High Performance Computing with GPU

Semester Project

Group Information – Section B

- Group Members
  - Hamza Saleem 22i-1185
  - Abdullah Mehmood 22i-1167
  - Aneeq Ahmed Malik 22i-0978

- Problem under Discussion
  - MNIST Dataset: 60,000 training, 10,000 test images (28x28 pixels)
  - Neural Network: 784 (input) → 128 (hidden) → 10 (output).
  - Challenge: CPU sequential processing limits performance
  - Solution: Leverage GPU parallelism for acceleration
  - Approach: Four versions (V1: CPU, V2: Naive CUDA, V3: Optimized CUDA, V4: cuBLAS)

- Github: https://github.com/Aneeq-Ahmed-Malik/Neural-Network-Acceleration-on-GPUs

Initial Implementation – CPU Version

- Description: C-based, processes images sequentially
- Components: Matrix multiplications: W1 (128x784), W2 (10x128)
- Stochastic Gradient Descent for forward/backward passes

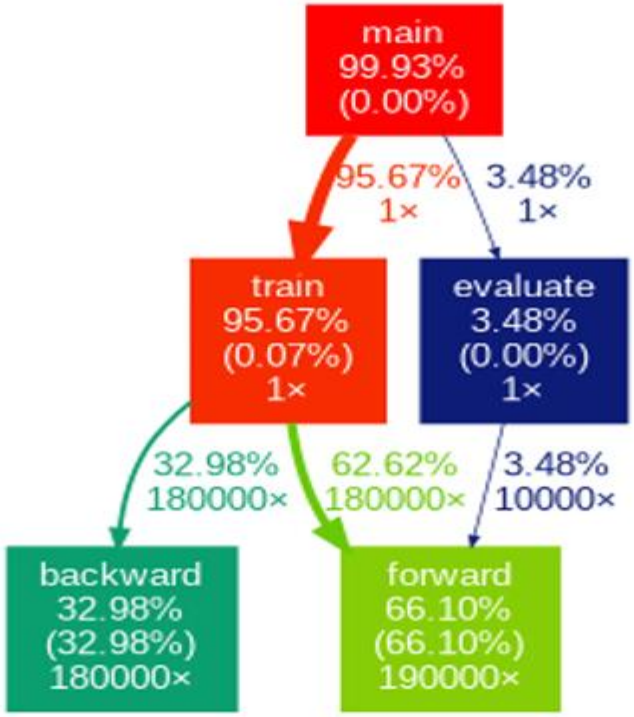


Figure 1: Call Graph of the Neural Network Application

Towards V2 – Forward

```
// Forward pass

void forward(// params) {
    for (int i = 0; i < HIDDEN_SIZE; i++) {
        hidden[i] = net->b1[i];
        for (int j = 0; j < INPUT_SIZE; j++)
            hidden[i] += net->W1[i][j] * input[j];
    }

    relu(hidden, HIDDEN_SIZE);

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

    softmax(output, OUTPUT_SIZE);

}
```

compute_hidden

compute_output

softmax

Towards V2 – Backward

```
void backward(//params) {
```

```
    for (int i = 0; i < OUTPUT_SIZE; i++)
        d_output[i] = output[i] - target[i];
```

d_hidden

```
    for (int i = 0; i < HIDDEN_SIZE; i++) {
        d_hidden[i] = 0;
        for (int j = 0; j < OUTPUT_SIZE; j++)
            d_hidden[i] += net->W2[j][i] * d_output[j];
        d_hidden[i] *= (hidden[i] > 0);
    }
```

d_output

```
    for (int i = 0; i < OUTPUT_SIZE; i++)
        for (int j = 0; j < HIDDEN_SIZE; j++)
            net->W2[i][j] -= LEARNING_RATE * d_output[i] * hidden[j];
    for (int i = 0; i < OUTPUT_SIZE; i++)
        net->b2[i] -= LEARNING_RATE * d_output[i];
```

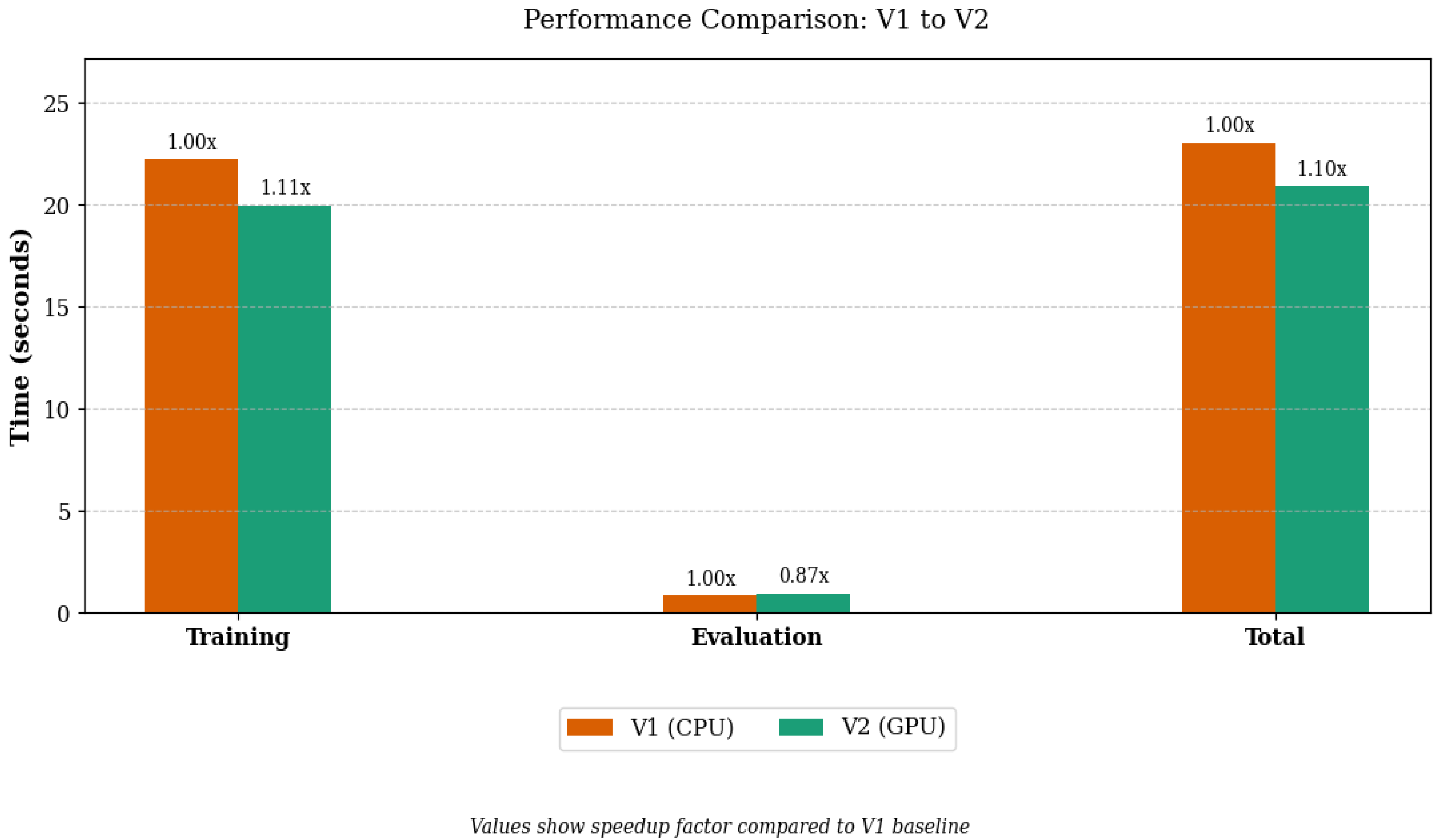
out weights

```
    for (int i = 0; i < HIDDEN_SIZE; i++)
        for (int j = 0; j < INPUT_SIZE; j++)
            net->W1[i][j] -= LEARNING_RATE * d_hidden[i] * input[j];
    for (int i = 0; i < HIDDEN_SIZE; i++)
        net->b1[i] -= LEARNING_RATE * d_hidden[i];
```

hid weights

```
}
```

Version – 2: Performance



Towards V3.1: Shared Memory (1)

Note: 2 more kernels follow the same pattern

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

__global__ void compute_output_layer(// params) {
    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);
    }
}
```

Reuse

Towards V3.1: Shared Memory (2)

```
__global__ void compute_hidden_layer(//params) {
    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;
    }
}
```

Local Id

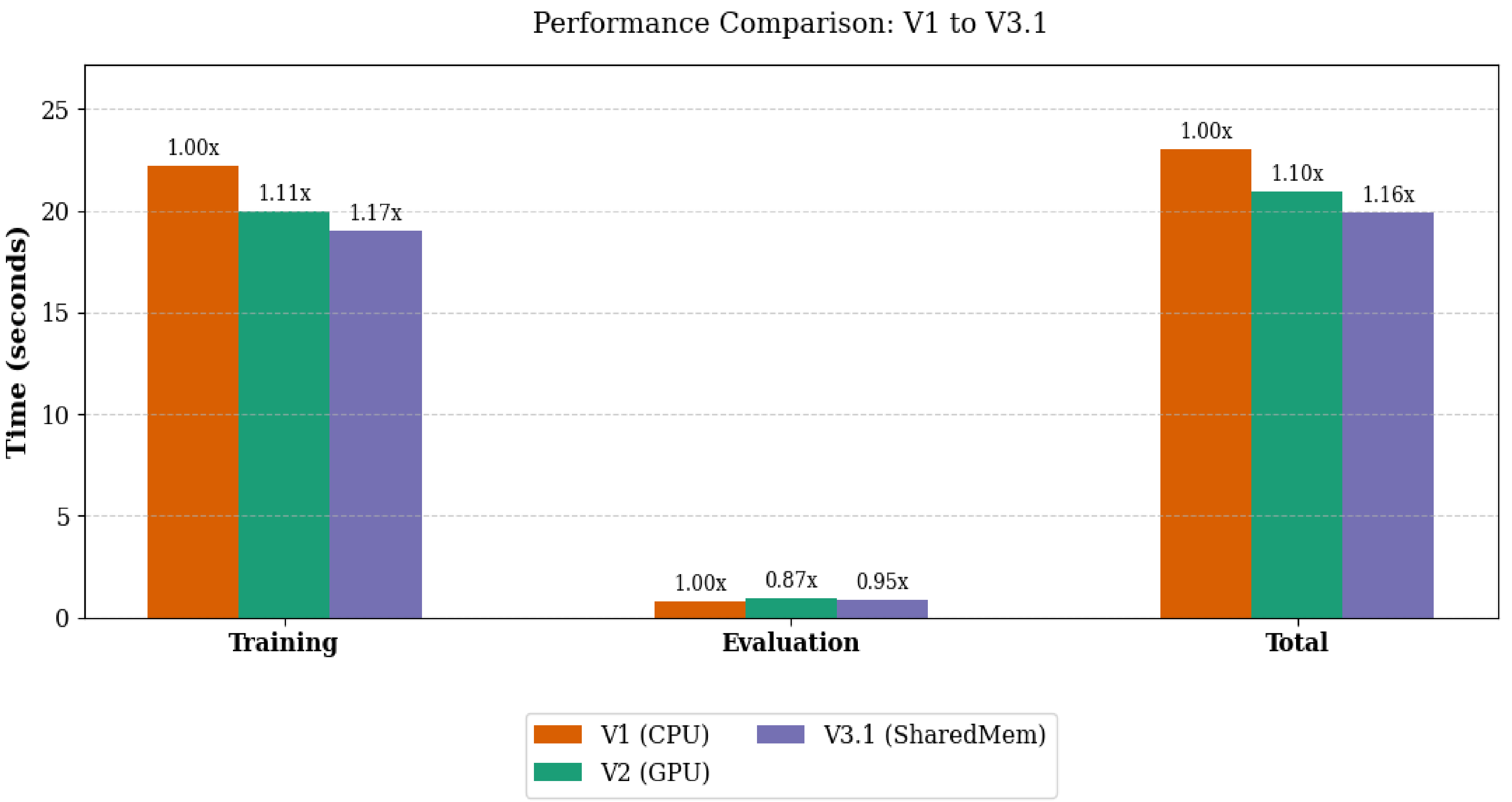
SharedMem

Local sync

Use Shared

Note: Other kernels
follow the same pattern

Version – 3.1: Performance



Values show speedup factor compared to V1 baseline

Towards V3.2: Cuda Streams (1)

```
void train(// params) {
    double* hidden, *output
    double* d_hidden, *d_output, *d_input, *d_label;
    // cuda Malloc & malloc for all device/host variables

    for (int epoch = 0; epoch < EPOCHS; epoch++) {
        for (int i = 0; i < numImages; i++) {

            cudaMemcpy(d_input, images[i], size_t, H2D);
            forward_cuda(net, d_input, d_output, d_hidden);
            cudaMemcpy(d_label, labels[i], size_t, H2D);
```

dependent

```
            backward_cuda(net, d_input, d_hidden, d_output, d_label);
            cudaMemcpy(output, d_output,size_t, D2H);

            // Compute loss & accuracy
        }
    }
}
```

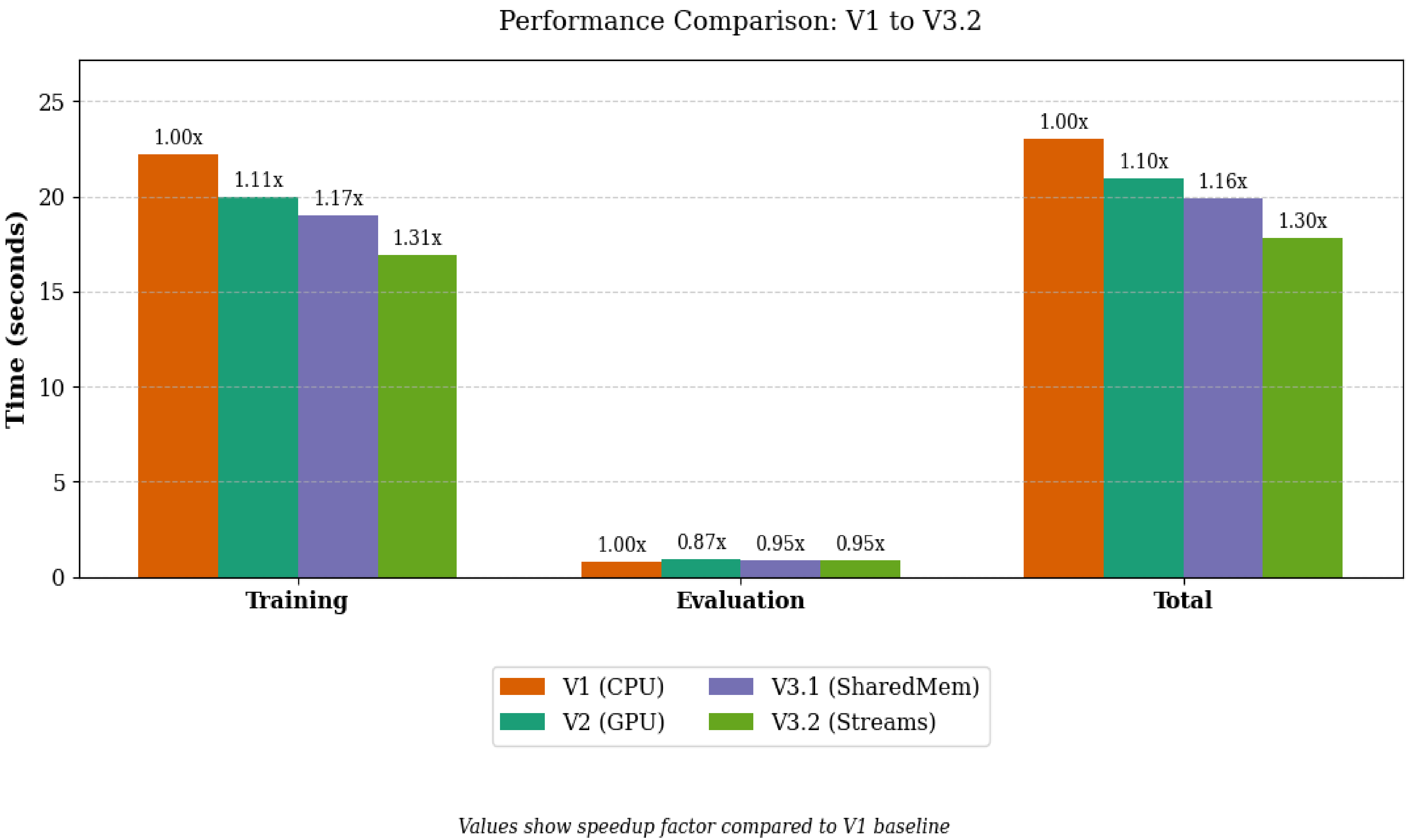
But d_input & d_label of next iter can overlap backward of current iter

Towards V3.2: Cuda Streams (2)

```
void train(// params) {
    cudaMemcpyAsync(d_input, images[0], size, H2D, streams[0]);
    cudaMemcpyAsync(d_label, labels[0], size, H2D, streams[0]);

    for ( /* epoch loop */ ) {

        forward_cuda(/*same*/, streams[0]);
        for ( /* numImages loop */ ) {
            int curr_str = i % 2, next_str = (i + 1) % 2;

            backward_cuda(/*same*/,streams[curr_str]);
            cudaMemcpyAsync(/* output */, streams[curr_str]);

            if (i + 1 < numImages) {
                cudaMemcpyAsync(/* input */, streams[next_str]);
                cudaMemcpyAsync(/* label */, streams[next_str]);
            }
            cudaStreamSynchronize(streams[curr_str]);
            forward_cuda(/*same*/,streams[next_str]);
        }
    }
}
```

overlap

Version – 3.2: Performance



Towards V3.3: Batch Processing (2)

- Problem: Still processing one image at a time → high overhead.
- Idea : Process multiple images simultaneously in a batch (e.g., 64 images).
- Implementation:
  - Allocate memory for entire batch.
  - Launch forward and backward CUDA kernels for the batch.
  - Use CUDA streams to overlap batches.
- Impact:
  - Reduces kernel launch overhead.
  - Increases GPU utilization.
  - Significant speedup compared to single image processing.
- Drawback:
  - Small loss in accuracy at large batch sizes.
  - Less frequent weight updates (compared to per-image SGD).

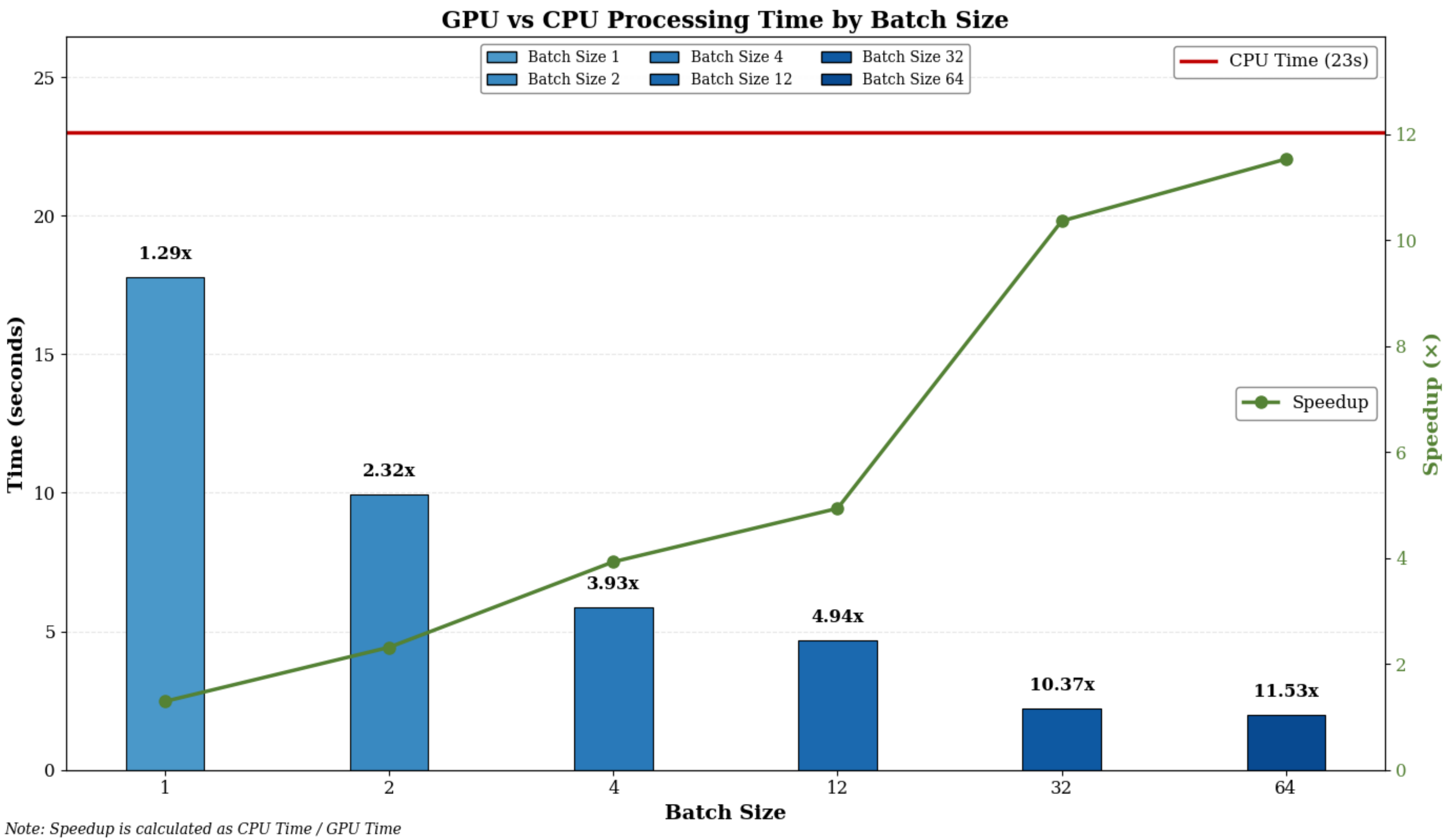
Towards V3.3: Batch Processing (2)

```
void train(//params) {
    for (int b = 0; b < min(BATCH_SIZE, numImages); b++) {
        cudaMemcpyAsync(d_input[0] + b * INPUT_SIZE, /*same*/);
        cudaMemcpyAsync(d_label[0] + b * NUM_CLASSES, /*same*/);
    }
    for (/*epoch loop*/) {
        forward_cuda(/*same*/, streams[0]);
        for (int i = 0; i < numImages; i += BATCH_SIZE) {
           int batch_size = min(BATCH_SIZE, numImages - i);

            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(/* input */, streams[next_str]);
                       cudaMemcpyAsync(/* label */, streams[next_str]);
                }
            }
        }
    }
}
```

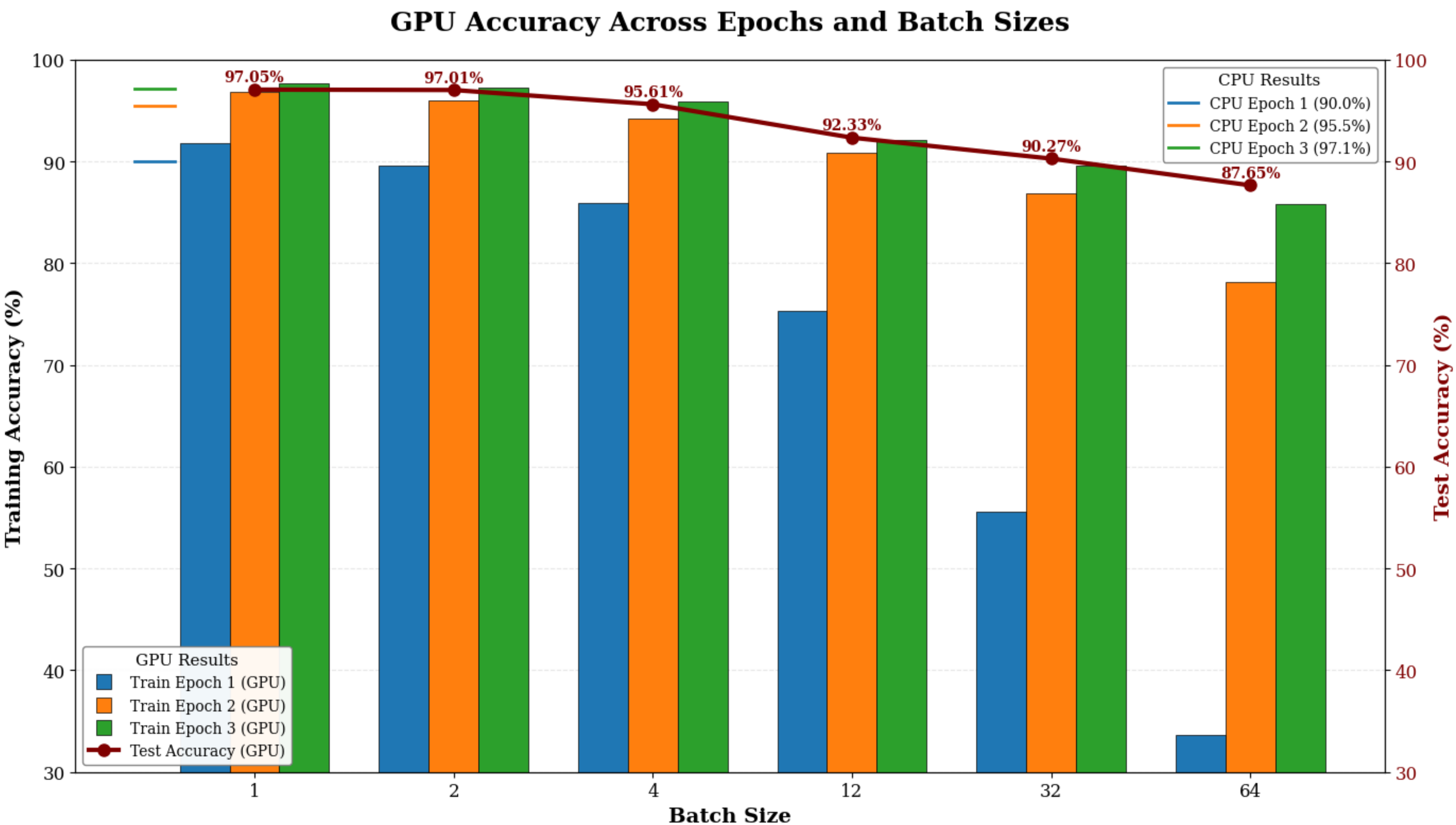
Batch Copy

Batch Copy

Version – 3.3: Performance



Version – 3.3: Drawback

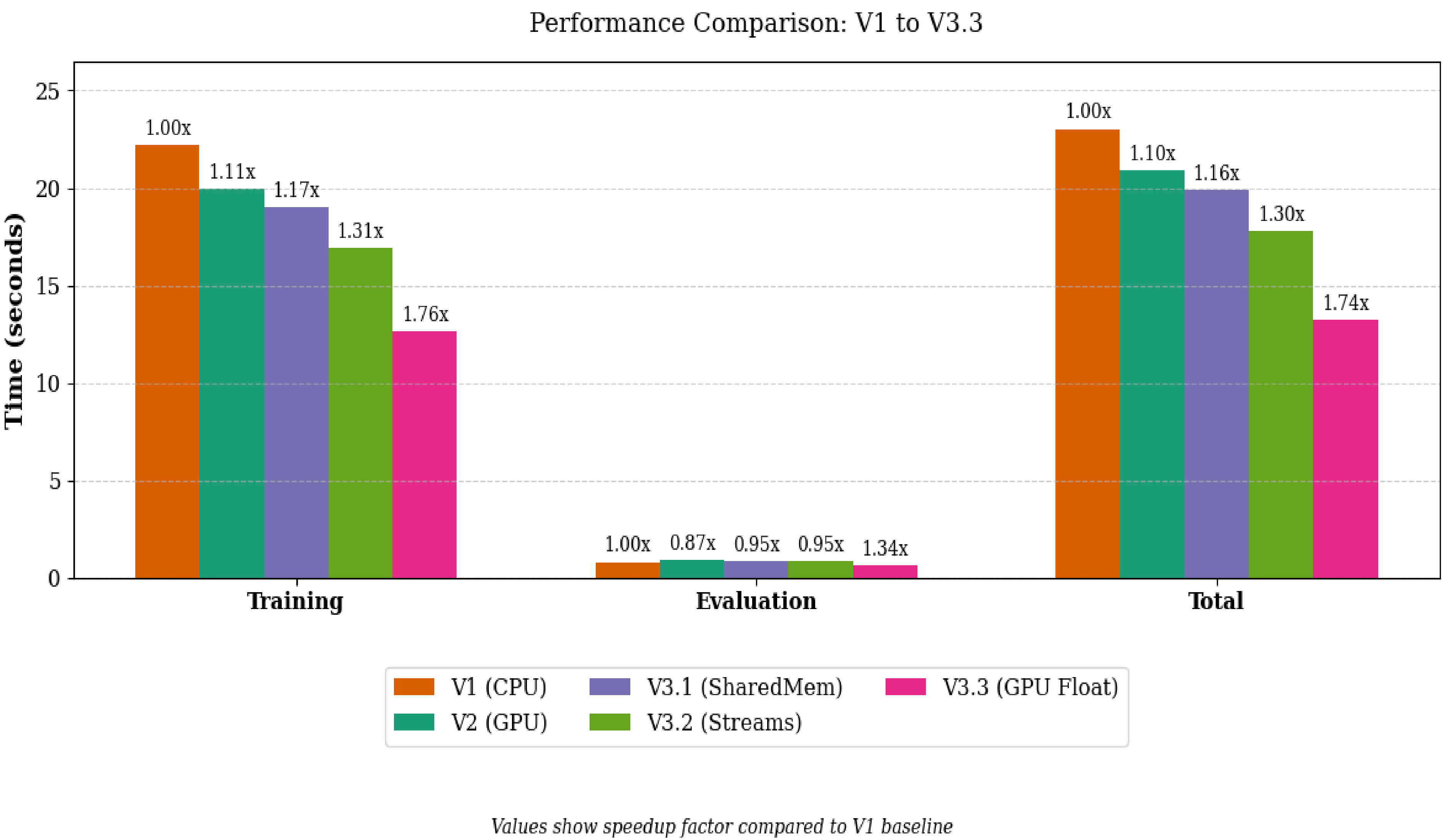


Note: CPU accuracies shown as small horizontal lines near y-axis.

Towards V3.4: FP32 Processing

- Problem: Double precision (double) variables consume more memory and cause slower memory transfers on GPU.
- Idea: Switch to single precision (float, FP32) for faster computation and lower memory usage.
- Implementation:
  - Templatized the code: (template<typename T>), so they work for both float and double.
- Impact:
  - Reduced memory consumption.
  - Faster memory transfers between CPU and GPU.
  - Increased kernel execution speed.
  - Improved training time significantly.
- Drawback:
  - Minor risk of numerical precision loss (but no noticeable accuracy drop for MNIST).

Version – 3.4: Performance



Towards V3.5: Evaluation Kernel (1)

```
void evaluate(// params) {
    int correct = 0;
    double* hidden = (double*)malloc(sizeof(double) * HIDDEN_SIZE);
    double* output = (double*)malloc(sizeof(double) * OUTPUT_SIZE);
    double* d_hidden, *d_output, *d_input;
    // initialize device memory

    for (int i = 0; i < numImages; i++) {

        cudaMemcpy(d_input, images[i], size, H2D);
        forward_cuda(net, d_input, d_output, d_hidden, 0);
        cudaMemcpy(output, d_output, size, D2H);

        int pred = 0, actua
        for (int j = 0; j < OUTPUT_SIZE; j++) {
            if (output[j] > output[pred]) pred = j;
            if (labels[i][j] > labels[i][actual]) actual = j;
        }
        if (pred == actual) correct++;
    }
}
```

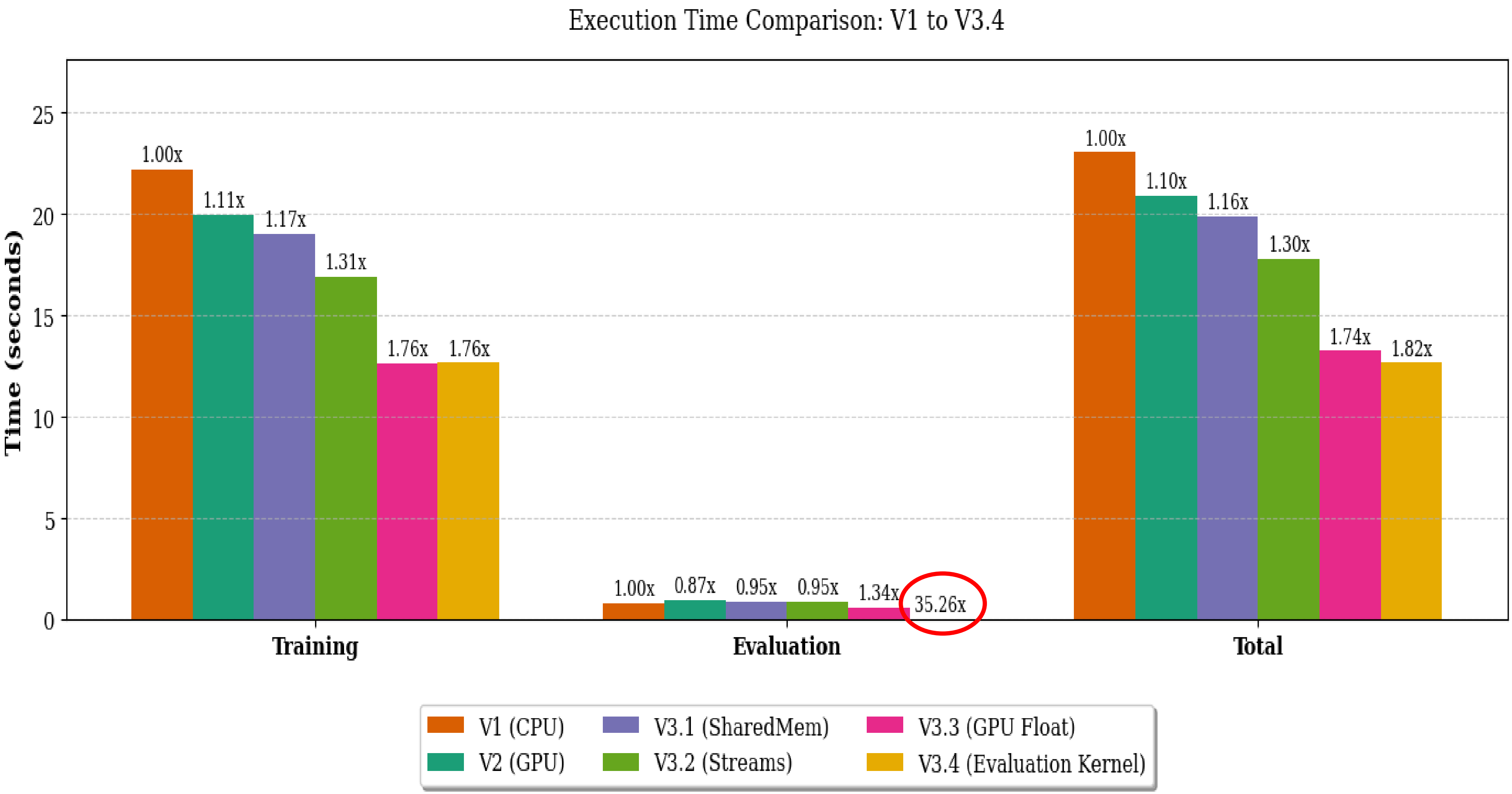
No dependency

Towards V3.5: Evaluation Kernel (2)

```
void evaluate(// params) {
    int correct = 0;
    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);

    for (int i = 0; i < numImages; i++)
        cudaMemcpyAsync(d_input + i * INPUT_SIZE, images[i], size, H2D);

    forward_cuda_eval(net, d_input, d_output, d_hidden, numImages);

    // Copy full output back to host
    cudaMemcpy(output, d_output, size , D2H);

    // Accuracy calculation
    for (int i = 0; i < numImages; i++) {
        // same check pred
    }
}
```

BatchSize

Version – 3.5: Performance



Values above bars show speedup factor compared to V1 baseline

Towards V4: Tensor Cores

- Problem: Even after optimization, custom CUDA kernels were slower than what optimized libraries could achieve.
- Idea: Replace manual matrix multiplications with highly-optimized cuBLAS library functions.

```
__global__ void compute_hidden_layer(//params) {
    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;
    }
}
```

Mat Mul of
(128 x 784) x (784 x 1)

Towards V4: Tensor Cores

- Implementation:
  - Used cublasSgemm() to perform fast batched matrix multiplications
  - It computes C = α×(A×B)+β×C (column major)
  - A and B are input matrices, C is the output matrix
  - alpha and beta are scaling factors (usually 1 and 0).
  - Here, $\boldsymbol{C_{128\,X\,1}} = \boldsymbol{A_{128\,X\,784}}\ \boldsymbol{X}\ \boldsymbol{B_{784\,X\,1}}$

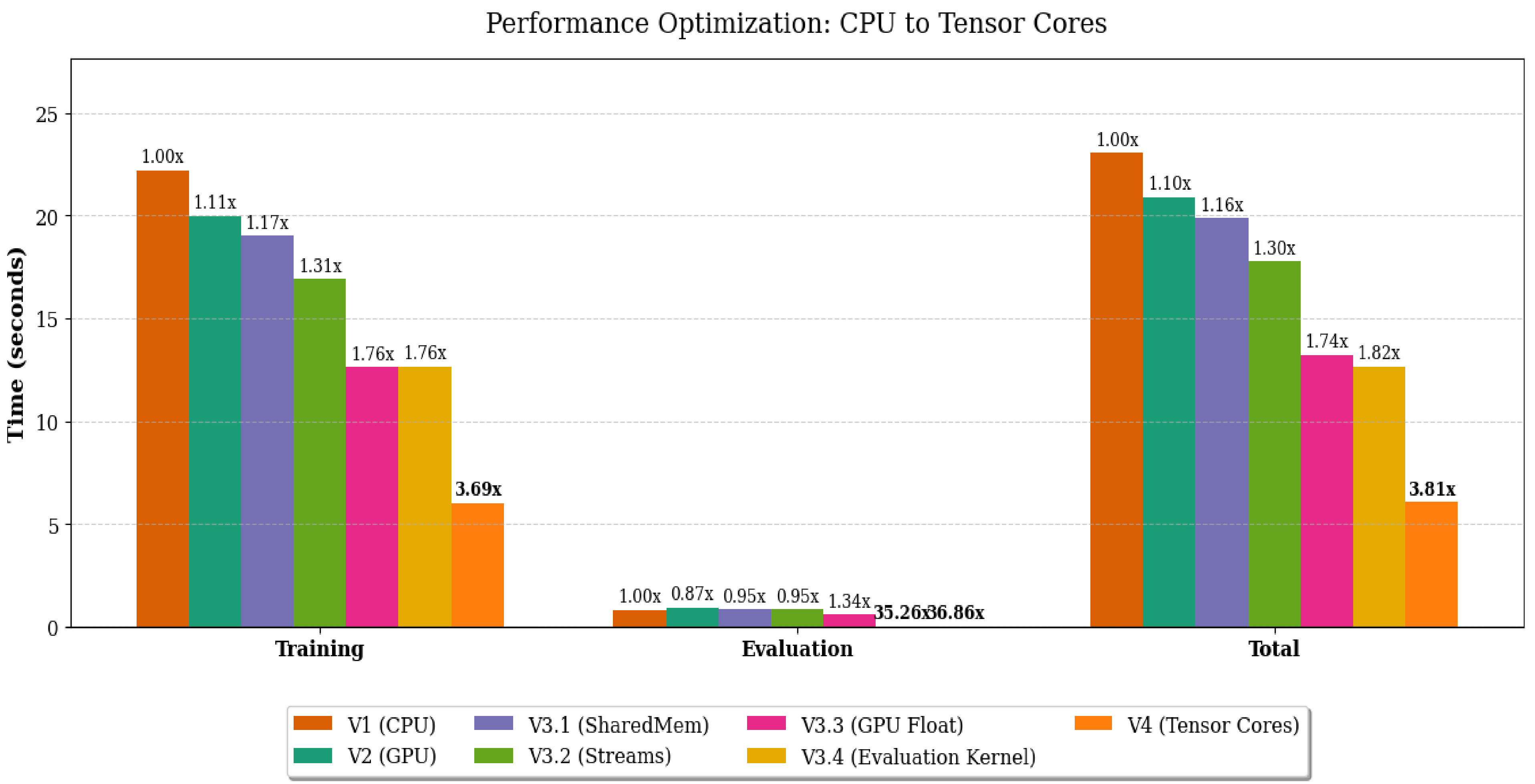
```
const T alpha = 1.0f;
const T beta = 0.0f;
cublasSgemm(handle,
   CUBLAS_OP_T, CUBLAS_OP_T,
   HIDDEN_SIZE, BATCH_SIZE, INPUT_SIZE,
   &alpha,
   h_net.W1, INPUT_SIZE,
   d_input, BATCH_SIZE,
   &beta,
   d_hidden, HIDDEN_SIZE);
```

Transpose input

Input Mat (A,B)

Output Mat (C)

Version – 4: Performance



Numbers above bars show speedup relative to V1 (CPU version)

Any Question So Far?