Neural network implementation with OpenMP and CUDA

Mattia Orlandi

Master in Artificial Intelligence - University of Bologna

A.Y. 2020-2021

Abstract

In this project I have addressed the implementation, using C++14, of a neural network, in which parallelism is achieved by means of OpenMP and CUDA. First of all, I provide a brief description of the headers and modules developed, and I discuss about the problem complexity, useful to test *strong* and *weak scaling efficieny*. Then, in the following section I show my OpenMP implementation, and provide an in-depth analysis of its performance and of possible alternatives. Finally, in the last section I provide a CUDA kernel which implements the forward pass of the neural network layers, and analyse its performance in terms of *effective bandwidth*, *computational throughput* and *speed-up* w.r.t. the OpenMP implementation.

1 Project structure

The project is organized in the following classes:

- `model` folder:
  - `Tensor` (`Tensor.hpp`/`Tensor.cpp`) - abstract class describing a tensor; it is a wrapper for arrays of floats, providing constructor, assignment operators and destructor that automate memory allocation and de-allocation.
  - `Vector` (`Vector.hpp/Vector.cpp`) - class derived from `Tensor` and describing a vector (*i.e.* a 1D tensor), useful to represent input data.
  - `Matrix` (`Matrix.hpp/Matrix.cpp`) - class derived from `Tensor` and describing a matrix (*i.e.* a 2D tensor), useful to represent layer weights.
- `nn` folder:
  - `SparseLinearLayer` (`SparseLinearLayer.hpp`/`SparseLinearLayer.cpp`) - class describind a single layer of the network, characterized by a matrix of weights, by a bias, and by a non-linearity (in this case, a sigmoig function).
  - `NeuralNetwork` (`NeuralNetwork.hpp`/`NeuralNetwork.cpp`) - class describing the whole neural network, consisting in a certain number of stacked `SparseLinearLayers`.

In particular, the `NeuralNetwork` class provides a `forward` method which, given an input `Vector`, calls sequentially the `forward` method of every `SparseLinearLayer`: such method is where the actual computation is performed, and its sequential implementation is shown below:

```
Vector SparseLinearLayer::forward(const Vector &in_vector) const {
    /* ... */
    for (int i = 0; i < _out_features; i++) {
        float val = 0;
        for (int r = 0; r < cols; r++)
            val += in_vector[i + r] * _weights[i * cols + r];
        out_vector[i] = _non_linearity(val + _bias);
    }
    return out_vector;
}
```

As it can be seen, it comprises two nested `for` loops:

- the external one iterates over the $(N - t(R-1))$ output nodes y_i, where t is the current layer number;
- the inner one iterates over the R relevant input nodes x_{i+r}, multiply them for the corresponding weights $W_{i,r}$, with $r = 0..(R-1)$, and accumulate the partial results.

Such method is called once for each layer, thus $(K-1)$ times. Therefore, in total we have that the "problem size" $\mathcal{P}$ is:

$$\mathcal{P} = \sum_{t=1}^{K-1} (N - t(R-1)) \cdot R = \left((K-1)N - \frac{K(K-1)}{2} \cdot (R-1) \right) \cdot R$$

Knowing the problem size will be useful to compute strong and weak scaling efficiency in the following chapters.

2 OpenMP implementation

As stated before, the `forward` method of the `NeuralNetwork` class consists in three `for` loops:

- one iterating over the network's layer (**not parallelizable** due to data dependency between layers);
- one iterating over the output nodes of a certain layer (**embarassingly parallel 1D stencil**);
- one iterating over the relevant input nodes of a certain output node in a layer and accumulating the partial result (**parallel reduction**).

In the next section both parallelization techniques will be discussed.

2.1 Parallel outer `for`

With this technique, each thread executes a subset of the iterations in the outer `for`: to implement it, we must add a `pragma` directive before such `for`, followed by some options which specify the behaviour of the thread pool. In particular, the schedule is set to `static` since the workload of each iteration is predictable and fixed (R multiplications, $R+1$ sums and 1 sigmoid); moreover, such choice leads to a better run-time performance, since scheduling is performed at compile-time.

2.2 Parallel inner `for` with reduction

With this technique, each thread executes a subset of the iterations in the inner `for` and accumulates a partial result which is then reduced into a single value. Again, it suffices to add a `pragma` directive before the inner `for`. The OpenMP options used are basically identical to the previous case: the main difference is that a reduction is applied on the variable `val`: each threads updates a private copy of `val`, and then a reduction operator (+) is applied to such private copies and to the initial value (0). Since, according to the problem's specification, we can assume that R is small, we expect this technique to be less effective than the previous one.

2.3 Performance evaluation

In this section the two techniques are evaluated in terms of both strong and weak scaling efficiency. All tests are performed with fixed $R = 3$ on a 12-core, Intel i7 CPU (10^{th} gen), and timings are averaged over 5 runs.

The `forward` method of both `NeuralNetwork` and `SparseLinearLayer` accepts an optional `int` argument `mode`, which determines the parallelization technique to be employed:

- `mode = 0`: parallel outer `for` (default);
- `mode = 1`: parallel inner `for` with reduction;
- `mode != 0, 1`: sequential version (for testing).

First of all, strong scaling efficiency is tested: the problem size remains constant whereas the number of processors is progressively increased.

As it can be seen from figure 1, the efficiency tends to monotonically decrease for both techniques w.r.t. the number of processors, but the parallelization of the outer for leads to a much higher efficiency then the reduction strategy. Moreover, we can notice that for the first technique there's a significant drop in efficiency in the transition between #processors = 6 and #processors = 7: this can be explained by the fact that the CPU has only 6 physical cores, and thus there may be a small overhead when using the other 6 logical cores. In fact, when $7 <$ #processors < 12 the decrease is again less steep.

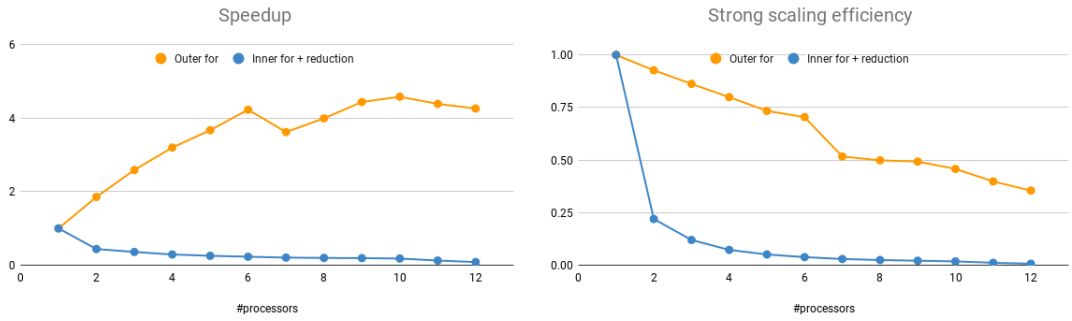

Figure 1: Speed-up and strong scaling efficiency with $N = 1000$, $K = 250$.

As far as weak scaling efficiency is concerned, we keep the per-processor work constant as we increase the number of processors; to do so, we need to find out how scaling the total problem size is related to scaling the inputs N and K. With $R = 3$, the problem size becomes:

$$\mathcal{P}_{R=3} = \left((K-1)N - \frac{K(K-1)}{2} \cdot 2\right) \cdot 3 \approx (K-1)N - K(K-1)$$

where the 3 can be omitted since it's constant.

Then, by fixing K, we can compute the scaled N as $N_p = (1-p)K_0 + pN_0$, where N_0 and K_0 are the number of initial input neurons and the number of layers for the base case.

In figure 2 (left) we can observe the trends of weak scaling efficiency w.r.t. the number of processors: again, as expected the first technique is much more efficient than the second one; the decrease in efficiency around #processors $= 6$ is still present. For #processors > 7 efficiency seems to reach a plateau around 0.50.

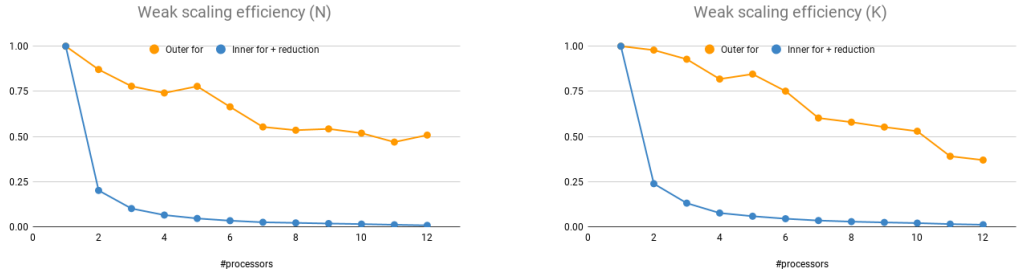

Figure 2: Weak scaling efficiency with $N_0 = 1000$ scaled and $K = 250$ fixed (left), and with both $N_0 = 1000$ and $K_0 = 25$ scaled (right).

Moreover, in certain cases ($K \ll N$) it's possible to scale K and re-adjust N according to the following rules:

$$K_p = p \cdot K_0, \quad N_p \approx \frac{K_p^2 + (N_0 - K_0)K_p - pN_0}{K_p - 1}$$

The related trend is depicted in figure 2 (right). Once again parallelizing the outer `for` is more effective, but its efficiency's trend is more irregular than in the previous case: this can be explained by the fact that the scaling rules shown before give approximated values for N_p, which has more impact on the problem complexity than K_p.

3 CUDA implementation

In the CUDA implementation, the project struture remains very similar to the OpenMP version, apart from the following differences:

- host memory in `Tensor` class is now *pinned* in order to enable DMA and speed-up memory operations between host and device;
- `Tensor` class provides methods to allocate/free memory on device, and to copy data between host and device;
- the layer's forward pass logic is implemented in the CUDA kernel `forwardKernel`;
- non-linearities are implemented as pointers to `__device__`-decorated functions instead of lambdas.

In order to handle arbitrarily deep networks in GPUs with limited memory, layers are initialized on host and copied to device on-the-go: more precisely, during the (asynchronous) execution of a layer's kernel, the next layer is (asynchronously) copied to device in advance in order to hide latency; this, together with DMA access, leads to a significant speed-up in the execution.

Inside the kernel, input data is initially copied into fast shared memory in order to contrast expensive data reuse: such memory takes into account also a $(R-1)$-sized halo, necessary to

compute the output element on the boundary. Conversely, shared memory is not used for the weights, since each weight is accessed only once. Then, to avoid data race conditions, the threads are synchronized; such synchronization was placed inside the `if` clause such that all the threads with an out-of-bounds global index may exit immediately without waiting. Finally, each thread accumulates the partial result, adds the bias and applies the non-linearity (*i.e.* sigmoid).

3.1 Performance evaluation

Tests are performed on an NVIDIA GeForce GTX 1650 Ti Mobile (Turing, SM75), which has a theoretical memory bandwidth $\text{BW}_{Th} = 192$ GB/s. The shrinking factor is again fixed to $R = 3$, whereas the number of blocks N_{blk} varies according to the number of input neurons N_{in} at each layer, s.t. $N_{blk} \cdot \text{BLKDIM} \approx N_{in}$, in order to handle arbitrarily-sized vectors.

Fixing the number of threads per block involves a trade-off: if they were too few the benefits of shared memory and thread cooperation would become minimal, whereas if they were too many the call to `__synchthreads` would require more time; for this reason, such value was empirically fixed to $\text{BLKDIM} = 256$. As before, metrics and timings are averaged over 5 runs.

Since in CUDA we do not have direct control on the number of cores used, traditional speed-up cannot be used as a metric: effective bandwidth (number of bytes read/written by the kernel w.r.t. execution time)[1], computational throughput (number of floating point operations performed w.r.t. execution time)[1] and speed-up w.r.t. CPU must be used instead. In particular, `forwardKernel` at each layer performs:

- $N_{in} + (R-1) \cdot \lfloor N_{in}/\text{BLKDIM} \rfloor$ reads from input data;
- $N_{in} + (R-1) \cdot \lfloor N_{in}/\text{BLKDIM} \rfloor$ writes to shared memory (not relevant for effective bandwidth computation[2])
- $N_{out} \cdot R$ reads from shared memory (not relevant for effective bandwidth computation[2]);
- $N_{out} \cdot R$ reads from weights;
- N_{out} writes to output data;
- $N_{out} \cdot R$ multiply-add (MAC) operations (without taking into account bias and sigmoid, since their contribution is negligible).

Therefore, for each layer t, given the execution time T_E^t of the t-th kernel (in milliseconds), the effective bandwidth and the computational throughput of such layer can be expressed as:

$$\text{BW}_{Ef}^{(t)} = \frac{R_B^{(t)} + W_B^{(t)}}{T_E^{(t)} \cdot 10^6}, \quad R_B^{(t)} + W_B^{(t)} = N_{in}^{(t)} + (R-1) \cdot \lfloor N_{in}^{(t)}/\text{BLKDIM} \rfloor + N_{out}^{(t)}(R+1)$$

$$\text{TP}^{(t)} = \frac{\#\text{FLOPs}^{(t)}}{T_E^{(t)} \cdot 10^6}, \quad \#\text{FLOPs}^{(t)} = 2 \cdot \#\text{MACs}^{(t)} = 2 \cdot N_{out}^{(t)} \cdot R$$

It can be easily seen that the kernel is *bandwidth-bound*, since $\#\text{FLOPs}^{(t)} \ll R_B^{(t)} + W_B^{(t)}$.

The effective bandwidth and computational throughput trends are shown in figure 3; such metrics were measured on a per-kernel basis, and then averaged over all the $(K-1)$ kernel calls. Since each layer process progressively less data, a large K leads to smaller throughput and bandwidth: thus, in order to test the maximum achievable metrics, K is fixed to a reasonable but not too big number ($K = 200$).

As it can be seen, the trends are almost identical: when $N = \{5\text{K}, 10\text{K}, 20\text{K}\}$, bandwidth is always below ~ 40 GB/s while throughput is always below ~ 12 GFLOP/s; then, for higher

[1] "How to implement performance metrics in CUDA C/C++", Mark Harris (NVIDIA's Developer Blog).
[2] as confirmed by NVIDIA Corporation employee Robert Crovella.

values of N (50K, 100K, 200K), both metrics experience a dramatic increase to around ~ 100 GB/s (for bandwidth) and around ~ 28 GFLOP/s range (for throughput). Finally, when N is very large (500K, 1M, 2M), bandwidth tends to saturate at a maximum of ~ 171 GB/s (89% of theoretical bandwidth) whereas throughput reaches ~ 50 GFLOP/s.

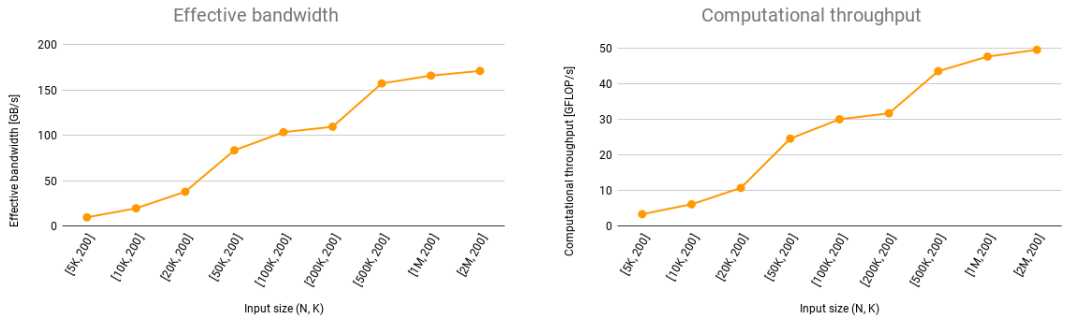

Figure 3: Effective bandwidth and computational throughput for different (N, K) pairs (N increased exponentially, $K = 200$ fixed).

Speed-up w.r.t. CPU (figure 4) was obtained by measuring the time required by the `forward` method of the entire neural network in CUDA (considering memory operations) and CPU (OpenMP with 12 cores) implementations. We can observe that the speed-up is negligible ($\sim$ 1x) for $N = 5000$ but increases steeply to a maximum of $\sim$ 3x ($N = 50000$); then, it decreases and stabilizes at around $\sim$ 2x ($N = \{1\text{M}, 2\text{M}\}$).

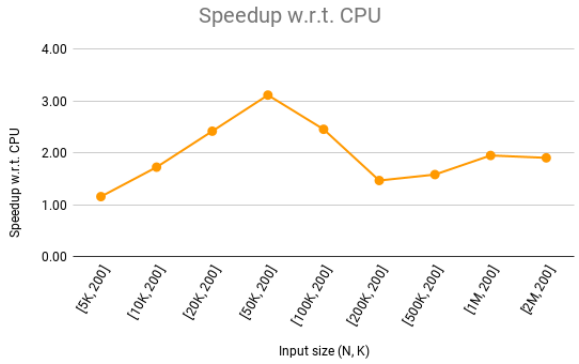

Figure 4: Speed-up w.r.t. the CPU implementation for different (N, K) pairs (N increased exponentially, $K = 200$ fixed).

The reason behind this behaviour could be that the GPU execution time takes into account also the time to copy the layers' weights from host to device memory: for small amount of data, the overhead due to memory operations is negligible but GPU's capabilities are not fully exploited; on the other hand, for large amount of data the GPU can process them more efficiently, but memory operations become a bottleneck reducing the overall speed-up. Therefore, $N = 50000$ represents the optimal trade-off between GPU utilization and contained memory overhead.

In conclusion, the CUDA version turns out to be approximately twice more fast than the best OpenMP version with 12 core for very large amount of data.