# MNIST DIGIT CLASSIFICATION THROUGH GPU


**Prepared By:**

- Muhammad Abdul Wahab Kiyani – 22i-1178 – CS-A
- Syed Ahmed Ali – 22i-1237 – CS-A

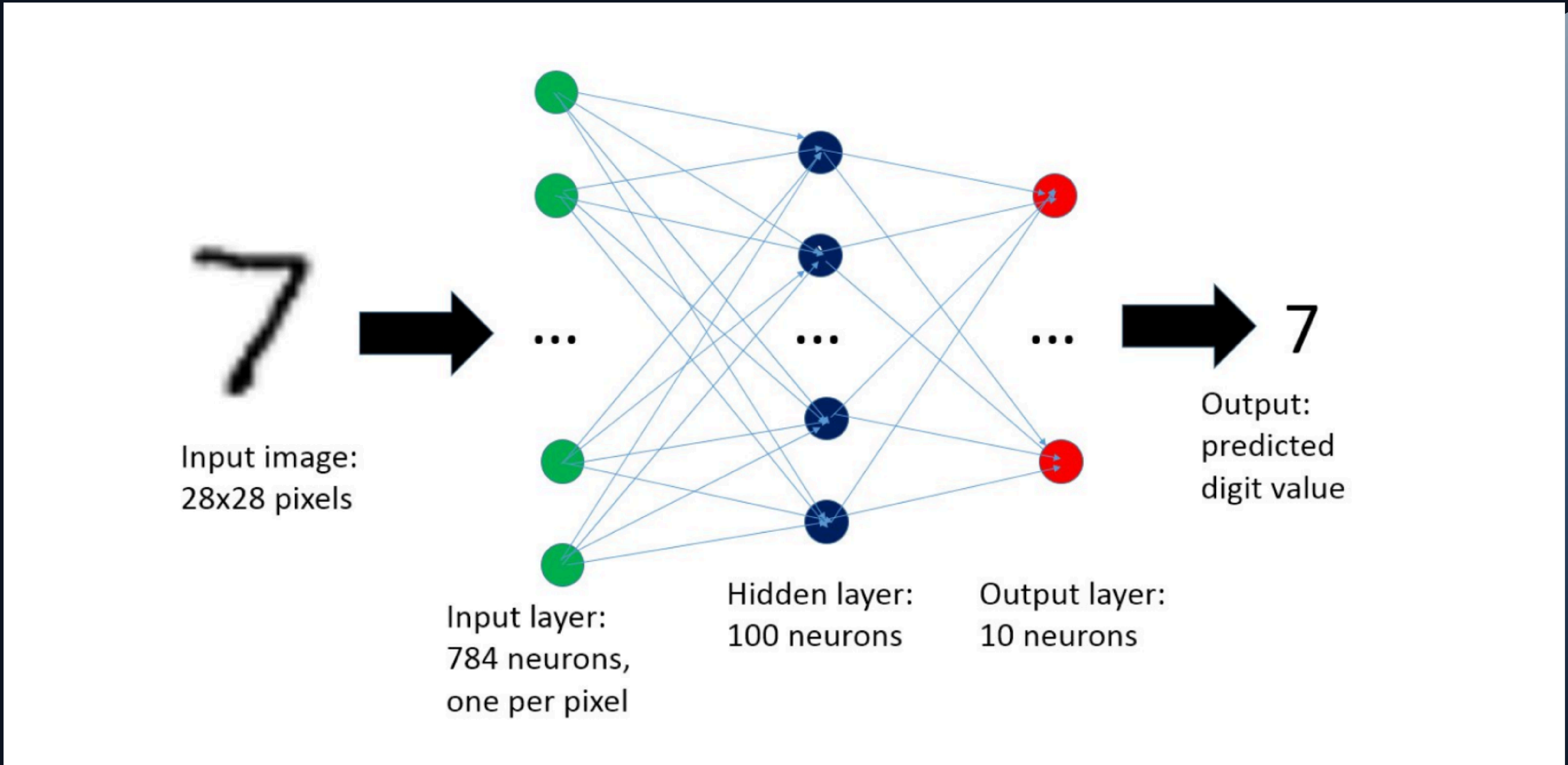
# WHAT IS MNIST?

It’s a dataset of handwritten digits (0–9) used to train and test machine learning models.

**Input Data:**

- 📦 60,000 training images
- 📦 10,000 test images
- 🖍️ Each image: 28 × 28 grayscale pixels
- 🎯 Normalized to [0, 1]

# NEURAL NETWORK



# TESTING CONDITIONS

**Model Parameters**

- 🧠 Input Size: 784 (28×28 pixels)
- 🧠 Hidden Layer: 128 neurons
- 🧠 Output Size: 10
- 🔁 Epochs: 3
- 📉 Learning Rate: 0.01

**Hardware Setup GPU**

- 💻 Development : RTX 3050 Ti
- ⚡ Benchmark : RTX 3080

**🧪 Tools Used**

- Analysis: NVIDIA Nsight Systems
- Compilers: CUDA 12.8, GCC 13

**🎯 Focus**

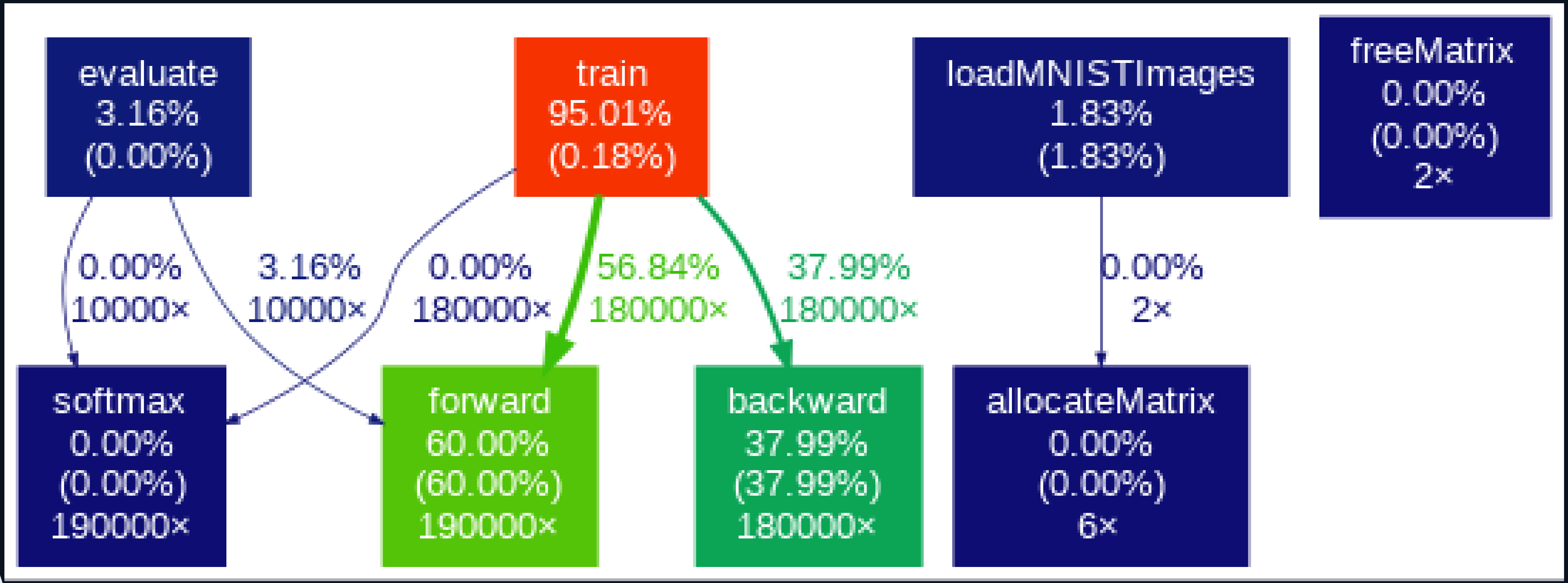
- Accuracy was the top priority
- All versions were trained for 3 epochs to ensure fair comparison

# V1 — CPU BASELINE

- **Approach:** Classic sequential C++ loops
- **Performance:** 🐢 22.50s
- **Purpose:** Baseline to compare other versions

Simplest of all, but no parallelism

# V1 — GPROF ANALYSIS



# V2 — NAIVE CUDA

- **Approach:** Offload matrix ops to GPU
- **Issues:**
  - Uncoalesced memory access
  - Frequent host ↔ device transfers
  - No use of streams, shared memory, or smart load balancing

**Performance:** 🐢 41.46s

# Code Snippet:

```
__global__ void update_weights_W2(double* d_W2, double* d_b2, double* d_hidden,
                                double* d_d_output, double lr) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < OUTPUT_SIZE) {
        for (int j = 0; j < HIDDEN_SIZE; j++) {
            int idx = i * HIDDEN_SIZE + j;
            d_W2[idx] -= lr * d_d_output[i] * d_hidden[j];
        }
        d_b2[i] -= lr * d_d_output[i];
    }
}
```

# V3 — OPTIMIZED CUDA

- **Approach:**
  - Tuned thread blocks
  - Used shared memory
  - Pinned host memory
  - CUDA streams

**Huge gains but complex.**

Performance: ⚡3.32s

## Code Snippet:

```
__global__ void update_weights_W2_shared(my_type* d_W2, my_type* d_b2, my_type* d_hidden,
                                         my_type* d_d_output, my_type lr) {
    __shared__ my_type shared_hidden[HIDDEN_SIZE];
    __shared__ my_type shared_d_output[OUTPUT_SIZE];
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    for (int idx = threadIdx.x; idx < HIDDEN_SIZE; idx += blockDim.x) {
        shared_hidden[idx] = d_hidden[idx];
    }
    for (int idx = threadIdx.x; idx < OUTPUT_SIZE; idx += blockDim.x) {
        shared_d_output[idx] = d_d_output[idx];
    }
```

# V4 — TENSOR CORES

- **Used:** FP16 + Tensor Core via WMMA

- **Issue:** Accuracy dropped from 96.85% to 70%
  (but more epochs could've helped)

**Observation:**

- FP16 precision struggles with subtle weight updates during training.
- Rounding errors heavily impact backpropagation.

# Code Snippet:

```
__global__ void tensor_matmul(const half_type* A, const half_type* B, my_type* C,
                              int M, int N, int K) {
    const int WMMA_M = 16;
    const int WMMA_N = 16;
    const int WMMA_K = 16;

    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half_type, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half_type, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, my_type> acc_frag;
```

# V5 — OPENACC

- **Pros:**
  - Less manual effort
  - Best performance

- Cons:
  - Programmer has less control
  - Tricky debugging

Performance: ⚡2.24s

# Code Snippet:

```
#pragma acc parallel loop gang vector
for (int i = 0; i < OUTPUT_SIZE; i++) {
    #pragma acc loop
    for (int j = 0; j < HIDDEN_SIZE; j++) {
        net->W2[i * HIDDEN_SIZE + j] -= LEARNING_RATE * d_output[i] * hidden[j];
    }
    net->b2[i] -= LEARNING_RATE * d_output[i];
}
```

# SPEEDUPS

# CONCLUSION

- GPU Parallelism drastically improves performance.
- Raw CUDA needs optimization for memory and communication efficiency.
- Memory management (shared memory, streams) is crucial for speedup.
- Tensor Cores offer great performance for matrix operations
- OpenACC simplifies parallelism with less manual effort but offers less control.