ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

Memory Management Strategies for GPU-Accelerated Shallow Neural Network Training

Mohamed El Amine Kherroubi
2CS-SIQ
mm_kherroubi@esi.dz

Badis Khalef
2CS-SID
mb_khalef@esi.dz

Mounir Sofiane Mostefai
2CS-SIQ
mm_mostefai@esi.dz

Youcef Tati
2CS-SIQ
my_tati@esi.dz

Mohamed Ishak Messadia
2CS-SIQ
mm_messadia@esi.dz

***Abstract*—This project was conducted at the École Nationale Supérieure d'Informatique (ESI), Algiers, in February 2026. Building upon prior work by Brouthen and Akeb [1] on GPU parallelization for shallow neural network training, we evaluate three memory management strategies for CUDA implementations. Their reference implementation allocates and frees GPU memory for every matrix operation. We implemented three alternatives: a streams-based approach using CUDA streams for concurrent execution, a pinned memory strategy using page-locked host memory, and a combined approach that integrates both optimizations with pre-allocated GPU resources. Our experiments on a Tesla T4 GPU show that the combined strategy achieves approximately 1.65× to 1.73× speedup over the reference implementation for the baseline network configuration, by eliminating repeated `cudaMalloc` and `cudaFree` calls. We validate functional correctness across all strategies and provide additional scalability analysis with varying numbers of neurons and network depths. All code is publicly available at github.com/amine-kherroubi/snn-optimization.**

***Index Terms*—CUDA, GPU Computing, Neural Networks, Pinned Memory, CUDA Streams, Memory Management, Performance Optimization**

I. Introduction

A. Motivation

Matrix multiplication is the primary computational bottleneck in neural network training, occurring repeatedly during both forward and backward propagation. Brouthen and Akeb [1] demonstrated that GPU acceleration with CUDA significantly speeds up shallow neural network training compared to CPU implementations.

However, their reference implementation allocates and frees GPU memory with `cudaMalloc` and `cudaFree` for each matrix operation. For a typical training run with 100 epochs and multiple batches, this leads to tens of thousands of allocation-deallocation cycles. With a dataset containing 100 batches, the reference network (1 hidden layer) performs 5 matrix multiplications per batch, yielding 50,000 allocation-deallocation pairs over 100 epochs. Additionally, synchronous transfers with pageable host memory via `cudaMemcpy` block the CPU, preventing overlap between computation and data transfers.

II. Objectives

This work evaluates three CUDA memory management strategies applied to the reference network:

1) **Streams-based approach:** Using CUDA streams to enable concurrent kernel execution and memory transfers
2) **Pinned memory:** Using page-locked host memory to accelerate data transfers
3) **Combined approach:** Integrating pinned memory and streams with pre-allocated GPU resources

Our goals are to quantify the performance impact of each strategy, validate functional correctness, and understand which optimizations provide meaningful speedups for shallow neural network training. In addition, we aim to determine how architectural parameters influence the effectiveness of memory-oriented optimizations.

III. Paper Organization

Section 2 reviews the neural network architecture. Section 3 analyzes the reference implementation. Section 4 describes our three optimization strategies. Section 5 details the experimental setup including functional validation. Section 6 presents performance results for the optimization strategies. Section 7 provides additional scalability analysis. Section 8 discusses findings. Section 9 concludes.

IV. Neural Network Overview

This section briefly reviews the shallow neural network used in our primary experiments. For complete details, refer to Brouthen and Akeb [1].

A. Reference Network Architecture

The reference network consists of:

- Input layer with 32 features
- One hidden layer with 256 neurons using ReLU activation
- Single output neuron for regression

Training uses stochastic gradient descent with batch size 256, learning rate 0.002, and 100 epochs.

B. Operations

Forward propagation:

$$Z_1 = XW_1, \quad H = \text{ReLU}(Z_1), \quad Y_{\text{pred}} = HW_2 \tag{1}$$

Backward propagation:

$$\text{dZ}_2 = 2\frac{Y_{\text{pred}} - Y}{n}, \quad \text{dW}_2 = H^T \text{dZ}_2 \tag{2}$$

$$\text{dZ}_1 = \text{dZ}_2 W_2^T \otimes \text{ReLU'}(Z_1), \quad \text{dW}_1 = X^T \text{dZ}_1 \tag{3}$$

Weight updates:

$$W_2 = W_2 - \alpha \text{dW}_2, \quad W_1 = W_1 - \alpha \text{dW}_1 \tag{4}$$

Loss (Mean Squared Error):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} \left(Y_{\text{pred}} - Y\right)^2 \tag{5}$$

C. Computational Pattern

Each training batch requires 5 matrix multiplications: 2 for forward propagation and 3 for backward propagation. With 100 epochs and multiple batches per epoch, matrix multiplication dominates execution time, making transfer and allocation overhead significant.

V. Reference Implementation

A. Overview

The reference implementation follows a consistent pattern for each matrix multiplication:

1) Allocate device memory using `cudaMalloc`
2) Copy inputs from host to device using `cudaMemcpy`
3) Launch CUDA kernel for computation
4) Synchronize with `cudaDeviceSynchronize`
5) Copy results from device to host using `cudaMemcpy`
6) Free device memory using `cudaFree`

The CUDA kernel implements standard matrix multiplication with each thread computing one output element:

```
__global__ void mat_mult_kernel(float *A,
float *B, float *C,
                                int A_rows,
int A_cols, int B_cols) {
  int row = blockIdx.y * blockDim.y +
threadIdx.y;
  int col = blockIdx.x * blockDim.x +
threadIdx.x;
  if (row < A_rows && col < B_cols) {
    float value = 0.0f;
    for (int k = 0; k < A_cols; k++) {
      value += A[row * A_cols + k] * B[k *
B_cols + col];
    }
    C[row * B_cols + col] = value;
  }
}
```

The implementation uses 16×16 thread blocks. The matrix multiplication wrapper allocates device memory on every call:

```
Matrix *mat_mult(Matrix *A, Matrix *B) {
  Matrix *C = allocate_matrix(A->rows, B-
>cols);

  // Allocated and freed on every call
  cudaMalloc((void **)&d_A, sizeA);
  cudaMalloc((void **)&d_B, sizeB);
  cudaMalloc((void **)&d_C, sizeC);

  cudaMemcpy(d_A, A->data, sizeA,
cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B->data, sizeB,
cudaMemcpyHostToDevice);

  mat_mult_kernel<<<numBlocks,
threadsPerBlock>>>(d_A, d_B, d_C, ...);
  cudaDeviceSynchronize();

  cudaMemcpy(C->data, d_C, sizeC,
cudaMemcpyDeviceToHost);
  cudaFree(d_A); cudaFree(d_B);
cudaFree(d_C);

  return C;
}
```

B. Observed Limitations

Repeated Memory Allocation: Allocating and freeing GPU memory for every matrix multiplication is the most significant limitation. Each `cudaMalloc` call incurs CUDA runtime overhead, and this cost accumulates over tens of thousands of calls during training.

Synchronous Transfers: The implementation uses `cudaMemcpy`, which blocks the CPU until each transfer completes, preventing overlap between data transfers and computation.

Pageable Host Memory: Standard `malloc` allocates pageable memory. Before transferring data to the GPU, CUDA must stage it through a temporary pinned buffer, adding latency.

No Resource Reuse: Device memory is never reused across matrix multiplication calls.

VI. Optimization Strategies

A. Strategy 1: Streams-Based Approach

a) *Implementation:*

We used CUDA streams to attempt overlapping host-to-device transfers, kernel execution, and device-to-host transfers:

- All host matrix allocations use `cudaMallocHost` (pinned memory), including weights, activations, and gradient matrices
- Three persistent device buffers are allocated per stream for tiling matrix A and the result matrix C
- Matrix B is copied to the device at the start of each `mat_mult` call and freed at the end
- Matrix A is divided into tiles of 128 rows; tiles are processed across the three streams in round-robin fashion
- Each stream performs: asynchronous H2D copy of the A tile → kernel launch → asynchronous D2H copy of the C tile
- All streams are synchronized before returning

The key code pattern:

```
Matrix *mat_mult(Matrix *A, Matrix *B) {
  cudaMalloc((void **)&d_B, sizeB);
  cudaMemcpy(d_B, B->data, sizeB,
cudaMemcpyHostToDevice);

  float *d_A_tiles[NUM_STREAMS],
*d_C_tiles[NUM_STREAMS];
  cudaStream_t streams[NUM_STREAMS];
  for (int s = 0; s < NUM_STREAMS; s++) {
    cudaMalloc((void **)&d_A_tiles[s],
tile_bytes_A);
    cudaMalloc((void **)&d_C_tiles[s],
tile_bytes_C);
    cudaStreamCreate(&streams[s]);
  }

  for (int row_start = 0, tile_idx = 0;
row_start < A->rows;
       row_start += TILE_ROWS, tile_idx++) {
    int s = tile_idx % NUM_STREAMS;
    cudaMemcpyAsync(d_A_tiles[s],
A_tile_host, ..., streams[s]);
    mat_mult_kernel<<<...,
streams[s]>>>(d_A_tiles[s], d_B, ...);
    cudaMemcpyAsync(C_tile_host,
d_C_tiles[s], ..., streams[s]);
  }

  for (int s = 0; s < NUM_STREAMS; s++) {
    cudaStreamSynchronize(streams[s]);
    cudaStreamDestroy(streams[s]);
    cudaFree(d_A_tiles[s]);
cudaFree(d_C_tiles[s]);
  }
  cudaFree(d_B);
  return C;
}
```

Streams and tile device buffers are allocated and freed on each `mat_mult` call, so GPU allocation overhead is not eliminated.

B. Strategy 2: Pinned Memory Approach

a) *Implementation:*

This strategy isolates the effect of pinned host memory:

- Intermediate matrices returned by `mat_mult` are allocated with `cudaMallocHost`
- Weight matrices, batch inputs, and batch targets are also allocated with `cudaMallocHost`
- Device memory allocation follows the same per-call pattern as the reference
- All transfers remain synchronous via `cudaMemcpy`

```
Matrix *allocate_matrix_pinned(int rows, int
cols) {
  Matrix *m = (Matrix
*)malloc(sizeof(Matrix));
  m->rows = rows; m->cols = cols;
  m->pinned = 1;
  cudaError_t err =
      cudaMallocHost((void **)&m->data, rows
* cols * sizeof(float));
  if (err != cudaSuccess) {
    m->data = (float *)malloc(rows * cols *
sizeof(float));
    m->pinned = 0;
  }
  return m;
}

void free_matrix(Matrix *m) {
  if (m->pinned) cudaFreeHost(m->data);
  else free(m->data);
  free(m);
}
```

Short-lived intermediate matrices (transpose buffers, ReLU derivative arrays) that do not participate in GPU transfers remain allocated with standard `malloc`.

C. Strategy 3: Combined Approach with Global Memory Pool

a) *Implementation:*

This strategy addresses the core bottleneck by pre-allocating all GPU resources once in a global context:

```
typedef struct {
  float *d_A_tiles[NUM_STREAMS];
  float *d_C_tiles[NUM_STREAMS];

  // Persistent streams
  cudaStream_t streams[NUM_STREAMS];

  size_t tile_bytes_A;
  size_t tile_bytes_C;
  int initialized;
} GlobalGPUContext;

static GlobalGPUContext g_gpu_ctx = {0};
```

Initialization occurs once on the first `mat_mult` call:

```
void init_global_gpu_context(int tile_rows,
int A_cols, int B_cols) {
  if (g_gpu_ctx.initialized) return;

  for (int s = 0; s < NUM_STREAMS; s++) {
    cudaMalloc((void **)
&g_gpu_ctx.d_A_tiles[s], tile_bytes_A);
    cudaMalloc((void **)
&g_gpu_ctx.d_C_tiles[s], tile_bytes_C);
    cudaStreamCreate(&g_gpu_ctx.streams[s]);
  }
```

```
  g_gpu_ctx.initialized = 1;
}
```

For each operation, the implementation reuses pre-allocated resources:

```
Matrix *mat_mult(Matrix *A, Matrix *B) {
  if (!g_gpu_ctx.initialized)
    init_global_gpu_context(TILE_ROWS, A->cols, B->cols);

  // Ensure tile buffers are large enough for this operation
  ensure_tile_capacity(A->cols, B->cols);

  // Tiled loop reuses g_gpu_ctx.d_A_tiles, d_C_tiles, streams
}
```

At program termination, all resources are freed once:

```
void cleanup_global_gpu_context() {
  for (int s = 0; s < NUM_STREAMS; s++) {
    cudaFree(g_gpu_ctx.d_A_tiles[s]);
    cudaFree(g_gpu_ctx.d_C_tiles[s]);
    cudaStreamDestroy(g_gpu_ctx.streams[s]);
  }
  g_gpu_ctx.initialized = 0;
}
```

Host memory uses `cudaMallocHost` (pinned), as in the streams approach.

b) *Key Differences:*

Unlike the streams-only approach, this strategy:

- Allocates GPU tile buffers and streams once at initialization
- Keeps streams persistent across operations

VII. Experimental Setup

A. Hardware and Software

All experiments used Google Colab with a Tesla T4 GPU:

TABLE I
Tesla T4 GPU Specifications

Parameter	Value
CUDA Cores	2560
Memory Size	16 GB GDDR6
Memory Bandwidth	320 GB/s
Compute Capability	7.5
Copy Engines	3

The software environment consists of CUDA Toolkit 11.8, compiled with `nvcc` using `-O3` optimization, and GCC 9.4.0 as the host compiler.

B. Datasets

We used three synthetic datasets identical to those in the reference solution:

TABLE II
Dataset Configurations

Dataset	Samples	Batches	Operations
Small	256	1	500
Medium	2560	10	5,000
Large	25600	100	50,000

Each dataset has 32 input features sampled uniformly from $[-1, 1]$. Target values are defined as $Y = \sum_{i=1}^{32} x_i^2 + \varepsilon$, where $\varepsilon \sim \mathcal{N}(0, 0.01)$. The total number of matrix operations equals the number of batches times 100 epochs times 5 operations per batch.

C. Functional Correctness Validation

To verify that our optimization strategies produce identical training outcomes to the reference, we:

1) Used identical random seeds for weight initialization across all implementations
2) Trained all four implementations on the same small dataset for 100 epochs
3) Compared final MSE values

TABLE III
Functional Correctness Validation (small dataset)

Strategy	Training Time (s)	Final MSE
Reference	0.3259	41.192574
Streams	0.2914	41.192574
Pinned	0.2770	41.192574
Combined	0.1958	41.192574

All four implementations converge to the same MSE, confirming functional correctness.

D. Metrics

Training Time: Total wall-clock time for 100 epochs, averaged over 10 independent runs.

Speedup: Ratio of reference time to alternative strategy time. Values greater than 1.0 indicate improvement.

E. Evaluation Protocol

For each strategy and dataset:

1) Load dataset and initialize weights with a fixed random seed
2) Run 10 independent training sessions, each re-initializing weights
3) Record training time per session
4) Compute mean training time
5) Calculate speedup relative to the reference mean

VIII. Performance Results

A. Dataset Size Variation

We evaluated all three optimization strategies across the small, medium, and large datasets:

TABLE IV
TRAINING TIME ACROSS DATASET SIZES

Strategy	Small (s)	Medium (s)	Large (s)
Reference	0.3259	3.0828	30.3694
Streams	0.2914	3.0978	30.7172
Pinned	0.2770	2.8901	29.2370
Combined	0.1958	1.7841	18.4545

TABLE V
SPEEDUP RELATIVE TO REFERENCE

Strategy	Small	Medium	Large
Reference	1.00×	1.00×	1.00×
Streams	1.12×	1.00×	0.99×
Pinned	1.18×	1.07×	1.04×
Combined	1.66×	1.73×	1.65×

B. Analysis of Optimization Strategies

Streams-only approach: This strategy provided minimal improvement (0.99× to 1.12× speedup) despite the Tesla T4's 3 copy engines. While the hardware supports concurrent transfers, streams and tile device buffers are allocated and freed on each `mat_mult` call, failing to eliminate the allocation overhead. Additionally, for batch size 256 and tile height 128 rows, each matrix produces only two tiles, providing insufficient pipeline depth to effectively utilize the available copy engines and amortize stream setup overhead.

Pinned memory approach: This strategy achieved modest but consistent improvements (4–18% speedup) by removing the intermediate staging copy that CUDA performs when transferring pageable memory. The gains are limited because device-side allocation overhead remains unchanged.

Combined approach: This strategy achieved the largest speedups (1.65× to 1.73×) by eliminating repeated GPU allocation calls. For the large dataset with 50,000 matrix operations, pre-allocating resources once and reusing them throughout training provides significant savings. The combination of pinned memory and persistent streams with global resource pooling addresses the primary bottleneck in the reference implementation.

IX. ADDITIONAL SCALABILITY ANALYSIS

Beyond the core optimization strategies, we conducted additional experiments to understand how network configuration affects performance.

A. Neuron Count Variation

We analyzed how the number of neurons in the hidden layer affects the speedup achieved by the combined strategy. Note: Due to experimental constraints, these measurements were conducted in a separate session with different baseline performance characteristics than Table 4. While absolute timing values differ between sessions, the relative trends in how neuron count affects optimization effectiveness remain valid.

TABLE VI
TRAINING TIME VS. NEURON COUNT (LARGE DATASET, SEPARATE EXPERIMENTAL SESSION)

Neurons	Reference (s)	Combined (s)	Speedup
128	13.7765	6.1372	2.24×
256	19.0889	8.9194	2.14×
1024	50.9751	55.0013	0.93×

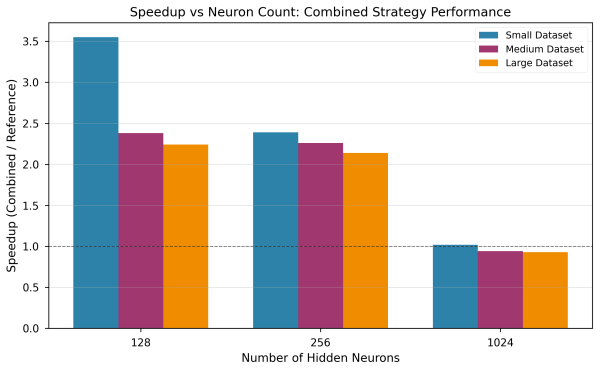

Fig. 1. Speedup of the combined strategy versus neuron count. Performance degrades at 1024 neurons.

For small to moderate network sizes (128–256 neurons), each matrix multiplication completes quickly, making allocation overhead a significant portion of total runtime. The combined strategy achieves substantial speedups by eliminating this overhead. For large networks (1024 neurons), computation time dominates, and the tiling overhead causes the combined strategy to perform worse than the reference (0.93× speedup).

The key finding is that optimization effectiveness depends heavily on the allocation-to-computation ratio: smaller networks benefit more from memory pooling optimizations.

B. Network Depth Variation

We extended the baseline reference network architecture to include additional hidden layers:

Two-hidden-layer network: Input → Hidden1 (256) → Hidden2 (256) → Output

Three-hidden-layer network: Input → Hidden1 (256) → Hidden2 (1024) → Hidden3 (256) → Output

Each additional hidden layer increases the number of matrix operations per batch from 5 (reference) to 8 (two-hidden-layer) to 11 (three-hidden-layer).

TABLE VII
TRAINING TIME: NETWORK DEPTH COMPARISON (LARGE DATASET)

Architecture	Reference (s)	Combined (s)	Speedup
Reference (1 hidden)	30.3694	18.4545	1.65×
Two-hidden-layer	69.7862	45.9064	1.52×
Three-hidden-layer	285.8243	282.2075	1.01×

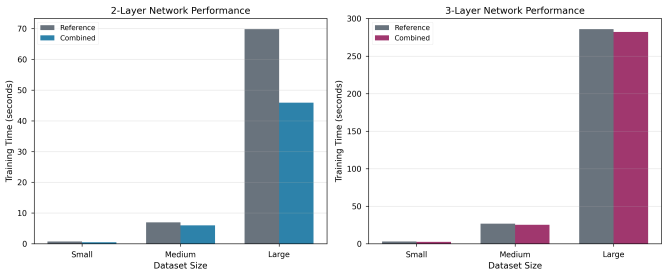

Fig. 2. Training time comparison across network depths. The combined strategy's advantage diminishes progressively as network depth increases.

The results show a gradual degradation in speedup as network depth increases. For the two-hidden-layer network, the combined strategy still achieves 1.52× speedup, demonstrating that memory pooling optimizations remain effective for moderately deep networks. However, for the three-hidden-layer network with a large 1024-neuron second hidden layer, speedup drops to 1.01×.

The primary factor is that as network depth increases from 1 to 3 hidden layers, the number of matrix operations per batch grows from 5 to 11. This growth in computational workload, particularly when combined with large intermediate activations (1024 neurons in the second hidden layer), causes computation time to dominate execution. The fixed allocation savings achieved by the combined strategy become negligible relative to the increased computation cost, reducing the relative benefit of memory pooling.

X. Discussion

A. Understanding the Results

The performance of memory management optimizations is not uniform across configurations. The combined strategy eliminates repeated `cudaMalloc` and `cudaFree` calls, which is most beneficial when those calls represent a large fraction of total runtime.

For the reference network configuration (256 neurons, 1 hidden layer), the combined strategy achieves 1.65× to 1.73× speedup across dataset sizes. Allocation overhead constitutes a significant portion of total time, and eliminating over 50,000 GPU allocation calls for the large dataset provides measurable benefits.

However, as network size or depth increases, computation time dominates. For 1024 neurons, the combined strategy performs 7% worse than the reference due to tiling overhead. For deeper architectures, speedup degrades progressively: the two-hidden-layer network maintains 1.52× speedup, while the three-hidden-layer network achieves only 1.01× speedup. The large second hidden layer (1024 neurons) in the three-layer architecture increases computation time relative to the fixed allocation savings, effectively nullifying the optimization benefits.

B. Why Streams Alone Did Not Help

The streams strategy provided minimal improvement (0.99× to 1.12×) due to two factors:

Unchanged allocation overhead: Streams, tile device buffers, and matrix B are allocated and freed on each `mat_mult` call, leaving the primary bottleneck unaddressed.

Small tile count: For batch size 256 and tile height 128 rows, each matrix produces only two tiles. With so few tiles, pipeline depth is insufficient to amortize stream setup overhead.

C. Pinned Memory: Modest but Consistent Gains

Pinned memory produced 4–18% speedup by removing the intermediate staging copy for pageable memory transfers. The gains are bounded because device-side allocation overhead is unchanged, and as dataset size grows, transfer latency becomes a smaller fraction of total runtime relative to computation.

D. Combined Strategy: Effective Within Limits

The combined approach yielded the largest speedups for the reference network configuration (1.65× to 1.73×) by addressing the core allocation bottleneck. However, its effectiveness is bounded by the relative cost of allocation versus computation. When computation dominates (1024 neurons or deeper networks), the tiling and caching overhead can outweigh the allocation savings.

XI. Conclusion

We evaluated three memory management strategies for GPU-accelerated neural network training on a Tesla T4, leading to the following observations:

Individual strategies: Streams alone provided minimal benefit (0.99× to 1.12×) due to hardware constraints and unchanged allocation overhead. Pinned memory achieved modest improvements (1.04× to 1.18×) by accelerating transfers.

Combined strategy: For the reference network configuration (256 neurons, 1 hidden layer), the combined approach achieved 1.65× to 1.73× speedup across different dataset sizes by eliminating repeated GPU allocation calls. Additional experiments showed that speedup ranges from 2.14× for moderate configurations to 0.93× for large configurations (1024 neurons), demonstrating that effectiveness depends critically on the allocation-to-computation ratio.

Configuration dependence: Performance gains are highly configuration-dependent. Memory pooling optimizations are most effective when allocation overhead dominates (smaller networks, fewer neurons), but provide minimal or negative benefit when computation dominates (larger networks, more neurons, deeper architectures).

Practical implications: The effective range of memory pooling optimizations is bounded by the relative cost of allocation versus computation. Profiling specific configurations is essential before applying these optimizations, as benefits observed in one configuration do not reliably transfer to others.

A. Future Work

Future work could explore shared memory tiling and tensor core operations to reduce computation time. Evaluating the

streams strategy on GPUs with multiple copy engines (e.g., A100, H100) would clarify whether hardware constraints fully account for the limited gains observed here. An adaptive tile size mechanism could address the performance degradation at 1024 neurons. Testing these strategies on real-world datasets and production workloads would provide a more complete picture of their applicability.

Acknowledgments

This work builds upon the research by Brouthen Kamel and Akeb Abdelaziz, whose baseline CUDA implementation and thorough documentation enabled our investigation.

We thank Professor Dr. Amina Selma Haichour, our High Performance Computing instructor, for her guidance throughout this project.

References

[1] Brouthen, K., & Akeb, A. (2024). Exploring parallelization of shallow neural network using CUDA.

[2] Haichour, S. A. (2024). High Performance Computing (HPC). Course lectures. Ecole Nationale Supérieure d'Informatique (ESI), Algiers.

[3] NVIDIA Corporation. (2024). CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/