Artificial Neural Network Acceleration over Network-On-Chip

Dual Degree Project

Spring 2016

Submitted in fulfilment of B.Tech and M.Tech in Electrical Engineering

by

Nishant Gurunath

Roll Number 11D070051

Under the supervision of

Prof. Sachin Patkar

Department of Electrical Engineering, IIT Bombay

June 22, 2016

Acknowledgements

I express my sincere gratitude towards my guide Prof. **Sachin Patkar** for his constant help, encouragement and inspiration throughout the project work. Without his invaluable guidance, this work would never have been possible. I am very grateful to **B. Y. Vinay Kumar** for his invaluable support as a mentor and as a colleague. He made immense contribution to this work and has been the key driving force. He truely has been a great inspiration to pursue this field in the future. I would like to express my sincere gratitude towards Prof. **Ganesh Ramakrishnan** for teaching the course on Machine Learning, which is a fundamental to this work. I would also like to thank Mandar Datar and Kulshreshth Dhiman for investing their time and supporting when it was most needed. Last, but not the least, I would like to thank the entire Electrical Engineering department for providing valuable suggestions, discussions and inspiration throughout my stay at IIT Bombay.

Nishant Gurunath
IIT Bombay
June 22, 2016

Approval Sheet

Dissertation entitled **'Artificial Neural Network Acceleration over Network-On-Chip'** by Mr. Nishant Gurunath is approved for the degree of Dual Degree (B.Tech and M.Tech) in Electrical Engineering.

Examiners

____

____

____

Supervisor

____

Chairman

____

Date:_______________

Place: ______________

Declaration

I declare that this written submission is my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified my idea/data/fact/source in my submission. I understood that any violation of the above will cause for disciplinary action by the institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has been taken when needed.

Nishant Gurunath
IIT Bombay
June 22, 2016

Abstract

FPGA based hardware acceleration of applications provides a low power solution. The introduction of parallel processing architectures on FPGA has paved way for acceleration of applications on FPGA. Logic simulation and artificial neural networks are two such applications that have a growing interest in this field. The introduction of parallel processing in logic simulation has paved way for novel architectures that provide parallel simulation acceleration. In this work, we explore the possibility of exploiting parallelism in the gate level netlist. This work presents two novel parallel processing architectures, namely, architecture with parallel threads and MIPS based parallel architecture. Architecture with parallel threads provides scope of local memory and reduction in amount of data communication but structurally complex as compared to MIPS based parallel architecture. MIPS based parallel architecture could achieve performance comparable to commercial processors. These architectures provide good insight into possible ways of parallel processing. This paved way for a generic logic simulation architecture that could employ the benefits both type of architectures. This work also presents circuit partition models that are essential to ensure the efficient division of task. The efficient division of task provides scope of exponential growth in the simulation speed. The idea could be extended to perform application-based circuit partition as the quality of partition will form the core of logic simulation acceleration.

The search for an efficient data communication protocol for the generic logic simulation architecture, motivated us to explore other applications with similar patterns of data communication. The communication pattern of testing phase of artificial neural networks (ANNs) show similarity to the communication pattern of logic simulation gate level netlist. The FPGA based hardware implementation for testing phase of artificial neural networks could benefit a lot of real-time applications such as, diagnostics of a high speed airplane or train, video analysis. This work proposes a design-time programmable generic hardware architecture for artificial neural networks. This work proposes a Network-On-Chip (NOC) based architecture that ensures elimination of clutter due to interconnects. The work suggests an efficient way of communication to utilize inherent parallelism in artificial neural networks. The pipelined implementation of the NOC is presented to enhance the performance of the architecture, making it suitable for real-time applications. The proposed PG network implementation, which is the alternate for pipelined implementation, reduces the resource utilization, making it easy to scale. A neuron model is proposed to help efficient scaling of the neural network and reduce the amount of data communication on the NOC. A computation method that allows block-by-block computation is suggested to ensure maximal utilization of available resources. A neural network accelerator tool is developed that provides semi-automated implementation of artificial neural networks on FPGA using in house domain specific language for mapping message passing application on NOC. The results and predictions show that the proposed architecture is already performing better than most existing commercial hardwares. The architecture shows promise for growth in performance with scaling.

Contents

Chapter 1

Introduction

The modern day, real time applications require low power and high performance systems. Such systems require low power acceleration of applications. The FPGA based implementation provides a great opportunity for low power acceleration. The implementation of parallel architectures on FPGA provides opportunity for application acceleration. The hardware acceleration of applications using FPGA has found interest in recent times. Logic simulation and artificial neural networks are two such applications that have a growing interest in this field.

Logic simulation is the most frequently used method in the current verification process. Many of the design aspects are validated using logic simulation. Most of the resources are spent solely on validation and verification of designs. As the technology scales down, the amount of resources spent on verification of design increases. The complexity of the modern circuits has increased, and simulation tools are struggling to meet the demands of the market. The complexity of the modern day designs diminishes the possibility of simulation of large circuits on a single Field-Programmable Gate Array(FPGA). The performance of simulation keeps falling behind the demand resulting in defective products to the market. Thus, semiconductor industries are always seeking for faster simulation solutions that could meet the complexity constraint of the FPGA. Therefore, it became essential to introduce parallel processors to enable simulation on multiple FPGA in parallel and to reduce simulation time. [1] [2]

The previous work in this area that we came across includes [2] and [1]. [1] focusses on the use of multiple threads in the Graphic Processing Units (GPU) for logic simulation acceleration. [2] is very similar to our work and presents superscalar processor architecture and stack processor architecture to accelerate the logic simulation. It focusses on custom processor based simulation acceleration. Our work is almost a follow up of [2] except we use graph partition algorithms to divide the task and an Network On Chip (NOC) to regulate data flow communication.

This work explores the possible ways of parallel processing in gate level simulations. The work proposes two parallel processing architectures, a) architecture with parallel threads, b) MIPS based parallel architecture. The proposed architectures could meet the parallelism requirements of logic simulation. The architecture with parallel threads consists of multiple threads inside each processor core. The threads are the processing units inside the processor core and function in parallel. Implementation of parallel threads also provides the opportunity to have local memory in each processor core and reduce the amount of data communication. In MIPS based parallel architecture, multiple processors each perform a part of the task. It does not contain any threads, and hence, has a simpler architecture than parallel threads architecture. These architectures enable the division of a single task itself. The task is divided into much smaller and simpler tasks and each of these tasks are assigned to a separate thread/ processor in the processor core. The division of task is performed using circuit partition, and each partition of the circuit represents single component of the circuit/ task. The quality/ symmetry of the partition determines the performance of the processor. Thus, the architecture demands for efficient circuit partitioning algorithms.

The graph partition algorithms provide highly efficient partitions.[4] Hence, this work represents the circuit/task in the form of a graph data structure and this is called a task graph. The task graph presented is a Directed Acyclic Graph(DAG) with nodes representing operations and edges representing data. We have used the METIS graph partition tool to partition the task graph. The focus is to generate the such partitions that reduce latency by minimizing the cross communication among the partitions. In parallel simulation, the quality of the partition will determine the speed of the simulation. Thus, we are keen on looking into application based graph partition algorithms that could provide high quality partitions based in specific application types.

The analysis of both the architectures, the architecture with parallel threads and MIPS based parallel architecture, provides insight into potential ways of parallel processing. This laid the foundation for a generic logic simulation architecture that could utilize the benefits of both the architectures. In the generic logic simulator, an efficient data communication mechanism is required to achieve high simulation speed. An understanding of various mechanisms involved in data communication could provide the solution to efficient data communication in logic simulations. This motivated our work to look for applications with similar data communication patterns. The communication pattern of testing phase of artificial neural networks (ANNs) show similarity to the communication pattern of logic simulation gate level netlist. An artificial neural network is an attractive tool for solving problems related to pattern recognition, prediction, function approximation, engineering and many more. The applications that do not require high speed, a software based neural network is sufficient. However, for the applications that require real-time computation, hardware seems to be a better solution. Moreover, artificial neural networks contain inherent parallelism. Hardware implementation of artificial neural networks provides a good opportunity to explore parallelism in ANNs. For most applications, training a neural network is a one time job, and hence, can be done offline. This idea could benefit a lot of real-time applications such as, diagnostics of a high speed airplane or train, video analysis. The FPGA based hardware implementation of artificial neural networks could provide a low power and high performance solution for testing phase of artificial neural networks.

This work proposes a hardware architecture for artificial neural networks. The architecture can support any type of neural network Single Layer Perceptron, Multilayer Perceptron, Feed Forward Deep Neural Networks and Neural Networks including Feedback. The hardware is design time programmable and can represent any artificial neural network independent of topology. As most applications use the feedforward neural network, our work is concentrated towards optimizing hardware for feed forward deep neural networks. The architecture uses 32 bit single precision floating point representation for both inputs and weights. This design is also adaptable to any kind of activation function for the neurons. The architecture supports easy scaling of the neural network independent of the topology of the network. The proposed design also provides possibility of a high throughput hardware neural network. This architecture also provides methods for efficient utilization of resources. The thesis is organized as follows, chapter-2 provides the introduction to ANNs, chapter-3 contains proposed hardware architecture for ANNs and chapter-4 onwards give insight into logic simulation acceleration using NOC based architecture. The ANN acceleration is described first in this work as, it provides a better understanding of the communication pattern and hence, better understanding of logic simulation acceleration.

Chapter 2

Artificial Neural Networks

2.1 Introduction

Artificial Neural Networks (ANNs) are used in the field of Machine Learning to estimate the function for a given set of inputs and outputs. ANN helps to predict the behaviour of dependent attributes as a function of independent attributes (or input). For example, relationship between the sales(dependent attribute) of a product and money spent on advertizing of that product(independent attribute). ANNs are inspired from biological neural netowrks, the central nervous system in animals. Neural network is a system of interconnected neurons that communicate with each other on receiving a stimulus to generate the desired response. The neuron connections have weights to determine which inputs influence the output most. These weights can be tuned based on the experience, making the system adaptable to inputs and capable of learning. The power of neural networks comes from the fact that they can even estimate a non-linear relationship. Consider the example of logical XNOR (y = $\overline{(x1 \oplus x2)}$) neural network.
The output(y) of an XNOR function is not a linear function of its inputs x1 and x2. In other words, if we were to group the points with y = 1 and y = 0 separately on $\{x1, x2\}$ plane, we cannot find a line that can separate them, we would need a non-linear curve. However, if we use the plane formed by $\{x1 \wedge x2, \overline{x1} \wedge \overline{x2}\}$, it's easy to find a linear separator for XNOR. And, it's trivial to find a linear separator for $x1 \wedge x2$ and for $\overline{x1} \wedge \overline{x2}$ in the $\{x1, x2\}$ plane. So, the intuition is, we can use one layer of neurons to get $\{x1 \wedge x2, \overline{x1} \wedge \overline{x2}\}$, and next layer to get $\overline{(x1 \oplus x2)}$. The figure 2.1 depicts the neural network for XNOR. The activation

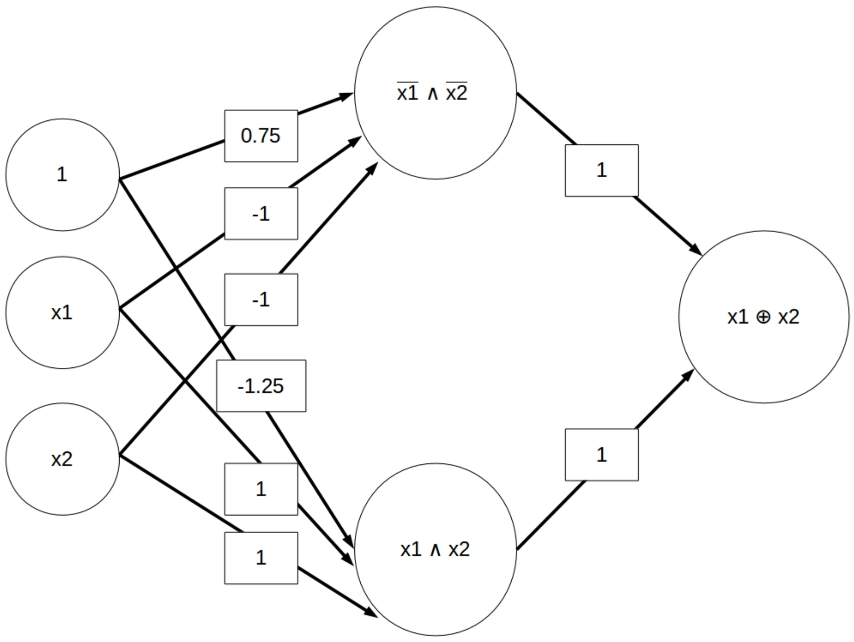

FIGURE 2.1: XNOR Artificial Neural Network

function in the XNOR example is such that, for any value greater than or equal to half, function gives the output 1, else 0. In the NN, as the input arrives, the neurons in the input layer (layer formed by neurons connected to the inputs) get activated and compute their response. This response is sent to the next layer of neurons, which then get activated and this continues till the response reaches the output layer, which is the layer formed by neurons that provide the final output. All the layers of neurons between input Layer and output layer are called the hidden Layers because their action is hidden from us, we only observe inputs and outputs. The response function of the activated neuron is called the activation function or transfer function. This function depends totally on the designer of the neural network. Some of the activation functions a) Sigmoid, b) Ramp, c) Piecewise quadratic, are shown in the figure 2.2. The activation function behaviours are shown in appendix. A typical response of a single neuron is

$$f(\sum_{i=1}^{n} data[i] * weight[i]) \tag{2.1}$$

where f is the activation function. It could be any of the functions discussed above, that depends on the designer. The summation represents the inner-product of data and corresponding weights.

Deep neural networks, as the name suggests, are neural networks with multiple hidden layers. Although, use of more than three hidden layers is not that common, deep neural networks can fit large range of non-linear data. Neural networks in which, neuron activation and data flow propagates in only one direction, are called feed-forward neural networks. Feed-forward NNs do not have any feedback loops. For a large range of applications like image processing, computer vision etc. feed-forward deep neural network is all that is required [7]. Therefore, in the following discussions, interpret neural networks as feed-forward deep neural networks only.

In machine learning, the learning system is provided some previous observations of input data and their known output values. The learning system is needed model the input-output relationship. The system is expected to learn from these past expreriences and tune its parameters (weights) to closely predict the output as a function of the inputs. This process of learning is referred as training. The process of using the trained system to predict the output of unobserved data is referred as testing. More often than not, training a system is a tedious and time consuming process. However, majority of applications do not require real time training [7]. On the other hand, testing, on most occasions, is faster than training by a few orders of magnitude. Moreover, real-time testing can be vital for some applications such as weather prediction and earthquake prediction.

2.2 Motivation for Hardware Neural Network

Artificial neural networks find use in a diverse range of applications such as pattern recognition, engineering, optimization and non-linear application mapping [11]. All the neurons in the same layer of an artificial neural network work independent of each other. Neuron activity is only dependent on data from the previous layer, weights corresponding to the data and activation funtion of the neuron. Therefore, artificial neural networks provide huge scope for parallelism. The most efficient implementation of parallel computing is possible either in hardware or in GPUs. Although, GPUs do provide a high speed solution for parallel computing, they also consume a lot of power [14]. They might be suitable for an offline application like training a neural network, however, one might not find GPUs as the best possible solution for real time applications such as testing a neural network. In a lot of applications like diagnostics of a high speed airplane or train, video analysis, high speed testing becomes as important as quality of training. In such situations, no matter how well trained your system is, if it cannot provide quick real time results and decisions, it's impracticable. Therefore, parallel computing hardware neural network seems to be the most promising high speed and low power solution for testing artificial neural networks.

Even though the idea of hardware neural networks is novel, the topology/ communication pattern of feed forward deep neural networks is not foreign to the hardware community. The communication pattern of gate level netlists in Gate Level Simulation(GLS) has similarity to the communication pattern of feed forward deep neural networks. The figure 2.3 shows the communication patterns of artificial neural networks and logic simulation. Input to the ANN is analogous to input vector in GLS. Both neural networks and GLS contain inherent parallelism. Each layer in an ANN corresponds to a level in gate level netlist. All neurons in the same layer can compute in parallel, similarly, all gates in the same level can be simulated in parallel. In artificial neural networks, response from each layer is fed to the next layer and in GLS, output from each level of gates is an input to the next level. The only difference between them is that Neural Networks involve highly complex computing and at present the size of the gate level netlists is much larger.

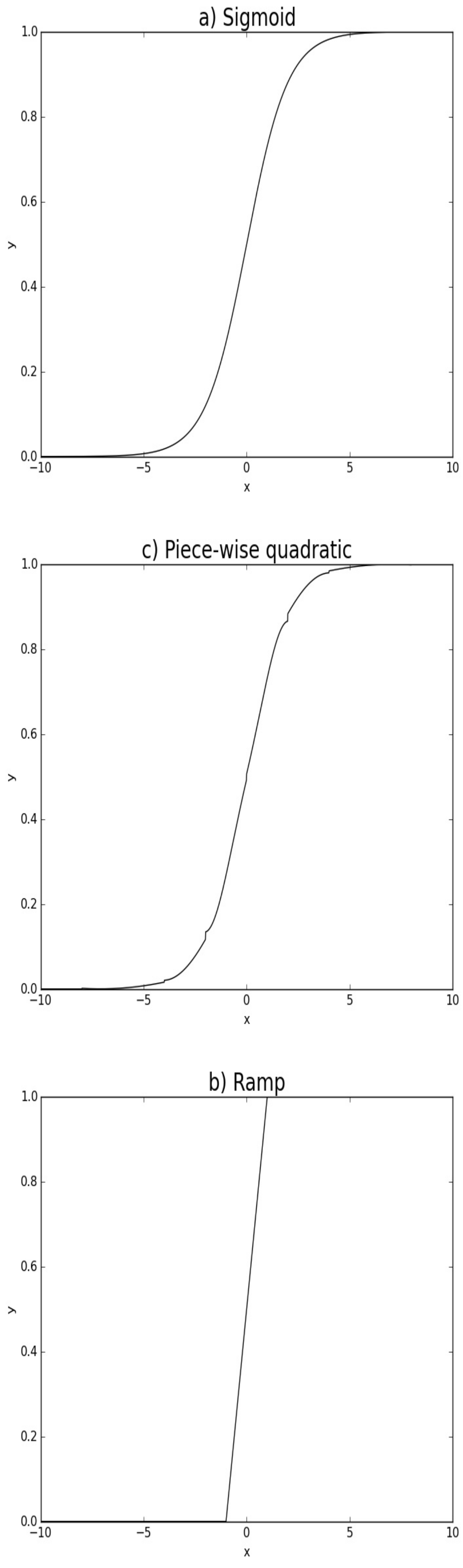

Figure 2.2: Activation Functions

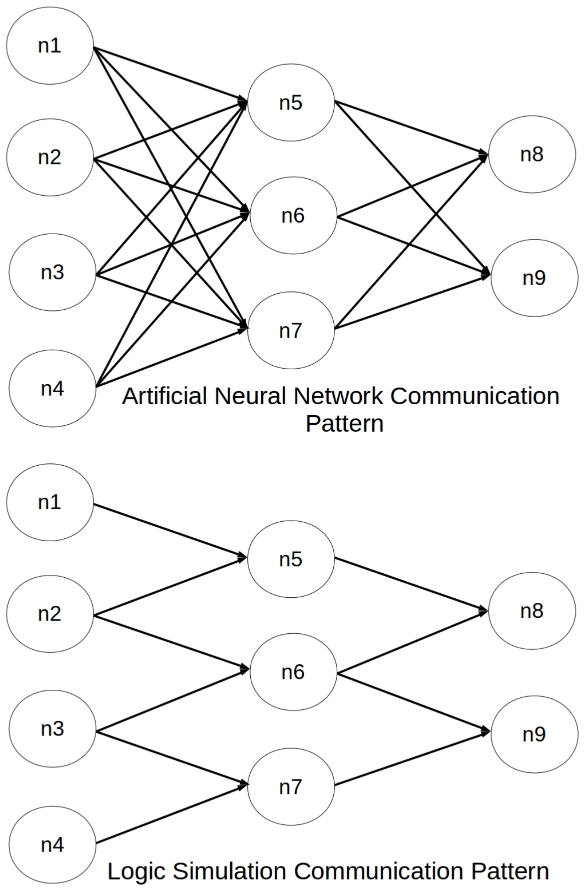
n1
n2
n3
n4
n5
n6
n7
n8
n9
Artificial Neural Network Communication Pattern
n1
n2
n3
n4
n5
n6
n7
n8
n9
Logic Simulation Communication Pattern

FIGURE 2.3: Communication Pattern Comparision between ANN and Logic Simulation

Chapter 3

Hardware Neural Networks

Hardware Neural Network, as the name suggests, is the implementation of an artificial neural network in hardware. It provides a way to represent the basic blocks - the neurons and their interconnects in hardware. It consists of an effective way to transfer data across different layers of neurons as well as an efficient way to store the weights corresponding to neuron connections. Hardware for neural network also supports complex floating point functions, inner-product of data and weights and the activation function for neuron. The hardware must be capable of harnessing the inherent parallelism that the Neural Networks offer. All this needs to be done keeping low latency, high throughput, low power and minimum resources(cost). The next section describes the complexity and challenges in designing a hardware neural network.

3.1 Complexity and Challenges

Although, hardware neural networks seem to be a very promising solution for testing phase of artificial neural networks, implementing the hardware comes with a lot of complexities and challenges. A major challenge in this regard is the number of interconnects required. Neural network is an all-to-all network. Every neuron is connected to every neuron in the next and the previous layer. This increases the interconnects drastically. Neural networks are a limiting case of logic simulation in this regard. This is what got us interested in this field in the first place. For example, consider a neural network with 10 neurons in input layer, 10 neurons in the hidden layer and 10 neurons in the output layer, the number of interconnects required for this is: $((10 * 10) + (10 * 10)) = 200$, which is huge. Number of interconnects increase quadratically with the number of neurons. This is a big hindrance in scaling neural networks in hardware.

Another challenge in the implementation of hardware neural networks is scaling. For example, in case of deep neural networks, as the network size and depth increases, no. of neurons increases and this demands for more hardware. So, the cost of manufacturing increases. Additionally, if one wants to implement hardware in a field programmable gate array(FPGA), scaling becomes even more difficult as FPGAs have limited resources and hence impose an area constraint. This constraint is absent in GPUs as they consist of thousands of threads. However, it has other challenges such as power consumption[14]. Therefore, reutilization of resources is a plausible solution. This will be discussed later in section 3.4.1 and in section 3.4.4.

As discussed earlier, one of the ways in which hardware neural networks differ drastically from gate level simulations is the high complexity of computation involved. Each neuron needs to perform the inner product of the input data (equal to the number of neurons in the previous layer) and the corresponding weights. All these data and weights are 32-bit floating point numbers, so, neuron has to perform as many floating point operations. For example, consider a neuron and 10 neurons in its previous layer, then, the number of floating point operations it needs to perform 10 multiplication operations and 9 addition operations. Further, the dot product is fed to the activation fuction of the neuron. Consider, the sigmoid activation function: $(1/(1 + \exp(-x)))$, where x is the inner product. This consists of an exponential function, so imagine the complexity of the hardware.

3.2 Previous Work

The previous works have provided some novel solutions for the hardware implementation of artificial neural networks. They have suggested a variety of architectures, some resource efficient and some high performance. These include load-store based architectures for fetching data and weights, NOC based architectures, local memory based, global memory based etc. [7] proposes a generic neural network hardware architecture. The architecture could represent multiple logic units into one physical unit and allows for dynamic configuration based on application specific requirements. [10] performs the analytical valuation and comparison of different configurable interconnect architectures. According to this work, multicast mesh topology has the best performance among the configurable interconnect architectures for hardware implementation of artificial neural networks. [11] proposes a hardware architecture for artificial neural networks that have multilayer perceptron toplogy. This work represents real numbers as fractions to reduce entire computation to integer operations. This also proposes a piece-wise quadratic approximation for sigmoid function. [9] propose a hardware architecture using tiny neural netwroks (TNNs), that are specialized in image recognition. This architecture allows for weight modification and dynamic recongurability at run-time. [14] proposes a FPGA based fixed point system that only uses on-chip memory. They use a 8-bit representation for inputs and 3-bit representation for weights.

3.3 Key Contributions

Designing of a hardware neural network comes with a lot of challenges as discussed earlier. All the previous works have tried to overcome these challenges in their own special way. However, harware neural networks like any other engineering problem becomes an optimization problem as we scale up the size of the artificial neural netwrok. This involves a trade off between using minimal resources and achieving the best performance. So, there is still a lot of scope left for improvement.

This work presents a novel way of implementing and scaling artificial neural networks on hardware. We intoduce a Network-On-Chip based architecture to eliminate the clutter due to interconnects present with scaling of neural networks. NOC also enables the architecture to exploit the inherent parallelism present in the artificial neural networks. Some of the other works have also used this method, however, we have pipelined the NOC network such that, all resources on the network are busy almost 100% of the time. This provides high throughput together with efficient utilization of available resources. For applications, where, resource minimization is more important than performance, this design also has a provision to reutilize the same resources for each layer. This enables us to represent a 10 layer deep neural network using resources equivalent to a single layer neural network. However, this requires to break the pipelining in the network.

This work also proposes a generic neuron model. The same model of neuron can be used to represent all the neurons in the neural network independent of the topology and size of the neural network. This enables our architecture to scale smoothly. We have also introduced the concept of partial computation in a neuron. That is, a neuron need not wait for all the data to arrive before it begins to compute, instead, it can perform a part of the whole computation for every data that arrives. This implies, every neuron is busy computing simpler tasks for a longer duration, instead of performing one single complex task. This requires less resources and ensures maximum usage of the resources. We further reduce the clutter due to interconnects and maximize resource utilization by representing a group of neurons as a single unit. This group of neurons share a common interface and also share the resources. The combined unit of neurons communicate using a common data packet and reduces the amount of communication required.

Moreover, this work proposes a semi-automated neural network accelerator tool. The tool generates the FPGA implementation of artificial neural network in five steps based on the user provided topology and training data for the network. The tool performs all the work from the offline training of the artificial neural network to the FPGA implementation of the final design.

3.4 Proposed Work

In this work, we propose a generic hardware architecture for testing phase of artificial neural networks. The aim is to provide a high performance hardware architecture for real-time applications. The hardware is design time programmable and can represent any topology of neural network. The architecture can support any type of neural network such as, single layer perceptron, multilayer Perceptron, feed forward deep neural netwoks and neural Networks including feedback. As most applications use feedforward neural networks, our work is concentrated towards optimizing hardware for feed forward deep neural networks. The architecture uses 32 bit single precision floating point representation for both inputs and weights. This design is also adaptable to all kinds activation functions. The architecture supports easy scaling of the neural network independent of the topology of the network. The proposed design also provides possibility of high throughput in hardware neural netowrks.

3.4.1 Architecture

We have adopted a Network-On-Chip(NOC) based architecture, the NOC is the medium for all data transfer accross the hardware neural network. The NOC is a network of routers, represented as nodes, each consisting of two data ports - input and output. The NOC has its own first-in-first-out (FIFO) buffers to control the data flow on the network. The NOC enables the communication between different layers of neural network and determines the data flow path for error free communication. This work uses a mesh topology NOC. The NOC based architecture provides a solution to the interconnect clutter in hardware neural networks. It does not require direct interconnects among the neurons, instead, communication is done using the routers with very few inteconnects. This is futher discussed in section 3.4.3

The architecture consists of following two units: 1. NOC, 2. neuron node. The figure 3.1 shows the NOC and neuron node arrangement in the architecture. The larger or the blue nodes represent NOC nodes and smaller or orange nodes represent neuron nodes. More importantly, the hardware neural network architecture

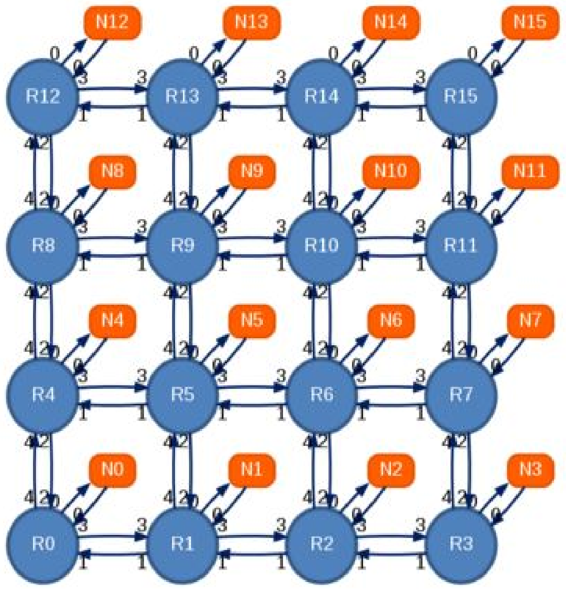

FIGURE 3.1: Network On Chip (NOC) [3]

and logic simulation architecture presented in this work both use exaclty the same arrangement for communication. Each neuron node consists of multiple neurons. The number of neurons in each node is design time programmable. Each neuron node consists of same number of neurons. The possibility of having variable number of neurons in a neuron node, provides a good opportunity to scale the HNN. The affect on the performance of the the architecture due to variable neurons in a neuron node will be discussed later in section 3.4.6. Neuron nodes, just like NOC nodes, consist of two data ports: input and output and their own data buffers for the respective data ports. Each neuron node is connected to exactly one NOC node and sends the data to other neuron nodes via NOC nodes. Every neuron node represents a set of neurons, which belong to the same layer of the artificial neural network. As the neural network is sequential from one layer to the next. There is a possibility of representing multiple sets of neurons using the same neuron node or resources. This will be discussed more later in section 3.4.6. Neuron node is generic in the sense, each neuron node consists of the same basic units. The behaviour of each neuron node depends on, the topology of the network and its position in the network, that is, which ANN layer it belongs to and which set of neurons it contains. Neuron node consists of three basic units 1. partial sum unit, 2. control unit, 3. activation unit. The overview of the design is shown in figure 3.2. Both, the partial sum and activation units are designed using Xilinx Vivado HLS. In this work, we tried to achieve best performance using minimum resources.

Each neuron node consists exacly one partial sum unit, one block ram(depends of number of neurons in the node) and one activation unit. The architecture has some special nodes, the input neuron nodes, they are not special because of their functionality, but because they are different from other nodes. They only send the input data to the next layer and do not perform any computation.

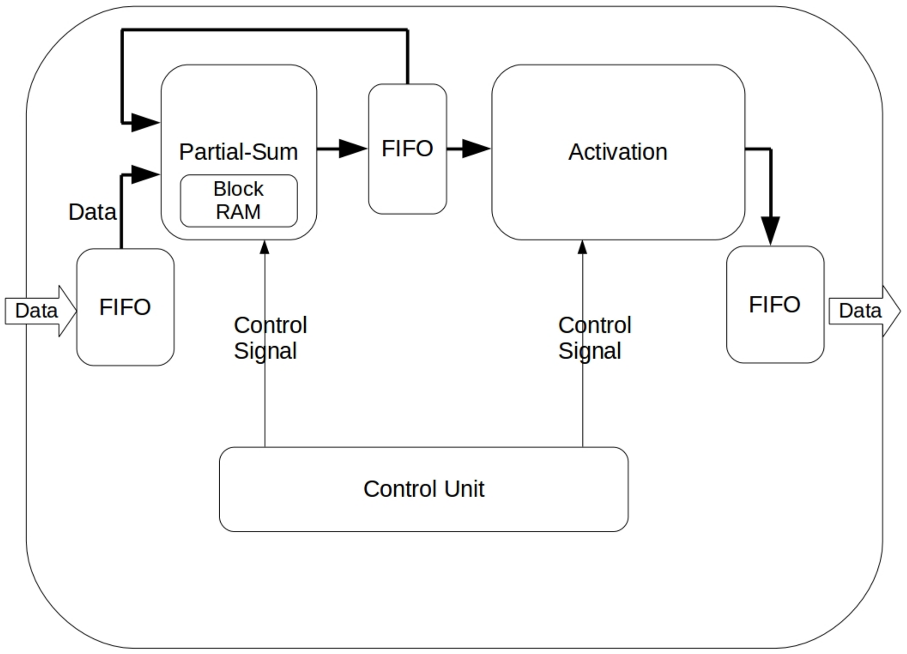

FIGURE 3.2: Neuron Node Architecture

Neuron node is implemented as a state machine. The control unit governs the state machine. It consists of following states: 1. Receive-Data, 2. Partial-Sum, 3. Activation, 4. Send-Data. It is initially in the Receive-Data state, when the data arrives it goes to the Partial-Sum state, if more data is awaited then it goes back to the Recieve-Data state, else it goes to the Activation state. And, finally it goes to the Send-Data state. Then it returns back to the Receive-Data state for next iteration of data. The figure 3.3 depicts the neuron node state machine.

The partial sum unit takes input data (data from other neuron nodes) from the neuron node(current neuron node), loads the weights stored in the block ram and computes the inner product of data and weights. This is termed partial because, not all the data from the neuron nodes in the previous layer arrive at the current neuron node simultaneously. So, for the first data that arrives, the partial sum unit computes the partial inner product and stores it in a register. Then for every subsequent data that arrives, it computes the partial inner product and adds it to

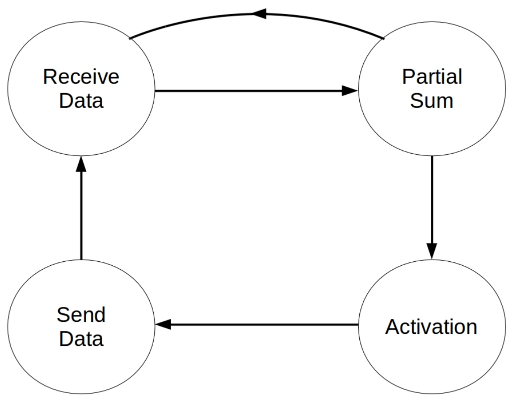

FIGURE 3.3: Neuron Node State Machine

the previously computed inner product and stores. This computation is analogous to block matrix-matrix product, where the complete product can be obtained by first computing block-by-block and then combining the results. A block is a part of the whole matrix. This sequential execution of different parts of a function, provides a great opportunity for easy scaling of the ANN, independent of the topology of the neural network. When all data has arrived and inner product is complete, the output is sent to the activation unit. The partial sum unit computes the inner product for all the neurons present in that neuron node. The activation unit takes the input from the partial sum unit, applies the activation function to the data corresponding to all the neurons in that node. The neuron node, then, sends the output of the activation unit to the neuron nodes in the next layer.

As discussed earlier, neural networks provide a huge scope of parallelism within a layer, while different layers of the NN operate sequentially by design. This work proposes a method to utilize the parallelism within the neural network. The proposed method is based on the ordering of, incoming data to a node and outgoing data from a node. The figure 3.4 depicts the method used. It shows two consecutive neural network layers, consisting 4 neuron nodes each. By design, each neuron node in the first layer sends data to each neuron node in the second layer. As there are 4 neuron nodes in the second layer, each neuron node in the first layer needs to send its response 4 times. Therefore, it requires 4 steps to complete the sending process. In step 1, each neuron node in the first layer, sends the data to one (preferrably distinct) neuron node in the second layer. In the subsequent iterations, they send to the next neuron node in the second layer in a circular manner.

For example, in iteration 1: N10 sends to N20, N11 sends to N21, N12 sends to N22 and N13 sends to N23. Then, in iteration 2: N10 sends to N21, N11 sends to N22, N12 sends to N23 and N13 sends to N20. Observe, N13 circled around from N23 to N20. With this method, we accomplish simulataneous sending from first layer and simultaneous receiving in the second layer, which provides parallelism.

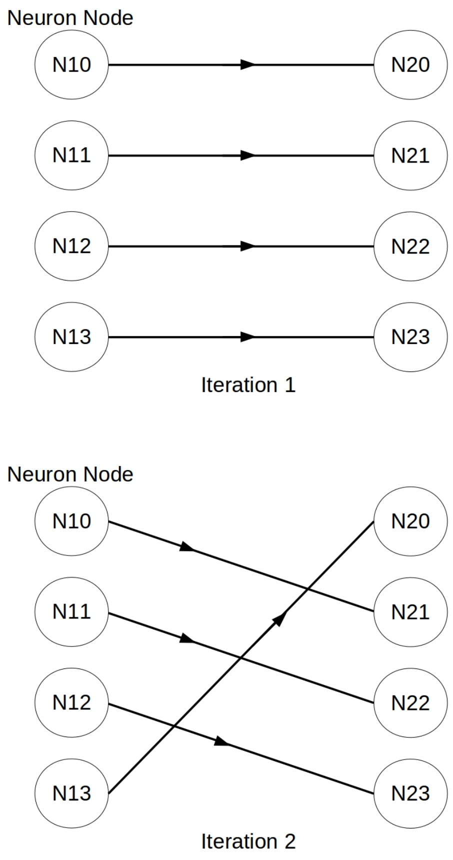

Figure 3.4: Parallelism Method

The sequential nature among the distinct layers on an ANN provides an opportunity to pipeline the neural network with different layers corresponding to different stages in the pipeline. This enables the design to achieve high throuput as required in case of real-time applications. However, the challenge in pipelining artificial neural networks is that every layer in the network has different number of neuron nodes, so, execution time for each layer is different. Therefore, it becomes difficult to obtain a effective pipeline design. For example, consider a ANN with n layers and k Neuron Nodes in each layer. Then, each node needs to recieve k data from the previous layer , and so, it needs to perform partial-sum computation k times.

Therefore, latency for each neuron node is of the form

$$l = ak + b \tag{3.1}$$

, where l is the node latency, a depends on the computation time of partial-sum unit, which in turn depends on the number of neurons in the neuron node and number of resources used for computation. And, b depends on the computation time for activation unit, which in turn depends on the activation function used, number of neurons in each neuron node and number of resources. a and b both also depend on the FPGA used for implementation. Because of the parallelism within a layer, latency of the entire layer is also of the form $ak + b$ with a difference that, b now also accomodates the additionaly delay caused by little skew there might be in the completion time of neuron nodes because of NOC. In the entire network, n-1 layers, excluding input layer, have a latency almost equal to l and NOC needs to carry the data n-1 times from one layer to the next. The total latency for the entire network is

$$L = (n - 1) * (l) + n * d \tag{3.2}$$

where L is the latency of the ANN and d is the time for data to move from one layer to the next via NOC. The idea behind pipelining is, there is no need to wait for one iteration to complete before sending in the next input data. Even if the execution times of different layers is very different, it is sufficient to delay the next input by a time equal to the execution time of the slowest neural network layer. Further analysis on this is done later in section 3.4.4.

An important factor in evaluating the performance of a hardware is the amount of resource utilization in achieving that performance. In our work, each neuron node consists of one partial-sum unit and one activation unit. The activation unit is more expensive than partial-sum unit as it requires more resources to implement the activation function. To curb the resource utilization in the neural network, one way is to share the activation unit among all neuron nodes of the same layer. However, individual nodes can only use activation unit one at a time, this will destroy the parallelism in the architecture. Also, this will cost additional delay through NOC, neuron node to activation unit and back. This will increase the latency of the neuron node. Another way to reduce resources is by using the same resources for all the layers of neural network. This is done by representing multiple neuron nodes by the same physical neuron node. We call this implementation

PG network implementation. PG network is in way NOC folded onto itself, so that, there is only one row of nodes left. This enables us to represent a 10 layer deep neural network using resources, equivalent to a single layer neural network. However, this requires to break the pipelining in the network.

The huge designs like this require some level of automation to avoid human errors in designing. In this work, we created a automated script, which is an in house domain specific language for mapping message passing application on NOC. The script takes a file(.na) that contains close to english language syntax as input and gives verilog/ bluespec design files as output. The automation is programmable for any choice of NOC, any topology of neural network and placement of neuron nodes on NOC. The input file format and syntax for a three neuron node HNN containing one input layer, one hidden layer and one output layer is shown in appendix. To put one more neuron node on NOC, it just suffices to instantiate the node with a name, eg. Output Layer, and provide the information of nodes to receive data from, operations to perform and nodes to send data to. The automation helps to design a new artificial neural network for testing phase with any toplpogy and any number of neurons instantly. This is further discussed in section 3.4.7. The development of this tool is still in progress, and, logic simulation and artificial neural network are also test cases for this script.

3.4.2 Performance Measure

The rate of multiplcation and accumulation operations and activation function computation during the testing period is termed as connections per second(CPS) [13]. A higher cps indicates better performance. CPS is a well recognized performance measure for testing hardware neural networks. The connections per second in case of our architecture is modelled in the following manner: consider a neural network with n layers (including input layer), k neuron nodes in each layer and m neurons in each neuron node. The number of connections occuring between two consecutive outputs (N) is:

$$N = C1((n-2)k^2 + k) + C2 * (n-1) * k \tag{3.3}$$

$$C1 = 2m * m \tag{3.4}$$

$$C2 = 2m \tag{3.5}$$

C1 represents the number of floating point operations in one single neuron node. For a single neuron, partial-sum unit requires m multiplications and m additions for each input data (32m bit size), that's where the first 2m comes from. Each Neuron node contains m such neurons, therefore, 2m*m. For C2, the simplest activation function takes 2 floating point operations for one neuron, and hence 2m.
Now, every neuron node takes k such data packets from the previous layer, there are k such neuron nodes in each layer and there are n-2 such layers (not including input layer and first hidden layer), hence, $(n-2)k^2$. The first hidden layer neuron nodes receive only one data packet from the previous layer, the input layer, therefore the additional k .
There are total (n-1)*k neuron nodes that execute the activation function. Therefore, the total number of connections due to all activation units is connections due to one neuron node multiplied by total number of neuron nodes, that is, C2*((n-1)*k).
Another way to look at this is, for each neuron-to-neuron connection we have 1 multiplication and 1 addition, and the number of connections between two consecutive outputs is 2*(total number neuron-to-neuron connections) + connections due to activation unit:

$$N = 2 * (km * km * (n-2) + km * m) + 2m * (n-1) * k \tag{3.6}$$

As we can see, second term on the right hand side is same as in equation 3.3 and from the first term, if we take out m^2 as factor, it is same as the equation 3.3. As discussed in section 3.4.1, pipelining in the neural netwrok can be achieved by delaying the next input by the time equal to the latency of the slowest layer of the network. The implies, the time between two consecutive outputs is equal to the latency of slowest layer of neural network. The latency of a single layer is same as the latency of one neuron node in that layer 3.4.1. In our case, all layers except the input layer have latency l as shown in equation 3.1. The input layer has a latency less than l as it does no computation, so, its not the slowest. All other layers have same latency, l, in this case. Therefore, the time between two consecutive outputs is l.

$$\frac{Connections}{second} = \frac{Connections}{cycle} * freqency = \frac{N}{l} * frequency \tag{3.7}$$

$$CPS_p = \frac{N}{l} * frequency; \tag{3.8}$$

where CPS_p means CPS for a pipelined network. For a non-pipelined network, the time between two consecutive outputs is equal to the latency of the network(L).

$$CPS_{np} = \frac{N}{L} * frequency; \tag{3.9}$$

where, CPS_{np} means CPS for a non-pipelined network. Another measure of performance for hardware neural networks is the connection primitives per second (CPPS). It is calculated as:

$$CPPS = CPS * b_in * b_w \tag{3.10}$$

where, b_in and b_w are the number of bits used for inputs and weights respectively. It allows precision to be included in the performance measure.

3.4.3 NOC Based Architecture

The introduction of NOC into the architecture provides a solution to the problem of high interconnects in hardware neural network. It does not require direct interconnects among the neurons, communication is done using the routers with very few inteconnects. Moreover, since, each neuron node contains multiple neurons, the number of interconnects required for each neuron further decreases. The number of interconnects in an artificial neural network with p layers and q neurons in each layer is of the order np^2. While in a NOC based architecture with a rxr NOC, the number of interconnets is

$$(4 * 2 + (4 * (r - 2)) * 3 + (r^2 - 4 * (r - 1)) * 4)/2 \tag{3.11}$$

Using this architecture, at 100 MHz, in a 3x3 NOC, we can represent upto 450 ANN interconnects with 5 neurons in each node and 15-15-15 configuration. While a 3x3 NOC has only 12 interconnects. Scaling to a bigger NOC increases the difference between the number of interconnects an NOC can represent and the number of interconnects in the NOC itself. However, the NOC based architecture also has a few limitations. NOC cannot provide absolute parallelism, as the communication time between any one NOC node to any other NOC node is variable. A HNN cannot be scaled as easily in NOC as in GPUs, that have thousands of threads.

3.4.4 Resource Reutilization vs Pipelining — Scaling vs Throughput(Higher CPS)

As dicussed earlier in section 3.4.1, scaling the neural network in hardware is a tough task. As the size of ANN increases, requirement of resources(area) increases and so, overall cost of design increases. A well known solution to this problem is resource re-utilization, as in a PG network discussed in section 3.4.1. Reuse the same hardware for multiple neurons and pipeline the hardware. However, this also comes with a price. It increases the latency of the overall network. This method prohibits us from pipelining the entire network. On the other hand, real-time testing of artificial neural networks requires high throughput from the hardware neural network. This is a major challenge for every hardware. And, as discussed in section 3.4.1, the time between two consecutive outputs for pipelined and non-pipelined design is l and L respectively. The difference between the CPS of pipelined and non-pipelined design can be huge for deep neural networks. In such cases, $L >> l$. Because, for a given topology CPS is inversely proportional to the time between two consecutive outputs. CPS of a pipelined network is much higher. For example, consider a neural network with 20 layers each having same number of neurons. In this case, the throughput of a pipelined architecture will be 20 times that of the throughput of a non-pipelined architecture. Hence, there is a tradeoff between scaling and performance. We have currently adopted a pipelined architecture to obtain maximum performance for a given amount of resources. Our pipelined architecture ensures almost 100% utilization of the resources present on the network as, every layer of the network is active processing some input all the time.

3.4.5 Computation Methods for Scaling

The complexity involved in the computation of partial-Sum and activation units is already discussed above in section 3.4.1. Hence, the computation requires large amount of resources, such as, Look-Up-Tables(LUTs), DSPs, Flip-Flops(FFs) and Block Rams. To get better performance we need more resources. More multipliers and more adders will do a much faster computation. However, more the amount of resources, greater the chip area and higher the design cost. A good designer always aims to get best performance using limited resources. The figure

LUT	DSP	Block Ram	FF	Latency
1052	7	1	937	37
1778	9	1	1195	36
1347	10	1	1319	35
1853	10	1	1220	32
2063	12	1	1475	28
2254	15	1	1698	27
2461	18	4	1291	26
2998	26	6	2403	25
3734	40	10	3272	24

FIGURE 3.5: Partial-Sum unit - Resources vs Performance

Activation Function	LUT	FF	DSP	Latency
Ramp	2547	2135	14	15
Piece-wise Quadratic	25207	21044	260	38
Sigmoid	33901	18340	145	34

FIGURE 3.6: Activation Functions - Resources vs Performance

3.5 shows the relation between performance and the amount of resources used for a partial-Sum unit of a neuron node with 5 neurons. The data is obtained from Xilinx Vivado HLS. As the number of resources increases the latency decreases or the performance increases. After a point, increasing the number of resources does not affect the performance as the additional resources are not utilized for computing. Since, the latencies do not change much after a point, we used the resources corresponding to the fifth entry of the table, so that we can fit our design on a single FPGA.

The activation unit could employ any of the available activation functions. Some of the activation fucntions are already discussed in section 2.1. The different activation functions differ in their computation complexity and, hence, differ in their resource requirement. Therefore, the activation function must be chosen carefully to optimize the performance with minimal resources. The figure 3.6 depicts the resource utilization and performance for three different activation functions for a neuron node with 5 neurons. The resource utilization and performance of piece-wise quadratic [11] and sigmoid functions are poor as compared to the ramp function. The ramp function seems to be the best suited for the purpose. Similarly, figure 3.7 describes the relation between performance and the amount of resources used for implementing ramp activation function. However, one cannot compromise on the correctness of the computation just for saving resources. Therefore, we checked the for the correctness of the ramp function for various topologies of

LUT	DSP	FF	Latency
2324	14	2032	20
2526	14	2135	15
2547	14	2135	15
7548	70	6352	12

FIGURE 3.7: Ramp Activation Unit - Resources vs Performance

neural network and for different number of neurons in each neuron node. The results were not only satisfactory but also perfectly correct. The reason for this observation is that, for the purpose of training in neural networks we require functions that are differentiable. Therefore, activation functions such as sigmoid are used for training. The sigmoid has a range $[0, 1]$ and threshold(point where curve crosses the x-axis) at 0. Since, all the outputs with such activation functions are between 0 and 1. Therefore, for purpose of testing in neural networks, ramp activation function with range $[0, 1]$ and threshold at 0 will provide the correct results. Therefore, we chose ramp as our activation function. The figure 3.7 shows the resource utilization and performance of ramp function activation unit for a neuron node with 5 neurons. To optimize performance with minimal resources, we chose the second entry in the table.
Scaling the neural network, requires more resources. With every extra neuron, more computation is needed. To meet the performance need with scaling, one needs to put in more resources. That is the reason, we settled with particular performance and resource combination for partial-sum and activation unit. Moreover, this architecture consists of only one partial-sum and one activation unit per neuron node. Therefore, if each neuron node consists of 5 neurons, we need one additional resource unit for 5 neurons and not every neuron. This helps to scale the neural network with less resource utilization. The relation of performance and resource requirement with increase in number of neurons per neuron node is discussed in the next section.

3.4.6 Scaling the number of Neurons

Scaling the number of neurons in each neuron node reduces the number of nodes required to represent the artificial neural network. Reduction in number of neuron nodes decreases the latency of the hardware neural network, and hence, improves the performance. This enables easy scaling of the neural network without losing

Number of Neurons	LUT	DSP	Block Ram	FF	Latency
1	358	7	1	460	8
2	1045	12	1	1108	13
3	1377	12	1	1273	18
4	1588	12	1	1374	23
5	2063	12	1	1475	28
10	4117	12	1	3077	62
10	23447	12	2	17566	97
20	8815	12	1	5696	217
20	22906	162	40	14769	84

FIGURE 3.8: Partial-Sum Unit - Latency variation with Number of Neurons in each Neuron Node

performance. However, if one keeps on putting more neurons in each neuron node without changing the amount of resources on the node, the performance will die down. This is because, now the same number of multipliers and adders need to compute for more number of neurons, this will take more time and hence, decline in performance. Therefore, one needs to add optimum amount of resources while scaling the number of neurons in each node. The figure 3.8 depicts the relation between performance, resource utilization and number of neurons in a neuron node for partial-sum unit. We try to restrict the number of block rams in each neuron node by storing the weights locally on that node because increase in one unit of block ram causes large increase in latency. The difference is shown in entries 6 and 7 of the figure 3.8. The figure 3.9 depicts the curve between latency and number of neurons for partial-sum unit. Observe, the relation is almost linear. We performed linear interpolation of this latency vs no. of neurons curve using linear model library in python. The figure 3.10 depicts the relation between performance, resource utilization and number of neurons in a neuron Node for ramp activation unit. And, the figure 3.11 depicts the curve between latency and number of neurons for activation unit. Again, the relation is linear.

3.4.7 Neural Network Accelerator Tool

The hardware Neural Network architecture is converted to a semi-automated neural network accelerator tool to implement artificial neural networks on FPGA. The accelerator tool implements variety of neural networks based on parameters such as, number of layers, number of neuron nodes in each layer, number of neurons in

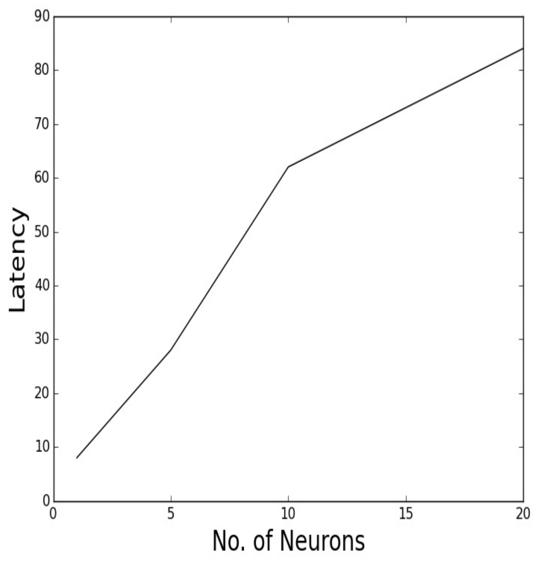

FIGURE 3.9: Partial Sum: Latency vs Number of Neurons in Neuron Node

Number of Neurons	LUT	DSP	FF	Latency
1	1484	14	1247	11
2	1949	14	1478	12
3	2223	14	1741	13
4	2390	14	1938	14
5	2526	14	2135	15
10	3375	14	2480	20
20	5429	14	2170	30

FIGURE 3.10: Ramp Activation Unit - Latency variation with Number of Neurons in each Neuron Node

each neuron node and training data for the artificial neural network. The figure 3.12 depicts the flow chart of the accelerator tool. The training data is given to Scilab which provides the weights for the neural network. These weights are extracted in the design file of partial-sum in Vidado HLS. Vivado HLS is used to generate the hardware design files (verilog files) for the two units - partial-sum unit and activation unit. The automation script as shown in appendix generates the rest of the verilog files for the given topology of the ANN. All the design files are then given to Vivado, which synthesizes the the hardware neural network and implements it on FPGA. This tool, however, can function only within the resource constraints of the given FPGA

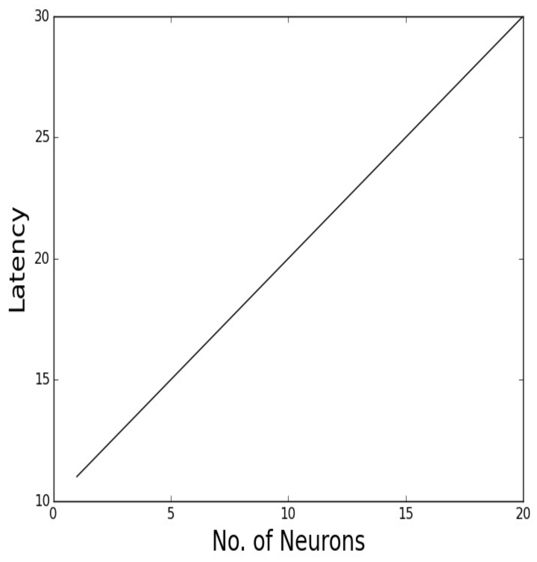

FIGURE 3.11: Activation: Latency vs Number of Neurons in Neuron Node

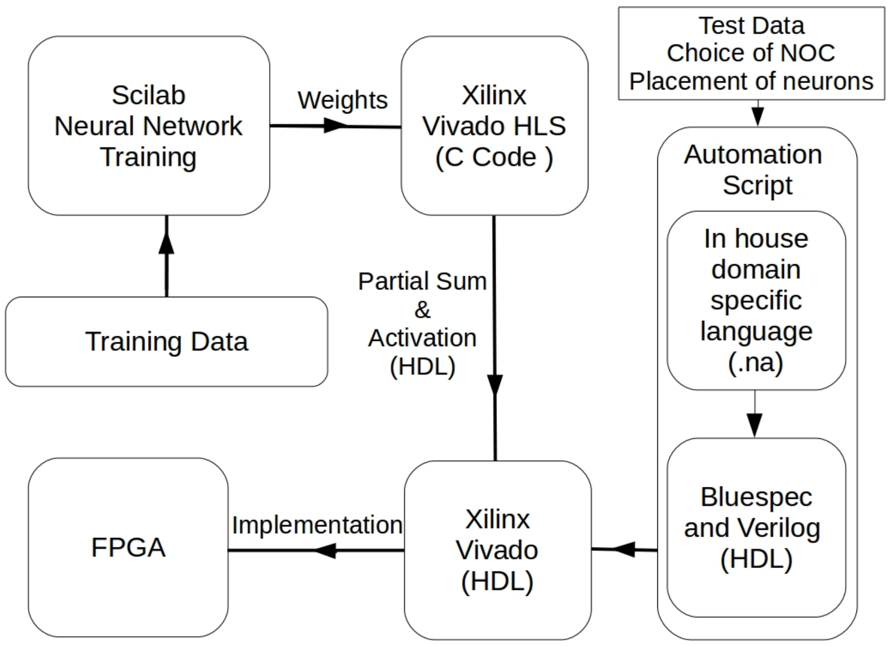
Scilab Neural Network Training
Weights
Xilinx Vivado HLS (C Code)
Test Data
Choice of NOC
Placement of neurons
Automation Script
In house domain specific language (.na)
Partial Sum & Activation (HDL)
Training Data
Bluespec and Verilog (HDL)
FPGA
Implementation
Xilinx Vivado (HDL)

FIGURE 3.12: Automation Tool Flow Chart

3.4.8 Results

The results within the resourse constraints were promising and actually much better than some of the existing commercial hardware neural networks. The results also followed the theoritical predictions made earlier. The architecure was implemented on ZedBoard (Zync xc7z020clg484-1) at 100Mhz clock frequency. Because

CPS Observed (in millions)	402
CPS Theoretical (in millions)	429
Slice LUTs	44102/53200 (82.9%)
DSPs	95/220 (43.18%)
Block Rams	2.5/140 (1.79%)
Slice Registers	48351/106400 (45.44%)

FIGURE 3.13: Results for 5-10-10-5 configuration

Topology	Observed (in millions)	Theoretical (in millions)
3-3-3	80	100
3-18-3	124	130
3-12-9-6-3	323	351
5-10-10-5	402	429

FIGURE 3.14: Simulation Results

of the resource constraint on zedboard, the pipelined architecture was tested for 5-10-10-5 configuration neural network. Each neuron node consists of 5 neurons each. Hence, the configuration for the neuron nodes is 1-2-2-1. The problem used for validation of the network was boolean right rotate. The results of the implementation are shown in the figure 3.13. Theoretical CPS is calculated at 100MHz on Zedboard as follows:

$$CPS_Th = \frac{(connections/output)}{latency} * frequency = (450/105) * 100M = 429M \tag{3.12}$$

There are total 5 neuron nodes (excluding input neuron) and each neuron node consists of only 1 MAC unit each for partial sum unit and activation unit. So, the DSP utilization is:

$$DSP(\%) = (CPS/5 * 1000000) * 100 = (4.29/5) * 100 = 85.8 \tag{3.13}$$

If we had taken a 1-2-2-2 configuration, which we could not due to resource constraints, we would have achieved theoritical CPS = 500M and 100% DSP utilization.

We also performed simulations on Xilinx Vivado kintex-7 for four different configurations namely, 3-3-3 XOR, 3-18-3 XOR, 3-12-9-6-3 XOR and 5-10-10-5 boolean right rotate. The simulation CPS closely match theoretical predictions. The figure 3.14 shows the simulation results at 100MHz. The theoretical results only differ from the observed results due to the bias caused by NOC. This proves, that the theoretical model is good enough to predict variations in performance with scaling. The figure 3.15 depicts the variation in performance of the pipelined NOC architecture with a) number of neurons per node (m) b) number of neuron nodes in a layer c) no. of layers, with all other factors constant. This work also includes simulation of a 10-10-10-10 configuration PG network implementation on kintex-7. This achieves a reduction in resource utilization by a factor equal to the number of layers, 4 in this case. The latency observed was equal to 352 cycles. This too matches our theoretical predictions. The resource table for the PG network implementation is shown in figure 3.16. For the same 10-10-10-10 configuration, a pipelined NOC implementation would take 3 times the resources used for PG network implementation.

The work in [14] tests its result on a 784-1022-1022-1022-10 configuration neural network on kittex-7 FPGA at 800MHz. They are able to test this on FPGA only because they use 3-bit precision for weights and 8-bit precision for inputs. The architecture achieves a performance of 70000 outputs/second. According to our theoretical model, we will achieve, for the same configuration, with 10 neurons per node, at 32 bit precision, 107498 outputs/second, which is a huge improvement considering the 32 bit precision we have. According to [14], a GPU can perform 250000 outputs/second, which is only 2.33 times faster, considering the amount of power GPUs consume. The figure 3.17 shows the CPPS(in GCPS*b^2 unit) comparison of some commercial hardware for feed-forward neural networks in their testing phase and our NOC architecture. The last entry of the figure 3.17 is obtained using our theoretical model for CPS. We are already doing better than most of the commercial harwares. We also designed hardware neural networks by directly using Vivado HLS. This design does not include an NOC. The figure 3.18 and figure 3.19 shows the comparison of pure Vivado HLS design and our NOC based architecture. Since, the configuration used for comparison is same, lesser the output to output delay better the CPS.

Currently, our only hindrance is resources required for scaling. The commercial architectures, especially [14], are using resources efficiently. That should be our next target in the future. Hence, we are futher exploring PG networks.

3.4.9 Conclusion

The hardware architecture proposed in this work is a generic artificial neural network architecture. The NOC based architecture ensures elimination of clutter due to interconnects. The efficient way of communication between two layers of the hardware neural network enables effective utilization inherent parallelism in artificial neural networks. The pipelined implementation of the NOC enhances the performance of the architecture, making it suitable for real-time applications. The PG network implementation, which is the alternate for pipilined implementation, reduces the resource utilization, making it easy to scale. The generic neuron model and the concept of the neuron node containing multiple neurons help in efficient scaling of the neural network and reduce the amount of data communication on the NOC. The concept of partial computing ensures maximal utilization of available resources. The neural network accelerator tool provides semi-automated implementation of artificial neural networks on FPGA. The results and predictions show that the proposed architecture is already performing better than most existing commercial hardwares. The hardware implementation of a 5-10-10-5 configuration ANN achieves 402 million connections per second. The architecture shows huge potential for growth in performance with scaling, however, efficient resource utilization is a major challenge.

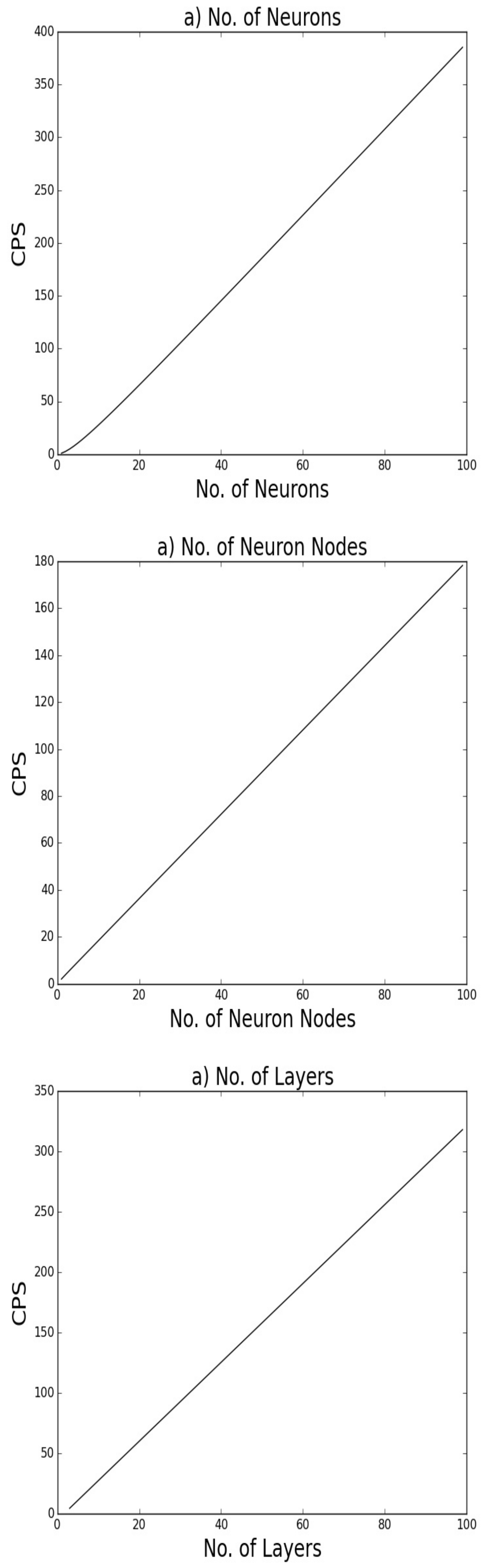

FIGURE 3.15: Connections per second vs a) No of Neurons, b) No. of Neuron Nodes, c) No. of Layers

CPS Observed (in millions)	182
CPS Theoretical (in millions)	186
Slice LUTs	30102/53200 (56.58%)
DSPs	48/220 (21.82%)
Block Rams	2/140 (1.43%)
Slice Registers	23961/106400 (22.52%)

Figure 3.16: Results for 10-10-10-10 PG network configuration

Name	Precision	Synapses (Weights)	CPS	CPPS
Intel ETANN	6bx6b	10280	2GCPS	72
Philips Lneuro-1	1b-16b	64	26MCPS	0.416
Philips Lneuro-2.3	16b-32b	-	720MCPS	368.64
Neuricam Nc 3001 Totem	32b-32b	32k	1GCPS	1024
Our Work	32b-32b	32k	860MCPS	880.64

Figure 3.17: Performance comparison of various hardware neural networks

	Vivado HLS	NOC architecture
Configuration	5-10-10-5	5-10-10-5
No. of Neurons in each Node	5	5
Output to Output delay	174	112
Block Rams	16	2.5
DSPs	80	95
LUTs	23985	44102

Figure 3.18: Performance comparison of Vivado HLS design and NOC architecture for 5-10-10-5 configuration

	Vivado HLS	NOC architecture
Configuration	5-10-10-10-10-10-10-10-10-10	5-10-10-10-10-10-10-10-10-10
No. of Neurons in each Node	5	5
Output to Output delay	544	112

Figure 3.19: Predicted performance comparison of Vivado HLS design and NOC architecture for 5-10-10-10-10-10-10-10-10-10 configuration

Chapter 4

Circuit Partition

4.1 Introduction

The perpetual need for faster simulation speed and memory constraint on single FPGA demands for the division of circuit into separate partitions. Each of these circuit partitions are assigned to separate processors that could function in parallel. This method enables us to divide the task into multiple smaller and simpler tasks. However, each partition is not entirely independent of the other partitions. There is always some data sharing and communication involved among the partitions. This inter-partition communication is called the communication volume of the network of processors. The increase in the communication volume could result in increased latency for processing. Hence, there is a need for circuit partition method that could minimize the communication volume.
In this project, we have adopted the method of graph data structure based representation of circuits and utilize the graph partition algorithms and tools that provide a minimum communication volume partition. We represented the circuit as a task graph. A task graph is a Directed Acyclic Graph (DAG). [2]

4.2 DAG representation of Logic Circuit

In the directed acyclic graph representation of logic circuits, each node denotes a task/ operation and the edges denote the data in and out of the nodes. Edges connected to only one node are either primary inputs or primary output. The

Figure 4.1 depicts a logical circuit with multiple primary inputs and single primary output and the method to convert it into a DAG. [2]

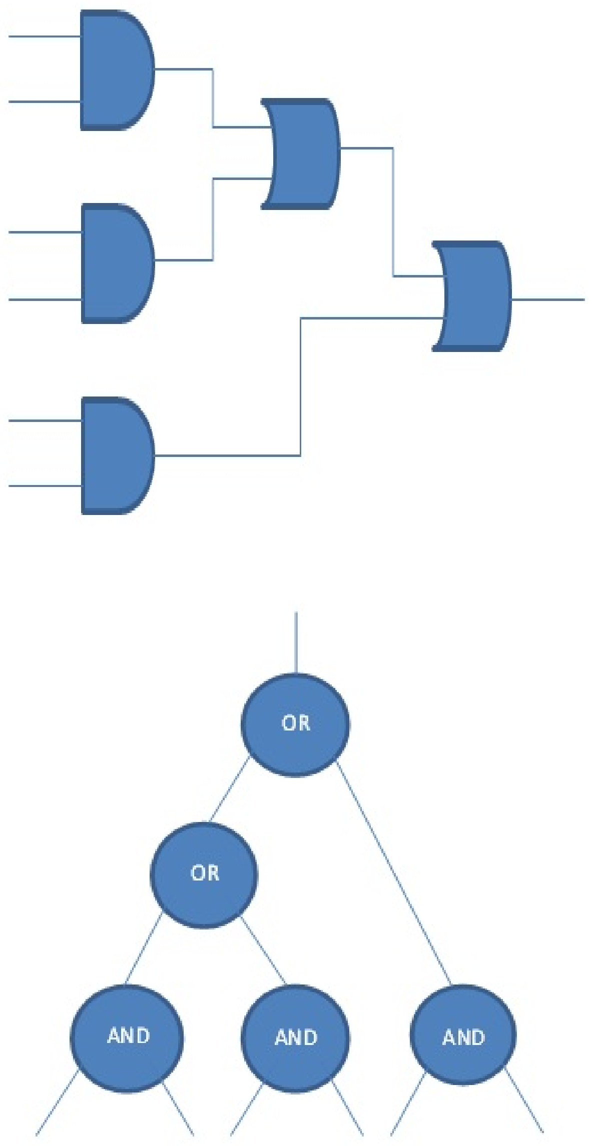

FIGURE 4.1: Logic Circuit Example

4.3 Levelization

The notion of parallel hardware computation is only possible if we could determine the order of operations in which they occur. We need to decide which operations can be performed in parallel and which operations require sequential processing.

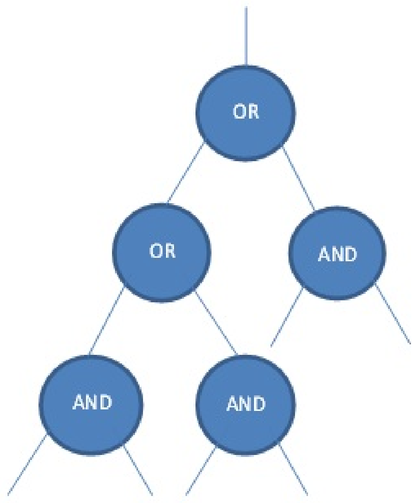

FIGURE 4.2: Levelized Graph

This approach for determining the order of operations is called Levelization of DAG. Each node in the graph is assigned a level based on its distance from the primary output, with primary output node serving as the reference 0 level. The next set of nodes that are directly connected to the primary output node are assigned level 1 and so on. Therefore, the nodes with the same level can be computed in parallel whereas, nodes in different levels are computed sequentially. The levelized graph is depicted in Figure 4.2.

4.4 Graph Partition

We used Metis – a multilevel k-partition graph partition tool that minimizes the communication volume. The following example is an application that confirmed the efficiency of the tool. We used the tool to partition the Matrix-Vector product expressions. We divided the expressions of 7x7 matrix and 7x1 Vector production expressions into symmetric blocks and represented these expressions as a DAG as mentioned in the previous sections. [4]

Expressions for partition-0

$$y00 = ((a00.x00 + a01.x10) + a03.x30) \tag{4.1}$$

$$y10 = (a10.x00 + a13.x30) \tag{4.2}$$

$$y30 = (a30.x00 + a31.x10) \tag{4.3}$$

where, aij denote matrix entries, $xij = xi$ and j is just to denote that it is used in jth partition and yij denote the intermediate values for partition j.

$$Y0 = ((y00{+}y04){+}y06); \quad \text{where, Yi denotes the ith element in the product vector.} \tag{4.4}$$

The expressions for other $ith - partitions$ are obtained in a similar pattern:

$$y(i)(i) = (aii{*}xii{+}a(i)((i{+}1)\%7){*}x((i{+}1)\%7)(i)){+}a(i)((i{+}3)\%7){*}x((i{+}3)\%7)(i)) \tag{4.5}$$

$$y((i+1)\%7)(i) = (a((i+1)\%7)(i) * xii + a((i+1)\%7)((i+3)\%7) * x((i+3)\%7))(i) \tag{4.6}$$

$$y((i+3)\%7)(i) = (a((i+3)\%7)(i) * xii + a((i+3)\%7)((i+1)\%7) * x((i+1)\%7)(i)) \tag{4.7}$$

The partitioned graph obtained was as shown in Figure 4.3. Each node colour represents a single partition.

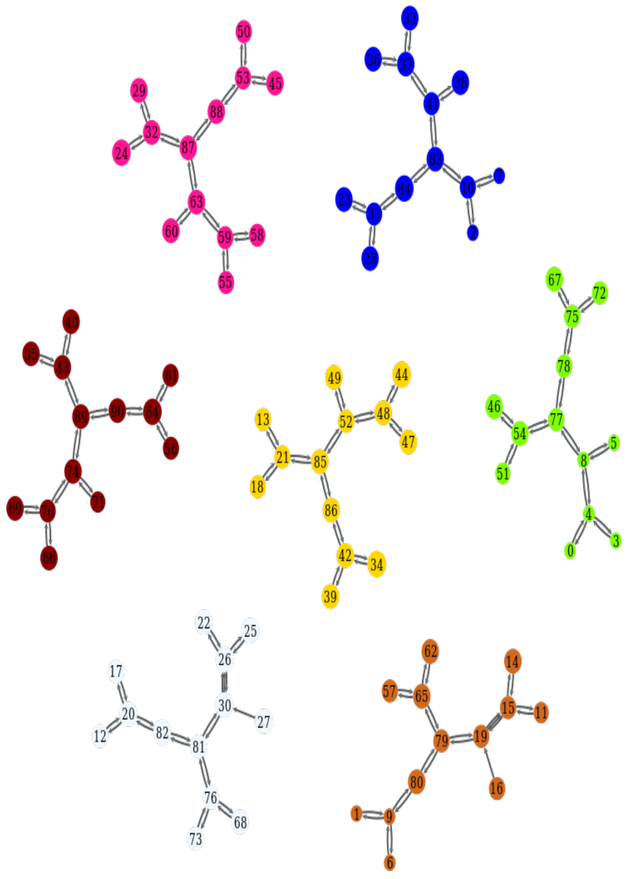

Figure 4.3: Matrix-Vector Product Partitioned Graph

Chapter 5

Architecture with Parallel Threads

This section presents the parallel architecture that we designed to increase the simulation speed. This architecture consists of a Network-On-Chip(NOC), Processing elements capable of parallel computing connected to the nodes of the NOC and Multi-Port Memory connected to another node of NOC. The Figure 5.1 depicts the basic design of the architecture.

5.1 Network On Chip (NOC)

We use the Network On Chip based architecture to control the data flow between multiple processing elements. The NOC is a network of routers represented as nodes where each consists if two data ports - input and output. The NOC has its own First In First Out (FIFO) Registers to control the data flow on the network. The NOC enables the communication between processor and memory and determines the data flow path for error free communication. We used the CMU CONNECT NOC [3] which is depicted in Figure 3.1.

5.2 Multi-Thread Processor

The processor core consists of multiple processor threads each capable of sending load and store requests to the memory. Currently, the architecture is designed in

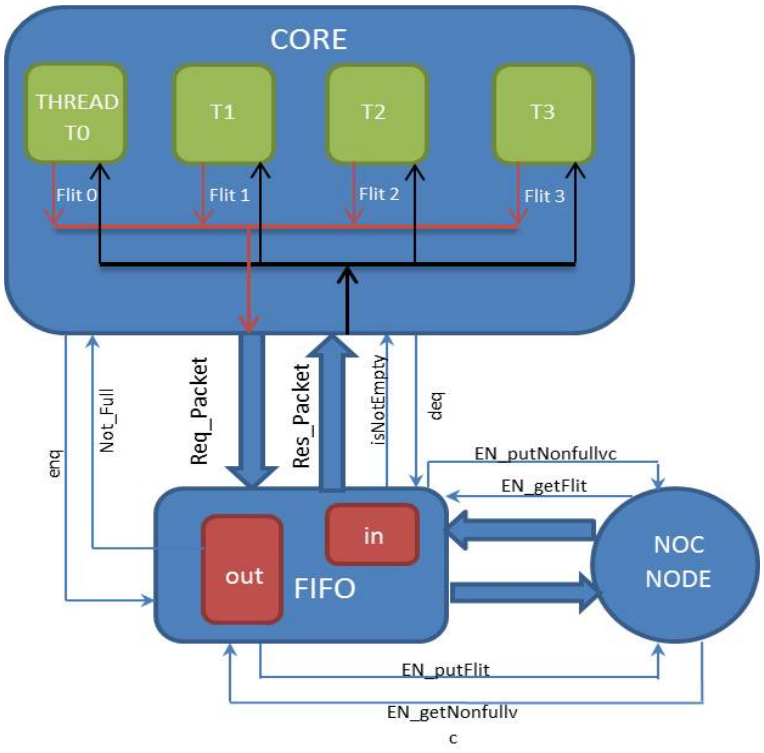

FIGURE 5.1: Architecture with Parallel Threads

such a way that load requests in the form of Flits for one primary input each is sent by all processors in parallel. While, requests for inputs in the same thread are sent sequentially. The thread processors wait till the responses arrive from the memory if it was a load request. When the responses for all the primary inputs at a particular thread is complete store requests are sent in parallel to the memory. Theoretically, this results in increase in simulation speed by a factor equal to (No. of Processors)raised to the power (No. of Levels) wrt the sequential processing.

The Figure 5.2 depicts the Finite State Machine for the thread processor. Initially, the processor is in $IDLE$ state. Then it moves to the $LOAD$ state when it gets the rdy signal from the NOC. The load/ store request is sent in the $LOAD$ state. It waits for the response in the $WAIT$ state. When the response arrives it moves to the $COMPUTE$ state where operation is executed. When all operations are finished processor moves to the $STORE$ state and sends a store request for data to be stored in the memory.

All the load/ store requests from the multiple threads are combined in the processor core to form a packet that is sent to the memory over the NOC. Similarly, all the response packet from the memory is split into multiple packets to send to the multiple processor threads. The data is sent and received through a FIFO to

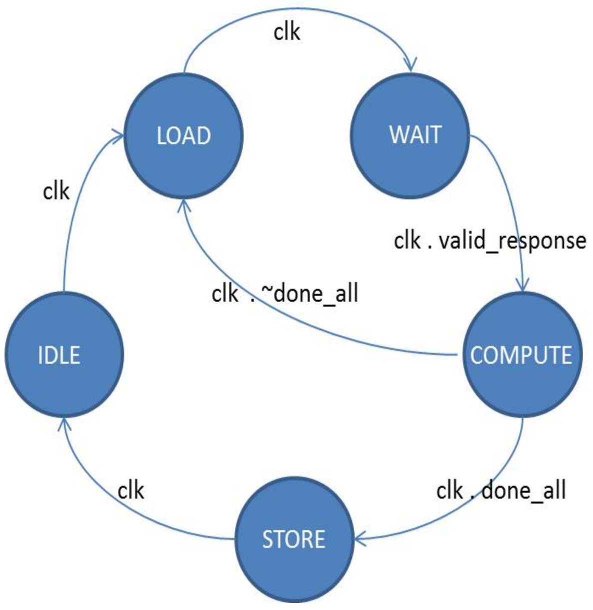

FIGURE 5.2: Processor - FSM

control the traffic.

The eight signals to the FIFO from MIPS and NOC perform:
enq - Enqueue to out-FIFO
not_Full - Checks whether out FIFO is full or not
deq - Dequeue from in-FIFO
is_Empty - Checks whether in-FIFO is empty or not
EN_putFlit - NOC node ready to accept flit
EN_getNonfullvcs - NOC node requests for the current virtual channel
EN_getFlit - NOC is ready to send flit
EN_putNonfullvcs - NOC node provides current virtual channel to the *in-FIFO*

Similarly, the memory core and all ports of the multi-port memory function in the same way except they receive requests from the processors and send responses back to the processors. The Figure 5.3 depicts the interaction of the memory with the NOC. The Figure 5.4 shows the request and response packet data.

5.3 Application

We applied this parallel threads hardware to a multilevel xor network. Network was composed of 9-input xor gates with 4 xors at each level. Data from the

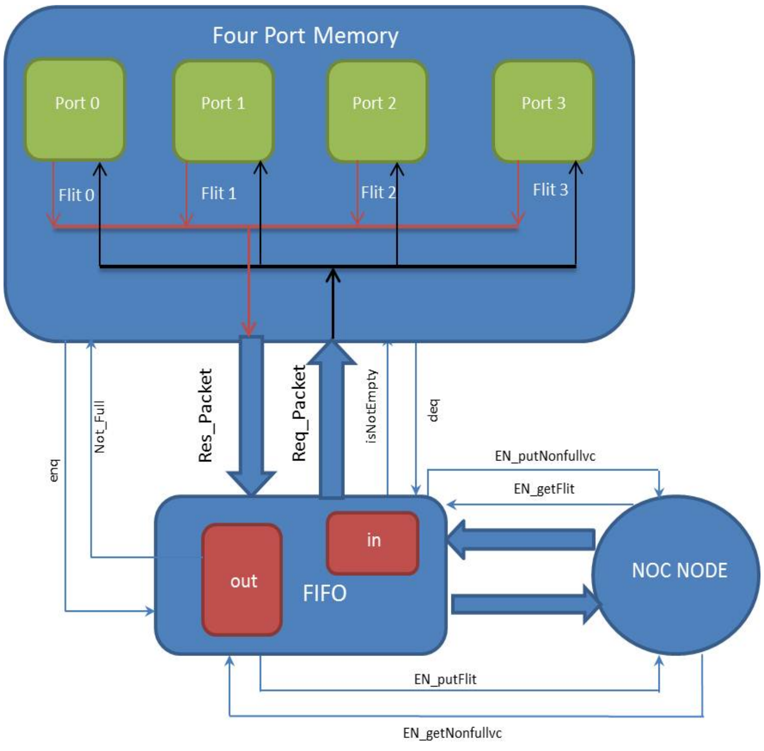

FIGURE 5.3: Memory NOC Interaction

Valid Bit	Tail Flit	Destination Address	Virtual Channel	Data

FIGURE 5.4: Data Packet

primary inputs as well as previous levels was stored in a register and randomly provided to the next level. The final results were verified by comparing them with the calculated expected values. The Figure 5.5 denotes the network of xor gates that was chosen to verify the architecture. We are still working on the timing and communication volume aspects of the network communication.

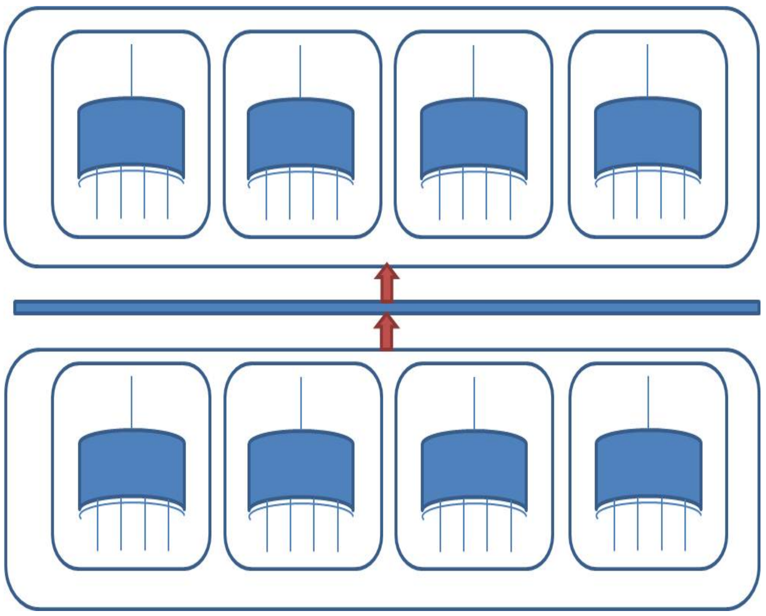

Figure 5.5: Xor Gates Logic Network

Chapter 6

MIPS Based Parallel Architecture

MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA). The instructions in a MIPS processor are stored in a local memory. The position of the program counter determines the address to fetch the instruction. The Figure 6.1 depicts the interaction of the MIPS processor with the local memory. The signals *memop* and r/w perform memory enable and read/write functions respectively. The address bus carries the PC address from the processor to the memory during the $update - PC$ state. The data bus fetches the instruction from the memory to the processor and instruction is received in the *read* state. The instruction operation is performed in the $Exec$ state. The data is written to the registers in the WB state. [5]

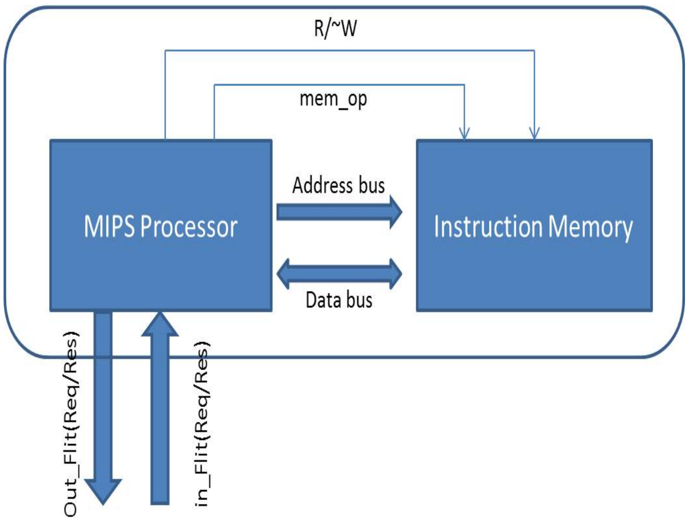

FIGURE 6.1: MIPS Based Parallel Architecture

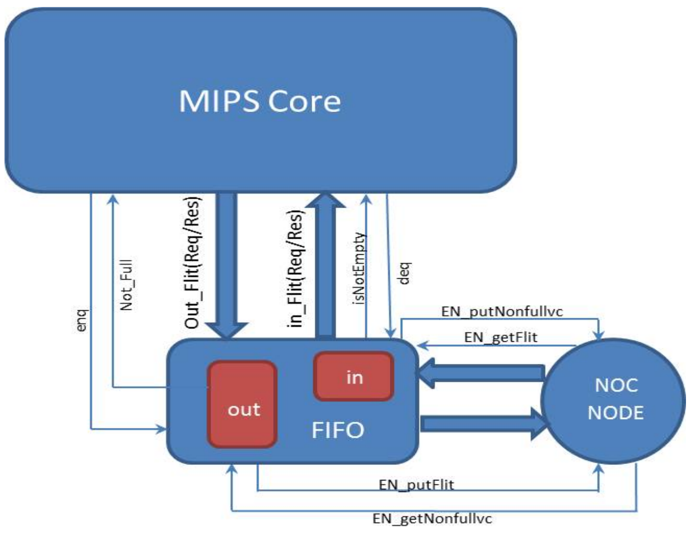

FIGURE 6.2: MIPS Memory Protocol

6.1 MIPS NOC Network

The MIPS processor is connected to the NOC node as shown in Figure 6.2. The data flow between the processor and the NOC node is regulated by a FIFO. The processor consists of two data ports - input and output that transfer the data to and from the NOC node. The MIPS processors are capable of requesting data from other processors and responding to the request from other processors. In addition to this the processor core is connected to a FIFO and establishes a handshake communication with the FIFO to control the data flow in the same way as discussed in the previous section. The MIPS protocol with the NOC is depicted in Figure 6.3. The $update - PC$ and $Read$ states function as mentioned above. The load/ store request is sent in the $Exec$ state and processor waits for the response to arrive in the WB state. The response data is written to the registers in the WB state. The data packet is same as represented in Figure 5.4.

When multiple MIPS processors are connected to the separate nodes of the NOC, parallel computation can be performed. The task is divided into smaller tasks and each smaller task is assigned to one MIPS processor. First, the processors perform the independent instructions and second, the dependent instructions are calculated by sending and receiving data over the NOC. This enables to increase the simulation speed (No. of processors) raised to the power (No. of instructions/ No. of processors) theoretically.

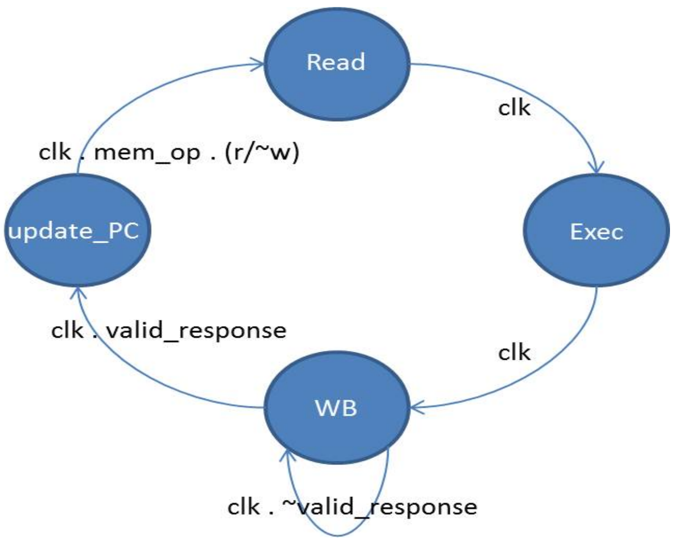

Figure 6.3: MIPS NOC Protocol

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48

Figure 6.4: Hamming Code Matrix Elements

6.2 Application

We applied this architecture to compute the sample Hamming code expressions. The hamming codes are error correcting codes used to detect and correct errors in computer data storage. The Hamming code expressions were generated considering a 7x7 matrix. The Hamming code expression for each element of the matrix was a function of all the elements that are in same row or column as the given element. The Figure 6.4 represents the matrix. For example, the element 14 will be a function of nodes 0, 7, 21, 28 , 35, 42, 15, 16, 17, 18, 19, 20.

The row based hamming code expressions for each node were as follows:
Dij = (((Di3 ^ Di4 ^ Di5 ^ Di6) ^ j[2]) + ((Di1 ^ Di2 ^ Di5 ^ Di6) ^ j[1]) + ((Di2 ^ Di4 ^ Di5 ^ Di6) ^ j[0])) & Dij_ prev ;

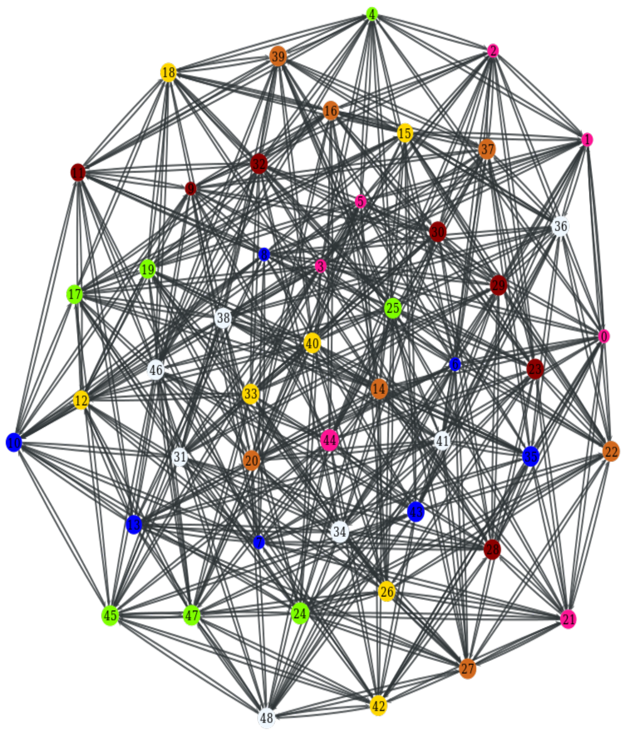

FIGURE 6.5: Hamming Code DAG Partition

where, Dij represent Matrix element values, j[k] represents kth bit of column no.(j) when it represented as a 3-bit value and Dij_ prev is the previous value of Dij.

Each element of the matrix was considered a node in the corresponding DAG. This forty nine node DAG was partitioned into 7 groups using Metis,the graph partition tool, and each partition was assigned to a single processor. Metis is an efficient graph partition tool with consideration to weighted edges and vertices, and minimizing communication volume. The partitions obtained are shown in Figure 6.5. So, we designed a network with 7 MIPS processors on 7 nodes of the NOC. First, each partition was made to fetch required data across the NOC and then perform the independent computations.

6.3 Conclusion

The results of the applications of the respective processors prove the correctness of the two architectures. The timing and communication volumes will be dealt with in future. Circuit partition allows the processor to divide a single task. The simpler tasks then can be simultaneously performed on parallel threads/ cores. Each thread could be a single FPGA and hence, potentially satisfying the circuit complexity constraint in FPGA and better simulation speed. The parallel architecture allows for data transfer of multiple packets of the same task across the network. The graph partition algorithms are essential to ensure the efficient division of task. The method parallel simulation acceleration provides scope of exponential growth in the the simulation speed. Reduction in the logic simulation time will speed up the verification process and hence, improving the quality of production process and reducing the time to market. Next, we would focus on the timing and communication volume aspects of architecture and develop a novel application-based graph partition algorithm to increase the efficiency of partition process. We would develop a generic logic simulator tool that could provide high speed simulation for any topology of gate level netlist.

Chapter 7

Additional Contribution

Block LU Matrix Factorization - Linear Equation Solver

The LU matrix factorization [15] expresses the given matrix as a product of a lower triangular matrix and an upper triangular matrix, $A = LU$. This method is used to solve a system of linear equations as described later in the section. However, this method requires to store the entire matrix. Since, the hardware has a limited memory, the hardware LU decomposition cannot be performed directly. For Example, for an NxN matrix with N=100,000 we require 76.3 gigabytes of storage for the matrix alone while the 32 bit processors are limited to 4 gigabytes of memory. Therefore, the matrix is stored in the software and blocks of matrix from software, whose sizes meet the memory constraints of the hardware, are sent to the hardware one at a time, and perform Block LU Factorization [16] as shown in Figure 7.1. Figure 7.2 depicts the first stage of hardware Block LU Decomposition. The following expressions describe the method:

$A00 = L00.U00$ (compute by L00, U00 by LU factorization)
$A01 = L00.U01, \quad U01 = L00/A01$
$A02 = L00.U02, \quad U02 = L00/A02$
$A10 = L10.U00, \quad L10 = A10/U00$
$A20 = L20.U00, \quad L20 = A20/U00$
$A11 = L10.U01 + L11.U11, \quad L11.U11 = A11 – L10.U01$ (obtained by LU factorization)
$A12 = L10.U02 + L11.U12, \quad U12 = (A12 – L10.U02)/L11$
$A21 = L20.U01 + L21.U11, \quad L21 = (A21 – L20.U01)/U11$

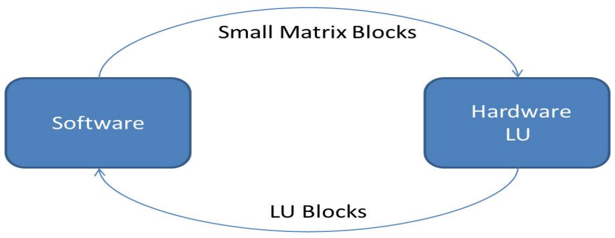

FIGURE 7.1: Hardware Matrix LU Factorization

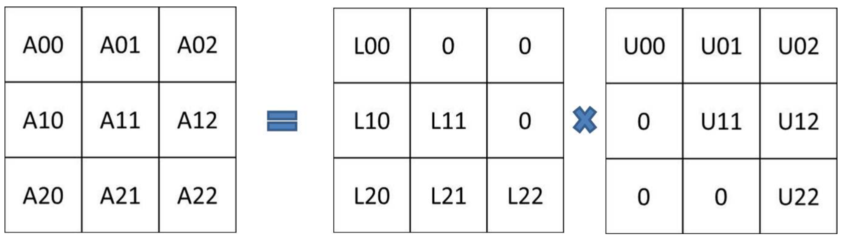

FIGURE 7.2: First Stage of Block LU

$A22 = L20.U02 + L21.U12 + L22.U22$, $L22.U22 = A22{-}L20.U02{-}L21.U12$ (obtained by LU factorization)

In the second stage, we start from A11 instead of A00 and same process is repeated till we start with the last block, A(N-1)(N-1).

Chapter 8

Appendix

8.1 Automation Script

Automation script for 1 input layer, 1 hidden layer and 1 output layer.

```
__fifo_interface__ __vivado_ap_none__ pe sigmoid(struct ...
    IOData sum, struct IOData &dataOut); # note: first ...
    argument type changed
__fifo_interface__ __vivado_ap_none__ pe ...
    partial_sum(struct IOData dataIn, struct IOData sum0, ...
    struct IOData &sum1, __state__ struct Label label);

__fifo_interface__  pe __read_next_data(struct IOData ...
    &data); # you replace
__fifo_interface__  pe __setcmd(struct Command &cmd); # ...
    you replace
__fifo_interface__  pe __init_zero(struct IOData &v); # ...
    vlot_dut provides
__fifo_interface__  pe __init_label(__state__ struct ...
    Label label, v_param int seed); # vlot_dut provides
__fifo_interface__  pe __incr_label(__state__ struct ...
    Label label); # vlot_dut provides


struct Label {
  int label;
};
```

```
struct IOData {
  float d0, d1, d2, d3, d4;
};
struct Command {
  8 : data;
};


InputLayer() {
  struct IOData data;
  struct Command cmd;
  loop (100) {
  __read_next_data(&data);
  send data to HiddenLayer;
  delay 60;
  }
}

HiddenLayer(){
  struct IOData din_0;
  struct IOData dout;
  struct IOData sum0;
  struct IOData sum1;

  __init_zero(&sum0);
  __init_label(label,seed= 0);

  recv din_0 from InputLayer;

  partial_sum(din_0,sum0,&sum1,label);
  activation(sum1,&dout);

  send dout to OutputLayer;
}

OutputLayer() {
  struct IOData din_0;
  struct IOData dout;
  struct IOData sum0;
  struct IOData sum1;

  __init_zero(&sum0);
  __init_label(label,seed= 1);

```

```
    recv din_0 from HiddenLayer;

    partial_sum(din_0,sum0,&sum1,label);
    activation(sum1,&dout);

    display dout;
}
```

8.2 Activation Functions

8.2.1 Sigmoid

```
y = 1/(1 + exp(−x));
```

8.2.2 Ramp

```
if(x > −1 && x < 1)
{
    y = x*0.5 + 0.5;
}
else if(x ≥ 1)
{
    y = 1;
}
else if(x≤−1)
{
    y = 0;
}
```

8.2.3 Piece-wise Quadratic

```
if(input≥0)
{
    x = input;
}
else {x = −input;}

if(x ≥ 0 && x < 2)
{
    negexp = 0.1987234*x*x − 0.8072780*x + 0.9748092;
}
else if(x ≥ 2 && x < 4)
{
    negexp = 0.0268943*x*x − 0.2168304*x + 0.4580097;
}
else if(x ≥ 4 && x < 8)
{
    negexp = 0.0016564*x*x − 0.0235651*x + 0.0840553;
}
else{negexp = 0;}

y = 1/(1+negexp);
if(input≥0){output = y;}
else {output = 1−y;}
```

8.3 Partial Sum

The following code shows partial-sum unit for a neuron node with three neurons.

```
#include "partial_sum.h"

void partial_sum(struct Label *label, struct Float *sum0, ...
    struct Float *sum1, struct IOData *dataIn)
{
#pragma HLS INTERFACE ap_none port=sum1
#pragma HLS INTERFACE ap_none port=dataIn
#pragma HLS INTERFACE ap_none port=sum0
#pragma HLS INTERFACE ap_none port=label
#pragma HLS ALLOCATION instances=fmul limit=2 operation
#pragma HLS ALLOCATION instances=fadd limit=2 operation
//#pragma HLS ALLOCATION instances=mul limit=1 operation
//#pragma HLS ALLOCATION instances=add limit=1 operation
float weights[216];
int n = label−>label;

// initialization
initialize(weights);

// calculation
sum1->sum0 = dataIn->data00*weights[9*n+0] + ...
    dataIn->data01*weights[9*n+1] + ...
    dataIn->data02*weights[9*n+2] + sum0->sum0;
sum1->sum1 = dataIn->data00*weights[9*n+3] + ...
    dataIn->data01*weights[9*n+4] + ...
    dataIn->data02*weights[9*n+5] + sum0->sum1;
sum1->sum2 = dataIn->data00*weights[9*n+6] + ...
    dataIn->data01*weights[9*n+7] + ...
    dataIn->data02*weights[9*n+8] + sum0->sum2;
}
```

8.4 Ramp Activation Unit

The following code shows ramp activation unit for neuron node with five neurons.

```
#include "sigmoid.h"
#include <math.h>

void sigmoid (struct IOData *sum, struct IOData *dataOut )
{
#pragma HLS INTERFACE ap_none port=dataOut
#pragma HLS INTERFACE ap_none port=sum
#pragma HLS ALLOCATION instances=fmul limit=1 operation
//#pragma HLS ALLOCATION instances=fadd limit=1 operation
//#pragma HLS ALLOCATION instances=mul limit=1 operation
//#pragma HLS ALLOCATION instances=add limit=1 operation
#pragma HLS ALLOCATION instances=icmp limit=3 operation

//--onetime
float sum1[NUM_NEURONS];

float dataOut1[NUM_NEURONS];

// copying struct to array
sum1[0] = sum->d0;
sum1[1] = sum->d1;
sum1[2] = sum->d2;
sum1[3] = sum->d3;
sum1[4] = sum->d4;

int i;
float x,y;

// calculation
sigmoid_label0:for(i=0;i<NUM_NEURONS;i++)
{
        x = sum1[i];
        if(x > -1 && x < 1)
        {
            y = x*0.5 + 0.5;
        }
        else if(x ≥ 1)
        {
            y = 1;
        }
        else if(x≤-1)
        {
            y = 0;
        }
        dataOut1[i] = y;

}
//copying array to struct
dataOut->d0 = dataOut1[0];
dataOut->d1 = dataOut1[1];
dataOut->d2 = dataOut1[2];
dataOut->d3 = dataOut1[3];
dataOut->d4 = dataOut1[4];
}
```

References

[1] Valeria Bertacco, Debapriya Chatterjee, High Performance Gate-level Simulation with GP-GPU Computing, Department of Computer Science and Engineering, University of Michigan

[2] Aman Khanna, Logic Simulation Acceleration using Custom Processor, Master's Thesis, IIT Bombay, 2015

[3] Michael Papamichael, Configurable Network Creation Tool, Computer Science Department, Carnegie Mellon University, 2012

[4] METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering, Karypis Lab, 2013

[5] Sanket Diwale, Application Mapping to Manycore Architectures, Dual Degree Dissertation, IIT Bombay, 2013

[6] Lutz Prechelt. PROBEN1 -A Set of Neural Network Benchmark Problems and Benchmarking Rules, July 1995.

[7] T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin and V. Srikantam. A GENERIC RECONFIGURABLE NEURAL NETWORK ARCHITECTURE IMPLEMENTED AS A NETWORK ON CHIP. IEEE 2004.

[8] Amos R. Omondi, Jagath C. Rajapakse. FPGA Implementations of Neural Networks. Springer 2006.

[9] Félix Moreno, Jaime Alarcón, Rubén Salvador, and Teresa Riesgo. Reconfigurable Hardware Architecture of a Shape Recognition System Based on Specialized Tiny Neural Networks With Online Training. IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, VOL. 56, NO. 8, AUGUST 2009.

[10] Dmitri Vainbrand, Ran Ginosar. Scalable network-on-chip architecture for configurable neural networks. Microprocessors and Microsystems 35 (2011) 152–166

[11] Nadia Nedjah, Rodrigo Martins da Silva, Luiza de Macedo Mourelle. Compact yet efficient hardware implementation of artificial neural networks with customized topology. Expert Systems with Applications 39 (2012) 9191–9206.

[12] Fernando Morgado Dias , Ana Antunes , Alexandre Manuel Mota. COMMERCIAL HARDWARE FOR ARTIFICIAL NEURAL NETWORKS: A SURVEY

[13] Janardan Misra, Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. Neurocomputing 74 (2010) 239–255

[14] Jinhwan Park and Wonyong Sung. FPGA BASED IMPLEMENTATION OF DEEP NEURAL NETWORKS USING ON-CHIP MEMORY ONLY.

[15] Alan George, Joseph Liu, and Esmond Ng, A Data Structure for Sparse QR and LU Factorizations, SIAM J. Sci. and Stat. Comput., 9(1), 100–121.

[16] Block LU Factorization, Accuracy and Stability of Numerical Algorithms, 13(1), 245-248.