Final report APAI - Module 1

Gabriele Ceccolini - 0001139333

15 gennaio 2025

1 Dependency Analysis and data preaparation

First, a dependency analysis was performed, which made it possible to identify the parallelizable and non parallelizable parts of the problem.

The neural network is composed of K layers of neurons, where the $0th$ layer represents the input activation values, and the $(K-1)th$ layer represents the output activation values. The size of these layers decreases by $R-1$ neurons for each layer moving away from the first.

The activation value of each neuron from the second layer onward depends on R values from the previous layer and their respective R unique weights. This means that the computation of activation values for each layer depends solely on the values of the previous layer and their associated weights.

Thus, there is a direct dependency between each layer and the one preceding it. More generally, calculating the activation values of any layer requires that all preceding layers have been computed.

The network must therefore be computed sequentially, layer by layer. However, within the computation of a single layer, each activation value is independent of the others—there are no dependencies between the activation values of the same layer.

In conclusion:

- The layers must be computed serially, one at a time, as layers are interdependent.
- The computation of activation values within a single layer can be parallelized.

1.1 Problem Data Setup

The data required by the program to store include the activation values of neurons and their associated weights.

Regarding neuron activation values, we have:

- N values in the $0th$ layer, generated randomly.
- For each subsequent layer, there will be $N - t(R-1)$ activation values to store.
- The total number of neurons, and thus the total number of activation values in the network, will be:

$$tot_{activations} = \sum_{t=0}^{K-1} (N - t(R-1))$$

For the weights, each layer will have R unique weights for each neuron. Therefore, the total number of unique weights in the network is:

$$tot_{weights} = \sum_{t=0}^{K-1} (N - t(R-1)) \cdot R$$

So, having a configuration of N, K, R values in advance we know how many neurons and how many unique weights values there will be for each individual layer.

To maintain a memory allocation that is as simple and minimal as possible, the activation values and weights are stored in two long, contiguous arrays. As a result, offsets will be required to access the desired values, as will be explained in detail later.

a_{00}	a_{01}	$\cdots$	$a_{0(N-1)}$	a_{10}	a_{11}	$\cdots$	$a_{1(N-(R-1))-1}$	$\cdots$	$a_{(K-1)0}$	$a_{(K-1)1}$	$\cdots$	$a_{(K-1)((N-t(R-1))}$

```
float *activations = (float *)malloc(total_neurons * size);
```

From the structure of the activation array above, it can be observed that the values are stored contiguously, starting from the first layer to the last. Within each layer, values are also stored contiguously, from the first to the last neuron in the layer.

w_{00}	w_{01}	$\cdots$	$w_{0(R-1)}$	w_{10}	w_{11}	$\cdots$	$w_{1(R-1)}$	$\cdots$	$w_{(N-1)0}$	$w_{(N-1)1}$	$\cdots$	$w_{(N-1)(R-1)}$

$\cdots \cdots$

$w_{(K-1)0}$	w_{01}	$\cdots$	$w_{0(R-1)}$	w_{10}	w_{11}	$\cdots$	$w_{1(R-1)}$	$\cdots$	$w_{(N-1)0}$	$w_{(N-1)1}$	$\cdots$	$w_{(N-1)(R-1)}$

```
float *weights = (float *)malloc(total_weights * size);
```

As mentioned earlier, for each activation value, there will be R unique weights, each of which will be used only once. These weights are randomly generated at the beginning of the program and stored contiguously in a long array. Access to the desired values during calculations will rely on proper offsets variables.

2 OpenMP Solution

In the OpenMP solution, the basic idea is to use the `#pragma omp for` directive to parallelize the computation of activation values for each output layer. Before doing this, it is necessary to define a generic `compute_layer(...)` function that can compute any output layer of the network while correctly managing the offsets and indices for data access.

```
    for (int t = 1; t < K; t++) {   // from layer 1 to layer K-1
        int input_layer_size = N - (t-1) * (R - 1);
        int output_layer_size = N - t * (R - 1);
        int output_idx = activations_offset + input_layer_size;
        compute_layer(activations, weights, output_layer_size, R,
           activations_offset, weights_offset, output_idx);
        activations_offset += input_layer_size;
        weights_offset += output_layer_size * R;
    }
```

We iterate through each output layer that the network needs to compute, from the 1th layer to the $(K-1)$th layer (the values of the 0th layer are provided as input to the problem). For each iteration, we calculate the size of the input (`input_layer_size`) and output layers (`output_layer_size`) using the previously discussed formulas and update the offset for the position of the output values, `output_offset`.

Finally, we call the `compute_layer(...)` function, passing, in addition to the data pointers and constants, the offsets for accessing the activation and weight arrays.

After the execution of `compute_layer(...)` is complete, I update `activations_offset`, moving it to the first element of the next layer.

Lastly, since each weight is used only once and R weights are accessed for each output neuron, we update `weights_offset` accordingly.

Finnaly, this is the computaion of the output layer:

```
void compute_layer(float *activations,  float *weights,  int
   output_layer_size, int R,   int   activations_offset, int
   weights_offset,  int output_idx){
    float sum;
    #pragma omp parallel for schedule(static) private(sum)
    for(int i = 0; i < output_layer_size; i++){
        sum = 0.0;
        for(int r = 0; r < R; r++){
            sum += activations[activations_offset + i + r] *
               weights[weights_offset + (i * R) + r];
        }
        activations[output_idx + i] = sigmoid(sum + bias);
    }
}
```

Given the nature of the computation, it is natural to parallelize the for loop using the `#pragma omp parallel` for directive. The goal is to optimally distribute the computation of the output neurons of the layer across the available processors.

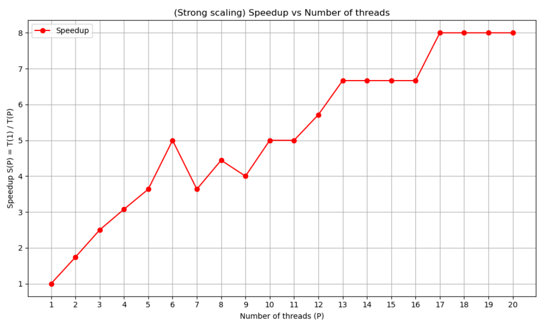
Since the execution time of each iteration is known in advance for every call to `compute_layer()` (exactly R multiplications and accumulations), the `static` scheduling clause was chosen.

With `static`, the workload is distributed to threads in advance in a balanced manner, minimizing overhead.

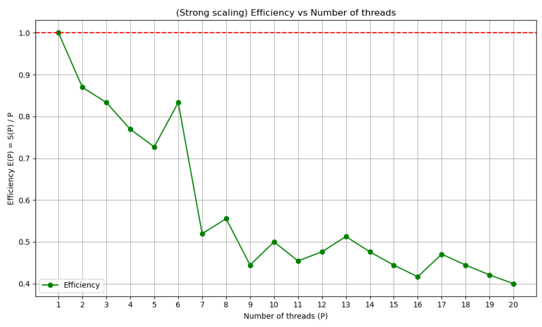
The variable used to accumulate partial data for calculating an output value, `sum`, must be private to each thread. For this reason, the `private(sum)` clause was added to the directive.

To access the correct elements in the `activation[]` and `weights[]` arrays, we use the `activation_offset` and `weights_offset` variables. These respectively point to the index of the first activation value of the input layer and the first weight to use in the current iteration. Combined with index variables such as `i` and `r`, this approach ensures the correct indexing of all values used in the computation.

Below, I briefly discuss the benchmarks of the application with respect to **weak** and **strong scalability**.

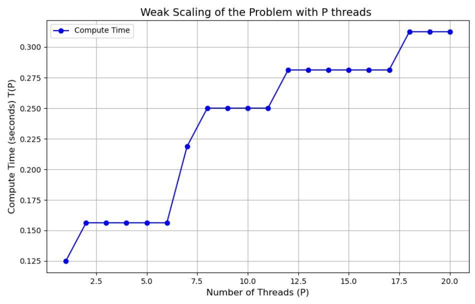


(a) Strong scaling speedup

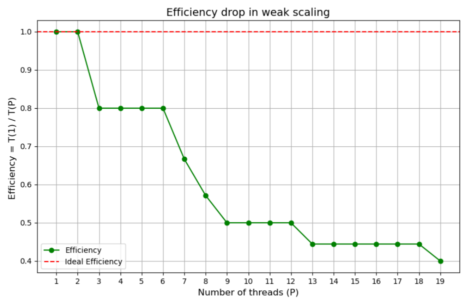


(b) Strong scaling efficiency

Strong scalability is based on running the same problem with a increasing number of processors, analyzing the speedup and the efficiency of the computation. In this case, the problem scales relatively well until we reach the number of physical processors of my machine. The efficiency drops consequently.



(a) Weak scaling execution times



(b) Weak scaling efficiency

The weak scalability study is based on running the problem at increasing sizes, simultaneously increasing the number of processors proportionally. The results are consistent with the previous trend, with scaling clearly capping at the number of physical processors of the machine.

Note that the inconsistency with the trends in the strong scalability study may be due to the specific architecture of my processor. I'm running all the OpenMP benchmarks on my laptop with an *Intel® Core™ i7-12700H* processor with 14 physical cores. These cores are not identical; in fact, they are divided into *Performance-cores* (6) and *Efficient-cores* (8). The maximum number of threads is 20 (which is also the number of processes when the speedup flattens out, which may make sense).

3 CUDA Solution

In the CUDA solution was possible to leverage heterogeneous computer architectures to handle larger problem sizes. In fact, when working with OpenMP, we were limited by the computational power and memory of our host machine. For my tests, I used an *Intel i7-12700H* CPU (6 performance cores + 8 energy-efficient cores) and 16GB of RAM.

However, when working with the DEI cluster, we had access to an *NVIDIA L40 GPU* (18,176 CUDA cores) with 48GB of RAM. This allowed us to organize the code so that all activations and weights (excluding the input and output layers of the activation values) were stored only in the GPU memory space. By doing so, I was able to test the code with much larger problems, which were previously constrained by memory limitations.

Similar to the OpenMP solution, activations and weights were organized into two large contiguous arrays. However, in this case, the allocation was performed in GPU memory. Using `cudaMalloc`, we allocated both arrays and initialized them directly on the GPU:

```
cudaMalloc(&weightsGPU, total_weights * size);
cudaMalloc(&activationsGPU, total_neurons * size);
```

The computation strategy was straightforward: GPUs and CUDA exploit parallelism at the thread level, so the objective was to decompose independent work into as many threads as possible.

It is already known that the computation of output neurons in the same layer is independent of one another. This independence allows to delegate the computation of each output neuron to a separate thread. For each output layer, I launched as many threads as there are neurons to compute. The GPU's multithreading hardware takes care of the actual scheduling. In theory, if the GPU has enough available CUDA cores, all threads could run in parallel, fully exploiting the massive parallelism of the GPU.

After testing on both GPUs (RTX 2080 and L40), the optimal `blockDim` value was found to be 512. The number of blocks to launch `numBlocks` is calculated automatically.

```
    for (unsigned long int t = 1; t < K; t++) {
        //...
        unsigned long int numBlocks = (output_layer_size +
           threadsPerBlock - 1) / threadsPerBlock;
        compute_layerGPU<<<numBlocks, threadsPerBlock>>>(
           activationsGPU, weightsGPU, output_layer_size, R,
           activations_offset, weights_offset, output_idx);
        //...
        activations_offset += input_layer_size;
        weights_offset += output_layer_size * R;
    }
```

At the end of each layer computation, the offsets are updated as in the OpenMP solution. Note that the kernel call is a synchronous operation: when it returns, we know that the computation has been completed.

The kernel for the computation of an output layer is shown here:

```
__global__ void compute_layerGPU(float *activations, float *weights
    ,unsigned long int next_layer_size, int R, unsigned long int
    activations_offset, unsigned long int weights_offset, unsigned
    long int output_offset)
{
    unsigned long int idx = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    for (int r = 0; r < R; r++) {
        sum += activations[activations_offset + idx + r] * weights[
           weights_offset + (idx * R) + r];
    }
    activations[output_offset + idx] = sigmoid(sum + bias);
}
```

The only difference between this and the `compute_layer(...)` function of the OpenMP solution is the indexing of the current neurons to compute. To do this, I locate the unique index of the thread in the grid with `unsigned long int idx = blockIdx.x * blockDim.x + threadIdx.x;`, so the thread in question needs to take care of the idx-th output neuron of the layer. Once this is done, to access the correct values for weights and activations, I used `activation_offset` and `weights_offset`, just as in `compute_layer(...)`.

Now, finally, we can copy back the output layer data of the network to the CPU for displaying the output results.

```
cudaMemcpy(final_layer_activations, &activationsGPU[activations_offset], layer_size
 * sizeof(float), cudaMemcpyDeviceToHost));
```

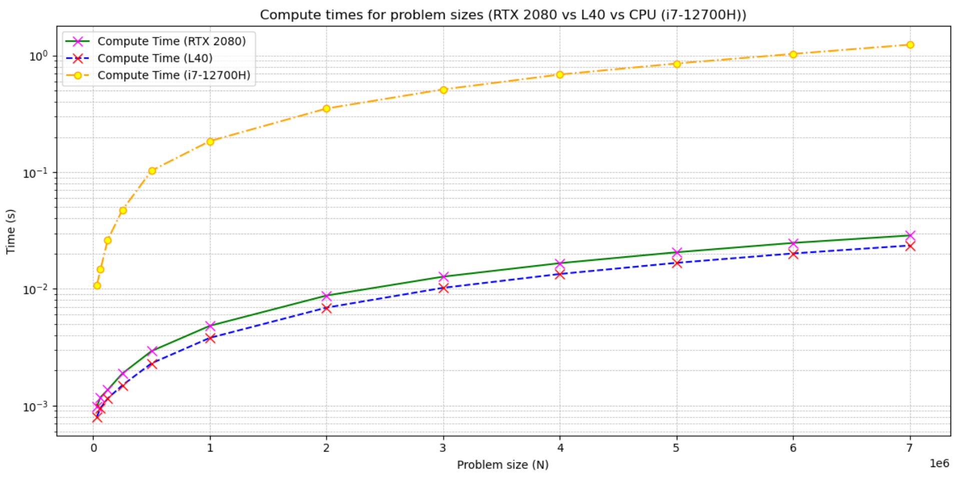


Figura 3: Compute times for problem sizes, RTX2080 vs L40 vs CPU (i7-12700H)

From these final results, Im able to say that the compute times scales linearly with the size of the problem in CUDA. Compared to the OpenMP performance on my local machine's processor, I observed a huge speedup with CUDA ($\approx$ 50 times faster than the OpenMP version, running with 20 OpenMP threads on *i7-12700H* with 14 physical cores). This is due to the intrinsic nature of the problem, which scales very well with many-core architectures.