Alma Mater Studiorum Università di Bologna - A.A. 2020/2021
Architecture and Platforms for AI - Module 2
Lucia La Forgia - 0000945383 - lucia.laforgia@studio.unibo.it

Programming Assignment Report

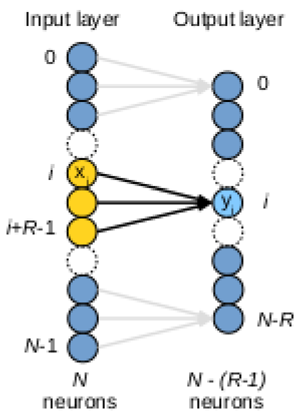
Abstract

The goal of this assignment is to write two distinct programs, one using OpenMP library and one using CUDA platform and interface, to perform efficient evaluation of feed-forward, sparsely connected, multi-layer Neural Networks.

Problem Specification and Analysis

The architecture of the NN has to follow the structure of fig.1, therefore it has to get an input of size *N*, meaning that the first layer will have *N* neurons, and it must be composed of *K* layers. The neurons of each layer are computed depending on specific *R* neurons of the preceding layer. Consequently, in order to be able to compute neurons of a certain layer safely, it is necessary to have calculated those of the previous layer; because of this, the evaluation of the NN layer by layer is conceived as the iteration of a sequential for-loop.

fig.1 multi-layer sparse NN



On the other hand, concerning the evaluation of single neurons within a particular layer, the task can be performed for each neuron independently, since they are not influenced by the computation of the adjacent neurons. For this reason, the neuron-evaluation task accomplished along each layer can be performed through the iteration of a parallelizable for-loop and it can be recognised as an *embarrassingly parallel task*. In particular, each iteration will execute the same computation to define any output neuron:

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

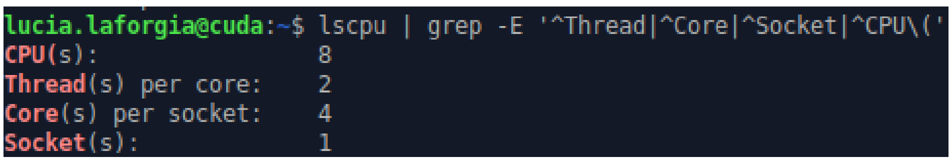
where *f* is the sigmoid function, *x* are the input, *W* the weights and *b* the bias. Thanks to the uniformity of the computation per neuron, it is possible to apply a *coarse-grained partitioning*.

A useful piece of information to remember is the equation to compute the total number of neurons to be evaluated given a certain combination of *N*, *K* and *R*:

$$\sum_{i=1}^{K-1} N - i(R-1) = (K-1)N - (R-1)\frac{K(K-1)}{2}$$

To finish, the characteristics of the machine where the programs had been tested are:

- concerning the CPU-based OMP version: *Intel(R) Core(TM) i7-2600 CPU @3.40GHz*



```
lucia.laforgia@cuda:~$ lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('
CPU(s):                8
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
```

- concerning the GPU-based CUDA version: *NVIDIA-SMI 390.116, GeForce GTX 580*

OpenMP Version

OpenMP is a cross-platform API that allows implementation and management of parallelisation within applications on shared memory systems. Supported by various programming languages and computer architectures and operating systems, this implementation is in C language.

1. Implementation - *nn_omp.c*

This implementation requires two input parameters at execution time which are read as a first step in the main function; these parameters have to be positive integers and the first (which is interpreted as the *N* parameter) has to be greater than the second (namely the *K* parameter) by *R-1* times and *R-1* units. Differently, the *R* parameter is hardcoded through the *#define* directive. Once the three fundamental parameters have been instantiated, the input neurons, the weights and the bias values are initialised with an array of random floats or array of arrays of random floats, in according sizes with *N* and *K*. Here is also allocated the memory space for the output neurons.

Then, there is the for-loop iterating along the *K* layers, which is executed sequentially as previously pointed out and it is surrounded by two calls to the *omp_get_wtime* function, in order to figure out the execution time of the actual program. Inside of this for-loop, the *compute_output* function is called, clearly to compute the output of each layer, one by one, iteration after iteration. This function contains another for-loop that iterates along the output neurons (outer for-loop) and yet within it a further, inner for-loop iterating over the relative *R* input neurons necessary to calculate each output neuron. The *compute_output* function holds all the needed calculation: input neurons are multiplied by the corresponding weight and summed up with the other *R* input neurons, plus the result goes through the sigmoid function. As previously seen, the iterations along the layers' neurons can be parallelised; therefore, over the outer for-loop, the following omp directive is applied, in order to parallelise the computation of the output neurons:

```
#pragma omp parallel for default(shared) schedule(static) num_threads(omp_get_max_threads())
```

This pragma allows to equally partition the computation of the output layer among the available threads, thanks to "*schedule(static) num_threads(omp_get_max_threads())*", having all threads performing the same amount of computation. The "*default(shared)*" piece states that all the input data are shared among the threads, while the index of iteration along the *R* needed neurons, namely *j*, is declared inside the *omp parallel* block so that it is created private, as it is better to be since each output neuron has its own *R* input neurons to iterate upon.

2. Evaluation

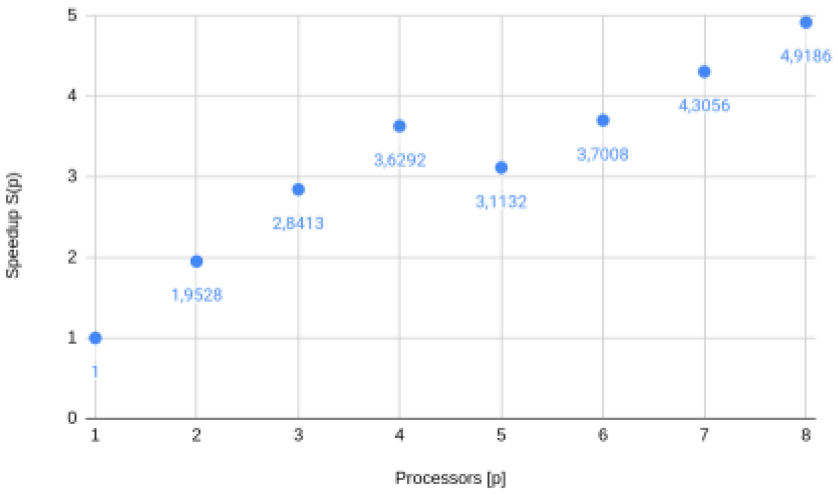
Firstly, some experiments have been carried out and here is a table containing the execution times in seconds (values are the mean of ten measurements) for different NNs but with equal problem size, namely with very similar amount of neurons to compute (≅250M):

N	K	p = 1	p = 2	p = 3	p = 4	p = 5	p = 6	p = 7	p = 8	p = 12	p = 16
55010	5000	10,6178	5,4372	3,7369	2,9256	3,4105	2,8690	2,4660	2,1587	2,8370	2,9677
251251	1000	10,5739	5,4314	3,7262	2,8764	3,4025	2,8615	2,4916	2,1630	2,9000	3,0099
2525353	100	10,6030	5,4454	3,7364	3,4990	3,4184	2,4756	2,4456	2,2633	2,6602	2,4660

These results show that, if the problem size remains constant, the program's execution takes the same time independently from N and K values: there is no difference, according to execution time, having a deep network or a more shallow one with larger input. It should also be noted that the duration of the executions keeps decreasing as threads are added up to 8, then, with 12 and 16 threads, the execution time increases again. This can be explained by the fact that the CPU has 4 cores with 2 threads per core, so there is an advantage when scaling to a maximum of 8 logical cores, while there is a larger "not-worth-the-wait" overhead, when using more than 8.

Next, it is important to consider the Speedup in latency of the program using parallelisation with respect to the one running with a single thread, in order to understand how much faster the task can be solved when using *p* workers. The graph of fig.2 represents the Speedup achieved exploiting from 1 to 8 threads (12-threaded and 16-threaded versions have not been taken into account given what has been said previously).

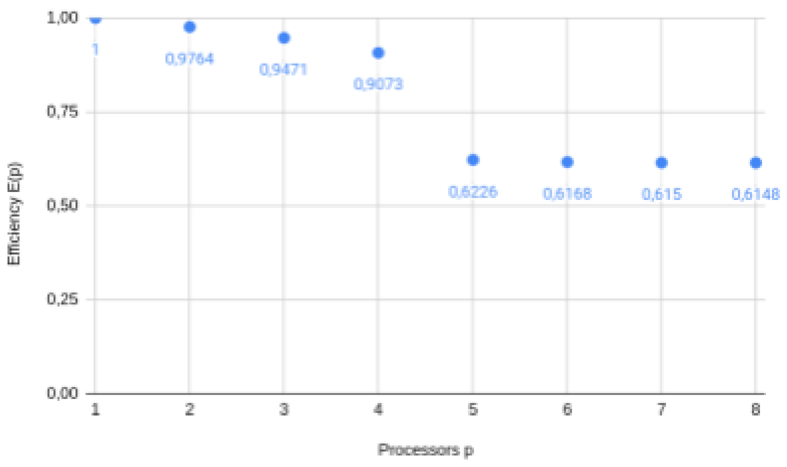
fig.2 Speedup with N = 55010 and K = 5000



As foreseen by looking at the execution times, fig.2 shows that the speedup increments almost linearly as the number of threads increases, with the exception of what is observed in the transition from 4 to 5 threads; this is attributable to the CPU having only 4 physical cores.

Moving forward in the evaluation of performance, the following graph (fig.3) depicts the Strong Scaling Efficiency of the previous executions. The *E(p)* is the result of increasing the number of processors involved while keeping the size of the problem fixed.

fig.3 Strong Scaling Efficiency with N = 55010 and K = 5000



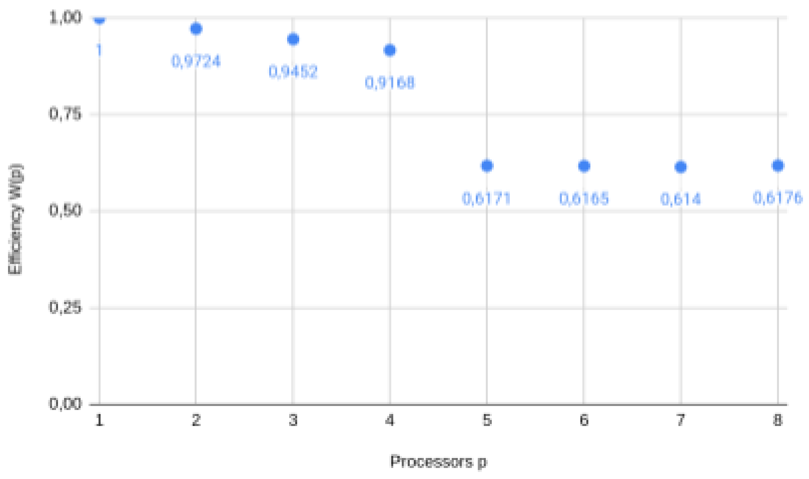
For fig.3, it has to be highlighted that *E(p)* tends to decrease monotonically with the number of threads from 1 to 4, then it faces a remarkable drop between 4 and 5 threads execution and it remains roughly constant afterwards, from 5 threads onwards.

Last evaluation consists of Weak Scaling Efficiency monitoring, which observes what is the execution time when increasing the number of processors and keeping the per-processor work fixed. In order to do so, it can be found a set of different *N*s or a set of different *K*s such that the amount of workload per processor remains the same no matter the increase of processors' number.
In this case, it has been decided to find different N_ps, starting from N_0=55010 and K_0=5000.

$$N_p = (1-p)K_0 + pN_0$$

The set of N_p can be computed with the formula just above, while the following graph (fig.4) sums up the trend of *W(p)*.

fig.4 Weak Scaling Efficiency with N = 55010 and K = 5000



As observed for *E(p)*, also *W(p)* decreases as *p* increases from 1 to 4, then it drops significantly for *p* = 5 and it remains nearly constant until *p* = 8 has been reached.

CUDA Version

CUDA (Compute Unified Device Architecture) is a parallel computing platform and API created by NVIDIA. It is a software layer giving direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

1. Implementation - *nn_cuda.cu*

As for the OpenMP version, the program requires *N* and *K* input parameters at execution time and it sees *R* parameter hardcoded through the *#define* directive. In this case, also the block dimension is hardcoded with the *#define* directive. Input neurons, weights, bias values and output neurons are host-side initialised coherently with *N* and *K*, like in the previous implementation; then, the *cudaMalloc* calls for device-side memory allocation are performed.

Next, after the call of *hpc_gettime* function, the input, weights and bias values need to be copied from host to device, through the function *cudaMemcpy*, and, to do so, a for-loop iterating along the *K* layers is exploited. Once again, this for-loop is to be executed sequentially.

Inside of this for-loop, there is also the *compute_output* kernel function call, after which the host waits for kernel execution to finish, because of the invocation to the *cudaCheckError* function, which contains *cudaDeviceSynchronize* function call.

The kernel function *compute_output* is clearly responsible for the computation of the output of each layer, using its specific, device-side input, weights and bias. In particular, the computation of each single output neuron is assigned to a distinct CUDA core, which also exploits data reuse through memory sharing for what concerns input and bias value, declared using the specifier __*shared*__, while weights are kept private since they are only used once each. Here, to avoid race conditions, all the threads are synchronized and, once that input and bias have been saved for threads' shared usage, each thread performs the due computation to get the value of its output neuron: it sums up the products of inputs and weights, adds the bias and applies the sigmoid function; eventually, the result is stored in the device memory. The output layers that are not the final one are not moved to the host memory, this is because of a well-known good practice, with the goal to reduce memory transfer overhead: only the last output neurons, which are the NN's final output, are copied back from device to host.

Finally, when the last layer has been reached, there is the *cudaMemcpy* function's call to copy the final output layer from device to host, as just mentioned, plus, another invocation of the *hpc_gettime* function, in order to figure out the execution time of the actual program, together with the returned value of the previous call of the same function.

It has to be highlighted that the computation of the execution time includes the memory transfers between host and device, because these are better to be taken into account when analysing GPU-based programs' performances, since they are part of their workload.

2. Evaluation

As a first step, there is the need to fix the number of threads per block and, to find the best value, a compromise is always needed: a quantity of threads that is too small brings minor improvements from sharing memory, but, on the other hand, having too many threads means a time-claimant execution of the __*synchthreads* function; therefore the block dimension was fixed to the reasonable size of 128 threads.

N	K	execution time
55010	5000	0,3773 s
251251	1000	0,3177 s
2525353	100	0,2101 s

An important aspect is that, due to this implementation, given a certain problem size, the biggest advantages are to be seen when evaluating a shallow network with a large input layer, as it can be seen in the table on the left (timing values are the mean of ten measurements).
This is because the iteration along the *K* layers is carried out sequentially, therefore there is no exploitation of GPU parallelism through it.

From the execution time values reported in this table, it can already be seen that there is a big improvement in terms of execution speed. For what concerns the evaluation of this program's performance, the concept of Speedup clearly is no longer meaningful, since there is little or no control over the number of CUDA cores used, plus, the hardware multiplexes CUDA threads to CUDA cores. Because of this, for the CUDA-based implementation, the evaluation metrics are Throughput and Speedup vs CPU implementation.

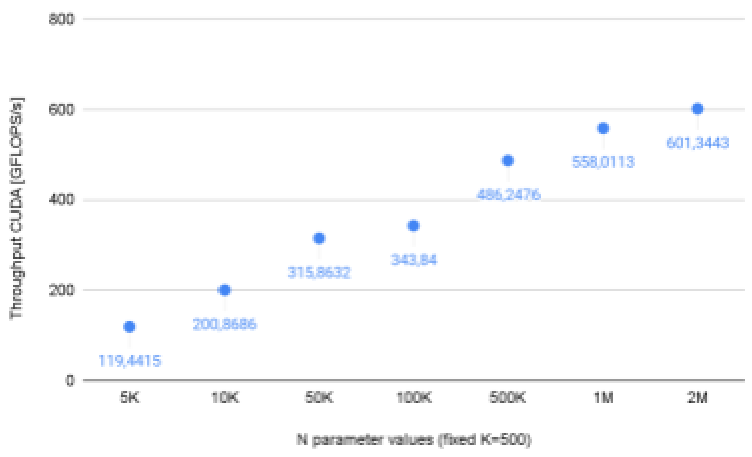
The Throughput is given by the formula on the right, where *C* is the total number of neurons to be computed (obtainable by the formula

$$TP = \frac{C \cdot op}{T \cdot 10^9}$$

previously reported), *op* is the number of floating point operations to be performed per output neuron, *T* is the time (in seconds) required for the overall execution by the NN and the denominator multiplication by 10^9 is because the unit of measure is Giga Flops/s. In particular, each output neuron needs: *R* multiplications and additions (operations among input and weights), 1 further addition (bias), 1 multiplication + 1 addition + 1 exponentiation (sigmoid operations); thus, in this case, *op* = 2**R* + 1 + 3.

The following graph (fig.5) sums up the Throughput values obtained by increasing the problem size keeping fixed *K* and increasing *N*.

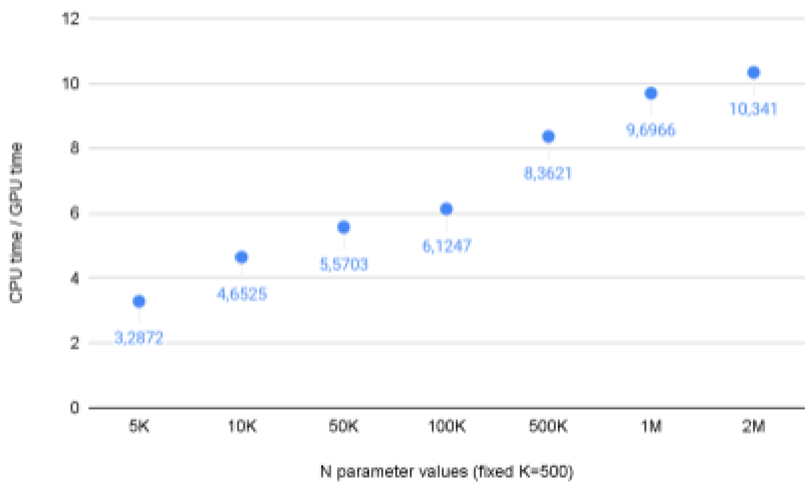
fig.5 Computational Throughput



The fig.5 graph displays that the throughput increases as the problem size increases, when the latter is solely due to an increase in *N* (having fixed *K*). This can be explained by the fact that CUDA cores process more data in parallel when having larger input.

In the end, the Speedup vs CPU implementation graph (fig.6) represents the ratio between the CPU execution time performed by using the OpenMP program with 8 threads and the GPU execution time required by the program just outlined.

fig.6 Speedup vs CPU implementation



This report illustrates that, as the problem size grows, the CUDA-based solution overcomes the OpenMP implementation.