ALMA MATER STUDIORUM BOLOGNA - University of Bologna
Architecture and Platforms for Artificial Intelligence

Programming Assignment Report

Author: Lorenzo Mario Amorosa - lorenzomario.amorosa@studio.unibo.it

1. Overview and Patterns

The scope of the assignment is to evaluate a sparse connected neural network made of `K` layers, in which each neuron of layer `i` depends from `R` neurons of layer `i-1`, being `N` the number of neurons for the first layer. As a consequence, in order to compute the layer `i` it is necessary to have entirely computed the layer `i-1`, so the evaluation of the network is done by an outer loop which evaluates the `K` layers sequentially.

The evaluation of all the neurons of a given layer is instead an **embarrassingly parallel task**, since each neuron can be computed using only a subset of the inputs, the related weights and the bias, independently from the other output neurons of the same layer. The amount of computation needed to evaluate a single output neuron is always the same (being `R` fixed), so a **regular coarse-grained partitioning** of the workload is adequate.

The total number of neurons to be computed, given `K` layers, `N` neurons for the first layer and `R`, is given by the following formula:

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

The overall complexity of the network evaluation is so bounded linearly by `N` and `K`.

The network evaluation is also a **stencil computation**: indeed the layers are updated according to a fixed pattern and many layers are computed in sequence, using the previous output layer as input. The computation of the new layer is embarrassingly parallel.

2. OpenMP: Implementation

The code described in this section is contained in the file `neural_network_omp.c`. In the `main` function, after the read of input parameters `N` and `K`, all the weights, the biases and the first layer are initialized. Then it follows a for loop that iterates `K-1` times to compute the `K-1` output layers, by invoking the function `compute_output_layer`. Here the wall clock time is taken, being the network evaluation (the core of the computation) performed through this loop. In order to allocate less memory, at each iteration (apart from the last one) the input layer `X` and the output layer `Y` are swapped, so that only the needed input and the current output are kept in memory.

The function `compute_output_layer` computes the output layer `Y` using the input layer `X`, the weights `W` and the bias `b`. The outer for loop iterates over `Y`: the current output neuron `Y_i` is firstly initialized, then it is updated with the products between `X` and `W` in the inner for loop `R` times, and in the end it is summed with `b` and passed through the sigmoid function. The number of steps to compute an output neuron is therefore constant.

On the outer loop it is applied the **omp directive** to parallelize the computation of the output neurons:

```
#pragma omp parallel for default(none) private(j) shared(X, W, Y,
out_dim, b) schedule(static)
```

In this way, the computation of the output layer is partitioned equally (due to the directive `schedule(static)`) to the available threads, which all receive the same amount of computation to perform. All the input data are shared among the threads (`X`, `W`, `Y`, `out_dim`, `b`) and a private variable (`j`) is given to each thread to perform the inner loop.

Alternatively, it could be possible to parallelize also the inner for loop, which updates the output neuron through weights and inputs, but it would be not a very efficient choice: indeed each update of the output neuron must be atomic to avoid race conditions. As a consequence, it is preferable to assign the update of an output neuron to a single thread and therefore exploit the parallelism to compute multiple output neurons at the same time.

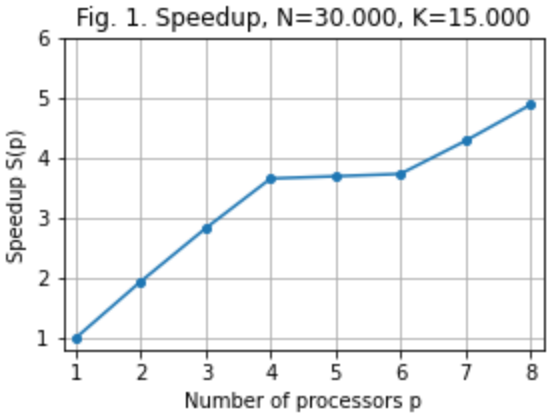
3. OpenMP: Performance evaluation

It is firstly verified that, having fixed the problem size given by the number of output neurons to be computed for each layer, the program performs the same way independently from `N` and `K`. This means that it makes no difference having a shallow network with many inputs or a deeper one with fewer inputs.

In the following table the timings to compute 100 millions of output neurons are shown, the needed network dimensions are computed using the formula shown in section 1 and the number of processors `P` used is varied.

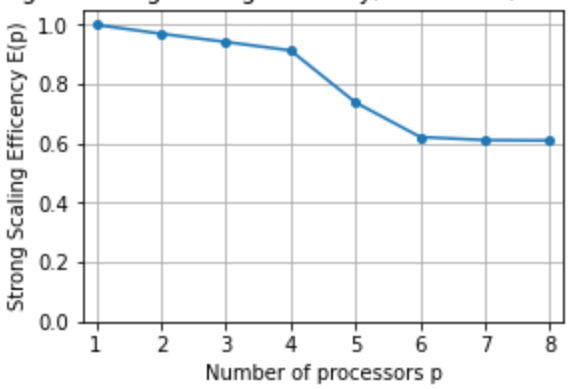
N	K	P=1	P=2	P=4	P=8
1.010.201	100	4.254s	2.186s	1.164s	0.883s
101.100	1.000	4.241s	2.182s	1.158s	0.871s
25.004	5.000	4.247s	2.187s	1.161s	0.878s
20.001	10.000	4.246s	2.188s	1.169s	0.884s

The **speedup** (Fig. 1) is evaluated using `N=30.000` and `K=15.000`, which are values that produce a bigger network with respect to those depicted in the previous table. The metric is in general almost linear, apart from the case with 5 and 6 threads where performances do not improve a lot despite the increase of thread number.



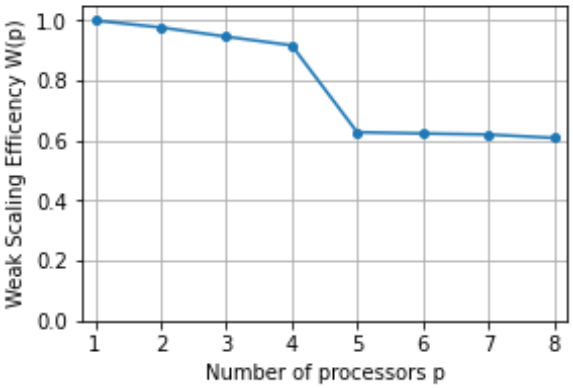
The **strong scaling efficiency** (Fig. 2) is evaluated using the previous N and K. It slightly decreases when the number of threads P increases, with a significant drop for P=5 and P=6.

Fig. 2. Strong Scaling Efficency, N=30.000, K=15.000



The workload to compute the **weak scaling efficiency** (Fig. 3) is calculated using the formula shown in section 1. The number of output neurons, which corresponds to the number of parallel executions of the for loop, is taken as a unit of computation. In particular, 10 millions of output neurons are computed by the program using 1 thread, 20 millions of output neurons are computed using 2 threads and so on. As in the previous case, the weak scaling efficiency slightly decreases when the number of threads P increases, and it drops significantly for P=5.

Fig. 3. Weak Scaling Efficency



It resulted that different values of N and K led to really similar speedup and strong scaling efficiency plots. Also changing the workload resulted in analogous weak scaling efficiency.

4. CUDA: Implementation

The code described in this section is contained in the file `neural_network_cuda.cu`. As it is done in OpenMP implementation, it is firstly performed the read of `N` and `K` parameters and then all the weights, the biases and the first layer are initialized. The required `cudaMalloc` calls are invoked afterwards.

The following for loop evaluates the `K-1` layers of the neural network: at each iteration weights are copied to the device memory and the kernel function `compute_output_layer` is invoked. After that, the host waits for the end of the kernel execution through the `cudaDeviceSynchronize` invocation, since the computation of the next layer cannot start before the computation of the output of the previous one. The pointers to the input and the output in device memory are then swapped for the computation of the following layer, in order to keep in memory only the current needed data without other allocations. It has to be noticed that only the last output layer is copied back to the host to reduce the memory transfer overhead.

The **kernel function** `compute_output_layer` computes the output layer `Y` using the input layer `X`, the weights `W` and the bias `b`. The computation of a single output neuron is assigned to a distinct CUDA core. The inputs `X` are firstly moved to shared memory in order to exploit data reuse (each input is used by `R` cores): for each block `R-1` threads take into account to move to shared memory a (`R-1`)-sized halo, necessary to compute the output neurons on the boundary. The bias is moved to the shared memory as well, since it is the same for each output neuron of a layer. On the other hand, weights are not moved to shared memory since each weight is used only once. Then the threads are synchronized, in order to have all the needed inputs to compute the output neurons and avoid data race conditions. In the end, each thread accumulates the products between `X` and `W`, adds `b` and applies the sigmoid function, and stores the final result in the device memory.

5. CUDA: Performance evaluation

It firstly pointed out that having fixed the problem size, i.e. the total amount of output neurons computed in all the layers, evaluating a shallow network with many inputs is far more efficient than evaluating a deeper one with fewer inputs, differently from the OpenMP implementation. In the following table the timings to compute 100 millions of output neurons are shown, varying `N` and `K` properly using the formula proposed in section 1.

N	K	Time
1.010.201	100	0.215s
101.100	1.000	0.325s
25.004	5.000	0.391s
20.001	10.000	0.441s

This result can be explained considering that many inputs (`N>>1`) allow better exploitation of the parallelism provided by the GPU, whereas several layers (`K>>1`) impose a constraint on parallelism, since each layer is evaluated one by one sequentially.

The following tests are performed having fixed the CUDA block size dimension to 128 threads, which is a reasonable tradeoff: if the threads per block were too few the advantages of shared memory would become less significant, instead if they were too many more time would be necessary to synchronize the threads before the computation of the neurons.

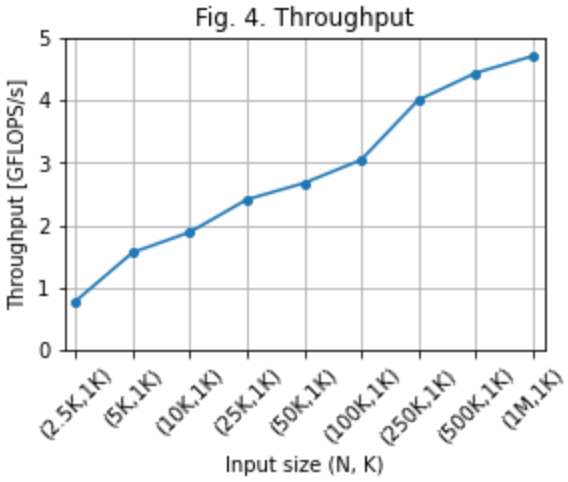
The **throughput** of the program corresponds to the number of processed data per seconds as a function of the input size (`N`, `K`). Each possible couple of input values results in the computation of a total number of neurons `C`, obtainable using the formula shown in section 1. In particular, we have that for each output neuron the following computations are performed:

- `R` multiplications and additions, because of the products between `X` and `W`
- 1 addition because of the bias
- 1 multiplication, 1 addition and 1 exponentiation because of the sigmoid.

So, defining `M` the overall number of floating point operations per output neuron and `T` the seconds required for the network evaluation, we calculate the throughput `PT` (in GFLOPS/s) as follows:

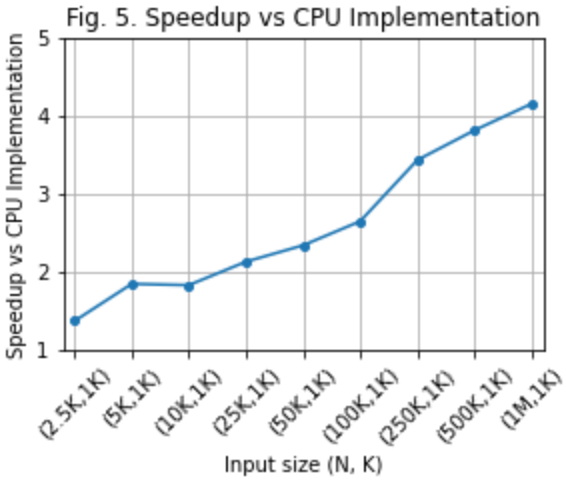
$$TP = \frac{C \cdot M}{T \cdot 10^9}$$

Using this information we can compute the metric shown in Fig. 4. It is worth noting that in the computation the execution time includes the memory transfers between host and device.



The results show that the throughput improves by increasing the input size (`N` in particular). In this way indeed, the amount of data that is processed in parallel by the CUDA cores increases, allowing more floating point operations per unit of time.

The **speedup vs CPU implementation** is computed comparing the performance of the CUDA and OpenMP programs, setting up for the latter the environment variable `OMP_NUM_THREADS=8`. In this way, the CPU version of the program can exploit multithreading and the comparison is fair.



Going to evaluate the execution time, we consider the time to compute the for loop over all layers of the network for both implementations, including memory transfer operations between host and device for the CUDA program. The comparison in Fig. 5 shows that for small networks the CPU program keeps the pace with GPU one. The bigger the network becomes, the more the CUDA implementation outperforms the OpenMP one. This result confirms that a better exploitation of the GPU occurs when many inputs are processed in parallel.