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 CPU 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 efficieny 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`; the corresponding code is the following:

```
    #pragma omp parallel for num_threads(omp_get_max_threads()) schedule(static) \
    shared(in_vector, out_vector, _weights, cols) default(none)
    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);
    }
```

In C/C++, OpenMP constructs can be used by means of `#pragma` directives to the compiler:

- `omp parallel`: forks additional threads (the original thread will become the "master" of the pool of threads);
- `for`: splits up loop iterations among the threads;
- `num_threads`: specifies the number of threads to be spawned (in this case, the maximum number available);
- `schedule`: specifies the scheduling policy (in this case it is static, *i.e.* iterations are assigned to threads before the loop is executed, and are divided evenly among threads);
- `shared`: specifies which variables are shared between different threads (in this case, both input and output arrays, the layer's weights, and the number of relevant input nodes are shared);
- `default`: specifies the default sharing policy (in this case "none", meaning that a sharing policy must be explicitly specified for each variable accessed inside the parallel section).

In particular, the schedule was 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; the corresponding code is the following:

```
1     for (int i = 0; i < _out_features; i++) {
2         float val = 0;
3         #pragma omp parallel for reduction(+:val) num_threads(omp_get_max_threads())
          ↪  schedule(static) \
4         shared(in_vector, out_vector, _weights, i, cols) default(none)
5         for (int r = 0; r < cols; r++) {
6             val += in_vector[i + r] * _weights[i * cols + r];
7         }
8         out_vector[i] = _non_linearity(val + _bias);
9     }
```

The same OpenMP constructs as before are used; the only differences are that the variable `i` is also shared, and 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 CPU.

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

- `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 thr fact that the CPU has only 6 physical cores, and thus there may be a small overhead when using the other 6 are logical cores. In fact, when $7 <$ #processors < 12 the decrease is less steep.

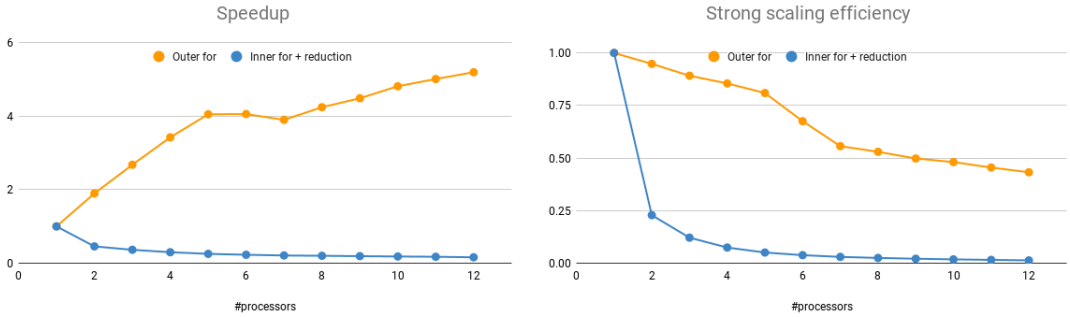

Figure 1: Speed-up and strong scaling efficiency with $N = 1000$, $K = 250$.

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 we can fix K and scale N according to the following formula:

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

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 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 between #processors $= 6$ and #processors $= 7$ is still present.

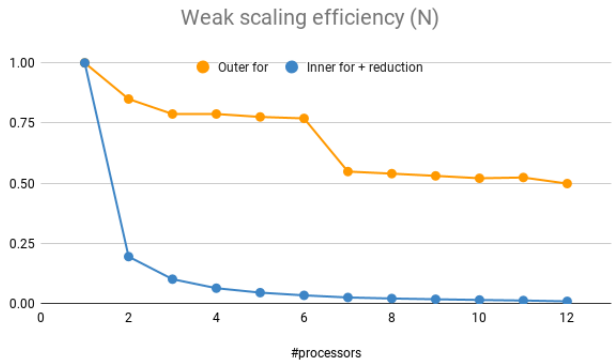

Figure 2: Weak scaling efficiency with $N_0 = 1000$, $K_0 = 250$, N_p scaled accordingly.

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

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

The related trend is depicted in figure 3. Once again parallelizing the outer `for` is more effective, but its efficiency's trend is much 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.

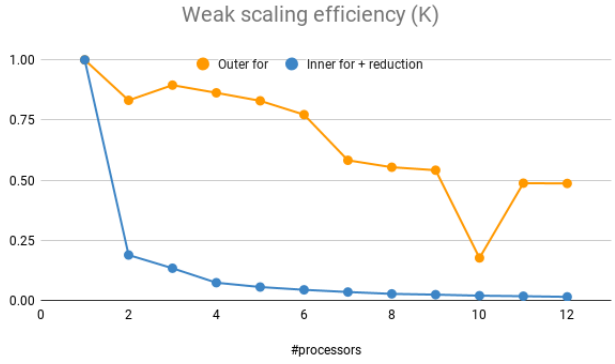

Figure 3: Weak scaling efficiency with $N_0 = 1000$, $K_0 = 25$, N_p and K_p scaled accordingly.

3 CUDA implementation

In the CUDA implementation, the project struture remains very similar to the OpenMP version; the only differences are:

- `Tensor`, `SparseLinearLayer` and `NeuralNetwork` classes provide `toDevice` and `toHost` methods to move data from host to device, and from device to host, respectively;
- the forward pass logic is implemented in the CUDA kernel `forwardKernel`;
- non-linearities are implemented as pointers to `__device__`-decorated functions instead of lambdas.

In particular, allocation and deallocation of memory in host and device, as well as memory copies, are automatically handled by `Tensor`'s constructors, assignment operators and destructors.

It is worth noticing that the execution time measured takes into account only the `forward` method of the neural network, and not the time required to allocate its layer's weights and to move them to device. Moreover, input vector is moved to device once, at the beginning of the forward pass, while all the intermediate results are kept on device for better performance; the result of the last layer is then moved back to host.

As before, the `forward` of the neural network calls sequentially the `forward` method of each layer; the latter then calls the kernel, whose code is shown below:

```
__global__ void forwardKernel(const float* in_data, float* out_data, const float*
↪  weights, float bias, non_linearity_t non_linearity, unsigned int in_total, unsigned
↪  int out_total) {
    __shared__ float cache[BLKDIM + R - 1];  // Shared memory for input data
    const unsigned int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
    const unsigned int local_idx = threadIdx.x;
    float result = 0;
    int offset;

    if (global_idx < in_total) {  // Check index boundaries for input
        // Copy data from global to local memory (also the right halo)
        cache[local_idx] = in_data[global_idx];
        if (local_idx < R - 1 && global_idx + BLKDIM < in_total)
            cache[local_idx + BLKDIM] = in_data[global_idx + BLKDIM];
        __syncthreads();  // Synchronize threads before computing output
        if (global_idx < out_total) {  // Check index boundaries for output
            // Accumulate result, add bias and apply non-linearity
            for (offset = 0; offset < R; offset++) {
                result += cache[local_idx + offset] * weights[global_idx * R + offset];
            }
            out_data[global_idx] = non_linearity(result + bias);
        }
    }
}
```

As it can be seen, 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. 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 a NVIDIA GeForce GTX 1650 Ti Mobile (Turing), which has a theoretical memory bandwidth $\mathrm{BW}_{Th} = 192$ 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 such reason, such value was empirically fixed to BLKDIM = 256.

Therefore, the kernel is called at each layer with the following execution configuration:

```
forwardKernel<<<(in_vector._total + BLKDIM - 1) / BLKDIM, BLKDIM>>>(...);
```

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 MAC 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;
- $2 \times (N_{in} + (R-1) \cdot \lfloor N_{in}/\text{BLKDIM} \rfloor)$ writes and reads to shared memory (not relevant for effective bandwidth computation[2])
- $N_{out} \cdot R$ reads from weights;
- N_{out} writes to output data;
- $5 + N_{out} \cdot R$ multiply-add operations (taking into account bias and sigmoid too).

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:

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

$$\text{TP} = \text{MACs}/\left(T_E^{(t)} \cdot 10^6\right), \quad \text{MACs} = 5 + N_{out}^{(t)} \cdot R$$

It can be easily seen that the kernel is **bandwidth-bound**.

The effective bandwidth and computational throughput trends are shown in figure 4; such metrics were measured on a per-kernel basis, and then averaged over all the $(K-1)$ kernel calls. As it can be seen, the trends are almost identical: for low values of N (from 50 to 1000) effective bandwidth and computational throughput are almost 0 GB/s and 0 GFLOP/s, respectively; then, for higher values of N (from 5000 to 500000), both metrics experience a dramatic increase to $\sim$ 150 GB/s (for bandwidth) and $\sim$ 22 GFLOP/s (for throughput). However, when N is too large (1 million, 5 millions) the increase becomes less pronounced: the effective bandwidth reaches a maximum of $\sim$ 175 GB/s (91% of theoretical bandwidth) whereas throughput reaches $\sim$ 26 GFLOP/s.

[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.

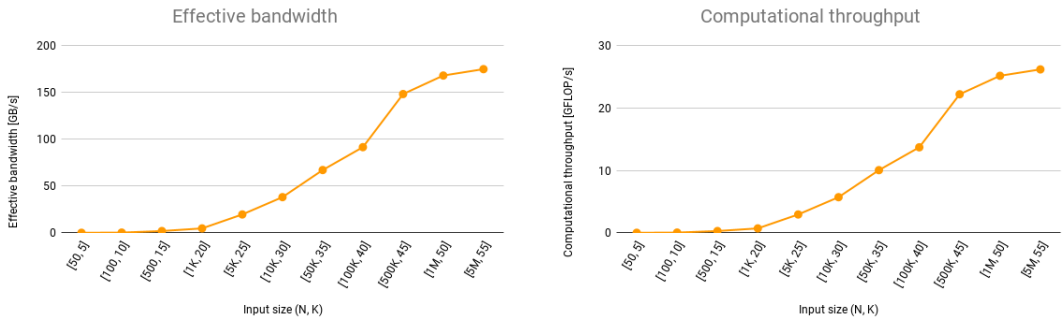

Figure 4: Effective bandwidth and computational throughput for different (N, K) pairs. Empirically, it was observed that the value of K has little impact on these metrics, but was included in this benchmark for completeness (small values of K, increased linearly, have been chosen in order to avoid Out-Of-Memory errors).

Concerning speed-up w.r.t. CPU (figure 5), it was obtained by measuring the time required by the `forward` method of the entire neural network in both CUDA and CPU implementations. We can observe that the speed-up is negligible for small values of N (from 50 to 500) but it increases steeply from $\sim$ 65x ($N = 1000$) to $\sim$ 930x ($N = 100000$); then, when $N = 500000$ it has a slight drop to $\sim$ 885x and then rises again to $\sim$ 1210x ($N = 5$ millions).

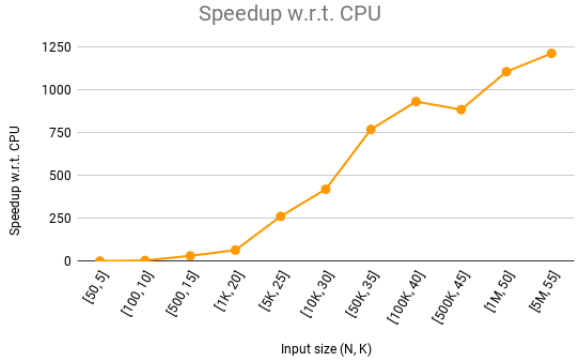

Figure 5: Speed-up w.r.t. the CPU implementation for different (N, K) pairs.