# Project Report

## Architecture and Platforms for AI (M2)

by Hanying Zhang

July, 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 left figure below. The activation function we use here is Sigmoid.

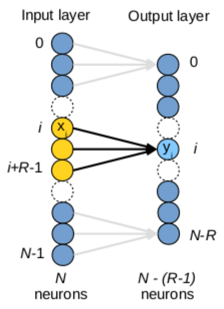


**Figure 1.** One layer of the NN

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.

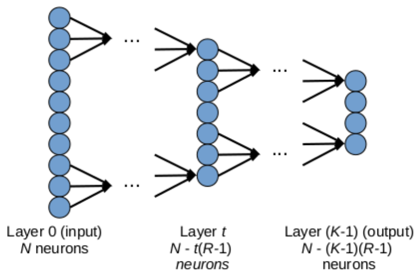
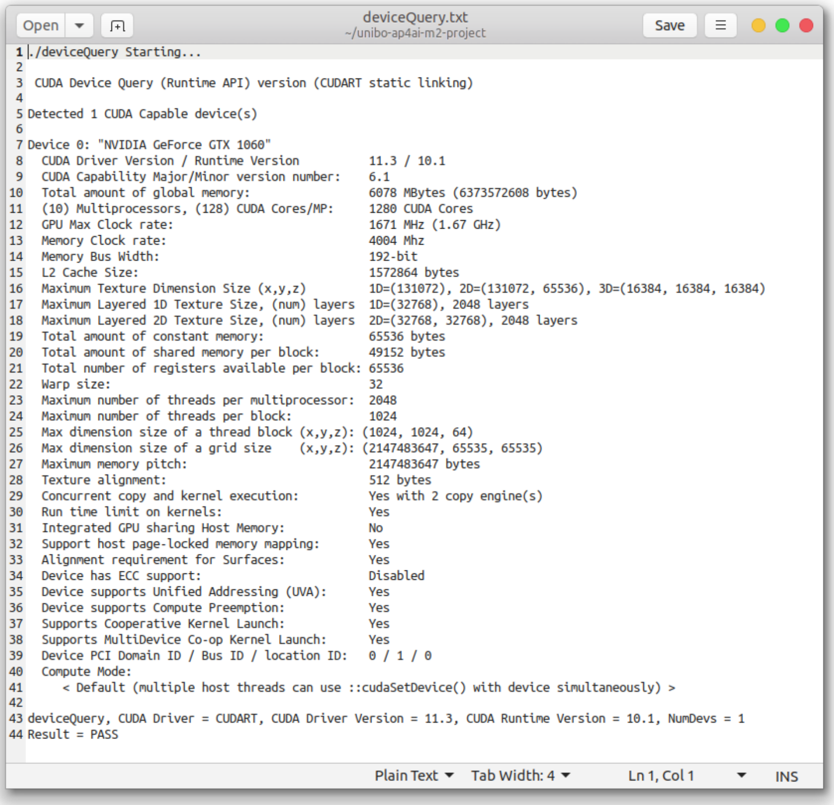


**Figure 2.** The NN with K layers

## 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 need to be turned off because the performance of the virtual cores is only 20% of the phisycal cores on average.

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



```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce GTX 1060"
  CUDA Driver Version / Runtime Version          11.3 / 10.1
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6078 MBytes (6373572608 bytes)
  (10) Multiprocessors, (128) CUDA Cores/MP:     1280 CUDA Cores
  GPU Max Clock rate:                            1671 MHz (1.67 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.3, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
```

**Figure 3.** GTX 1060 specifications

The memory of this machine is 16 GB xxHz dual channels.

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

## 3 Serial Implementation

The maim purpose of the serial implemention is to check the correctness of the OpenMP and CUDA implementations.

The serial implementation is very straitforward. In the main function, I loop over all K layers. For each layer, I wrote a function to perform the culculation in which I loop over all the neurons in the output layer. For each neuron, I loop over the R MACs.

## 4 The Memory Management Evolution

In the first versions of both the OpenMP and CUDA implementations, I used one array in the stack to store the latest layer result got calculated. I re-used the *N-t(R-1)* elements from the beginning of the array when the *t-th* layer got the result. 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, in future, we want to extend our project to be able to do the back propagation, then we will be lack of information.

So I changed the memory management method by storing all the results of all layers in the heap. In this method we consumed more memory but we also kept the necessary information.

The second benefit of this method is that we reduces the number of times calling CUDAMemcpy. In the first version, we need to call CudaMemcpy *K-1* times because for each layer we need to do that. In the second version, we can do that once for all. As the CudaMemcpy costs lots of time, the total time costage using the second version could be much smaller.

The performance of this method is not good, cudaMemcpy cost lots of time.

So the idea is to reduce the times of the mem cpy between host and device.

## 5 OpenMP Implementation

The OpenMP implementation is very similar to the serial one.

TODO: only use FOR seems very little knowledge about OpenMP. Schedule? Reduction?

Introduce how to get the parameters of N and K from terminal

## 6 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. In this way we can also use the shared memory in each block.

## 7 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 versions. I use a random seed to make sure the random values got are the same.

I also manully calculated the result of N=7 and K=3 to make sure that the serial implementation is also correctly calculated.

## 8 Performance Analysis

In order to get the accurate result as much as possible, I switched off all 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 I used is the *hpc_gettime()* function in *hpc.h* provided in the course.

### 8.1 OpenMP

#### 8.1.1 Strong Scaling

#### 8.1.2 Weak Scaling

### 8.2 CUDA

#### 8.2.1 Strong Scaling

#### 8.2.2 Weak Scaling