# Exploiting OpenMP and CUDA parallelization to enable efficient Neural Network evaluation

Michele Iannello

March 27, 2021

**Abstract**

This report describes two different C code implementations - the former making use of the **OpenMP**[1] API and the latter of **NVIDIA CUDA**[2] - of a simple *feed-forward sparsely-connected multi-layer* **Neural Network**. In particular, the aim of the presented work is to exploit parallelization techniques in order to allow an efficient evaluation of the network and to evaluate the performances through the assessment of different metrics. Results are presented by means of plots realized using Python module **Matplotlib**[3].

## Outline

This project was developed as part of the final assessment for the course *Architecture and Platforms for Artificial Intelligence* within the Master Course in Artificial Intelligence at the University of Bologna. The subject of the project is a feed-forward sparsely-connected multi-layer Neural Network whose structure is described in the assignment document. The development of the work was performed exploiting the *lab machine*, a GPU-equipped computer located at the University and made (remotely) available to the course students thanks to the course professor.

### Executable files

It is useful to firstly provide some brief instructions, in order to correctly run the executable files[1] obtained by compiling the source files `nn_omp.c` and `nn_cuda.cu` according to the specifications contained both within the `README.md` file and the source files themselves. Both programs accept two command line arguments: the size of the input for the first layer ($N$) and the number of layers ($K$) of the network; if no argument is provided, they default to $N = 500000$ and $K = 150$ (which were observed to generate data so as to approach – but not exceed – the memory limit of about 3000 MB of the *lab machine* GPU). At the execution of both programs, the **execution time** is printed to the console. This measure refers to the actual *computation* time – not taking e.g. memory allocation and data transfer into account – and is computed exploiting the function `hpc_gettime()` contained in the header file *hpc.h*, provided by the professor.

### Data generation

The constant `R=5` is `#define`d at the beginning of both source programs, i.e. it is a *compile-time* constant. In the second program, the constant `BLKDIM=1024` is `#define`d as well. Input data is randomly generated at run-time, right before the evaluation of the network, and it is organized

[1]We will refer to the files either as *Program 1/2* or *CPU/GPU implementation*

in $K$ `structs`, each storing inputs (only for the CPU implementation[2]), weights and bias for a different layer. This choice makes the code clearer and easier to read, along with allowing a simpler management of the input data: since each layer has a different number of (inputs and) weights, storing data in few big arrays (instead of many `structs`) would result in non-trivial indexing in order to retrieve the corresponding data, requiring further computations.

#### Functions

The *forward pass* of the network, i.e. the actual computation of the activations, is performed by the repeated call of the function `kernel()`, which is invoked $K$ times inside the `forward()` function. At each iteration, the latter takes care of storing the activations – computed by the former – as input for the following layer (namely in the corresponding `struct` for Program 1). Inside `kernel()`, a nested loop goes firstly through output neurons, initializes each activation to the bias value $b_k$ and then repeatedly updates it – through MAC operations – looping over the corresponding $R$ values of the input neurons.

## 1 CPU implementation

### 1.1 Parallelization techniques

First of all, some considerations about the problem need to be done in order to account for the deployed parallelization techniques.

- Each layer depends on the previous one, since activations of layer $k-1$ serve as input for layer $k$. This means that parallelizing over `kernel()` calls – e.g. assigning different layers to different `tasks` – is meaningless: layer $k$ needs to be evaluated *strictly after* layer $k-1$.
- The compile-time constant $R$ is assumed to be *small*, according to the assignment specifications. As a consequence, we can conclude that parallelizing the loop over the $R$ input values that contribute to each output neuron would not be worthwhile.

Taken these considerations into account, we can efficiently **parallelize** only over **output neurons** belonging to the same layer, which is achieved through a `#pragma omp parallel for` directive inside `kernel()`. Essentially, output neurons within a single layer are distributed among threads, each taking care of computing the activations for one neuron at a time. The clause `private(i,j)` ensures that the two loop variables are private to each thread, while the clause `schedule(static)` causes the iterations to be evenly divided contiguously among threads: due to the structure of the network, we know that each iteration requires *exactly* the same number of operations, thus threads have an even workload.

It is worth pointing out that other parallelization techniques could be suitable for this problem – at least in principle – and briefly illustrating the reasons why they were not deployed. The `collapse` clause would allow to extend the parallelization also to the innermost loop (although with arguably little advantage, as mentioned above), but it would require to move the initialization and the application of the activation function outside the loops so as to obtain *perfectly nested loops*, which would result in greater effort overall. Likewise, although it could seem very appropriate to use the `reduction` clause, the core computation of the network being the MAC operation, this would require to instantiate *private* temporary variables to store activations –

[2] In the GPU implementation, an array was used to store inputs (and was overwritten for each layer) in order to lower the amount of required GPU memory.

since the clause does not accept array elements as accumulation variables – producing a copy latency that would end up worsening the performance instead of improving it.

### 1.2 Performance evaluation

The performance evaluation for the first program was carried out on the above mentioned *lab machine* exploiting an Intel® Core$^{TM}$ i7-2600 CPU working at 3.40GHz – which has 8 logical cores available – and determined according to 3 different metrics:

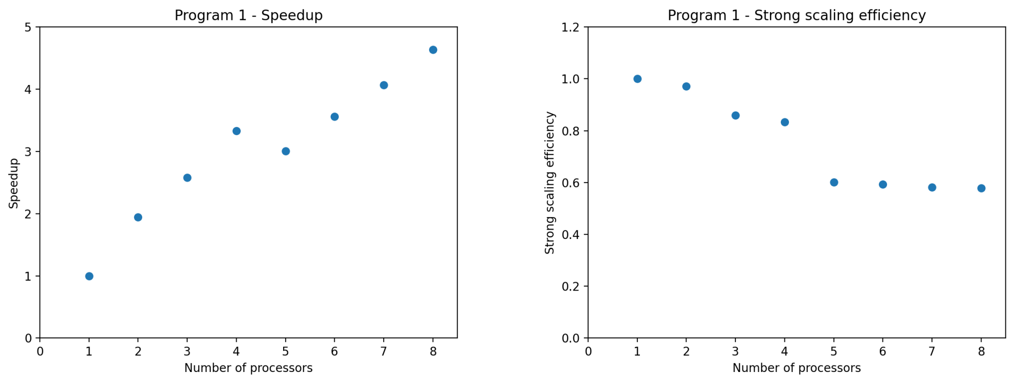

**Figure 1:** Speedup and Strong scaling efficiency for the CPU implementation using default values $N = 500000$ and $K = 150$. Execution times employed to compute both metrics result from an average on 5 different runs.

- **Speedup** was computed as $S(p) = T_{parallel}(1) / T_{parallel}(p)$ for each number of processors $p$, where $T_{parallel}(p)$ indicates the execution time of the program using $p$ processors. It basically measures the improvement achieved by exploiting a certain number of processors instead of a single one, and its ideal value is $S(p) = p$.

- **Strong scaling efficiency**, computed as $E(p) = S(p) / p$ for each $p$, is a sort of *relative speedup*, i.e. weighted on the number of employed processors, and has a (constant) ideal value of $E(p) = 1$.

- **Weak scaling efficiency** was computed as $W(p, K) = T_1(1, K) / T_{p,K}(p)$ for each $p$ and for different values of $K$, where $T_p$ refers to the execution time spent to complete $p$ *work units*. It aims at measuring performance keeping the per-processor work fixed.

Figure 1 shows both **Speedup** and **Strong scaling efficiency** over the number of exploited processors, obtained by averaging the execution times on 5 different runs (using default values for parameters $N$ and $K$) in order to get a more reliable result. Inspecting the plots, we note a slightly sub-linear increase in $S(p)$ until $p = 4$, after which it drops and starts rising again with a similar trend; we can observe $E(p)$ decreasing up to $p = 5$ – with a significant drop in the last step – to remain roughly constant afterwards. It is possible to notice how the results of both plots seem to be well divided into two groups of 4 points, which is probably related to the *hyper-threading* technology underneath. Indeed, the first 4 exploited processors are *physical* cores, while the last 4 are *virtual* cores: the latter are only able to exploit the stalls of the former, thus granting a poorer performance and affecting both evaluation metrics.

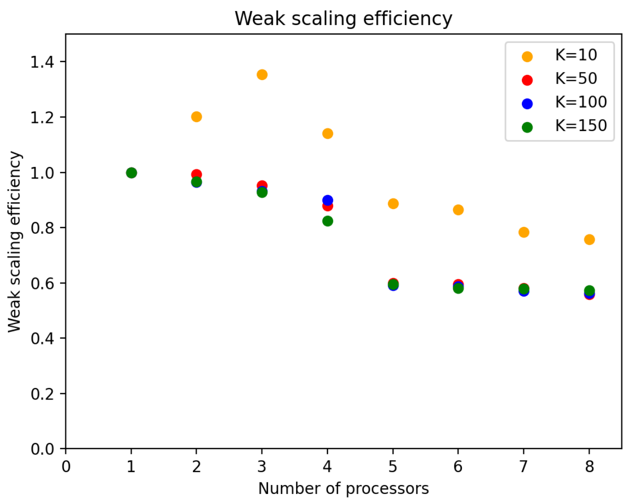

**Figure 2:** Weak scaling efficiency for the CPU implementation using values $N_p = 50000 * p$ for each $p$ and $k = 10, 50, 100, 150$. Execution times employed to compute it result from an average on 5 different runs.

On the other hand, **Weak scaling efficiency** – reported in Figure 2 – exhibits a peculiar behaviour. We firstly point out that computing $W(p, K)$ required several sets of runs, since the total amount of work needs to be proportional to $p$, which was achieved by setting $N_p = p * N$ (with very good approximation, given that $R$ is *small*[3]) and fixing $K$ for each set of 5 executions. The runs with $K = 50, 100, 150$ show a trend resembling that of $E(p)$: we observe a slow decrease in $W(p)$ up to $p = 4$, a significant drop at $p = 5$ and a roughly constant value afterwards, which we could relate to the already mentioned reasons. However, when it comes to $K = 10$ we can observe an unexpected rise in $W(p)$ up to $p = 3$, when it starts decreasing with a trend comparable to the other runs. We could maybe relate this behaviour to the relatively very small number of computations required by a network with just $K = 10$ layers, which can produce the CPU to enact memory optimizations such as *caching* techniques, which dramatically improve performance for small numbers of computations.

## 2 GPU implementation

### 2.1 Parallelization techniques

Recalling the observations made at the beginning of Section 1, we can know discuss the parallelization techniques used for the second program. The computation for each layer is performed calling `kernel<<<(n+BLKDIM-1)/(BLKDIM), BLKDIM>>>()` – using CUDA *execution configuration* syntax – where `BLKDIM=1024`, i.e. the maximum number of threads per block for the device (which allows, at least in principle, to fully exploit the computational power of each *SM*[4]), and $n$ is the number of output neurons of the layer. Accordingly, each thread computes the activation for a single, different element of the output array.

[3]Actually, layer $k$ has $N - (k-1) * (R-1)$ inputs, thus $N_p$ is slightly lower than $p * N$

[4]CUDA Streaming Multiprocessor

Just like before, we could deploy further techniques with respect to this configuration, which however would not improve the performance:

- **Parallelizing over the $R$ loop** would allow multiple (precisely $R$) threads to concurrently update a single output value, requiring to handle *race conditions* and to deal with *synchronization overhead*.
- The use of **shared memory** is usually exploited to improve performance when a certain amount of data needs to be accessed multiple times within the same block – which in our case happens for bias $b_k$ and inputs $x_{1,k}, ..., x_{n,k}$ of layer $k$. However, in our case it would imply a *synchronization overhead* anyway more costly than the benefits provided by the faster memory access: in the latter case, in particular, each value is reused only $R$ times (which we recall being small), while in the former case the reuse is equal to `BLKDIM`.[5]

## 2.2 Performance evaluation

The performance evaluation was carried out on the above mentioned *lab machine* exploiting a NVIDIA GeForce® GTX 580 GPU running at 1.57 GHz and with a global memory of 3005 MB. Two different evaluation metrics were employed to assess the performance of the Program 2:

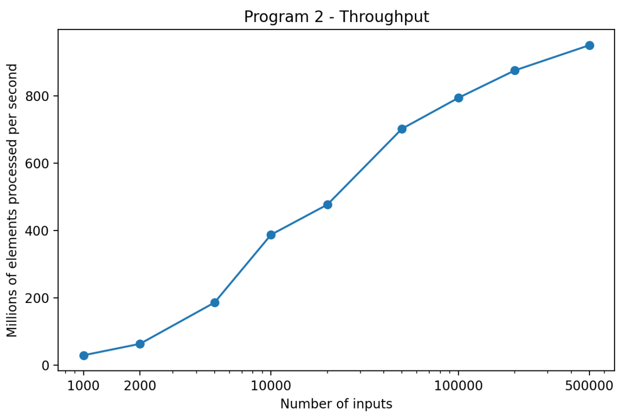

**Figure 3:** Throughput of the GPU implementation for different input sizes, with $k = 150$. Execution times employed to compute it result from an average on 5 different runs.

- **Throughput** was computed as $T(N_{processed}) = N_{processed}/T_{GPU}$, where $N_{processed}$ is the total number of processed data and $T_{GPU}$ is the execution time of Program 2. It gives information about the amount of data processed in the time unit, and is commonly used based on the fact that GPU parallel programming addresses acceleration aiming at performing *a lot of* computations at the same time, rather than *each single* computation faster.
- **Speedup with respect to the CPU implementation** was computed as $S(N) = T_{CPU}(N)/T_{GPU}(N)$, where $N$ is the usual first layer's input size and $T_{CPU}$ refers to the

[5]These techniques were tested and were observed not providing benefits, rather producing a worsening in the performance of the program.

execution time of the CPU implementation *exploiting all available processors*. This allows a fair comparison between the two implementations, avoiding spurious speedups being recorded.

**Throughput** is reported in Figure 3 and exhibits a logarithmic trend in $N$ – evidenced by setting a logarithmic scale for the x-axis. This highlights that improvement provided by GPU parallelization grows more significantly when the GPU remains mainly unused – and thus the additional effort for increasing the input size amounts to scheduling more blocks –, and less meaningfully as more *SMs* become gradually active and allow to take more and more advantage of the computational power of the device.

Figure 4 shows **Speedup with respect to the CPU implementation** over $N$ for fixed $K$. We can notice a dramatic escalation in the first few points, then a drop and a more gentle rise afterwards: this can be explained taking into account that the first 3 execution times for Program 1 grow with $N$, while the corresponding ones for Program 2 are nearly equal to each other – probably due to the still partial usage of the GPU *SMs*. Indeed, at higher values of $N$ – which bring along a more significant exploitation of the device – speedup acquires a more meaningful trend: the GPU implementation exhibits a considerable improvement in performance with respect to the CPU one.

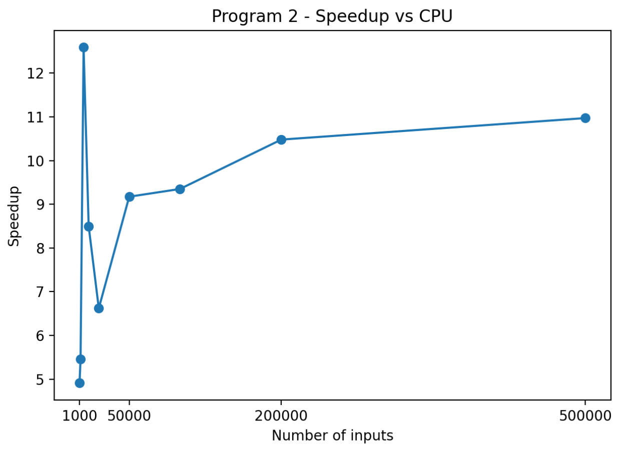

**Figure 4:** Speedup of the GPU implementation with respect to the CPU implementation for different input sizes, with $k = 150$. Execution times for both CPU and GPU implementations result from an average on 5 different runs.

## References

[1] OpenMP Architecture Review Board. OpenMP api version 4.0, 2013.

[2] NVIDIA. CUDA Technology, 2007.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.