NEURAL NETWORK

OPTIMIZATION USING CUDA & TENSOR CORES

Presented by
MUHAMMAD OWAIS
HASSAN WAQAR
MAHAD KHAN

TABLE OF CONTENTS

2. NEURAL NETWORK ARCHITECTURE

- The neural network is a feedforward model with:
- Input Layer: 784 neurons (28×28 pixel images).
- Hidden Layer: 128 neurons with ReLU activation.
- Output Layer: 10 neurons with softmax activation (one for each digit 0-9).
- Training Parameters:
- Learning rate: 0.01 (V1, V2, V3); 0.1 (V4, adjusted for single precision stability).
- Epochs: 3.
- Batch size: 64 (V1, V2, processes one sample at a time); 128 (V3); 256 (V4).

OPTIMIZED VERSIONS

SEQUENTIAL

CPU-based
base line
high computation
No parallelism

VERSION 2

Naive GPU
parallelism
bottleneck
communication
overhead

VERSION 3

optimized CUDA
shared memory
streams
cudamemcpyAsync
Batch processing

VERSION 4

tensor cores
Single Precision
(float):
cuBLAS
cublasSgemm

CPU SEQUENTIAL IMPLEMENTATION

KEY POINTS

- Matrix Operations: Sequential nested loops.
- Memory Management: Uses dynamically allocated 2D arrays
-

WORKFLOW

1. Load MNIST data.
2. Initialize weights/biases.
3. Per-sample: Forward pass, compute loss, backward pass, update weights.
4. Evaluate on test set.

APPROACH

- Forward Pass: Performs matrix multiplication and activation functions (ReLU, softmax) sequentially.
- Backward Pass: Computes gradients and updates weights sequentially.

ISSUES

- Slow execution (48.023s for 3 epochs).
- High memory allocation overhead.
- No parallelism.

CPU SEQUENTIAL IMPLEMENTATION

EXECUTION TIME

Version	Total Training Time (s)	Speedup (Relative to Previous)	Test Accuracy (%)
V1 (Sequential)	48.023	-	96.75

OPTIMIZED VERSIONS

SEQUENTIAL

CPU-based
base line
high computation
No parallelism

VERSION 2

Naive GPU
parallelism
bottleneck
communication
overhead

VERSION 3

optimized CUDA
shared memory
streams
cudamemcpyAsync
Batch processing

VERSION 4

tensor cores
Single Precision
(float):
cuBLAS
cublasSgemm

GPU NAIVE IMPLEMENTATION

KEY POINTS

- Transition computations to GPU.
- Kernel Fusion: Two kernels (fused_forward_kernel, fused_backward_kernel) for matrix operations
- CUDA Parallelism: GPU threads for matrix operations.
- Device Storage: Weights/biases on GPU.
-

APPROACH

- Kernels:
- fused_forward_kernel: Matrix multiplication, ReLU, softmax (1D blocks, 128 threads).
- fused_backward_kernel: Gradient computation, weight updates (2D blocks, 32×32).
- Memory: Parameters on host and device; per-sample input transfers.
- Synchronization: __syncthreads() within blocks, cudaDeviceSynchronize() for host-device.

GPU NAIVE IMPLEMENTATION

FUSED FUNCTIONS

WHY????

```
__global__ void fused_forward_kernel(double* W1, double* b1, double* W2, double* b2,
                                     double* input, double* hidden, double* output,
                                     double* temp, double* softmax_sum,
                                     int input_size, int hidden_size, int output_size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
__global__ void fused_backward_kernel(double* W1, double* b1, double* W2, double* b2,
                                      double* input, double* hidden, double* output,
                                      double* target, double* d_output, double* d_hidden,
                                      double* temp, double learning_rate,
                                      int input_size, int hidden_size, int output_size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
```

GPU NAIVE IMPLEMENTATION

KERNEL LAUNCH

```
void backward(NeuralNetwork* net, double* d_input, double* d_hidden, double* d_output,
              double* d_target, double* d_d_output, double* d_d_hidden, double* d_temp) {
    dim3 blockDim(32, 32);
    dim3 gridDim((INPUT_SIZE + 31) / 32, (HIDDEN_SIZE + 31) / 32);
    fused_backward_kernel<<<gridDim, blockDim>>>(net->d_W1, net->d_b1, net->d_W2, net->d_b2,
                                                 d_input, d_hidden, d_output, d_target,
                                                 d_d_output, d_d_hidden, d_temp,
                                                 LEARNING_RATE, INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE);
```

```
void forward(NeuralNetwork* net, double* d_input, double* d_hidden, double* d_output,
             double* d_temp, double* d_softmax_sum) {
    int threadsPerBlock = 128;
    int blocksPerGrid = (HIDDEN_SIZE + threadsPerBlock - 1) / threadsPerBlock;
    fused_forward_kernel<<<blocksPerGrid, threadsPerBlock>>>(net->d_W1, net->d_b1, net->d_W2, net->d_b2,
                                                             d_input, d_hidden, d_output,
                                                             d_temp, d_softmax_sum,
                                                             INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE);
```

GPU NAIVE IMPLEMENTATION

__SYNCTHREADS()

```
__global__ void fused_forward_kernel(double* W1, double* b1, double* W2, double* b2,
                                     double* input, double* hidden, double* output,
                                     double* temp, double* softmax_sum,
                                     int input_size, int hidden_size, int output_size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < hidden_size) {
        double sum = 0.0;
        for (int j = 0; j < input_size; j++) {
            sum += W1[idx * input_size + j] * input[j];
        }
        hidden[idx] = (sum + b1[idx] > 0) ? (sum + b1[idx]) : 0.0;
    }

    __syncthreads();

    if (idx < output_size) {
        double sum = 0.0;
        for (int j = 0; j < hidden_size; j++) {
            sum += W2[idx * hidden_size + j] * hidden[j];
        }
        output[idx] = sum + b2[idx];
        temp[idx] = exp(output[idx]);
    }

    __syncthreads();
```

GPU NAIVE IMPLEMENTATION

MEMORY MANAGEMENT

```
for (int i = 0; i < numImages; i++) {
    cudaMemcpy(d_input, images[i], INPUT_SIZE * sizeof(double), cudaMemcpyHostToDevice);
    forward(net, d_input, d_hidden, d_output, d_temp, d_softmax_sum);
    cudaDeviceSynchronize();

    cudaMemcpy(h_output, d_output, OUTPUT_SIZE * sizeof(double), cudaMemcpyDeviceToHost);
```

Copies outputs back to the host (cudaMemcpy) for loss/accuracy computation, adding overhead.

GPU NAIVE IMPLEMENTATION

WORKFLOW

- Load MNIST data into host memory.
- Initialize weights and biases, copy to device.
- For each sample:
- Transfer input and target to device.
- Execute fused_forward_kernel.
- Compute loss, execute fused_backward_kernel.
- Transfer output to host for evaluation.
- Synchronize after each sample.
- Evaluate on test set.

ISSUES

- Data Transfers: Per-sample host-device transfers dominate (18.789s total).
- GPU Underutilization: Single-sample processing limits parallelism.
- Double Precision: Slows computation.
- Synchronization Overhead: Frequent cudaDeviceSynchronize().
- Memory Access: Non-coalesced accesses reduce efficiency.

GPU NAIVE IMPLEMENTATION

EXECUTION TIME

Version	Total Training Time (s)	Speedup (Relative to Previous)	Test Accuracy (%)
V2 (NAIVE)	48.023	-	96.75
V2 (NAIVE)	18.789	~2.56x (vs. V1)	95.64

OPTIMIZED VERSIONS

SEQUENTIAL

CPU-based
base line
high computation
No parallelism

VERSION 2

Naive GPU
parallelism
bottleneck
communication
overhead

VERSION 3

optimized CUDA
shared memory
streams
cudamemcpyAsync
Batch processing

VERSION 4

tensor cores
Single Precision
(float):
cuBLAS
cublasSgemm

GPU OPTIMIZED IMPLEMENTATION

KEY POINTS

- Batch Processing: 128 samples/batch.
- Memory Coalescing: Row-major data for coalesced access.
- Shared Memory: Cache intermediate results.
- Optimized Threads: 2D blocks (16×16).
- Reduced Synchronization: Sync at batch boundaries.
- Kernels:
- Matrix multiplication with shared memory (2D blocks, 16×16).
- Synchronization: __syncthreads() in kernels, reduced cudaDeviceSynchronize().

WORKFLOW

1. Load MNIST data.
2. Initialize weights/biases, allocate batch buffers.
3. Per epoch:
4. Shuffle data.
5. Process batches (128 samples): Copy inputs/labels, run forward/backward kernels, update weights.
6. Synchronize at epoch end.
7. Evaluate in batches.

GPU OPTIMIZED IMPLEMENTATION

ISSUES

- Double Precision: Limits speed.
- Batch Size: 128 samples still requires transfers (0.432s total).
- No Tensor Cores: Custom kernels lack tensor core acceleration.
- Synchronization: Required at batch boundaries.
- Shared Memory Overhead: Adds complexity and minor latency.

V3 vs V2

1. Batch processing reduces transfer overhead.
2. Coalesced memory access improves bandwidth.
3. Shared memory lowers global memory latency.
4. Reduced synchronization via batch processing.

GPU OPTIMIZED IMPLEMENTATION

EXECUTION TIME

Version	Total Training Time (s)	Speedup (Relative to Previous)	Test Accuracy (%)
V2 (NAIVE)	18.789	~2.56x (vs. V1)	95.64
V3 OPTIMIZED	0.432	~43.49x (vs. V2)	95.01

OPTIMIZED VERSIONS

SEQUENTIAL

CPU-based
base line
high computation
No parallelism

VERSION 2

Naive GPU
parallelism
bottleneck
communication
overhead

VERSION 3

optimized CUDA
shared memory
streams
cudamemcpyAsync
Batch processing

VERSION 4

tensor cores
Single Precision
(float):
cuBLAS
cublasSgemm

TENSOR CORE IMPLEMENTATION

KEY POINTS

- Batch Processing: 256 samples/batch.
- Tensor Cores: cuBLAS with TF32 precision.
- Single Precision: Reduces computation time.
- CUDA Streams: Four streams for overlapping operations.
- Memory Coalescing: Row-major aligned data.
- AsynchronousTransfers: cudaMemcpyAsync.
- cublasSgemm for matrix multiplications.
- Synchronization: Streams sync at epoch end.
- Softmax: Host-based with asynchronous transfers.

WORKFLOW

- Load MNIST data into host memory.
- Initialize weights and biases, allocate batch buffers on device.
- For each epoch:
- Shuffle training data indices.
- Process batches (128 samples):
- Copy batch inputs and labels to device.
- Execute forward pass using custom matrix multiplication, ReLU, and softmax kernels.
- Compute loss and execute backward pass kernels.
- Update weights and biases.
- Synchronize at epoch end.
- Evaluate on test set in batches.

TENSOR CORE IMPLEMENTATION

ISSUES

- Host-based softmax requires synchronization, adding overhead.
- Large batch buffers may cause memory constraints on low-end GPUs.
- Suboptimal stream overlapping limits full concurrency.
- TF32 precision introduces minor numerical tradeoffs, though mitigated by initialization.

V3 vs V2

- Batch processing with larger batch size (256) eliminates more data transfer overhead.
- Tensor core acceleration via cuBLAS boosts matrix operation performance.
- Single precision reduces computation time and memory usage.
- CUDA streams enable concurrent execution of operations.
- Optimized memory access patterns ensure coalesced global memory access.

TENSOR CORE IMPLEMENTATION

EXECUTION TIME

Version	Total Training Time (s)	Speedup (Relative to Previous)	Test Accuracy (%)
V2 (NAIVE)	18.789	~2.56x (vs. V1)	95.64
V3 OPTIMIZED	0.432	~43.49x (vs. V2)	95.01
V4 TENSOR	0.188	~2.30x (vs. V3)	93.11

SUMMARY AND KEY TAKEAWAYS:

Version 1 (Sequential CPU)

Sequential matrix operations on CPU with nested loops.
Time: 48.023s | Test Accuracy: 96.75%.

Version 2 (Naive GPU)

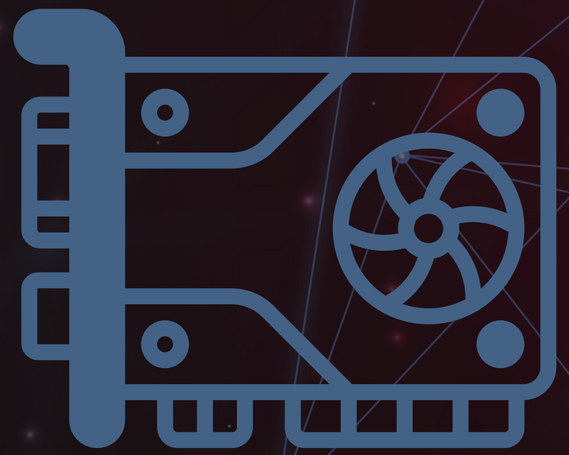
CUDA with fused kernels (fused_forward_kernel, fused_backward_kernel); 1D/2D blocks.
Time: 18.789s | Speedup: ~2.56x (vs. V1) | Test Accuracy: 95.64%.

Version 3 (Optimized GPU)

Batch processing (128 samples); shared memory, coalesced access, custom kernels; 2D blocks (16×16).
Time: 0.432s | Speedup: ~43.49x (vs. V2) | Test Accuracy: 95.01%.

Version 4 (Tensor Core Optimized)

Tensor cores with cuBLAS (TF32); single precision, 256-sample batches, 4 CUDA streams; custom kernels.
Time: 0.188s | Speedup: ~2.30x (vs. V3) | Test Accuracy: 93.11%.



THANK YOU