Architectures and Platforms for Artificial Intelligence Module 1 Project

Parsa Mastouri Kashani
University of Bologna
parsa.mastouri@studio.unibo.it

September 26, 2025

1 Introduction

Neural networks form the foundation of modern artificial intelligence. This report explores the parallelization of a specific class of neural networks in which each layer performs **1D stencil operations**. Parallelization is implemented on both CPU and GPU platforms using **OpenMP** and **CUDA**. The primary goal is not to achieve maximum computational performance, but rather to present a clear and comprehensible implementation that demonstrates how to effectively leverage available parallel hardware.

Following the project description on Virtuale, we assume a small stencil width R. In our implementation, we use $R = 3$. Here, R denotes the number of neighboring neurons from the previous layer that contribute to the activation of a neuron in the current layer.

2 The Core of the Task

Consider a neural network layer computed via an R-neuron stencil with bias and activation function. For the i-th neuron in the current layer:

$$y_i = \phi\left(\sum_{k=-\lfloor (R-1)/2 \rfloor}^{\lceil (R-1)/2 \rceil} W_{i,k}\, x_{i+k} + b_i \right)$$

where:

- x_{i+k} are the R neighboring activations in the previous layer,
- $W_{i,k}$ are neuron-specific stencil weights,
- b_i is the bias for neuron i,
- $\phi(\cdot)$ is a nonlinear activation function (sigmoid),
- y_i is the output of neuron i in the current layer.

This formulation generalizes the classical 1D stencil operation to arbitrary R, supports neuron-specific weights, and introduces nonlinearity via ϕ. All neurons within a layer can be computed independently, making the computation **embarrassingly parallel**.

However, layers must be computed sequentially: the activations of layer l must be available before layer $l+1$ can be computed. Thus, parallelization can only be applied **within a layer**, not across layers.

For simplicity, we process only one input sample at a time. In practice, batching inputs would allow multiple forward passes to be executed in parallel (since different samples do not depend on each other, aside from sharing weights). This was not implemented here, in line with the project requirements.

Since we were free to choose R, I selected $R = 3$, as suggested in the project description. With such a small stencil, parallelizing the computation of a single neuron across R threads would introduce overhead that outweighs any potential benefit. Instead, each thread computes one neuron entirely, which is both simpler and more efficient.

3 OpenMP Implementation

In the first iteration of the OpenMP implementation, I defined a struct containing four components:

1. Two integers, N and K,
2. A 1D array for layer sizes,
3. A 2D array for neurons (indexed by layer and neuron),
4. A 3D array of weights (indexed by layer, neuron, and weight index).

This version was easy to understand but memory-intensive, as it stored all neurons and weights simultaneously. It also required significant runtime overhead when initializing large networks from the command line.

Update

After completing the CUDA implementation, I revisited the OpenMP version and created **OpenMP v2.0** with several improvements:

- Instead of storing all neuron activations, we only store the previous and current layer. After computing the current layer, we swap pointers, reducing memory usage to that of a single layer.
- The weight matrix was flattened into a 1D array. This improves cache locality and reduces indexing overhead.

Behavior of the OpenMP Loops

Parallelization is applied over the neurons in each layer. Each thread is responsible for one neuron's computation: multiplying the R previous activations by their corresponding weights, summing them, and adding the bias.

We also used `#pragma omp simd`, which can generate SIMD instructions where supported. This provides a modest single-digit percentage speedup on top of thread-level parallelism.

4 CUDA Implementation

In a standard 1D stencil, the input x and output y have the same size, as shown in `stencil1d-shared.cu` from class. In our case, however, the output of each layer is smaller: it has size $N - (R - 1)$ instead of N. For simplicity, no padding was added.

Correctness was verified by using fixed weights and inputs, then comparing the results with expected outputs. As in the stencil example, I swap x and y pointers after each layer. Since the output size decreases at every step, no additional allocations are required.

Parallelization

Each CUDA thread is responsible for computing one output neuron. Thus, each thread performs R multiplications (previous activations times weights), sums the results, and adds the bias.

Use of Shared vs Global Memory

The first kernel relied only on global memory. In a second kernel, I optimized memory access by copying the neuron activations (x) into **shared memory**. This is beneficial because each neuron in the current layer depends on overlapping groups of 3 activations from the previous layer, so shared memory reduces redundant global reads.

For small networks (e.g., 2 layers with 10 neurons each), this produced a visible performance improvement. However, the advantage diminished for larger networks, where memory access patterns and occupancy dominated performance.

I also experimented with loading the weights into shared memory, but as expected, this provided no significant performance benefit.

5 Results

The OpenMP benchmarks were executed on a MacBook Pro 16-inch (2019) equipped with an Intel Core i9-9980HK processor (8 physical cores, 16 threads, base frequency 2.4 GHz, turbo boost up to 5.0 GHz). The system has macOS as the operating system, with compilation performed using Apple Clang (not GCC).

Due to the thermal design of this laptop, sustained high CPU utilization can cause thermal throttling, which may introduce variability in the results. Additionally, the distinction between physical cores and logical threads (via Hyper-Threading) can influence performance scaling.

As the MacBook Pro hardware lacks an NVIDIA GPU, the CUDA benchmarks were performed on Google Colab. The experiments were executed on a virtual machine equipped with an NVIDIA A100 GPU, which served as the computational backend for all CUDA results.

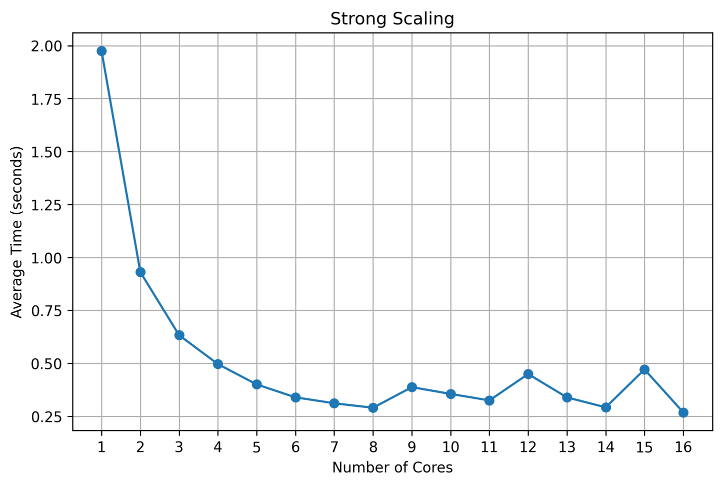


Figure 1: OpenMP Strong-scaling performance N=524288 K=512

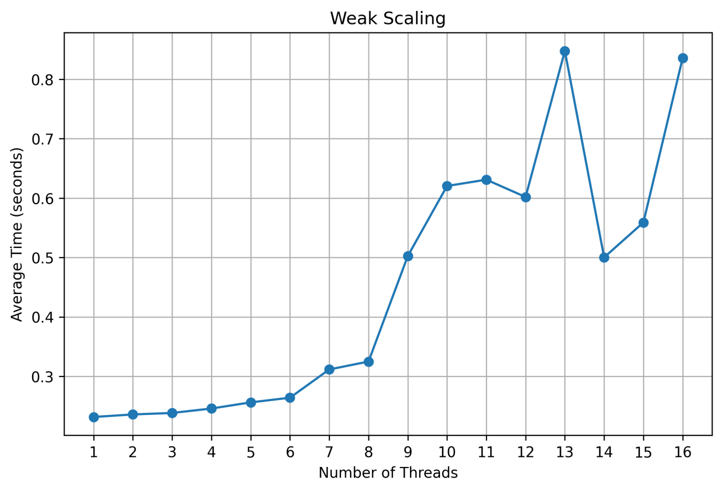


Figure 2: OpenMP Weak-scaling performance N0=65536 K=512

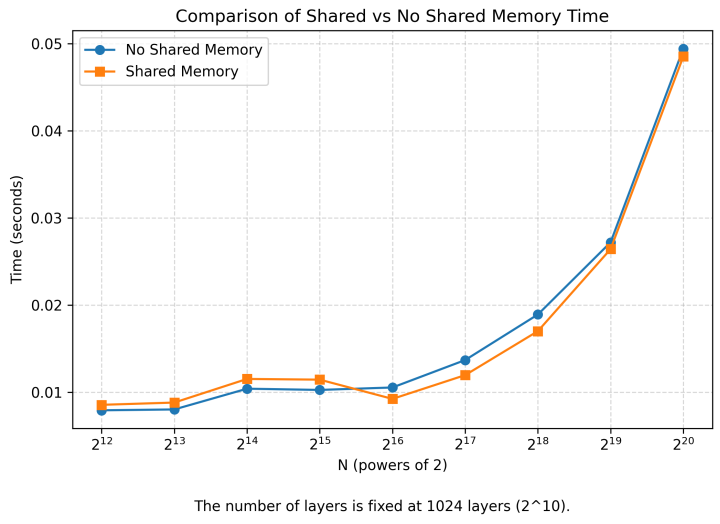
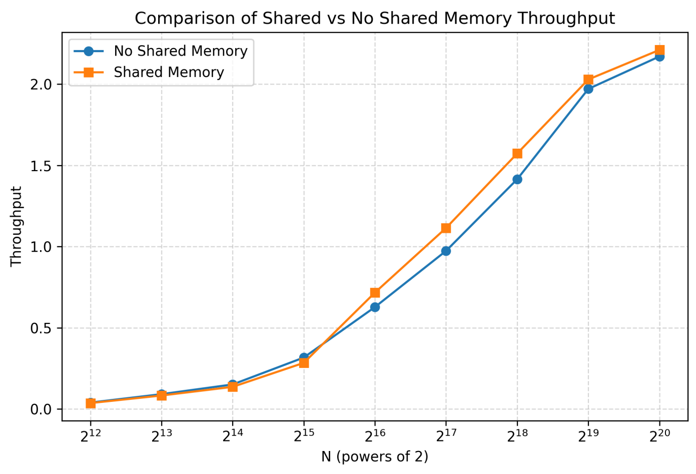
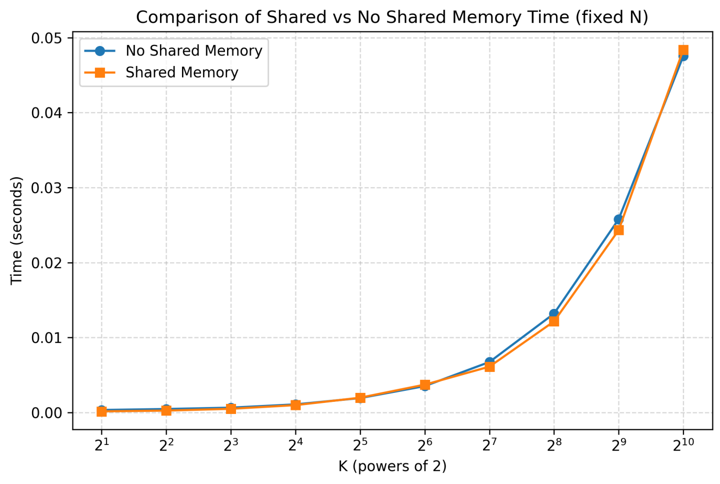


Figure 3: CUDA timing bencharks for K=1024



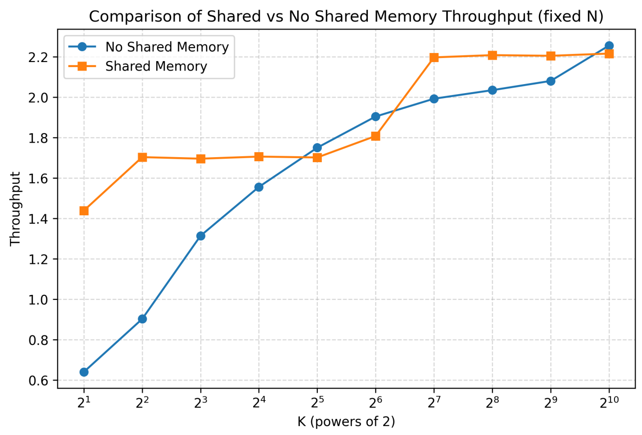
The number of layers is fixed at 1024 layers (2^10). All the throuput numbers are divided by 1e10.

Figure 4: CUDA throughput for K=1024



The number of neurons in the first layer is fixed at 1048576 (2^20).

Figure 5: CUDA timing benchmarks for N=1048576



The number of neurons in the first layer is fixed at 1048576 (2^20). All the throuput numbers are divided by 1e10.

Figure 6: CUDA throughput for N=1048576