APAI Module 1 - Project Report

Luca Tedeschini - 0001120155

luca.tedeschini3@studio.unibo.it

April 2025

1 Introduction

The final project for *Architectures and Platforms for Artificial Intelligence (APAI)* consists of exploring parallelization strategies to speed up the computation of a specific instance of a Feed-Forward Neural Network (FFNN). The software artifact must be implemented in both *C with OpenMP* and *CUDA*. In this report, I will explain my approach to the assignment, the design choices made during implementation, and the performance results obtained.

2 Project Description

The assignment specifications require the implementation of a sparsely connected feed-forward neural network. Given N as the number of nodes in the input layer, R as a constant representing connectivity reach, K as the total number of layers, W as the weight matrix, $f(x)$ as the sigmoid activation function, and b as a bias value, we construct a neural network where the $(i+1)$-th layer is computed using the formula:

$$x_{i+1} = f\left(\sum_{r=0}^{R-1} x_{i+r} \cdot W_{i,r} + b\right)$$

The constant R is assumed to be small, and the weight matrix W is shared between nodes. This particular architecture resembles a 1D convolution operation more than a standard feed-forward neural network (due to weight sharing), but for simplicity and to leverage established programming patterns, I will treat it as a sparsely connected neural network throughout the implementation.

3 OpenMP

3.1 Introduction

OpenMP provides a set of Application Programming Interfaces (APIs) that enable multi-platform shared-memory parallel programming. This framework allows source code to run on multiple hardware platforms without modification, provided they support the OpenMP APIs. For this assignment, I used the C implementation of OpenMP. The APIs are implemented as C directives that can be automatically ignored when compiling without OpenMP support, ensuring cross-platform compatibility. To compile a C program with OpenMP support using the GNU C Compiler (GCC), the '-fopenmp' flag must be included.

3.2 Code structure

To maintain a clean implementation, the code is organized into three logical components: *main.c*, containing the program entry point and main function; *network.c*, housing all functions related to network creation and management; and *utilities.c*, providing various utility functions.

The network is implemented as a composition of C *struct* components: the *Network* struct contains a pointer to an array of *Layers*, which contains an array of *Nodes* that store individual node values. Additionally, the *Network* struct maintains the weight matrix W for convenient access. Figure **??** illustrates this structure.

In *main.c*, the forward pass implementation consists of three nested for loops: the outer loop iterates through the layers, the middle loop cycles through the current layer's nodes, and the inner loop processes the previous R layers' nodes to compute each node's value.

3.3 Parallelization strategies

Since computation across different layers cannot be easily parallelized due to dependencies, the chosen parallelization strategy focuses on the middle for loop to distribute computations across threads within each layer. As threads don't need to write to the same memory locations, this problem qualifies as *embarrassingly parallel*. The implementation uses the *pragma omp parallel for* directive to automatically distribute the computational workload among threads.

Each output node operates independently of others, requiring synchronization only between layers. It's worth noting that the program does not implement sophisticated memory management, freeing memory only after complete execution. This approach limits the maximum input size to $N = 2^{21}$ and $K = 1000$. Future improvements could include more efficient memory management, though the current maximum size is adequate for CPU-based neural network execution.

3.4 Performance

Performance evaluation was conducted on the University of Bologna's HPC cluster. Both OpenMP and CUDA implementations were benchmarked on the L40 partition. The evaluation focused on strong and weak scaling. For strong scaling, a fixed problem size ($N = 2^{20}$, $K = 1000$) was used while varying the number of OpenMP threads. For weak scaling, the problem size per thread was kept constant ($N = 2^{16} \times n_{\text{threads}}$, $K = 1000$), thus scaling the total problem size with the thread count n_{threads}. As depicted in Figure 4, strong scaling performance is near-optimal up to 8 threads, which corresponds to the number of physical CPU cores on the cluster nodes. Beyond this threshold, performance diminishes due to the utilization of logical cores (hyper-threading), an expected outcome. Weak scaling analysis reveals an efficiency of approximately 78% when utilizing 8 threads. This is likely attributable to the chosen memory management strategy, which prioritizes logical clarity and readability over raw performance optimization.

4 CUDA

4.1 Introduction

CUDA is a proprietary parallel computing platform and application programming interface (API) that enables software to leverage supported GPUs for accelerated, parallel, and general-purpose computation.[1] In Deep Learning applications, where high parallelization is essential for reasonable computation times, GPUs offer significant advantages due to their architectural design. They are built to process numerous parallel instructions simultaneously, a capability derived from their primary function of rendering real-time graphics. This is achieved through a large number of processors, though with lower individual processor performance compared to CPUs. This approach to utilizing GPUs for non-graphical tasks is termed General-Purpose computing on Graphics Processing Units (GPGPU) and represents one of the key technological enablers of modern Artificial Intelligence systems.

4.2 Code structure

The code implementation follows a straightforward design: the source file defines two kernels that accept identical inputs and produce the same outputs. The principal difference lies in memory usage: one kernel leverages shared memory while the other relies solely on global memory. The user can select which kernel to execute, with execution times reported to stdout for performance comparison.

Unlike the OpenMP implementation that used higher-level structures, the CUDA version implements the network using one-dimensional arrays to optimize memory access speed. While this approach necessitates more careful index management, the code includes comprehensive comments explaining the indexing strategy.

4.3 Parallelization strategies

Both implemented kernels employ a double-buffering strategy to optimize the forward pass computation. This technique operates by alternating the roles of the *current_layer* and *next_layer* arrays in each iteration. This alternation is achieved by simply swapping the pointers to their respective memory locations. Since the array designated as *next_layer* will then contain the results computed in the previous iteration (which become the inputs for the current iteration), the core computational code remains unchanged after the pointer swap, allowing the propagation to proceed correctly. A key advantage of this strategy is that memory for these layer arrays needs to be allocated only once, prior to the commencement of the first iteration. Figure 2 provides a visual representation of this double-buffering approach.

[1] CUDA - Wikipedia

Furthermore, the weight matrix is flattened into a one-dimensional array. To guarantee coalesced global memory access, a specific indexing scheme for this flattened weight array was implemented. Efficient global memory reads require that threads executing in parallel access memory addresses that are close to each other. Given that threads within a CUDA warp execute the same instruction at any given moment (SIMT architecture), the weight matrix elements were laid out to ensure that for each step of the inner loop within the kernel, the memory locations accessed by concurrent threads in a warp are contiguous. Figure **??** illustrates this memory access pattern.

Although computationally efficient, this double-buffering and static weight allocation strategy can be suboptimal in terms of memory utilization. This suboptimality is especially pronounced for the weight matrix: at each iteration, approximately $R \times (R/2)$ values becomes unused for those narrower layers, yet remains allocated. Given that R in the previous formula scales quadratically, slightly increasing R leads to a much bigger memory waste. However, for the scope of this work, R is assumed to be relatively small. Future improvements could explore dynamic memory allocation or deallocation strategies to address this limitation and enhance memory efficiency. For this case study, the prioritization was on maximizing computational performance over optimizing memory management.

The shared memory kernel variant differs from the non-shared memory (global memory only) version primarily in its utilization of shared memory for intermediate computations. Specifically, it caches the *current_layer* activation values into shared memory before they are used in calculations with the weights. Storing the entire weight matrix in shared memory was not feasible due to its potential size relative to the limited capacity of shared memory per Streaming Multiprocessor (SM). For example, considering a typical shared memory size of 48KB per SM, the weight matrix size, calculated as $\frac{N \times R \times \text{sizeof(float)}}{1024}$ KB (assuming `sizeof(float)` is 4 bytes), imposes constraints. If we fix R to 100, the maximum value that N can assume to allow the weight matrix W to fit entirely within 48KB of shared memory is approximately 122 (since $(122 \times 100 \times 4)/1024 \approx 47.66$ KB).

Interestingly, the non-shared memory kernel demonstrated slightly better performance, in terms of both computational time and throughput, compared to the shared memory version, albeit by a small margin. This occurred despite the conventional expectation that shared memory usage should yield speedups due to its lower latency compared to global memory.

The precise reasons for this counter-intuitive result were not investigated further in this study, as both kernels achieved satisfactory performance levels meeting the project's objectives. Potential factors could include the overhead associated with loading data into shared memory outweighing the benefits for the specific problem size, or perhaps memory bank conflicts in shared memory.

4.4 Performance

For performance evaluation, both execution time and throughput (defined as the number of network nodes processed per second) were measured. The results are presented in Figure 5. Notably, the kernel implementation without shared memory outperforms the shared memory version in both metrics. The following section will provide a comparative analysis of OpenMP and CUDA results, graphically demonstrating the superior performance of GPUs for parallel computation.

5 Final results

As illustrated in Figure 6, the CUDA implementation unequivocally outperforms the OpenMP program by a substantial margin across all tested configurations. When comparing single-core OpenMP performance against the CUDA kernel utilizing shared memory, a remarkable speedup of approximately 1971× is observed. Even when OpenMP leverages its maximum tested parallelism with 8 threads, the shared memory CUDA version still demonstrates a pronounced advantage, achieving a speedup of 246×. For the non-shared memory CUDA kernel, the speedups, while comparatively lower than its shared memory counterpart, remain exceptionally significant. Against a single OpenMP thread, this version yields a speedup of 1362×, and when compared to the 8-thread OpenMP execution, the speedup is 170×. A discernible trend is also evident from the results: the performance disparity, and thus the speedup achieved by CUDA, tends to increase with larger input problem sizes (N). Unfortunately, further exploration of this trend with even larger N values for the OpenMP version was impeded. Attempts to run the OpenMP program with increased N values consistently resulted in segmentation faults, a limitation likely stemming from the intricacies of its memory management approach when handling extensive datasets. This constraint prevented a more comprehensive comparison at the upper limits of input size.

6 Images

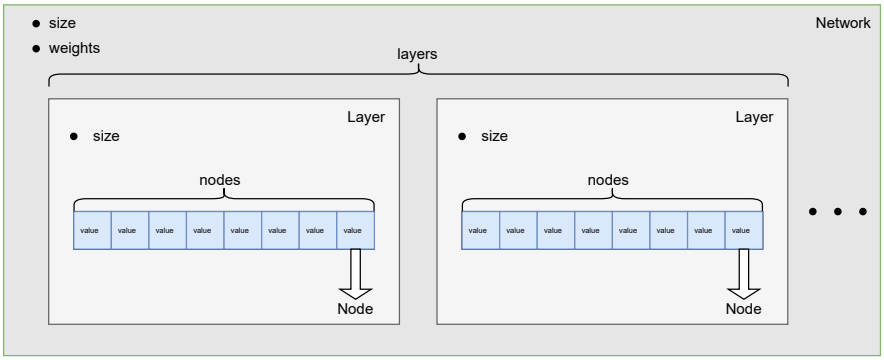

Figure 1: Hierarchical data structure for the OpenMP network implementation, illustrating the ‘Network‘ object composed of ‘Layer‘ objects, which in turn contain ‘Node‘ objects.

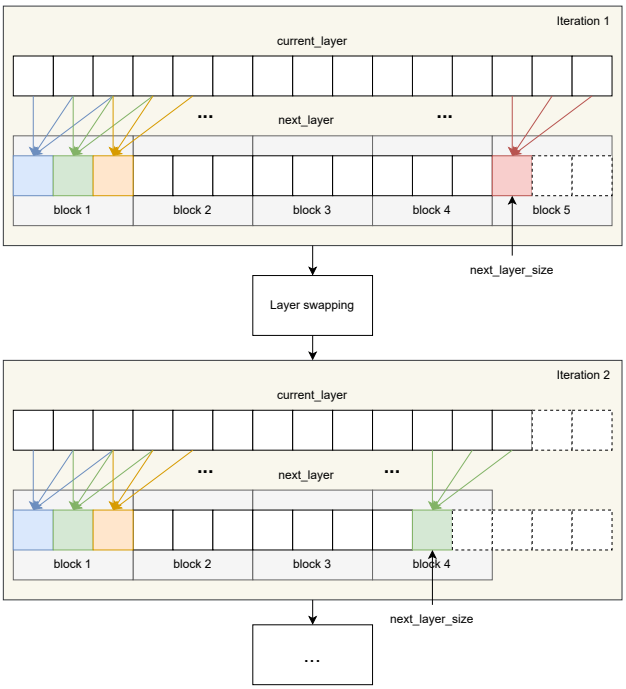

Figure 2: Double buffering strategy employed in the CUDA forward pass. This technique, utilized between network layers, aims to minimize memory allocation overhead during computation.

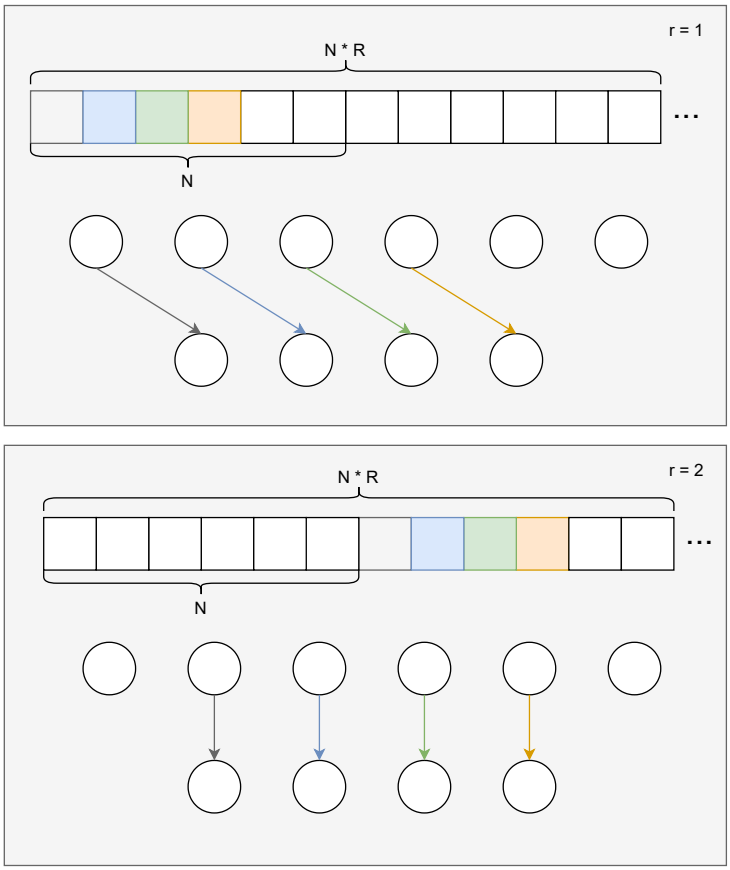

Figure 3: Visualization of the weight matrix access pattern implemented by CUDA threads during the neural network's forward pass.

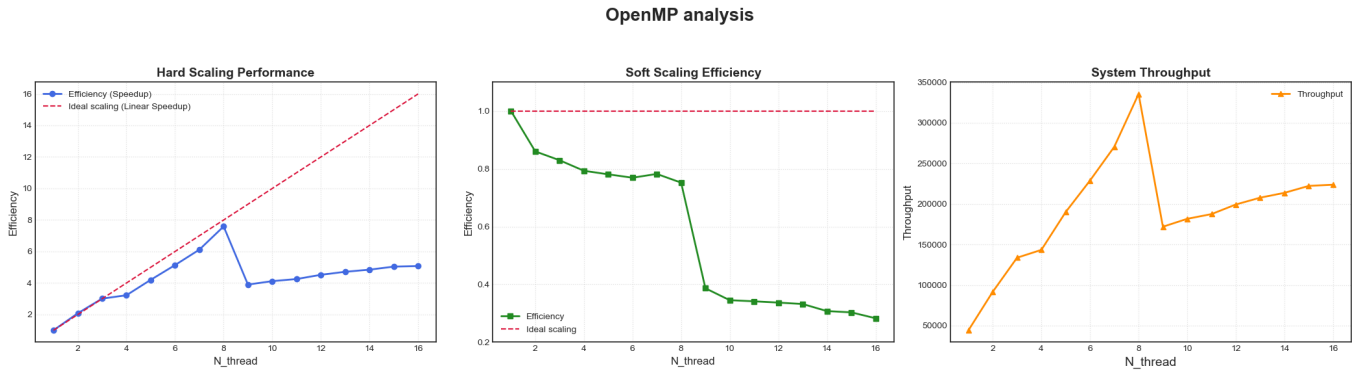

Figure 4: OpenMP parallel performance: strong scaling speedup, weak scaling efficiency and throughput (nodes/second) plotted against the number of utilized threads with N fixed to 2^{20}.

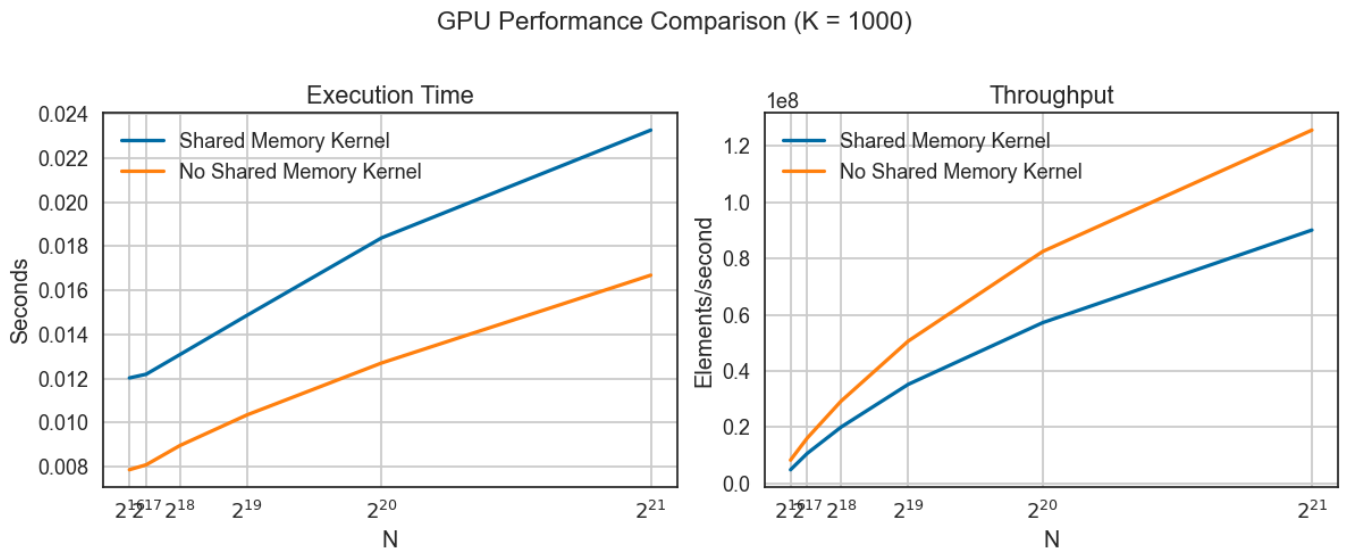

Figure 5: Throughput (nodes/second) comparison between CUDA kernels utilizing shared memory versus those without, evaluated across varying input sizes.

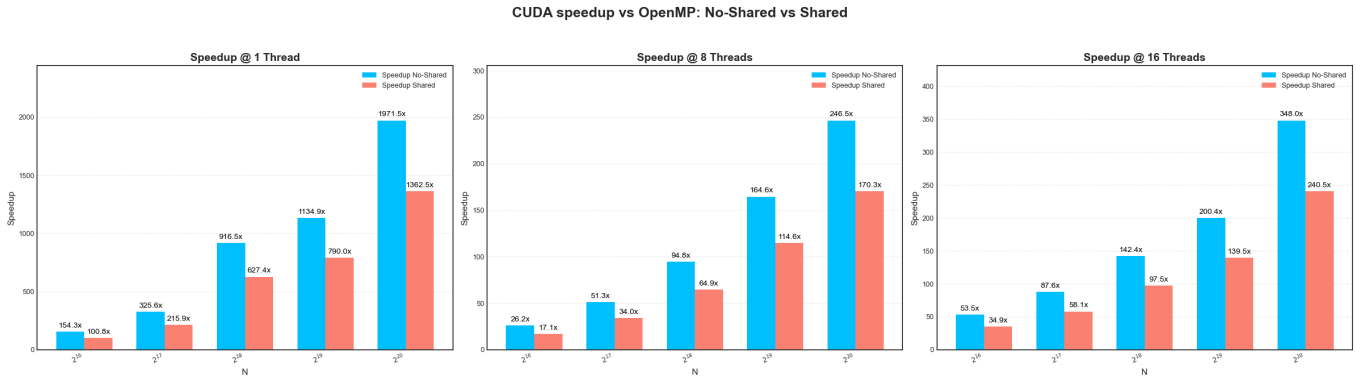

Figure 6: Speedup achieved by CUDA implementations (with and without shared memory) relative to the OpenMP version, presented for different input problem sizes.