## **Optimizing MNIST Classification Using CUDA**

The MNIST dataset, consisting of 28×28 grayscale images of handwritten digits (0–9), is a widely used benchmark for evaluating image classification algorithms. The goal of this project is to implement and optimize a neural network-based solution for classifying MNIST digits, with a focus on leveraging NVIDIA GPUs through CUDA to significantly accelerate training and inference. The optimization process is approached incrementally, with each version introducing performance improvements or architectural enhancements. The starting point is a basic sequential CPU implementation, which serves as a reference for evaluating the gains achieved in subsequent CUDA-based versions.

### **V1 Sequential Implementation**

The initial version (V1) of the neural network for MNIST classification is a straightforward sequential implementation written in C. It features a single hidden-layer architecture with 128 neurons using ReLU activation and a softmax-activated output layer for multi-class classification. All computations, including matrix operations and gradient updates, are performed using nested loops without any parallelism or hardware acceleration. Data is loaded directly from the MNIST binary files, with pixel values normalized and labels one-hot encoded. Training is conducted using stochastic gradient descent over individual samples, and accuracy and loss are reported per epoch. While functional and clear, this version prioritizes simplicity over performance and does not yet utilize batching, vectorization, or optimized numerical libraries.

Execution time for sequential Implementation is Approximately **23.479 seconds**

### **GPU Implementation - Version 2 (Naive Approach)**

In this version of the neural network, computation is offloaded to the GPU using CUDA, allowing parallel execution of core operations such as matrix multiplication, activation functions, softmax, and gradient computations. However, this version is implemented without advanced memory or kernel-level optimizations, serving as a functional but unrefined baseline for GPU-accelerated training.

### **Objective**

The primary goal of Version 2 is to leverage CUDA to parallelize the forward and backward passes of a basic two-layer feedforward neural network. This implementation is meant to establish a working prototype on GPU while identifying performance bottlenecks to be addressed in future optimized versions.

### **Architecture and Design**

**Network Topology:**

* Input Layer: 784 units (28×28 MNIST images)
* Hidden Layer: 128 neurons using ReLU activation
* Output Layer: 10 neurons using Softmax activation
* Loss Function: Cross-entropy

**CUDA Kernels Used:**

1. *matrixMulKernel*: Computes dot products for both hidden and output layer activations.
2. *reluKernel*: Applies ReLU element-wise on the hidden layer.
3. *softmaxKernel*: Applies softmax on the output layer using shared memory for reduction.
4. *backwardOutputKernel* and *computeGradients*: Computes gradient updates for weights and biases.
5. *updateParameters*: Applies gradient descent updates.

**Batching: Training is performed on a per-sample basis rather than using batched matrix operations, which significantly affects parallelism and memory coalescing.**

### **Notable Implementation Details**

* **Matrix Multiplication:** The matrix multiplication kernel uses a simple nested loop inside each thread for computing the dot product. This avoids use of shared memory and thus is memory-bound and inefficient for large datasets.
* **Softmax Optimization:** The softmax kernel performs two critical reductions — one for the maximum value and another for normalization. It uses shared memory but does not exploit warp-level primitives or multi-block reduction.
* **Atomic Operations:** Gradient accumulation in the backward pass uses atomicAdd to prevent race conditions. While this ensures correctness, it significantly limits scalability when using larger models or batch sizes due to serialization overhead.
* **Parameter Updates:** Weights and biases are updated directly on the GPU using a simple subtractive update rule.
* **Memory Management:** The network initializes and copies all weights, biases, and activations between host and device memory at the beginning. Temporary device buffers are used per sample during training and evaluation and freed after use.

### **Limitations**

1. **Lack of Batching:** Training one image at a time limits the benefit of GPU parallelism, especially during matrix multiplications and gradient updates.
2. **Memory Access Patterns:** No memory coalescing, caching, or shared memory optimizations are implemented.
3. **AtomicAdd Overhead:** Heavy reliance on *atomicAdd* can become a major bottleneck under high parallel loads.
4. **Static Block Size:** Hardcoded *BLOCK\_SIZE* may not be optimal across different GPUs or kernels.
5. **Redundant Device-Host Transfers:** Frequent memory transfers to evaluate loss and accuracy could be minimized using device-side aggregations.

### **Results and Observations**

Contrary to initial expectations, the naive GPU implementation in Version 2 underperforms when compared to the sequential CPU-based Version 1. The training process consistently takes 2 to 3 seconds longer per epoch, primarily due to several critical inefficiencies:

* **Frequent Memory Transfers:** Excessive host-to-device and device-to-host memory transfers introduce significant latency, especially during forward and backward passes for each individual training sample.
* **Lack of Batching:** The absence of mini-batch processing prevents effective utilization of the GPU’s parallel processing capabilities. Running inference and backpropagation on a single sample at a time leads to underutilized threads and idle compute resources.
* **Kernel Launch Overheads:** Numerous lightweight kernel launches, particularly for small vector operations or reductions, accumulate substantial overhead over time.
* **Inefficient Memory Access Patterns:** No memory coalescing or shared memory usage is employed, resulting in scattered and slow global memory accesses.

These issues collectively negate the benefits of parallel computation and result in a slower overall execution time. While the implementation demonstrates functional correctness and successful execution on GPU, it highlights the importance of thoughtful data movement and kernel design in CUDA programming.