“CPU vs GPU comparison for Logistic Regression model for network traffic classification”

Siena 2026

1 Project Objective

The objective of this project is to implement and analyze a high-performance Logistic Regression model for network traffic classification, focusing on the architectural performance differences between CPU and GPU implementations. The task is to classify network traffic flows as benign or malicious using large-scale numerical data, while studying the impact of thread-level parallelism, memory hierarchy behavior, branch prediction effects, and accelerator usage. The project emphasizes performance analysis rather than model complexity, using a simple and well-understood machine learning algorithm to expose architectural bottlenecks and scalability limits on modern computing platforms.

2 Dataset Description

This project uses the CICIoT2023 dataset, a modern and large-scale dataset developed by the Canadian Institute for Cybersecurity for research in IoT network security and intrusion detection.

The dataset represents realistic IoT network traffic generated from a testbed simulating multiple IoT devices, including smart TVs, webcams, thermostats, and wearable devices, under both benign operation and a wide range of cyber-attack scenarios.

The dataset contains flow-based network features extracted from packet traces, including traffic statistics, protocol flags, timing information, and packet size distributions.

Each sample is labeled, enabling supervised learning, and includes multiple attack categories such as Denial of Service (DoS), Distributed Denial of Service (DDoS), Mirai botnet attacks, reconnaissance, spoofing, brute-force, and web-based attacks.

For this project, the dataset is organized into predefined training, validation, and test splits with the following sizes:

1) Training set: 5,491,971 samples × 47 features
2) Validation set: 1,176,851 samples × 47 features
3) Test set: 1,176,851 samples × 47 features

To simplify the learning task and focus on performance analysis, all attack types will be grouped into a single malicious class, while benign traffic is assigned to a benign class, resulting in a binary classification problem suitable for Logistic Regression.

3 Data Preparation

The dataset preparation process involved several critical steps to transform the raw CICIoT2023 data into a format suitable for high-performance Logistic Regression training.

Label Binarization

All 34 attack classes were grouped into a single malicious class (label_bin = 1), while BenignTraffic was assigned to the benign class (label_bin = 0). The resulting class distribution shows approximately 97.6% attack samples and 2.4% benign samples across all splits, with a ratio of ~41:1.

```
# Create label_bin column: 0 = BenignTraffic, 1 = all attack classes
train_df['label_bin']=(train_df['label'] != 'BenignTraffic').astype(int)
```

Feature Matrix Preparation

The string-based `label` column was removed, and all 46 numerical features were retained. Feature matrices (X) were converted to `float32` for GPU/CPU compatibility, while target vectors (y) were converted to `int8`.

```
# Remove string label column and extract features
feature_columns = [col for col in train_df.columns if col != 'label_bin']
X_train = train_df[feature_columns].values.astype(np.float32)
y_train = train_df['label_bin'].values.astype(np.int8)
# Same for val and test...
```

NaN / Inf Check

All datasets were verified to contain no NaN or infinite values. The original data was already clean, requiring no additional preprocessing.

Normalization

Mean and standard deviation were calculated **only on the training set** to prevent data leakage. Zero-variance features (2 features found) were handled by setting their standard deviation to 1.0. Standardization was then applied to all datasets using the training statistics.

```
# Calculate statistics ONLY on train
mean = np.mean(X_train, axis=0, dtype=np.float32)
std = np.std(X_train, axis=0, dtype=np.float32, ddof=0)
# Handle zero variance
epsilon = 1e-8
zero_var_mask = (std < epsilon)
std[zero_var_mask] = 1.0
# Apply standardization
X_train_norm = (X_train - mean) / std
X_val_norm = (X_val - mean) / std
X_test_norm = (X_test - mean) / std
```

Binary Dataset Storage

All preprocessed datasets were saved in NumPy binary format for efficient loading during training and inference:

- **Feature matrices**: `X_train.npy`, `X_val.npy`, `X_test.npy`
- **Target vectors**: `y_train.npy`, `y_val.npy`, `y_test.npy`

Final dataset dimensions:
- Training: 5,491,971 samples × 46 features
- Validation: 1,176,851 samples × 46 features
- Test: 1,176,851 samples × 46 features

4 Algorithm implementation

What is Logistic Regression?

Logistic regression is a binary classification algorithm. Given input features X, it predicts the probability that a sample belongs to class 1 (malicious traffic) vs class 0 (benign traffic). The model learns a weight vector w (one weight per feature) and a bias term b.

Core Mathematical Operations

Linear Combination

For each sample, we compute a weighted sum of features:

```
z = b + sum(x[j] * w[j]) for j = 0 to D-1
```

where D = 46 (number of features in our dataset).

Sigmoid Activation

The linear output z is transformed into a probability using the sigmoid function:

```
float sigmoid(float z) {
    if (z > 20.0f) return 1.0f;
    if (z < -20.0f) return 0.0f;
    return 1.0f / (1.0f + expf(-z));
}
```

This maps any real number to the range (0, 1). The clamping at +/-20 prevents numerical overflow.

Prediction

For a batch of N samples:

```
for (size_t i = 0; i < N; ++i) {
    float z = b;
    for (size_t j = 0; j < D; ++j) {
        z += X[i * D + j] * w[j];
    }
    probs[i] = sigmoid(z);
}
```

The final class label is determined by threshold: if prob > 0.5, predict class 1, else class 0.

Training Algorithm

We use mini-batch gradient descent to learn the optimal weights.

Loss Function: Binary Cross-Entropy

```
loss = -mean( y * log(p) + (1-y) * log(1-p) )
```

where y is the true label (0 or 1) and p is the predicted probability.

Gradient Computation

For each sample in a mini-batch, we compute:

```
error = predicted_prob - true_label
grad_w[j] += error * x[j]   (for each feature j)
grad_b += error
```

Then average over the batch:

```
for (size_t i = 0; i < batch_size; ++i) {
    float error = probs[i] - y_batch[i];
    for (size_t j = 0; j < D; ++j) {
        grad_w[j] += error * X_batch[i * D + j];
    }
    grad_b += error;
}
// Average gradients
for (size_t j = 0; j < D; ++j) {
    grad_w[j] /= batch_size;
}
grad_b /= batch_size;
```

Parameter Update

After computing gradients, we update weights using gradient descent:

```
for (size_t j = 0; j < D; ++j) {
    w[j] -= learning_rate * grad_w[j];
}
b -= learning_rate * grad_b;
```

Training Loop Structure

One epoch processes all training samples in mini-batches:

```
for each epoch:
    shuffle training indices
    for each mini-batch:
        1. Extract batch of samples
        2. Compute predictions (forward pass)
        3. Compute gradients (backward pass)
        4. Update weights
    end
    Evaluate on validation set
end
```

Data Layout

The feature matrix X is stored in row-major order:

```
X[N][D] where:
  - N = number of samples
  - D = 46 features
  - X[i * D + j] accesses feature j of sample i
```

Labels y are stored as int8: 0 = benign, 1 = malicious.

Computational Complexity

Per mini-batch of size B with D features:

- Forward pass: O(B * D) multiplications and additions
- Gradient computation: O(B * D) multiplications and additions
- Weight update: O(D) operations

Total per epoch with N samples: O(N * D)

Memory Access Pattern

The inner loop accesses:

- X sequentially (good cache behavior for row-major layout)
- w repeatedly (fits in L1 cache since D=46, only 184 bytes)

This makes the computation memory-bound for large N, as each sample requires loading 46 floats from main memory.

Key Implementation Files

- `logistic_regression.h` - Core math functions (sigmoid, predict, gradient)
- `experiment.cpp` - Training loop with timing and metrics
- `npy_loader.h` - Loads preprocessed numpy array

5 CPU Implementation

The CPU implementation of logistic regression was developed in plain C++ to serve as the baseline for performance comparison with the GPU version. All computations are performed in single-precision floating point (float32), and the same initialization strategy, learning rate, sigmoid clamping, and mini-batch gradient descent logic are used as in the GPU implementation to ensure algorithmic parity.

The feature matrix is stored in row-major format (X[N][D]) which provides sequential memory access when iterating over features within a sample. This layout is cache-friendly for the CPU architecture since each sample's features are stored contiguously. The weight vector w has only 46 elements (184 bytes), which fits entirely in L1 cache and is reused across all samples in a batch.

All core operations are implemented as explicit loops without external library dependencies. The forward pass computes the dot product of each sample with the weight vector, adds the bias, and applies the sigmoid function. Gradient computation accumulates partial gradients across all samples in a mini-batch using a single pass through the data, then averages the result. Weight updates are applied immediately after processing each mini-batch.

To maintain algorithmic parity with the GPU version, training samples are shuffled at the beginning of each epoch using a fixed random seed (default 42). Mini-batches are then extracted by copying the relevant samples into a contiguous batch buffer based on the shuffled indices, rather than accessing them in-place with indirect indexing.

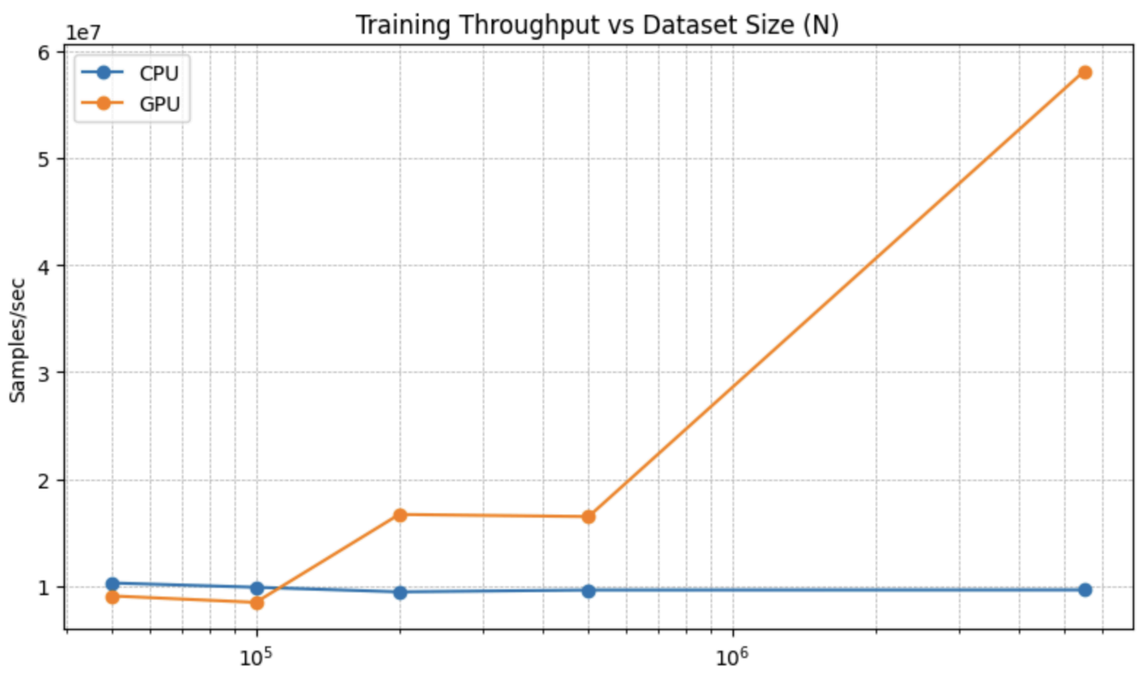
Training time measurements are taken using high-resolution chrono timers and include only the compute phase (gradient computation and weight updates), excluding data loading and preprocessing. The first epoch is treated as warm-up and excluded from timing statistics. Inference performance is measured separately using repeated forward passes after a warm-up phase to obtain stable latency and throughput estimates.

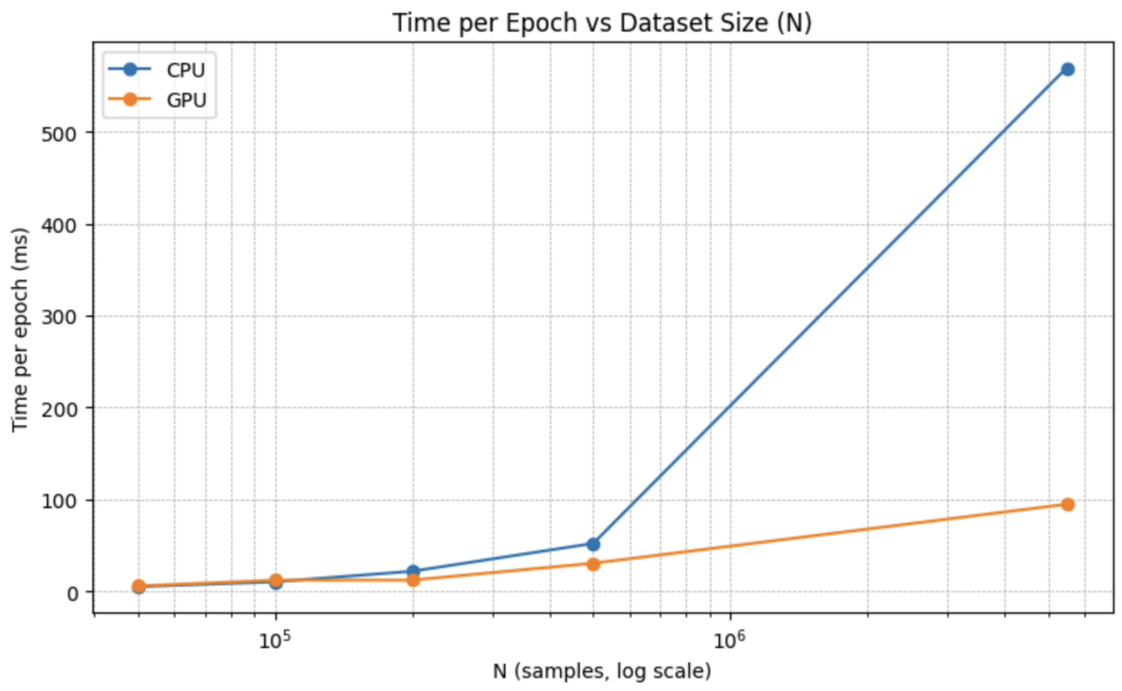
6 GPU Implementation

The GPU implementation of logistic regression was developed in CUDA to match the CPU algorithm exactly(The logistic Regression algorithm), to make sure that there is a fair performance comparison. All computations are performed in single-precision floating point, and the same initialization strategy, learning rate, sigmoid clamping, and mini-batch gradient descent logic are used. The feature matrix is stored in column-major format on the device to align with cuBLAS expectations, to make the core linear algebra operations to be expressed as sgemv calls for both forward propagation and gradient computation. Lightweight custom CUDA kernels are used for element-wise operations, including sigmoid evaluation, error computation, gradient scaling, weight updates, and bias reduction. To keep algorithmic parity with the CPU version, training samples are shuffled on the host at the beginning of each epoch using a fixed seed and then gathered on the GPU into mini-batches using an indexing kernel, instead of relying on contiguous memory slices. Training time measurements are taken using CUDA events and include only the compute phase, excluding data loading and preprocessing. Inference performance is measured separately using repeated forward passes after a warm-up phase to obtain stable latency and throughput estimates.

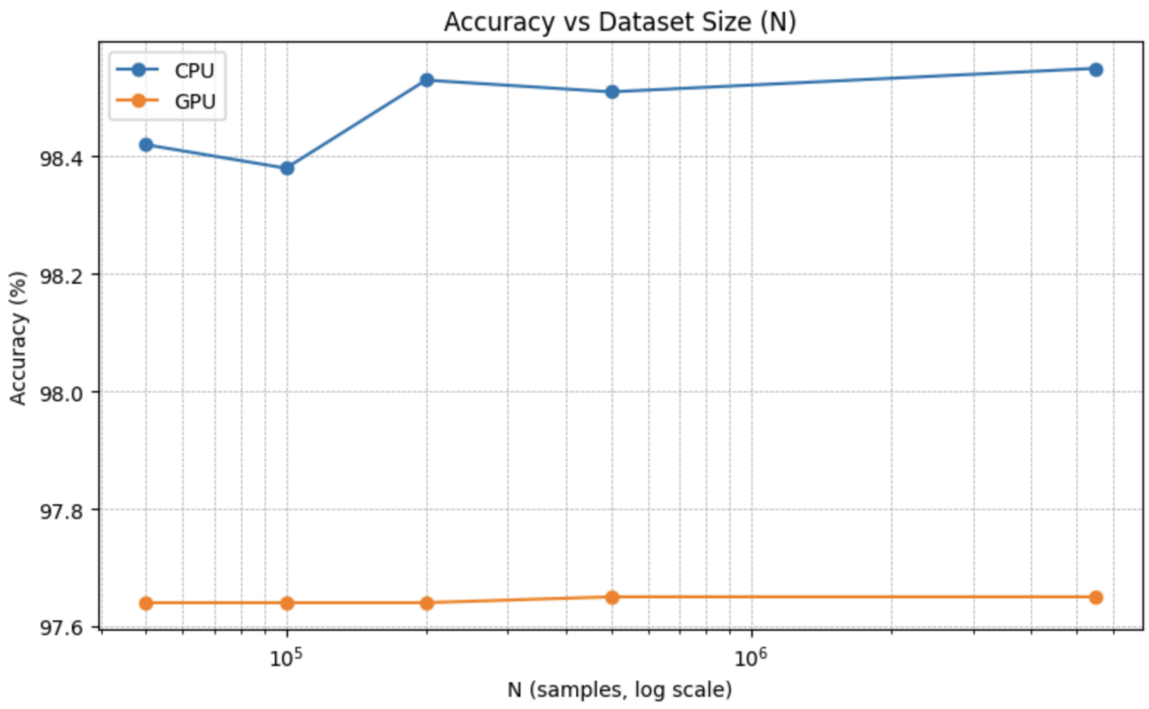
7 Results

Imp	N	D	Batch	Epochs	Threads	Time/Epoch (ms)	Samples/Sec	Acrurracy	Loss
GPU	50000	46	512	5	N/A	5.5824	9.09E+06	97.64%	0.10853
GPU	100000	46	512	5	N/A	11.8632	8.49E+06	97.64%	0.10857
GPU	200000	46	1024	32	N/A	11.95	1.67E+07	97.64%	0.10886
GPU	500000	46	1024	10	N/A	30.3074	1.65E+07	97.65%	0.11009
GPU	5491971	46	4096	5	N/A	94.5095	5.81E+07	97.65%	0.1117
CPU	50,000	46	512	5	1	4.87	1.03E+07	98.42%	0.22370
CPU	100,000	46	512	5	1	10.11	9.90E+06	98.38%	0.13530
CPU	200,000	46	1024	32	1	21.59	9.47E+06	98.53%	0.04640
CPU	500,000	46	1024	10	1	51.89	9.64E+06	98.51%	0.05140
CPU	5,491,971	46	4096	5	1	569.34	9.65E+06	98.55%	0.04520

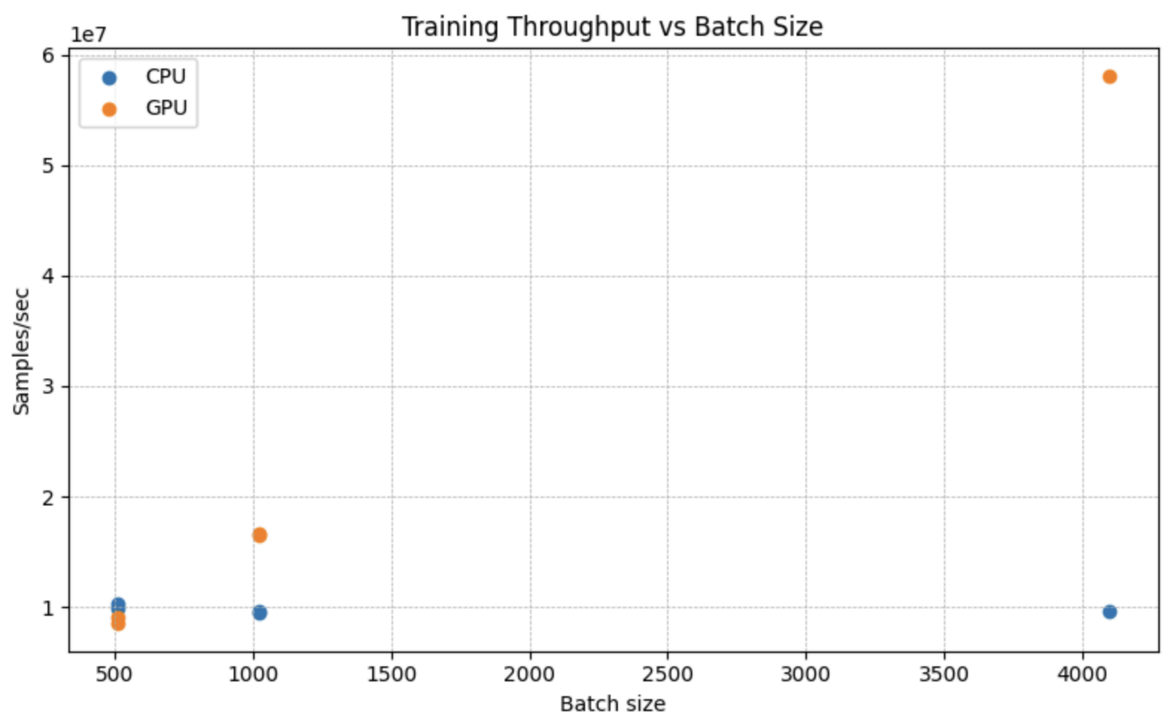


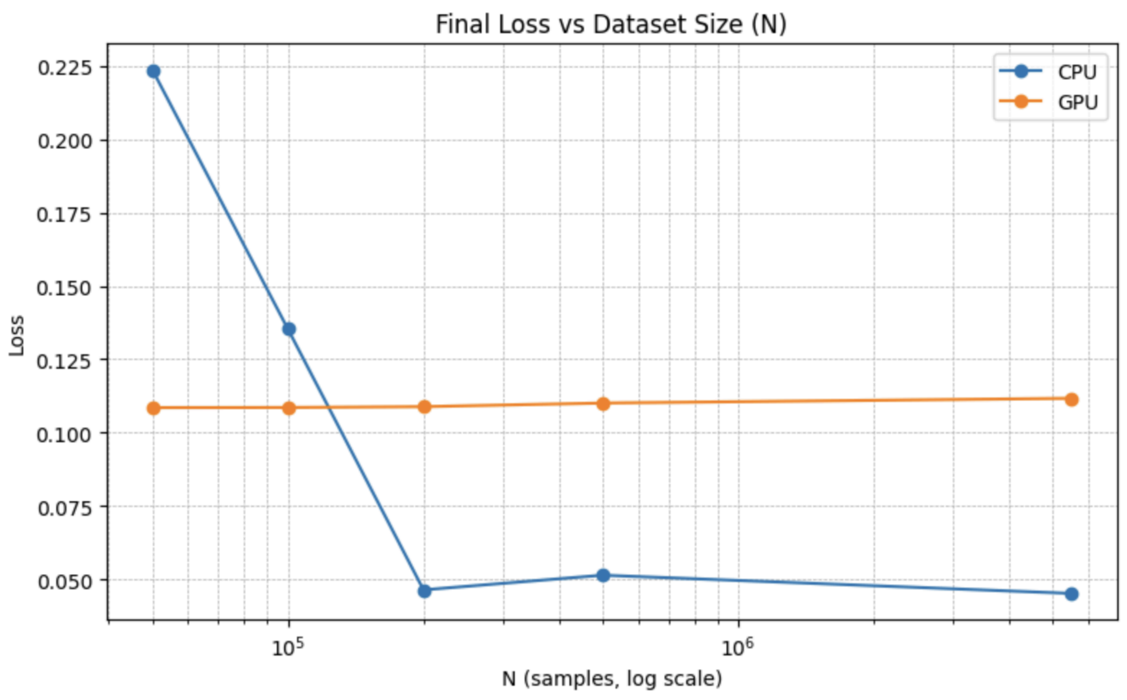


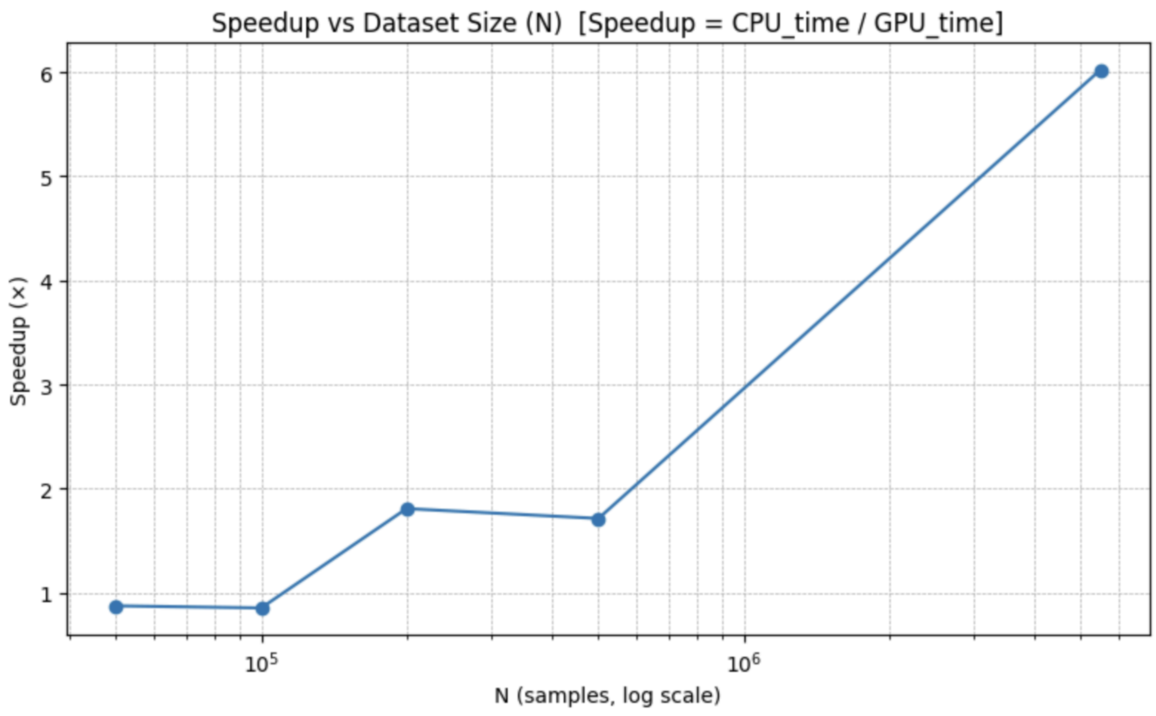
Figures 1 and 2 show the training throughput and time per epoch as a function of dataset size. For small datasets (50k–100k samples), CPU throughput is comparable to or slightly higher than GPU throughput (≈1.0×10^7 vs. ≈9.0×10^6 samples/sec), which means that GPU execution is dominated by kernel launch and memory access overheads at this scale. As the dataset size increases beyond 200k samples, GPU throughput improves substantially and begins to saturate, reaching approximately 5.8×10^7 samples/sec for the full dataset of 5.49 million samples (Figure 1). While, CPU throughput stays nearly constant at around 1.0×10^7 samples/sec across all dataset sizes, which shows limited scalability for this workload. This divergence is reflected in the time-per-epoch results (Figure 2), where GPU training time grows much more slowly than CPU training time as N increases.



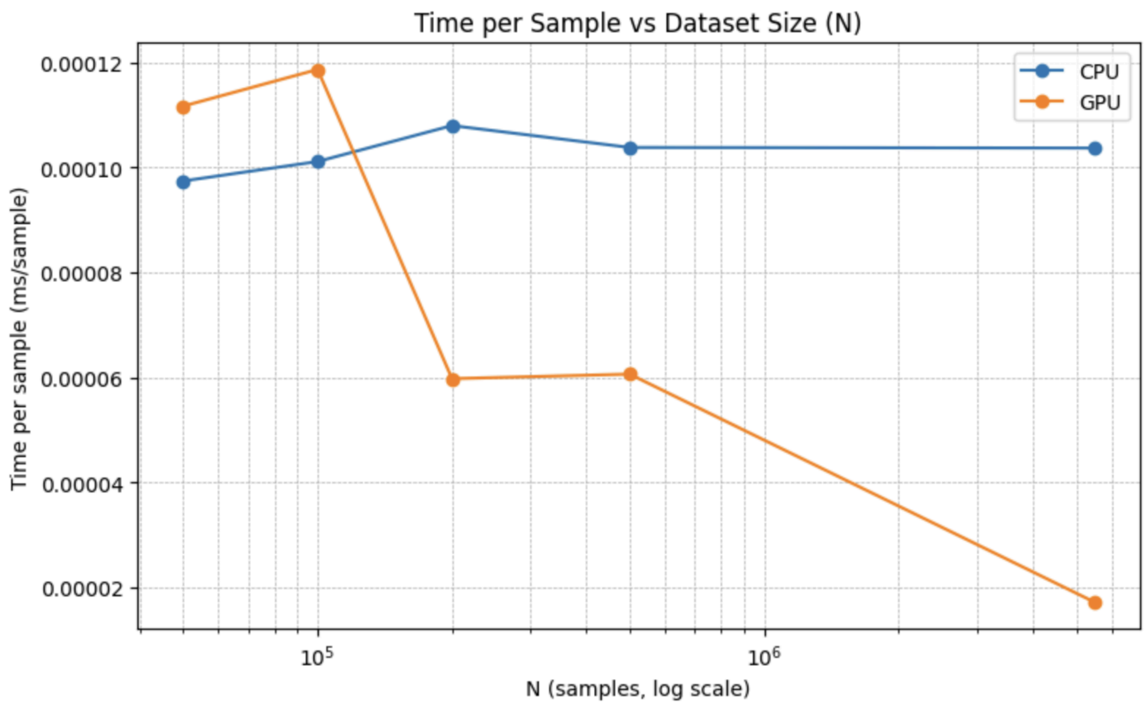
The impact of batch size on GPU performance is shown in Figure 4. Increasing the batch size from 512 to 4096 leads to a substantial increase in GPU throughput, indicating improved device utilization and reduced per-batch overhead. The CPU implementation shows little sensitivity to batch size, which shows its limited ability to exploit additional parallelism for this problem. Despite the large performance differences, both implementations exhibit stable and consistent convergence behavior. As shown in Figure 3, classification accuracy stays nearly constant across dataset sizes, with GPU accuracy stabilizing around 97.64–97.65% and CPU accuracy around 98.4–98.6%.


Training Throughput vs Batch Size
1e7
6
5
4
3
2
1
Samples/sec
500
1000
1500
2000
2500
3000
3500
4000
Batch size
CPU
GPU


Final Loss vs Dataset Size (N)
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.050
Loss
10^5
10^6
N (samples, log scale)
CPU
GPU



At the largest dataset size, the GPU completes an epoch in approximately 95 ms, compared to 569 ms on the CPU, which is an overall speedup of about 6× (Figure 6). These trends confirm that once the workload is large enough to amortize overheads, the GPU's parallelism and higher memory bandwidth provide significant performance advantages.



Final loss values (Figure 5) follow similar trends, with small numerical differences attributable to floating-point ordering and parallel reduction effects rather than algorithmic discrepancies. The time-per-sample plot (Figure 7) further emphasizes the scalability advantage of the GPU, which shows a decreasing per-sample cost as dataset size increases, while CPU per-sample time stays roughly constant. Overall, these results show that logistic regression with a low feature dimension (D = 46) is moderately memory-bound, and that GPU acceleration becomes increasingly effective as dataset size and batch size grow.

8 Conclusion

In this project, we looked at how CPU and GPU performance compares when running logistic regression. To make sure we were seeing a fair comparison, we kept the core settings, like feature count, learning rate, and precision, exactly the same. This allowed us to focus entirely on how the hardware handles the work as the datasets grow. We tested everything from a small set of 50,000 samples up to the full 5.49 million, which helped us identify the specific point where the GPU's parallel processing actually becomes an advantage.

The results show that for smaller datasets, the CPU is often just as fast, or even slightly faster, because it doesn't have the overhead of transferring data to a separate processor. However, once the workload exceeds 200,000 samples, the GPU's speed increases significantly. It reached a peak throughput of 5.8 \times 10^7 samples per second, which is roughly 6× faster than the CPU's consistent rate of 1.0 \times 10^7. Since the accuracy and loss were identical for both, we can confirm that these performance gains come strictly from the hardware architecture. Ultimately, these tests show that while the CPU is efficient for smaller scales, the GPU is much more effective once the dataset is large enough to saturate its cores.

Resources used for a project

1. Sun, H. *Machine Learning C++: Logistic Regression Example*. Medium. https://medium.com/@dr.sunhongyu/machine-learning-c-logistic-regression-example-1bec1168db97
2. G. Molnar. *Hands-On Machine Learning with C++*. Packt Publishing, 2021.
3. Brownlee, J. *Basic Data Cleaning for Machine Learning*. Machine Learning Mastery. https://machinelearningmastery.com/basic-data-cleaning-for-machine-learning/
4. KDnuggets. *Essential Data Cleaning Techniques for Accurate Machine Learning Models*. https://www.kdnuggets.com/essential-data-cleaning-techniques-accurate-machine-learning-models
5. Rogers, C. *cnpy: NumPy Array Loading and Saving for C++*. GitHub repository. https://github.com/rogersce/cnpy
6. Lei Mao. *Load and Save NumPy Files in C++*. https://leimao.github.io/blog/CPP-Npy-Load-Save-xtensor/
7. Terminal Root. *How to Use the zlib Library with C++*. https://terminalroot.com/how-to-use-the-zlib-library-with-cpp/
8. NVIDIA Corporation. *cuBLAS Library User Guide*. https://docs.nvidia.com/cuda/cublas/index.html
9. Quf. *cnpy Alternative Implementation*. https://sr.ht/~quf/cnpy/
10. NVIDIA Corporation. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
11. Oak Ridge Leadership Computing Facility. *CUDA C Basics*. https://www.olcf.ornl.gov/wp-content/uploads/2019/12/01-CUDA-C-Basics.pdf
12. Boston University Research Computing. *CUDA C/C++ GPU Computing*. https://www.bu.edu/tech/support/research/software-and-programming/gpu-computing/cuda-c/
13. Bisqwit. *How to Use OpenMP*. https://bisqwit.iki.fi/story/howto/openmp/
14. GeeksforGeeks. *Parallel for Loop in OpenMP*. https://www.geeksforgeeks.org/c/c-parallel-for-loop-in-openmp/
15. West Virginia University HPC Center. *Introduction to OpenMP*. https://wvuhpc.github.io/2018-Lesson_4/03-openmp/index.html
16. cplusplus.com. *Discussion on C++ Performance and Multithreading*. https://cplusplus.com/forum/general/282113/
17. ResearchGate. *Speedup with Increasing Thread Count in Parallel Applications*. https://www.researchgate.net/figure/The-speedup-with-the-increasing-thread-number-for-the-evaluation-of-the-internal-forces_fig4_327893787
18. FavTutor. *Multithreading in C++*. https://favtutor.com/blogs/multithreading-cpp
19. Aerospike. *CPU vs GPU: Performance Characteristics*. https://aerospike.com/blog/cpu-vs-gpu/
20. LearnCppPlusPlus. *Understanding Sigmoid Functions in Neural Networks*. https://learncplusplus.org/what-you-need-to-know-about-sigmoid-functions-in-neural-nets/
21. Olamendy, J. *The Power of Batch Size in Gradient Descent*. Medium. https://medium.com/@juanc.olamendy/the-power-of-batch-size-a-comprehensive-guide-to-gradient-descent-23f25a2183ff

Siena 2026