ARCHITECTURES AND PLATFORMS FOR ARTIFICIAL INTELLIGENCE

Module 2

Giacomo D'Amicantonio

The assignment of the second module of the course of APAI required to implement a function to compute the sigmoid activation function of a Neural Network with multiple layers using OpenMP and CUDA parallelization methods. In this report, I will discuss about the two implementations and their performances. The code for the CUDA implementation was ran on Colab using a Nvidia Tesla V100S – 16 GB GPU, while the OpenMP implementation was run on a PC with a i7 10^{th} generation CPU with 4 cores/8 threads and 16GB of memory.

CUDA

Introduction

In the CUDA implementation I fixed the number of blocks and the number of threads at 1024, which seems to be the maximum number of threads for the GPU in use. An array of random floats defines the input layer's values and, for every hidden layer, I created a random array of weights and a random bias. The user can insert the dimension of the starting layer and the number of hidden layers from the terminal.

The kernel function uses a similar approach to the one we used for the *Stencil1D* example, modified for this specific setting. The layer and its weights are indexed using the thread ID, the block ID, the block dimension, and the value of R, that is the number of neurons to be used to compute one neuron of the following layer. I have used an array of floats shared between the threads to store the result of the product between a neuron's value and its weight. This required to synchronize the threads right after each neuron was processed.

The kernel function also contains an implementation that is used only if it is using 1 block and 1 thread. I used it to perform some tests and compare performances with the parallelized solution.

Memory allocation

I want to briefly discuss the allocation of memory on the GPU. The function *cudaMalloc*, similarly to the classical *malloc* function used by C/C++, allocates the memory required by the data to be transferred to the GPU. The first time this function is uses it also serves as the first call to a Cuda API, which means that the GPU has to setup the CUDA sub-system. This requires significantly more time than the all the following calls. In my case, the difference between the *cudaMalloc* of the input layer and the one of its weights was about 0.18 seconds

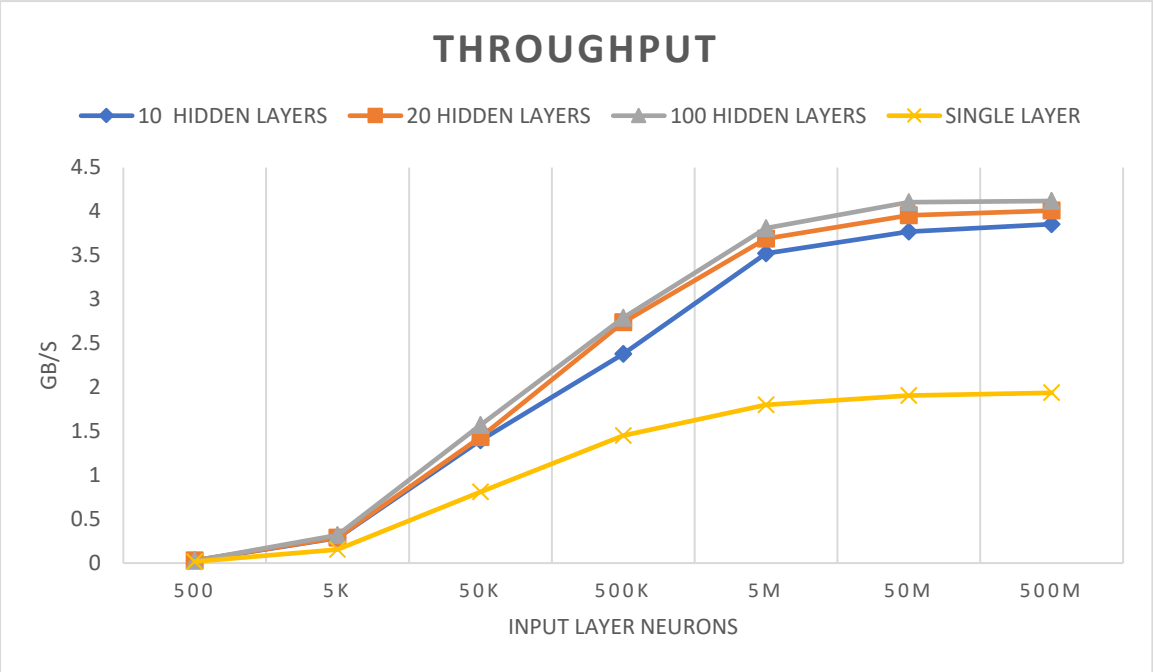
(0.1863s vs 0.0022s). For this reason, my code allocates a fake array in the beginning and deletes it immediately, so the computation of the performances is not influenced by it.

Throughput

Since the code cannot control the number of CUDA cores that will be used to execute the computation, it is better to use throughput instead of speedup to evaluate the performance of the program. The throughput is defined as:

$$Throughput = \frac{\#\ Elements\ processed\ \times 4\ (bytes)}{Seconds}$$

I have run multiple tests, using different dimensions for the input layer and the number of hidden layers. The results are shown in the following plot.

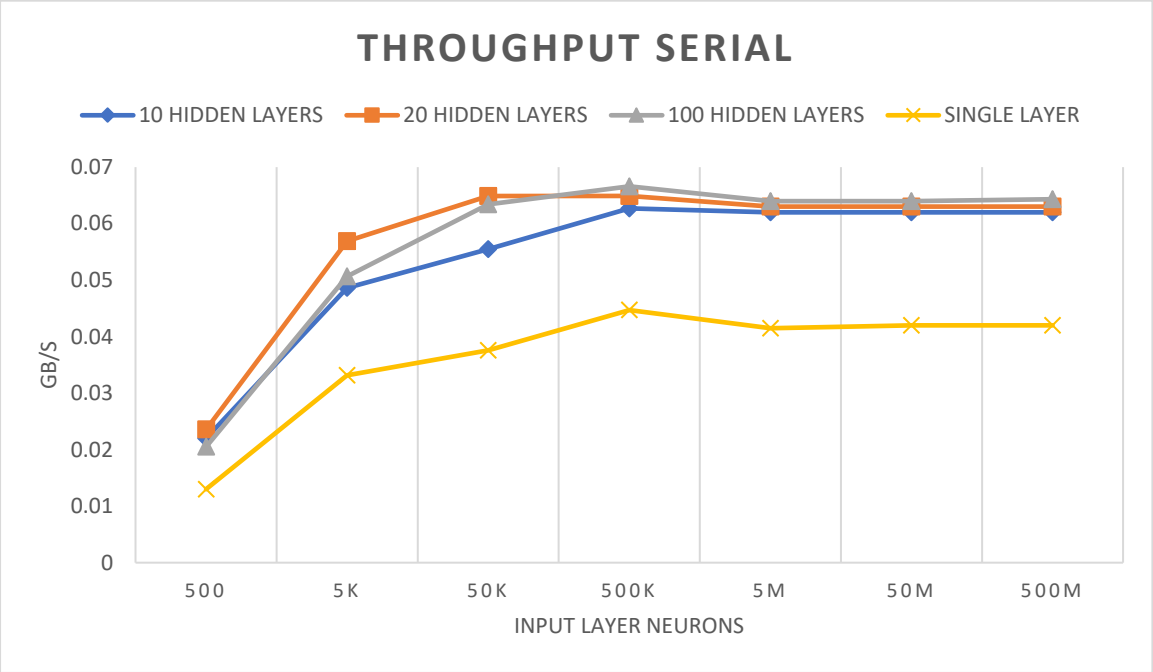


We can see how the throughput generally increases with the number of elements processed, but not in a proportional fashion. The difference between the throughputs of the three multi-layer cases, for each input layer dimension, is about the same. It is relevant to see how, considering only one layer with 500M neurons, the throughput is almost half of the throughput of the 100 layers, 500k neurons in the first layer case even though the number of processed elements is higher.

I have used the *nvprof* profiler to look deeper into it, especially to understand how much time the GPU spends on the actual computations. It showed that, on average, about 55% of the time is spent transferring data from the host to the device, about 44% is spent transferring data from the device to the host and less than 1% is spent on computations of the kernel function. One possible explanation would be that, even though the GPU has enough parallelization and computing capacity to efficiently perform the sigmoid function on large layers, the memory cannot transfer data fast enough to fully exploit it. But once we reach the case with 500M neurons, the GPU seems not able to store the whole layer, its weights, and

the following layer in its memory, causing the throughput to plateau. This is most likely due to the Colab environment since the GPU has 16GB of memory and only 6GB are required.

By comparison, I show the plot of the throughput without parallelization, meaning with number of blocks and number of threads set to 1.



OPENMP

Introduction

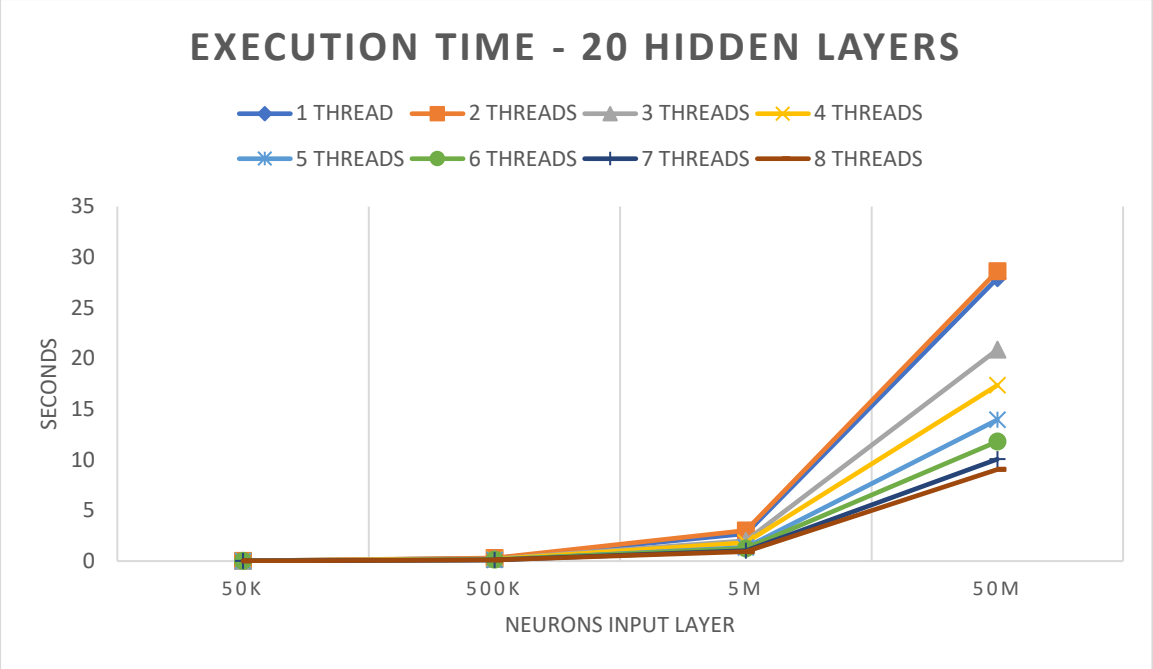
The OpenMP implementation has the same setup of the CUDA, so the values of the input layer, its weights and its bias are assigned randomly. The *"for"* loops in the function to compute the following layer are parallelized over multiple threads. Each thread has its own temporary variable to store the partial sums, while the array of values of the layer to be computed is shared between them. The number of threads was empirically set to 8, which appears to be the best performing case.

Speedup

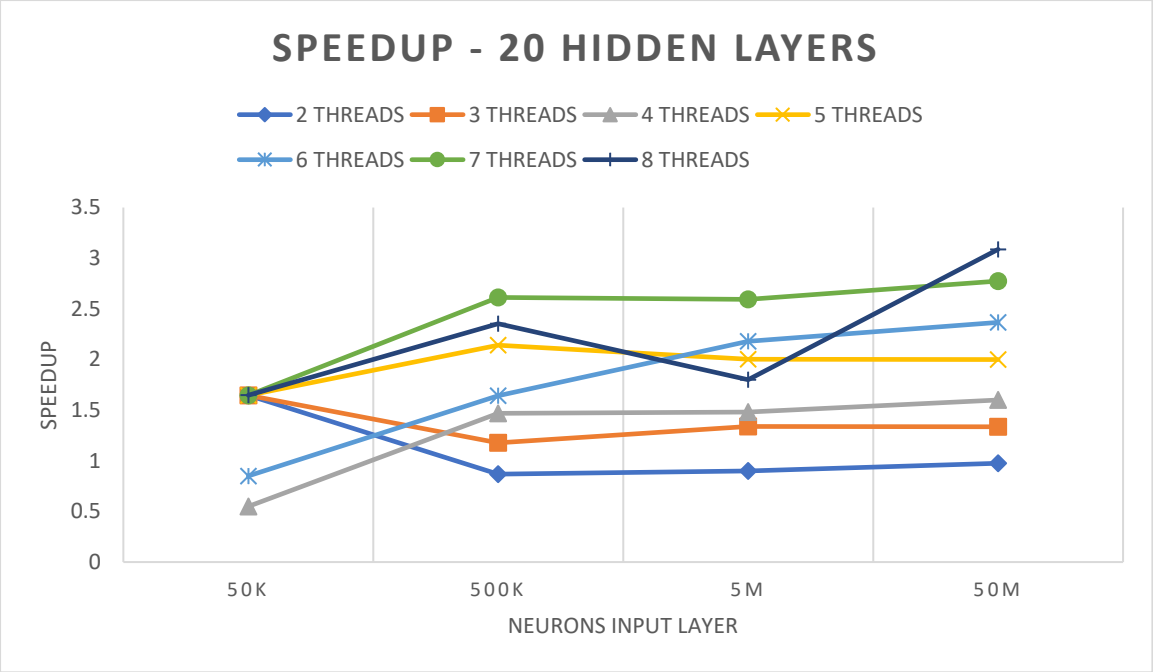
The speedup is defined as

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)} \approx \frac{T_{parallel}(1)}{T_{parallel}(p)}$$

I have run multiple tests using a different number of processors, using *#pragma omp parallel num_thread(p)* as shown in the plots below. The plot takes into account only the time required to execute the activation function.



The best performing setup seems to be the one with 8 threads, which is also the number of threads of the CPU. The speedup for each case is shown in the following plot:



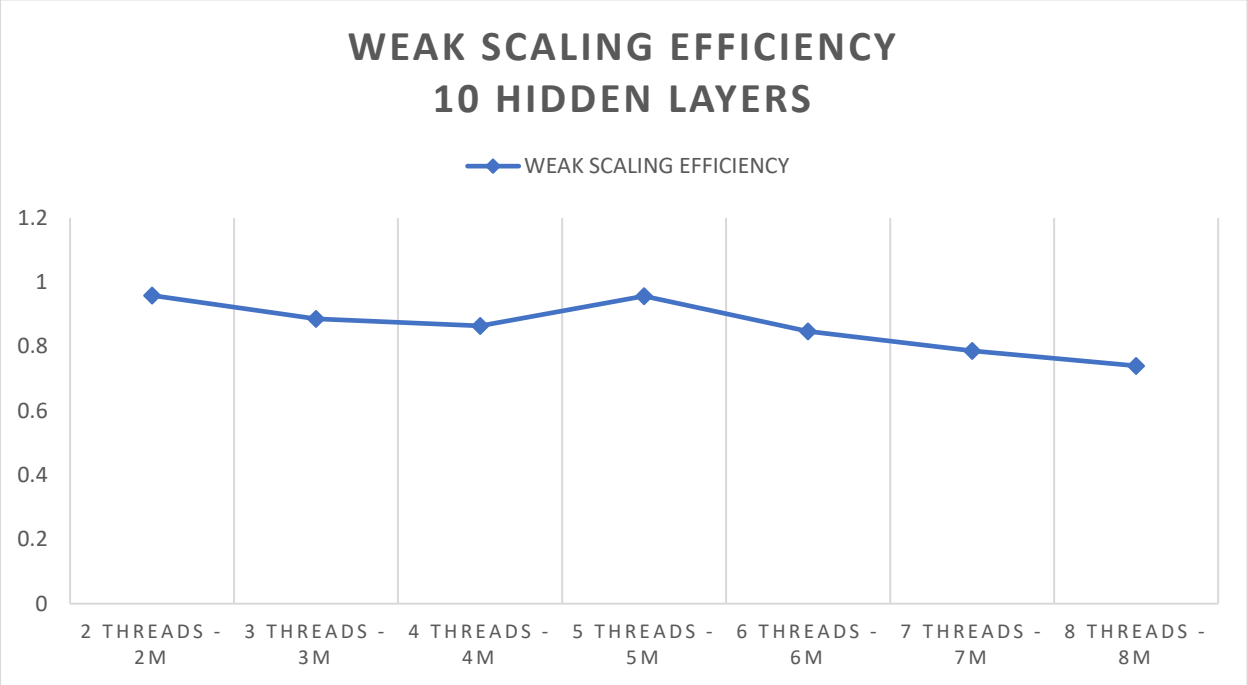
The best speedup is generally obtained by the 7 threads case.

I performed some test to understand also if and how much the number of hidden layers impacted the performances. The results confirmed that the best performing cases are the 8 threads and the 7 threads.

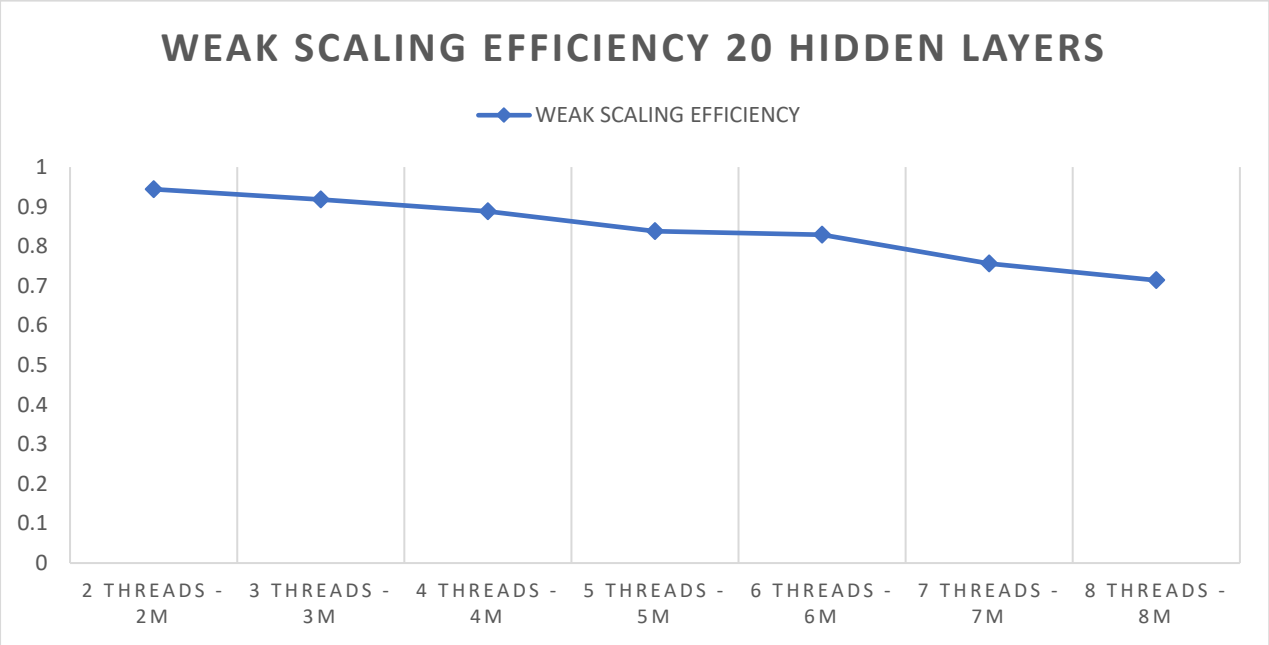
Weak Scaling Efficiency

The Weak Scaling Efficiency is a value that helps understand the scalability of the program. It is defined as:

$$W(p) = \frac{T_{1\ unit\ of\ work\ with\ 1\ processor}}{T_{p\ units\ of\ work\ with\ p\ processors}}$$



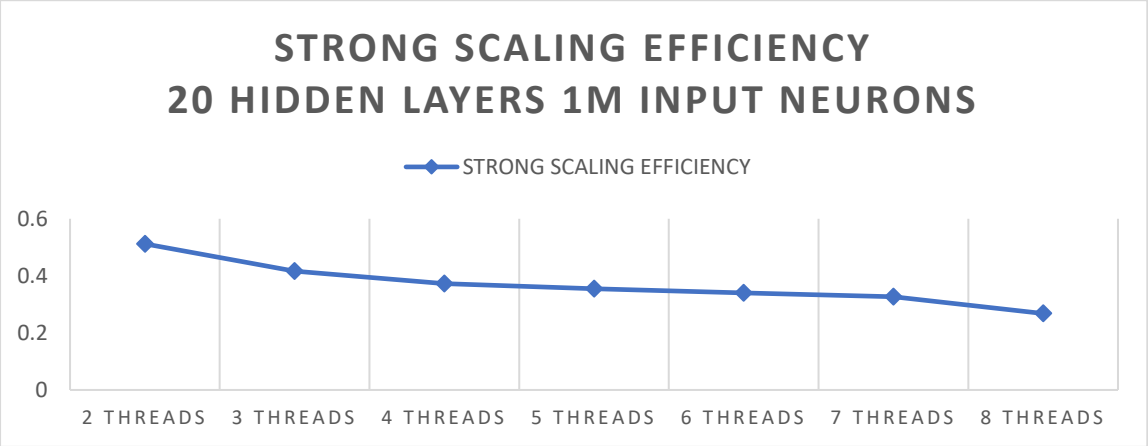
In my tests I have found that this value decreases slowly but steadily as the number of threads and the dimensions of the input grow. This means that the implementation may not be optimal. I have also done some tests using OpenMP's *schedule* to manually or dynamically schedule better the execution of the threads but it seemingly did not improve the performances.

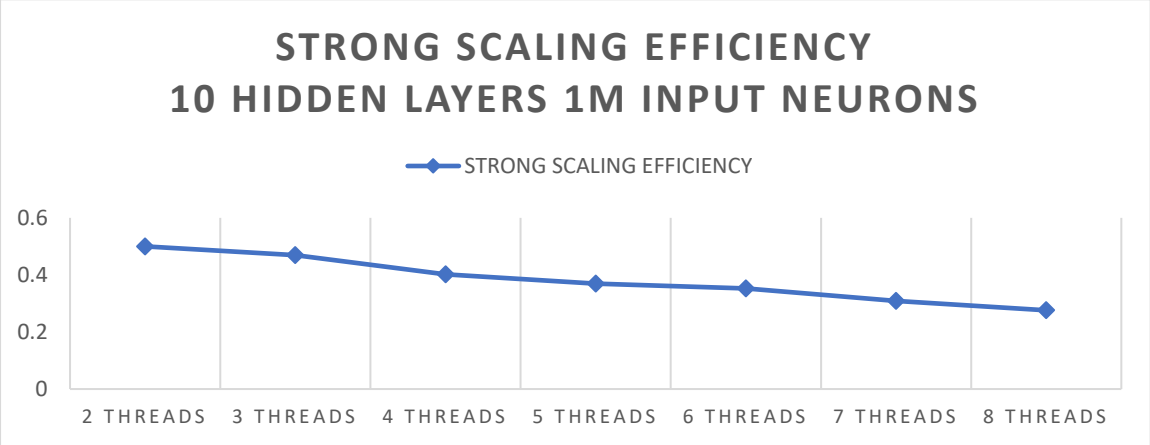


Strong Scaling Efficiency

The Strong Scaling Efficiency is the opposite of the Weak Scaling Efficiency. Instead of increasing the dimension of the input according to the number of threads, we keep it fixed. It is defined as:

$$E(p) = \frac{T_{parallel}(1)}{p \times T_{parallel}(p)}$$





In the plots we can see how the Strong Scaling Efficiency is not influenced by the number of layers and decreases steadily as the number of threads goes up. Further tests showed that it decreases steeply for more than 8 threads, once more confirming that 8 threads is the best choices for the implementation.