Parallelization Strategies for a Multi-Layer Neural Network

Matteo Fasulo
matteo.fasulo@studio.unibo.it

1 Introduction

Deep learning applications increasingly rely on large neural networks. This project focuses on accelerating a multilayer feedforward network composed of 1D locally connected layers, distinct from standard convolutional layers as each neuron possesses unique weights (Figure 1). Our goal is to optimize the forward propagation phase on multi-core CPUs using OpenMP and GPUs using CUDA by exploiting the inherent parallelism within each layer's computation. We evaluate performance by varying input size (N) and layer depth (K), measuring execution time, scalability, and throughput. Code is available at: `https://github.com/MatteoFasulo/Multi-layer-Neural-Network`.

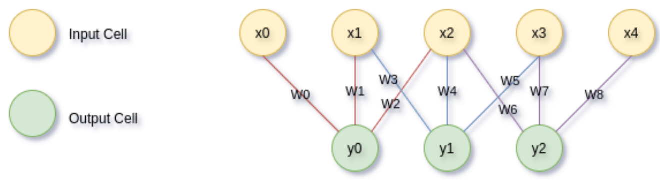


Figure 1: A 1D locally connected layer ($R = 3$). Each output neuron y_i connects to a distinct input window via unique weights W_i.

2 Parallelization Strategy

While layers execute sequentially ($k \to k+1$), computing the output $\mathbf{y}^{(k+1)} \in \mathbb{R}^M$ (where $M = N - R + 1$) from input $\mathbf{x}^{(k)} \in \mathbb{R}^N$ is parallelizable. Each output element $y_i^{(k+1)}$ involves an independent dot product and activation:

$$y_i^{(k+1)} = \sigma\Big(\sum_{r=0}^{R-1} W_{i,r}^{(k)}\, x_{i+r}^{(k)} + b_i^{(k)}\Big), \quad \text{for } i = 0, \dots, M-1.$$

Key challenges are handling redundant reads of shared input elements and ensuring coalesced memory access on GPUs.

2.1 OpenMP Implementation

We parallelize the outer loop ($i = 0, \dots, M-1$) over CPU cores using OpenMP's 'parallel for' with 'static' scheduling, suitable for uniform workloads. The inner dot product loop (sum over R) is manually unrolled to reduce loop overhead and potentially enable compiler optimizations like SIMD vectorization, although the specific impact of SIMD was not explored in detail in this report.

2.2 CUDA Implementation

Each output neuron $y_i^{(k+1)}$ is mapped to a CUDA thread. We use a standard grid-block hierarchy with a large block dimension (`BLKDIM`, e.g., 1024) to maximize occupancy. The number of blocks ensures all M outputs are computed: Blocks = $\lceil M + \texttt{BLKDIM} - 1/\texttt{BLKDIM} \rceil$. Two memory strategies were compared:

1. **Global Memory Access:** Threads directly read required inputs $\mathbf{x}^{(k)}$ and weights $W^{(k)}$ from global memory. Performance relies on coalesced access patterns.

2. **Shared Memory Caching:** Threads within a block collaboratively load a tile of $\mathbf{x}^{(k)}$ (including halo regions for the receptive field R) into fast on-chip shared memory. This reduces global memory traffic by reusing shared input elements but still requires global reads for unique weights $W_{i,\cdot}^{(k)}$.

GPU buffers (input $\mathbf{x}$, weights W, output $\mathbf{y}$) are managed within a single `NeuralNet` struct, allocated contiguously to minimize fragmentation using a centered-stencil indexing (RADIUS = $(R-1)/2$). Layer dimensions are computed dynamically. After each layer's computation, input and output pointers $(\mathbf{x}, \mathbf{y})$ within the struct are swapped, avoiding redundant memory copies. Host-side timers (`hpc_gettime`) measure end-to-end execution time, averaged over multiple trials. Throughput is calculated as total output elements computed across all layers per second.

3 Performance Results

Experiments were conducted on the Giano HPC cluster (8-core/16-thread CPU, NVIDIA L40 GPU). $R = 3$ is used unless stated otherwise. We denote input length by N, number of layers by K, receptive field width by R, and CPU threads by P.

3.1 CPU Scaling

Strong Scaling. We fix the problem size at $N = 2^{20}$, $K = 1000$, $R = 3$ and vary P from 1 to 16. Speedup is defined as

$$\text{Speedup}(P) = \frac{T_1}{T_P},$$

where T_1 is the single-threaded runtime and T_P the runtime on P OpenMP threads. Strong scaling (Figure 2) shows near-ideal speedup up to 8 physical cores (max 7.3×), limited by hyperthreading contention beyond that (only 5.3× on 16 threads).

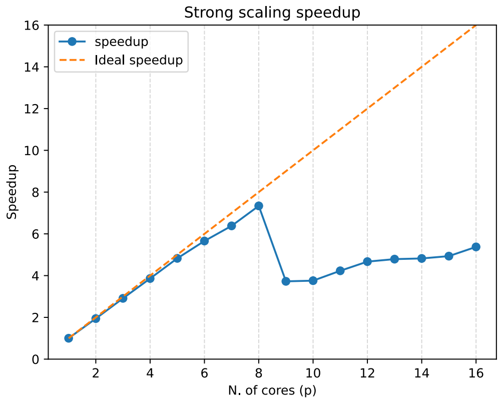
Weak Scaling. For weak scaling, each thread handles a fixed subproblem of size $N/P = 2^{20}/P$ over $K = 1000$ layers. Efficiency is

$$E_{\text{weak}}(P) = \frac{T_1}{T_P},$$

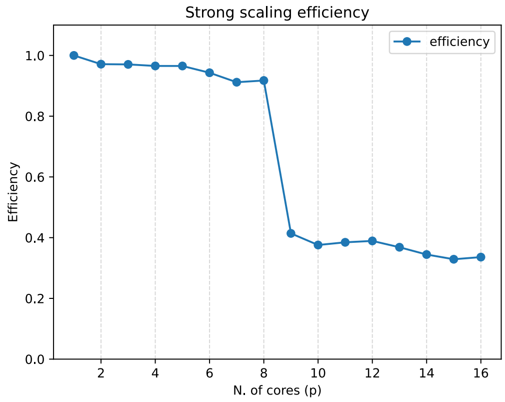
where T_P measures the time to solve P subproblems in parallel. Weak scaling efficiency (Figure 3) remains high ($> 90\%$) up to 8 cores but drops significantly when relying on logical cores, confirming memory/cache saturation limits.

3.2 GPU Performance

We compare CUDA global and shared memory approaches against the 8-core OpenMP baseline.



(a) Strong scaling speedup



(b) Strong scaling efficiency

Figure 2: OpenMP strong scaling on CPU ($N = 2^{20}, K = 1000$). (a) Speedup vs. 1 thread. (b) Efficiency = Speedup / Threads.

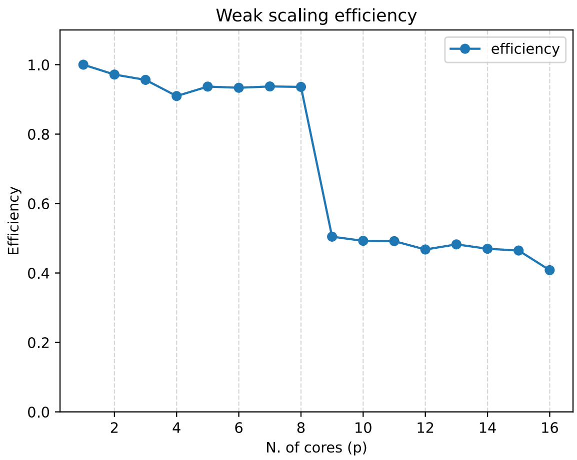


Figure 3: OpenMP weak scaling efficiency on CPU threads ($K = 1000$).

Execution Time and Throughput. Figure 4 shows execution time scaling linearly with input size N and layer count K. Shared memory consistently outperforms global memory due to efficient data reuse via on-chip memory, with the benefit increasing for larger N. We define throughput as

$$\Theta = \frac{\sum_{t=1}^{K-1} \left[N - t\,(R-1) \right]}{T} \quad \left[\text{M elements/sec} \right],$$

i.e. total number of output elements computed across all layers divided by runtime T, expressed in millions of elements per second. Correspondingly, Figure 5 shows throughput increasing with N, indicating better GPU utilization. Shared memory achieves significantly higher throughput by reducing global memory latency and bandwidth usage. Throughput is relatively stable across different K for a fixed N.

Relative Speedup. Figure 6 presents the speedup of CUDA implementations relative to the 8-core OpenMP baseline. Speedup generally increases with N, showcasing the GPU's advantage for larger problems. Shared memory (green bars) consistently provides higher speedup than global memory (blue bars). For the largest input size ($N = 2^{21} \approx 2.1 \times 10^6$), shared memory achieves impressive speedups up to 354× (for $K = 4000$). Increasing the number of layers K

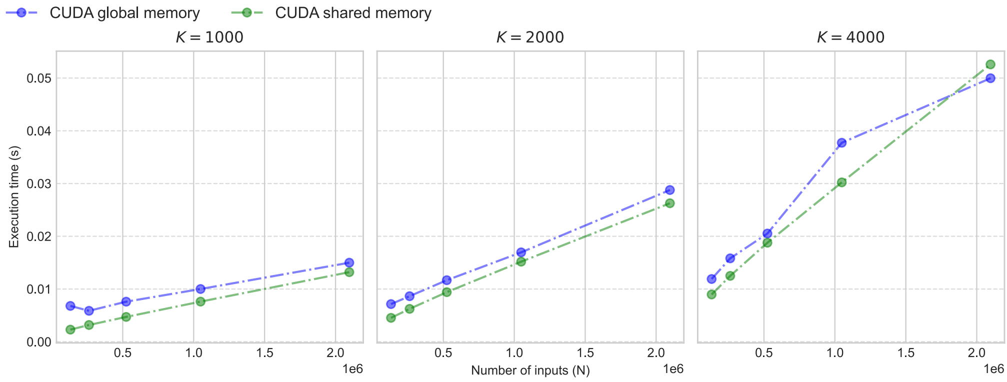


Figure 4: Execution time vs. N for CUDA global and shared memory across different K.

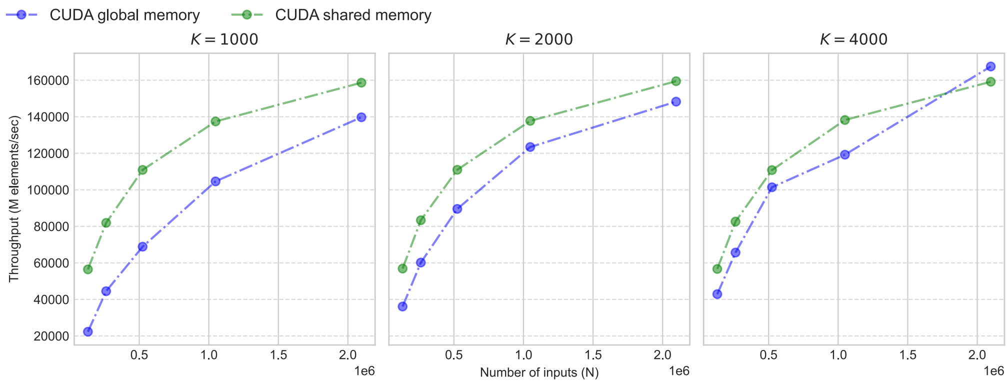


Figure 5: GPU throughput (M elements/sec) vs. N for CUDA global and shared memory across different K.

also tends to improve speedup by amortizing kernel launch overheads, although the gains may lessen for very deep networks (compare $K = 2000$ vs $K = 4000$ panels). At $N = 2^{20}$ (approx 1.05×10^6), the speedups are $295\times$ ($K = 1000$), $342\times$ ($K = 2000$), and $333\times$ ($K = 4000$) for shared memory.

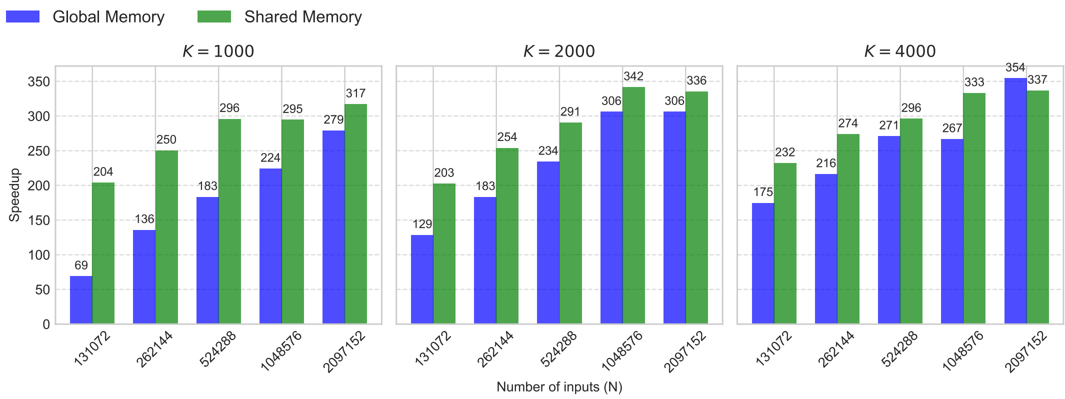


Figure 6: Speedup of CUDA kernels relative to OpenMP (8 threads) baseline, varying N and K. Speedup values annotated.

Appendix: Sliding-Window Reformulation

Given an input vector $\mathbf{x} = [x_0, x_1, \ldots, x_{N-1}]$ of dimension N and a receptive field size R, the output vector $\mathbf{y}$ has a dimension $M = N - R + 1$. For an example case where $N = 5$ and $R = 3$, the output dimension is $M = 3$. The first two output elements (y_0, y_1) are computed individually as:

$$y_0 = \sigma\big(x_0 W_{0,0} + x_1 W_{0,1} + x_2 W_{0,2} + b_0\big),$$
$$y_1 = \sigma\big(x_1 W_{1,0} + x_2 W_{1,1} + x_3 W_{1,2} + b_1\big).$$

Here, $W_{i,j}$ represents the weight connecting the j-th input in the window for output i to the i-th output neuron, and b_i is the bias for the i-th output neuron. Weights are generally *not* shared between different output neurons in this locally connected formulation.

To potentially improve data locality and leverage matrix operations, one can construct an intermediate matrix X by applying a sliding window to the input vector $\mathbf{x}$ (often called 'im2col' or 'unfold'):

$$X = \begin{bmatrix} x_0 & x_1 & \cdots & x_{R-1} \\ x_1 & x_2 & \cdots & x_R \\ \vdots & \vdots & \ddots & \vdots \\ x_{M-1} & x_M & \cdots & x_{N-1} \end{bmatrix} \in \mathbb{R}^{M \times R}.$$

Let $W \in \mathbb{R}^{M \times R}$ be the weight matrix where row i contains the weights for output y_i, and let $\mathbf{b} \in \mathbb{R}^M$ be the bias vector. The entire output vector $\mathbf{y} \in \mathbb{R}^M$ can then be expressed as:

$$\mathbf{y} = \sigma\big(\operatorname{diag}(XW^\top) + \mathbf{b}\big).$$

In this equation, $\operatorname{diag}(\cdot)$ extracts the main diagonal of the $M \times M$ matrix product $XW^\top$, yielding the required weighted sums.

Although this sliding-window ('im2col') approach enables potentially fully coalesced memory reads and leverages highly optimized Basic Linear Algebra Subprograms (BLAS) for the internal matrix–matrix product ($XW^\top$), it introduces complexity and overhead. Firstly, explicit transformation of the input activations of each layer into the matrix X can be costly, potentially compensating for the gains in BLAS performance, especially for small R or deep networks requiring repeated transformations. Secondly, it's crucial to note that even after performing the computation via $XW^\top$, the desired result is the vector $\mathbf{y}$ obtained from the *diagonal* (plus bias and activation). This output vector $\mathbf{y}$ is analogous in structure to the original input vector $\mathbf{x}$; it is **not** automatically in a matrix format (like X) suitable for direct use as an operand in a subsequent layer's computation if that layer also relies on a GEMM-based strategy (e.g., another 'im2col' followed by matrix multiplication). Such a subsequent layer would require its *own* 'im2col' transformation applied to $\mathbf{y}$. Consequently, considering the transformation overhead and the fact that the output requires further processing for subsequent GEMM-based layers, the direct, per-neuron formulation might be preferred for clarity, simplicity, and potentially better end-to-end performance in certain scenarios.