Ingénierie des Systèmes d'Information
Vol. 30, No. 4, April, 2025, pp. 1101-1110
Journal homepage: http://iieta.org/journals/isi

ineuralFPGA: Implementation of an Optimized DNN Model for Real-Time Applications

Vineet Jain[1], Abhishek Sharma[1], Prateek Jain[2], Augusto Bezerra[3], Sunil Sharma[4*], Sultan Alasmari[4,5]

[1] Electronics & Communication Department, and L-CST, The LNMIIT, Jaipur 302017, India
[2] Electronics & Instrumentation Engge. Department, Nirma University, Ahmedabad 382481, India
[3] SpaceLab, Federal University of Santa Catarina, Florianopolis 88040-900, Brazil
[4] Department of Information Systems, College of Computer and Information Sciences, Majmaah 11952, Saudi Arabia
[5] College of Technology and Business, Riyadh Elm University, Riyadh 12734, Saudi Arabia

Corresponding Author Email: s.sharma@mu.edu.sa

https://doi.org/10.18280/isi.300425

Received: 3 April 2024
Revised: 31 January 2025
Accepted: 14 February 2025
Available online: 30 April 2025

Keywords:
FPGA, Artificial Intelligence

ABSTRACT

The real-time uses of intelligent computing models are becoming most prominent part of recent technologies. Implementation of computing model is required using multiple framework for design perspective. To create the smart era, the real-time implementation is needed for different applications. This paper presents efficient, faster, and hardware friendly implementation for synthesizable deep neural network (DNN) which are targeting low-cost hybrid FPGA platforms like Xilinx Zynq, Micro Blaze, Micro Zed, etc. "neural net" is an IP core neural-network written in Verilog HDL, parameterized via Python and based on hardware-software co-design approach providing flexibility to utilize both hardware and software aspects of design. Results show significant performance and accuracy with minimal resource utilization. The Fully Connected network with 8-bit of data with int. part of 6bits and "relu" activation function achieves best-optimized results for small architecture like [784,30,20,10] with the accuracy of around 98%, power of 0.57 Watt with 195.407 MHz of max frequency on EDGE Zynq FPGA. The implementation of proposed "neuralnet" on FPGA demonstrated the large applications in consumer electronics era.

1. INTRODUCTION

In the present era, with the rapid growth of technology in the field of IoT and Embedded systems, we generally require computational devices [1]. It is mandatory to target to achieve energy efficient, highly secured with low cost, power, and resource consumption and an increased performance of the device. With an era of increasing complexity, it is challenging to solve problems by the algorithmic approach. Hence the foundation for the machine learning approach was laid. Neural networks find a way of transforming data into decisions [2].

Nowadays, with an increase in demand of IoT-enabled applications, this interest of neural nets implementation on low-cost edge computing devices has increased [3]. The applications of neural network for IoT enabled devices has been represented in Figure 1. It has been shown many times that FPGA-based models can outperform their software implementation of DNN because of its concurrency, re-configurability etc. [4]. There are few hardware platforms available with their own factors and constraints (like limited power, size, etc.). These are kept in mind while developing an algorithm or any hardware design that differs totally from where scientific research over these occurs. Due to cost, power, and development time on data-centers, these methods are now becoming recent trends for consumer era [5]. Till present time, AI models have been validated and tested using virtual platforms. But it is required to implement the AI or predicting models for real time application, which enhance visibility to adapt the new technology for smart scenario. The performance parameters are the important benchmarks in the era of implementation.

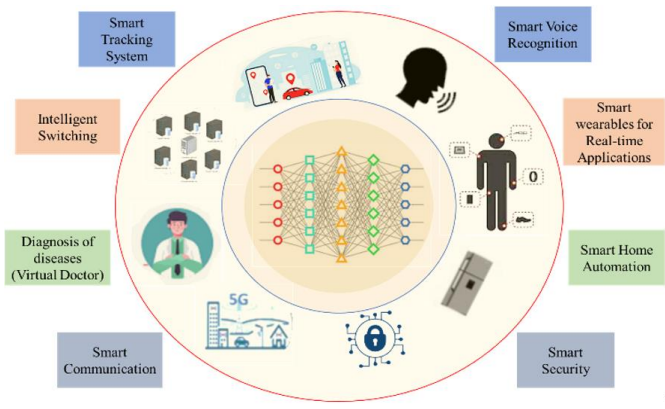

Figure 1. General applications of computing model for IoT enabled devices

The organization of the paper has been represented, which demonstrate the connecting discussion on proposed work. In Section 3, the necessity of neural networks in the current context is elucidated. Additionally, this section delves into the advantages and disadvantages of neural networks, reviews

relevant prior work, and explores the novel contributions presented in this study. Description of the proposed model with architecture selection and simulation have represented in Sections 4, 5 and 6 respectively. Applicability in different scenario for proposed design and conclusion of the work has been explored in Sections 7 and 8. This particular organization enlightens the feasibility of the proposed model in real time domain.

2. LITERATURE REVIEW

2.1 Why neuralnet implementation is required for consumer electronics

As per the recent demand in the market, the computing or predicting model should be highly efficient in terms of performance parameters after real time implementation. The complicated computing model should be implemented with minimum hardware resources for better results. With the rapid growth of technology, neural computing has become a prominent way of dealing with complex computing problems. Nowadays, with the prevalence of smart and IoT-based applications, the shift in momentum towards smart designs using neural networks has opened new opportunities for low-cost Hybrid FPGAs as they are based on concurrent computing and freely programmable configurable logic blocks (CLBs), etc. A digital circuit representing neural network embedded on FPGA introduced a physical design for intelligent computation with high speed and low cost solution. The proposed programmed FPGA can be used for multiple paradigm which are represented in Figure 2. Due to significant advantages of physical computation model design, it is highly recommended for real-time applications.

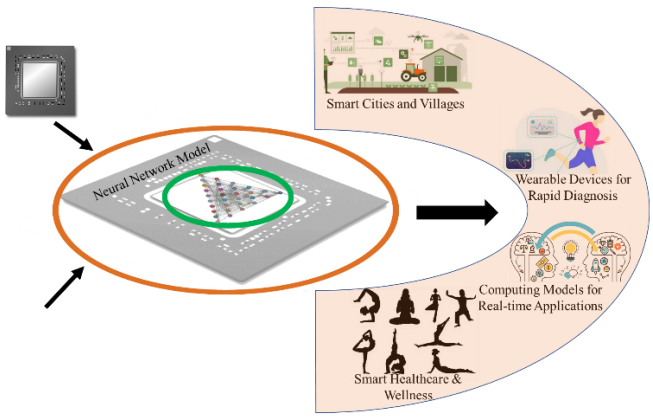

Figure 2. Applications of DNN model implementation on FPGA for consumer electronics

2.2 Prior related work

This section reviews some important observations and results captured in neural network implementation, focusing on FPGAs-based methodology and solutions. The DNN model implementation has been done for different applications. For architecture point of view, many people represented the challenges to optimize the parameters such as speed, power & resource consumption and precision level of implemented model. Depending upon the various features of DNN models, the parameters were optimized at individual level for multiple applications. But, there is a requirement to set the parameters as an optimized gradient which would be applicable for multiple purpose. For such kind of objective, people are still working to set the standard parameters for their applications. In this way, Guo et al. [6] represented a design flow of CNN mapping on an FPGA. Various FPGA platforms have been analyzed to get the optimized parameters. Radial basis function based neural network classifier is proposed for FPGA implementation [7]. Wang et al. [8] demonstrated deep CNN model on Xilinx Zynq ZC706 and Virtex VC707 boards with 86.25% claimed accuracy. Shawahna et al. [9] represented neural networks in image detection and recognition applications. Lian et al. [10] developed an FPGA-based DNN model using Virtex-7 XC7VX690T FPGA with more than 1000 DSPs. Nguyen et al. [11] explained CNN model for object detection with reduced DRAM power consumption. CNN model implementation has been done using 512 DSP blocks in terms of an optimized implemented model [12]. Abd El-Maksoud et al. [13] explored the CNN model for image recognition and object detection. 91% average classification accuracy has been claimed with 3.92 W power consumption only. A low resource CNN based accelerator is demonstrated for drones with 1.9 W power consumption. The model is implemented on two FPGA-based platforms Ultra96-V2 and PYNQ-Z1 [14]. The presented work enlightens the implementation with optimized parameters. The challenges have been overcome for future perspective. Still, the parameters are required to optimize for specific applications in design perspective. Hence, the DNN model implementation on low-cost FPGA platform has been proposed for wearable-end solutions.

2.3. Advantages and recent challenges of computing models implementation

Hybrid FPGAs like Xilinx's Zynq, Intel's Cyclone, etc., have an advantage in these regards because of their freely programmable, highly configurable logic blocks (CLBs), look-up tables (LUTs), and DSP architecture blocks that harden functional units inside fabric. They are relatively cheap, consumed less power, and have support of hardware-software based design which provides a better way to utilize both hardware and software aspects of design hence providing better flexibility. The configuration stored in temporary memory cells inside FPGA can be reprogrammed many times, and hence it can be used in terms of specialized accelerators over specific applications (improving efficiency) or prototyping of new design on hardware. There are some challenges that FPGAs implementation of network hardware outperforms their software counterpart. However, there are AI/ML development environment provided by different FPGA vendors but they generally targeted for high-end devices [15]. Also, methods are not friendly as on hardware against their software counterparts [16]. To overcome the challenges, an efficient, faster, and hardware friendly implementation is proposed for low cost device. This low cost computing device can be used for real-time applications.

2.4. Novel contribution in the paper

This paper discusses the methodology and presents an efficient and hardware-friendly implementation of synthesizable DNN code and integrate it with Zynq and Microblaze with other required peripherals to develop a complete system-design based solution. In this case low-cost hybrid FPGA platforms are the targeted hardware device, but the approach and solution are not limited and can be extended

to any FPGA independent of their vendor, and the DNN code can be synthesized and implemented with any logic synthesizer tool. The contribution in present work is judiciary explored to represent the effort in term of advancement. The novel contribution is presented as:

1) Implementation of cost effective IP core using DNN model has been done for real-time applications.

2) DNN features such as Hidden layers, activation function and line buffers are explored to form an activated DNN model.

3) Utilization of Microblaze, BRAM for development has been enlighten for Smart IP core.

4) Softcore, easy to implement edge-IP based solution has been proposed using deep learning features.

5) AI based approach has been involved for smart utilization of AXI-DMA controller for streaming application.

As per the presented contribution, the DNN model is proposed with utilization of Microblaze hardware for reliable efficiency and other performance parameters. The low power hardware is needed with portable device. These presented features demonstrate the fulfillment of required benchmarks.

3. A DESCRIPTION OF PROPOSED INEURALFPGA IMPLEMENTATION

The core architecture of neural network despite recent developments have been the same. All input pixels or flattened features are provided to the next layer of neurons (less in size as of preceding layers), and after the operation, the output is provided as an input to the next layers. The following section discusses the architecture of neuralnet and number system which is used.

3.1 Representation of number

There are numerous ways but majorly two ways to store real numbers in modern computing i.e., floating-point and fixed-point representation. Recently, posits have also been new on the list, which solves the major problem of floating point with an ability to toggle between dynamic range and precision given out by a number or bits and also removes numerous ways to represent NaN (not a number), Infinity, etc. [17]. The number representation carries a significant impact on performance and resource utilization of hardware used. Floating points are precise more accurate. But on hardware, the calculations are slower than fixed-point representation due to its complexity in operation such as controlling both mantissa and exponent values for various operation. But, it has been analyzed shown in context of FPGAs, the fixed-point approach results in better performance [18].

Algorithm 1. Fixed-point floating notation to binary

1. Given:
   num: floating point number to convert
   IDW: input data width
   fracbits: number of fraction bits
2. scaled_num = int(num × 2^fracBits)
3. **if** scaled_num > 0
4. **if** scaled_num = 0 then, Return '0'
5. **else**, Return 'scaled_num'
6. **elseif** scaled_num < 0
7. Return '(2^IDW) + scaled_num'

Hence, in this case floating fixed-point data representation approach is used, which makes design more simple, resource friendly and robust. If given is *IDW* (input data width) bits in size, then the addition and multiplication at max takes around *2*IDW-1* bits. Also, since the output of addition is presented as input to next layer, the final result after operation is then truncated to match inputs characteristics. Algorithm 1 shows the mechanism to convert fixed-point notation to binary (2's complement) value.

3.2 Mechanism behind designing of neuron

Each and Every neuron has its own set of configurations (i.e. weights, bias) stored in memory, and parameters (activation function, etc.) are declared from the instantiating module above it. All the weights and basis of the pre-trained network are stored in a Weight memory, as shown in Figure 3. The depth of memory for each neuron is indicated by the number of inputs (or no. of neurons in the preceding layer) and in each memory location. The corresponding weight values are stored. Irrespective of the number of neurons in preceding layer, there will be a single input interface for accepting data. By this method, the efficiency, clock performance, and scalability of design is increased for a large network while compromising with the latency of system.

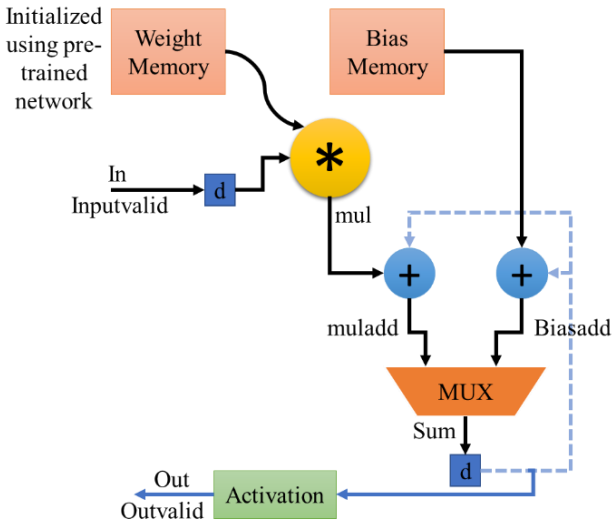

Figure 3. Block representation of proposed DNN model implementation for single neuron

The Single Neuron architectural implementation is shown in Figure 3. The memory is instantiated as ROM and initialized with pre-trained weights and bias. Since, it has one cycle of read latency, the *memout* is multiplied by delayed version of input i.e. *inputd*. After multiplication, the value is added with its previous accumulated value i.e. sum. Depending upon if multiplication is valid i.e. *mulvalid* the operation will continue for all Inputs coming inside neuron. On the falling edge of *mulvalid* or last cycle of valid multiplication operation, the total sum is added with bias value. The $ signed task of Verilog is used to preserve 2's complement representation for multiplication and addition. At last, the final sum is given to activation function. The type of activation function configures a look-up-table (for non-linear function), such as sigmoid, softsign, tanh, etc. The depth of LUT for these can be configured as per required constraints. A circuit-based function (for linear function) such as relu, linear, etc. is implemented. The type of activation function and its depth has a direct impact on resource utilization, performance and accuracy of the network. After the completion of operation, the

outvalid is asserted and out is produced by the neuron.

Algorithm 2. Designing of neuron

```
Given:
    N: no. of weights
    IDW: input data width
    WS: weights Int size
    AF: activation function
    DS: depth size of activation function
    biasFile, weightFile
    Main Neuron:
if rst | outvalid then
memRdAddr,sum = 0
else if inputvalid then
memRdAddr += 1
end if
READ BIAS FROM BIASFILE
bias⇐{biasReg[0][IDW −1 : 0],{IDW{1′b0}}} ▷ adjusting bias according to
fixed point representation
INSTANTIATE WEIGHT MEMORY
mul ⇐$signed(inputd)×$signed(memout)
sumadd = mul+sum
biasadd = bias+sum
if memRdAddr==N & pmultvalid then ▷ If previous cycle was vaild last
if! bias[msb] &! sum[msb] & biasadd[msb] then ▷ If positive overflow with
bias
sum={1'b0, else(1)} ▷ positive saturate
else if bias[msb] & sum[msb] & !biasadd[msb] then ▷ If negative overflow
with bias
sum={1'b1, else(0)} ▷ negative saturate
else
sum⇐biasadd
end if
else if multvalid then ▷ if multiplication is valid
Overflow with sum and mult
else
sum⇐muladd
end if
INSTANTIATE ACTIVATION FUNCTION
```

Algorithm 3. Activation function

```
GIVEN:
    IDW: input data width
    WS: weights Int size
    DS: depth size of activation function
Implementing Relu: ▷ linear activation function follows the same
if signed(input) >= 0 then
if input[2 ∗ IDW −1− :WS+1] then ▷ if positive overflow in integer part
out⇐ {1'b0, else(1)} ▷ positive saturate
else
out ⇐input[2 ∗ IDW −1−WS− : IDW]
end if
else
out ⇐0
end if
Implementing Sigmoid: ▷ Non-linear activation function (softsign, tanh) follows the same
READ SIGMOID LUT MEMORY ▷ look up table
if $signed(input) >= 0 then
out ⇐input +(1 << (DS−1)) ▷ provide offset
else
out ⇐input −(1 << (DS−1))
end if
```

The Algorithm 2 shows the Hardware implementation of a Single neuron. Initially the weight memory read address register is reset. If *reset* is applied or the operation in finished (indicated by *outvalid*) else, it will increment by 1. Also, the bias values are read from BIASFILE and padded with 0 at the back to transform it according to Fixed-point representation format followed by other operands like *sum* and *mul*. If the output of arithmetic operations in neuron Overflows or Underflow, it is saturated accordingly.

The Algorithm 3 shows the Hardware implementation of a "relu" and "sigmoid" Activation Function but the same approach can be used to design other functions. The non-linear function requires LUTs for mapping input to output with required offset and also with various depths (or size) it. The advantage of linear (vs non-linear) function in terms of hardware is the memory size requirement. The more depth of LUTs, more would be accuracy and also the memory size requirement. While linear function just requires rather simple computation which is easy inferred by target hardware platform or FPGAs. For Linear activation function like "relu" input follows output until positive or negative saturates occurs (due to datawidth constraints), on the other hand for non-linear activation functions we need to dump a sampled values and read these value as LUT look-up table.

3.3 Designing of hidden layers of proposed DNN model

The no. of neurons in each layer specified are instantiated and stitched together, forming path for data movement. Since in our implementation, only a single input is provided to neuron at a time. All output values generated by the neurons (*outvalid* is true) between layers. It is stored in shift register and shifted out at every clock cycle as shown in Figure 4. Since inputs are provided at same time, the output and *outvalid* is also produced at same time and a small FSM decides the state of operation. At last the outputs from the final layer is given to hardmax activation function which calculates the max. value from output and generates "*intr*" (interrupt) to the PS indicating completion of task.

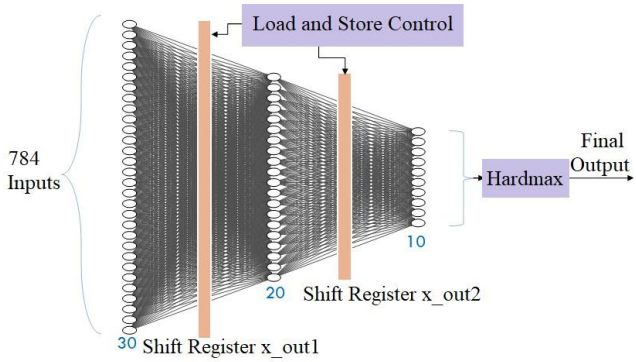

Figure 4. Description of layers' design of ineuralnet

3.4 Architectural description of proposed neuralnet

The complete system architecture implementation is shown in Figure 5 using Zynq platform. The neural network is packaged as an IP core (called neural net) and it is integrated with Zynq PS via AXI-Lite interface and connected to its GP0(General purpose) port. The interface is used for writing the soft-reset to the IP, reading/displaying out the weight and bias value of particular neuron, the status of network operation and the final ouput (i.e. after harmax layer) value from network. The AXILite interface is generated with help of

Vivado's IP's generator tool and the neural net design is not restricted to Zynq but any platform with AXI-Lite support (i.e. Microblaze, etc.) is required. The neural net IP is connected with AXI DMA controller via AXI4 stream point-point interface (axis port) and DMA is connected to Zynq HP0.

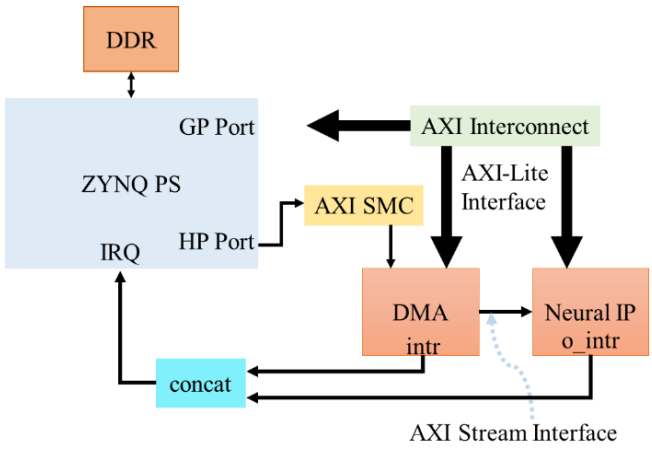

Figure 5. An FPGA representation of implementation paradigm

(High Performance) port via AXI-Lite (i.e. axi) port, in which turn is connected to the external memory on FPGA. This allows the inputs to neural net IP to be directly streamed from external memory via DMA. The AXI-stream provides 784 inputs to IP one after the another. s axis data is given as input to first layer. The s axis valid as inputvalid and s axis ready are driven to constant 1'b1. The Interrupt signals from both DMA and neural net is concatenated and connected to Zynq PS though IRQ (interupt request handler) port. After receiving the interrupt, the PS can read out the output result from base IP+0x4(offset) address. The control register is attached to baseIP+0x8(offset) address, which is used for issuing soft-reset and the status register attached to baseIP+0xC(offset) address gives status of network (i.e. completion of operation) and it is attached to" intr" signal of IP (i.e. clears the status when valid output is generated).

4. CONVOLUTION ARCHITECTURE OF PROPOSED INEURALNET

Images are stored as 2D or 3D array format depending upon the channel required to represent them. Graycale have one channel and pixel size is 1 byte in size while RGB image contain 3 channels and pixel size is 24-bit (considering each channel of 1byte). In point processing, the output of pixel depends upon value of pixel at that corresponding position and the transformation being performed on it. While in Neighborhood processing, the output depends upon corresponding pixel too. The Convolution performed over image falls under this category in which the kernel (or filter) is shifted over image (the shift amount is known as Stride) and MAC operation implemented in fully streaming architecture, rather it is stored over a memory location (generally BRAM). Image pixels are streamed as complete row at a time. Since the required pixels for operation/processing are not consecutive. It is also shown in Figure 6. An output pixel requires different position of input image. It needs small buffer (or called Line Buffers) for operation. As, it is almost impractical to buffer the entire image. If the image size is 512×512 bits and considering greyscale it requires 512×512×1=262144 bytes of memory location in FPGA, which is just for a single layer. It is not possible in small platforms using BRAMS (BRs), LUTs and FF, even if, it is required to store on external memory. The most of useful space, resource and operation time of hardware are lost in storage and accessing data rather than utilization it for computation. The idea with line buffers is to store only minimum required buffers depending upon Y-dimension of kernel (such as 3×3 kernel need at least 3 line buffers). In contrast of storing complete image, it now required 512×3=1536 bytes of memory to compute convolution operation at a time. The Algorithm 4 shows implementation of line buffer which simply storing a row of inputs and outputs a block of array value depends upon Kernel's X-dimension without any latency. The Algorithm 5 generates control logic for filling, selecting buffers and outputs to MAC module.

Algorithm 4 describes the designing of control logic for convolution neural network. Control logic instantiate and contain FSM for line buffers.

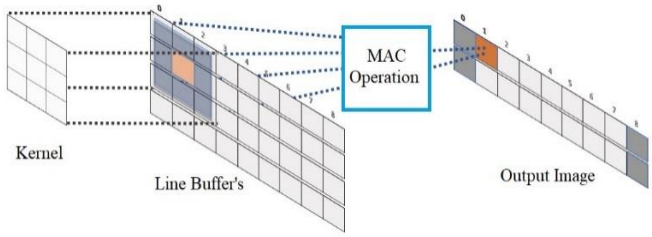

Figure 6. Architectural representation of convolution layers

Algorithm 4. Designing of line buffer

1. Given:
   IIS_X, IIS_Y: input image size x and y dimension *KS_X, KS_Y*: kernel size x and y dimension
   nDS: depth size of particular layer
   S: no. of stride
   nLB: no. of line buffer

Single Line Buffer:

2 if rst(readvalid & (rdPntr >= *IIS_X* - (*KS_X*-1) - S)) (inputvalid & (wrPntr == *IIS_X* - 1)) then ▷ either reset or last read,write pointer
3 wrPntr, rdPntr = 0 ▷ reset all to 0
4 else if inputvalid then
5 wrPntr += 1 ▷ increase write pointer
6 else if rdPntr > *IIS_X* - (*KS_X* - 1) - S then
7 rdPntr += S ▷ increase read pointer
8 end if
9 if inputvalid then
10 linebuffer[wrPntr] = input data ▷ storing data
11 end if

pre-fetching Output Data

12 for (i = 0; i (*KS_X*-1); i = i + 1) begin
13 assign o_data [i nDS+: nDS] = linebuffer[rdPntr + ((*KS_X*-1)-i)]
14 end

Algorithm 5. Designing of control logic for convolution

GIVEN:
nDS: depth size of particular layer S: no. of stride
tPC: total pixel counter iPC: input pixel counter
rC: output pixel read counter
cWLB: select which line buffer to write lBDV: selects line buffer write enable cRLB: select which line buffer for read rLB: selects line buffer read enable

Implementation:
1) Counting total pixel value for complete block

```
if rst then
        tPC = 0
else if (inputvalid & output ready) & !rdlinebuffer then
        tPC += 1
else if !inputvalid & rdlinebuffer then
        tPC -= (S*S)
else if (inputvalid & output ready) & rdlinebuffer then
        tPC -= (S*S) - 1
end if
```

2) Counts the number of valid input data and reset it at the end of line buffer

```
if rst || ((inputvalid & output ready) & iPC== IIS_X) then
        iPC = 0
else if (inputvalid & output ready) then
        iPC += 1
end if
```

3) Selects which line buffer to write on

```
if rst then
        cWLB = 0
else if ((iPC==IIS_X) & (inputvalid & output ready) then
        cWLB = (cWLB + 1) % nLB▷ cyclic rotation
end if
lBDV = numLineBuffer 1'b0
lBDV [cWLB] = (inputvalid & output ready)
```

4) Counts the number of valid output data of a row in line buffer (read count)

```
if rst (rLB & (rC == (IIS_X- KS_X) / S)) then
        rC = 0
else if (rLB) then
        rC += 1
end if
```

5) Selects which line buffer to read from (current read line buffer)

```
if rst then 0
        cRLB = 0
else if ((rC == (IIS_X - KS_X) / S) & (rLB) then
        cRLB = (cRLB + 1) % nLB  ▷ cyclic rotation
end if
```

6) Main FSM for control logic

```
if rst then
        rdState, rLB, Empty = 0
else
        case(rdState) 0:
        Empty = 0            ▷ Not ready to take inputs
        if (tPC    ((IIS X*KS Y) - 1) & (inputvalid & out put
ready)) then
        rLB, rdState = 1
        end if
    case(rdState)  1 :
        if (rC == ((IIS X*KS X)/S)) then
        if (tPC    ((IIS X*KS Y) - 1) & (inputvalid & out put
ready)) then
        rLB, rdState = 1
        else
        rLB, rdState = 0
        end if
        end if
        Empty = 1            ▷ buffers are empty, ready to take
inputs
        endcase
end if
```

5. HARDWARE-SOFTWARE CODESIGN APPROACH

The proposed hardware-software co-design approach leverages the strengths of both hardware (FPGA) and software components to optimize the performance of the DNN model while ensuring flexibility and efficient resource utilization. In this section, we provide a detailed breakdown of the division of tasks between the hardware and software components, illustrating how each part contributes to the overall system.

5.1 Hardware responsibilities (FPGA implementation)

The primary responsibility of the hardware component in our design lies in handling the computationally intensive tasks of the DNN, such as matrix multiplications, activation functions, and weight calculations. FPGAs, with their highly parallel architecture, are well-suited for such tasks, enabling real-time performance and low power consumption.

Key responsibilities of the FPGA in our design include:

Neuron and layer computations: The FPGA is responsible for executing the core computations within each neuron, including weighted sums and activation functions. As described in the architectural design of the neuron, these operations are performed using fixed-point arithmetic to ensure speed and efficiency while minimizing resource usage.

Weight storage and management: The FPGA stores pre-trained weights and biases in its memory blocks (e.g., BRAMs) and performs all necessary weight-fetching and bias addition during inference. This offloads these repetitive tasks from the software side, ensuring that the data is processed in parallel with minimal latency.

Parallel data processing: With the FPGA's ability to handle multiple inputs concurrently, the design implements parallelism across multiple layers of the network. Each neuron within a layer operates simultaneously, leading to significant speed improvements, especially in large-scale networks. This parallel processing is handled entirely in hardware.

Activation functions: The implementation of activation functions, such as relu and sigmoid, is performed using look-up tables (LUTs) on the FPGA. These functions are calculated in hardware, reducing the computational load on the software and ensuring that nonlinear transformations are applied efficiently.

Control of data flow: The FPGA manages the flow of data between layers, ensuring that each neuron receives inputs at the appropriate time and that outputs are transmitted between layers with minimal delays. This is critical for ensuring high throughput in real-time applications.

5.2 Software responsibilities (processor/microcontroller)

The software component, running on a general-purpose processor (e.g., ARM core in the Zynq SoC or an external microcontroller), is primarily responsible for high-level control, initialization, and peripheral management. Unlike hardware, software is more flexible and easier to modify, making it well-suited for tasks that require configuration or interaction with external systems.

Key responsibilities of the software in our co-design approach include:

Model initialization and configuration: Before inference begins, the software initializes the FPGA by loading pre-trained model parameters (weights, biases) into the memory. It also configures the FPGA by setting parameters such as input size, precision level, and control registers via the AXI-Lite

interface.

Data preprocessing: In some applications, raw data (e.g., images or sensor data) may require preprocessing, such as normalization or transformation, before being fed into the neural network. These preprocessing tasks are performed in software to reduce the complexity of the hardware design and to allow for easy modification depending on the application.

Control and status monitoring: The software manages high-level control of the system, issuing commands such as reset, start, and stop signals to the FPGA. It also monitors the status of the FPGA during inference, reading status registers to determine when computations are complete or if errors occur.

Data input and output management: The software is responsible for managing data inputs and outputs to and from the FPGA. Inputs (such as sensor data) are fed into the FPGA via the AXI-Stream interface, and the processed outputs (such as classification results) are retrieved from the FPGA and displayed or sent to other systems for further action.

Peripheral interaction: For systems that require interaction with external devices (e.g., sensors, cameras, or user interfaces), the software handles communication with these peripherals. This allows the FPGA to focus exclusively on the DNN computations, ensuring optimal performance.

5.3 Benefits of the co-design approach

The hardware-software co-design allows for an efficient distribution of tasks that maximizes the strengths of both FPGA and software systems. The following benefits arise from this approach:

Parallel processing in hardware: The FPGA accelerates the DNN's core computations, leveraging its parallel architecture to perform operations like matrix multiplications, weight updates, and activation functions simultaneously. This dramatically improves speed compared to a software-only solution.

Flexibility in software: The software component provides flexibility, allowing for easy reconfiguration of the model parameters, input preprocessing, and control logic. This flexibility is especially useful in applications where the system needs to adapt to changing requirements or be easily updated.

Energy efficiency: Offloading the computationally expensive tasks to the FPGA significantly reduces the energy consumption of the system, as FPGAs are more power-efficient than general-purpose processors for these types of operations.

Modularity and scalability: The system can be easily scaled by modifying either the hardware (e.g., increasing the number of neurons in hardware) or the software (e.g., changing the input data format or preprocessing algorithms), making it adaptable for a wide range of applications.

Example of operations:

To illustrate the co-design approach in practice, consider an example where the DNN is used for real-time image classification:

Software initialization: The software initializes the system by loading the DNN model parameters (weights, biases) into the FPGA and setting up the input data format (e.g., resizing images).

Data input: The software reads an image from a camera sensor, preprocesses it (e.g., normalizes pixel values), and streams the preprocessed data into the FPGA via the AXI-Stream interface.

Hardware processing: The FPGA processes the image through the DNN layers, performing matrix multiplications and activation functions in parallel for each neuron in the network. Intermediate results are stored and passed to subsequent layers without software intervention.

Software monitoring: While the hardware processes the data, the software monitors the status of the FPGA, checking status registers to determine when the computation is complete.

Output handling: Once the DNN inference is complete, the FPGA sends the output (e.g., classification result) back to the software via the AXI-Lite interface. The software then handles post-processing or communicates the result to other systems for further action.

6. SIMULATION AND RESULTS OF THE PRESENT WORK

The presented neuron module is written in Verilog HDL and Python is used to parameterize the number of neuron in each layers and" neuralnet" IP to support any general type of fully connected neural network architecture. For evaluation and validation of neural network, the popular MNIST handwritten-digit dataset is used which contains over 30,000 for training and 10,000 images for testing. The weights and biases for network are generated by using Tensor Flow and python scripts. These values are used as pre-trained hardware values and loaded inside the memory. For sake of testing the design, all implementation follows a 4 layered fixed architecture, with 784 inputs in input layer, 1 hidden layer with 30 neurons, 1 hidden layer with 20 neurons and an output layer with 10 neurons. The output or final layer is attached to hard-max function which is analogues to software implementation of max-finder (or softmax) but implemented in hardware, which detect max value out of 10 neurons.

For training of MNIST dataset, the Tensor-Flow library is used with above archi- tecture but with different activation function. The software implementation after 20 epoch gives around 95.969% accuracy for the testing data set. For simulation and implementation of all designs, the Xilinx Vivado 2018.3 version is used and for hardware validation, EDGE-Zynq FPGAs with xc7z020clg484-1 SoC part. 512MB. external DDR3 memory is used to prototype the design. The testing is done in the default setting of Vivado without any specific optimization over fabric part of FPGAs (i.e. partial reconfiguration, timing, power, etc.). The simulation results of images and the graphical representation of parametric results through the implemented model are presented in Figure 7.

Tables 1 and 2 compare the resource utilization of DNN for different input data widths in terms of LUTs, flip-flops, BRAMs (BRs) and DSP slices using different activation functions. For non-linear activation functions having larger data size, the lookup table is used for mapping values to corresponding input values to BRAMs (BRs) which exponentially increases its utilization. Table 2 represents the similarities with few parameters change in proposed work compared to prior related work.

The Table 3 contains the comparision of results with prevoius implemneted models as discribed in references [11, 13].

In summary, the empirical data and comparative analysis presented in this section demonstrate that the proposed DNN model on FPGA offers several advantages over existing implementations.

Higher accuracy: Achieves near state-of-the-art accuracy for real-time applications with minimal loss in precision.

Lower power consumption: Significantly reduces power usage, making it ideal for power-sensitive environments like IoT and portable devices.

Faster processing: Outperforms existing implementations in terms of clock speed, ensuring real-time operation for critical applications.

Balanced resource utilization: While the model consumes more resources in certain areas, it achieves superior performance, offering a good balance between computational demands and FPGA resources.

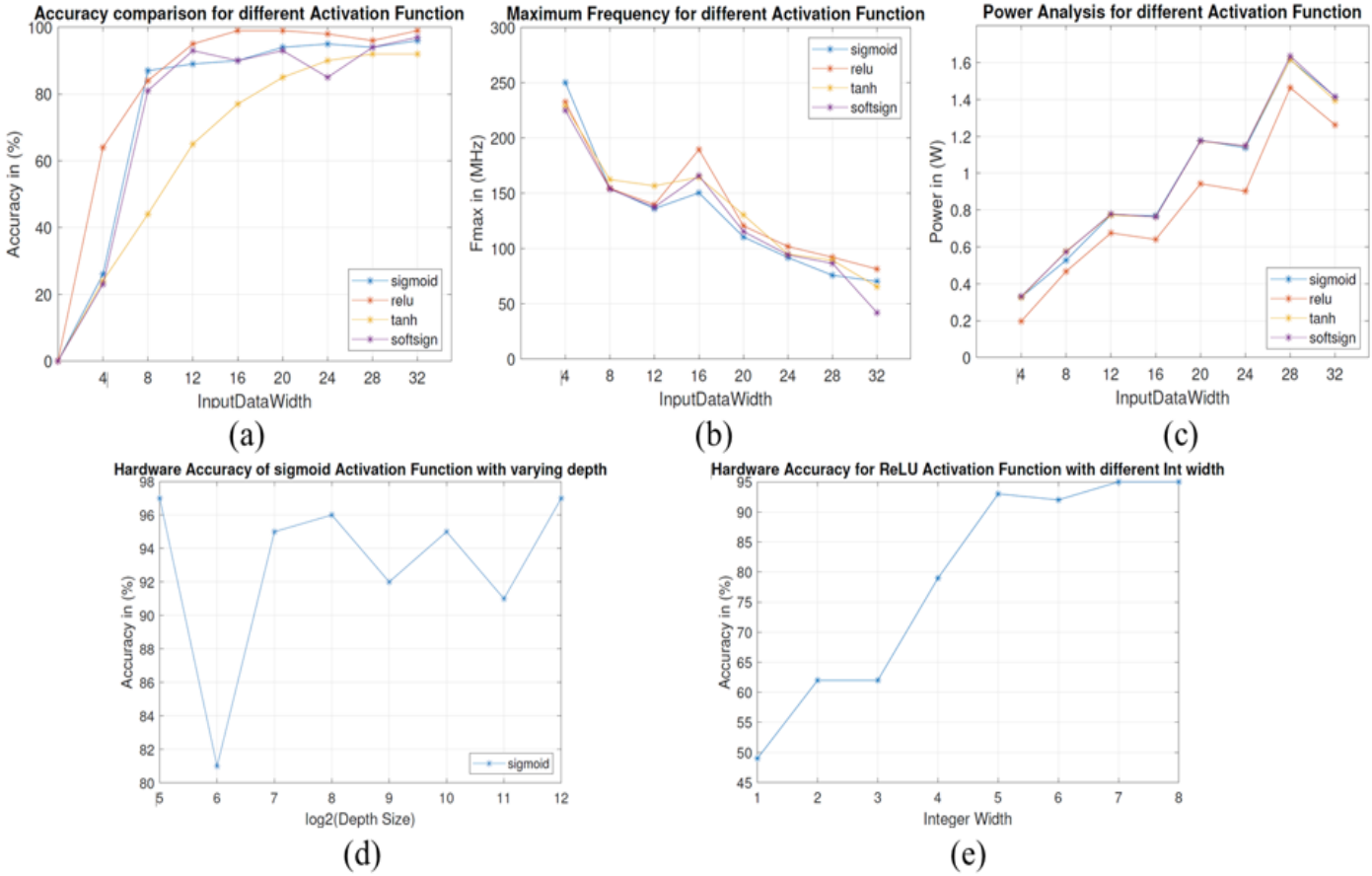

Figure 7. Results in terms of accuracy, clock performance and power analysis for different neural network implementation

Table 1. Analysis of resource utilization for various activation functions-I

Act	Tanh			Soft Sign			All
Depth	LUTs	FFs	BRs	LUTs	FFs	BRs	DSPs
4	4038	2309	15	4325	2330	15	0
8	7279	3873	30	7906	3904	30	0
12	4054	3394	30	4070	3394	30	120
16	4756	4096	30	4767	4075	30	120
20	12582	8300	60	12635	8310	60	60
24	9454	6490	60	94643	6480	60	120
28	17263	9108	61	17279	9034	61	220
32	19705	9863	70	19583	9863	70	220

Table 2. Analysis of resource utilization for various activation function-II

Act	Sigmoid			Relu			All
Depth	LUTs	FFs	BRs	LUTs	FFs	BRs	DSPs
4	3517	2288	15	3889	2521	0	0
8	7895	3884	30	8597	4373	15	0
12	4054	3405	30	5144	4053	15	120
16	4756	4107	30	6294	5013	15	120
20	12630	8299	60	13705	9513	30	60
24	9464	6480	60	10917	7915	30	120
28	17242	9012	61	19844	10885	30	220
32	19790	9916	70	22679	11864	40	220

Table 3. Comparison of neuralnet model with previous work

Metric	Proposed Model (Zynq-7000 SoC)	Existing Model 1 (Virtex-7)	Existing Model 2 (Ultra96-V2)
Accuracy	98%	91%	95%
Power Consumption	0.57 W	3.92 W	1.9 W
Max Frequency	195.407 MHz	100 MHz	150 MHz
LUTs Utilization	17,242	19,844	12,630
BRAM Utilization	60	60	70

7. APPLICABILITY OF THE MODEL FOR DIFFERENT APPLICATION SCENARIOS

The Implementation of a NerualNet on FPGA, as presented in this work, demonstrates configurablitiy and flexibilty to tweek the KPI in terms of accuracy, speed, and power efficiency. For different real-world applications, we can control parameters like, layer-depth, data type precision, activation function, etc. and have diferrent requirements based on their specific operational needs. Below are the few application scenarios, discussing how the model's parameters can be adjusted to optimize performance for each case.

Consumer electronics (IoT devices, wearables): In consumer electronics, particularly in wearable devices and IoT systems, there is a high demand for real-time processing, low power consumption, and moderate accuracy. For instance,

fitness trackers or smart home devices require models that can perform continuous data processing but with a focus on extending battery life. By configuring the model with reduced complexity (e.g., fewer layers or lower precision), it can further optimize power usage for such low-power environments, making it an excellent choice for portable consumer electronics.

Automotive industry (autonomous vehicles, driver assistance): Autonomous driving systems and advanced driver-assistance systems (ADAS) require real-time decision-making capabilities with high accuracy and reliability, as any delays could result in safety risks.

Smart cities (traffic management, public safety): In smart city applications, such as traffic management and public safety monitoring, systems must process data from a wide array of sensors in real-time while being energy-efficient, especially when deployed in large numbers.

8. CONCLUSION AND FUTURE DIRECTION

The proposed work discussed the implementation of" neuralnet", a DNN generator IP core targeting low-cost configurable FPGA based devices. Implementation results of" neuralnet" shows better performance, achieving accuracy close to software implementations with better throughput by an order of magnitude. Simulation is done using Xilinx Vivado 2018.3 and EDGE Zynq FPGA is used for prototyping the design. The proposed implemented DNN model with physical framework will be utilized as sophisticated computing model for real-time applications. The proposed work environment is compared in context of similarities with parametric changes, which represents an optimized solution as computing model in consumer system era. The proposed implementation has been analyzed and compared with existing FPGA model in terms of accuracy and other parameters. The standard data has been taken for prior validation of the DNN model implemented on FPGA. The work will be enhanced to implement for portable systems. Such kind of implemented system can be used in wearable devices for distinct applications. The optimization in terms of design, precision and portability of model is further required for low cost, high speed, easy to handle and accurate system formation. The modifications in terms of necessary parameters are future challenges of impact full design for future perspectives. More complicated and precise models are required to propose to enhance precision level of measuring systems. The proposed DNN model will be further tested by real time dataset. The proper sensing paradigm will be calibrated with required constraints and benchmarks for the data collection. The collected data will be tested further and accuracy will be analyzed with hardware performance. The particular objectives will be covered in future work.

REFERENCES

[1] Jain, P., Joshi, A.M., Mohanty, S.P. (2019). iGLU: An intelligent device for accurate noninvasive blood glucose-level monitoring in smart healthcare. IEEE Consumer Electronics Magazine, 9(1): 35-42. https://doi.org/10.1109/MCE.2019.2940855

[2] Joshi, A.M., Jain, P., Mohanty, S.P., Agrawal, N. (2020). iGLU 2.0: A new wearable for accurate non-invasive continuous serum glucose measurement in IoMT framework. IEEE Transactions on Consumer Electronics, 66(4): 327-335. https://doi.org/10.1109/TCE.2020.3011966

[3] Sundaravadivel, P., Kougianos, E., Mohanty, S.P., Ganapathiraju, M.K. (2017). Everything you wanted to know about smart health care: Evaluating the different technologies and components of the internet of things for better health. IEEE Consumer Electronics Magazine, 7(1): 18-28. https://doi.org/10.1109/MCE.2017.2755378

[4] Li, H., Fan, X., Jiao, L., Cao, W., Zhou, X., Wang, L. (2016). A high performance FPGA-based accelerator for large-scale convolutional neural networks. In 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, pp. 1-9. https://doi.org/10.1109/FPL.2016.7577308

[5] Rachakonda, L., Bapatla, A.K., Mohanty, S.P., Kougianos, E. (2020). SaYoPillow: Blockchain-integrated privacy-assured IoMT framework for stress management considering sleeping habits. IEEE Transactions on Consumer Electronics, 67(1): 20-29. https://doi.org/10.1109/TCE.2020.3043683

[6] Guo, K., Sui, L., Qiu, J., Yu, J., et al. (2017). Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(1): 35-47. https://doi.org/10.1109/TCAD.2017.2705069

[7] Mohammadi, M., Krishna, A., Nalesh, S., Nandy, S.K. (2017). A hardware architecture for radial basis function neural network classifier. IEEE Transactions on Parallel and Distributed Systems, 29(3): 481-495. https://doi.org/10.1109/TPDS.2017.2768366

[8] Wang, J., Lin, J., Wang, Z. (2017). Efficient hardware architectures for deep convolutional neural network. IEEE Transactions on Circuits and Systems I: Regular Papers, 65(6): 1941-1953. https://doi.org/10.1109/TCSI.2017.2767204

[9] Shawahna, A., Sait, S.M., El-Maleh, A. (2018). FPGA-based accelerators of deep learning networks for learning and classification: A review. IEEE Access, 7: 7823-7859. https://doi.org/10.1109/ACCESS.2018.2890150

[10] Lian, X., Liu, Z., Song, Z., Dai, J., Zhou, W., Ji, X. (2019). High-performance FPGA-based CNN accelerator with block-floating-point arithmetic. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 27(8): 1874-1885. https://doi.org/10.1109/TVLSI.2019.2913958

[11] Nguyen, D.T., Nguyen, T.N., Kim, H., Lee, H.J. (2019). A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 27(8): 1861-1873. https://doi.org/10.1109/TVLSI.2019.2905242

[12] Gilan, A.A., Emad, M., Alizadeh, B. (2019). FPGA-based implementation of a real-time object recognition system using convolutional neural network. IEEE Transactions on Circuits and Systems II: Express Briefs, 67(4): 755-759. https://doi.org/10.1109/TCSII.2019.2922372

[13] Abd El-Maksoud, A.J., Ebbed, M., Khalil, A.H., Mostafa, H. (2021). Power efficient design of high-performance convolutional neural networks hardware accelerator on FPGA: A case study with GoogLeNet. IEEE Access, 9: 151897-151911. https://doi.org/10.1109/ACCESS.2021.3126838

[14] Zhang, Z., Mahmud, M.P., Kouzani, A.Z. (2022). Fitnn: A low-resource fpga-based cnn accelerator for drones. IEEE Internet of Things Journal, 9(21): 21357-21369. https://doi.org/10.1109/JIOT.2022.3179016

[15] AMD. (2021). Block-by-Block Configurable Fast Fourier Transform Implementation on AI Engine (XAPP1356). https://docs.amd.com/r/en-US/xapp1356-fft-ai-engine.

[16] OpenVINO. (2024). Accelerate Generative AI. https://docs.openvino.ai/2025/_static/download/GenAI_Quick_Start_Guide.pdf.

[17] Jaiswal, M.K., So, H.K.H. (2019). PACoGen: A hardware posit arithmetic core generator. IEEE Access, 7: 74586-74601. https://doi.org/10.1109/ACCESS.2019.2920936

[18] Hettiarachchi, D.L.N., Davuluru, V.S.P., Balster, E.J. (2020). Integer vs.floating-point processing on modern FPGA technology. In 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, pp. 0606-0612. https://doi.org/10.1109/CCWC47524.2020.9031118