Exploring Parallelization of Shallow Neural Network using Cuda

Students BROUTHEN Kamel & AKEB Abdelaziz
Professor HAICHOUR Amina Selma
Lab Parallel Programming using Cuda

1 Introduction

In this work, we focus on parallelizing the execution of a shallow neural network (SNN) with a single hidden layer, emphasizing the parallelization of matrix multiplication—one of the most computationally intensive operations in neural network execution. Matrix multiplication plays a crucial role in both the forward and backward propagation phases of the network, and optimizing its performance can lead to significant improvements in overall execution time.

The primary objective of this report is to explore the effectiveness of parallelizing matrix multiplication using CUDA, a powerful parallel computing platform, and compare it to both sequential execution and parallelization via pthreads. By implementing and analyzing these approaches, we aim to provide a comprehensive understanding of how different parallelization strategies impact performance, particularly as data size scales.

The report is structured as follows: we first describe the algorithmic details of the shallow neural network, including the forward and backward propagation processes. We then examine the parallelization strategies, followed by detailed implementations using pthreads and CUDA. Finally, we present a comparison of performance across various data sizes, shedding light on the advantages and trade-offs of each parallelization approach.

2 Shallow Neural Network Algorithm

In this section, we describe the algorithm for training a shallow neural network (SNN) with a single hidden layer. The network performs forward and backward propagation using matrix operations, with the key computational task being matrix multiplication. The primary focus is on presenting the algorithm flow along with a brief mathematical overview.

2.1 Network Parameters

The following table summarizes the key hyperparameters used in the network:

Parameter	**Value**
Input Feature Size	32
Hidden Layer Size	256
Output Size	1
Batch Size	256
Epochs	100
Learning Rate (α)	0.002

Table 1: Hyperparameters for the Shallow Neural Network

2.2 Algorithm Flow

The algorithm follows these steps during each training epoch:

1. **Initialize Weights:** Initialize the weights $\mathbf{W}_1$ and $\mathbf{W}_2$ randomly.
2. **Forward Propagation:**
   - Compute the hidden layer output: $\mathbf{Z}_1 = \mathbf{X} \cdot \mathbf{W}_1$
   - Apply the ReLU activation function: $\mathbf{H} = \text{ReLU}(\mathbf{Z}_1)$
   - Compute the output layer: $\mathbf{Y}_{\text{pred}} = \mathbf{H} \cdot \mathbf{W}_2$
3. **Compute Loss:** Calculate the Mean Squared Error (MSE) between the predicted and true outputs:
$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{Y}_{\text{pred}} - \mathbf{Y})^2$$
4. **Backward Propagation:**
   - Compute the gradient of the output layer: $\mathbf{dZ}_2 = 2 \cdot (\mathbf{Y}_{\text{pred}} - \mathbf{Y})$
   - Compute the weight gradients for the output layer: $\mathbf{dW}_2 = \mathbf{Z}_1{}^T \cdot \mathbf{dZ}_2$
   - Update the weights for the output layer: $\mathbf{W}_2 = \mathbf{W}_2 - \alpha \cdot \mathbf{dW}_2$
   - Compute the gradient of the hidden layer: $\mathbf{dZ}_1 = \mathbf{dZ}_2 \cdot \mathbf{W}_2{}^T$
   - Apply the ReLU derivative and compute the weight gradients for the hidden layer:
$$\mathbf{dW}_1 = \mathbf{X}^T \cdot (\mathbf{dZ}_1 \cdot \text{ReLU}'(\mathbf{Z}_1))$$
   - Update the weights for the hidden layer: $\mathbf{W}_1 = \mathbf{W}_1 - \alpha \cdot \mathbf{dW}_1$
5. **Repeat:** The process repeats for each batch and continues for all training epochs.

2.3 Mathematical Definitions

The primary operations in the algorithm are matrix multiplications and the application of activation functions. The key mathematical steps are as follows:

$$\mathbf{Z}_1 = \mathbf{X} \cdot \mathbf{W}_1, \quad \mathbf{H} = \text{ReLU}(\mathbf{Z}_1), \quad \mathbf{Y}_{\text{pred}} = \mathbf{H} \cdot \mathbf{W}_2$$

Where $\mathbf{X}$ is the input matrix, $\mathbf{W}_1$ and $\mathbf{W}_2$ are the weight matrices for the hidden and output layers, respectively. The activation function applied is ReLU, defined as:

$$\text{ReLU}(x) = \max(0, x)$$

The loss function used is Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{Y}_{\text{pred}} - \mathbf{Y})^2$$

Backpropagation computes the gradients with respect to the weights using the chain rule, and the weights are updated using gradient descent.

2.4 Training Process

The network is trained using Stochastic Gradient Descent (SGD), where the dataset is divided into mini-batches. For each batch, the following steps are performed:

1. Extract a batch ($\mathbf{X}_{\text{batch}}, \mathbf{Y}_{\text{batch}}$) from the dataset.
2. Perform forward propagation to compute the predicted outputs $\mathbf{Y}_{\text{pred}}$.
3. Calculate the loss (MSE) and report it at regular intervals.
4. Perform backpropagation to compute the gradients and update the weights $\mathbf{W}_1$ and $\mathbf{W}_2$.

This process repeats for each batch across multiple epochs, updating the weights iteratively to minimize the loss.

2.5 Dataset Generation Process

The dataset used in this experiment consists of synthetic data with randomly generated input features and corresponding target values. The input features $\mathbf{X}$ are generated uniformly in the range $[-1, 1]$ with a specified number of samples and features. The target output $\mathbf{Y}$ is generated using a convex function $f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$, where x_i is the i-th input feature. To introduce realistic noise in the dataset, Gaussian noise with a standard deviation of 0.1 is added to the target values.

The dataset is generated with the following parameters, summarized in the table below:

Parameter	Value
Dataset size (small)	256 (1 Batch)
Dataset size (medium)	2560 (10 Batches)
Dataset size (large)	25600 (100 Batches)

Table 2: Dataset Specifications

The dataset is stored in CSV format for further use in training the neural network.

2.6 Training Progress as a Proof of Code Correctness

To verify the correctness of the implementation, the training process was monitored by logging the Mean Squared Error (MSE) loss at regular intervals throughout the training. The network was trained on the synthetic dataset with a specified batch size and learning rate. The training loss was recorded every epoch and used to validate that the model is learning as expected.

The MSE loss decreased as the number of epochs increased, which indicates that the model was successfully learning from the dataset. The first and last MSE values recorded during training were as follows:

$$\text{Epoch 1, MSE: 52.19,} \quad \text{Epoch 99, MSE: 1.52.}$$

For a more comprehensive view of the training progress, including intermediate values, please refer to the training plot shown in Figure 1.

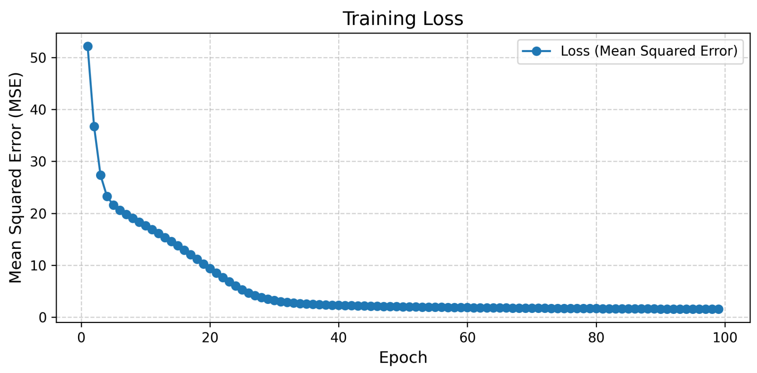

Figure 1: Training Loss (MSE) Progress Over Epochs. The plot shows the Mean Squared Error (MSE) at each epoch of training, demonstrating the model's learning progression.

3 Parallelization Strategy for Neural Network Training using Pthreads

The primary focus of parallelization in this implementation is matrix multiplication, which is central to both the forward pass (to compute activations) and the backward pass (to compute gradients). Matrix multiplication is a computational bottleneck, but its row-wise independence allows for efficient parallelization using threads.

- **Forward Pass:** Parallelized matrix multiplication is used to compute activations for each layer.
- **Backward Pass:** Matrix multiplication for gradient computations during backpropagation is parallelized.

Figure 2 illustrates the distribution of matrix multiplication computations across threads.

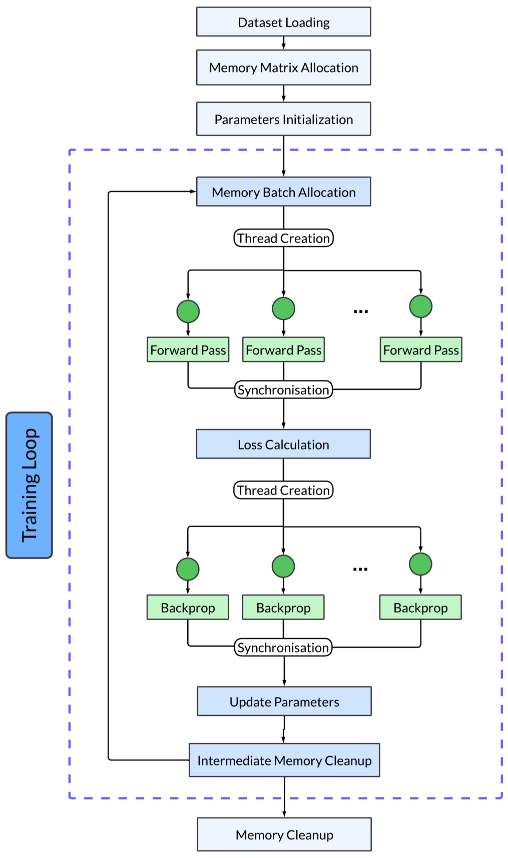

Figure 2: Parallelization Strategy Using Pthreads for Matrix Multiplication in Neural Network Training.

3.1 Algorithm for Parallel Matrix Multiplication

Algorithm 1 outlines the implementation of parallel matrix multiplication using `pthreads`. It distributes rows of the first matrix (A) among multiple threads, with each thread computing the corresponding rows of the resulting matrix (C).

Algorithm 1 Parallel Matrix Multiplication

```
Require: Matrices $A$ ($m \times n$) and $B$ ($n \times p$), number of threads $T$
Ensure: Matrix $C$ ($m \times p$) as the product of $A$ and $B$
1: function ParallelMatrixMultiply(A, B, T)
2: if $A$.cols $\neq$ $B$.rows then
3:     Raise Error ("Incompatible matrices")
4: end if
5: Initialize $C \leftarrow$ AllocateMatrix($A$.rows, $B$.cols)
6: RowsPerThread $\leftarrow$ $A$.rows $\div$ $T$
7: Initialize threads and thread_data arrays
8: for $t \leftarrow 0$ to $T-1$ do                    ▷ Distribute rows among threads
9:     thread_data[t] $\leftarrow$ {
10:        $A$: matrix $A$, $B$: matrix $B$, $C$: matrix $C$,
11:        start_row: $t \cdot$ RowsPerThread,
12:        end_row: $(t = T-1)?A.\text{rows} : (t+1) \cdot$ RowsPerThread }
13:    CreateThread(threads[t], MatMultThread, thread_data[t])
14: end for
15: for $t \leftarrow 0$ to $T-1$ do                   ▷ Wait for all threads to finish
16:    JoinThread(threads[t])
17: end for
18: return $C$

19: function MatMultThread(data)
20: for $i \leftarrow$ data.start_row to data.end_row $-1$ do
21:    for $j \leftarrow 0$ to $B$.cols $-1$ do
22:        $C[i][j] \leftarrow 0$
23:        for $k \leftarrow 0$ to $A$.cols $-1$ do
24:            $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$
25:        end for
26:    end for
27: end for
```

3.2 Benefits of Parallel Matrix Multiplication

This parallelization strategy, as illustrated in 2, ensures:

- **Load Balancing:** Each thread is assigned a roughly equal number of rows, ensuring fair distribution of the workload.
- **Scalability:** Performance scales with the number of threads and matrix size.
- **Efficiency:** Computational bottlenecks in forward and backward passes are mitigated, improving overall training speed.

3.3 Parallelization Results

In this subsection, we explore the performance of neural network training using parallel execution with Pthreads, comparing the execution times for three data sizes: small, medium, and large. As shown in Figure 3, Pthreads effectively accelerates the training process across all three data sizes. The parallel execution reduces the overall execution time significantly when compared to sequential execution, demonstrating the potential of parallel computing in neural network training.

A common observation across all three data sizes is the behavior of parallelization efficiency. Initially, as the number of threads increases, the execution time continues to decrease, showing improved performance. However, beyond a certain point, we begin to notice diminishing returns, where the performance gains slow down or even reverse. This behavior can be attributed to the overhead introduced by managing additional threads, which eventually offsets the benefits of parallelization.

For each data size, the optimal number of threads can be observed where the performance improvement plateaus. This point indicates the threshold at which further increasing the number of threads no longer leads to better performance and can even result in increased overhead.

Table 3 presents the detailed execution times for both sequential and parallel executions for each dataset size. The results confirm that Pthreads significantly accelerates training, though the overhead effect becomes more pronounced as the number of threads increases.

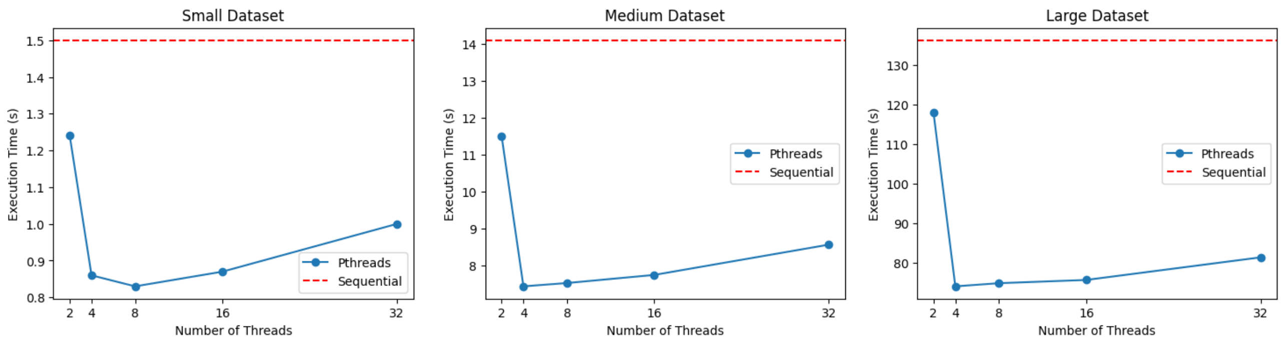

Figure 3: Execution time comparison between sequential and parallel execution (Pthreads) for small, medium, and large data sizes. The red dashed line represents the sequential execution time, and the solid lines represent the execution times for varying numbers of threads.

Dataset / Threads	**Sequential**	2	4	8	16	32
Small	**1.50**	1.24	0.86	**0.83**	0.87	1.00
Medium	**14.11**	11.51	**7.44**	7.53	7.75	8.57
Large	**136.26**	117.93	**74.00**	74.80	75.62	81.34

Table 3: Execution times (in seconds) for different numbers of threads for small, medium, and large data sizes.

4 Parallelization Strategy for Neural Network Training using CUDA

Graphics Processing Units (GPUs) are inherently suited for parallel computation due to their massive number of cores and efficient thread management. In this section, we focus on parallelizing matrix multiplication using CUDA, a critical operation in neural network training. We outline the suitability of GPU parallel computation for this problem and detail the specific parallelization strategy employed.

4.1 Suitability of CUDA for Matrix Multiplication in Neural Networks

Matrix multiplication is a highly parallelizable operation, especially in the context of neural networks where large matrices are common. Each element of the output matrix $C[m, p]$ is independent, as it results from a dot product between a row of $A[m, n]$ and a column of $B[n, p]$. This independence makes it ideal for GPU parallelization, where thousands of threads can execute simultaneously.

CUDA enables this parallelism through a grid and block hierarchy:

- **Grid:** The computation grid is shaped to match the dimensions of the output matrix $C[m, p]$, with one thread assigned to compute each element.
- **Blocks:** Each grid is divided into blocks of threads. For efficient memory access and execution, we use two-dimensional blocks (e.g., `threadsPerBlock.x` $\times$ `threadsPerBlock.y`).
- **Threads:** Each thread calculates a single element of $C[m, p]$ by performing a dot product of the corresponding row of A and the column of B.

This structure ensures that the workload is evenly distributed among threads, leading to optimal utilization of GPU resources.

4.2 CUDA Parallelization Strategy for Matrix Multiplication

The core of CUDA parallelization for matrix multiplication lies in leveraging thread-level parallelism to calculate the elements of the result matrix $C[m, p]$, as depicted in Figure 4. The workflow is as follows:

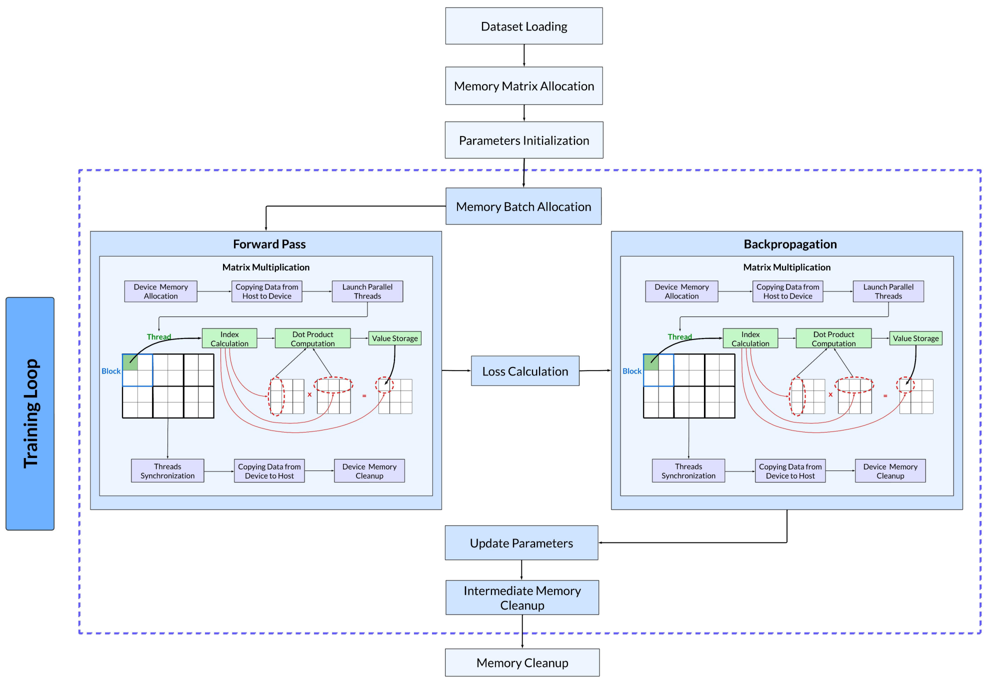

Figure 4: Parallelization Strategy Using Cuda for Matrix Multiplication in Neural Network Training.

- **Device Memory Management:** Allocate device memory for the input matrices $A[m, n]$ and $B[n, p]$, and the output matrix $C[m, p]$. Copy A and B from the host (CPU) to the device (GPU) memory.
- **Kernel Execution:** Launch a grid of threads, where each thread computes one element $C[row, col]$. This involves:
  - Determining the thread's `row` and `col` indices based on its position in the grid and block.
  - Performing boundary checks to ensure threads do not access out-of-bounds elements.
  - Calculating the dot product of the `row`-th row of A and the `col`-th column of B, storing the result in $C[row, col]$.
- **Thread Synchronization and Data Transfer:** Synchronize threads and copy the result matrix C back to host memory.
- **Cleanup:** Free device memory for A, B, and C.

4.3 Kernel Logic and Pseudo-code for CUDA Matrix Multiplication

Below, we present pseudo-code for the kernel and the CUDA matrix multiplication workflow, closely reflecting the implementation structure.

Algorithm 2 CUDA Matrix Multiplication Kernel

```
1: Input: Matrices $A[m, n]$, $B[n, p]$, and dimensions $A\_rows, A\_cols, B\_cols$
2: Output: Resultant matrix $C[m, p]$
3: Determine row and col indices of the thread:
       $\texttt{row} = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y}$
       $\texttt{col} = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$
4: if $\texttt{row} < A\_rows$ and $\texttt{col} < B\_cols$ then
5:     Initialize $\texttt{value} \leftarrow 0.0$
6:     for $k = 0$ to $A\_cols - 1$ do
7:         $\texttt{value} \leftarrow \texttt{value} + A[\texttt{row} \times A\_cols + k] \times B[k \times B\_cols + \texttt{col}]$
8:     end for
9:     $C[\texttt{row} \times B\_cols + \texttt{col}] \leftarrow \texttt{value}$
10: end if
```

Algorithm 3 CUDA Matrix Multiplication Workflow

```
1: Input: Host matrices $A[m, n]$, $B[n, p]$
2: Output: Resultant matrix $C[m, p]$
3: Check matrix compatibility: Ensure $A\_cols == B\_rows$
4: Allocate device memory for $d\_A, d\_B, d\_C$ based on their sizes
5: Copy $A$ and $B$ from host to device memory
6: Define grid and block dimensions:
       $\texttt{threadsPerBlock.x} = \texttt{THREADS\_PER\_BLOCK}, \quad \texttt{threadsPerBlock.y} = \texttt{THREADS\_PER\_BLOCK}$
       $\texttt{numBlocks.x} = \lceil B\_cols/\texttt{THREADS\_PER\_BLOCK} \rceil, \quad \texttt{numBlocks.y} = \lceil A\_rows/\texttt{THREADS\_PER\_BLOCK} \rceil$
7: Launch the kernel:
       mat_mult_kernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, A_rows, A_cols, B_cols)
8: Synchronize threads and check for kernel launch errors
9: Copy $d\_C$ from device to host memory
10: Free device memory for $d\_A, d\_B, d\_C$
```

4.4 Parallelization Results

The results of CUDA parallelization for matrix multiplication, a key component of neural network training, demonstrate a significant improvement in execution time compared to both sequential execution and the best configuration achieved using pthreads. Table 4 summarizes the execution times for datasets of different sizes (small, medium, large), comparing sequential, pthreads, and CUDA implementations with varying thread-per-block configurations.

Dataset / Threads Per Block	Sequential	Pthreads Best	(2x2)	(4x4)	(8x8)	(16x16)	(32x32)
Small	**1.50**	**0.83**	0.50	0.49	**0.45**	0.46	**0.45**
Medium	**14.11**	**7.44**	2.86	2.69	2.69	**2.59**	2.65
Large	**136.26**	**74.00**	25.75	23.02	23.20	**22.22**	22.31

Table 4: Execution times (in seconds) for sequential and CUDA kernel implementations with varying threads per block configurations for small, medium, and large data sizes.

Figure 5 visualizes the execution times of CUDA parallelization across different datasets and thread configurations, highlighting the comparison with sequential and pthreads best executions. The results clearly show that CUDA execution significantly surpasses both the sequential execution and the best pthreads parallelization, achieving remarkable speedups. This demonstrates the efficiency of GPU parallelization in matrix multiplication for neural network training, leveraging the full power of the CUDA architecture.

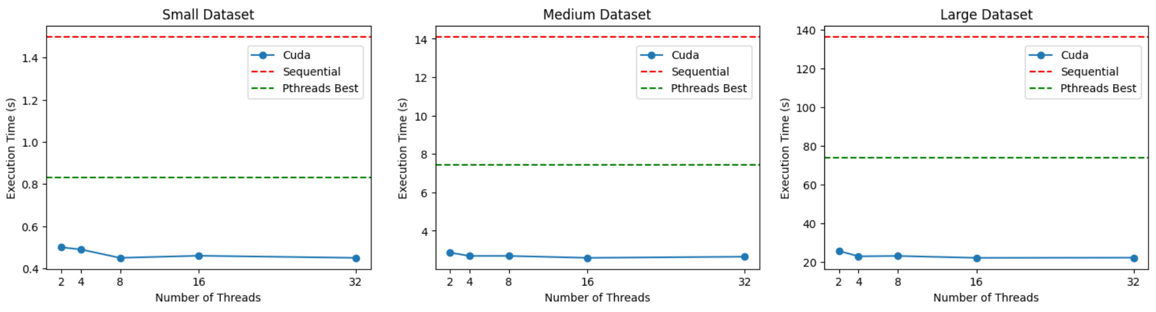

Figure 5: Execution times for CUDA kernel implementations with varying thread-per-block configurations compared against sequential and pthreads best implementations. Each sub-plot corresponds to a dataset size (small, medium, large).

4.4.1 Experimental Setup

The experiments were conducted on a system with the following GPU configuration, as detailed in Table 5.

Feature	**Specification**
Number of CUDA-capable devices	1
Device	NVIDIA GeForce GTX 1650
Compute Capability	7.5
Total Global Memory	3718 MB
Multiprocessor Count	14
Max Threads Per Block	1024
Max Threads Dimensions	1024 x 1024 x 64
Max Grid Size	2147483647 x 65535 x 65535

Table 5: Specifications of the experimental GPU setup.

4.4.2 Observations and Analysis

The prominent observations from the results are as follows:

- **Performance Gains:** CUDA execution times are significantly faster than both sequential execution and best pthreads parallelization across all dataset sizes. This demonstrates the superiority of GPU parallelization for matrix multiplication, with CUDA effectively utilizing the massive parallelism of modern GPUs to achieve substantial speedups.
- **Optimal Configurations:** For small (closely), medium, and large datasets, the 16 $\times$ 16 thread-per-block configuration consistently delivers the best performance. This configuration appears to strike an optimal balance between thread utilization and computational efficiency, leveraging the hardware capabilities to their fullest extent.
- **Scalability and Efficiency:** The scalability of CUDA execution is evident, as the execution time for larger datasets is considerably reduced when compared to sequential execution, even in the presence of high thread

counts. The GPU's architecture is well-suited for handling the increased computational demand of larger matrices, achieving efficiency even as the problem size grows.

- **Comparison with Pthreads:** When compared to the pthreads implementation, CUDA not only reduces execution time but also demonstrates greater scalability as the dataset size increases. The best pthreads parallelization configuration shows a notable performance gap when compared to CUDA, especially for large datasets, emphasizing the hardware-accelerated advantage of GPU computing.

5 Conclusion

In this work, we explored the parallelization of matrix multiplication, a critical operation in the training and execution of shallow neural networks (SNNs) with a single hidden layer. Matrix multiplication is integral to both the forward and backward propagation phases, where optimizing its execution can significantly reduce overall training time. Our primary goal was to investigate the performance improvements achieved through parallelization using CUDA, a high-performance parallel computing platform, and compare it to sequential execution and pthreads parallelization.

Throughout the experimental process, we proposed parallelization strategies for both pthreads and CUDA, comparing the performance of different configurations across small, medium, and large datasets. Our experiments confirmed that CUDA execution far outperforms both sequential execution and best pthreads parallelization in terms of execution time. This superiority can be attributed to CUDA's ability to fully utilize the GPU's parallel architecture, exploiting thread-level parallelism and high-throughput memory access to achieve remarkable performance gains.

The results of the experimentation highlight the efficiency of CUDA in maximizing resource utilization, enabling faster matrix multiplication and, by extension, faster neural network training. Given the computational intensity of matrix multiplication in neural networks, this makes CUDA a preferred choice in deep learning applications. With frameworks such as PyTorch and TensorFlow being highly compatible with CUDA, its widespread adoption in training large-scale deep learning models is a testament to the power and flexibility of GPU acceleration. By leveraging CUDA's capabilities, deep learning practitioners can achieve faster model training, which is crucial in advancing research and development in the field.

In conclusion, our study underscores the pivotal role of GPU-based parallelism in modern neural network training. The efficiency and scalability of CUDA parallelization not only provide substantial speedups in matrix operations but also open the door for further optimizations and innovations in deep learning workflows.