Implementation of a Sparsely Connected multi-layer Neural Network (NN) using OpenMP and CUDA

Angely Jazmín Oyola Suárez
angely.oyolasuarez@studio.unibo.it
Architectures and Platforms for Artificial Intelligence, Mod 2, 2020-2021
Prof. Moreno Marzolla
University of Bologna

I ABSTRACT

A Sparsely connected multi-layer neural network has been implemented with use of parallelism by means of OpenMP, CUDA, and C as the programming language. The programs have been deployed on a 6-core Intel i7-4600M CPU and a Nvidia GTX 1070 with CUDA capability 2.0 GPU, more details on the machines can be found in the further sections as well as in the Readme file.
The report has been organised as follows: II Methodology with the project structure, a brief description of the OpenMP and CUDA versions as well as an analysis of correctness, and finally III Results with the performance evaluation of the versions and their approaches.

Keywords: *Artificial Intelligence, Neural Network, C, OpenMP, CUDA, parallelism*

II METHODOLOGY

The proposal implements a multi-layer neural network with parallelism using OpenMP and CUDA in C. The OpenMP version has been firstly developed to establish a solid analysis of the problem, its complexity and the different ways to solve it, and it has been later adapted to CUDA.

A Project Structure

The following headers have been created:

- $Layer(.h/.cuh)$: Struct that represents a layer: its bias, weights and the size of its output.
- $Model(.h/.cuh)$: Struct that represents the neural network: its size, the value of R and the array of layers.

In this proposal, a Neural Network is represented as a Model struct and it is responsible of computing the output by calling the *predict* function, seen below.

```
double* predict(Model nn, double* input) {
  /*...*/
  for (int i = 0; i < nn.size_nn; i++) {
    output = (double*)realloc(output, nn.layers[i].n_neurons * sizeof(double));
    for (int j = 0; j < nn.layers[i].n_neurons; j++) {
      for (int r = 0; r < nn.r; r++) value += input[j+r] * nn.layers[i].weights[j][r];
      output[j] = activation(value+ nn.layers[i].bias);
      value = 0;
    }
    /*...*/
  }
  return output;
}
```

Listing 1: Predict function

As it can be seen, the predict function works with three for loops:

- The outer one that iterates over the L layers of the model. Each layer has a set of weights and biases.
- The intermediate one that iterates over $N - t(R-1)$ output nodes of the layer t, where N is the size of the input.
- The inner one that iterates over the R adjacent input nodes x_{i+r}, relevant to compute the output y_i. Each input is multiplied for its corresponding weight $w_{i,r}$, the result is then accumulated and finally added to the bias and passed through the activation function (sigmoid function, equation 1). To summarize, each output y_i will be calculated with equation 2.

Such function is called for each layer, thus L times. Therefore, we have a problem size p as in the equation 3.

It is worth to mention that both versions, OpenMP and CUDA, share the same structure previously described. The point of difference is the *model* file.

Below, each version will be explained in greater details.

$$f(x) = \frac{1}{1+e^{-x}} \tag{1}$$

$$y_i = f(\sum_{r=0}^{R-1} x_{i+r} \times w_{i,r} + b) \tag{2}$$

$$p = 2R\sum_{t=0}^{L} N - t(R-1) = 2R(LN - \frac{L(L+1)}{2}(R-1)) \tag{3}$$

B OpenMP

The parallelization strategy has been chosen considering the nature of the neural network, since there is a high dependency between the layers, such that the layer L_{i+1} needs the output of the previous layer L_i to can proceed with the forward pass. For that reason, the outer for loop that iterates the layers could not be parallelized.
It means that only the two inner loops, which iterate over the output nodes and the adjacent R input nodes respectively, could be parallelized. Both parallelization techniques are discussed below:

- **The outer for loop**: The *pragma* directive has been added at the beginning of this for loop and the *scheduling* has been set to *static* since the iterations take roughly equal time, with a fixed workload known a prior: R multiplications ($x_{i+r} \times w_{i,r}$) plus R summations of the weighted inputs (bias included) and one *activation*. Thus, in this case static scheduling work best with a little overhead in comparison with dynamic scheduling since it is done at compile-time.

- **The inner for loop**: The $pragma$ directive has been added at the beginning of this for loop and each thread compute a portion of the multiplications which are then reduced in a parallel fashion. Since R is pretty small, it is likely this technique performs worst than the previous.

C CUDA

As it was mentioned before, the CUDA version was based on the OpenMP solution, keeping the main structure considerably similar, and only applying specific changes that will be described in detail below:

- The predict function, already described in the OpenMP version, works as host function and it is responsible of the CPU and GPU memory allocation as well as the computation of the kernel's setting: gridSize, blockSize and sharedMemory. The host is also responsible of launching the kernel for each iteration of the outer for loop (which iterates over the layers).
- The kernel is the responsible of the output's computation and runs on the device (GPU), different performances can be achieved by changing the gridSize ($number\ of\ blocks$), blockSize ($number\ of\ threads\ per\ block$) and the way how the shared memory is used.
- In the OpenMP version the weights have been used as a 2D matrix of dimension $sizeoutput$ x R, however in the CUDA version such matrix has been flattened in a 1D array to facilitate the device access.
- The functions: $CudaCheckError$ and $CudaSafeCall$, available in the header helper_functions.cuh, have been used to check the status of the CUDA API calls.
- Since the shared memory has been used and it is accessed by the threads of a same block, it has been necessary to synchronize the threads before some specific actions in order to avoid race conditions.
- As mentioned before, there are different strategies to tackle the problem, some of them can be achieved while varying the kernel's setting: gridSize, blockSize, shared memory (dynamic/static/memory to be shared). In order to compare the performance of such variations, two kernels have been created and described below:
  - $maxthreads\_compute\_layer$: Launches the kernel with the maximum number of threads allowed by the GPU (1024 in our case) and the minimum number of blocks N, where N follows the equation $(output\_size * R)/BLKDIM$ and $BLKDIM$ (number of threads per block). Therefore, it has been exploited the blocks' shared memory, which has been set to $dynamic$ [1] since is reduced along the forward pass and unknown at compilation time.
    The shared memory stores not only the $input$ but also the $weights$, which have been also used to save the multiplications' partial results: $(x_{i+r} \times w_{i,r})$, which have been computed in parallel by R threads. The final step involving the application of the activation function to the accumulated weighted inputs has been computed using two different approaches:
    - $maxthreads\_sumreduc$: Computes each output by using a parallel accumulation strategy.
    - $maxthreads\_nonsumreduc$: Only a single thread per output element deals with the accumulation procedure.

Limitations : As is know the $GridSizeMax$ and $BlockSizeMax$ are limited and strictly depending on the GPU's architecture. The limits in our case are 65535 blocks and 1024 threads. With these informations we can compute the maximum computable input's size:

$$\begin{aligned} Output's\ size &\leq \frac{GridSizeMax. \times BlockSizeMax.}{R} \\ Input's\ size - (R-1) &\leq \frac{GridSizeMax. \times BlockSizeMax.}{R} \\ Input's\ size &\leq \frac{GridSizeMax. \times BlockSizeMax.}{R} + R - 1 \\ &\leq \frac{65535 \times 1024}{3} + 3 - 1 \\ &\leq 22.369M \end{aligned} \tag{4}$$

– $rthreads\_compute\_layer$: Launches the kernel with N blocks ($gridSize$), where N is as big as the output's size, while the number of threads per block has been fixed to R. It works with a $static\ shared\ memory$ known at compilation time and equal to 2 x R, used to store the corresponding inputs and weights of a block.
Finally, each output is computed by the first thread of the corresponding block.
Limitations : It is worth to mention that there is a huge limitation to consider with this strategy, since the output's size of a layer should be less or equal than the $GridSizeMax$ allowed by the GPU, in our case 65535. It was the first strategy done during the project's development.

In the next paragraph it will be reported the correctness check performed on the described approaches.

D Correctness of the versions and their approaches

To prove the correctness of the different versions: OpenMP, CUDA and each one of their approaches (maxthreads_sumreduc, maxthreads_nonsumreduc, rthreads), their outputs have been compared with the one given by the sequential version applied to three neural networks of input size: 5, 10, 50 and fixed biases, weights and inputs. All of the approaches' outputs matched with the sequential program's, except for the CUDA maxthreads_nonsumreduc, which gave results with an average relative error of %0.00106.
More information about the details of this experiment and the neural network's outputs can be found at:
$/Results/Correctness$

III RESULTS OF THE PERFORMANCE EVALUATION

In this section the performance of the two versions OpenMp and CUDA has been evaluated.
OpenMP: The evaluation of the two strategies has been performed in terms of elapsed time, speedup, strong and weak scaling efficiency with fixed R=3 on a Intel Core i7-4600M with 2 physical cores and 4 logical, averaging the timings over 3 runs for each case.

It has been kept fixed the size of the input N to 5000 as well as the size of the model L to 500, whereas the number of cores (p) varies from 2 up 12 in order to compare the different strategies of parallelism: parallelization of the inner loop + reduction and parallelization of the outer for loop.

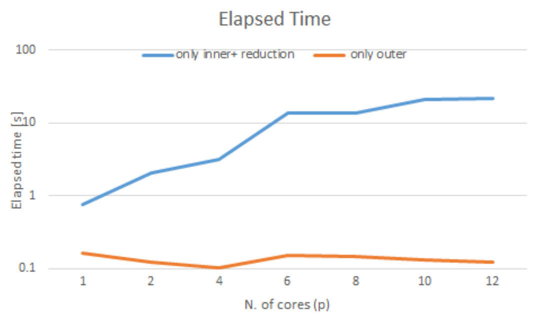

Figure 1: Elapsed time in seconds with N=5000 and L=500

As it can be seen from the log-scaled Figure 1 the parallelization of the outer for loop has shown the least elapsed time. It can be also seen from Figure 2 that such strategy tends to have the best speedup wrt its corresponding sequential program.

Although, the efficiency of both strategies decreases monotonically, the outer for loop seems to be the most efficient. It can be explained given that R is small, the paralellization of the inner for loop will require less resources. However, it can be seen the drop of efficiency of both strategies when p $\geq$ 2, it can be explained by the CPU has only 2 physical cores and it may be added an overhead while using the logical cores.

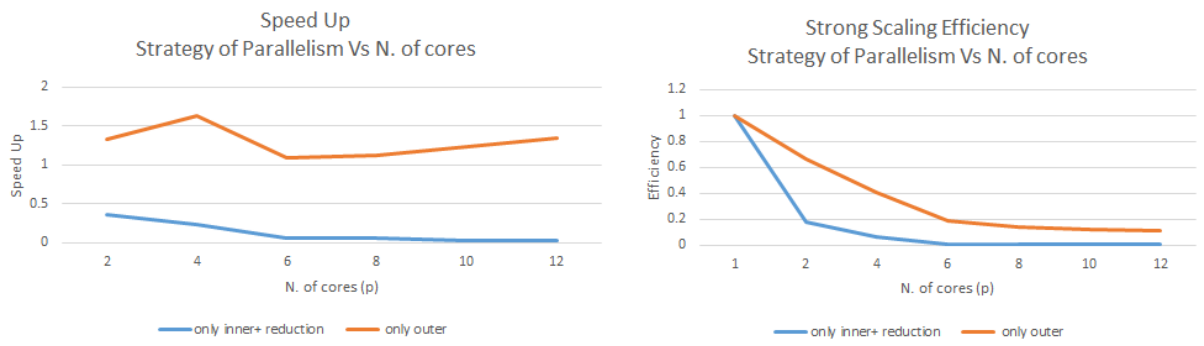

Figure 2: Speed-Up and Strong Scaling Efficiency with N=5000 and L=500

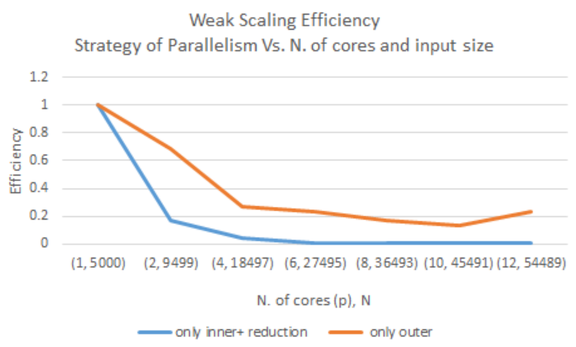

Figure 3: Weak Scaling Efficiency with No=5000, L=500 and R=3

What concerns to the weak scaling efficiency, it has been kept constant the per-processor work as well as the R=3 and the model's size L = 500, but nevertheless the input's size N has been scaled according to the number of cores p in order to make a fair comparison. To achieve this last, it has been calculated firstly the complexity in equation 5 omitting the terms that involve R since it is a constant. Finally calculate the corresponding input N to p in equation 6.

Where N_0 is the input's size of the base case when working with one single core. In figure 3 can be seen again that the outer for loop has a better efficiency and also can be seen the drop of efficiency at p=2.

$$P_{R=3} = 2R(LN - \frac{L(L+1)}{2}(R-1)) \approx LN - L(L+1) \tag{5}$$

$$\begin{aligned} LN_p - L(L+1) &= p(LN_0 - L(L+1)) \\ N_p &= pN_0 + (K+1)(p-1) \end{aligned} \tag{6}$$

CUDA: The evaluation of the three strategies has been performed in terms of computational throughput[2] (equation 7) with fixed R=3 on a Nvidia GTX 1070 with CUDA capability 2.0, BlockSize Máx = 1024 and GridSize Máx.= (1024, 1024, 64), averaging the timings over 3 runs for each case.

It has been kept fixed the size of the model L to 100, whereas the input's size (N) varies from 500 up 1M to compare the different strategies. Where N is the number of elements in the kernel, t is the elapsed execution time in seconds of the kernel, obtained by using the CUDA event API and not the CPU timer since the last needs a host-device synchronization which stalls the GPU.

$$GFLOP/sEffective = \frac{2N}{t \times 10^9} \quad (7)$$

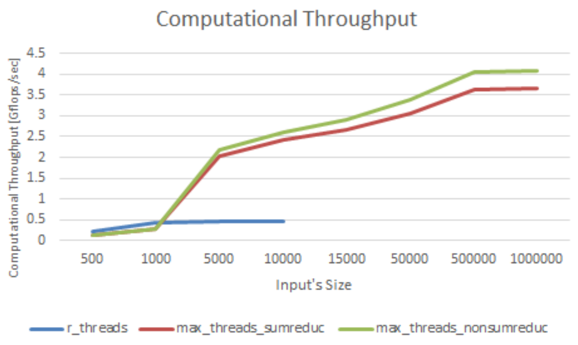

Figure 4: Effective Computational Throughput with different Input's size, L=100 and R=3

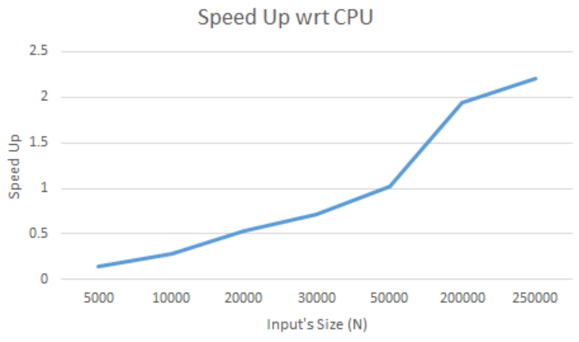

Figure 5: Speedup wrt CPU with different Input's size, L=100 and R=3

It can be seen from Figure 4 that both strategies used in the maxthreads_compute_layer kernel have shown the best throughput in comparison with the $r\_threads$ strategy, it also can be seen their absolute difference increases slowly up that the input's size = 500K where it has started to be constant but less than 0.5. The reason behind the $nonsumreduc$ performs better than the $sumreduc$ could be that for small values of R a sum reduction strategy won't be full exploited but it only adds the overhead of the thread's management. It also can be seen the limitation of the $r\_threads$ strategy with respect to the input's size (up 10K), described in the previous section.

Speed Up wrt CPU (figure 5) was calculated measuring the time required by the $predict$ function of the $max\_threads\_nonsumreduc$ CUDA strategy and the one-core OpenMP version that parallelizes the outer loop. The test was done with a fixed R=3 and L=100 while varying the input's size. We can see that the speed up is over 1x when the input's size (N) $\geq$ 50K and increases up $\approx$ 2.5x when N=250K. The reason behind the absence of gain wrt CPU version on N<50K could be that with few data the GPU's capabilities are not full exploited but there is always the overhead of memory allocation.

Summing up, the CUDA version performs almost twice faster than the one-core OpenMP version for large amount of data.

REFERENCES

[1] N. Mark Harris. (2013). "Using shared memory in cuda c/c++," [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ (visited on 01/08/2021).

[2] ——, (2012). "How to implement performance metrics in cuda c/c++," [Online]. Available: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/ (visited on 01/08/2021).