MNIST With Parallelism

Liam Pilarski
Georgia Institute of Technology

Abstract

This project delves into a classic machine learning problem, neural networks trained on the MNIST handwritten digit dataset, but with a twist: a heavy reliance on parallelism. This is a topic of much importance as AI models grow in size. Large models rely on distributed and parallel computing to efficiently train and operate as datasets grow exponentially larger and as the models themselves increasingly involve operations that are prohibitively slow to execute in serial. Processing large datasets in parallel and executing compute operations in parallel are two ways to speed up the model training process, and are explored in detail in this report.

1. Introduction

For this demonstration, I implement a modular codebase in the C programming language that allows for creation of custom models. My understanding of the mathematics behind this project, i.e. calculation and application of gradients using backpropagation, comes from Georgia Tech coursework in classes such as CS 4641, CS 4644. Additionally, I referenced external resources such as Michael Nielsen's book, *Neural Networks and Deep Learning* [1] and NotesByNick's *CUDA Crash Course: Cache Tiled Matrix Multiplication* [2]. This project utilizes three forms of parallelism: OpenMPI for data parallelism, CUDA for GPU access, and OpenMP for shared memory.

2. Technical Approach

2.1. Codebase

I began by implementing the codebase without parallelism to get the core functionality working. An mnist_loader.c file handles loading, formatting and splitting into test and train sets the data to be utilized during training. In train.c, this data is passed to a Model struct, which consists of an input buffer to load the data in, an array of layers (of type dense, sigmoid, leakyReLU or softmax), and a cross entropy loss struct. Each layer has its own forward method, which takes output from a previous layer and processes and passes it to the next layer, and its own backward method, which takes upstream gradients from a following layer, updates weights accordingly, and processes and passes it to the preceding layer as downstream gradients. For the purpose of this project, I used model architecture illustrated in Figure 1:

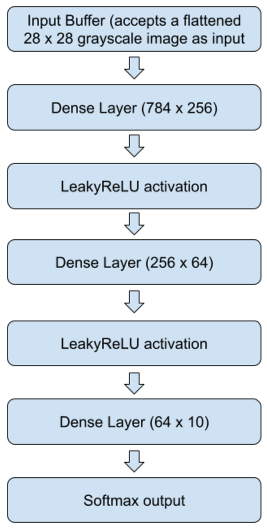


Figure 1. Sample predictions from model with corresponding ASCII art of the input digits

After completing the initial framework for the codebase, I was able to begin working on introducing parallelism.

2.2. OpenMPI

OpenMPI was utilized for data parallelism. Rather than using a single machine to train on all of the data in the dataset, I split the data into N chunks and distribute the data evenly across N machines. Each machine's trainable parameters are initialized exactly the same. For each batch, the generated gradients of the entire model are copied into a buffer, `broadcast_weight_grads` , and are averaged across the N machines before being used for gradient updates. This ensures that model weights across all N machines remain identical, while theoretically reducing the time to train over the entire dataset by a factor of N (not counting costs of communication overhead). Additionally, usage of OpenMPI in this manner greatly simplifies load balancing considerations. Forwarding data through the model takes the same time regardless of what machine is running, so evenly distributing the data across machines inherently balances the workload, as each machine processes an equal number of batches that require roughly the same computation time.

2.3. OpenMP

To support environments with and without GPU access, my codebase leverages OpenMP for CPU-based parallelization, defaulting to four threads for consistent performance. I introduced `#pragma omp parallel for` directives to parallelize computationally intensive loops wherever feasible. For many tasks, such as element-wise matrix operations (e.g., multiplying all elements by a scalar), parallelization was straightforward due to their inherently independent nature, yielding significant speedups.

However, matrix multiplication and the softmax function required careful consideration due to data reuse and memory access patterns. For matrix multiplication, I optimized cache locality by restructuring the computation to process partial products row-by-row, aligning with the row-major memory layout. This approach minimizes cache misses by enhancing spatial and temporal locality, particularly when repeatedly accessing shared elements of matrix B in the operation A @ B. Specifically, I reorganized the loop over the inner dimension k, splitting it into two stages: an initial pass computes the first term, and subsequent terms accumulate into the result using local temporaries. By combining this memory-efficient design with OpenMP's `collapse(2)` pragma for parallelizing nested loops, I achieved both multithreading speedups and improved memory efficiency.

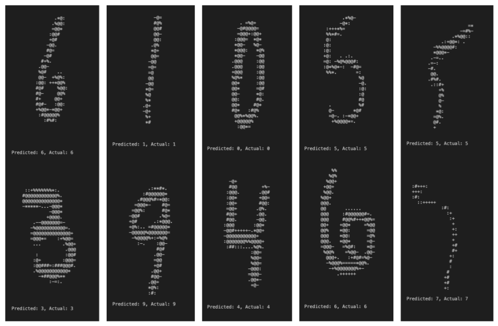
For the cross-entropy layer, aggregating average loss and accuracy necessitated `reduction` clauses in the `#pragma` directives to ensure thread-safe updates.

2.4. CUDA

For environments with NVIDIA GPUs, I implemented CUDA equivalents for each OpenMP-parallelized function, naming them `cuda_{function_name}` and invoking corresponding `cuda_{function_name}_kernel` kernels. As with OpenMP, simple operations like element-wise matrix computations translated easily to CUDA. However, matrix multiplication posed similar memory access challenges as in the OpenMP implementation, where naive approaches incur significant inefficiencies due to repeated global memory accesses. Building on the memory-aware design from the OpenMP section, I adopted a tiled matrix multiplication strategy using CUDA's shared memory [2]. This approach loads sub-blocks of input matrices into fast, on-chip shared memory, drastically reducing redundant global memory accesses. Unlike the OpenMP implementation, which relies on CPU cache hierarchies and row-major traversal, CUDA's tiled approach leverages the GPU's parallel thread blocks and shared memory to optimize data reuse within each block. This not only accelerates computation but also enhances memory throughput and cache utilization. Given matrix multiplication's central role in machine learning workloads, this optimization significantly improves performance across models, complementing the memory-efficient design principles established in the CPU-based implementation.

2.5. Verification Tests

After getting my codebase working without parallelism, I wrote code to allow me to visually see my model's output (Figure 2) and confirm whether reported classification accuracy scores made sense. After getting these visualizations working and confirming that high accuracy scores weren't being misreported while the model was performing poorly, I was able to use accuracy as a reliable indicator as to whether my model was still operating as expected as I began integrating parallelism. Additionally, I trained my unoptimized model on the entire dataset before integrating parallelism to get a baseline accuracy (95.3%) that my optimized model should be able to hit, assuming there were no implementation errors on my part.



```
Predicted: 6, Actual: 6
Predicted: 1, Actual: 1
Predicted: 0, Actual: 0
Predicted: 5, Actual: 5
Predicted: 5, Actual: 5
Predicted: 3, Actual: 3
Predicted: 9, Actual: 9
Predicted: 4, Actual: 4
Predicted: 6, Actual: 6
Predicted: 7, Actual: 7
```

Figure 2. Model architecture

Usage of unit tests also helps verify implementation correctness. For individual operations (matrix multiply, transpose, etc.) and each layer (softmax, dense, leakyReLU, etc.), I calculate by hand the expected output for small example inputs. I then set up test functions that take in these small example inputs and compare them against my calculated expected outputs. After making changes, I execute my test file to ensure that each component of my code continues to work as expected.

3. Results

3.1. Accuracy

In order to verify that my code was working as expected, I ran several tests using accuracy metrics and visual confirmation. I split my dataset (60000 train images, 10000 test images) into 4 equal chunks (chunks 0 through 3). For testing without MPI as a baseline, I trained one model with OpenMP and one model with CUDA on chunk 0 (15000 train images, 2500 test images). In terms of accuracy, due to my use of seeding, both of these models performed identically, further confirming implementation correctness. Their training curve is shown in Figure 3.

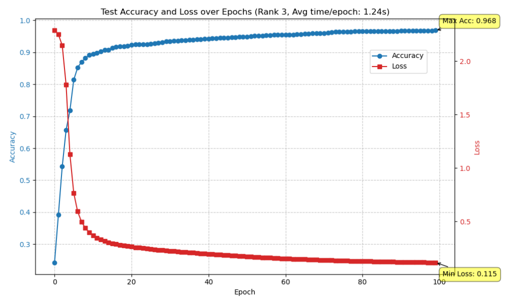


Figure 3. Accuracy / loss curve for non-MPI models over 100 epochs. Due to use of random seeding, results for the OpenMP and CUDA models are identical. Only one plot is pictured to avoid redundancy.

My hypothesis was that when training with OpenMPI enabled gradient sharing across all 4 chunks, accuracy would improve, as my codebase's gradient sharing mechanism would allow the model to learn from gradients obtained from chunks other than the chunk each machine was handling itself. For consistency, when recording results while using MPI, I took results specifically from the machine handling chunk 0. This was indeed the case: compared to the non-MPI enabled model trained only on chunk 0, the MPI enabled model on the machine training with chunk 0 displayed improved performance, with a maximum accuracy of 97.4% compared to 96.8% (Figure 4).

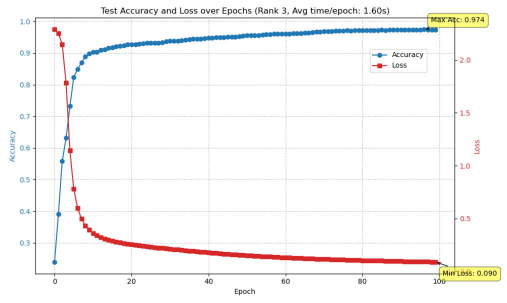


Figure 4. Accuracy / loss curve for MPI models over 100 epochs. Improved accuracy is seen compared to Figure 3.

3.2. Latency

Another key metric I recorded was the time per epoch for each of the 4 model variations I tried. The CUDA implementation was the fastest, coming in at around 0.23 seconds total per epoch. OpenMP was slower, and the naive approach, without speedups from OpenMP or CUDA, was the slowest (the timing for the naive approach was recorded by running the code with only 1 OpenMP thread enabled, effectively behaving as if there was no parallelism). While using MPI resulted in improved accuracy, it does come with the cost of additional communication overhead between machines. This overhead should in theory be constant no matter what task level parallelism. This hypothesis is confirmed by inspection of the slowdown penalty experienced by each of the modes of task level parallelism when combined with MPI, which hovers at around 0.30 seconds regardless of what type of task level parallelism was employed.

Table 1. Time cost for different parallelism configurations

Parallelism Configuration	**Time per epoch (s)**	**MPI cost (s)**
CUDA only	0.23	N/A
OpenMP only (4 threads)	1.24	N/A
Naive (no OpenMP or CUDA)	2.56	N/A
CUDA + MPI	0.52	0.29
OpenMP + MPI (4 threads)	1.60	0.36
Naive + MPI	2.93	0.37

Note: The MPI epoch times are calculated with four machines, each processing $\frac{1}{4}$th of the dataset. Similarly, the non-MPI epoch times are calculated for a single machine processing $\frac{1}{4}$th of the dataset. These times are to illustrate the overhead associated with using MPI while the problem size (# of datapoints processed) per machine remains constant.

3.3. Scaling / Performance studies

I measured strong scaling in this project for my OpenMP implementation specifically by varying the number of OpenMP threads my code could use at any given time (Figure 5). An interesting trend emerges. For smaller numbers of OpenMP threads, there is a clear improvement in time per epoch as we increase the number of threads from 1 to 8. However, the overhead of managing multiple threads is apparent through the decreasing marginal utility of increasing the number of threads, as evidenced by the minute gains achieved from going from 4 threads to 8 threads. After 8 threads, additional threads actually hurt performance, presumably by increasing overhead more than decreasing overall execution time. Presumably, with a more complex model / problem, where there are more parallelizable operations per layer, we would see this U-shaped curve bottom out at a larger number of threads.

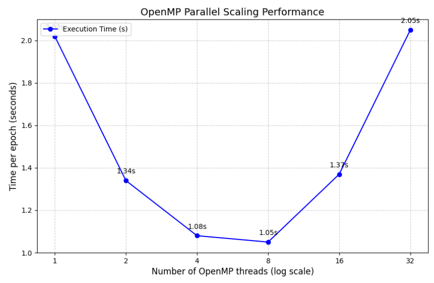


Figure 5. Time per epoch varying with number of OpenMP threads

4. Conclusion

The goal of this project was to implement a modular neural network training framework in C and explore the effects of parallelism—specifically OpenMP, CUDA, and OpenMPI—on both training speed and model performance using the MNIST dataset. Through careful construction of a base codebase and systematic integration of parallelism, I was able to isolate and evaluate the impact of each method on training efficiency and model accuracy.

The results clearly demonstrate that parallelism can significantly accelerate neural network training without compromising accuracy. CUDA provided the greatest time savings, reducing epoch duration to just 0.23 seconds, while OpenMP offered considerable speedup over a naive implementation. When distributed training with OpenMPI was introduced, model accuracy improved—achieving a maximum of 97.4%—due to the effective aggregation of gradients across partitions of the dataset. However, this accuracy gain came at the cost of additional communication overhead.

These findings underscore the importance of hybrid parallelism in modern machine learning pipelines, especially as models continue to scale. The tradeoffs between communication overhead and computational gain must be carefully balanced depending on the hardware and scale of the problem. The OpenMP scaling studies suggest that for relatively small networks like the one used here, thread-level parallelism plateaus quickly, but larger, more complex models could benefit more substantially.

In future work, this framework could be extended to support model parallelism in addition to data parallelism and scale to larger datasets and deeper networks, with a more diverse array of layers (i.e. transformers, convolutional layers, etc.). Ultimately, this project serves as an experiment demonstrating the broader challenge of accelerating AI through efficient systems design and parallelism, an essential field as neural networks continue to grow in size, complexity and impact.

References

[1] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com/. 1

[2] NotesByNick. Tiled matrix multiplication — cuda programming. https://www.youtube.com/watch?v=3xfyiWhtvZw, n.d. Retrieved April 17, 2025. 1, 2