NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY POLITEHNICA BUCHAREST
Faculty of Electronics, Telecommunications and Information Technology

PAO Project Documentation

Multilayer Perceptron Training Acceleration on Fashion-MNIST dataset

Author: Eng. Blîndu Andrei-Samuel

2024-2025

Table of Contents

1. Incentive for the project 2
2. Introduction 2
3. State-of-the-Art 4
3.1. Performance Comparison of Multilayer Perceptron (Back Propagation, Delta Rule and Perceptron) algorithms in Neural Networks 4
3.2. Acceleration of Backpropagation Training with Selective Momentum Term 4
3.3. Training Acceleration of Multi-Layer Perceptron using Multicore CPU and GPU under MATLAB Environment 5
3.4. An Accelerated Learning Algorithm for MLP Layer by Layer (OLL) 5
3.5. An Accelerated Learning Algorithm for Multilayer Perceptron Networks (ABP) 6
3.6. FPGA acceleration on a multi-layer perceptron neural network for digit recognition 6
3.7. Parallelizing Backpropagation Neural Network Using MapReduce and Cascading Model 7
3.8. Accelerating Neural Network Training Process on Multi-Core Machine Using OpenMP 8
3.9. OpenCL Optimization for FPGA Deep Learning Application 8
3.10. Parallel Backpropagation Neural Network Training Techniques using Graphics Processing Unit 9
3.11. Hardware Accelerated Neural Networks (Thesis) 9
4. Selected application 10
5. Single-threaded naive implementation 10
5.1. Initializations 11
5.2. Training 11
5.3. Performance Summary 15
6. Optimization limitations and opportunities 15
6.1. Theoretical performance limits 15
6.2. CPU optimized 16
6.2.1. Compilation flags 16
6.2.2. OpenMP 17
6.2.3. AVX2 17
6.3. GPU optimized - OpenCL 18
7. Results 19
8. Conclusions 20
9. References 21

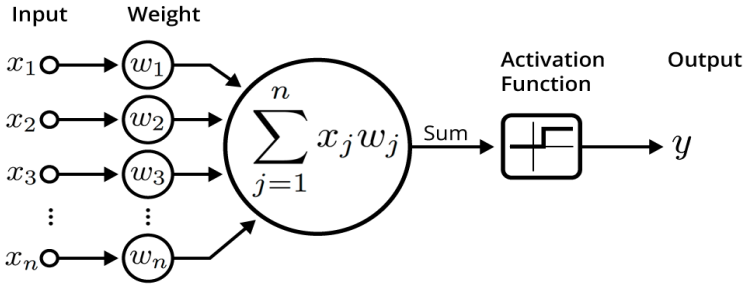
1. Incentive for the project

The purpose of this project is to accelerate the training time required for a multilayer perceptron (MLP) to achieve significant classification performance on large datasets. Since all interconnected layers contain associated weights that must be adapted in response to training data features, sequential training becomes computationally intensive, and architecture optimization proves time-consuming. Heterogeneous computing has emerged as a promising solution to address these challenges, and this project focuses on both software and hardware approaches for accelerating MLP training and testing processes.

2. Introduction

Machine learning represents a transformative computational paradigm that enables systems to automatically learn patterns and make predictions from data without explicit programming for each specific task. Supervised learning constitutes a fundamental branch of machine learning where algorithms learn to map input data to desired outputs using labeled training examples. In supervised learning frameworks, the system receives a known set of input-output pairs, called patterns, and trains a model to generate accurate predictions for previously unseen data. The effectiveness of supervised learning algorithms depends critically on the quality and quantity of training data, with complex, multidimensional problems typically requiring hundreds of thousands or even millions of training examples to achieve optimal performance.

MLPs represent one of the most fundamental and widely used architectures in supervised neural network learning, consisting of interconnected layers of computational nodes called neurons. “Each unit performs a relatively simple job. Receive input from neighbors or external sources and use this to compute an output signal which is propagated to other units. Apart from this processing, a second task is the adjustment of the weights. The system is inherently parallel in the sense that many units can carry out their computations at the same time” [11]. Each neuron in the network calculates a weighted sum of its inputs and applies an activation function—commonly the sigmoid function sigmoid(x) = 1/(1+e^-x)—to introduce nonlinearity into the system:



An illustration of an artificial neuron. Source: Becoming Human.

Figure 1. The MLP neuron

An MLP typically comprises an input layer that receives data, one or more hidden layers that perform nonlinear transformations, and an output layer that produces final predictions (see Figure 2). The training process involves two primary phases: **feed-forward propagation**, where input stimuli flow from the input layer through hidden layers to generate output predictions, and **backpropagation**, which computes gradients of the loss function with respect to network weights and iteratively adjusts these parameters to minimize prediction errors. This optimization process enables MLPs to learn

complex nonlinear mappings between inputs and outputs, making them powerful tools for pattern recognition, function approximation, and classification tasks across numerous engineering and scientific applications.

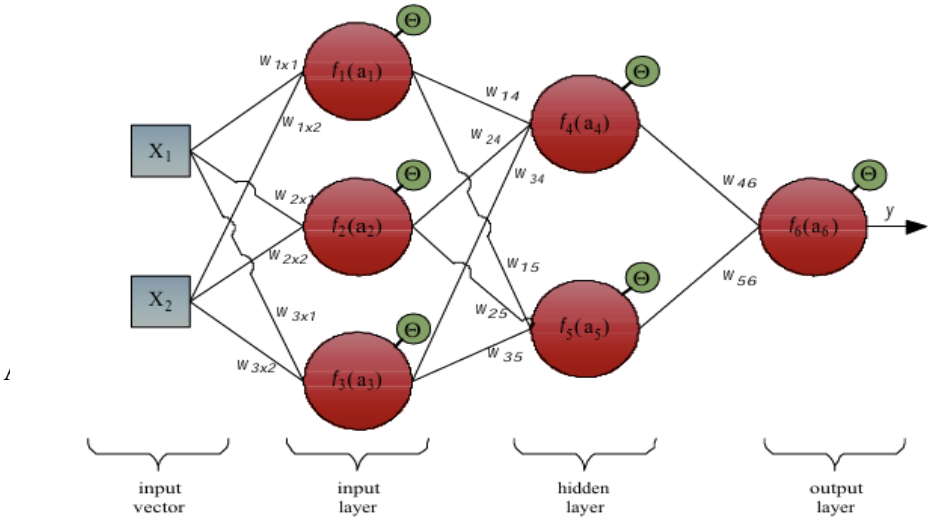


Figure 2. Feed-Forward network topology [11]

In case of the **forward propagation** phase of the algorithm, the weighted sum at each node is calculated using [12]:

$$aj = \sum(wi,j * xi)$$

Where,

- *aj* is the weighted sum of all the inputs and weights at each node;
- *wi,j* represents the weights between the i-th input and the j-th neuron;
- *xi* represents the value of the i-th input;

After applying the activation function to *aj*, we get the output of the neuron, which will serve as input into the next layer:

$$oj = \text{activation function}(aj)$$

Where activation function can be sigmoid, ReLU or others.

Finally, the error is calulated as the correction factor used to be backpropagated into the archtiecture and which will consequently update the weights:

$$Errorj = ytarget - yj$$

In the case of **backpropagation**, the change in weight is determined by [12]:

$$\Delta w_{ij}(n) = -\eta \times \partial \mathscr{E}(n)/\partial v_j(n) \times y_i(n)$$

Where:

- *$\partial \mathscr{E}(n)/\partial v_j(n)$ is the partial derivate of the error according to the weighted sum of the input connections of neuron (backpropagation makes use of the chain rule of calculus to calculate the gradient of the loss function with respect to each weight).*
- *η is the learning rate;*

For classification, the highest value of the output layer represents the MLP's prediction.

3. State-of-the-Art

3.1. Performance Comparison of Multilayer Perceptron (Back Propagation, Delta Rule and Perceptron) algorithms in Neural Networks

Alsmadi et al. conduct a systematic comparative analysis of three fundamental training algorithms for multilayer perceptrons: backpropagation, delta rule learning, and perceptron algorithms. The study employs fish pattern recognition as the benchmark problem, utilizing 12 distinct patterns with 7 designated for training and 5 for validation. The research methodology maintains consistent network parameters across all algorithms to ensure fair comparison, focusing on convergence characteristics and training efficiency for classification tasks.

The experimental results demonstrate significant performance variations among the algorithms. The backpropagation algorithm achieved convergence in 1,277 iterations over 15 seconds with a final error of 3.55186, while delta rule required over 2,500 iterations across 30 seconds without satisfactory convergence. The perceptron algorithm exhibited superior performance, processing 25,958 neurons over 45 minutes with 191 total inputs and achieving an exceptionally low error rate of 0.0019999, establishing it as the optimal choice for the tested multilayer perceptron classification scenarios [1].

Table 2. Back Propagation Results

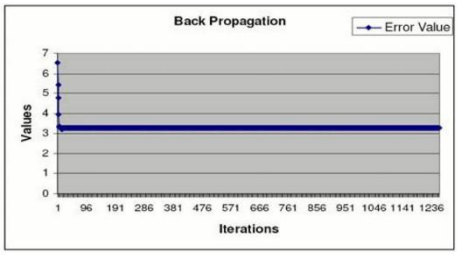
Training Algorithm	NO. Neurons	NO. Iterations	Elapsed Time	Actual Error
Back Propagation	3	1277	15 sec	3.55186

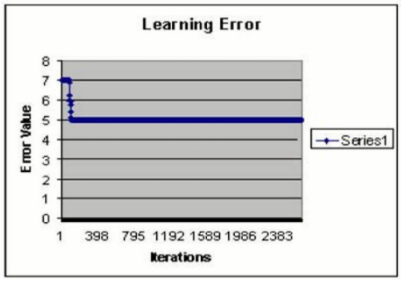
Table 3: Delta rule Results

Training Algorithm	NO. Neurons	NO. Iterations	Elapsed Time	Actual Error
Delta Rule	-	2866	30 sec	4.9999

Table 4. perceptron learning result

Training Algorithm	NO. Neurons	NO. Iterations	Elapsed Time	Actual Error
Perceptron		25958	45 minute	0.0019999





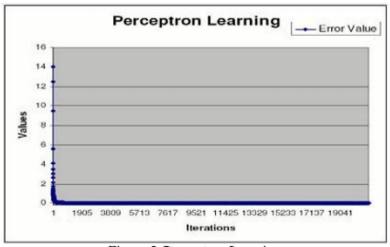


Figure 5. Perceptron Learning

Figure 3. Backpropagation, Dela rule and Perceptron learning curves [1]

3.2. Acceleration of Backpropagation Training with Selective Momentum Term

Carvalho and Neto introduce a selective momentum application methodology that applies momentum terms only to specific network weights based on correlation coefficient analysis. The approach utilizes Pearson and Spearman correlation coefficients to identify weights exhibiting strong linear relationships with network error, selectively applying maximum momentum ($\alpha = 1$) to these weights while leaving others unmodified. This strategy addresses the conventional practice of uniform momentum application across all weights regardless of their individual contribution to error reduction.

The methodology achieves remarkable training acceleration with iteration reductions ranging from 50.06% to 97.70% compared to traditional backpropagation. Networks with 7 hidden neurons completed training in 6.28 seconds using selective momentum compared to 88.27 seconds with

traditional backpropagation, representing a 93% reduction in training time. The approach maintains classification accuracy at 98.12% on average while providing 62% faster convergence than standard momentum methods and completely avoiding local minima entrapment that affected maximum momentum implementations [2].

3.3. Training Acceleration of Multi-Layer Perceptron using Multicore CPU and GPU under MATLAB Environment

Dawwd and AL Layla investigate parallel processing implementations for neural network training acceleration using both multicore CPU and GPU architectures within MATLAB. The research employs encoder networks of varying complexities across five datasets (P1-P5) with computational loads ranging from 256 to 65,536 patterns. The study implements pattern parallel training where training datasets are distributed across processing cores, with systematic evaluation of performance scaling characteristics relative to problem size and hardware configuration.

The multicore CPU implementation achieves up to **4.16x** speedup using 8-core processors for large datasets, though small datasets experience performance degradation due to communication overhead. GPU implementation delivers exceptional results with up to **304.6x** speedup compared to single-core CPU processing for large-scale problems using a GeForce 610M with 48 cores. The GPU approach demonstrates 98% efficiency utilization for large datasets, processing them in approximately 386 seconds compared to 117,628 seconds for single-core CPU execution, establishing GPU acceleration as the optimal solution for large-scale neural network training [3].

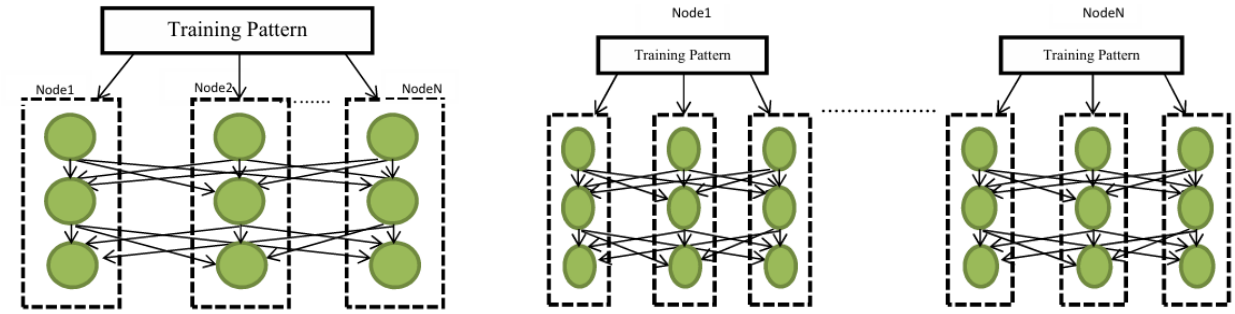


Figure 4. Training parallelizing schemes for multi-core CPUs (left) and computer clusters (right) [3]

3.4. An Accelerated Learning Algorithm for MLP Layer by Layer (OLL)

Ergezinger and Thomsen develop the Optimization Layer by Layer (OLL) learning algorithm that optimizes multilayer perceptron weights through iterative layer-wise optimization rather than simultaneous weight updates. The algorithm employs linearization of hidden node sigmoidal activation functions combined with penalty terms to compensate for linearization errors, enabling analytical solution of weight optimization through linear equation systems. This approach eliminates user-adjustable parameters except network structure, addressing fundamental limitations of gradient-based methods requiring careful parameter tuning.

Experimental validation using Lorenz and Mackey-Glass time series prediction demonstrates extraordinary performance improvements. OLL-learning achieved normalized root mean square error

of -48.9 dB (0.0036) for Lorenz prediction, representing nearly an order of magnitude improvement over existing methods. Convergence analysis shows 25x speedup over Fletcher-Reeves Conjugate Gradient optimization after 10^5 iterations and 420x speedup over Bold Driver method after 10^6 iterations. For 8:8:8:3 MLP configurations, OLL-learning reached target accuracy in 2 iterations compared to 500 iterations for BV2 and 2000 iterations for standard backpropagation [4].

3.5. An Accelerated Learning Algorithm for Multilayer Perceptron Networks (ABP)

Parlos et al. present the Accelerated Backpropagation (ABP) algorithm that incorporates the ratio of current and previous error gradients in weight update mechanisms to address slow convergence and local minima entrapment in standard backpropagation. The algorithm modifies gradient descent by scaling weight updates according to successive gradient computations, providing adaptive step sizing responsive to error surface topology. ABP specifically targets analog function approximation problems where it demonstrates superior performance compared to binary classification tasks.

Experimental validation across multiple benchmark problems shows dramatic performance improvements. For sinusoidal function approximation using 1-15-1 networks, ABP achieved convergence in 968 iterations compared to 25,844 iterations for batch backpropagation and 7,336 iterations for individual backpropagation. Two-dimensional function approximation results using 2-10-1 networks demonstrate ABP requiring only 516 iterations compared to 6,713 iterations for batch backpropagation, establishing consistent performance advantages across different problem complexities while maintaining comparable final accuracy [5].

TABLE I
SINUSOIDAL FUNCTION APPROXIMATION USING A 1–15–1 NETWORK

Update rule	Trials	η	Converged trials	Average of converged trials	Standard deviation
BP (batch)	10	0.0028	10	25844	4217
BP (ind.)	10	0.011	10	7336	1461
QP ($\mu = 1.7$)	10	0.009	10	2521	2011
ABP	10	0.05	10	968	363

TABLE III
A SIMPLE TWO-DIMENSIONAL FUNCTION APPROXIMATION USING A 2–10–1 NETWORK

Update rule	Trials	η	Converged trials	Average of converged trials	Standard deviation
BP (batch)	10	0.0015	10	6713	1833
BP (ind.)	10	0.011	10	2225	2052
QP ($\mu = 1.5$)	10	0.008	10	3424	2760
ABP	10	0.011	10	516	223

Figure 5. Comparison between training algorithms required iterations to target RMS convergence [5]

3.6. FPGA acceleration on a multi-layer perceptron neural network for digit recognition

This research implements a scalable multi-layer perceptron neural network on FPGA for handwritten digit recognition applications. The methodology employs an eight-stage pipelined architecture deployed on Xilinx Ultrascale FPGA hardware, integrating custom RTL controller design with multiple Vivado intellectual property blocks. The implementation features a single-hidden-layer MLP structure containing 12 neurons and 9,550 weight and bias parameters, utilizing the MNIST database for training and validation. The design emphasizes logic-only implementation with off-board training parameters to minimize complexity and optimize inference performance, while maintaining scalability for different accuracy constraints and hardware specifications.

The FPGA implementation achieves remarkable performance metrics, delivering 1.55 microseconds inference latency per digit recognition with 93.25% classification accuracy. The system demonstrates significant computational acceleration, providing 40.4× speedup over the fastest software implementation and **127.2x** speedup over the longest software execution when compared to equivalent MATLAB implementations running on Intel i7-7500U processors. Resource utilization encompasses 44,668 LUTs and 14,274 flip-flops, with energy consumption totaling 0.88 μJ comprising 99% dynamic and 1% static power consumption. The architecture achieves superior performance compared to existing works while maintaining comparable hardware resource requirements [6].

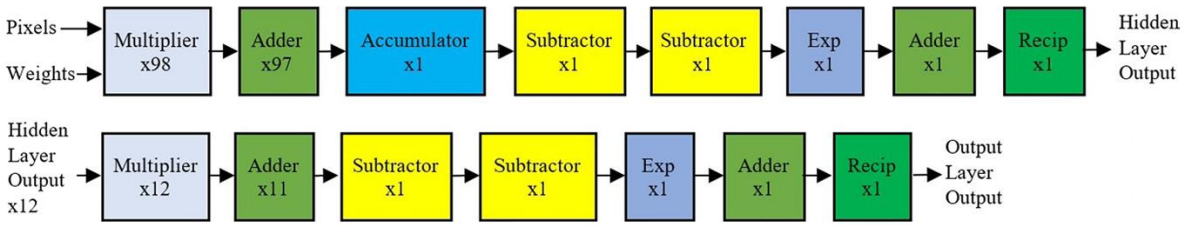


Figure 6. Application design pipeline [6]

3.7. Parallelizing Backpropagation Neural Network Using MapReduce and Cascading Model

Liu, Jing, and Xu implement a parallelized backpropagation approach based on MapReduce computing model integrated with Hadoop distributed framework. The research introduces a cascading classification system that processes training data subsets in parallel while maintaining accuracy through ensemble decision-making mechanisms. The implementation leverages MapReduce's fault tolerance, data replication, and load balancing capabilities, with multiple classifier groups trained on different data classes using majority voting for result refinement.

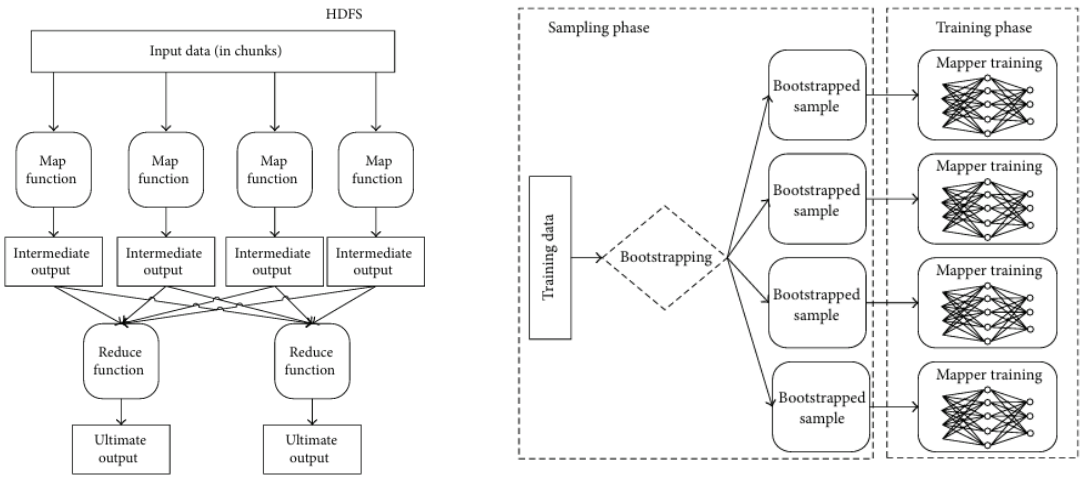


Figure 7. MapReduce model and parallel training model [7]

Experimental validation using Iris, Wine, and Delta elevators datasets demonstrates significant efficiency improvements for large-scale datasets while maintaining classification accuracy comparable to sequential implementations. The distributed approach successfully processes datasets exceeding single-machine memory constraints, enabling enterprise-level data volume handling. Performance

analysis shows the MapReduce implementation scales effectively across multiple processing nodes, with individual mappers handling specific data subsets and reducers aggregating results to produce final classifications with maintained algorithmic robustness [7].

3.8. Accelerating Neural Network Training Process on Multi-Core Machine Using OpenMP

Mohammed, Paznikov, and Gorlatch develop an OpenMP-based parallelization approach for neural network training acceleration on multi-core CPU architectures, implementing parallel mini-batch training with intelligent subset exclusion based on gradient convergence criteria. The methodology distributes training examples across multiple threads while maintaining learning quality through selective workload distribution and synchronization mechanisms. The approach addresses computational bottlenecks in sequential training while providing superior accessibility compared to GPU-based solutions.

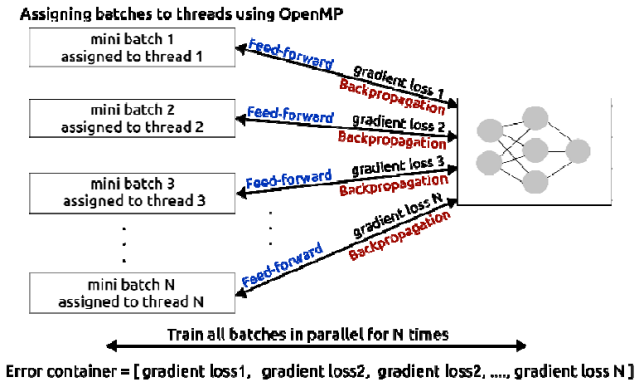


Figure 8. Training OpenMP thread assignment [8]

Experimental evaluation using banknote authentication dataset on 12-core machines demonstrates significant acceleration compared to sequential training, with performance scaling correlating with available core count for sufficiently large datasets. The implementation achieves near-linear speedup for large datasets with proper thread management, though smaller datasets experience minimal benefits due to synchronization overhead. Results establish optimal thread utilization strategies based on dataset characteristics, making parallel neural network training accessible through standard multi-core processors commonly available in contemporary computing environments [8].

3.9. OpenCL Optimization for FPGA Deep Learning Application

This research develops FPGA acceleration for convolutional neural network operations using OpenCL programming frameworks, focusing on energy efficiency and performance optimization for deep learning inference. The methodology encompasses computational model development, memory access pattern optimization, and parallel processing architecture design tailored to FPGA reconfigurable hardware capabilities. The approach leverages FPGA's inherent parallelism and energy efficiency advantages over traditional GPU implementations while maintaining computational throughput.

Performance evaluation demonstrates **9.76x** speedup compared to single-threaded CPU execution and **2.8x** improvement over 16-threaded CPU implementations. Comprehensive analysis across eight convolution layer configurations shows consistent improvements ranging from **8.9x** to **40.1x** speedup compared to Xilinx reference programs. Energy consumption analysis reveals FPGA

implementations achieving 134-233 pictures per second processing rates with only 25 watts power consumption, contrasting favorably with GPU implementations requiring 235 watts for 500-824 pictures per second, establishing FPGA as a viable energy-efficient alternative for deep learning applications. “This computational model is used to help software programmers without fundamental hardware knowledge for a quick implementation in deep learning algorithm with high performance using FPGA” [9].

3.10. Parallel Backpropagation Neural Network Training Techniques using Graphics Processing Unit

Arslan et al. implement GPU-based parallelization strategies for backpropagation training using CUDA architecture, leveraging massive parallel processing capabilities through thousands of concurrent threads. The research focuses on data parallelism where training examples are distributed across GPU cores, enabling simultaneous processing of multiple patterns within training iterations.

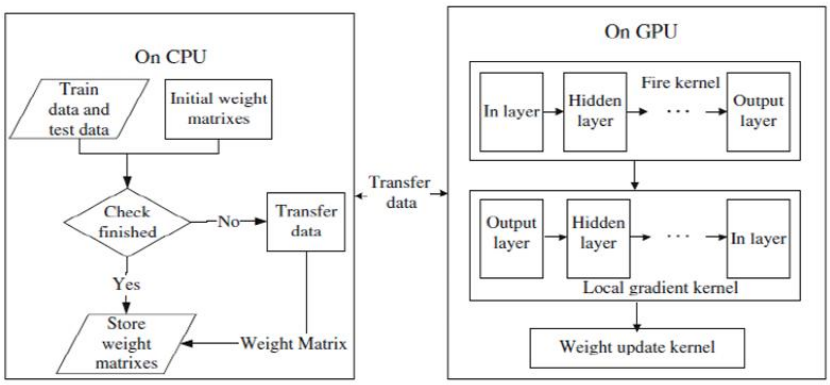


Figure 9. CPU and GPU workload distribution [10]

The implementation addresses both forward propagation and backward error propagation phases, with careful attention to memory management and thread organization for optimal GPU utilization. Performance evaluation is not demonstrated in a specific testing environment, but the paper concludes that a highly parallelized CUDA training architecture would benefit large datasets, whereas smaller datasets would be more suitable for a CPU approach.

3.11. Hardware Accelerated Neural Networks (Thesis)

This thesis investigates parallelization techniques for accelerating backpropagation neural network training through GPU-based CUDA framework implementation. The methodology combines dataset parallelization with atomic operations parallelization to exploit the massive parallel processing capabilities of graphics processing units. The research utilizes the Tesla C2050 multi-core platform and employs the MNIST handwritten digit database containing 10,000 patterns for performance evaluation. The implementation focuses on optimizing the computational bottlenecks inherent in traditional sequential backpropagation algorithms, particularly addressing the time-intensive nature of training processes for pattern recognition applications.

The parallel implementation demonstrates substantial performance improvements, reducing training epoch duration from 3,450 milliseconds in sequential processing to 630 milliseconds using the parallel algorithm, representing an approximate **5.5x** acceleration factor. The complete training process achieves error minimization to 0.04 within 167 seconds across 265 epochs, utilizing a network

configuration of 784-25-15-10 neurons with a final convergence error of 0.0399. The parallelization approach proves effective for large-scale dataset processing while maintaining training accuracy, establishing GPU-based acceleration as a viable and cost-effective solution for neural network applications compared to traditional CPU cluster implementations.

4. Selected application

MLPs first gained significant traction in the 1980s during the revival of neural networks, when backpropagation algorithms made training of deep architectures computationally feasible. Despite their established theoretical foundation, I decided to implement my own naive single-threaded C++ version to gain deeper understanding of the underlying computational mechanics. This implementation serves as a baseline for exploring various optimization strategies, from architectural improvements to parallelization techniques. The complete implementation is available at https://github.com/Blinduzzi/pao-inference-mlp, which demonstrates the evolution from basic single-threaded computation through multiple build iterations with progressive optimizations.

While MNIST remains a foundational benchmark dataset, its simplicity has led to near-human performance levels, with convolutional networks achieving accuracies exceeding 99.7%. To address this limitation, I selected Fashion-MNIST, introduced by Zalando in 2017 as a direct replacement for MNIST classification tasks [15]. Fashion-MNIST (F-MNIST) has been extensively studied, with state-of-the-art methods like PreAct-ResNet18 + FMix achieving benchmark performance, while recent CNN-3-128 models with data augmentation have reached accuracies of 99.65% [13]. As the GitHub repo specifies, “Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. We intend Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms” [14][15].

The target was to start off with an implementation with an intensive computational load (number of neurons and layers), but also that performs good on the F-MNIST dataset. Although the goal of this project was not to get optimal results in terms of model accuracy, it is desirable to optimize code that performs well on the target task. The optimal architecture configuration that I determined was initially inspired by [11] and implements a four-layer network structure: 784 input neurons (corresponding to 28×28 pixel F-MNIST images), two hidden layers with 128 and 64 neurons respectively using ReLU activation functions, and a 10-neuron output layer with softmax activation for multi-class classification. The model was trained on 5,000 samples with a 500-sample test set, employing a learning rate of 0.01 and achieving a final accuracy of **87%**. This architecture represents a balance between computational intensity and classification performance, providing sufficient model complexity to learn discriminative features while maintaining reasonable training time for iterative optimization experiments.

5. Single-threaded naive implementation

The initial single threaded naïve implementation of the algorithm is available in the project repository as ***naive_main.cpp***.

5.1. Initializations

The implementation begins by loading the Fashion-MNIST dataset, which contains 28x28 grayscale images of clothing items across 10 categories (T-shirt, Trouser, Pullover, etc.). The data loading functions handle the binary format.

The network follows a standard feedforward architecture: {784, 128, 64, 10} - an input layer receiving the 784 pixel values, two hidden layers with 128 and 64 neurons respectively using ReLU activation, and an output layer with 10 neurons using softmax for classification. The architecture is initialized in the MLP class constructor. The weights use Xavier initialization to maintain proper variance across layers, preventing vanishing or exploding gradients [16]:

```
// Xavier initialization
    void initialize_weights() {
        std::uniform_real_distribution<double> dist(-1.0, 1.0);

        for (size_t i = 0; i < weights.size(); ++i) {
            double xavier_scale = std::sqrt(6.0 / (layer_sizes[i] +
layer_sizes[i + 1]));

            for (size_t j = 0; j < weights[i].size(); ++j) {
                for (size_t k = 0; k < weights[i][j].size(); ++k) {
                    weights[i][j][k] = dist(rng) * xavier_scale;
                }
            }

            // Initialize biases to small random values
            for (size_t j = 0; j < biases[i].size(); ++j) {
                biases[i][j] = dist(rng) * 0.01;
            }
        }
    }
```

Figure 10. Xavier initialization implementation [16]

5.2. Training

The train function orchestrates the entire training process with a stochastic approach. Note that the live validation of the dataset was commented out, as it was used for the initial architecture development, but is not relevant for timing measurements going forward:

```
void train(const std::vector<std::vector<double>>& train_data,
               const std::vector<int>& train_labels,
               const std::vector<std::vector<double>>& test_data,
               const std::vector<int>& test_labels,
               int epochs) {

        std::vector<int> indices(train_data.size());
        std::iota(indices.begin(), indices.end(), 0);
```

```
        for (int epoch = 0; epoch < epochs; ++epoch) {
            // Shuffle data for each epoch
            std::shuffle(indices.begin(), indices.end(), rng);

            double total_loss = 0.0;

            // Sequential training - process each sample individually
            for (size_t i = 0; i < train_data.size(); ++i) {
                int idx = indices[i];

                // Create one-hot encoded target
                std::vector<double> target(10, 0.0);
                target[train_labels[idx]] = 1.0;

                // Forward and backward pass for each sample
                std::vector<double> output = forward(train_data[idx]);
                total_loss += calculate_loss(output, target);
                backward(target);
            }

            // // Calculate accuracy on test set every 5 epochs -- Used in
finding optimal architectures
            // if ((epoch + 1) % 5 == 0) {
            //     int correct = 0;
            //     for (size_t i = 0; i < test_data.size(); ++i) {
            //         if (predict(test_data[i]) == test_labels[i]) {
            //             correct++;
            //         }
            //     }
            //
            //     double accuracy = 100.0 * correct / test_data.size();
            //     double avg_loss = total_loss / train_data.size();
            //
            //     std::cout << "Epoch " << std::setw(3) << epoch + 1
            //               << " | Loss: " << std::fixed <<
std::setprecision(4) << avg_loss
            //               << " | Test Accuracy: " << std::setprecision(2) <<
accuracy << "%" << std::endl;
            // }
        }
    }
};
```

Figure 11. Naive train function

The training uses **stochastic gradient descent**, processing one sample at a time with data shuffling each epoch to prevent overfitting and ensure better convergence. Each label is converted to one-hot encoding (e.g., label 3 becomes [0,0,0,1,0,0,0,0,0,0]).

The forward function implements the feed-forward pass through the network:

```
std::vector<double> forward(const std::vector<double>& input) {
        // Set input layer
        activations[0] = input;

        // Forward propagation through hidden layers
        for (size_t layer = 0; layer < weights.size(); ++layer) {
            for (size_t neuron = 0; neuron < weights[layer].size(); ++neuron)
{
                double sum = biases[layer][neuron];

                for (size_t prev_neuron = 0; prev_neuron <
weights[layer][neuron].size(); ++prev_neuron) {
                    sum += weights[layer][neuron][prev_neuron] *
activations[layer][prev_neuron];
                }

                z_values[layer][neuron] = sum;

                // Apply activation function
                if (layer == weights.size() - 1) {
                    // Output layer - softmax
                    activations[layer + 1][neuron] = sum;
                } else {
                    // Hidden layers - use ReLU
                    activations[layer + 1][neuron] = relu(sum);
                }
            }
        }

        // Apply softmax to output layer
        activations.back() = softmax(activations.back());
        return activations.back();
    }
```

Figure 12. Naive forward pass function

For each layer, it computes weighted sums (z-values), applies ReLU activation for hidden layers, and uses softmax for the output layer. The softmax ensures output values are equal to class probabilities, and therefore sum to 1.

The backward function implements backpropagation using the calculus chain rule:

```
void backward(const std::vector<double>& target) {
        int num_layers = layer_sizes.size();
        std::vector<std::vector<double>> deltas(num_layers - 1);

        // Initialize deltas
```

```
        for (int i = 0; i < num_layers - 1; ++i) {
            deltas[i].resize(layer_sizes[i + 1]);
        }

        // Calculate output layer deltas (cross-entropy loss with softmax)
        for (size_t i = 0; i < activations.back().size(); ++i) {
            deltas.back()[i] = activations.back()[i] - target[i];
        }

        // Backpropagate deltas through hidden layers
        for (int layer = num_layers - 3; layer >= 0; --layer) {
            for (int neuron = 0; neuron < layer_sizes[layer + 1]; ++neuron) {
                double error = 0.0;

                for (int next_neuron = 0; next_neuron < layer_sizes[layer +
2]; ++next_neuron) {
                    error += deltas[layer + 1][next_neuron] * weights[layer +
1][next_neuron][neuron];
                }

                deltas[layer][neuron] = error *
relu_derivative(z_values[layer][neuron]);
            }
        }

        // Update weights and biases
        for (size_t layer = 0; layer < weights.size(); ++layer) {
            for (size_t neuron = 0; neuron < weights[layer].size(); ++neuron)
{
                // Update bias
                biases[layer][neuron] -= learning_rate *
deltas[layer][neuron];

                // Update weights
                for (size_t prev_neuron = 0; prev_neuron <
weights[layer][neuron].size(); ++prev_neuron) {
                    weights[layer][neuron][prev_neuron] -=
                        learning_rate * deltas[layer][neuron] *
activations[layer][prev_neuron];
                }
            }
        }
    }
```

Figure 13. Naive backpropagation function

The calculate_loss function implements cross-entropy loss, which penalizes confident wrong predictions more heavily and works well with softmax outputs for classification tasks.

The predict function handles inference. It runs forward propagation and returns the index of the neuron with highest activation (maximum probability), corresponding to the predicted class. This is used during testing to evaluate model accuracy by comparing predicted labels with ground truth labels:

```
int predict(const std::vector<double>& input) {
        std::vector<double> output = forward(input);
        return std::max_element(output.begin(), output.end()) -
output.begin();
    }
```

Figure 14. Inference function used for testing

5.3. Performance Summary

This single-threaded naive MLP implementation demonstrates reasonable but computationally intensive performance on the Fashion-MNIST classification task. It is important to note that the compilation flags used implied no optimizations (**-o0**). Training on 5000 images for 50 epochs with a {784, 128, 64, 10} architecture on an **Intel Core i5-9300H CPU @ 2.40GHz** required **937.104 seconds** (approximately **15.6 minutes**), averaging **18.742 seconds per epoch**, with an **87%** prediction accuracy. The sequential processing of individual samples, combined with the lack of vectorization or parallelization optimizations, results in significant computational overhead - each training sample requires separate forward and backward passes through all network layers.

6. Optimization limitations and opportunities

6.1. Theoretical performance limits

CPU Performance Limits (Intel Core i5-9300H):

- Processing cores: 4 cores, 8 threads (Hyper-Threading enabled)
- Clock speeds: Base: 2.4 GHz, Turbo boost: up to 4.1 GHz
- Cache hierarchy: L1: 4x32KB instruction + 4x32KB data, L2: 4x256KB, L3: 8MB shared
- Architecture: Coffee Lake, 14nm++ process
- TDP: 45W

Memory Performance Limits:

- Total capacity: 16GB (2x 8GB dual-channel configuration)
- Memory type: **DDR4-2667**
- Theoretical bandwidth: 21.3 GB/s per channel = 42.6 GB/s total in dual-channel mode
- Memory controller support: Up to DDR4-2666 natively supported

GPU Performance Limits (NVIDIA GeForce GTX 1650):

- Number of cores: 896
- GPU clocks: Base: 1485 MHz, Boost: 1665 MHz
- GPU memory: 4GB GDDR5
- Memory bandwidth: 128 GB/s

- Memory interface: 128-bit
- Power consumption: 75W maximum

Optimization Strategy Theoretical Limits:

- **Multi-threading**: Up to **8** concurrent threads maximum
- **SIMD/AVX:** Intel CPUs support **AVX2** instruction sets for parallel operations

Architecture: {784, 128, 64, 10}

Training Configuration: 5,000 samples × 50 epochs = 250,000 total training iterations

Forward Pass:

- Layer 1: 784 × 128 = 100,352 multiply-adds + 128 bias + 128 ReLU ≈ **100,608 ops**
- Layer 2: 128 × 64 = 8,192 multiply-adds + 64 bias + 64 ReLU ≈ **8,320 ops**
- Layer 3: 64 × 10 = 640 multiply-adds + 10 bias + 10 softmax ≈ **660 ops**
- **Forward Total: ~109,588 ops**

Backward Pass:

- Output deltas: 10 ops
- Layer 2 deltas: 64 × 10 + 64 ReLU derivatives ≈ **704 ops**
- Layer 1 deltas: 128 × 64 + 128 ReLU derivatives ≈ **8,320 ops**
- Weight updates: (784×128 + 128) + (128×64 + 64) + (64×10 + 10) ≈ **109,386 ops**
- **Backward Total: ~118,420 ops**

Total per sample: 228,008 operations Total for 50 epochs: 250,000 × 228,008 = 57.002 GOPS

1. **CPU 8-Thread with AVX:** 8 threads × 4.1 GHz × 8 FMA (fused multiply-add) operations (AVX2) = **262.4 GFLOPS peak =>** 57.002 / 262.4 = **0.217 seconds**.
2. **GPU Implementation:** 896 cores × 1.665 GHz × 2 ops/cycle **= 2982 GFLOPS peak;** Input data: 5,000 × 784 × 4 bytes = 15.68 MB, model weights: ~109,386 parameters × 4 bytes = 0.44 MB => total transfer**:** ~16.12 MB × 2 (to/from GPU**) = 32.24 MB** @ PCIe 3.0 x16 (12 GB/s): 32.24 MB ÷ 12 GB/s = **0.0027 seconds =>** 57.002 / 2982 + 0.0027 = **0.0218 seconds.**

6.2. CPU optimized

6.2.1. Compilation flags

This optimization iteration introduces no code changes in relation to the naive single-threaded CPU implementation. All other iterations will be stacked on top of the changes of the previous iteration. The only change is that instead of the -o0 (no compilation optimizations) flag, -o3 was used (maximum compiler optimizations enabled for improved performance).

```
set (CMAKE_CXX_FLAGS "-o3")
```

Figure 15. Setting compilation flag for maximum compiler optimizations

The compiler optimized code required **928.242 seconds** (**15.5 minutes**) to complete 50 epochs of training on 5000 Fashion-MNIST images using an Intel Core i5-9300H CPU, attaining an **86.5%** prediction accuracy. At **18.565 seconds per epoch**, this performance demonstrates that the compiler does not notice any glaring code optimization, and the bottlenecks remain evident: costly sequential approach, with intensive forward pass and backpropagation.

6.2.2. OpenMP

The OpenMP optimized implementation introduces several key improvements over the previous approach to achieve efficient parallel neural network training. First, the training process was restructured to use mini-batch processing with 8 OpenMP threads, where threads are utilized for parallel forward/backward propagation computations, gradient accumulation within batches, and final model evaluation across test samples. To eliminate segmentation faults and race conditions, each thread operates on separate local variables including independent copies of activations, z-values, and gradient containers, ensuring no shared memory conflicts during computation. Finally, the weight update strategy follows a master-slave pattern where slave threads independently compute gradients on their assigned mini-batch samples, and the master thread averages all thread-specific weight and bias modifications before applying the final updates to the network architecture (inspired by the training architecture in [8]), ensuring stable convergence while maximizing parallel efficiency.

The OpenMP optimized code required **86.529 seconds** (**1.4 minutes**) to complete 50 epochs of training on 5000 Fashion-MNIST images using 8 parallel threads. At **1.731 seconds per epoch**, this performance demonstrates a significant speedup over sequential implementations, achieving **85.60% test accuracy** while effectively utilizing multi-core parallelization through separated thread computation, mini-batch processing, and averaged weight updates to eliminate segmentation faults and race conditions.

6.2.3. AVX2

The AVX2 optimized version introduces three fundamental optimizations over the previous OpenMP implementation. First, it replaces the memory-inefficient thread update container *std::vector<std::vector<std::vector<std::vector<double>>>>* with aligned memory structures (**AlignedMatrix** and **AlignedVector**) that use 32-byte aligned _mm_malloc for optimal cache performance and AVX2 compatibility. The implementation adds AVX2 SIMD intrinsics including **_mm256_fmadd_pd** for **fused multiply-add** (FMA) operations, **_mm256_load_pd/store_pd** for vectorized memory access, and **_mm256_max_pd** for parallel ReLU computations, processing 4 double-precision values simultaneously. Additionally, it consolidates the complex per-thread gradient storage into a clean ThreadLocalData structure, eliminating the scattered memory allocations and pointer chasing that plagued the original nested vector approach.

The AVX2 optimized code required **30.322 seconds** (**0.5 minutes**) to complete 50 epochs of training on 5000 Fashion-MNIST images using 8 parallel threads with SIMD acceleration. At **0.606 seconds per epoch**, this performance demonstrates a **2.85x speedup** over the OpenMP-only implementation, achieving **85.80% test accuracy** while effectively utilizing both multi-core parallelization and AVX2 vectorization through aligned memory structures, fused multiply-add operations, and 4-wide SIMD processing that combines multiplication and addition operations for optimal cache performance and computational efficiency.

6.3. GPU optimized - OpenCL

The GPU OpenCL implementation demonstrates an effective approach at neural network training acceleration through five specialized OpenCL kernels that collectively orchestrate the complete training pipeline. The **forward_pass** kernel executes dense layer matrix multiplication [17] operations using a 1D work-item arrangement where each work-item computes a single output activation, with work-item assignment calculated as batch_idx * output_size + neuron_idx. The **relu_activation** kernel applies elementwise ReLU transformations using simple 1D indexing for optimal memory access patterns. The **softmax_activation** kernel implements numerically stable softmax computation utilizing work-groups with local memory for reduction operations to compute maximum and sum values efficiently. The **compute_deltas** kernel handles backpropagation error computation using a 2D NDRange organization (batch_size × layer_size) where get_global_id(0) represents the batch index and get_global_id(1) represents the neuron index, enabling direct mapping between work-items and computation elements. The **update_weights** kernel performs gradient accumulation and parameter updates using a 2D work-item arrangement mapping output neurons to the first dimension and input neurons to the second dimension.

```
// Update weights for all layers
        for (int layer = 0; layer < layer_sizes.size() - 1; layer++) {
            update_weights_kernel.setArg(0, weight_buffers[layer]);
            update_weights_kernel.setArg(1, bias_buffers[layer]);
            update_weights_kernel.setArg(2, activation_buffers[layer]);
            update_weights_kernel.setArg(3, delta_buffers[layer + 1]);
            update_weights_kernel.setArg(4, learning_rate);
            update_weights_kernel.setArg(5, (int)batch_size);
            update_weights_kernel.setArg(6, layer_sizes[layer]);
            update_weights_kernel.setArg(7, layer_sizes[layer + 1]);

            size_t weight_global_size[2] = {static_cast<size_t>(layer_sizes[layer
+ 1]),
                                            static_cast<size_t>(layer_sizes[layer])}
;
            err = queue.enqueueNDRangeKernel(update_weights_kernel, cl::NullRange,
                                             cl::NDRange(weight_global_size[0],
weight_global_size[1]), cl::NullRange);
            cl_check(err);
        }
```

Figure 16. Updating weights kernel enqueueing

All network parameters including weight matrices, bias vectors, activation buffers, gradient arrays, and delta computation buffers are allocated in GPU global memory and persist throughout the training process. Most kernels use 1D NDRange configurations with global work sizes calculated as the product of relevant dimensions (e.g., batch_size × layer_size for activation functions). The forward pass kernel creates batch_size × output_neurons work-items arranged in a single dimension, where each work-item independently computes one neuron activation. The softmax kernel employs work-groups with local work sizes dynamically adjusted based on device capabilities, typically capped at 64 work-items per group. The backward propagation and weight update kernels utilize 2D NDRange

configurations that directly map batch samples and neuron indices to the first and second dimensions respectively, enabling natural indexing patterns that align with the mathematical structure of backpropagation computations.

The implementation incorporates compiler optimizations including fast relaxed math operations (-cl-fast-relaxed-math), multiply-add enablement (-cl-mad-enable), and unsafe math optimizations that enhance floating-point throughput on GPU architectures. The system completes 50 training epochs on 5,000 MNIST samples in **4.207 seconds** on an NVIDIA GeForce GTX 1650, averaging **0.084 seconds** per epoch while achieving **86.40%** test accuracy. Performance optimizations include strategic use of local memory for reduction operations in softmax computations, memory access patterns designed for coalescing, and elimination of unnecessary CPU-GPU synchronization points through persistent GPU-resident data structures.

7. Results

	Train time (s)	Train time / epoch (s)	Predict accuracy	**Speedup**
Naive	937.104	18.742	87%	**x1**
O3	928.242	18.565	86.5%	**x1.009**
OpenMP	86.529	1.731	85.6%	**x10.829**
AVX2	30.322	0.606	85.8%	**x30.905**
OpenCL	4.207	0.084	86.4%	**x222.748**
*Theoretical CPU	*0.217	*0.004	-	***x4318**
*Theoretical GPU	*0.022	*0.0004	-	***x42595**

*- these values are only theoretical estimates

Figure 17. My own optimized iterations performance results

Paper & Platform	Train time	Baseline	**Speedup vs Baseline**
[3] Multicore CPU (8-Core)	47.041-117.628 s	MATLAB single-core implementation	**x1.12-4.16**
[3] GPU (48 cores) - MATLAB	386-378 s	MATLAB single-core implementation	**x304.6**
[6] FPGA vs MATLAB	1.55 μs per image	MATLAB software implementation	**x40.4-127.2**
[8] OpenMP (10 threads)	28-35ms	Existing optimized mini-batch training approach	**x1.25**
[8] OpenMP (12 threads)	18-29ms	Existing optimized mini-batch training approach	**x1.33-1.58**
[9] OpenCL FPGA	9.76ms	Intel Xeon 2.20GHz with -O3 & 16 threads	**x9.7**
[9] OpenCL vs Xilinx baseline	9.76ms	Intel Xeon 2.20GHz with -O3 & 16 threads	**x8-40**
[10] Single GPU CUDA	-	CUBLAS optimized matrix operations	**x11.99**
[10] Multiple GPUs CUDA	-	CUBLAS optimized matrix operations	**x51.33**
[10] GPU - Cancer dataset	-	CUBLAS optimized matrix operations	**x46**
[10] GPU - Mushroom dataset	-	CUBLAS optimized matrix operations	**x63**

[10] GPU - Multiple Networks	-	CUBLAS optimized matrix operations	**x496**
[11] Tesla C2050 GPU	167s (265 epochs)	FANN (Fast Artificial Neural Network Library)	**x5.47**

Figure 18. SOTA paper relative comparison

8. Conclusions

The experimental results demonstrate a clear hierarchical progression in computational acceleration capabilities, with each optimization technique building upon fundamental principles of parallel computing architecture. While compiler-level optimizations (O3) yielded negligible performance gains (1.009× speedup), the transition to explicit parallelization strategies revealed substantial computational benefits. OpenMP implementation achieved a 10.829× speedup through multicore CPU utilization, while AVX2 vectorization leveraged SIMD instruction sets to attain a 30.905× performance improvement. The OpenCL implementation represents the pinnacle of this optimization hierarchy, delivering a remarkable 222.748× speedup through GPU acceleration while maintaining competitive classification accuracy at 86.4%. Notably, the accuracy degradation across optimization levels remained minimal (87% to 85.6%).

The convergence of performance metrics across different computational paradigms underscores the critical importance of hardware-software co-design in modern machine learning applications. The OpenCL implementation's exceptional performance gain validates the effectiveness of massively parallel architectures for neural network training, particularly when computational workloads can be decomposed into independent, data-parallel operations. The sustained accuracy levels across all optimization approaches confirm that the Fashion-MNIST classification task maintains its inherent complexity regardless of the underlying computational framework. These findings establish a compelling case for GPU-accelerated neural network training in production environments, where the 222× reduction in training time from 937 seconds to 4.2 seconds represents a transformative improvement in computational efficiency that directly translates to enhanced development cycles and reduced infrastructure costs.

9. References

[1] M. K. Alsmadi, K. B. Omar, S. A. Noah, and I. Almarashdah, "*Performance Comparison of Multi-layer Perceptron (Back Propagation, Delta Rule and Perceptron) algorithms in Neural Networks*," in *2009 IEEE International Advance Computing Conference (IACC 2009)*, Patiala, India, Mar. 6-7, 2009, pp. 296-299. DOI: 978-1-4244-1888-6/08. Available: https://ieeexplore.ieee.org/document/4809024. [Accessed: June 14, 2025].

[2] D. S. M. Carvalho and A. A. Neto, "*Acceleration of Backpropagation Training with Selective Momentum Term*," in *Proceedings of the 11th International Conference on Agents and Artificial Intelligence (ICAART 2019)*. DOI: 10.5220/0007272004430450. Available: https://www.scitepress.org/Papers/2019/72720/72720.pdf. [Accessed: June 14, 2025].

[3] S. A. Dawwd and N. M. AL Layla, "*Training Acceleration of Multi-Layer Perceptron using Multicore CPU and GPU under MATLAB Environment*," College of Engineering, University of Mosul, Mosul-Iraq, 2015. Available: https://scispace.com/pdf/training-acceleration-of-multi-layer-perceptron-using-yoqe1s0f2d.pdf. [Accessed: June 14, 2025].

[4] S. Ergezinger and E. Thomsen, "*An Accelerated Learning Algorithm for Multilayer Perceptrons: Optimization Layer by Layer*," *IEEE Transactions on Neural Networks*, vol. 6, no. 1, pp. 31-42, Jan. 1995. DOI: 10.1109/72.363440. Available: https://ieeexplore.ieee.org/abstract/document/363452. [Accessed: June 14, 2025].

[5] A. G. Parlos, B. Fernandez, A. F. Atiya, J. Muthusami, and W. K. Tsai, "*An Accelerated Learning Algorithm for Multilayer Perceptron Networks*," *IEEE Transactions on Neural Networks*, vol. 5, no. 3, pp. 493-497, May 1994. DOI: 10.1109/72.286917. Available: https://ieeexplore.ieee.org/document/286921. [Accessed: June 14, 2025].

[6] Isaac Westby, Xiaokun Yang, Tao Liu, and Hailu Xu, "*FPGA acceleration on a multi-layer perceptron neural network for digit recognition*," *The Journal of Supercomputing*, vol. 77, pp. 14356-14373, 2021. DOI: 10.1007/s11227-021-03849-7. Available: https://link.springer.com/article/10.1007/s11227-021-03849-7. [Accessed: June 15, 2025].

[7] Y. Liu, W. Jing, and L. Xu, "*Parallelizing Backpropagation Neural Network Using MapReduce and Cascading Model*," *Computational Intelligence and Neuroscience*, vol. 2016, Article ID 2584570, 11 pages, 2016. DOI: 10.1155/2016/2584570. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC4863083/. [Accessed: June 15, 2025].

[8] O. T. Mohammed, A. A. Paznikov and S. Gorlatch, "*Accelerating Neural Network Training Process on Multi-Core Machine Using OpenMP,*" *2022 III International Conference on Neural Networks and Neurotechnologies (NeuroNT)*, Saint Petersburg, Russian Federation, 2022, pp. 7-11, DOI: 10.1109/NeuroNT55429.2022.9805549. Available: https://ieeexplore.ieee.org/document/9805549. [Accessed: June 15, 2025].

[9] Zhang S, Wu Y, Men C, He H, Liang K (2019) "*Research on OpenCL optimization for FPGA deep learning application*". PLoS ONE 14(10): e0222984. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0222984. [Accessed: June 15, 2025].

[10] Amin, Muhammad & Kashif, Muhammad & Sarwar, Muhammad & Rehman, Abdur & Waheed, Fiaz & Rehman, Haseeb. (2019). "*Parallel Backpropagation Neural Network Training Techniques using Graphics Processing Unit*". *International Journal of Advanced Computer Science and Applications*. 10. 10.14569/IJACSA.2019.0100270. Available: https://www.researchgate.net/publication/331474278_Parallel_Backpropagation_Neural_Network_Training_Techniques_using_Graphics_Processing_Unit. [Accessed: June 14, 2025].

[11] S. Toman, "*Hardware Accelerated Neural Networks,*" M.Sc. thesis, Faculty of Electrical Engineering and Computer Science, VŠB – Technical University of Ostrava, Ostrava, Czech Republic, 2011. Available: https://dspace.vsb.cz/handle/10084/87126. [Accessed: June 15, 2025].

[12] GeeksforGeeks. "Backpropagation in Neural Network." *GeeksforGeeks*. Available: https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-network/. [Accessed: June 15, 2025].

[13] Elhoseny, M., & Shankar, K. (2024). "Design and Analysis of Deep Learning Models for Multi-Class Classification in the IoMT Environment." *Mathematics*, vol. 12, no. 20, art. 3174. Available: https://www.mdpi.com/2227-7390/12/20/3174. [Accessed: June 16, 2025].

[14] LeCun, Y. "The MNIST Database of Handwritten Digits." *Yann LeCun's Website*. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: June 16, 2025].

[15] Zalando Research. "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms." *GitHub Repository*. Available: https://github.com/zalandoresearch/fashion-mnist. [Accessed: June 16, 2025].

[16] GeeksforGeeks. "Xavier Initialization in Deep Learning." *GeeksforGeeks*. Available: https://www.geeksforgeeks.org/deep-learning/xavier-initialization/. [Accessed: June 16, 2025].

[17] A. Jaokar. "Explaining Multilayer Perceptrons in Terms of General Matrix Multiplication." *LinkedIn*. Available: https://www.linkedin.com/pulse/explaining-multilayer-perceptrons-terms-general-matrix-ajit-jaokar-c5aje. [Accessed: June 17, 2025].