching kernels for every sample added cumulative overhead.

---

5. Version 3: Optimized GPU Implementation

- **5.1. Optimization Strategy**
  - **Objective:** Overcome V2's data transfer bottleneck and improve GPU utilization.
  - **Key Techniques Implemented:**
    - **Batch Processing:** Processed data in batches of 128 samples.
    - **Memory Coalescing:** Ensured input data and weights were arranged in row-major order for contiguous access by threads within a warp.
    - **Reduced Synchronization:** Synchronized primarily at batch boundaries rather than per sample.
    - **(Optional) Shared Memory:** Matrix multiplication kernels potentially used shared memory tiling to reduce global memory accesses.
- **5.2. Implementation Details**
  - Kernels were modified to operate on batches. Larger, pre-allocated device buffers held batch data. Data layout optimized for coalescing. . Batch size set to 128.
- **5.3. Workflow Changes**
  - Data transfer now occurred once per *batch* (128 samples). Kernels processed the entire batch in parallel. Host-Device communication significantly reduced.
- **5.4. Observed Bottlenecks (V3)**
  - **Compute Bound (Kernel Efficiency):** With data transfer reduced, the primary bottleneck became the execution time of the CUDA kernels themselves. Custom kernels, while parallel, might not reach the theoretical peak performance of the hardware.
  - **Precision Limitations:** If still using double precision, it inherently limited computational throughput compared to single precision.
- **5.5. Improvements Over V2**
  - Massive reduction in execution time due to eliminating the per-sample transfer bottleneck. GPU utilization drastically improved. Memory bandwidth usage became more efficient.
  - **Total Training Time: 22.248s** (~7.549s/epoch)

    - **Test Accuracy: 97.07%** (stable, perhaps minor variation)
    - **Speedup vs V2: ~39% or 1.64x**
    - **Speedup vs V1: ~47.52% or 1.91x**

---

6. Version 4: Advanced GPU Optimization (Tensor Core Leveraged)

- **6.1. Optimization Strategy**
    - **Objective:** Maximize performance using specialized hardware features and libraries.
    - **Key Techniques Implemented:**
        - **cuBLAS Integration:** Replaced custom matrix multiplication kernels with highly optimized `cublasSgemm` calls.
        - **Single Precision (FP32) & TF32:** Switched computations primarily to single precision (`float`). cuBLAS on compatible GPUs (Ampere+) likely used TF32 precision internally for matrix multiplication via Tensor Cores, offering speed close to FP16 with FP32's numerical range.
        - **Increased Batch Size:** Raised batch size to 256 to better utilize Tensor Cores and further amortize fixed overheads.
        - **CUDA Streams:** Employed multiple CUDA streams to potentially overlap data transfers (using `cudaMemcpyAsync`) with kernel execution.
- **6.2. Implementation Details**
    - Linked against the cuBLAS library. Code modified to call `cublasSgemm`. Data types changed to `float`. Implemented asynchronous copies and kernel launches across ~4 streams. Xavier initialization might have been used for better convergence with the larger learning rate/batch size. Learning rate likely increased (e.g., to 0.05 or 0.1).
- **6.3. Workflow Changes**
    - Matrix multiplications handled by cuBLAS. Data transfers and kernel launches pipelined across streams where possible. Larger batches processed per iteration.
- **6.4. Observed Bottlenecks & Trade-offs (V4)**
    - **Accuracy Impact:** The use of TF32 precision (implicitly by cuBLAS) and potentially a higher learning rate resulted in a noticeable drop in test accuracy compared to previous versions. This highlights the speed-accuracy trade-off.

- **Non-Optimized Kernels:** Custom kernels for activations (ReLU), bias addition, and gradient calculations (non-matmul parts) became relatively more significant bottlenecks compared to the highly accelerated cuBLAS operations.
- **Stream Effectiveness:** Actual overlap achieved via streams might be limited if kernels are very fast or if dependencies exist.
- **Memory Footprint:** Larger batch size (256) increased GPU memory consumption.

- **6.5. Improvements Over V3**
  - Further substantial speedup achieved, primarily from Tensor Core acceleration via cuBLAS and the switch to single precision. Stream usage provided some additional latency hiding.
  - **Total Training Time:** (~/epoch)
  - **Test Accuracy: %**
  - **Speedup vs V3: ~**
  - **Speedup vs V1: ~**

---

7. Comparative Performance Analysis

- **7.1. Execution Time Summary**

Version	Total Training Time (s)	Speedup (vs Previous)	Speedup (vs V1 Baseline)
V1 (Sequential CPU)	42.391	-	1.00x
V2 (Naive GPU)	36.470	**~1.162x**	**~1.162x**
V3 (Optimized GPU - Batching)	22.248	**~ 1.64x**	**~1.91x**
V4 (Advanced GPU - cuBLAS/Tensor Core)	0.055	**~**	**~**