Parallel implementation of locally connected multi-layer NN

Architectures and Platforms for AI – Module 2 Moreno Marzolla

AA 2020/2021

Daniele Verì

Introduction

This project implements the locally connected multi-layer neural network regressor forward process. It exploits the parallelism of CPUs with OpenMP library and Nvidia GPUs with CUDA programming. The network takes batches of random data as input and initialize weights and biases according to a uniform distribution, so that input files are not required. Testing and profiling is automatized and executed on cloud based solutions.

1. Project organization

The only dependencies needed to build and run the project are: `gcc`, `make` and `nvcc`.

The two main sub folders are `src` which contains the code and `profile` which contains the shell scripts used to analyze the performance of the programs.

```
.
├── Doxyfile
├── Makefile
├── profile
│   ├── colab_notebook.ipynb
│   ├── colab_profile.sh
│   ├── gcp_profile.sh
│   ├── local_profile.sh
│   ├── out
│   ├── prof_cuda.sh
│   └── prof_omp.sh
├── README.md
├── report.pdf
└── src
    ├── constants.h
    ├── cuda
    │   ├── cuda.cu
    │   └── cuda.h
    ├── main_cuda.c
    ├── main_omp.c
    ├── openmp
    │   ├── openmp.c
    │   └── openmp.h
    └── utils
        ├── hpc.h
        ├── log.c
        ├── log.h
        ├── model.c
        └── model.h
```

Inside the `src` folder there are `cuda` and `openmp` implementation specific folders and the respectively entry points `main_cuda.c` and `main_omp.c`. Other sources like `constants.h` and the content of the `utils` folder are shared between both the programs, in particular `model.h` contains the definition of the data structures while `hpc.h` provides utility functions to measure the elapsed time and call safely cuda API.

Inside the `profile` folder, `prof_cuda.sh` and `prof_omp.sh` performs the actual analysis while the other scripts run them on local or remote machines.

Finally `Makefile` takes care of the compilation of both the implementations.

command	description
`make`	build all
`make openmp`	build openMP implementation
`make openmp_dbg`	build openMP debug
`make cuda`	build CUDA implementation
`make cuda_dbg`	build CUDA debug
`make cuda_legacy`	build CUDA for arch `compute_20` (cuda version < 9)
`make doc`	create documentation with doxygen
`make clean`	clean objects

`Doxyfile` is used to generate the documentation from the comments in header files while `README.md` contains detailed instructions to run the profiling scripts.

2. Data structures

The basic entities used are of course vectors and matrices. The `vector_t` type is implemented as a `float` array while `matrix_t` type is a 2-dimensional `float` array allocated in a contiguous memory region for performance purpose. `model_t` type represents the NN weights and biases (respectively a matrix and a vector for each layer).

The input data is a matrix where the columns represent the features and the rows the elements in the batch (the `BATCH_SIZE` is defined in `constants.h` and fixed to 1 during the profiling).

Both input and network parameters are uniformly initialized in the interval ± 2.

3. Serial implementation

The naive serial implementation of the locally connected NN regressor consists in 4 nested loops that iterates respectively over: the number of layer, the batch elements, the layer outputs and the R neighbors. The activation function is applied to the accumulated sum-product in each layer except the last one.

In order to verify the correctness of the algorithms, the output of the serial and parallel versions are compared element-wise with a certain tolerance (`EPS` defined in `model.h` with the value of `1e-5`) due to the floating point accuracy.

4. OpenMP

4.1 Implementation

The loop that iterates over the batch elements and the one that iterates over the layer neurons are both embarrassingly parallelizable, so they can be collapsed together and distributed among the available cores.

It is however not convenient to parallelize the inner most loop (that iterates over the neighbors) with a reducer because the thread context switch overhead would be higher than the actual execution time (as `R` is small by definition).

Since there is a predictable amount of work the static scheduling strategy is preferable over the dynamic one.

Finally, according to the best practices, the default `omp` variable scope is `none` while those used are marked as `firstprivate`.

4.2 Analysis

The allocation and de-allocation of the output buffer performed at each layer represent the intrinsically serial part of the code. Measuring the time elapsed during this operation over the total execution time with only one thread results in α=0.046, meaning that the possible asymptotic speedup is 21.3. Using the Amdahl law is possible to calculate the ideal speedup given the number of threads and compare it with the actual value obtained on different processors. The executable was run on a VM instance of Google Cloud Platform with 32GB of RAM and CPUs:

- AMD EPYC 7B12 (8 cores, 16 threads, 2.25GHz, 16MB L3 cache).
- Intel Xenon E5-2697 v4 (16 virtualized threads, 2.3GHz, 45MB L3 cache)

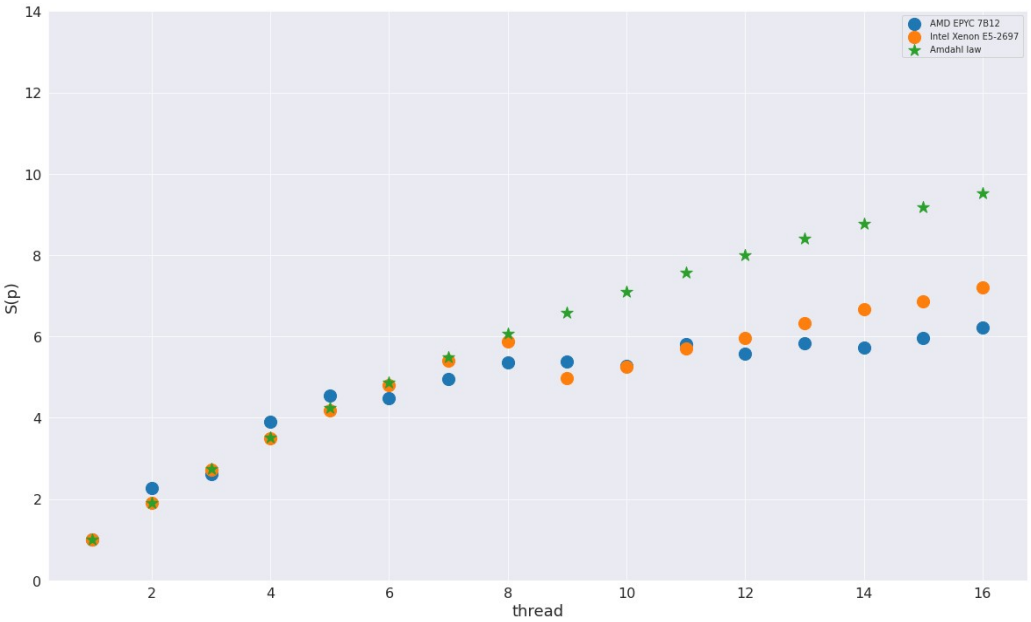

Illustration 1: Speedup with $N=10^6$ and $K=100$

Considerations: the effective speedup is not high as expected, maybe due to the overhead of OMP or the virtual environment. The AMD times seem to be noisier than the Intel Xenon, even if the latter shows an important drop after the 8 threads and then it recover the linear trend.

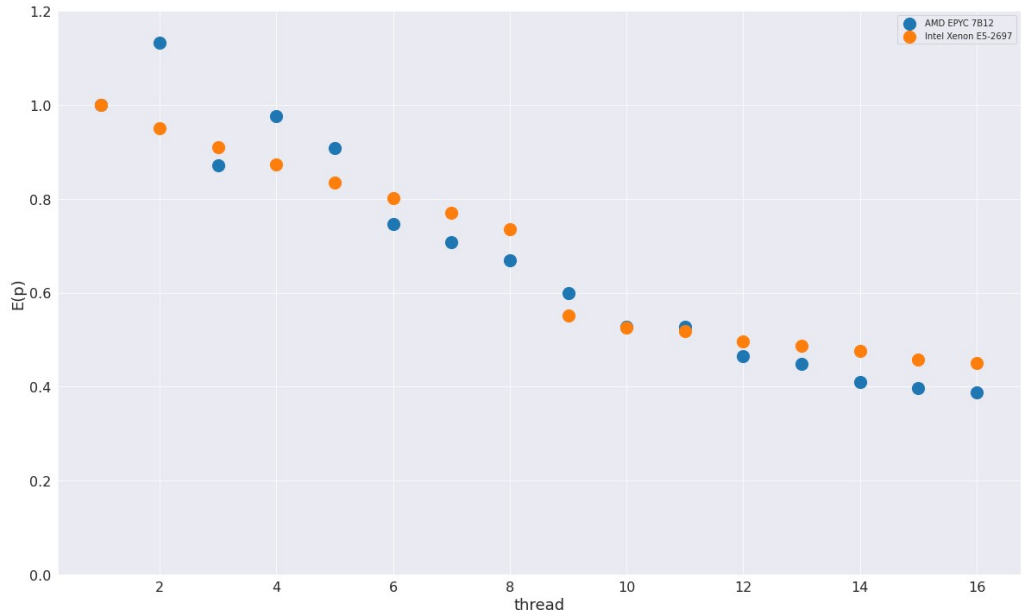

Illustration 2: Strong scaling efficiency with $N=10^6$ and $K=100$

Considerations: The noise of AMD measurements is even more visible in strong scaling efficiency plot. From this plot it is possible to estimate the real value of α to about 0.1.

About the weak scaling efficiency, given the initial input size N_1 the amount of work is obtained as follows:

$$w_1 = RK\left(N_1 - \frac{(R-1)(K+1)}{2}\right)$$

then the input size required for the instance with p threads N_p is given by:

$$N_p = p\,\frac{w_1}{RK} + \frac{(K+1)(R-1)}{2}$$

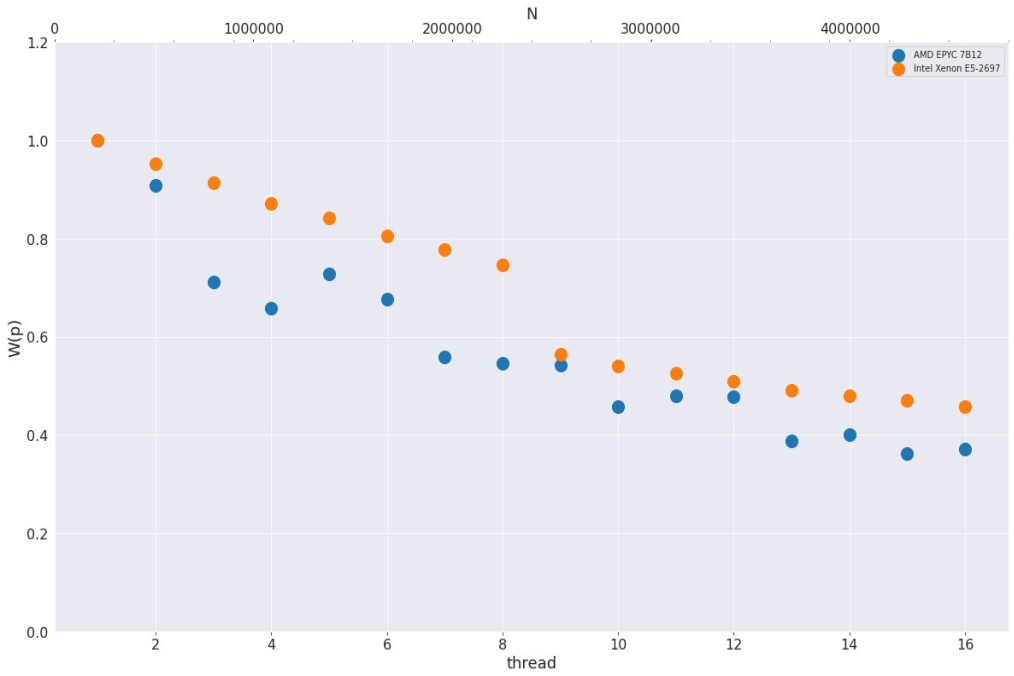

Illustration 3: Weak scaling efficiency with K=100

5. CUDA

5.1 Implementation

Despite the CUDA implementation shares the model with OpenMP, it is necessary to re-implement the allocations using paged memory. Other functions take care to to copy data structures back and forth the device memory (by copying first the the data arrays, then whole `struct` and finally updating the reference with that one pointing the array on the device memory). In order to avoid memory leaks the function `test_device_mem_leak` performs numerous allocation/deallocations of large data structures so that any error would lead to an increasing memory consumption.

The cuda_forward_mlp function copies the input matrix and the model to the device memory, then invokes the kernel for each layer and finally returns the output on host memory.

Each kernel computes a neuron output of the current layer and are indexed with a 2D grid and 1D thread block.

In order to exploit data reuse, the shared memory of each block is used to store input elements, also handling the case where input elements are not multiple of `BLOCK_DIM`.

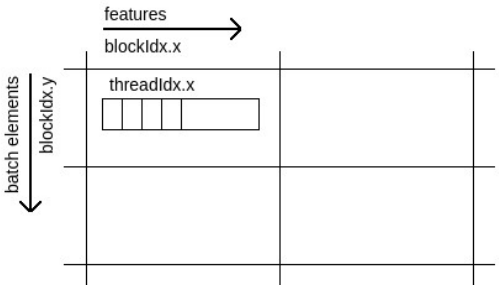

Illustration 4: Grid and blocks

5.2 Analysis

Throughput measurements are taken dividing the number of computed elements by the elapsed kernel time. `BATCH_SIZE` is fixed to 1.The executable is run on machines with different GPUs:

- Tesla V100-SXM2: 5120 CUDA cores, 1.53GHz, 32GB
- Tesla P100-PCIE: 3584 CUDA cores, 1.12GHz, 16GB
- Tesla T4: 2560 CUDA cores, 1.59GHz, 16GB
- GeForce MX130: 384 CUDA cores, 1.19GHz, 2GB

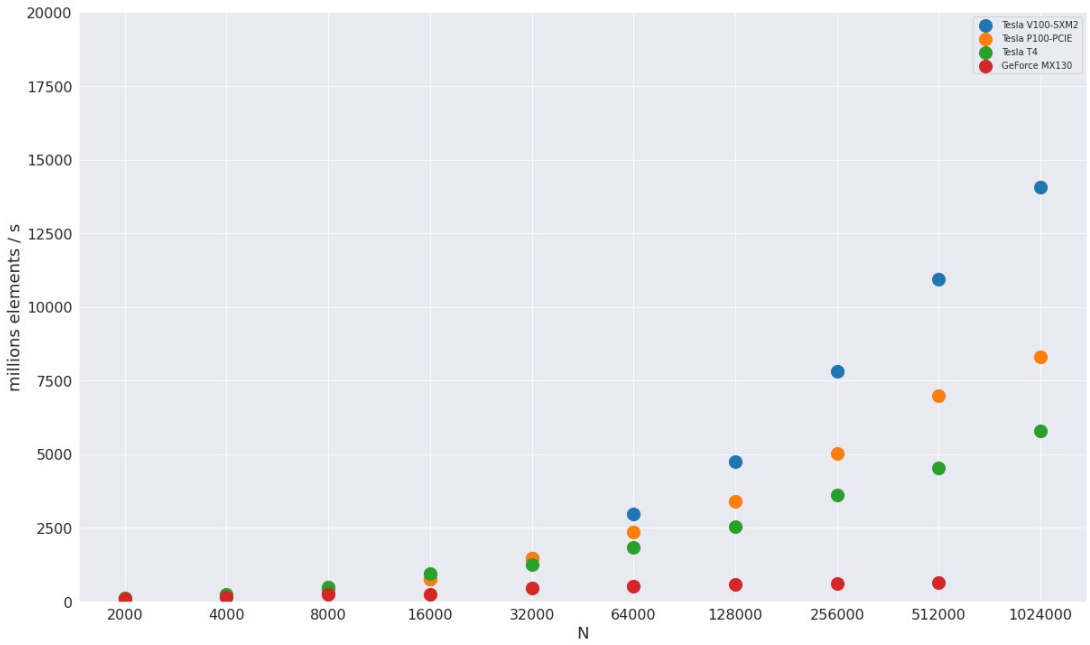

Illustration 5: Throughput with K=100

Considering that each element is a float encoded with 4 bytes, these measurements are further confirmed by the output of the Nvidia profiler `nvprof` about the average `Global Store Throughput`:

V100: 63.473GB/s, **P100**: 33.807GB/s, **T4**: 23.315GB/s, **MX130**: 2.1721GB/s.

6. Extension

Both the implementations can handle efficiently batches of input data without lose performance on the test case with only one element.

Another little feature is the ReLU activation implemented in a branch-less fashion:

`x > 0 ? x : 0` becomes `(x > 0) * x`

in this way branch efficiency is preserved, in particular on CUDA warp where each branch is executed serially.

Particular attention is paid to the compilation targets:

- `standard`: disable the `cudaSafeCall` macros.
- `debug`: set the `-g` flag and enable the `cudaSafeCall` macros.
- `release`: set the `-O2` flag and disable the `cudaSafeCall` macros.

However, all tests are executed with the standard target.