NEURAL NETWORK IMPLEMENTATION USING CUDA

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Nishant Kumar
Roll No. 801732032

Under the supervision of:
Dr. V. P. Singh
Associate Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
PATIALA – 147004

June 2019

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Neural Network Implementation using CUDA*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar Institute of Engineering and Technology, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. V.P. Singh* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Nishant Kumar.

Signature:

(Nishant Kumar)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. V.P. Singh)

Associate Professor, CSED

ACKNOWLEDGEMENT

The successful completion of any task would be incomplete without acknowledging the people who made it possible and whose constant guidance and encouragement secured the success.

First of all, I wish to acknowledge the benevolence of omnipotent God who gave me strength and courage to overcome all obstacles and showed me the silver lining in the dark clouds with the profound sense of gratitude and heartiest regard. I express my sincere feelings of indebtedness to **Dr. V.P. Singh** for their positive attitude, excellent guidance, constant encouragement, keen interest, invaluable co-operation, generous attitude and above all their blessings. He has been a source of inspiration for me.

I am grateful to **Dr. Maninder Singh**, Head of Department and **Dr. Ashutosh Mishra**, P.G. Coordinator, Computer Science and Engineering Department, Thapar Institute of Engineering and technology for the motivation and inspiration for the completion of this thesis. I will be failing in my duty if I don't express my gratitude to **Dr. S.S. Bhatia**, Senior Professor and Dean of Academics Affairs in the institute for making provisions of infrastructure such as Library facilities, Computer Lab equipped with internet facility immensely useful for the learners to equip themselves with latest in the field.

Nishant Kumar

Nishant Kumar

(801732032)

ABSTRACT

With the invention of cheaper GPU hardware and easy to use Application Programming Interface (API) like NVIDIA CUDA, which is an API developed by NVIDIA for the parallel computation on the GPU. Since for the large neural network, there is a need of powerful machines which will perform computations. But those high-performance machines are not available easily in day to day life. Any programmer who wish to compute a large neural network will seldom find work time on the high-performance machines. So, the Graphics Processing Unit which comes with the personal computer is not use for graphics purpose only. We can harness the power of that GPU to perform the complex computations using parallel programming. NVIDIA provides the API for the programming on GPU which follow the CUDA (Compute Unified Device Architecture).

In this thesis an Artificial Neural network consisting of Linear Layer, Sigmoid Layer and RELU Layer is implemented using NVIDIA CUDA programming model. This Artificial Neural Network is developed for Binary Classification Problem which classifies the two-dimensional data points. A total of 21 batches of two-dimensional data points with 100 data points in each batch is fed to the network. Here, 20 batches are used to train the network and last batch is used for testing. This Artificial Neural Network running on NVIDIA GeForce RTX 2080Ti GPU gives better performance when compared to the same neural network developed using already present machine learning API and this network is run on DualCore i3 4005 Intel CPU. The results demonstrate that the CUDA improves performance compared to the equivalent CPU.

Keywords: Artificial Neural Network, CUDA, GPU, NVIDIA, Parallel Computations

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Artificial neural networks are a domain of computational tools which is used in fields like pattern recognition, machine learning, and knowledge discovery. It is influenced by the inherent characteristics of actual neurons which is available in the human brain. A very tough task is performed by tiny element which work together, these are called neurons. Example includes, the classification of data in which, after training the artificial neural network we can find the relationship between massive and multiple dimension dataset. Training procedure of neural network can be very slow which depends on the size of the data, that can stop the usage of the training, so development of procedures for increasing the training rate can bring large and more difficult data sets to be worked upon. To analyze more massive and more complex dataset, a high-performance computing environment is required. Different approaches have been used in the past decade to reduce the time taken for the training process. With the invention of the cheaper GPU (Graphics Processing Unit) hardware and easy to use parallel programming APIs (Application Programming Interfaces) like NVIDIA CUDA provides high performance computing environment for more complex datasets. This thesis describes the approach which uses GPGPU with CUDA APIs provided by NVIDIA to train the artificial neural network and the comparison of time taken to train the network on GPU and CPU. In this chapter, all the related information of CUDA architecture and ANN (Artificial Neural Network) is presented. First CUDA architecture and then ANN is explored in brief.

1.1 BACKGROUND

The processors nowadays have thousands of cores which allows them to compute in parallel on the multicore CPUs and GPUs. Rate at which the program executes will be increased if software takes advantage of the parallel nature of the multiprocessor. So that is why a significant requirement for designing and developing the software so that it uses multithreading through which that software can take advantage of the multiple cores, each one of the threads which runs with concurrency on a processor, that automatically increases the speed of the execution of the application by a huge amount of times. For developing such an application which is extensible as well as coextensive,

CUDA provides the parallel model program that gives a programmable model for the multiprocessor machines. GPUs developed by NVIDIA are strong in terms of computational power. GPUs consist of many cores and a large number of threads run on those cores, and therefore, These GPUs are very fast and takes less time for computation that the CPUs take for the same computation, the performance comparison of recent CPU and GPU are shown in Figure 1.1. The performance comparisons are between the CPUs, GeForce GPUs, and Tesla GPUs.

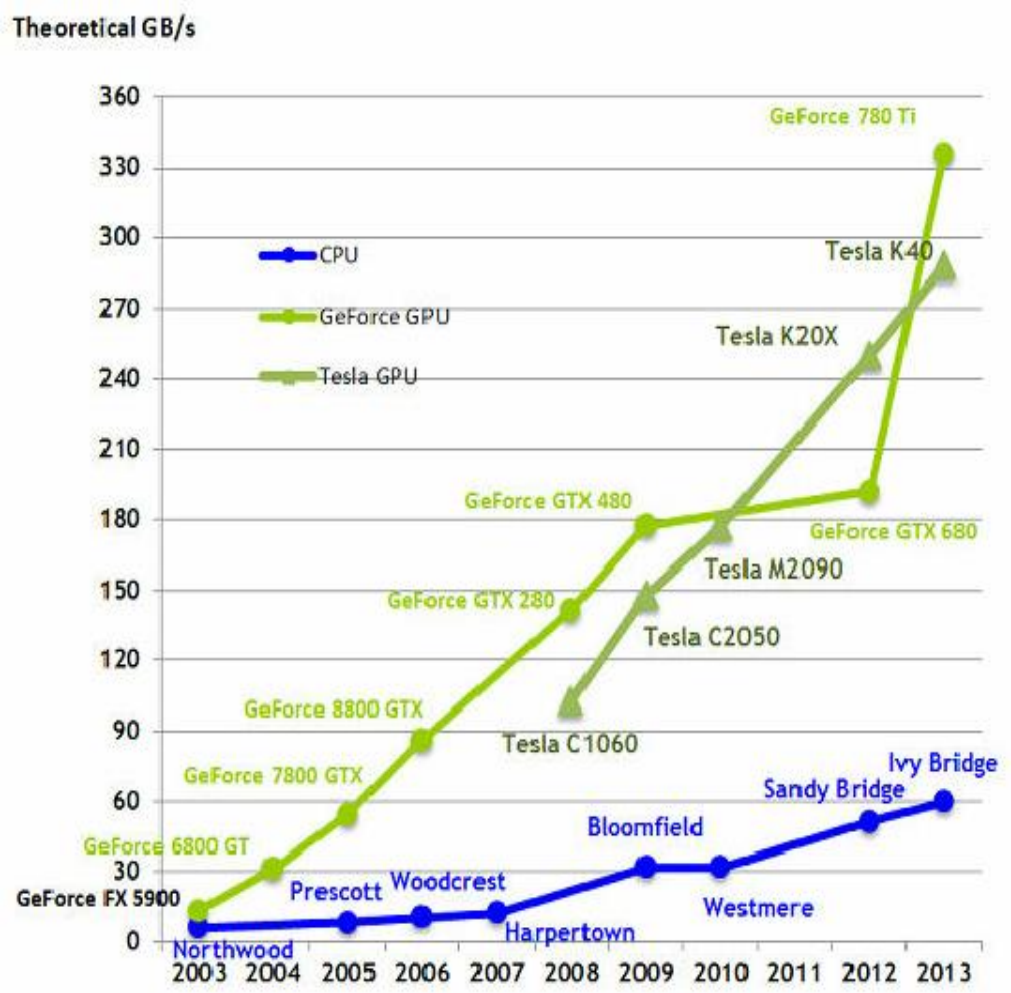

Figure 1.1: GPU vs. CPU Performance

In the beginning, these GPUs were doing the job of graphics. Nowadays GPUs are used in other applications such as computing large calculations for massive neural network but not only for the graphics purpose only. The programs designed and developed for GPU (Graphics Processing Units) run on the multi-processors using a large number of threads concurrently. At the end, these programs become extremely fast. Now in the following sections, the CUDA architecture, system model, memory model is presented.

1.2 The CUDA Model

CUDA is the acronym for Compute Unified Device Architecture. This platform is very useful when we want to utilize the parallel nature of the computation. This programming technique is used to developed parallel applications for the Graphics Processing Unit. CUDA programming is almost similar to C or C++ language but it lacks the object-oriented feature so it is more C like language. In CUDA programming language there is a function called kernel which is run on GPU and parallelizes the task. This kernel function is run the 'n' number of times on the GPU, n denotes the total number of the thread. Memory synchronization and memory management functions is also provided by CUDA. The full description of CUDA is given below.

1.2.1 Model for CUDA System

GPU previously were used for graphical software. Now in the recent times GPUs are used for computation heavy application. The use of parallel multicore and multiprocessors provides the strength to the GPU which can easily provide the speed which requires the multiple cores for the computations. In Figure 1.2, the memory and the multiprocessor layout as follows:

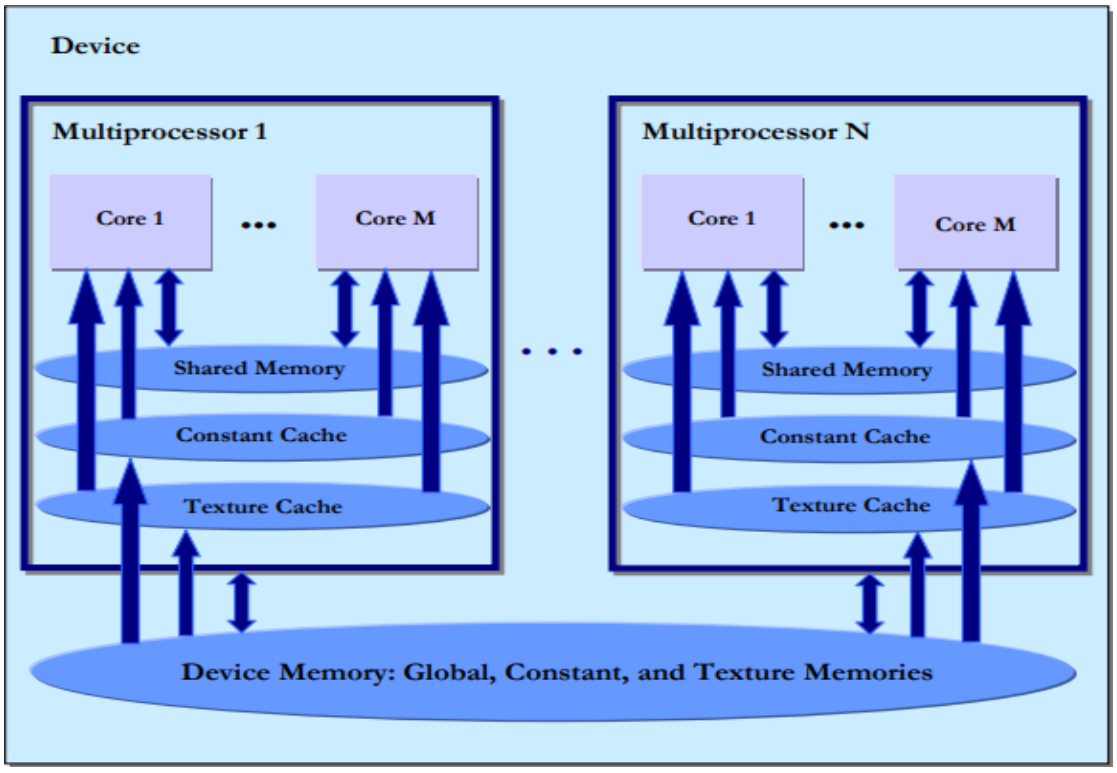

Figure 1.2: Memory Model for CUDA

All the application in which a common instruction runs multiple times, this kind of problem is well suited for the GPU. CUDA is developed and designed by NVIDIA which provides the power to harness the strength of the GPU. It is just like procedural language C. It is very easy to use and it also provides the memory management functions.

1.2.2 Heterogeneous Architecture

A mixture of serial-wise and parallel-wise implementation technique is followed in CUDA programming. Figure 1.3 illustrates type of this programming technique. The program written in C language is run on the CPU and this is called the host, the program written in CUDA language is run on the GPU, and is called Device. The grid and block are the variable which specifies the number of how many blocks are there in a grid, and how many threads are there in a block respectively. The calling syntax of the kernel function is <<<grid, block>>>.

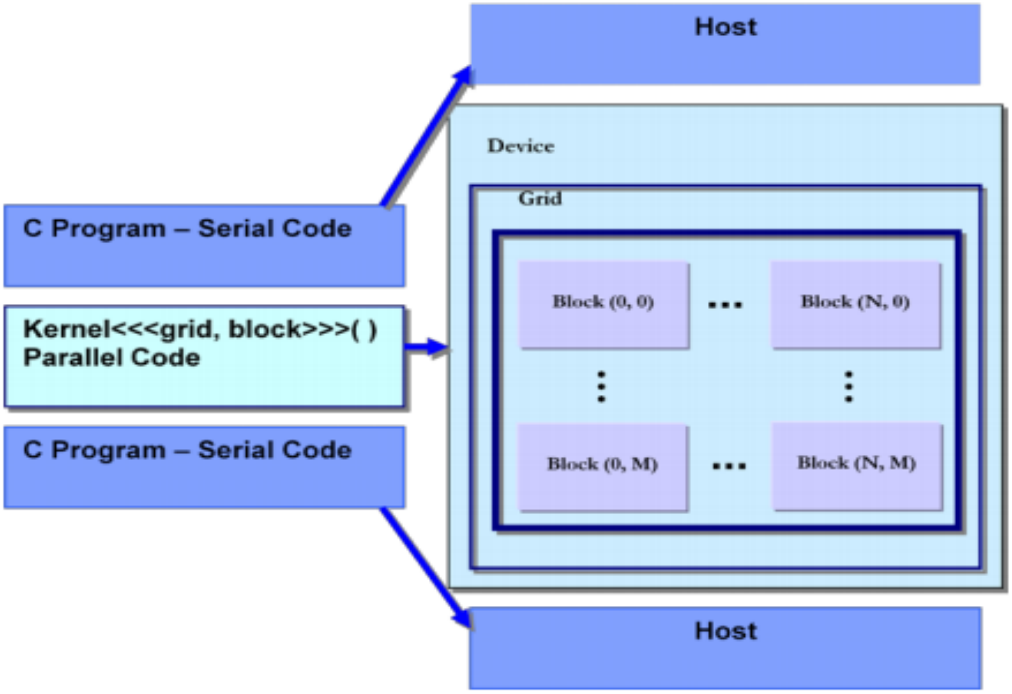

Figure 1.3: Heterogeneous Architecture of CUDA

1.2.3 Structures of Grid and Block

The grid and block structures can be one dimensional, two dimensional or three dimensional. A grid can have 1024, 1024 and 64 blocks in each dimension. The dimension of the grid can be obtained from gridDim. Similarly, blockDim gives the

dimension of the block where blockIdx and threadIdx gives the identification of the block and thread respectively. All the block inside a grid share the global memory area and all the thread inside the block have access to the shared memory area. All the threads inside a block can communicate with each other. But the thread from different block can't communicate to each other.

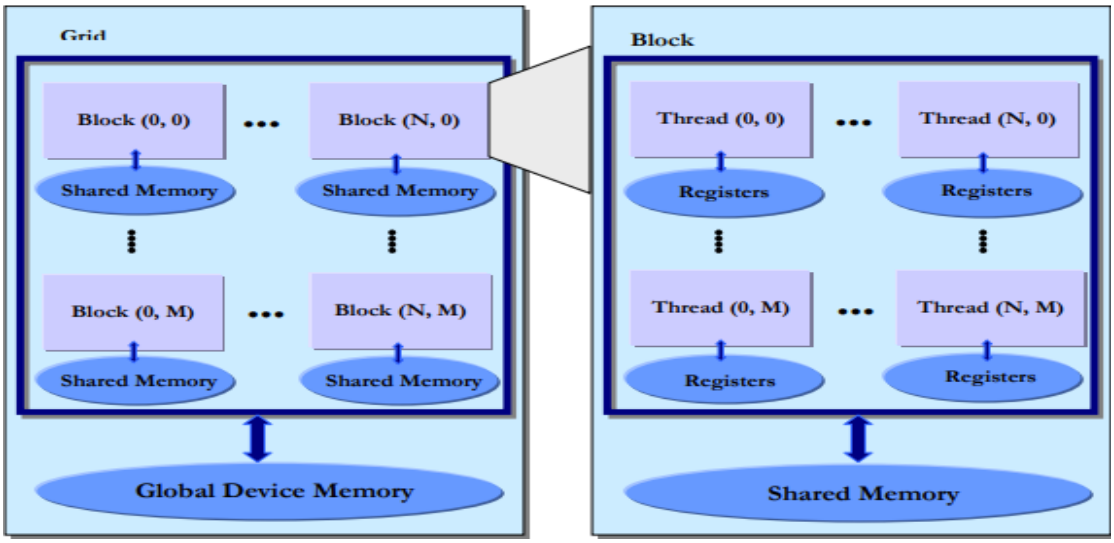

Figure 1.4: The CUDA Grid structure and Block Structure

1.2.4 Model for CUDA Memory

There are two types of memory area one is shared memory area and another one is device memory area. For collecting and distributing data among the host and device all of the multiprocessors demands a way to get accessibility to the device memory which is global in nature. The CUDA memory management is shown in the Figure 1.2. When there is no caching then the underlying memory becomes very slow compared to the memory area which allows caching. Since there is no caching in the global device memory, this memory area becomes slower. The time needed to access any register is very minimal, so memory should be as fast as compared to the access time of a register. Shared memory area takes the same amount of time as the registers, which is comparatively faster than the global memory area because in case of the global memory area data caching is not available. Shared memory area is also called as parallel data cache. When a block of threads is assigned some chunk of shared memory area that chunk becomes local to that block and synchronization among the threads is done using that chunk of the memory. Every block gets a processor on which it is executed and each such block gets their shared memory area so if any other block running on the

same processor can't access the shared memory area of other block which will be running on same processor or on another processor. That is why this memory area is local to that particular block. When a block gets deallocated from the processor all the threads inside that block also exits the processor same time. A block has its shared memory area and all the threads inside that block uses that same shared memory area for the purpose of writing and reading, when synchronization happen among the threads the also this shared memory area is used by the participating threads in the synchronization. Usage of the shared memory area should be done carefully because the size is very small generally it is of 16KB. Declaration of variables in the shared memory area and in the global memory area is done using the keywords __shared__ and __device__ respectively. To get the faster access to the data for reading, processors also equipped with the memory which is the local memory to the processor and it is read only memory. So, block has memory, processor has local memory and threads also has its local memory where the local variables of the thread is created and it as also sometimes created in the global memory area.

1.2.5 Synchronization of Threads

For the synchronization among the threads CUDA ha a predefined function syncthread() which provides a point at which all the threads waits to join and then when all the threads arrive at this point further execution happens. Since all the thread in a single block shares the same shared memory area that is why all the threads of a block can use this synchronization but threads from different blocks can't synchronize because different block have different shared memory area and any thread can't access another block shared memory area.

1.2.6 Threads for each Block

Total number of threads assigned to each block and total number of blocks assigned to each thread should be done carefully. Because we have to harness the power of GPU this configuration should be done very carefully.

1.2.7 Control Flow

- To allocate memory on host and device, the allocation is done without depending on one other.
- There is need to send the data from host to device using cudaMemcpy function.
- Kernel is executed on multiple cores so that it gives optimum performance.

- Now once the computation is done, the data is brought back from the device memory area to the host memory area.

In the below figure 1.5 two arrays are declared one on the host memory and another on the device memory. The data which are present in the host memory is sent to the device memory using the CUDA API cudaMemcpy. Then in the device memory the array is available for the calculations. After performing the computations on the data and generally these computations are massive in nature, the kernel processes these data using the capability of the GPU to compute the end result. It is highly recommended that GPU should be used carefully, means it should be fully utilized. All the computation power that the GPU possesses should be harnessed. The number of thread inside a block and the number of block inside a grid should be given in such a way that the kernel should use the hardware to its full potential.

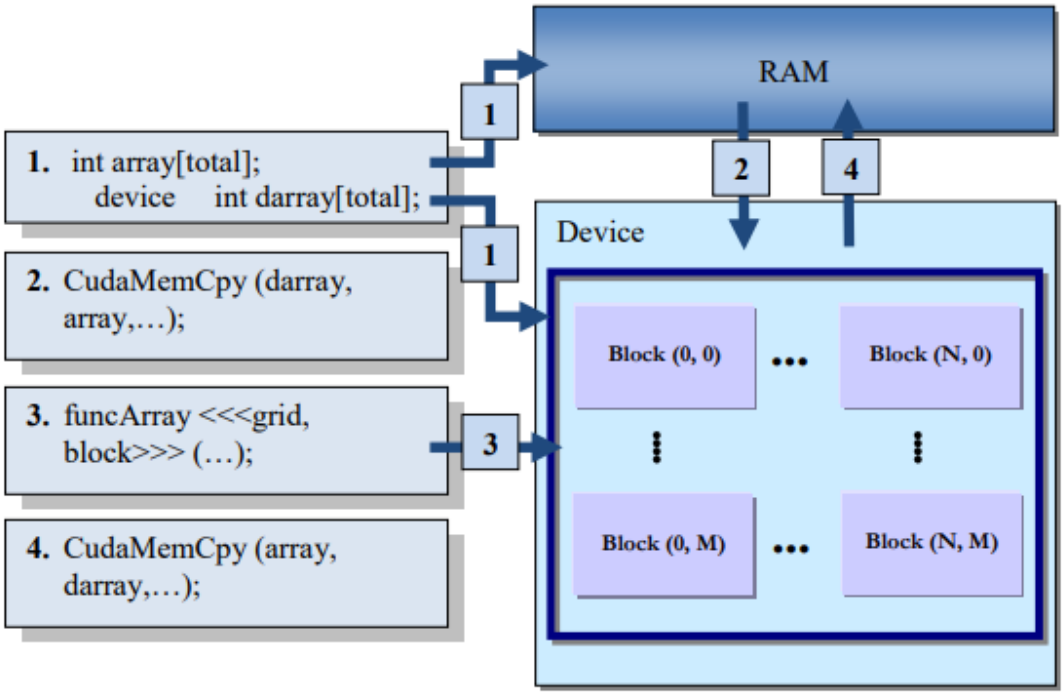

Figure 1.5: An Example of Processing Flow

1.2.8 Transferring Data Between Host and Device

The amount of time required to communicate between host memory and device memory is more compared to the amount of time required to communicate between device memories because in first case the bandwidth is very minimum between the host and device memory area and in later case the bandwidth is high. Minimum

communication should be done between host and device because it is very expensive. Any unnecessary communication will put extra overhead in transferring the data. That is why all the required arrays should be declared as global and then the work will be much easier for the GPU because the overhead is much less in case the data resides in the global memory area. For reducing the transfer overheads for large transfer of the data use batching up approach to transfer many small amounts of the data.

1.2.9 Commonly used CUDA API

Following are the qualifiers in CUDA which is used with function, these are device, global, and host.

1.2.9.1 Qualifiers for Routines:

(1) device: The routines which is declared and defined using device as qualifier is run on the device. This type of programs is called by only the device only.

(**2) global:** The routines which is declared and defined using global qualifier is run on the device but it is only called by host.

(**3) host:** Routines which is declared and defined using host qualifier is run on the host and are only called by the host. In the absence of any qualifiers, it indicates routine would execute on the host, this type of routine is declared and defined using host qualifier.

1.2.9.2 Variable Type Qualifiers:

Following are most important variety of variable qualifiers in CUDA, device, constant, and shared which is explained below:

(1) device: Variables declared using device gets memory on the device. Another type of the qualifiers can be assigned with device alternatively. Suppose a variable is declared using device qualifier then this variable gets allocated memory inside the global memory area and its duration is the total existence of the program. it is present in the global-memory, it can be fetched from entire threads that are inside the grid.

(2) constant: Constants gets memory on the device. This qualifier can be used together optionally with device qualifier. It is present inside constant memory, and have duration of the program. It can be fetched from entire threads that are within the grid.

(3) shared: By using it shared variable is declared. It is together optionally with device qualifier. It resides in shared memory of a block, and has the duration of a block. It can be retrieved from entire threads inside the block.

1.2.9.3 Built-in Variables:

These are the already present variables in CUDA are as follows:

- **Grid Dim:** Its datatype is dim3 and this has the value of Grid dimension
- **Block Idx:** Its datatype is uint3 and this has the value of the index of the thead inside grid.
- **Block Dim:** Its datatype is dim3 and this has the value of the Block dimensions.
- **Thread Idx:** Its datatype is uint3 and this has the value of the index of the thread inside block.
- **Warp Size:** Its datatype is integer and it has the value of the warpsize of the thread.

1.2.9.4 Memory Management:

- **Allocation of the Memory:**

  int* d_arr;

  cudaMalloc((void**)&d_arr, 512 * sizeof(float));

- **Deallocation of the memory:**

  cudaFree(d_arr);

1.2.9.5 Transferring Content from Host Memory to Device Memory:

- **Transferring Content from host array to device memory:**

  cudaMemcpyToSymbol(const T& symbol, void* source,size_t count)

  Illustration:

  int cpuArr[1000];

  int dArr[1000];

  cudaMemcpyToSymbol (dArr, cpuArr, sizeof(cpuArr));

- **Other Approach:**

  int cpuArr[1000];

  int p = sizeof(cpuArr);

  int* d_Arr;

  cudaMalloc((void**)&d_Arr, p);

  cudaMemcpy(d_Arr, cpuArr, p, cudaMemcpyHostToDevice);

- **Transferring Content from host memory array to constant memory:**

  Illustration:

  int constArr[1020];

  int cpuArr[1000];

  cudaMemcpyToSymbol(constArr, &cpuArr, sizeof(constArr));

1.2.9.6 Transferring Content from Device Memory to Host Memory:

- **Transferring Content from device array to host memory:**

  cudaMemcpyFromSymbol(void *dest, const T& symbol, size_t count)

  Example:

  int cpuArr[1000];

  device int d_Arr[1000];

  cudaMemcpyFromSymbol (&cpuArr, d_Arr, sizeof(d_Arr));

- **Another method Example:**

  float cpuArr[1000];

  int p = sizeof(cpuArr);

  int* d_Arr; cudaMalloc((void**)&d_Arr, p);

  cudaMemcpy(cpuArr, dArr, p, cudaMemcpyDeviceToHost);

1.2.9.7 Component for Run-time Device:

It has the usage in the device program and __ symbol is used at the start of it

The following is the list of these functions:

- **Routine related to Math Equation:**

  __sing(y), __cosg(y), sqrt(y)

- **Routine for Synchronizing Threads:**

  __syncthreads();

- **Routine for Atomicity:**

  atomicAdd() etc.

1.2.9.8 Mode for Emulating Device:

--deviceemu is the command which is used for debugging situation. It is generally used with the CUDA compiler command which is nvcc. Through this emulation of device is done not the simulation.

1.3 ANN (Artificial Neural Network)

Before going too deep into the working and characteristics of neural networks, few basic concepts need to be defined. First of them is the fundamental concept of basic artificial neural networks (ANN). Dr. Robert Hecht-Nielsen, who invented the neurocomputers defines a neural network as "a computing system which are made up of a number of simple, highly interconnected processing elements, information is processed by their dynamic state response to external inputs". But how do actually the artificial neural networks work? To answer this question, first lets to go back to the source system that we are trying to simulate, neural network inside the human brain.

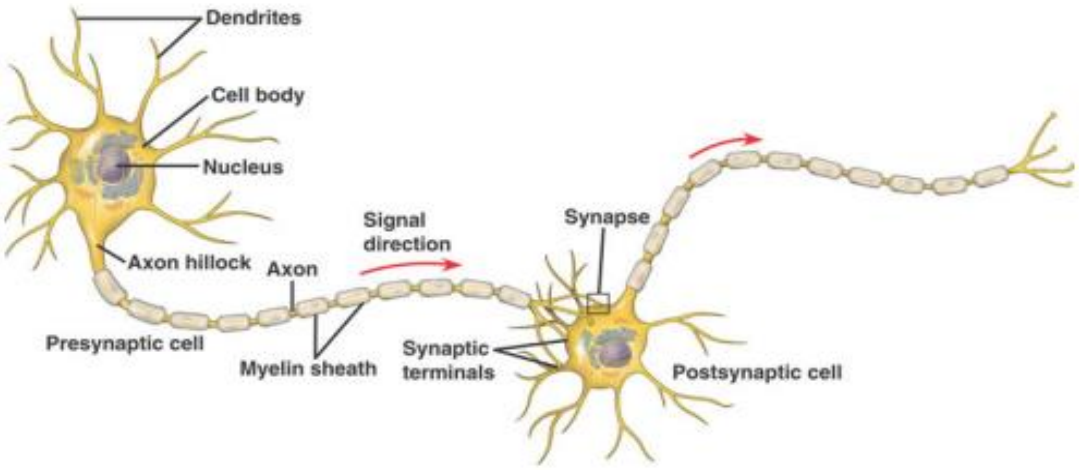

Figure 1.6: Key Components of a Biological Neuron

In the above Figure 1.6 you can see the basics of communication between neurons. Every neuron comprises of multiple processes called dendrites, which create a branch-like structure. Those act as receivers of the information from other cells, like other neurons, sensory receptors or muscle cells. From there the received information is transferred as an electrical signal through the cell body to an axon. The part of the neuron cells which is called Axon has the function to carry the information to other cells. There is a branch-like structures on the end of an axon called the axon terminals. Those are the points for transferring the information to other cells. The transfer site is known as synapse and the transfer itself is handled using chemical processes between receptors on the end of dendrites of the postsynaptic neuron and axon terminals of the presynaptic neuron. The postsynaptic neuron manages the transfer of the information after the exchange. So now basics of how the communication in real neural network works is stated, but how to simulate this concept using artificial neural networks? Artificial neural network in general can be seen as a graph, where nodes are the neurons and edges are the synapses (dendrites and axons). Every node comprises of three basic components, which are shown on Figure 1.7:

- **Weight vector:** which is the vector of synapses, assignment of weights to edges of the graph.
- **Transfer (Summation) function:** this function computes the sum of signals multiplied by concrete weights.
- **Activation function:** this function is used to map the result of the net input to the output of the neuron.

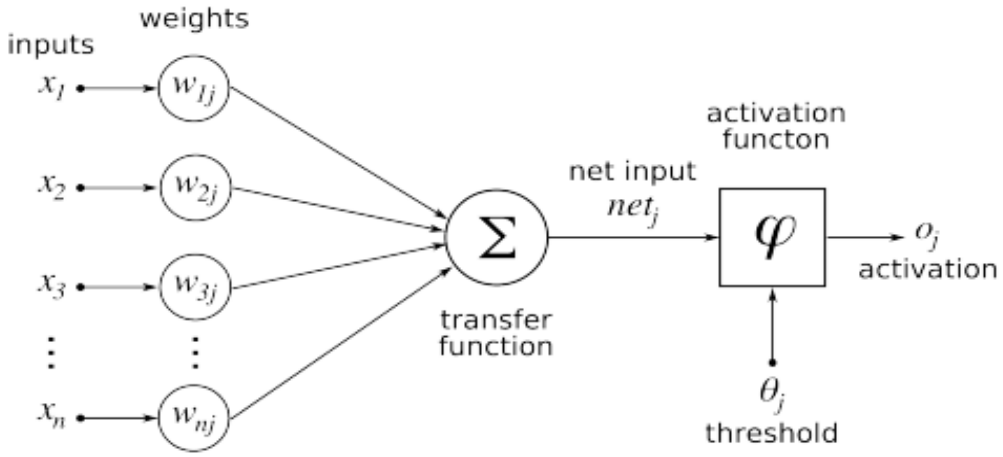

Figure 1.7: Artificial Neural Network Node

On the Figure 1.7 you can also see threshold by the activation function, which means that if the signal is too weak (weaker than the set threshold), it is not propagated further and is stopped in the neuron. In general, the network can have one or more layers. If there are more layers, the first one is usually called the input layer and the last one the output layer. Input layer is the part of the network, where the external input enters the network in means of signals. Neurons of this layer serve to process the external input and transfer it further. After the input layer there might be one or more hidden layers, whose neuron acts exactly like stated on the Figure 1.7. The neurons of the output layer act the same as the ones from the hidden layer, but their output is propagated into the final output of the whole network. If we allow the feedback edges, edges that go within the same layer or to one of the previous layers, it means that there can appear cycles. Neural networks with cycles are called recurrent and they will be in detail described in further chapters. The last thing that needs to be explained before we move to more advanced topics is how do the artificial neural networks learn. I will only explain this for one subtype of learning, which is the supervised learning. There are also other types of learning like unsupervised or reinforcement learning, but those are beyond the scope of our needs right now. You can imagine the newly created artificial neural network as a brain of a newly born child. The newly born child also doesn't know how to classify items based on their color and has to learn it first. It learns the way that it tries to assign an item to the category he thinks it fits the most, for example green. After that his mother tells him that he either assigned it correctly or it should have belonged to another color, for example red. On the next attempt of the same or similar color item it will be more likely that he will assign the item correctly. And this is almost the same way as how the supervised learning works. The commonly used supervised learning algorithm is the Backpropagation algorithm. For every input in the learning set there is also given a model output. The Backpropagation algorithm has two major phases:

- **Forward phase:** computation of outputs of all the neurons in the network, at the end error is computed based on the model and real output,
- **Backward phase:** error is propagated back through the network the weights are being changed in order for the error rate to be minimized.

The Backpropagation leads to minimization of the error function to minimum, which doesn't necessarily have to be the global one. The speed of the training can be changed in order to prevent the overlearning with every single incorrectly recognized input.

1.3.1 Variants of Activation Function

Some of the variants of Activation functions are as follows:

1.3.1.1 Linear Function:

- **Equation:** Equation for the Linear function is similar to the Straight line, i.e., $f = bg$.
- **Range:** Its range is from -infinity to +infinity
- **Uses:** Linear activation function is used at only output layer.
- **Issues:** The main purpose of the linear function is to make the data linear. If the differentiate of it makes the data non-linear then it leads to failure of the function. And the dependency on the input value will get removed.

1.3.1.2 Sigmoid Function:

The graph of the sigmoid function is same as the capital letter 'S' of the English alphabet.

- **Equation:** $F(y) = \frac{1}{1+e^{x}}$
- **Nature:** The nature of Sigmoid function is Non-linear. The value of the sigmoid function lies in between the range [-1,1], whereas the value of Y coordinates lies between [0,1].
- **Value Range:** Its range is from 0 to 1.
- **Uses:** The output of the sigmoid function will be either 0 or 1in case of binary classification.

1.3.1.3 Tanh Function:

- The advanced version of the sigmoid function, also known as Tangent Hyperbolic Function. The difference is only in terms of the y-coordinate range which lies from [-1,1]. One can easily derive the sigmoid function from the Tanh function or vice versa.
- **Equation:**
  $f(x) = \tanh(x) = 2/(1 + e^{-2x})–1$ OR
  $\tanh(x) = 2*\text{sigmoid}(2x)-1$
- **Value Range:** Its value ranges from [-1, +1].
- **Nature:** Its nature is non-linear.
- **Uses:** The main use of this function is its advantage over the other function, i.e, its range over the y-coordinate which helps to convert the data over a normal

distribution. This makes the data to be close near the mean values. This function is quietly used in the hidden layers of the neural network model to make the data insights better.

1.3.1.4 RELU:

Rectified Linear unit function is most used function nowadays the reason behind this is its feature of not creating the vanishing gradient problem. But beside these features this also suffer from the die at the start making it useless for sparse network.

- **Equation:** Its equation is A(x) = max(0, x***)***. It generates an output x if x is positive and 0 otherwise.
- **Value Range:** Its equation is A(x) = max(0, x). i.e always the output will be positive irrespective of the input.
- **Nature:** It is non-linear in nature making it quite useful in the hidden layers of the network in finding solution as well as in the back-propagation part.
- **Uses:** ReLu function is simple function that makes is light by means of computing power requirements in comparison with tanh function and sigmoid function. Only a handful of neurons are activated at a time which makes the neural network sparse which further makes it efficient and easy for computation. So, ReLu function learns much faster than sigmoid and Tanh functions.

1.3.1.5 Softmax Function:

The softmax function is similar to sigmoid but it squeezes the output in a small interval by averaging method.

- **Nature:** Its nature is non-linear.
- **Uses:** It is quite useful in cross-entropy/ multi classification problems as it squeezes the input in order to find exact class by defining the region for each class in range [0,1], which is nothing but the probability for the real class.
- **Output:** Output of this function is in fraction which define the probability of the specific class to which the input data belongs.

1.4 Formation of Thesis

Chapter 2 Literature Review: In this chapter a thorough study is provided of the CPU and GPU computations of neural network. The approaches used when large amount of data needs to be processed on GPU and the comparison of performances among CPU and GPU is provided. How CUDA paradigm is more efficient is also looked upon.

Chapter 3 Problem Statement: In this chapter a problem is identified when training the neural network with large amount of data which requires longer times on the less powerful machines.

Chapter 4 Proposed Methodology: In this chapter a methodology is proposed for a feedforward neural network for Binary Classification of two-dimensional data points. This neural network is developed using CUDA programming.

Chapter 5 Results and Discussions: In this chapter the performance of the developed neural network, in terms of time taken for training the neural network and accuracy.

Chapter 6 Conclusions and Future Scope: In this chapter Conclusions and future scope is discussed.

CHAPTER 2

LITERATURE REVIEW

This chapter covers the study of time taken for training the artificial neural network on CPU and GPU using CUDA. Literature review is as follows:

A. Garg et al. (2019) [1] have discussed about the importance of soft computing approaches in computer and Information technology where fuzzy logic, genetic algorithms and neural networks act as the fundamentals of soft computing. CPU takes longer time to run any complex neural network, this soft computing technique is widely used nowadays and more the complex structure of the network, time taken for computation will more on CPU. Further, in this paper it is shown that how GPU provides an efficient way for these types of computations. CUDA provides a parallel framework for these types of computations. In this paper, an analytical review of several soft computing approaches such as fuzzy logic, genetic algorithms, neural networks are presented. Examples include NNP and Parallel NNP technique in machine learning on K10 Tesla GPU produced by NVIDIA, Contrastive Divergence on NVIDIA GeForce GTX 460, ANN on SVM with NVIDIA Tesla K20c GPU with CUDA 7.5, Genetic Algorithms on Nvidia GeForce 660 GPU etc. CUDA framework and GPU definitely speeds up the computations of the soft computing approaches.

Kayid, Amr & Khaled, et al. (2018) [2] have discussed about deep learning and how computationally intensive task is to train model for deep neural network which demands Significant computing horsepower. For development of new algorithms and for improvement of the existing algorithms in the deep learning, the speed at which the models can be trained and tested, this decides the capability of the development and improvement of that particular algorithm. To obtain the significant performance, this can be done through hardware acceleration. In this paper, comparison of CPU and GPU in terms of performance for the training of the models is done. CPU performs better in sequential task whereas GPU performs better in parallel task.

E. Buber et al. (2018) [32] have compared the performance of CPU and GPU in deep learning computations. The author talks about how many mathematical operations performed in deep learning are best suitable for parallelization and since, GPU has a large number of cores compared to CPU. For comparing the performance between CPU and GPU, a benchmarking test is performed with Tesla K10 GPU and Intel Xenon Gold

6126 CPU. In all the tests performed, it is observed that GPU runs at least 4-5 faster than CPU.

Lin Wang et al. (2018) [13] have talked about the nearest neighbor partitioning (NNP) approach which is used for the improvement of the neural network classifier. The construction of the NNP model is much more time consuming for large data sets. The author proposed a parallel NNP method to accelerate the NNP based on the CUDA (Compute Unified Device Architecture). In this approach, blocks and threads structures of CUDA are used to calculate the potential neural networks and to perform parallel tasks, respectively. The results suggest that the proposed parallel method improves performance significantly of NNP neural network classifier.

S. Choi et al. (2017) [3] have discussed about the training of the CNN (convolution neural network) on GPGPU based on CUDA. The authors have performed the training of CNN on CPU, CUDA GPU and OpenMP. As the recent times the amount of data which needed to be processed increased many times, in such case the training time is very high. The authors proposed a parallel method to parallelize the algorithm for the training of CNN. The result in this approach is 2.5 times faster than the CPU.

A. A. Awan et al. (2017) [4] have stated that Deep Learning (DL) frameworks such as Caffe, Tensor-Flow, and Cognitive Toolkit took advantage of GPUs to accelerate the training process. This acceleration of the training was due to the massive improvements in the parallel hardware that are available today and by using the framework provided by cuDNN and cuBLAS. But in recent times, the CPU has also become faster compared to earlier due to the regular enhancements of the hardware. So, DL can be trained on CPU also and we can use the framework for enhancing the training process. The authors have proposed a complete performance evaluation of CPU- and GPU-based DNN training. Performance of DNN training for AlexNet and ResNet-50 are characterized for a variety of CPU and GPU. The authors have provided many key insights.

John Lawrence et al. (2017) [20] have compared the performance of Tensorflow which is the deep learning framework on different single and cloud node configurations. A convolution neural network was trained for the detection of the early mouse embryos. It was established that a local node with a high-performance GPU is still better than the cloud-based node in this case and gives better performance. It is much more helpful for the most people who do not have resources to design bigger system implementations.

N. V. Sunitha et al. (2017) [31] have proposed a technique to minimize the overhead which exists because the exchange of the data among host and device memories in

CUDA architecture. The input data which is to be processed are available in the host memory in the CPU-GPU based heterogenous computing Environment. Both the address spaces for the host and device is separate that is the reason device is not permitted to get access to the host. Transfer of the data is done between these two address spaces. This process takes a lot of time and it is an overhead and this type of communication overhead can be overcome by overlapping the data transfer process to the kernel execution. The authors have explored the effects of overlapping of these two processes and the effect of this overlap on the execution time on CUDA application. The result showed by using different levels of concurrency improves the CUDA software performance.

R.R. Chhajed et al. (2017) [12] have given a survey on the training time and testing time of a Back Propagation-Artificial Neural Network (BP-ANN) on different datasets and provided the comparison of CPU and GPU time for training and testing. Neural Network is fastly increasing meta learning tool used in different types of applications. The execution times of these different applications vary depending upon the size of the datasets. Neural network produces the best result when the datasets are very large. To process this large amount of data is time consuming. Graphics processing Unit (GPU) can be used for high speed computation. Thousands of cores are available in the GPU which provide a means for parallel processing. Long training times of any neural network decreases significantly on GPU during learning process.

S. Zoican et al. (2017) [9] have implemented an algorithm to find the routes which is based on neural network on Compute Unified Device architecture (CUDA) which takes advantage of the parallel processing nature of the neural networks. The author investigated and evaluated the performance of the algorithm for the efficient implementation of such algorithm. The implementation of the algorithm is not affected by the network topology so it can be used in dynamic networks

J. H. Park et al. (2016) [18] have stated that open programming models such as open computing language (OpenCL) or compute unified device architecture (CUDA) are frequently used for accelerating the computations on general purpose graphics processing units (GPGPUs). The authors have proposed a method to accelerate Artificial neural network (ANN). A forwarding computation of ANN in CUDA have been implemented and further it has been optimized using the scratchpad memory of GPUs. As a result, the proposed method is 2.32 faster compared to the conventional

CPU. An optimized GPU implementation of ANN is done which is 2 times faster than the un-optimized version of the ANN.

D. A. Shulga et al. (2016) [27] have proposed a scheduler which selects an appropriate Executing device after the prior training based on the given input data. Because in the heterogeneous computing systems the efficient use of all the computing devices is an important issue. For having a positive impact on the performance of the GPGPU-systems, the ability to choose CPU or GPU for a specific computation is very important. It helps to minimize the total processing time and to achieve the uniform system utilization.

Valentin Stoica et al. (2015) [26] have proposed an implementation model of GPU architectures which has the advantage over CPU version using CUDA. Due to their inherent architecture, the discrete time model of Cellular nonlinear networks (CNNs) for image processing are a good candidate for efficient implementation using massively parallel architectures. Several techniques of GPU resource were used and the result shows that the speed up is about 64 times the CPU implementation.

Daniel Schlegel et al. (2015) [21] have stated that the tasks which were very complex few years back and required high performance computational power can be now realized with the increasing use of the GPU. Deep Machine Learning is such task which requires a high computation on GPU. The author has reflected on the Neural Networks and tools to describe them. The benefits and issues of the GPU for running a neural network is also described.

S. Zhang et al. (2015) [7] have proposed a Back-Propagation (BP) neural method on the CUDA for the purpose of the training which is parallel in nature. The authors have implemented the Back-Propagation training in batch mode by developing the neurons at layers named input, hidden and output in matrix data structure. cuBLAS which is the basic linear subroutines were used to perform the matrix and vector computation and kernel. A cluster of GPUs are used in this approach in to gain the acceleration. Every one of the GPU is having the similar neural network and the weight vector. Distribution of the training data is done equally among all those GPU and every GPU calculates its Error in training the network which is local error and gradient is also calculated. After these calculations all the data is transferred to the first GPU for the addition to update the weights.

H. Rahmani et al. (2014) [29] have implemented Huffman coder which is a widely used Entropy Encoding Algorithm and parallelizing it on CUDA. First Huffman codeword is implemented in procedural approach then a generation of stream of byte is captured where the compressed output is represented by one bit. In the last step 8 byte of the output is taken together to produce the final output in parallel. The results obtained is up to twenty-two times faster than the Huffman coder which is implemented Serially o CPU, any precondition such as maximum length of the codeword.

J. Zimon et al. (2013) [23] have presented an implementation of Finite Element Method (FEM), based on the Nvidia CUDA architecture. The problem solved by the author is solving sparse matrix with a smaller number of cores and more number of cores. Based on the hardware level, the results are compared and the author stated that FEMM application is 70% less efficient than the proposed approach on different hardware level specification with using the CUDA architecture.

J. Pendlebury et al. (2012) [15] proposed an approach in the form a program called MineHunter. The program takes the land mines as data points over a 2-D latent space and the try to find out the 8192 mines with the help of 128 Minehunter in a parallel manner. This is done by using the 128 CUDA core of the Nvidia GTX 480 GPU. The whole process shows the performance gain over the CPU processing of the same program by 80%. As the program is implemented on CUDA in java multithreading.

S. Scanzio et al. (2010) [6] have described a block mode back-propagation learning algorithm for the implementation of neural network model using multi threating implementation for speech recognition. The authors have taken advantage of high-performance SIMD (Single Instruction, Multiple Data) architecture of GPU using CUDA API and its C like language interface. The authors of this paper have also compared the speed-up obtained by implementing the training process of the neural network with the help of the cores of the GPU by means of multi-threading. In the proposed approach, the model is trained over the large speech recognition datasets and the overall process takes 6 times less time to train over that data as compare to an already optimized implementation using the Intel MKL libraries.

CHAPTER 3

PROBLEM STATEMENT

3.1 Research Gap

As discussed in the chapter 2 that a high-performance computing environment is needed for the computation of the massive neural network with large set of data. There are already available machine learning API which takes advantage of the thousands of cores of the Graphics processing Units. Training of the neural network takes a lot of time and it requires high-performance computing systems. Since NVIDIA provides Graphics Processing Units which is a powerful GPU. These GPU can be used for the purpose of computations. NVIDIA provides CUDA programming model with which the neural network can be implemented that takes advantage of the GPU by parallelizing the computation. Since high-performance computing machines is not available for everyone. So, the NVIDIA GPU can be used to perform these time taking computations on the normal machine and we can harness the power of the Graphics Processing Units.

3.2 Objectives

The objective of this thesis is:

- To implement the Artificial Neural Network in CUDA programming environment.
- To compare the performance of the GPU and CPU for training time of the neural network.
- To test and validate the proposed Neural Network model.

CHAPTER 4

PROPOSED METHODOLOGY

In the previous chapter of the Literature Review, it is established that the Artificial Neural Network requires a high-performance machine for the complex computations. Large amount of data needs a lot of time for training the Artificial Neural Network. Graphics Processing Units have thousands of cores which is be used to reduce the training time by parallelizing the computations which needs to be performed while training the Artificial Neural Network on CUDA. In this Proposed Methodology, an Artificial Neural Network consisting of linear layer, ReLu Activation layer and Sigmoid Activation layer, which is developed using CUDA API is presented. This Artificial Neural Network is used as a classifier, to solve the Binary Classification Problem, to group together the two-dimensional data points. Further, this chapter is divided into two subsections as Experimental Setup which has the descriptions of software and hardware and Implementation Plan describes the complete implementation of the Artificial Neural Network.

4.1 Experimental Setup

- **IDE Used:** Eclipse Nsight
- **Language Used:** C++
- **Compilers Used:** nvcc 8.0, g++ 6.0
- **Platform Used:** CUDA 9.0
- **Hardware Used:** NVIDIA GeForce RTX 2080Ti

4.2 Implementation Plan

Since all the basics of CUDA programming is covered, now the implementation of a simple feedforward neural network can be done to harness the power of the Graphics processing Unit. Here, in this implementation of the feedforward neural network, the network consists of linear layer followed by ReLu activation and another linear layer followed by sigmoid activation. In figure 4.1, this neural network is illustrated. There is a need of a cost function since this neural network solves the Binary Classification Problem therefore, Binary Cross Entropy (BCE Cost) function is used. For implementing the whole neural network, these classes are needed which is as follows:

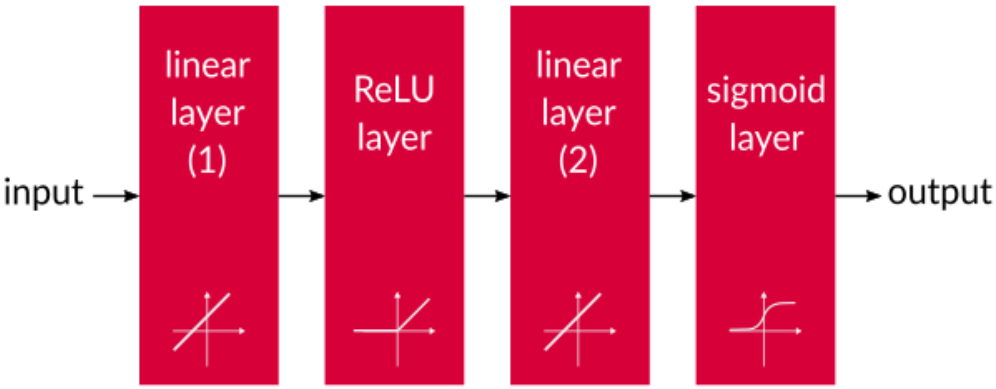

Figure 4.1: Architecture of Proposed Neural Network

- **Matrix:** As most part of the neural network requires tensor-operations, in this case a matrix is needed which is a second order tensor.
- **Layers:** A forward and backward pass is implemented for each of the Sigmoid Activation, ReLu Activation and Linear Layer.
- **Binary Cross Entropy (BCE Cost):** This class will compute the Binary Cross Entropy and its derivative.
- **Neural Network:** This class will keep every element of the neural network together and it will also manage the communication between all the elements of the neural network.

Now in the further subsections all the implementation details are provided, we will go through each class one by one.

4.2.1 Matrix Class

A data structure is needed that will store all the numbers and parameters hence, a matrix is well suited for that purpose. This class will manage memory allocation and the communication between host and device becomes easier. The matrix should be available at both host and device but the initialization of the matrix will be done on the host only. This class keeps track of the allocated memory on host and device for the X and Y dimensions which is the two-dimensional data points. To keep everything simple this is implemented on the host.

4.2.2 Layer Class

Forward and Backward propagation are performed for each class that implements any neural network. Following are the technical details of the layers that are used in this neural network:

(1) Sigmoid Layer: Sigmoid layer computes the sigmoid function for each element of the matrix in the forward pass. Equation of sigmoid function is

$$\sigma(x) = \frac{e^x}{(1 + e^x)}$$

Chain rule is used in the backward pass which is the foundation of the backward pass then the computation of error is done which is introduced by layer's input let's say its z and it is denoted by dZ. Let's assume 'J' to be the cost function then $\frac{dJ}{d\sigma(x)}$ will be the error introduced by sigmoid layer. dA is used to denote it. So, from the chain rule

$$dZ = \frac{dJ}{d\sigma(x)}\frac{d\sigma(x)}{dx} = dA\,.\sigma(x)\,.(1 - \sigma(x))$$

(2) ReLu Layer: It stands for Rectified Linear Activation Function. The equation of this function is

$$ReLU(x) = \begin{cases} x & for\ x > 0 \\ 0 & otherwise \end{cases} = maximum\ (0, x)$$

The derivative of this function is *1* where $x > 0$ and it is *0* otherwise. So, for backpropagation by using the chain rule following formula is obtained

$$dZ = \frac{dJ}{ReLU(x)}\frac{ReLU(x)}{dx} = \begin{cases} dA\ for\ x > 0 \\ 0 \quad otherwise \end{cases}$$

(3) Linear Layer: In this layer gradient descent is implemented to update the parameters w and b during backpropagation. Linear layer should implement the below equation, $Z = WA + b$ where W is the weights matrix, b is the bias vector and A is the layer's input. Now Z needs to be computed not A as was the case in the activation layer. There is need of three derivatives, one to find what error was generated by input A, this will be passed to the previous layer during backpropagation. For updating the parameters there is need to calculate error introduced by W and b using gradient descent.

- $dA = \frac{dJ}{dZ}\frac{dZ}{dA} = W^T.dZ$
- $(dW = \frac{dJ}{dZ}\frac{dZ}{dW} = \frac{1}{m}.dZ.A^T$
- $db = \frac{dJ}{dZ}\frac{dZ}{db} = \frac{1}{m}.\sum_{i=0}^{m} dZ^{(i)}$ where $dZ^{(i)}$ is the $i'th$ column of dZ and m denotes the size of any batch.

4.2.3 Binary Cross-Entropy Class

Binary cross-entropy function is used to calculate the cost and returns the gradient accordingly. The equation of the Binary Cross-Entropy is as follows

$$BCE = -\frac{1}{m}\sum_{i=0}^{m}(ylog(\hat{y}) + (1-y)\log(1-\hat{y}))$$

After calculating the derivative of the above function, gradient is also calculated

$$\nabla BCE = -(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}})$$

where y denotes the ground truth value and $\hat{y}$ denotes the predicted value.

4.2.4 Neural Network Class

This Neural Network Class manages all the elements of the network components and this class is also responsible for the communication among the layers during the backward pass and during the forward pass.

CHAPTER 5

RESULTS AND DISCUSSIONS

Every element of the proposed neural network is implemented using CUDA programming. Linear layer, Sigmoid Activation, ReLu Activation and Binary Cross-Entropy. Using these a neural network is developed for the Binary Classification Problem. Let's first discuss about the dataset and results obtained from the neural network.

- **Dataset:** A CUDA program is developed which generates two-dimensional data points and this neural network will classify the data points based on if both (X, Y) coordinates are either positive or negative, this network will classify it as group '1' otherwise the data point is in group '0'.

```
(X,Y) = (0.310841,-0.259593)   Group = 0
(X,Y) = (0.14626,-0.296648)   Group = 0
(X,Y) = (-0.296404,0.430039)   Group = 0
(X,Y) = (-0.33755,-0.207596)   Group = 1
(X,Y) = (-0.125761,-0.402265)   Group = 1
(X,Y) = (0.229232,0.238363)   Group = 1
(X,Y) = (-0.202459,0.00611627)   Group = 0
(X,Y) = (0.261935,0.111027)   Group = 1
(X,Y) = (-0.348995,0.0392732)   Group = 0
(X,Y) = (0.0992196,0.233694)   Group = 1
(X,Y) = (0.106843,-0.378288)   Group = 0
(X,Y) = (0.312359,0.220336)   Group = 1
(X,Y) = (-0.422524,-0.0149573)   Group = 1
(X,Y) = (-0.307824,-0.362857)   Group = 1
(X,Y) = (-0.333731,-0.0323973)   Group = 1
(X,Y) = (0.204238,0.477111)   Group = 1
(X,Y) = (0.208009,-0.149501)   Group = 0
(X,Y) = (-0.319537,0.411606)   Group = 0
(X,Y) = (-0.219463,-0.157087)   Group = 1
(X,Y) = (-0.295991,0.154776)   Group = 0
```

Figure 5.1: Snap Shot of Dataset

The data points will be seen in two-dimension as shown in Figure 5.2. Quadrant first and third data points are in group '1' and others are in group '0'. Now let's create the neural network, the neural network created is shown in Figure: 5.3.

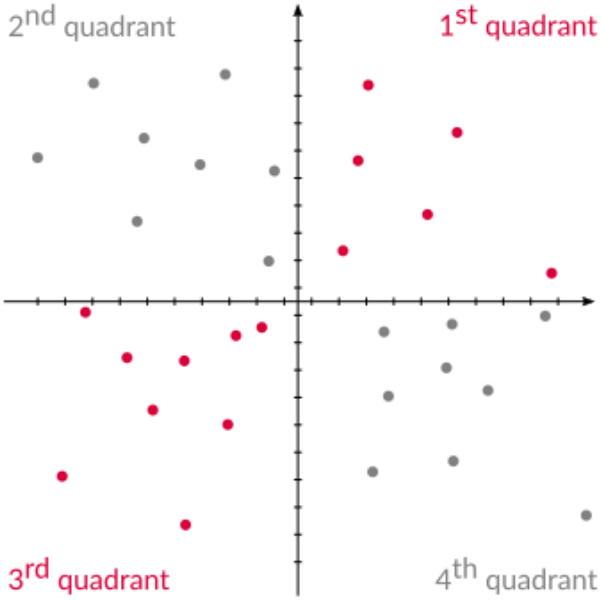

Figure 5.2: Two-Dimensional Data Points

```
NeuralNetwork nn;
nn.addLayer(new LinearLayer("linear_1", Shape(2, 30)));
nn.addLayer(new ReLUActivation("relu_1"));
nn.addLayer(new LinearLayer("linear_2", Shape(30, 1)));
nn.addLayer(new SigmoidActivation("sigmoid_output"));
```

Figure 5.3: Proposed Neural Network

- **Results:** There are 100 data points in each batch. So, the results, time taken to train the neural network and accuracy of the neural network, obtained from the 11, 21, 31, 41 and 51 batches where one batch is used for testing and remaining are used for training purpose are provided in below figures 5.4, 5.5, 5.6, 5.7 and 5.8. Here, the training time of this neural network on NVIDIA GeForce RTX 2080Ti GPU is obtained which is significantly less than the DualCore i3 4005 Intel CPU. In figures 5.9, 5.10, 5.11, 5.12 and 5.13 the result of the CPU performance is presented.

```
Epoch: 0, Cost: 0.630138
Epoch: 100, Cost: 0.63004
Epoch: 200, Cost: 0.629962
Epoch: 300, Cost: 0.62984
Epoch: 400, Cost: 0.629631
Epoch: 500, Cost: 0.629261
Epoch: 600, Cost: 0.628606
Epoch: 700, Cost: 0.627457
Epoch: 800, Cost: 0.625461
Epoch: 900, Cost: 0.622065
Epoch: 1000, Cost: 0.616491
ToTal Training Time = 3524.03 ms
Accuracy: 0.63
```

Figure 5.4: GPU Result Total No. of Batch: 11

```
Epoch: 0, Cost: 0.660119
Epoch: 100, Cost: 0.6597
Epoch: 200, Cost: 0.659224
Epoch: 300, Cost: 0.657683
Epoch: 400, Cost: 0.652808
Epoch: 500, Cost: 0.638823
Epoch: 600, Cost: 0.607317
Epoch: 700, Cost: 0.560611
Epoch: 800, Cost: 0.511397
Epoch: 900, Cost: 0.453014
Epoch: 1000, Cost: 0.3826
ToTal Training Time = 6965.98 ms
Accuracy: 0.81
```

Figure 5.5: GPU Result Total No. of Batch: 21

```
Epoch: 0, Cost: 0.670776
Epoch: 100, Cost: 0.67031
Epoch: 200, Cost: 0.6684
Epoch: 300, Cost: 0.657825
Epoch: 400, Cost: 0.616212
Epoch: 500, Cost: 0.543278
Epoch: 600, Cost: 0.453894
Epoch: 700, Cost: 0.348972
Epoch: 800, Cost: 0.281606
Epoch: 900, Cost: 0.242199
Epoch: 1000, Cost: 0.21612
ToTal Training Time = 10613.3 ms
Accuracy: 0.94
```

Figure 5.6: GPU Result Total No. of Batch: 31

```
Epoch: 0, Cost: 0.676235
Epoch: 100, Cost: 0.675616
Epoch: 200, Cost: 0.67057
Epoch: 300, Cost: 0.634891
Epoch: 400, Cost: 0.547807
Epoch: 500, Cost: 0.406467
Epoch: 600, Cost: 0.248069
Epoch: 700, Cost: 0.181314
Epoch: 800, Cost: 0.147909
Epoch: 900, Cost: 0.127539
Epoch: 1000, Cost: 0.113583
ToTal Training Time = 13921 ms
Accuracy: 1
```

Figure 5.7: GPU Result Total No. of Batch: 41

```
Epoch: 0, Cost: 0.679543
Epoch: 100, Cost: 0.678361
Epoch: 200, Cost: 0.661127
Epoch: 300, Cost: 0.56226
Epoch: 400, Cost: 0.362059
Epoch: 500, Cost: 0.207306
Epoch: 600, Cost: 0.152053
Epoch: 700, Cost: 0.124344
Epoch: 800, Cost: 0.107239
Epoch: 900, Cost: 0.0954123
Epoch: 1000, Cost: 0.0866389
ToTal Training Time = 17445.5 ms
Accuracy: 1
```

Figure 5.8: GPU Result Total No. of Batch: 51

Now the result of the neural network developed by machine learning library and which run on DualCore i3 4005 Intel CPU.

```
1469/1469 [==============================] - 0s 32us/step - loss: 0.0169 - acc: 0.9986
Epoch 996/1000
1469/1469 [==============================] - 0s 37us/step - loss: 0.0168 - acc: 0.9986
Epoch 997/1000
1469/1469 [==============================] - 0s 28us/step - loss: 0.0167 - acc: 0.9986
Epoch 998/1000
1469/1469 [==============================] - 0s 32us/step - loss: 0.0167 - acc: 0.9993
Epoch 999/1000
1469/1469 [==============================] - 0s 43us/step - loss: 0.0168 - acc: 0.9986
Epoch 1000/1000
1469/1469 [==============================] - 0s 32us/step - loss: 0.0171 - acc: 0.9986
Total time: 57.82258099572573
```

Figure 5.9: CPU Result Total Batch: 11

```
1469/1469 [==============================] - 0s 43us/step - loss: 0.0166 - acc: 0.9993
Epoch 996/1000
1469/1469 [==============================] - 0s 32us/step - loss: 0.0166 - acc: 0.9993
Epoch 997/1000
1469/1469 [==============================] - 0s 32us/step - loss: 0.0168 - acc: 0.9986
Epoch 998/1000
1469/1469 [==============================] - 0s 54us/step - loss: 0.0168 - acc: 0.9993
Epoch 999/1000
1469/1469 [==============================] - 0s 29us/step - loss: 0.0167 - acc: 0.9993
Epoch 1000/1000
1469/1469 [==============================] - 0s 32us/step - loss: 0.0166 - acc: 0.9993
Total time: 94.6027191572648
```

Figure 5.10: CPU Result Total Batch: 21

```
1469/1469 [==============================] - 0s 64us/step - loss: 0.0160 - acc: 0.9993
Epoch 996/1000
1469/1469 [==============================] - 0s 64us/step - loss: 0.0160 - acc: 0.9986
Epoch 997/1000
1469/1469 [==============================] - 0s 64us/step - loss: 0.0163 - acc: 0.9993
Epoch 998/1000
1469/1469 [==============================] - 0s 64us/step - loss: 0.0159 - acc: 0.9993
Epoch 999/1000
1469/1469 [==============================] - 0s 75us/step - loss: 0.0157 - acc: 0.9993
Epoch 1000/1000
1469/1469 [==============================] - 0s 61us/step - loss: 0.0159 - acc: 0.9986
Total time: 101.82723528396127
```

Figure 5.11: CPU Result Total Batch: 31

```
1469/1469 [==============================] - 0s 80us/step - loss: 0.0131 - acc: 0.9993
Epoch 996/1000
1469/1469 [==============================] - 0s 73us/step - loss: 0.0130 - acc: 0.9993
Epoch 997/1000
1469/1469 [==============================] - 0s 85us/step - loss: 0.0129 - acc: 0.9993
Epoch 998/1000
1469/1469 [==============================] - 0s 74us/step - loss: 0.0129 - acc: 0.9993
Epoch 999/1000
1469/1469 [==============================] - 0s 74us/step - loss: 0.0133 - acc: 0.9993
Epoch 1000/1000
1469/1469 [==============================] - 0s 85us/step - loss: 0.0130 - acc: 0.9986
Total time: 153.03863137723238
```

Figure 5.12: CPU Result Total Batch: 41

```
1469/1469 [==============================] - 0s 171us/step - loss: 0.0112 - acc: 0.9993
Epoch 996/1000
1469/1469 [==============================] - 0s 172us/step - loss: 0.0114 - acc: 0.9993
Epoch 997/1000
1469/1469 [==============================] - 0s 331us/step - loss: 0.0109 - acc: 0.9986
Epoch 998/1000
1469/1469 [==============================] - 0s 188us/step - loss: 0.0109 - acc: 0.9993
Epoch 999/1000
1469/1469 [==============================] - 0s 207us/step - loss: 0.0111 - acc: 0.9993
Epoch 1000/1000
1469/1469 [==============================] - 0s 253us/step - loss: 0.0114 - acc: 0.9980
Total time: 254.70274439390414
```

Figure 5.13: CPU Result Total No. of Batch: 51

Here a table is presented which shows the time taken to train the neural network on GPU and CPU for performance comparison in seconds and accuracy is also compared. Clearly the GPU gives better performance compared to the CPU.

Batch Size	GPU Time (sec)	GPU Accuracy %	CPU Time (sec)	CPU Accuracy %
11	3.52403	63	57.82	99.9
21	6.96598	81	94.60	99.9
31	10.6033	94	101.82	99.9
41	13.921	100	153.03	99.9
51	17.4455	100	254.70	99.9

Table 5.1: Performance Comparison of GPU and CPU

CHAPTER 6

CONCLUSIONS AND FUTURE SCOPE

6.1 Key Findings

Following are the key findings while developing this Artificial Neural Network:

- A massive neural network requires a lot of time for its computations.
- By parallelizing the computations which needs to be carried out while training of the artificial neural network, the time taken to train the network gets reduced significantly.
- CUDA programming paradigm is an alternate to those programmers who wish to perform massive computations but the CPU takes more time to process the large data, they can harness the computing power of the GPU by using CUDA API.
- Neural Network developed with the CUDA also gives then desired accuracy of the computation.

6.2 Contributions

My contributions towards this area of field is listed below:

- Developed an Artificial Neural Network for the Binary Classification Problem which classifies the two-dimensional data points.
- Time taken to train the model on the GPU is compared to the equivalent CPU.
- The performance of the neural network which is being implemented in CUDA environment, running on GPU, has significantly better performance than the equivalent neural network developed using already available machine learning API such as Tensorflow, running on CPU.

6.3 Future Scope

In future when the machines will become more powerful than they are nowadays, this approach of developing the neural network by the programmer who wish to avail the computation power of the GPU can be used. A more complex neural network can be implemented using this approach which will benefit to take advantage of the underlying GPU of the machine. Training time of the neural network will decrease significantly.

REFERENCES

[1] A. Garg, D. Gupta, P. P. Sahadev and S. Saxena, *"Comprehensive Analysis of the Uses of GPU and CUDA in Soft-Computing Techniques," 2019 6th International Conference on Signal Processing and Integrated Networks (SPIN)*, Noida, India, 2019, pp. 584-589.

[2] Kayid, Amr & Khaled, Yasmeen & Elmahdy and Mohamed, *"Performance of CPUs/GPUs for Deep Learning workloads"*, 2018.

[3] S. Choi and K. Lee, *"A CUDA-based implementation of convolutional neural network," 2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, Kuta Bali, 2017, pp. 1-4.

[4] Ammar Ahmad Awan, Hari Subramoni and Dhabaleswar K. Panda*, "An In-depth Performance Characterization of CPU- and GPU-based DNN Training on Modern Architectures",* 2017.

[5] X. Sierra-Canto, F. Madera-Ramirez and V. Uc-Cetina, *"Parallel Training of a Back-Propagation Neural Network Using CUDA," 2010 Ninth International Conference on Machine Learning and Applications*, Washington, DC, 2010, pp. 307-312.

[6] S. Scanzio, S. Cumani, R. Gemello, F. Mana and P. Laface, *"Parallel implementation of artificial neural network training," 2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, Dallas, TX, 2010, pp. 4902-4905.

[7] S. Zhang, P. Gunupudi and Q. Zhang, *"Parallel back-propagation neural network training technique using CUDA on multiple GPUs," 2015 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*, Ottawa, ON, 2015, pp. 1-3.

[8] L Ly, Daniel. *"Neural networks on gpus: Restricted boltzmann machines"*, 2009.

[9] S. Zoican, R. Zoican and D. Galatchi, *"Neural network routing implementation using CUDA technology," 2017 13th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS)*, Nis, 2017, pp. 275-278.

[10] H. Jang, A. Park and K. Jung, *"Neural Network Implementation Using CUDA and OpenMP," 2008 Digital Image Computing: Techniques and Applications*, Canberra, ACT, 2008, pp. 155-161.

[11] X. Li, G. Zhang, H. H. Huang, Z. Wang and W. Zheng, *"Performance Analysis of GPU-Based Convolutional Neural Networks," 2016 45th International Conference on Parallel Processing (ICPP)*, Philadelphia, PA, 2016, pp. 67-76.

[12] R.R. Chhajed, S.C. Dharmadhikari and S.H. Chandak, *"A Survey of GPU based Back Propogation Algorithm"*, 2017

[13] Lin Wang, Xuehui Zhu, Bo Yang, Jifeng Guo, Shuangrong Liu, Meihui Li, Jian Zhu, Ajith Abraham, *"Accelerating nearest neighbor partitioning neural network classifier based on CUDA",* Engineering Applications of Artificial Intelligence, Volume 68, 2018, Pages 53-62, ISSN 0952-1976.

[14] Danilo De Donno, Alessandra Esposito, Luciano Tarricone, and Luca Catarinucci, *"Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD",* 2010.

[15] J. Pendlebury, H. Xiong and R. Walshe, *"Artificial Neural Network Simulation on CUDA," 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, Dublin, 2012, pp. 228-233.

[16] L. Bakó, Á. Kolcsár, S. Brassai, L. Márton and L. Losonczi, *"Neuromorphic Neural Network Parallelization on CUDA Compatible GPU for EEG Signal Classification," 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation*, Valetta, 2012, pp. 359-364.

[17] E. Cengil, A. Çinar and Z. Güler, *"A GPU-based convolutional neural network approach for image classification," 2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, Malatya, 2017, pp. 1-6.

[18] J. H. Park and W. W. Ro, *"Accelerating forwarding computation of artificial neural network using CUDA," 2016 International Conference on Electronics, Information, and Communications (ICEIC),* Da Nang, 2016, pp. 1-4.

[19] Mocanu, Irina., *"An INTRODUCTION TO CUDA Programming. Journal of Information Systems & Operations Management."* 2008. 2. 495-506.

[20] John Lawrence, Jonas Malmsten, Andrey Rybka, Daniel A. Sabol, and Ken Triplin*, "Comparing TensorFlow Deep Learning Performance Using CPUs, GPUs, Local PCs and Cloud",* 2017.

[21] Daniel Schlegel, *"Deep Machine Learning on GPUs",* 2015.

[22] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau and A. Veidenbaum, *"Efficient simulation of large-scale Spiking Neural Networks using CUDA graphics processors," 2009 International Joint Conference on Neural Networks, Atlanta, GA,* 2009, pp. 2145-2152.

[23] J. Zimon and M. Zoworka, *"Implementation of selected numerical algorithms for solving sparse matrixes using CUDA technology," 2013 International Symposium on Electrodynamic and Mechatronic Systems (SELM)*, Opole-Zawiercie, 2013, pp. 77-78.

[24] Wei Cao, Lu Yao, Zongzhe Li, Yongxian Wang and Zhenghua Wang, *"Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format," 2010 International Conference on Computer Application and System Modeling (ICCASM 2010),* Taiyuan, 2010, pp. V11-161-V11-165.

[25] Vivek K. Pallipuram, Mohammad Bhuiyan, and Melissa C. Smith, *"A comparative study of GPU programming models and architectures using neural networks"*, 2011.

[26] Valentin Stoica, George & Dogaru, Radu and Cristina Stoica, Elena, *"Speeding-up image processing in reaction-diffusion cellular neural networks using CUDA-enabled GPU platforms",* 2015, 39-42. 10.1109/ECAI.2014.7090162.

[27] D. A. Shulga, A. A. Kapustin, A. A. Kozlov, A. A. Kozyrev and M. M. Rovnyagin, *"The scheduling based on machine learning for heterogeneous CPU/GPU systems," 2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW)*, St. Petersburg, 2016, pp. 345-348.

[28] Felix Weninger, Johannes Bergmann and Bjorn Schuller, *" Introducing CURRENNT: The Munich Open-Source CUDA RecurREnt Neural Network Toolkit"*, 16(17):547−551, 2015.

[29] H. Rahmani, C. Topal and C. Akinlar, *"A parallel Huffman coder on the CUDA architecture," 2014 IEEE Visual Communications and Image Processing Conference*, Valletta, 2014, pp. 311-314.

[30] L. Zhan and C. Qin, *"A graph-theory-based method for parallelizing the multiple-flow-direction algorithm on CUDA compatible graphics processing units," Proceedings 2011 IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services*, Fuzhou, 2011, pp. 137-141.

[31] N. V. Sunitha, K. Raju and N. N. Chiplunkar, *"Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead," 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, Coimbatore, 2017, pp. 211-215.

[32] E. BUBER and B. DIRI, *"Performance Analysis and CPU vs GPU Comparison for Deep Learning," 2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, Istanbul, Turkey, 2018, pp. 1-6.

[33] M. Arora, S. Nath, S. Mazumdar, S. B. Baden and D. M. Tullsen, *"Redefining the Role of the CPU in the Era of CPU-GPU Integration,"* in *IEEE Micro*, vol. 32, no. 6, pp. 4-16, Nov.-Dec. 2012.

[34] N. Kasmi, S. A. Mahmoudi, M. Zbakh and P. Manneback, *"Performance evaluation of sparse matrix-vector product (SpMV) computation on GPU architecture," 2014 Second World Conference on Complex Systems (WCCS)*, Agadir, 2014, pp. 23-27.

[35] M. Wezowicz and M. Taufer, *"On the Cost of a General GPU Framework: The Strange Case of CUDA 4.0 vs. CUDA 5.0," 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Salt Lake City, UT, 2012, pp. 1535-1536.

[36] J. Fang, A. L. Varbanescu and H. Sips, *"A Comprehensive Performance Comparison of CUDA and OpenCL," 2011 International Conference on Parallel Processing*, Taipei City, 2011, pp. 216-225.

[37] Carlos González-Gutiérrez, María Luisa Sánchez-Rodríguez, José Luis Calvo-Rolle and Francisco Javier de Cos Juez, "*Multi-GPU Development of a Neural Networks Based Reconstructor for Adaptive Optics"* 2018.

LIST OF ABBREVIATIONS

ANN	Artificial Neural Network
CNN	Convolution Neural Network
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
CPU	Central Processing Unit
API	Application Programming Interface
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
PDC	Parallel Data Cache
KB	Kilo Byte
RELU	Rectified Linear Unit
Tanh	Tangent Hyperbolic function
SVM	Support-Vector Machine
NNP	Nearest Neighbor Partitioning
OpenMP	Open Multiprocessing
DN	Deep Learning
DNN	Deep Neural Network
BP-ANN	Back Propagation-Artificial Neural Network
OpenCL	Open Computing Language
CNN	Cellular Nonlinear Networks
BP	Back Propagation
BCE	Binary Cost Entropy