# Project Report

## Architecture and Platforms for AI (M2)

by Hanying Zhang

August, 2021

## 1 Brief Problem Description

In this project we implement a simplified Neural Network. For each layer, the difference with a standard fully connected layer is that, when computing the value of each neuron node in the output layer, instead of using all neurons in the input layer, we use only R neurons, as shown in the figure below. The activation function we use here is *Sigmoid*.

We then combine K such layers to form a Neural Network. The W and b parameters of each layer may be different while R is the same for all layers.
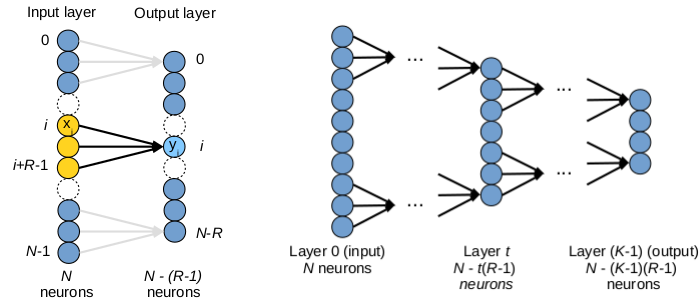


**Figure 1.** One layer and K layers of the NN

## 2 Machine Specific

The whole project was written and tested on my local machine. The CPU is a i7-6700HQ with 4 physical cores. It also uses *Intel*'s *Hyper-threading* technology which needs to be turned down because the performance of the virtual cores is only 15-30% of the physical cores on average.[1]

The GPU is a Nvidia GTX 1060 with 6 GB global memory. The specifications is shown in the image below:

---

1. https://en.wikipedia.org/wiki/Hyper-threading

**Figure 2.** GTX 1060 specifications

The operating system is Ubuntu 20.04.1 LTS. The GCC version is 9.3.0. The CUDA version is 11.3 and NVCC version is 10.1.243.

# 3 Base(Serial) Implementation

The main purpose of the serial implemention is to check the correctness of the OpenMP and CUDA implementations. It also provides the skeleton for the parallel implementations.

The serial implementation is very straitforward. It's basiclly a 3-layer nested loop. In the main function, all K layers are looped over. For each layer, there's a function to perform the culculation in which there's a loop over all the neurons in the output layer. For each neuron, a loop over the R corresponding neurons in the previous layer is implemented.

The header file *hpc.h* provided during the lectures is used in this assignment to provide facility functions such as time recording.

## 3.1 Parameter Parsing

The C library function named *getopt*[2] which is included in *unistd.h* is used for parameter parsing. It is a part of the POSIX specification, and is universal to Unix-like systems. For the serial and CUDA implementations, the parameters N and K are parsed by *-n* and *-k* options respectivelly. For the OpenMP implementation, the number of threads is also parsed by a *-t* option.

---

2. https://en.wikipedia.org/wiki/Getopt

## 3.2 The Memory Management Method

In the first versions, one array in the stack is used to store the latest layer got calculated. Then the *N-t(R-1)* elements from the beginning of this array are re-used when calculating the *t-th* layer. The benefit of this method is the small memory costage. But it is very different from the real life implementation of Neural Networks. For instance, if we want to extend our project to be able to do the back propagation in future, then we will be lack of information.

So the memory management method is changed by storing all the parameters and results in the heap. In this method we consumed more memory but we also kept the necessary information.

Specifically, one array named y is used to store all layers of results, input layer included. Two other arrays named b and W are used to store all the bias and weights respectively. When doing the calculation of the t-th layer, the starting indices of the corresponding data are calculated.

TODO: Illustate the diagram

# 4  OpenMP Implementation

The OpenMP implementation is very similar to the serial one. In the layer claculation function, 2 *omp for* under *omp parallel* are used. The first one is used to do the MAC calculation in which an *array reduction* and a *collapse(2)* is used. The second one is used to do the *Sigmoid* function. For both of them a *static schedule* is used because they are pre-determined and predictable work.

TODO: Try different schedule strategies

# 5  CUDA Implementation

When doing the CUDA implementation, the first idea came into mind is using one block to calculate one neuron in the output layer. The problem with this method is that most threads in each block are wasted if R is a small value. TODO: smaller than 32?

So using one block to calculate more than one neuron is a better idea. The BLKDIM is defined to *(n_nodes/R\*R)\*R* in which *(n_nodes/R\*R)* is the number of neurons being able to be divided by *R*.

One array named *local_y[BLKDIM]* is used in the shared memory to store the values of multiplications of x and W. Then these local values are accumulated and feed to the Sigmoid function. The Sigmoid function is a device function in which *expf()* is used instead of *exp()*.

# 6  Correctness Checking

The method we use is to make sure that the OpenMP version and CUDA version could get the same result as the serial version.

In order to do this, we must make sure that the initial values of the parameters are all the same in these 3 implementations. A random seed is used to make sure the random values are the same. We also have to make sure that all values are initialized in the exactly same order.

One function named *random_init_small()* is used to provide random float values between -1 and 1.

# 7  Performance Analysis

In order to get the result as accurate as possible, text mode is activited instead of graphic mode to avoid the interruptions of other applications not necessary. For each time costage, I run the code for 5 times and used their mean result.

What we need here is **Wall Clock Time** instead of **CPU Time**, thus the *clock()* function defined in *time.h* is not suitable. What used in this assignment is the *hpc_gettime()* function in *hpc.h* provided in the course.

## 7.1 OpenMP Performance

### 7.1.1 Strong Scaling

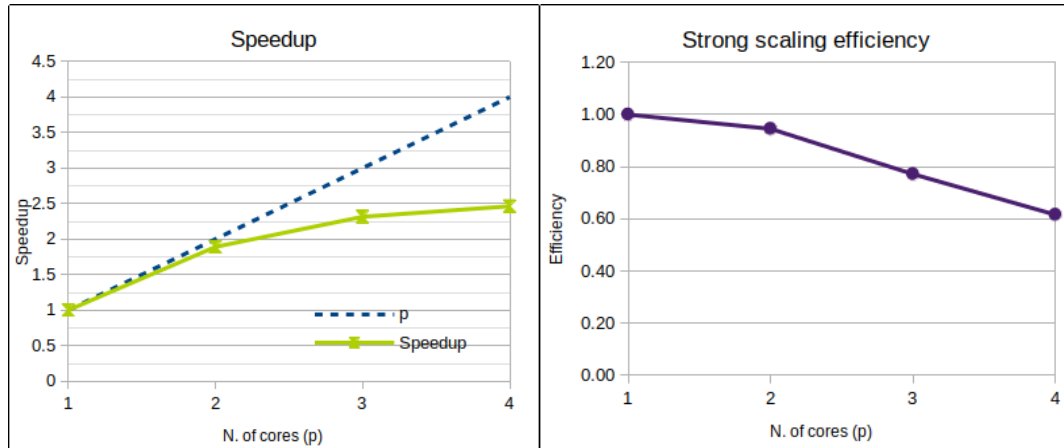For strong scaling of OpenMP implementation, the parameters used are N=2M and K=100.



**Figure 3.** OpenMP Strong Scaling Performance
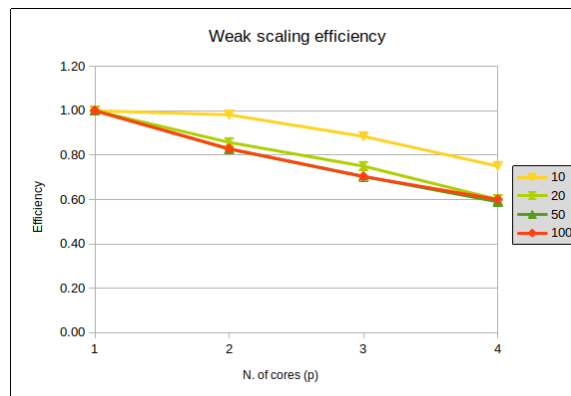
### 7.1.2 Weak Scaling

The parameters used is N=24k.



**Figure 4.** OpenMP Weak Scaling Performance

## 7.2 CUDA Performance

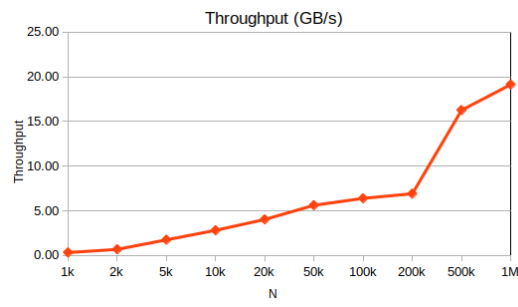The parameter used here is K=300.

### 7.2.1 Throughput

say something

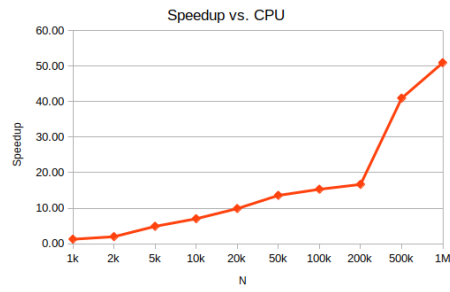**Figure 5.** CUDA Throughput

### 7.2.2 Speedup vs. CPU

say something



**Figure 6.**