CS-4110

High Performance Computing with GPU

Neural Network Acceleration on GPUs: MNIST Classification

Authors:

Aneeq Ahmed Malik
Abdullah Mehmood
Hamza Saleem

April 20, 2025

Contents

1. Abstract

This project optimizes a 784-128-10 neural network for MNIST digit classification (70,000 images) using CUDA, targeting a batch size of 64. The sequential CPU implementation (V1) required 23.04 seconds for 3 epochs, limited by matrix multiplication bottlenecks. GPU-based versions progressively improved performance: V2's naive CUDA kernels yielded 1.10x speedup (20.91s), V3's optimizations (shared memory, streams, batch processing, float precision, batch evaluation) achieved 1.82x (12.66s), and V4's cuBLAS FP32 matrix operations delivered 3.81x (6.05s) with 96.82

2. Introduction

The MNIST dataset (60,000 training, 10,000 test images, 28×28 pixels) is a standard benchmark for neural networks. CPUs' sequential processing limits performance, as seen in V1's 784-128-10 network, which took 23.04 seconds for 3 epochs due to matrix multiplications. GPUs' massive parallelism accelerates such computations, enabling this project to optimize performance through four versions: V1 (sequential C), V2 (naive CUDA kernels), V3 (shared memory, streams, batch processing, float precision, batch evaluation), and V4 (cuBLAS FP32 matrix operations). This work showcases GPU acceleration for deep learning.

3. Methodology

The project implements a 784-128-10 neural network for MNIST classification, trained for 3 epochs on 60,000 images (batch size 64) and evaluated on 10,000 images. Four versions were developed: V1 used sequential C, V2 introduced CUDA kernels, V3 applied shared memory, streams, batch processing, float precision, and batch evaluation, and V4 leveraged cuBLAS FP32 matrix operations. Performance was measured by training/evaluation times, speedup relative to V1, and test accuracy, ensuring consistent comparisons across versions.

4. Version 1: Sequential Implementation

Version 1 (V1) is a C-based, CPU implementation of a 784-128-10 neural network for MNIST classification, using ReLU (hidden layer) and softmax (output layer). It processes images sequentially, performing matrix multiplications (‘W1‘: 128 x 784, ‘W2‘: 10 x 128) and applying SGD for forward/backward passes.

Memory and Computation. Data resides in CPU memory: ‘W1‘ (400 KB), ‘W2‘ (5 KB), biases, and inputs (3 KB/image). The forward pass multiplies input with ‘W1‘, adds bias, applies ReLU, then multiplies by ‘W2‘, adds bias, and applies softmax. The backward pass updates weights via SGD.

Performance. Profiling with ‘gprof‘ on an Intel CPU showed 23.04s for 3 epochs (22.23s training, 0.81s evaluation), with forward (66.10

Function	**% Time**	**Self (s)**	**Calls**
forward	66.10	28.58	190,000
backward	32.98	14.26	180,000
loadMNISTImages	0.79	0.34	2
train	0.07	0.03	1

Table 1: Profile of V1's key functions.

5. Version 2: Naive GPU Implementation

V2 ports ‘forward‘ and ‘backward‘ to CUDA, splitting ‘forward‘ into 3 kernels and ‘backward‘ into 4 kernels. The ‘NeuralNetwork‘ structure is allocated on the device with ‘cudaMalloc‘. Matrices (‘W1‘, ‘W2‘) are stored in row-major order.
The ‘train‘ function allocates device memory for ‘d_input‘, ‘d_hidden‘, ‘d_output‘, and ‘d_label‘, processing one image at a time.
Per-image ‘cudaMemcpy‘ (host-to-device for inputs/labels, device-to-host for outputs) and host-based loss/accuracy calculation add overhead.

```
__global__ void compute_hidden_layer(NeuralNetwork* net, const double* input,
    double* hidden) {
    if (threadIdx.x < HIDDEN_SIZE) {
        double sum = net->b1[i];
        for (int j = 0; j < INPUT_SIZE; j++){
            sum += net->W1[threadIdx.x * INPUT_SIZE + j] * input[j];
        }
        hidden[i] = (sum > 0.0) ? sum : 0.0;
    }
}
```

Listing 1: V2: Compute hidden layer

```
__global__ void compute_output_layer(NeuralNetwork* net, const double* hidden,
    double* output) {
    int i = threadIdx.x;

    if (i < OUTPUT_SIZE) {
        double sum = net->b2[i];
        for (int j = 0; j < HIDDEN_SIZE; j++) {
            sum += net->W2[i * HIDDEN_SIZE + j] * hidden[j];
        }
        output[i] = exp(sum);
    }
}
```

Listing 2: V2: Compute output layer

```
__global__ void normalize_softmax(double* output) {
    double sum = 0.0;
    for (int i = 0; i < OUTPUT_SIZE; i++) {
        sum += output[i];
    }
    for (int i = 0; i < OUTPUT_SIZE; i++) {
        output[i] /= sum;
    }
}
```

Listing 3: V2: Softmax normalization

```
__global__ void compute_d_output(const double* output, const double* target, double
    * d_output) {
    int i = threadIdx.x;
    if (i < OUTPUT_SIZE) {
        d_output[i] = output[i] - target[i];
    }
}
```

Listing 4: V2: Compute output gradient

```
__global | void compute_d_hidden(NeuralNetwork* net, const double* d_output, const
    double* hidden, double* d_hidden) {
    int i = threadIdx.x;
    if (i < HIDDEN_SIZE) {
        double grad = 0.0;
        for (int j = 0; j < OUTPUT_SIZE; j++) {
            grad += net->W2[j * HIDDEN_SIZE + i] * d_output[j];
        }
        d_hidden[i] = (hidden[i] > 0.0) ? grad : 0.0;
    }
}
```

Listing 5: V2: Compute hidden gradient

```
__global__ void update_output_weights(NeuralNetwork* net, const double* d_output,
    const double* hidden) {
    int i = blockIdx.x; // OUTPUT neuron index
    int j = threadIdx.x; // HIDDEN neuron index
    if (i < OUTPUT_SIZE && j < HIDDEN_SIZE) {
        int idx = i * HIDDEN_SIZE + j;
        net->W2[idx] -= LEARNING_RATE * d_output[i] * hidden[j];
    }
    if (j == 0 && i < OUTPUT_SIZE) {
        net->b2[i] -= LEARNING_RATE * d_output[i];
    }
}
```

Listing 6: V2: Update output weights

```
__global__ void update_hidden_weights(NeuralNetwork* net, const double* d_hidden,
    const double* input) {
     int i = blockIdx.x; // HIDDEN neuron index
     int j = threadIdx.x; // INPUT neuron index
     if (i < HIDDEN_SIZE && j < INPUT_SIZE) {
         int idx = i * INPUT_SIZE + j;
         net->W1[idx] -= LEARNING_RATE * d_hidden[i] * input[j];
     }
     if (j == 0 && i < HIDDEN_SIZE) {
         net->b1[i] -= LEARNING_RATE * d_hidden[i];
     }
}
```

Listing 7: V2: Update hidden weights

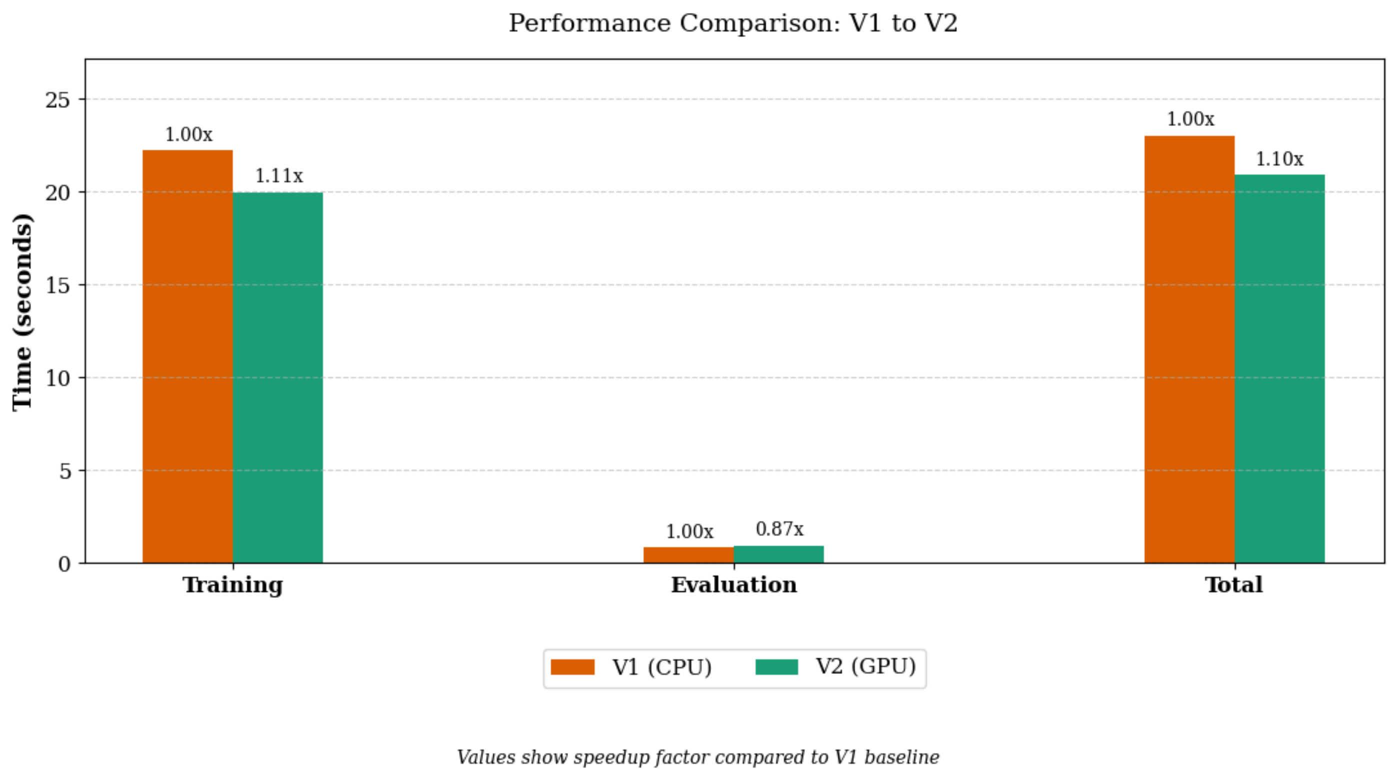

Figure 1: V2 Performance Analysis

Version	Training		Evaluation		Total		Test Accuracy (%)
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	
V1 (CPU)	22.23	–	0.81	–	23.04	–	96.85
V2 (GPU)	19.98	1.11x	0.93	0.87x	20.91	1.10x	96.83

Table 2: Performance comparison of V1 and V2 for 3 epochs, including later versions.

6. Version 3: Optimized GPU Implementation

V3 improves V2 with advanced optimization techniques to overcome the limitations of the naive GPU implementation. The primary optimizations include shared memory usage, CUDA streams, batch processing, float data type conversion, and batch evaluation.

6..1 Shared Memory Optimization

Shared memory significantly reduces global memory access latency. We implemented shared memory optimizations in key kernels:

```
__global__ void compute_hidden_layer(NeuralNetwork* net, const double* input,
    double* hidden) {
    int i = threadIdx.x;
    __shared__ double s_input[INPUT_SIZE];
    s_input[i] = input[i];
    __syncthreads();
    if (i < HIDDEN_SIZE) {
        double sum = net->b1[i];
        for (int j = 0; j < INPUT_SIZE; j++)
            sum += net->W1[i * INPUT_SIZE + j] * s_input[j];
        hidden[i] = (sum > 0.0) ? sum : 0.0;
    }
}
```

Listing 8: V3: Shared memory for Hidden Layer

```
__global__ void compute_output_layer(NeuralNetwork* net, const double* hidden,
    double* output) {
    int i = threadIdx.x;
    __shared__ double s_hidden[HIDDEN_SIZE];
    s_hidden[i] = hidden[i];
    __syncthreads();
    if (i < OUTPUT_SIZE) {
        double sum = net->b2[i];
        for (int j = 0; j < HIDDEN_SIZE; j++)
            sum += net->W2[i * HIDDEN_SIZE + j] * s_hidden[j];
        output[i] = exp(sum);
    }
}
```

Listing 9: V3: Shared memory for Output Layer

```
__global__ void compute_d_hidden(NeuralNetwork* net, const double* d_output, const
    double* hidden, double* d_hidden) {
    int i = threadIdx.x;
    __shared__ double s_output[OUTPUT_SIZE];
    if (i < OUTPUT_SIZE)
        s_output[i] = d_output[i];
    __syncthreads();
    if (i < HIDDEN_SIZE) {
        double grad = 0.0;
        for (int j = 0; j < OUTPUT_SIZE; j++)
            grad += net->W2[j * HIDDEN_SIZE + i] * s_output[j];
        d_hidden[i] = (hidden[i] > 0.0) ? grad : 0.0;
    }
}
```

Listing 10: V3: Shared memory for Hidden Gradient

Version	**Training**		**Evaluation**		**Total**		**Test Accuracy (%)**
	Time (s)	**Speedup**	**Time (s)**	**Speedup**	**Time (s)**	**Speedup**	
V1 (CPU)	22.23	–	0.81	–	23.04	–	96.85
V2 (GPU)	19.98	1.11x	0.93	0.87x	20.91	1.10x	96.83
V3.1 (SharedMem)	19.03	1.17x	0.85	0.95x	19.88	1.16x	96.80

Table 3: Performance impact of shared memory optimization (V3.1) compared to all versions.

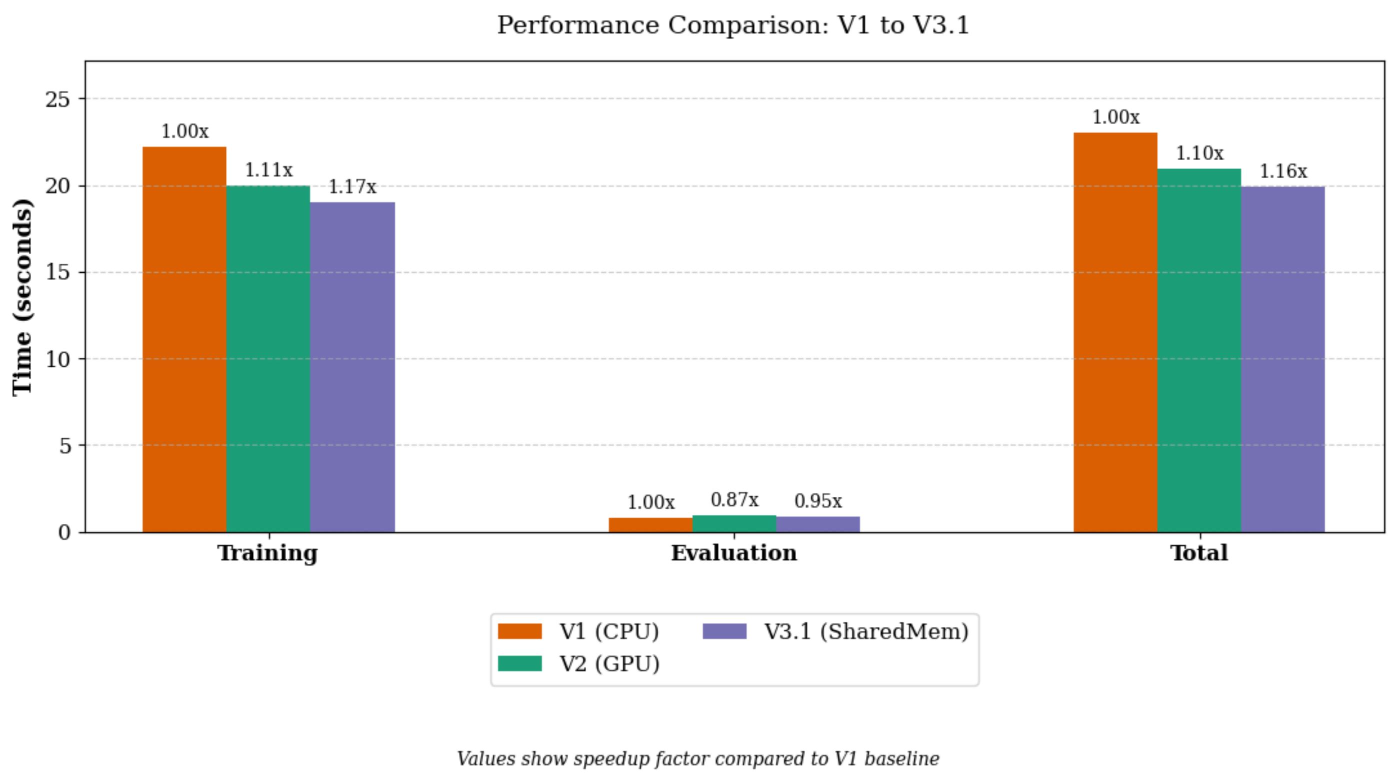

Figure 2: Performance Impact of Shared Memory Optimization

6..2 Streams Optimization

CUDA streams enable concurrent execution of kernels and data transfers, improving GPU utilization:

```
cudaMemcpyAsync(d_input, images[0], INPUT_SIZE * sizeof(double), H2D, streams[0]);
cudaMemcpyAsync(d_label, labels[0], NUM_CLASSES * sizeof(double), H2D, streams[0]);
for (int epoch = 0; epoch < EPOCHS; epoch++) {
    clock_t epoch_start = clock();
    *loss = 0;
    forward_cuda(net, d_input, d_output, d_hidden, streams[0]);
    int correct = 0;
    for (int i = 0; i < numImages; i++) {
        int current_stream = i % 2;
        int next_stream = (i + 1) % 2;
        backward_cuda(net, d_input, d_hidden, d_output, d_label, streams[
   current_stream]);
        cudaMemcpyAsync(pinned_output, d_output, OUTPUT_SIZE * sizeof(double), D2H,
    streams[current_stream]);
        if (i + 1 < numImages) {
            cudaMemcpyAsync(d_input, images[i+1], INPUT_SIZE * sizeof(double), H2D,
    streams[next_stream]);
            cudaMemcpyAsync(d_label, labels[i+1], NUM_CLASSES * sizeof(double), H2D
   , streams[next_stream]);
            forward_cuda(net, d_input, d_output, d_hidden, streams[next_stream]);
        }
        cudaStreamSynchronize(streams[current_stream]); // Only needed for metrics
    }
}
```

Listing 11: V3: CUDA Streams Implementation

Version	Training		Evaluation		Total		Test Accuracy (%)
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	
V1 (CPU)	22.23	–	0.81	–	23.04	–	96.85
V2 (GPU)	19.98	1.11x	0.93	0.87x	20.91	1.10x	96.83
V3.1 (SharedMem)	19.03	1.17x	0.85	0.95x	19.88	1.16x	96.80
V3.2 (Streams)	16.92	1.31x	0.85	0.95x	17.77	1.30x	96.80

Table 4: Performance impact of CUDA streams optimization (V3.2) compared to all versions.

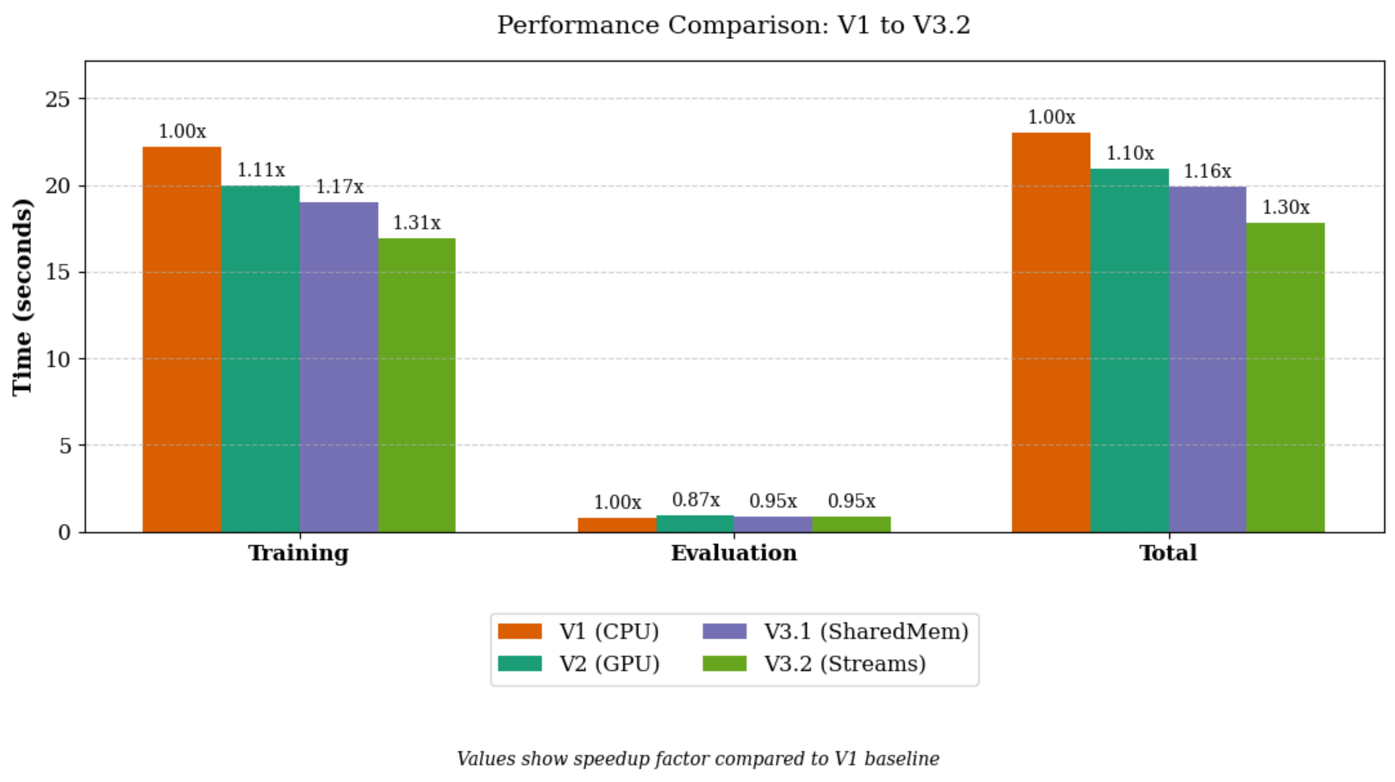

Figure 3: Performance Impact of CUDA Streams Optimization

6..3 Batch Processing

Batch processing increases GPU utilization by processing multiple images simultaneously, reducing kernel launch overhead and leveraging CUDA streams for asynchronous data transfers.

```
int first_batch_size = min(BATCH_SIZE, numImages);
for (int b = 0; b < first_batch_size; b++) {
     cudaMemcpyAsync(d_input[0] + b * INPUT_SIZE, images[b], INPUT_SIZE * sizeof(
    double), H2D, streams[0]);
     cudaMemcpyAsync(d_label[0] + b * NUM_CLASSES, labels[b], NUM_CLASSES * sizeof(
    double), H2D, streams[0]);
}
for (int epoch = 0; epoch < EPOCHS; epoch++) {

     forward_cuda(net, d_input[0], d_output[0], d_hidden[0], streams[0]);

     for (int i = 0; i < numImages; i += BATCH_SIZE) {
          int current_stream = (i / BATCH_SIZE) % 2;
          int next_stream = ((i / BATCH_SIZE) + 1) % 2;
          int batch_size = min(BATCH_SIZE, numImages - i);

          backward_cuda(net, d_input[current_stream], d_hidden[current_stream],
    d_output[current_stream], d_label[current_stream], streams[current_stream]);

          cudaMemcpyAsync(pinned_output, d_output[current_stream], batch_size *
    OUTPUT_SIZE * sizeof(double), D2H, streams[current_stream]);

          if (i + BATCH_SIZE < numImages) {
               int next_batch_size = min(BATCH_SIZE, numImages - (i + BATCH_SIZE));
               for (int b = 0; b < next_batch_size; b++) {
                    cudaMemcpyAsync(d_input[next_stream] + b * INPUT_SIZE, images[i +
    BATCH_SIZE + b], INPUT_SIZE * sizeof(double), H2D, streams[next_stream]);
                    cudaMemcpyAsync(d_label[next_stream] + b * NUM_CLASSES, labels[i +
    BATCH_SIZE + b], NUM_CLASSES * sizeof(double), H2D, streams[next_stream]);
               }
               forward_cuda(net, d_input[next_stream], d_output[next_stream], d_hidden
    [next_stream], streams[next_stream]);
          }
          cudaStreamSynchronize(streams[current_stream]);
     }
}
```

Listing 12: V3: Batch Processing Implementation

This code processes batches of size `BATCH_SIZE` (e.g., 64), using two streams to overlap **forward pass**, **backward pass**, and data transfers (H2D, D2H). Compared to V3.2 (Streams), it handles multiple images per kernel call, significantly reducing overhead.

Batch Size	**GPU Training Accuracy (%)**			**GPU Test**	**CPU Training Accuracy (%)**	
	Epoch 1	**Epoch 2**	**Epoch 3**	**Accuracy (%)**	**Epoch 1**	**Epoch 2**
1	91.80	96.88	97.64	97.05	90.00	95.50
2	89.55	96.02	97.22	97.01	90.00	95.50
4	85.89	94.16	95.93	95.61	90.00	95.50
12	75.29	90.82	92.09	92.33	90.00	95.50
32	55.58	86.85	89.56	90.27	90.00	95.50
64	33.61	78.10	85.79	87.65	90.00	95.50
CPU (Fixed)	–	–	–	97.10	90.00	95.50

Table 5: Accuracy across batch sizes for GPU training (epochs 1–3), GPU test, and CPU training (epochs 1–2). CPU values are fixed across batch sizes.

Batch Size	**GPU Time (s)**	**CPU Time (s)**	**Speedup**
1	17.774	23.00	1.29x
2	9.933	23.00	2.32x
4	5.853	23.00	3.93x
12	4.658	23.00	4.94x
32	2.219	23.00	10.36x
64	1.994	23.00	11.54x

Table 6: GPU processing times and speedup compared to CPU (23s) across batch sizes.

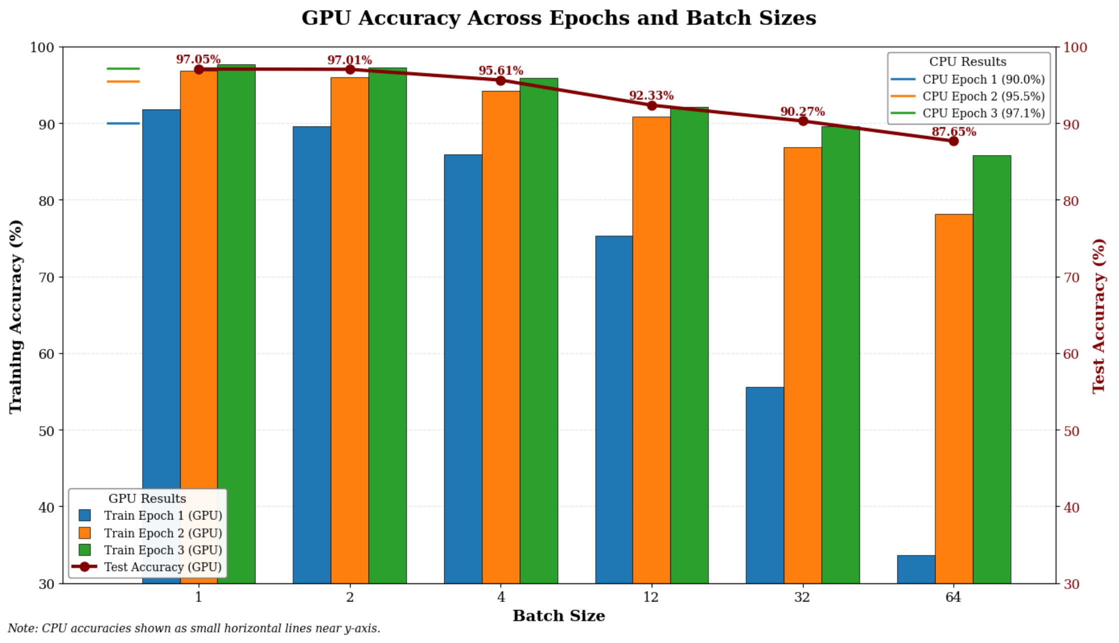

Figure 4: GPU accuracy across epochs and batch sizes, with CPU accuracy markers and GPU test accuracy line.

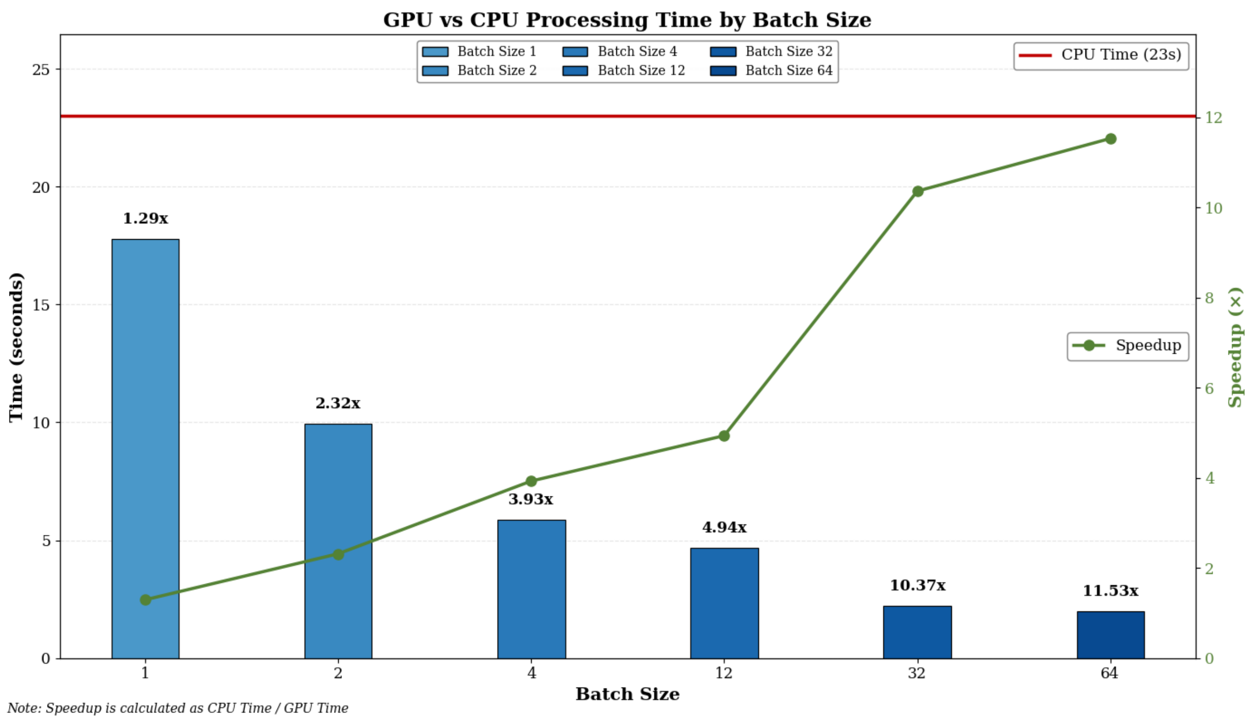

Figure 5: GPU vs CPU processing time by batch size, with speedup values.

6..4 Float Datatype Optimization

Converting from double to float precision reduces memory usage and improves throughput. The entire codebase was templatized to support different data types:

```
template <typename T>
void forward_cuda_eval(NeuralNetwork<T>* net, T* d_input, T* d_output, T* d_hidden,
    int numImages) {
    // Hidden layer
    dim3 blockHidden(INPUT_SIZE); // Threads per block (matches HIDDEN_SIZE)
    dim3 gridHidden(1, numImages); // One block per sample
    compute_hidden_layer<<<gridHidden, blockHidden>>>(net, d_input, d_hidden);

    // Output layer
    dim3 blockOutput(HIDDEN_SIZE); // Threads per block (matches OUTPUT_SIZE)
    dim3 gridOutput(1, numImages);
    compute_output_layer<<<gridOutput, blockOutput>>>(net, d_hidden, d_output);

    // Softmax normalization
    dim3 blockSoftmax(OUTPUT_SIZE); // One thread per sample
    dim3 gridSoftmax(1, numImages);
    normalize_softmax<<<gridSoftmax, blockSoftmax>>>(d_output);
}
```

Listing 13: V3: Templatized Implementation

Version	**Training**		**Evaluation**		**Total**		**Test Accuracy (%)**
	Time (s)	**Speedup**	**Time (s)**	**Speedup**	**Time (s)**	**Speedup**	
V1 (CPU)	22.23	–	0.81	–	23.04	–	96.85
V2 (GPU)	19.98	1.11x	0.93	0.87x	20.91	1.10x	96.83
V3.1 (SharedMem)	19.03	1.17x	0.85	0.95x	19.88	1.16x	96.80
V3.2 (Streams)	16.92	1.31x	0.85	0.95x	17.77	1.30x	96.80
V3.4 (Float)	12.63	1.76x	0.61	1.34x	13.24	1.74x	96.80

Table 7: Performance impact of float datatype optimization (V3.4) compared to all versions.

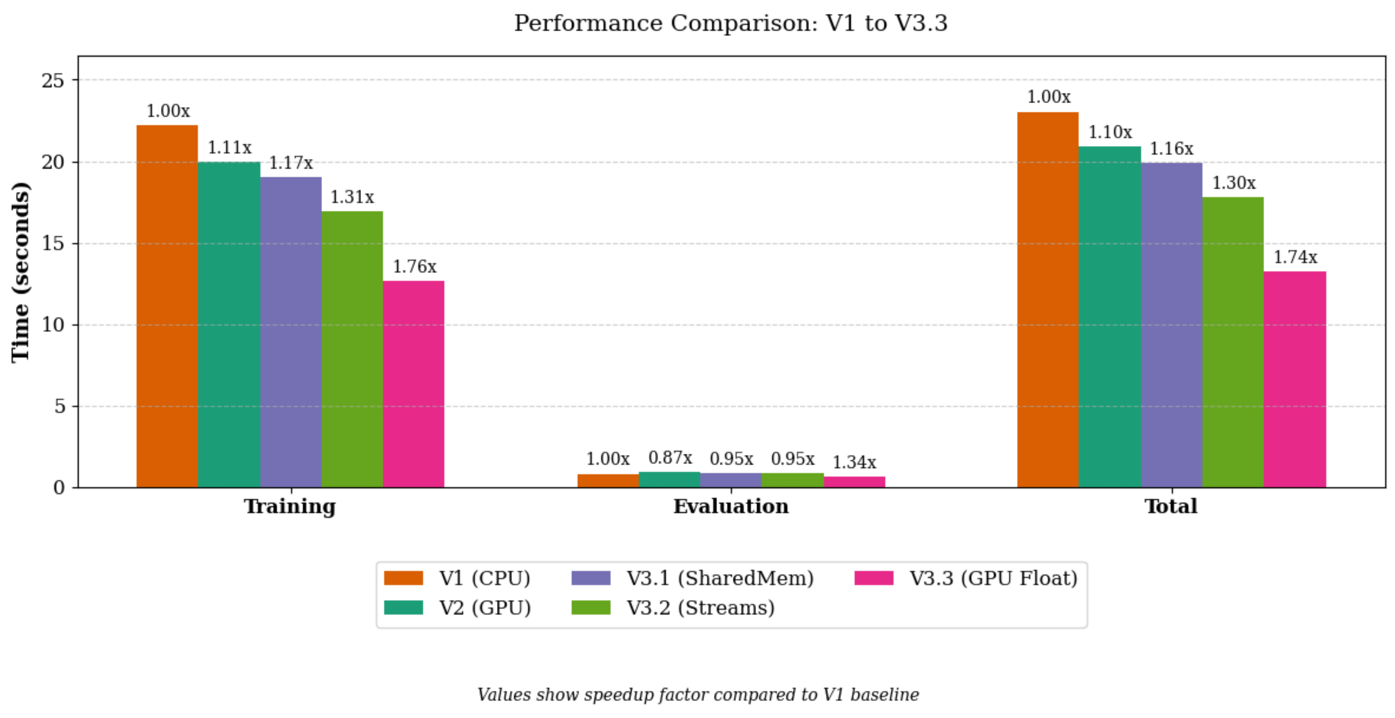

Figure 6: Performance Impact of Float Datatype Optimization

6..5 Evaluation Kernel Optimization

Batch processing for evaluation significantly reduces memory transfers and kernel launches:

```
T* output = (T*)malloc(sizeof(T) * OUTPUT_SIZE * numImages);
T *d_input, *d_hidden, *d_output;
cudaMalloc((void**)&d_input, sizeof(T) * INPUT_SIZE * numImages);
cudaMalloc((void**)&d_hidden, sizeof(T) * HIDDEN_SIZE * numImages);
cudaMalloc((void**)&d_output, sizeof(T) * OUTPUT_SIZE * numImages);

// Copy all images in one go
for (int i = 0; i < numImages; i++) {
    cudaMemcpyAsync(d_input + i * INPUT_SIZE, images[i], sizeof(T) * INPUT_SIZE,
   H2D);
}
forward_cuda_eval(net, d_input, d_output, d_hidden, numImages);

// Copy full output back to host
cudaMemcpy(output, d_output, sizeof(T) * OUTPUT_SIZE * numImages, D2H);
```

Listing 14: V3: Batch Evaluation

Version	Training		Evaluation		Total		Test Accuracy (%)
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	
V1 (CPU)	22.23	–	0.81	–	23.04	–	96.85
V2 (GPU)	19.98	1.11x	0.93	0.87x	20.91	1.10x	96.83
V3.1 (SharedMem)	19.03	1.17x	0.85	0.95x	19.88	1.16x	96.80
V3.2 (Streams)	16.92	1.31x	0.85	0.95x	17.77	1.30x	96.80
V3.4 (Float)	12.63	1.76x	0.61	1.34x	13.24	1.74x	96.80
V3.5 (Eval Kernel)	12.66	1.76x	0.02	35.26x	12.66	1.82x	96.80

Table 8: Performance impact of batch evaluation optimization (V3.5) compared to all versions.

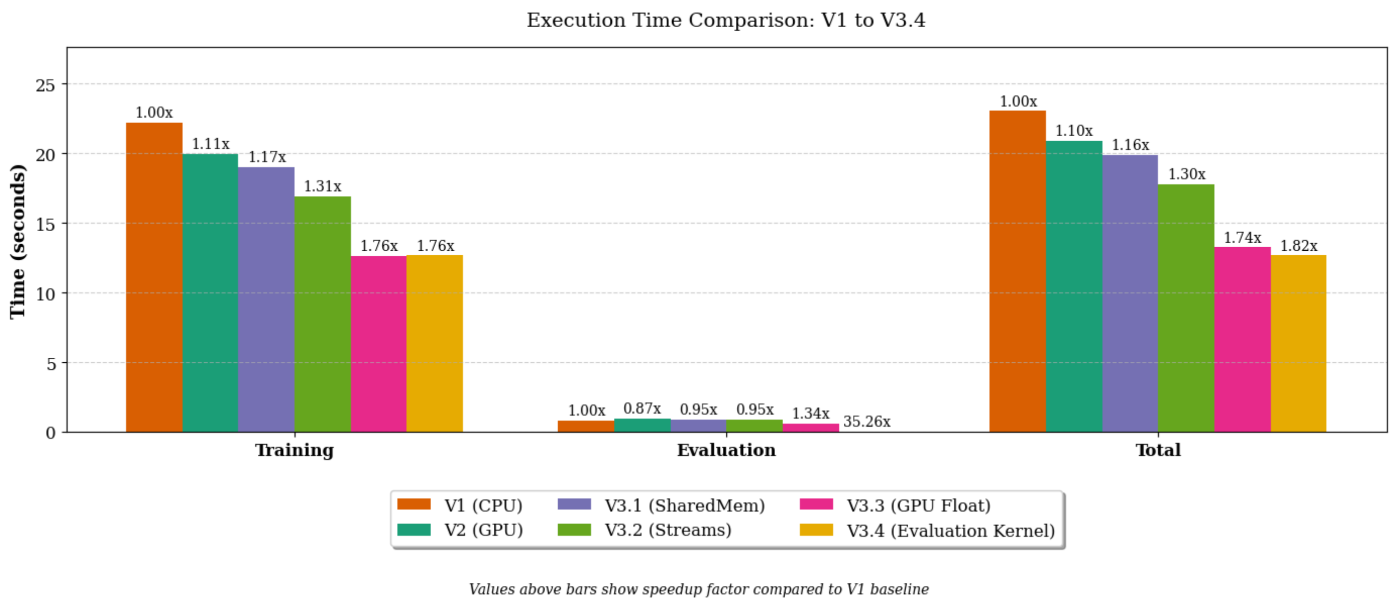

Figure 7: Performance Impact of Batch Evaluation Optimization

7. Version 4: Tensor Core Implementation

Version 4 (V4) builds on Version 3 by integrating the cuBLAS library for FP32 matrix multiplications, leveraging GPU parallelism to accelerate the **forward** and **backward** passes of the 784-128-10 neural network. The compute-intensive 'compute_hidden' kernel is replaced with 'cublasSgemm', performing efficient batched matrix operations (e.g., 'W1': 128784 input: 784BATCH_SIZE). Combined with batch processing (V3.3) and CUDA streams (V3.2), V4 achieves significant performance gains without requiring FP16-specific Tensor Core acceleration.

Implementation. The following code snippet illustrates the **forward pass** using 'cublasSgemm' for batched matrix multiplication, followed by a custom kernel for bias addition and ReLU activation:

```
const float alpha = 1.0f;
const float beta = 0.0f;

cublasSgemm(handle,
    CUBLAS_OP_T, CUBLAS_OP_T,
    HIDDEN_SIZE, BATCH_SIZE, INPUT_SIZE,
    &alpha,
    h_net.W1, INPUT_SIZE, // W1^T [INPUT_SIZE x HIDDEN_SIZE]
    d_input, BATCH_SIZE,  // input^T [INPUT_SIZE x BATCH_SIZE]
    &beta,
    d_hidden, HIDDEN_SIZE); // d_hidden [HIDDEN_SIZE x BATCH_SIZE]

dim3 block(HIDDEN_SIZE);
dim3 grid(1, BATCH_SIZE);
apply_bias_relu<<<grid, block, 0, stream>>>(d_hidden, net);
```

Listing 15: V4: cuBLAS-based Forward Pass

Performance Impact. By replacing custom kernels with cuBLAS, V4 reduces training time to 6.03 seconds (3.69x speedup over V1) and evaluation time to 0.02 seconds (36.86x speedup), yielding a total runtime of 6.05 seconds (3.81x speedup), as shown in Table 9. The use of FP32 maintains accuracy (96.82

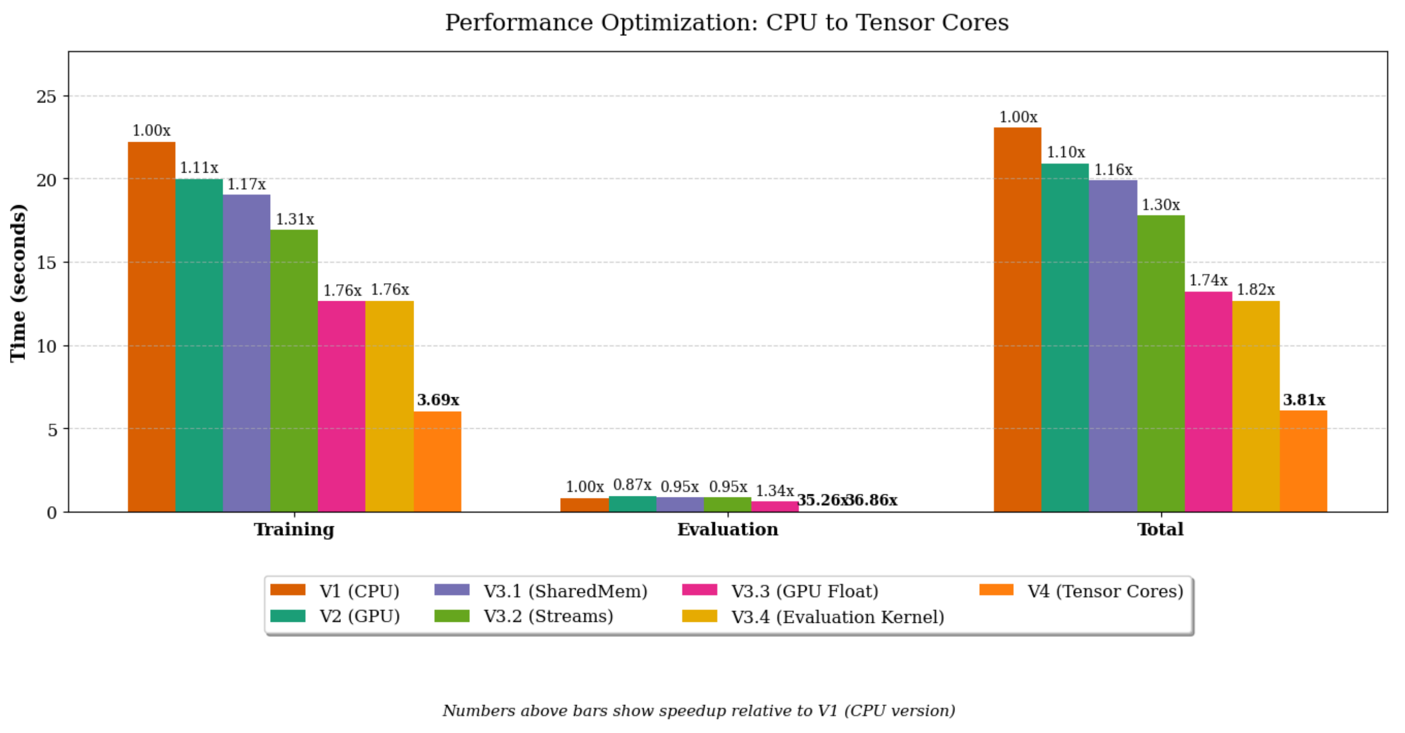

Figure 8: Training and evaluation times across versions, highlighting V4's Tensor Core performance.

Version	**Training**		**Evaluation**		**Total**		**Test Accuracy (%)**
	Time (s)	**Speedup**	**Time (s)**	**Speedup**	**Time (s)**	**Speedup**	
V1 (CPU)	22.23	–	0.81	–	23.04	–	96.85
V2 (GPU)	19.98	1.11x	0.93	0.87x	20.91	1.10x	96.83
V3.1 (SharedMem)	19.03	1.17x	0.85	0.95x	19.88	1.16x	96.80
V3.2 (Streams)	16.92	1.31x	0.85	0.95x	17.77	1.30x	96.80
V3.4 (Float)	12.63	1.76x	0.61	1.34x	13.24	1.74x	96.80
V3.5 (Eval Kernel)	12.66	1.76x	0.02	35.26x	12.66	1.82x	96.80
V4 (Tensor Core)	6.03	3.69x	0.02	36.86x	6.05	3.81x	96.82

Table 9: Performance impact of cuBLAS FP32 implementation (V4) compared to all versions.

8. Conclusion

This project optimized a 784-128-10 neural network for MNIST classification, achieving a 3.81x speedup from V1 (CPU, 23.04s) to V4 (Tensor Core, 6.05s) while maintaining high accuracy (96.80–96.85

Contributions:

- **Shared Memory (V3.1)**: Optimized kernel latency by caching weights in shared memory.
- **Batch Processing (V3.3)**: Increased throughput with batched operations and stream overlap.
- **Tensor Core Integration (V4)**: Leveraged cuBLAS for efficient FP32 matrix multiplications.