NeuroNoC: A Network on Chip Approach for Implementing Dynamic and Scalable Neural Networks on Reconfigurable Platforms

Authors

***Abstract*—The utilization of FPGA in implementation of Neural Networks (NN) presents better performance in contrast to other hardware realizations and software-based implementations of NN. For real-time applications, FPGAs are well suited and more efficient in terms of time and cost. Large number of neurons in NN has been always a challenge in hardware realization in terms of resource utilization. Additionally, on-chip learning of NN leads to consumption of more resources of FPGA as the learning is performed by many multiplication processes and it requires higher precision. In this work, we present the design and implementation of FPGA-based NN, called NeuroNoc, which utilizes off-chip learning. The proposed NeuroNoC allows the user the novel technique to adjust the number of neurons, interconnections, weighs and biases of each neuron by introducing the configuration packet format of NN that differs from the input signal packet format. The packets, which alters the NN parameters, are sent from the same node as the input signals to NN. The implementation is supported by both on-premise FPGAs as well as cloud-based instances. The NeuroNoC platform is available as an open-source, allowing other researchers and small research centers to transfer the software-based NN to hardware.**

***Index Terms*—FPGA, Artificial Neural Network, Reconfigurability, Configuration Packet**

I. Introduction

The growing demand for automated complex intelligent systems leads to dramatic changes in the development and deployment of Artificial Neural Networks (ANN). The capabilities of ANN to map, model and classify nonlinear systems have allowed to incorporate it in various applications in such fields as science, engineering and economics [n]. The development and implementation of ANN has been done mostly in software. However, despite the benefits of software-based ANN such as high level of abstraction (i.e. no need to know the inner workings of ANN for designers), it has severe problems in real-time applications in terms of execution time in contrast to its hardware-based counterparts [e]. In order to address that issue, there have been proposed several hardware adaptations of ANN [f]. Indeed, even though the software implementation offers flexibility, the high-speed operation in real-time applications is only achievable with the hardware-based networks [g].

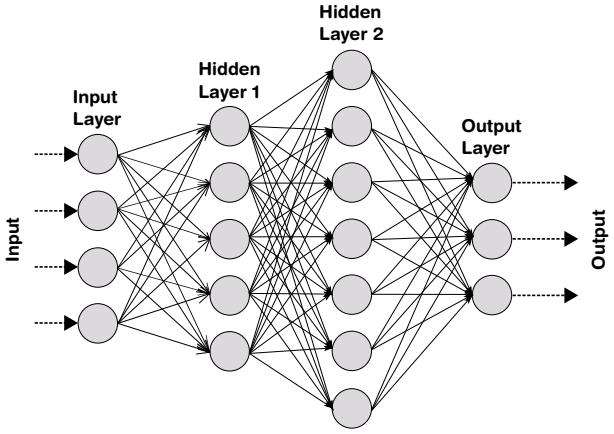
There has been proposed numerous hardware architectures of ANN, which can be classified into two groups: analog and digital systems. The latter is more popular, since it provides higher accuracy, noise immunity, better scalability, higher flexibility and compatibility. There are three types of digital implementations of hardware-based ANN: field programmable gate array (FPGA) based, digital signal processor (DSP) based and application specific integrated circuit (ASIC) based implementations. The latter two types of architectures are less suitable for ANN implementation than the first type. DSP based architecture, for instance, is mainly sequential, as a consequence, it lacks to provide parallelism in ANN. ASIC based architectures, on the other hand, does not offer flexibility to reconfigure the ANN by the user [n]. It must be noted that ASIC and DSP can be designed to be highly parallel, however, it is expensive and complicated process to design [e]. FPGA is the best option to implement ANN, since it offers fully parallel and reconfigurable design capabilities.

Reconfigurability of FPGA allows the user to design application specific hardware architecture [g]. FPGA implementations of ANN with a large number of neurons is one of the challenges in hardware-based realizations, since ANN algorithms usually consists of vast amount of multiplication processes and requires higher precision [e]. Hence, on-chip learning is considered to be difficult and useless, as it causes a loss of efficiency in a hardware realization. As a result, off-chip learning is usually chosen when there is no necessity in dynamic learning. Also, the design changes can be made within a few hours, which significantly saves the time and cost of design production. Most of the hardware-based realizations are implemented in such a way that the structure of ANN can be altered through the reprogramming of FPGA [e].

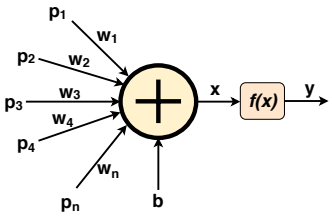
In this paper, we propose the open-source reconfigurable hardware-based realization of ANN, which allows to alter the structure of ANN through the routing of packets in a system. ANN is implemented by the use of mesh topology, which consists of four-way switches with a programming element (PE) connected to it. Each switch with its PE represents a single neuron in ANN. Off-chip learning is chosen in our architecture due to the above-mentioned advantages over on-chip learning. The novelty of the proposed architecture is in the capability of changing the interconnections, weighs and biases on each neuron without explicitly reprogramming the FPGA, but by introducing the ANN configuration packet format, which can be given to the chip the same way as the usual input data packet.

The main contributions of this work are:

- Detailed architecture description of the FPGA-based implementation of ANN



(a) Layered Architecture of Neural Network



(b) A Single Artificial Neuron

Fig. 1. Architecture of a traditional feed-forward neural network and a single artifical neuron

- An open-source implementation of ANN targeting the Xillinx Virtex-7
- Characterization of the proposed platform in the terms of resource utilization and performance.

The rest of the paper is organized as following: Section 2 provides the background, Section 3 discusses the background, Section 4 presents the detailed description of the proposed architecture, Section 5 provides the results f the resource utilization and performance analysis, and Section 6 concludes the paper.

II. Background and Related Works

[1] Artificial Neural Networks (ANNs) are inspired and adapted models of biological mammalian brain []. An ANN follows a layered architecture, where each layer is constructed form several neurons as shown in Fig. 1(a). In biological neurons, the signal transition takes place between neurons through a structure called *synapse*. Each synapse has its own weight, which affects the signal passing though it: the signal is multiplied by the weight of the corresponding synapse. The inputs to the neuron from the synapses are summed up in the neuron and passed further to the activation function. Activation function transforms the input signals to output signals, and its main function is to make decision regarding how the output should behave to certain inputs. Fig. depicts the neuron with the basic operations it performs.

ANN can consist of many layers, which are classified into three groups: input layer, hidden layer and output layer. One layer can be fully or partially connected to the next layer. Hidden layer can be built from many intermediate layers between input an output.

Number Representation

One critical factor in hardware implementation of ANN is the number representation of different data - synapse weights, biases and inputs/outputs of the neurons. Despite its widely usage in software ANNs, floating point arithmetic due to its prohibitively expensive cost is not well suited for hardware ANNs. As a result, the two's complement fixed binary point representation was chosen for data representation as it brings considerable optimization in terms of area usage and speed performance. However, this implies limited precision which is enough for many applications. Nevertheless, with this scheme learning process is accomplished off-chip in software using floating point representation. The conversions from decimal fractions to fixed point binary is done through equation 1.

$$b_x(d_x) = \lfloor d_x \cdot \frac{2^{n_x-1}-1}{2^{i_x-1}-2^{-f_x}} \rceil \tag{1}$$

$$d_x(b_x) = b_x \cdot \frac{2^{i_x-1}-2^{-f_x}}{2^{n_x-1}-1} \tag{2}$$

Here, n_x, x, f_x amount to total number of bits, integer bits and fractional bits of the given variable.

The hardware options utilized for machine learning are Central Processing Units (CPU), Graphic Processing Units (GPU), FPGA and Application-Specific Integrated Circuit (ASIC). Each of these options has their own advantages over others. CPUs are extremely flexible in terms of programmability, but they have issues regarding parallelism, cost and heat. GPUs have become most widely used hardware option for executing machine learning and deep learning tasks [tpu2]. They are designed to provide high level of parallelism, and thus well suits for the demand of machine and deep learning, where a lot of matrix multiplications and convolutions are involved. However, GPUs need to be incorporated with other chips (e.g. CPU, FPGA) that can rapidly execute NN on it (i.e. inference), since they are more suitable for training rather than inference. FPGAs have recently become popular for machine learning, and companies such as Microsoft and Baidu have invested in FPGAs heavily. FPGAs have much less power usage compared to CPUs, and their flexibility offers low latency and high bandwidth. Almost all implementations of ANN on FPGAs use various methods to utilize the reconfigurability of FPGA hardware. According to recent studies, FPGAs have outperformed GPUs in many applications, and it is predicated that they will soon be a better option than GPUs in deep learning [tpu2]. Even though ASICs are the least flexible among other above-mentioned hardware options, they compensate this drawback by offering highest efficiency. ASIC can be designed for both training or inference processes. There are many examples of ASIC that are designed to meet requirements of machine learning and ANN. Googles Tensor Processing Unit (TPU) and IBMs TrueNorth are the best examples of successful ASIC deployments for NN purposes. Being

originally focused on 8-bit integers for inference tasks, TPUs now provides floating point precision and can be deployed in training as well [tpu2]. High performance and power efficiency of TPUs are achieved due to lack of extraneous logic[tpu2]. Hence, unlike other abovementioned hardware options, TPUs cannot be reprogrammed. IBMs TrueNorth is a digital chip that consists of one million spiking-neurons incorporated with 256 million synapses. TrueNorth provides high efficiency due to spiking-neuron technology, and high scalability as several chips can be tiled in two- dimensions, as well as flexibility by offering independent configurability of individual neurons and synapses (noc2). TrueNorth is fabricated using CMOS technology and uses offline learning in NN purposes (noc2).

III. Proposed Architecture

The overall architecture of NeuroNoC is depicted in Fig. **??**. The NoC is arranged in mech topology with each PE representing a neuron, except the one at the bottom-left corner. Mesh topology was selected based on its relatively higher bisection bandwidth as well as its suitability in implementing multicast systems. The size of the NoC is configurable and can be set by users based on the type of the application and the resource capacity of the target FPGA. The bottom-left PE (with zero X and Y coordinates) is used to inject data to the network from external world. In other words this PE represents the entire input layer of the NN. The output layer of the NN sends back the network output to this PE for sending it back to the host machine. Due to the packet-switched nature of NoC architecture, the packets may reach the output PE in out-of-order. Packets are reordered and sent back to the host computer based on the sequence number embedded in the data packets.

A. Packet Formats

NeuroNoC implements a variety of packet formats for supporting network and neuron configurations as well as for inter-neuron communication. Fig. 2 depicts the different packet types and their corresponding fields. For a given NeuroNoC implementation, all packet types have same size but depending on the size of the NoC, packet size varies. This is mainly due to the variation in the size of the address fields which are NoC size dependent.

Data Packet: Data packets are used for inter-neuron communication. Once a neuron receives data packets from all its predecessors, it calculates the output based on inputs, weight values, bias and the activation function. It sends out another data packet with the calculated output in the *Data* field and the neuron address in the *source* field. The index field has significance only when the packet source is zero (bottom-left PE). Since the entire input layer is modeled by this single PE, the index number differentiates the different neurons of this layer.

Input number and Bias Configuration Packet: This configuration packet is used for setting two parameters of each neuron. The *Input Number* field configures the number of predecessors from the previous layer. The *Bias* field sets the neuron bias for calculating its output. The *Destination* field is used by the NoC to route the packet to appropriate neurons.

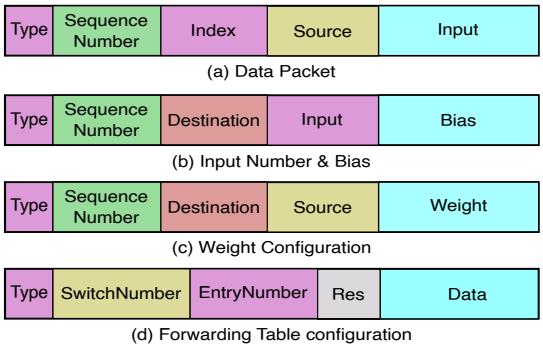


Fig. 2. NeuroNoC packet formats for supporting neuron and network configurations and data communication

Weight Configuration Packet: The Weight Configuration Packet sets the input weights for each neuron by storing the *Weight* field in its internal Weight Table. The *Source* field enables mapping a a particular weight with a particular input. Infact the Source field is used as the write address when storing weight in the weight table.

Routing Table Configuration Packet: These packets are used for configuring the NoC switch routing tables, enabling multicast routing algorithm described in subsection III-D.

B. Artificial Neuron

Individual neurons of the NN are implementing by the processing elements (PEs) of the NoC. Each neuron receives inputs from neurons in the previous layer, multiplies them with corresponding weights, sums up and generates the output based on an activation function. During configuration stage, each neuron is configured with the number of input neurons, weights corresponding each input and bias. Fig. 3 shows the different submodules of an artificial neuron and are described in the subsequest sections.

1) **Sequence Number Checker(SNC):** Since packet switched NoC does not preserve packet delivery ordering, neurons support out-of-order packet delivery by storing and reassembling the incoming data packets. The input layer (bottom-left PE) assigns each data packet a sequence number in the *sequence number* field. It is to be noted that the number assigned corresponds to an input number rather than to a single packet. In other words, an input with *n-feature vector space* will be sent as n packets with the same sequence number. After processing the input data, the output packet from a neuron retains the same sequence number as that of the input packets.

The data packets arriving from the network interface are initially stored in a hash table (HT). The table consists of 2^{SEQ_WIDTH} blocks, where SEQ_WIDTH amounts to the number of bits in the seqNum field. For a given NoC of size S_{NOC}, in the worst case each neuron may receive inputs from S_{NOC} neighboring neurons. Thus, every block in HT is further divided to S_{NOC} memory units. The data packets are mapped to HT memory through the following hash function,

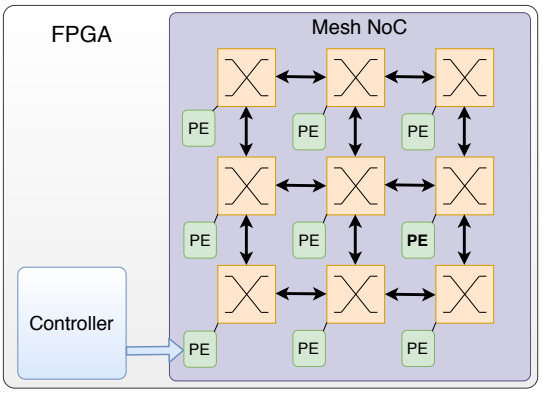


Fig. 3. Architecture of the proposed artificial neuron

$$hf(seqNum, count[seqNum]) = concat(seqNum, count[seqNum]) \quad (3)$$

Where *count[seqNum]* is the number of packets with the same sequence number already received by the neuron. The sequence number count is traked by the counter array (CR), which has a dedicated counter for each sequence number. After a packet is stored in the HT, the corresponding sequence number counter is incremented, so that the next packets with the same sequence number is stored in the next memory location. The SNC State Machine (SSM) keeps track of the current sequence number and the corresponding counter to retrieve packets from the HT once the neuron receives packets from all its neighbors.

2) **Multipy-Accumulate Unit (MAC):** The computation of the total synaptic input to the neuron is implemented by successive multiplication and addition operations (MAC operations). Once the SNC module detects that the neuron has received input packets from all its neighbors, it sends a control signal to the MAC module. The MAC module then reads the input one-by-one from the hash table (HT) and multiplies and accumulates using the corresponding weight values read from the weight table (WT). Weights are stored in WT through the configuration packet described in Section III-A. Assuming the same format for neuron input and output data, the number of bits to represent weighted sum is

$$N_s = \lceil \log_2(S_{NOC} \cdot (2^{n_w-1})(2^{n_z-1}) + 2^{n_b-1} \cdot 2^{f_z+f_w+f_b}) \rceil + 1 \quad (4)$$

The number of inputs and bias are configured with the same control packets which were used during SNC.

3) **Activation Function (AF):** The ANN implementation supports a variety of activation functions with the help of look-up-tables (LUTs). By changing the contents of the LUT, the activation function can be modified. LUT-based implementation of a complex and non-linear activation function, such as sigmoid function, may require a large memory depending on the desired precision. If we define n_s as the most significant bits of N_s, increasing the value of n_s improves the accuracy at the expense of memory size. The minimum value at which all the possible output values are present in the LUT is given by

$$n_s = i_s - \lceil \log_2(\frac{d_z(1)}{f'(0)}) \rceil \quad (5)$$

Moreover, if we consider sigmoid function, most of the entries located far from 0 are duplicated. Considering this, it is possible to reduce the size of LUT to store the values in the interval $[x_{high}, x_{low}]$, in where the expressions for x_{min} x_{max} are given by

$$x_{high} = d_s(\ln 2^{f_z-1}), x_{low} = -x_{high} \quad (6)$$

and the number of bits to address the LUT is $n_{LUT} = \lceil \log_2(x_{high} - x_{low} + 1) \rceil$. If the computed weighted sum falls within this interval, the output is taken from the LUT otherwise the output is assigned 1 or 0 based on whether the value is above x_{high} or below x_{low}.

C. Switch

The detailed architecture of the NeuroNoC switch is depicted in Fig. III-C. Each switch is capable of sending and receiving packets from five direction; North, South, East, West and the associated neuron. Separate FIFOs are present for receiving and transmitting packets from each direction. The interfaces follow AXI4-Stream interface for inter-swich as well as neuron-switch communication. Switches serve the FIFOs in a queue in a clockwise direction. Flow control is implemented through AXI-stream control signal as shown if figure. The o_valid wires are asserted whenever switch transmits the data for certain direction including PE. Similarly, whenever the data comes from other switches or PE the i_valid wire of incoming side of the switch is asserted. The switch asserts the o_ready signal whenever the FIFO has empty slot to accept the data. All FIFOs should hold the data on the bus until data is transmitted to all necessary FIFO. Each switch has registers for routing table, size of total size of NoC rows with 5 bits length. Switch has finite state machine controller, which routes the packet for certain directions depending on the packet type and refreshes the routing table. In one process time one packet can be transmitted to several FIFOs. Switches communicate with each other through FIFOs.

D. Packet Routing

Switch receives the packet from one FIFO and routes the data for required FIFOs. Finite state machine controller has three main approaches for routing the packets depending on the type of incoming packet. For weight and bias configuration packet switch sends the packet using the number of switch in the NoC. Depending on the number of destination switch, the present switch can route the packet for four directions. If the destination switch is located to the above, below, to the left or the right, it sends to North,South, East, West FIFOs respectively. Once the packet came to the predefined switch, instantly it will be sent to the PE FIFO corresponding to that

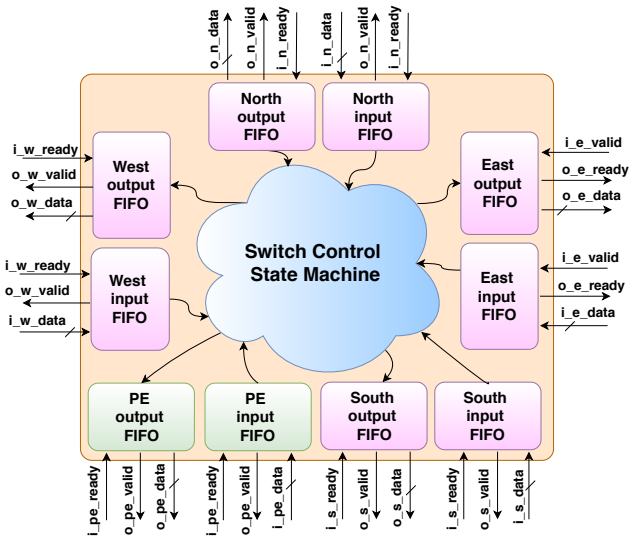


Fig. 4. Switch Architecture

switch for configuring the weight and bias. For routing table configuration packets switches behave the same as for weight and bias configuration oriented switches. The one difference is, when the packet came to the destination switch, it will refresh one slot of routing table and switch will serve next FIFO.

While previous packets can be sent only for one direction(unicast) from one switch, the data packets can be replicated and sent for all direction including PE in one process time (multicast). It was achieved by the logic of routing table. Each switch has routing table, which must be configured before usage. The rows in routing table is equal to the total amount of switches in NoC. The actual meaning of rows in routing table is address of source switch. Whenever data packet comes, switch checks the source provider address of this packet and takes the row, which number is equal to source address. Each row has 5 bits for the decision to sending the packet for certain direction. Each bit corresponds to 5 directions: North, South, East, West and PE. If the bit is equal to 1 then the incoming packet must be sent to corresponding direction,conversely, if bit is equal to 0 then the packet must not be sent to that direction. The routing table configuration directly depends on the neural network model and configuration packets must be created manually or using other software. For reliability of data in multicasted transmission, switch waits until the data is sent for all necessary direction, only after that it checks next FIFO. For implementation of packet routing the finite state machine principle was used and it is shown in figure **??**.

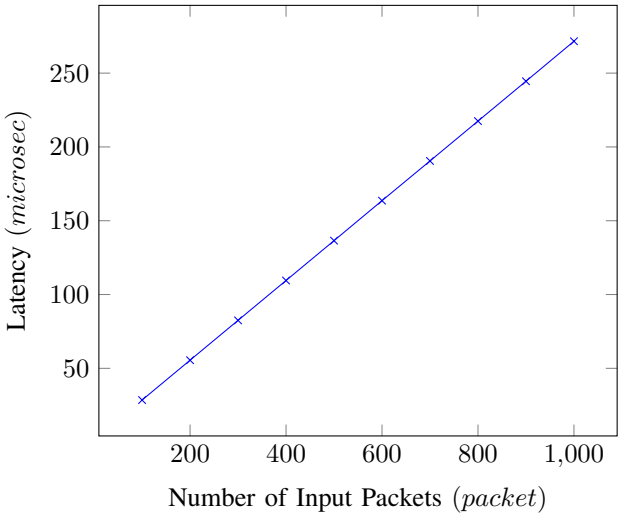


Fig. 5. Relationship between latency and the number of input packets

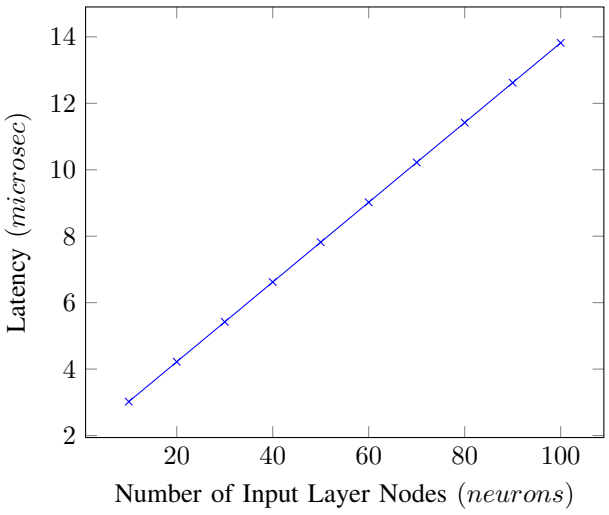


Fig. 6. Relationship between latency and the number of nodes in the input layer

IV. Results and Discussion

V. Conclusion and Future Works

References

[1] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown, "Overview of the SpiNNaker system architecture," *IEEE Transactions on Computer*, vol. 6, pp. 2454–2467, 2013.