Architectures and Platforms for Artificial Intelligence - module 1

multi-layer Neural Network

Davide Sonno, davide.sonno@studio.unibo.it
Student ID: 0001147866

1 Introduction

In this report, I will discuss the design choices used to gain the most advantages from parallel architectures, namely OpenMP and CUDA.

Feed Forward Neural Networks are a widely used model in machine learning in which each output value is computed independently of the others. This property allows for many parallelization strategies on multi-core processors or GPUs.

The source code is available in the following repository.

2 Implementation Details

For the network, R is fixed to 29, to introduce enough complexity for the performance evaluation. The input and weights are `float` values, generated uniformly from $[-1, 1]$. The bias is fixed at 0.005 for all layers. The weights matrix is initially stored in memory as a contiguous array. To compute the output value y_i, the input elements $x_i, ..., x_{R-1}$ and the weights $w_{i,0}, ..., w_{i,R-1}$ are used. The network is evaluated by repeatedly applying the same function/kernel to all the layers. Each layer computation is independent, and the focus is on optimizing a single output layer.

In both approaches, the times were measured using the utility function provided in the header file `hpc.h`.

3 OpenMP

The sequential program computes each output value by accessing R input values and multiplying them by their corresponding weights.

To split the workload across the cores, output values can be assigned contiguously to each core, or alternated between them. By having to store output values contiguously, alternating cores is not a good idea because it would introduce race conditions. Instead, if each core processes a chunk of **y** values, they can work freely without conflicts.

The cores could also be used together for each output computation. However, this strategy introduces necessary synchronizations to sum the values, instead of being able to process each computation independently.

3.1 Functions

The proposed scheduling distributes the cores in a static way, ensuring that each core computes as many contiguous elements as possible. Each core declares a private variable `sum`, which is updated during computation. The input and weight variables, along with the input size, can be shared because they are read-only. The output array can also be shared because each output value is only written once by one thread. Each weight is read only once; each output value is only stored once and the adjacent elements are processed by the same core; input values are read, at most, R times by the same core, except for the very last assigned elements that can be read by two cores. In practice, this last scenario might only cause a bank conflict if the input is really small; in every other case, the two cores are not going to read those values at the same time. These elements are the last accessed by a core, but the first read by the following one.

As a final consideration, the weighted sum computation can be manually unrolled to a factor of 4 or 8, reducing loop overhead and improving instruction-level parallelism.

4 CUDA

4.1 Kernels

The proposed kernels follow this structure: within a block, each thread computes one output value. The main idea is that each thread loads the values needed for its computation and stores the result in the output array **y**.

The block dimension used is 1024, the maximum available in the GPU used. Given that each block processes BLKDIM=1024 output values, we will need $\lceil \frac{N-R+1}{BLKDIM} \rceil$ total blocks. Two indexes are assigned to each thread: `local_index` of the thread inside the block, `global_index` of the thread across all the threads. Each thread with a global index less than the output size will compute the corresponding output value.

Kernels are introduced in an ordered way, reflecting the process of developing them step by step.

4.1.1 Global x and w, 1D Stencil: First of all, we start with a simple stencil kernel, performing the sum of R input values, without multiplying them by the weights. By doing so, we will be able to see the impact that accessing the weights will have.

4.1.2 Global x and w: The sum from the previous kernel is now a weighted sum. This kernel is valid for the network evaluation previously described.

Introducing the weights, the kernel is about 30 times slower than the stencil. Note that the new complexity is not caused by the multiplication itself, but by the memory accesses required to read the weights.

The memory access pattern are shown in Figure 1

4.1.3 Global x and w, transposed weights: In the previous kernel, the accesses are not coalesced: consecutive memory locations are reached by the same thread. To overcome this problem we can consider the matrix as an $R \times N$ matrix.

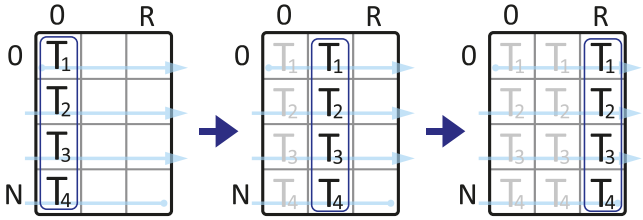

Figure 1: Accesses to the $N \times R$ weights matrix.

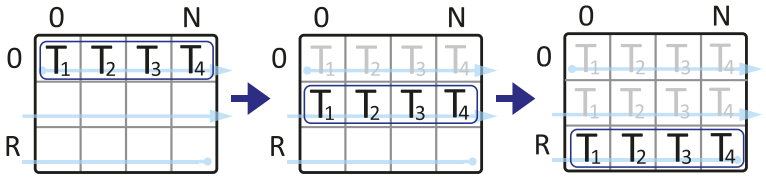

Figure 2: Accesses to the $R \times N$ weights matrix.

With the new pattern shown in Figure 2, consecutive threads access consecutive memory locations, acquiring coalesced accesses. As a result, we achieve a great speed-up compared to the previous kernel, soon reported in the results.

4.1.4 Shared x, global w, transposed weights: This kernel exploits the benefits of the shared memory to efficiently access the input values multiple times.

Part of the input layer is copied from the global memory to the shared array `x_shared`. It needs to store enough values to compute BLKDIM outputs, so BLKDIM $+ R - 1$.

Each thread copies from `x[global_index]` to `x_shared[local_index]` whenever `global_index < input_size`. The first $R - 1$ threads also copy that number of remaining values. The weights accesses are coalesced.

4.1.5 Shared x, global w, read-only transposed weights: To speed up access to the weights, this kernel informs the compiler that the weights are read-only, hopefully allowing for optimized memory access. To achieve this, the weights are marked with `__restrict__` and accessed using `__ldg(*w[i])`.

4.1.6 Global x and w, structured weights: Another way to enhance the global memory accesses is to structure the variables as a structure of array. In particular, our $N \times R$ matrix **w** will be structured as R arrays of length N. By doing so, we have once again coalesced accesses, as shown in Figure 3.

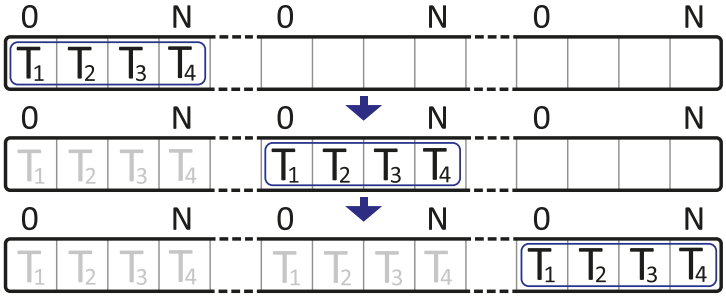

Figure 3: Accesses to the structured weights. T_i is the thread that accesses that memory location.

If the arrays are stored contiguously, we have exactly the case of transposed weights kernel. The only difference would be having to work with R different pointers, one for each array, instead of a singular pointer to the whole matrix.

4.1.7 Shared x, global w, structured weights: We once again move part of the input to the shared memory, because it is accessed multiple times. The weights are stored in the structure of arrays introduced with the previous kernel, achieving coalesced accesses.

5 Performance Evaluation

5.1 OpenMP

The machine used for the experiments has eight physical cores, so we expect the performances to increase up to that number of threads. After that, the system creates virtual threads, which are handled by the CPU's scheduler and may not offer the same performance benefits as physical cores, potentially leading to a decrease in efficiency.

5.1.1 Metrics:

- **Strong Scaling Efficiency,** $E(p)$: expresses if adding processors to the computation helps performances.
- **Weak Scaling Efficiency,** $W(p)$: expresses the impact of increasing the problem size while keeping the workload of each processor fixed.

5.1.2 Results: From the graphs results, Figure 4, unrolling the loop that computes the sum speeds up the function. In particular, the average speed-up is 1.5× when unrolling four operations. An unrolling factor of eight performs a little worse, but still better than the unrolled loop.

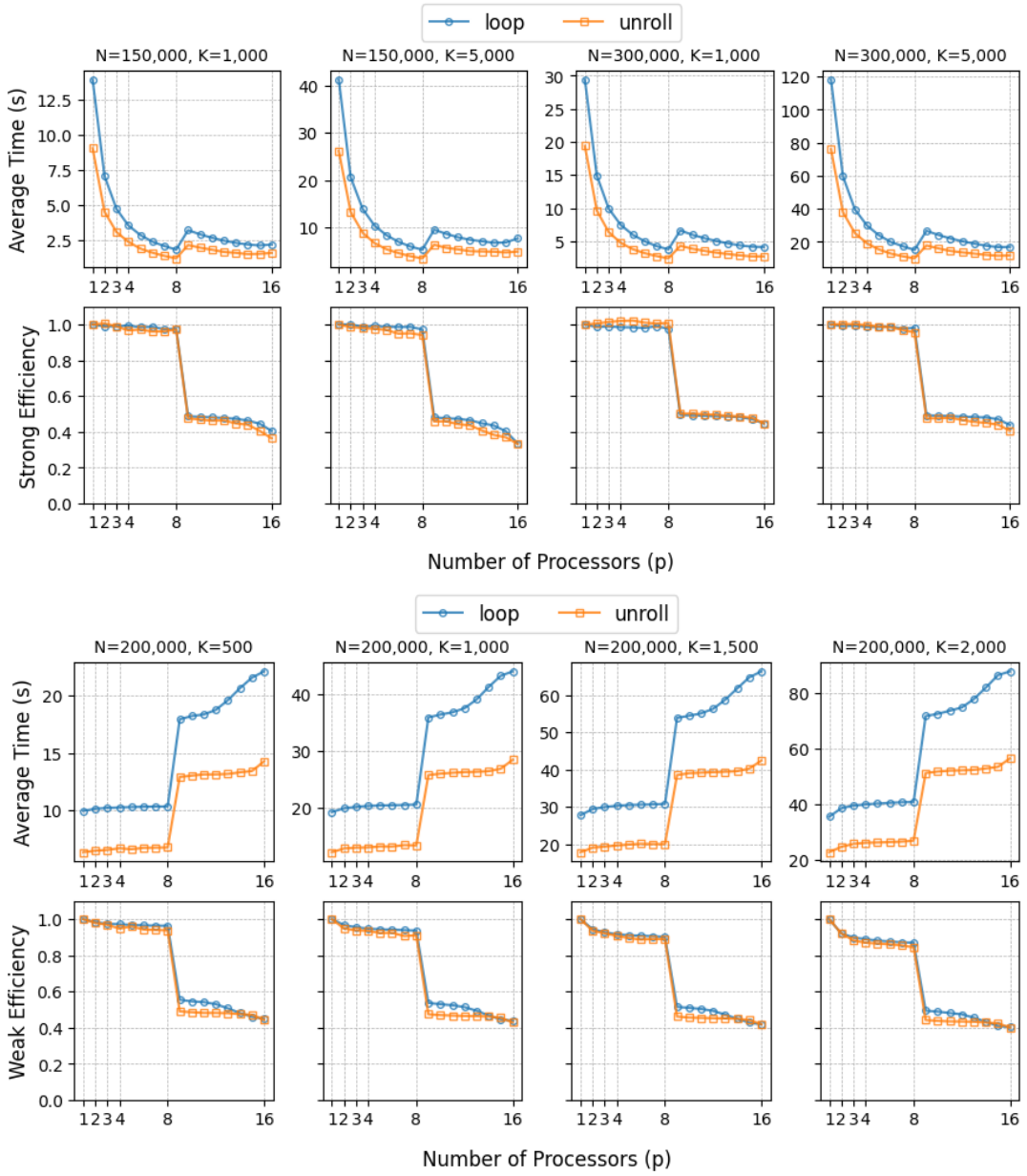

Figure 4: Strong and Weak scaling efficiencies

Both the loop and unrolled versions have nearly identical strong and weak efficiency graphs, characterized by a slowly decreasing behaviour. With eight processors, the average efficiencies are $\overline{E(p)} = 0.97$ and $\overline{W(p)} = 0.90$.

The sudden step that occurs with nine processors was expected and is consistent with the machine running the program. As mentioned, the times got worse after using eight of them.

A high value of the strong scaling metric means that the program scales very well when adding cores, and that the overheads introduced by them are really small, as if they were working independently. A high value of the weak scaling metric means that the program scales well when both the problem size and the number of cores increase. This indicates that the overheads associated with larger problem sizes and additional cores are minimal, allowing the system to handle the increased workload efficiently.

5.2 CUDA

The test cases have been run on an Nvidia *l*40 GPU. Different input dimensions and layer number have been tested with the proposed kernels. Meaningful results have been obtained with input dimension above one million elements, executing progressively more layers until the last possible layer for that input size.

5.2.1 Metric: The main metric used to compare CUDA kernels is throughput: the number of output values, or operations, computed each second by the kernel. In our case, this value is obtained by dividing the total number of output values by the execution time.

The total number of output values, given an input of size N and the network composed of K layers, is the following: $\texttt{num\_outputs} = \sum_{i=1}^{K-1} [N - i(R-1)] = \cdots = (K-1)\left[N - \frac{K(R-1)}{2}\right]$

5.2.2 Results: From Figure 6, we can notice that the kernel with unoptimized global memory accesses, GLBL_MEM, is by far the worst. The other kernels seem to have a really similar throughput. The graphs have an increasing behaviour, stronger with smaller input sizes, and quickly approaching a plateau with larger inputs. This growth may be influenced by the shrinking of the input size at each layer, resulting in less operations needed for the latter layers.

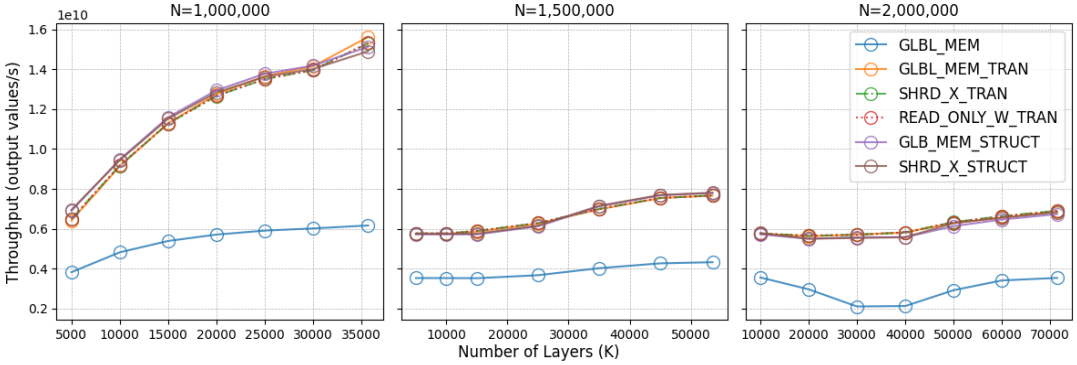

Figure 5: Kernel throughputs with different input sizes, executed with the maximum number of layers.

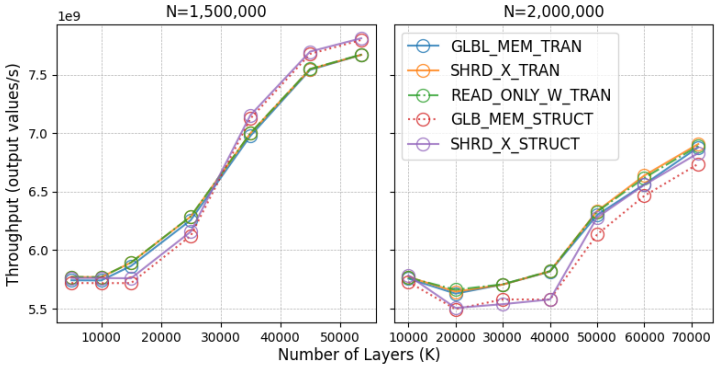

Figure 6: Throughput graphs for input sizes of length $N = 1.5\text{M}, 2\text{M}$

With all input sizes, the starting throughput is roughly the same. For bigger N the benefit mentioned above is not very present. Even if the layers are getting smaller, the computations needed are still more than when having smaller inputs. More computations translates to more weight accesses, the most time consuming operation performed by the kernels. This results in a lot more threads waiting for their accesses to be fulfilled and therefore less time spent computing the output. A demonstration of the impact of the weights is the fact that the stencil kernel is incredibly faster because it does not need to access the weights. A second proof can be found

in the kernels themselves: all of them try to gain the most benefits from the shared memory and the coalesced accesses, but none of them seems to gain much in terms of performance.

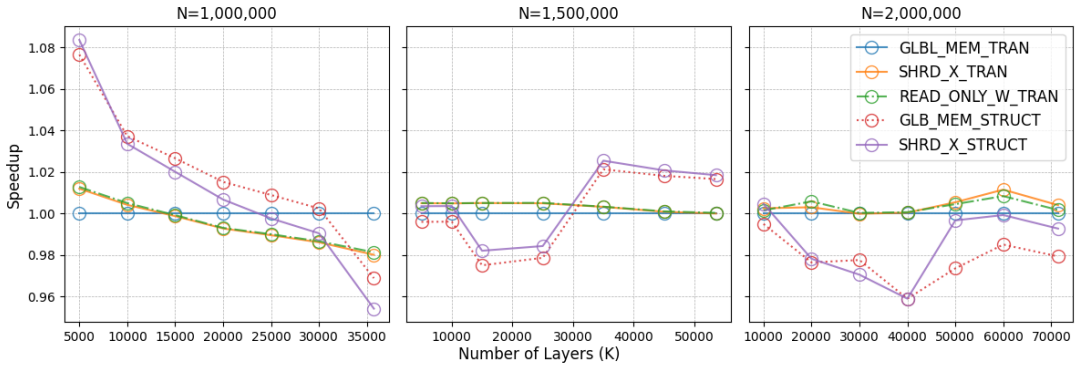

Figure 7: Kernel speed-ups with different input sizes, executed with the maximum number of layers. The baseline for the speed-up is the global memory kernel with transposed weights accesses.

Even if the kernels perform very similarly, we can further inspect the throughputs for the larger inputs, seen in Figure 6, and their speed-up with reference to the kernel that introduced transposed weights accesses, Figure 7. From these graphs, we can notice that the kernels using structured weights start slower than the others, but they can increase their throughput, often surpassing other kernels. They seem to work better with smaller inputs or after many layers. Weights stored in a unique array have progressively larger portions that are not used because the input values they correspond to are no longer part of the input. Structured weights, on the other hand, are separate non-consecutive arrays, and might be optimized better by the compiler for the latter layers. The kernel using read-only accesses has in most cases the same performance as SHRD_X_TRAN. On a side note, with different GPUs, like the *rtx*2080, it performs slightly better than all other kernels except for GLBL_MEM_TRAN, which being also faster than the kernels using shared memory.

5.3 Comparison

N	K	OMP Times	CUDA Times	CUDA Speedup
50000	1000	0.969120	0.009254	104.724x
100000	1000	2.320956	0.025686	90.3588x
300000	1000	7.252310	0.070827	102.395x

Table 1: Comparison of OMP and CUDA times, run on the *rtx2080* partition. As expected, the CUDA kernels outperform the multi-core program.

6 Conclusion

The OpenMP approach achieved great scalability over the input size and number of kernels, shown using the strong and weak scaling efficiency metrics. The parallelization strategy used is quite simple, assigning consecutive portions of the output to each thread. This simple solution can be extended by unrolling some operations to achieve lower execution times and similar scaling efficiencies.

Multiple CUDA kernels have been tested without any outstanding solution. A considerable bottleneck for every kernel is caused by accessing the weights in global memory, which can only be reduced in part by coalescing the accesses. To overcome this obstacle, kernels could be designed to compute the whole network instead of a singular layer. By doing so, the weights can be stored in shared memory and used many times by each block. This would of course introduce some complexity, but the impact of the weights could be much smaller.