OpenMP and CUDA implementation of a simple multi-layer Neural Network

Simone Gayed Said

Master in Artificial Intelligence, University of Bologna, Italy

Abstract

The goal of this assignment is to write two programs, one using OpenMP and one using CUDA, that can efficiently evaluate a sparsely connected multi-layer NN. This report will provide a concise description of the parallelization strategies that have been used and a performance analysis of the programs using the most appropriate techniques.

1. Introduction

From the problem specification we know that:

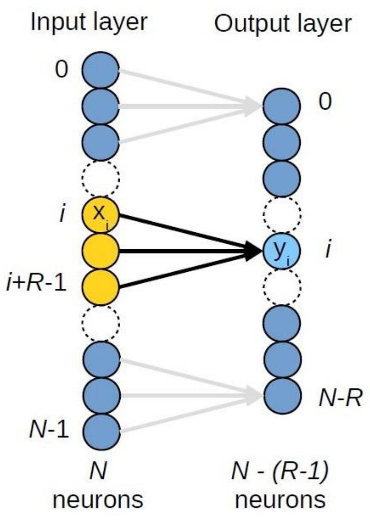

Figure 1: Structure of two adjacent layers

Let N be the number of nodes (neurons) in the input layer; each neuron holds a real value x_i , $i = 0, \dots, (N-1)$. The output layer has $[N-(R-1)]$ neurons, each one holding a real value $y_j, j = 0, \dots, (N-R)$, where R is our shrinking factor and it is constant. For each, $i = 0, \dots, (N-1)$, y_i depends only on the R values $x_i, \dots\ x_{i+R-1}$ according to the following formula: $y_i = f\left(\sum_{r=0}^{R-1} x_{i+r} \times W_{i,r} + b\right)$ where $f(x) = 1/(1+e^{-x})$ (*sigmoid* function), $W_{i,r}$ and b are real number (Figure 1).

In a multi-layer scenario then it is possible to build a K layer NN by taking K layers connected as described above. If layer 0 has N nodes, then layer t has $[N - t(R-1)]$, with $t = 1, \dots, K-1$.

Having said that, we can now write down a possible pseudocode for the function/kernel, which should then be called repeatedly ($K-1$ times) to evaluate all K layers of the multi-layer NN, to study its complexity.

Algorithm 1: Function/Kernel pseudocode

```
for outnode in NextLayer do
    sum = 0
    for innode in Window do
        sum += innode * weight
    outnode = f(sum + bias)
```

The algorithm is made of two nested loops, the outer one that iterates over the $N-t(R-1)$ output nodes y_i, instead, the innermost iterates over the $x_i, \dots\ x_{i+R-1}$ adjacent input nodes, multiply them by the corresponding weights, and accumulates the partial result. The computational complexity is thus $O(\sum_{t=1}^{K-1}(N-t(R-1)))$.

The loop that iterates over the network's layer is not parallelizable because each layer depends on the previous one (data dependency between layers). In the next sections, possible parallelization strategies, for the other two loops, will be discussed.

2. OpenMP

2.1 Parallelization strategies

1. **Parallelization of the outermost loop.** In OpenMP, it can be easily implemented by using the *#pragma omp parallel for* directive before such *for* loop. In this way, the code of the parallel region is duplicated, and all threads will execute it. Since the work per iteration is pre-determined and predictable the schedule *static* clause has been chosen, this also led to less work at runtime because the scheduling is done at compile-time.

2. **Parallelization of the inner loop.** As in the previous strategy, we can parallelize it by using the *#pragma omp parallel for* directive before such *for* loop. In this case, we must ensure that only one thread at the time updates the partial result shared variable. In OpenMP to do this, it can be used the *#pragma omp atomic* directive which forces all threads to serialize during the update of the shared variable. Another method, which is the one that has been used, is to apply the *reduction* clause to the partial result variable. It creates one private copy of the reduction variable for each thread, each private copy is initialized with the neutral element of the reduction operator, in our case 0 because the reduction operator is (+), then each thread executes the parallel region and when all threads finish, the reduction operator is applied to the last value of the local reduction variable.

2.2 Performance Analysis

All the analysis were performed on a Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz which uses multithreading, it has 4 physical cores and 2 threads per core, so the total number of virtual processors is 8, and by fixing $R=5, \mathrm{N}=2000, \mathrm{K}=500$. To have more meaningful and statistically correct results, all the measurements were repeated 10 times and then averaged.

The two aforementioned strategies have been evaluated using different techniques:

- **Speedup $\boldsymbol{S(p)}$.** $S(p) = \frac{T_{serial}}{T_{parallel}(p)} \approx \frac{T_{parallel}(1)}{T_{parallel}(p)}$

  To compute the speedup, instead of using the serial program, that might lead to a spurious superlinear speedup that is not there, it has been used the parallel program with $p = 1$ processors. From the Speedup graph (Figure 2) it can be seen that the second strategy, the parallelization of the inner loop with the reduction clause, seems to not take advantage of the number of cores available, increasing the number of cores increases the time needed to complete the task. The parallelization of the outer loop, instead, seems to be quite effective, its speed up curve follows an upward trend even if it is very far from a linear speedup.

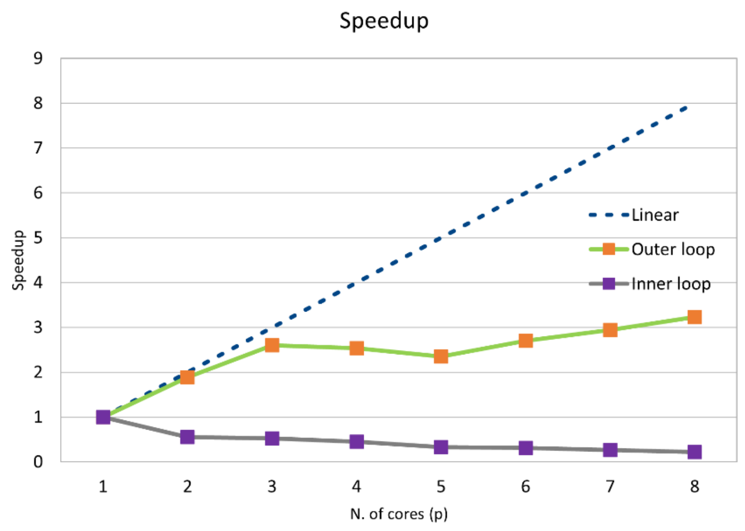

Figure 2: Speedup (N=2000, K=500, R=5)

- **Strong Scaling efficiency.** $E(p) = \frac{S(p)}{p} = \frac{T_{parallel}(1)}{p \times T_{parallel}(p)}$

  Strong Scaling means increase the number of processors p keeping the total problem size fixed. The total amount of work remains constant instead, the amount of work for a single processor decreases as p increases. The aim is to reduce the total execution time by adding more processors.

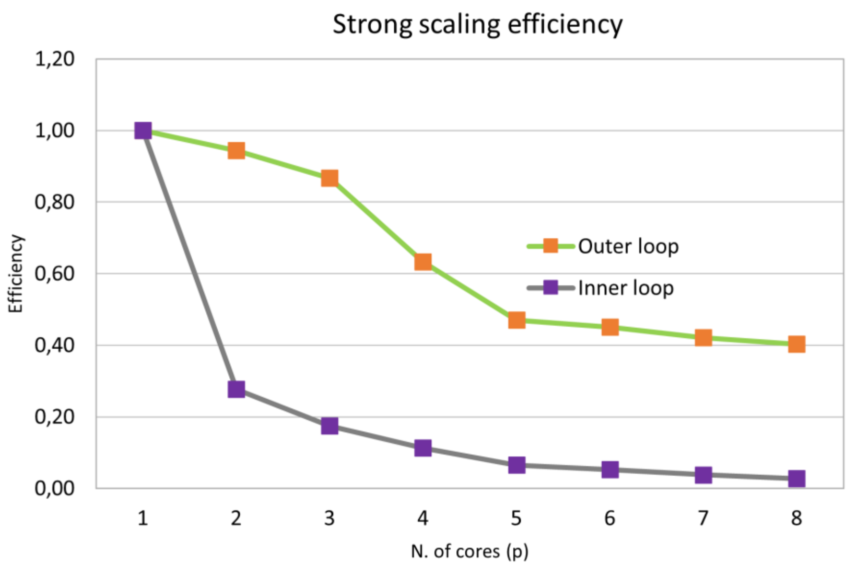

Figure 3: Strong scaling efficiency (N=2000, K=500, R=5)

As it can be seen from the graph below (Figure 3) that even if the strong scaling efficiency of both strategies follows a downward trend, as expected, the parallelization of the outer loop leads to a much higher efficiency w.r.t the parallelization with the reduction clause of the innermost loop.

- **Weak Scaling efficiency.** $W(p) = \frac{T_1}{T_p}$, where T_1= time required to complete 1 work unit with 1 processor and T_p = time required to complete p work units with p processors.

  Weak scaling means to increase the number of processors p keeping the per-processor work fixed. The total amount of work grows as p increases instead, the amount of work for a single processor remains the same as p increases. The aim is to solve larger problems within the same amount of time.

  To keep the per-processor work fixed we have to scale correctly the total problem size. We know that the problem complexity is $O(\sum_{t=1}^{K-1}(N - t(R-1)))$ so the total problem size is approximately equal to $(K-1)N - K(K-1)$, since R is constant. By fixing K we can compute the scaled N as $N_p = (1-p)K + pN$.

  Even in this case the first strategy is much more efficient compared to the other one.

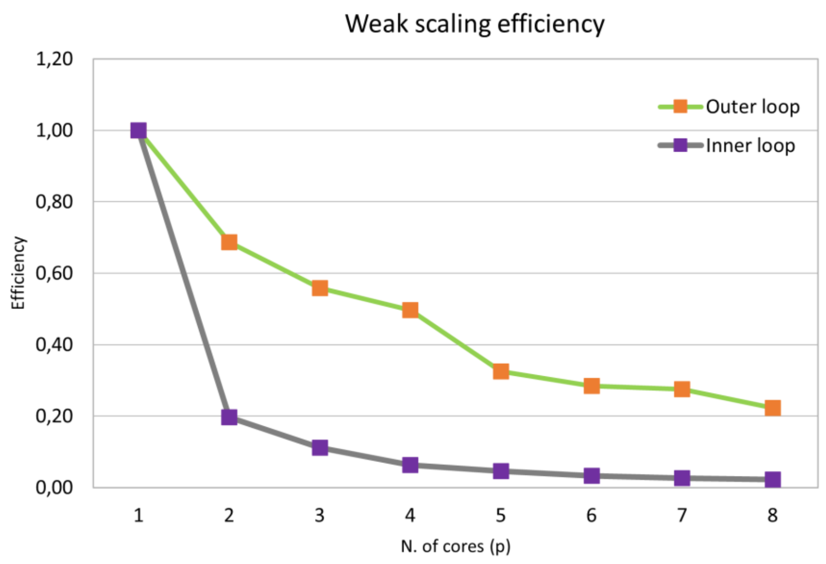

Figure 4: Weak scaling efficiency (Initial N=2000, K=500, R=5)

All the techniques used to evaluate the two strategies led to the conclusion that the program that the parallelization of the outer loop is a better strategy w.r.t the parallelization of the innermost loop with the reduction clause, for this particular problem and the given specifications. It able to exploit better the number of cores available, it has a better strong and weak scaling efficiency.

3. CUDA

In CUDA the parallelization is achieved by launching parallel kernels on the GPU. Hence, the core step is the definition of a kernel function, using the CUDA/C keyword *__global__* , that will be called by the host and execute on the device.

3.1 Parallelization strategies

1. **1D Stencil kernel.** Since each output element is the sum of input elements within a given window then we can see this problem as the application of a stencil pattern. Inside the kernel when we accumulate the partial result, we simply access the global memory directly.

2. **1D Stencil kernel with shared memory.** In this case, to reduce the number of reading operations to global memory, we use the *__shared__* memory to cache the data needed by each block to compute its portion of the result. In particular, inside the kernel, input data is initially copied into shared memory, because they expose a clear reuse pattern, instead, shared memory is not used for the weights, since each weight is accessed only once. All threads are not necessarily fully synchronized so to prevent RAW / WAR / WAW hazards all threads within a block have been synchronized using *__syncthreads()* so when we start the computation we are sure that the shared memory has been filled.

3.2 Performance Analysis

The conventional concept of speedup cannot be used. The program has little or no control over the number of CUDA cores used. The two kernels have been evaluated by testing the achieved speedup vs the best OpenMP CPU implementation, so using the best parallelization strategy and the maximum number of OpenMP threads (8).

All the analysis were performed on an Nvidia GeForce GTX 580 with a total amount of global memory of 3005 Mbytes, 512 CUDA Cores and as maximum dimension size of a thread per block of (1024, 1024, 64). The parameter R and K are fixed ($R = 5, \mathrm{K} = 250$). To have more meaningful and statistically correct results, all the measurements were repeated 10 times and then averaged.

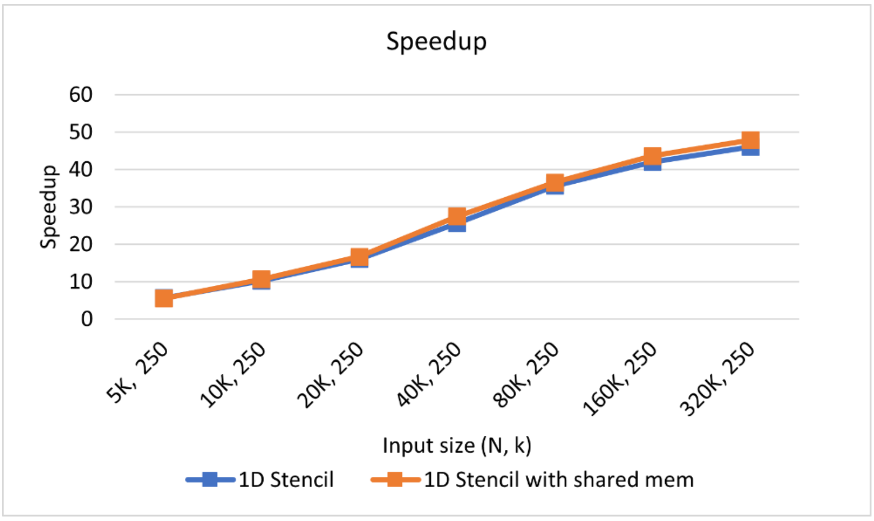

Figure 5: Speedup w.r.t OpenMP implementation

From the graph above (Figure 5) it can be noticed that the two kernels show pretty similar results in terms of speedup w.r.t the CPU implementation. This could be due to the fact that as shrinking factor R we are using a small constant so, the amount of input data reuse is not so significant and so the benefits of the shared memory are limited. Another possible reason is that maybe the number of threads per block ($BLKDIM = 1024$) is too high so the call to *__synchthreads* inside the kernel becomes the bottleneck.

What is sure is that the CUDA implementation is vastly faster than the best OpenMP one, the speedup is directly proportional to the input size and for $N = 320.000, K = 250$ it is around 50 times faster.