Received June 15, 2021, accepted June 26, 2021, date of publication July 5, 2021, date of current version July 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3094675

A Scalable System-on-Chip Acceleration for Deep Neural Networks

FAISAL SHEHZAD[1], MUHAMMAD RASHID[2], MOHAMMED H. SINKY[2], SAUD S. ALOTAIBI[3], (Member, IEEE), AND MUHAMMAD YOUSUF IRFAN ZIA[2]

[1]Integrated Digital Systems, University of Bremen, 28359 Bremen, Germany
[2]Computer Engineering Department, Umm Al-Qura University, Mecca 21955, Saudi Arabia
[3]Department of Information Systems, Umm Al-Qura University, Mecca 21955, Saudi Arabia

Corresponding author: Muhammad Rashid (mfelahi@uqu.edu.sa)

This work was supported in part by the King Abdulaziz City for Science and Technology (KACST) under Grant 14-ELE1049-10, and in part by the Science and Technology Unit (STU), Mecca, Saudi Arabia.

ABSTRACT The size of neural networks in deep learning techniques is increasing and varies significantly according to the requirements of real-life applications. The increasing network size and scalability requirements pose significant challenges for a high performance implementation of deep neural networks (DNN). Conventional implementations, such as graphical processing units and application specific integrated circuits, are either less efficient or less flexible. Consequently, this article presents a system-on-chip (SoC) solution for the acceleration of DNN, where an ARM processor controls the overall execution and off-loads computational intensive operations to a hardware accelerator. The system implementation is performed on a SoC development board. Experimental results show that the proposed system achieves a speed-up of 22.3, with a network architecture size of 64×64, in comparison with the native implementation on a dual core cortex ARM-A9 processor. In order to generalize the performance of complete system, a mathematical formula is presented which allows to compute the total execution time for any architecture size. The validation is performed by taking Epileptic Seizure Recognition as the target case study. Finally, the results of the proposed solution are compared with various state-of-the-art solutions in terms of execution time, scalability, and clock frequency.

INDEX TERMS Deep neural networks, system-on-chip, scalability, hardware accelerator, epileptic seizure recognition.

I. INTRODUCTION

Deep learning algorithms are getting increasingly popular for image classification [1], object detection [2] and data prediction [3] in numerous real-world applications. Consequently, it has dramatically increased the development speed of machine learning (ML) and artificial intelligence (AI) [4]. It has been shown in [5] that the architecture of deep learning algorithms is almost identical to the conventional artificial neural networks. However, the salient characteristic of deep learning techniques/algorithms is that they work with many hidden layer [6]. In other words, the basic concept behind all the deep learning applications is to use a multilayer neural network model for the extraction of high-level features. These high-level features combine various low-level abstractions to find some distributed data features [7]. One of the most widely used deep learning algorithm is the deep neural networks (DNN) [8].

Despite the increased popularity of DNN, there are certain challenges which may hinder its use for practical applications. These challenges include but not limited to: (a) the amount of data required to process, particularly during the training phase, is exponentially increasing, (b) the accuracy requirements in modern ML and AI applications are more demanding [1], [2], (c) the size of neural networks are becoming very large [5], [7] and (d) the network size for various real-life applications is heterogeneous, and therefore, requires a scalable solution [9], [10]. In order to address the aforementioned challenges, the acceleration of deep learning algorithms with some scalability features on a dedicated platform is critical.

A. RELATED WORK

Different solutions for the acceleration of DNN algorithms include Graphic Processing Unit (GPU) [11],

The associate editor coordinating the review of this manuscript and approving it for publication was Cheng Chin.

Application Specific Integrated Circuit (ASIC) [12], and Field-Programmable Gate Array (FPGA) [13]. The advantages of GPU-based solutions are programmability and comparatively low cost while providing the high computational power, which is typically required during the training phase of deep learning algorithms. However, the implementation of networks on GPUs is very slow and power hungry.

The ASIC-based solutions, on the other hand, provide lower power consumption with a moderate performance. As compared to GPU-based solutions, ASICs have relatively limited computing resources, and therefore, it is challenging to develop complex and massive network architectures. Furthermore, they do not provide flexibility and require a larger development cycle. In order to address the issues with conventional ASICs, various commercial alternatives are available [14]. A typical example of this trend is the Tensor Processing Unit [15], which is not designed for just one neural network model. Instead, it is a programmable complex instruction set computer for the execution of miscellaneous neural network models.

In the context of flexibility and programmability, FPGA-based solutions for deep learning applications are getting more importance as it allows designers to build their entire systems on a single chip. In addition to the reconfigurability and short development cycles, FPGA have numerous hard components such as on chip RAM (Random Access Memory), I/O (Input/Output) transceivers and digital signal processing units. These components are significantly helpful in designing full-scale systems [16]. Although the clock frequency of FPGA is less than GPU, the total number of required clock cycles in FPGA-based solutions are significantly less. It provides a competitive advantage in the low power consumption design and the real-time data processing. Furthermore, it shows a powerful processing ability in massive matrix operations and multiply-accumulation operations which are the main building blocks of deep neural networks [17].

A high performance solution for the acceleration of a large-scale DNN is proposed by Chen *et al.* [18]. The presented solution, termed as Dian-Nao, targets an energy efficient solution with a high throughput and reduced area. The design technique in [18] is mainly based on the optimization of memory accesses in different layers. Although the challenges like scalability and limited memory resources are addressed, the Dian-Nao is not implemented using a reconfigurable hardware like FPGA. Therefore, it cannot adapt to different application demands.

To provide adaptability, an FPGA-based solution is presented in [19] to accelerate the restricted boltzmann machine (RBM). An RBM is a stochastic neural network architecture which is used to model the analytical behaviour of a particular data set with a provision of patterns distribution. The network in [19] builds an internal model which is capable of recognizing the new data from the same distribution. Moreover, the authors in [19] have created some dedicated hardware processing cores for the architecture size up to 128 × 128. Another FPGA-based accelerator for RBM is presented in [20]. The work in [20] advocates the use of multiple RBM processing modules in parallel. Each module is responsible for a relatively lower number of nodes.

The work in [9] presents a deep learning accelerator unit (DLAU) which is a scalable accelerator architecture for large-scale DNN using FPGA. The DLAU uses three pipelined processing units for the improvement of throughput and utilizes a tile technique to partition the processing data. Furthermore, it employs FIFO (First-In-First-Out) buffers at both input and output sides. The input FIFO buffer is used to receive the input sample data, transferred using the DMA (Direct Memory Access). Similarly, the output buffer is employed to transfer intermediate outputs. Moreover, the block RAMs ((Random Access Memory) are used to store weights of the neural network.

B. LIMITATIONS OF EXISTING PRACTICES

It can be observed from Section I-A that the current practices for the acceleration of DNN mainly focus on the implementation of a particular application and suffer with a variety of problems such as limited network size, flexibility, and difficult to use or no software interface. Another common problem is the choice of size for the neural network architectures. Therefore, there is a need for a scalable platform for the acceleration of DNN which can accommodate various architecture sizes. Although, the problems of heterogeneous network sizes and a scalable hardware architecture are addressed in [9], it mainly discusses the speed acceleration and no discussion is provided on the accuracy of obtained results. Furthermore, the total execution time and the power consumption issues (clock frequency) are required to be further optimized.

C. PROPOSED SOLUTION

This article proposes a scalable System-on-Chip (SoC) solution to speed-up the DNN algorithm using a co-design approach, as shown in Fig. 1. The design process starts by constructing (designing) a DNN in Python, followed by the training of the network for the given data. Subsequently, the accuracy of the trained DNN is tested. Once the training of DNN is accomplished and the final weights are obtained, the system implementation phase is started.

The system implementation phase is further sub-divided into two stages: The first stage employs the sequential execution of DNN (C Code) on the Hard Processing System (HPS) of the SoC development board. The second stage of system implementation involves the development of an hardware accelerator (VHDL language). The developed hardware accelerator is then integrated with the C-implementation such that the computationally intensive parts of the algorithm are executed using the developed hardware accelerator. The results of second stage (C+VHDL co-simulation) are then benchmarked with the native C-implementation (first stage).

The proposed system consists of two major modules: Processor System Module and Accelerator System Module. These modules are combined to make a complete system

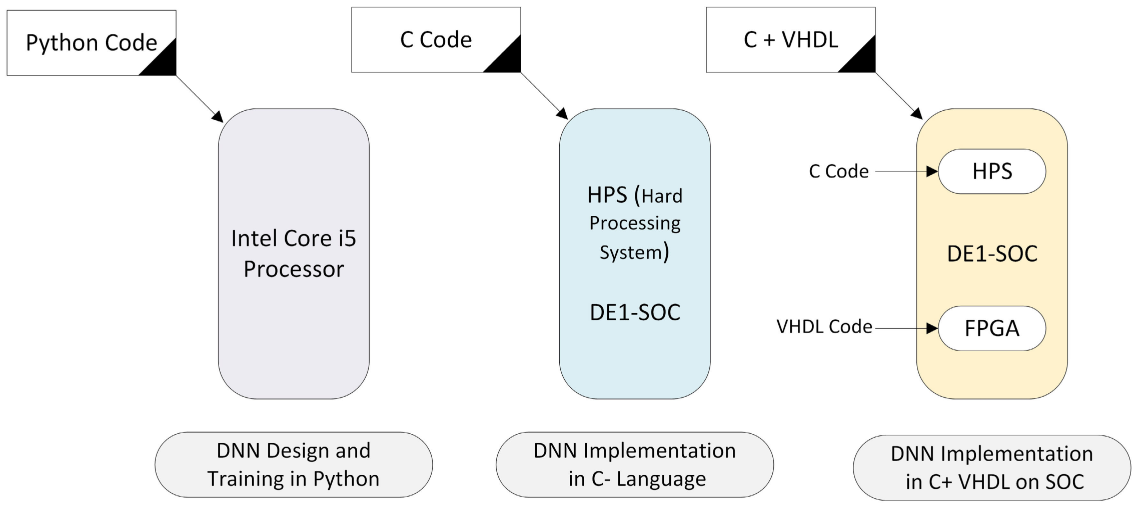

Python Code
C Code
C + VHDL
Intel Core i5 Processor
HPS (Hard Processing System)
DE1-SOC
C Code
HPS
DE1-SOC
VHDL Code
FPGA
DNN Design and Training in Python
DNN Implementation in C- Language
DNN Implementation in C+ VHDL on SOC

FIGURE 1. Overview of system development.

called Hardware Accelerator for Deep Learning (HADL). The transfer of data from the Processor System Module to the Accelerator System Module is performed through some on-chip block RAMs (BRAMs), using the DMA technique. The use of BRAMs, instead of registers arrays and FIFO buffers, along with the DMA technique makes the data transfer operations significantly fast. In addition to the BRAMs, some general purpose registers are also used for various configuration and control functionalities.

The DE1-SoC development board from Altera [16] is used as the target implementation platform as it possesses a powerful processing capacity and abundant hardware resources. Another reason for selecting the DE1-SoC development board is the availability of ARM core processor. The devices, which utilize both the embedded processor and FPGA, opens a significant amount of opportunities for designers. The SoC chip in DE1-SoC board integrates ARM cortex A-9 processor with Cyclone V FPGA through a high bandwidth interconnect backbone.

From the application point of view, the proposed system is scalable and can be used in multiple scenarios such as facial recognition, video surveillance, medical diagnosis and so on. However, in order to validate the proposed method, we have selected the Epileptic Seizure Recognition as our target case study. The researchers have proposed different methods for epileptic seizure detection using the features extracted with the help of electroencephalogram (EEG) signals [21]. The dataset for epileptic seizure detection, used in this article, is published and publicly available at [22].

D. CONTRIBUTIONS

The major contributions of this article are given as:

1) It provides a scalable implementation of DNN algorithm to achieve a performance improvement, as compared to stat-of-the-art FPGA-based implementations, in terms of computational time, power consumption and clock frequency.
2) The accuracy of obtained results is also discussed in this article, along with the computational time, which has not been discussed in existing solutions. Furthermore, it is also analysed that how the network architecture size affects the accuracy of obtained results.
3) While the DMA technique is also used in [9], the proposed SoC solution uses only BRAMs for the transfer of weights and input data samples. Consequently, the proposed solution is easy to implement and involves a lesser hardware complexity as compared to [9].

E. ORGANIZATION

The article is organized as follows: Section II provides the necessary background on the epilepsy detection problem and DNN. Section III presents the employed materials and methods by discussing the overall design methodology, tools, testing mechanism and various parameters of the target case study. Section IV elaborates the design, training and testing of DNN for the target case study.

Section V explains the design of proposed hardware accelerator system (HADL) by discussing its internal architecture. Section VI compares the results of pure software implementation with the proposed HADL system. Additionally, the HADL results are benchmarked with various state-of-the-art techniques. Finally, Section VII concludes the article.

II. BACKGROUND AND MOTIVATION

The necessary background of the target application, along with the motivation for its acceleration through DNN using some dedicated hardware resources, is presented in Section II-A. Subsequently, a brief introduction of DNN, RBM and the training process of DNN are provided in Section II-B, II-C and Section II-D respectively.

A. CASE STUDY: EPILEPTIC SEIZURE RECOGNITION

Epilepsy is a significant chronic disease of brain which causes sensory loss, seizures and unbalanced gesture. According to the world health organization, about 50 million people in the world today are suffering with epilepsy [23]. The most effective technique, commonly used for the detection of epilepsy, is electroencephalogram (EEG) which records the electrical activities of neurons in the human brain [24]. In other words, the EEG is a clinical process that monitors the activity of the human brain while performing some cognitive task.

In order to detect epilepsy, the analysis of EEG signals is critical. Generally, the EEG readings are examined to detect and categorize different patterns into seizures and non-seizures. However, the visual examination of EEG readings needs considerable time and efforts. Moreover, it requires the services of an expert (neurologists). This time consuming analysis of patient's data (EEG readings) creates a heavy burden on neurologists. The aforementioned limitations have urged researchers to develop some automated systems for the assistance of neurologists so that they can classify the patient's data into epileptic and non-epileptic EEG brain signals [21]. In other words, the detection of epileptic and non-epileptic signals is considered as a classification problem.

An EEG signal contains a kind of hierarchy among the low-frequency features and high-frequency features. Therefore, deep learning techniques can be employed to encode a hierarchy of these features. Recently, the idea of employing neural networks instead of using the services of experts, for investigating the data of epileptic patient, has shown great results [25]. Nevertheless, the machine learning algorithms in DNN involve a lot of multiplications and addition operations. As a result, a large computational time is required. Therefore, some dedicated architectures such as GPUs [11], FPGAs [13] and ASICs [14] are required.

B. DEEP NEURAL NETWORKS

A DNN consists of different units (neurons), arranged in certain layers, as shown in Fig. 2. Each unit takes an input, applies some (often non-linear) function, and then passes the output to the next layer. Mathematically, the intermediate output can be described with Equation 1.

$$Output = Relu\left[\sum_{n=1}^{N}\left(inp_{(n)} * W_{(n)}\right)\right] \tag{1}$$

$$Relu(X) = max(0, X) \tag{2}$$

where, *N* is the total number of input nodes, *inp(n)* denotes the input node values and *W(n)* are the weights of the network. The DNN model in Fig. 2 is a generalized model which contains n input nodes while the sizes of hidden layers are $h1$ and $h2$. The output layer contains m number of nodes. For the epileptic detection application (case study in this article), we have selected these different parameters (such as the number of hidden layers, the number of input nodes, weights of the network etc.) of the network model according to the requirement of the problem. For example, in the epileptic data set [22], the size of one sample is 178, and therefore, we have selected 178 nodes in the input layer. For the hidden layers, two different sizes 32×32 and 64×64 are used separately to create two different architectures. It is important to note that the overall objective in the target case study is to classify the seizure or non-seizures from EEG data, therefore, this type of binary classification requires only one node in the output layer.

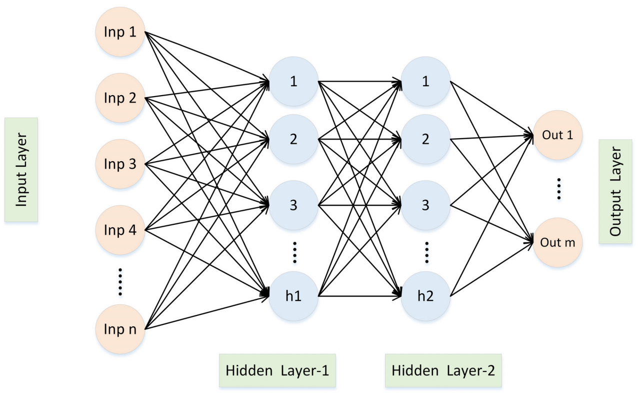


FIGURE 2. General architecture for a deep neural network with two hidden layers.

C. RESTRICTED BOLTZMANN MACHINE (RBM)

RBMs are commonly used to efficiently train each layer of a DNN. An RBM is a special class of Boltzmann machines (BM). The BM is a parallel computational model for the implementation of simulated annealing which is one of the frequently used heuristic search algorithms [26], [27]. It is a stochastic neural network which can learn internal representations to solve combinatorial optimization problems. Its massive parallelism, which leads to solve optimization problems without having a comprehensive knowledge of the target problem, is highly significant. In a typical BM architecture, the neurons are connected not only to the neurons in other layers but also to the neurons within the same layer. Essentially, every neuron is connected to every other neuron in the network. Consequently, it poses a serious challenge in training BM. In other words, an *Unrestricted Boltzmann Machine (UBM)* has very little practical significance.

The RBM model, on the other hand, eliminates the connection requirements between the neurons in the same layer. It facilitates in training the network. Practically speaking, an RBM is employed in several applications due to a relatively simpler training process as compared to an UBM architecture. Conventional training algorithms for RBMs are not efficient in terms of time due to their slow convergence rate [26]. Therefore, RBMs are generally trained with approximate training algorithms [27].

D. TRAINING OF DEEP NEURAL NETWORK

The objective of the training process is to find an optimal set of network parameters for solving the given task [8]. Furthermore, the network parameters must be initialized before starting the training process. Although, the random numbers are commonly selected for the initial values of DNN weights, however, some heuristics may result in faster adjustment of the parameters towards the optimal values [7]. Subsequently,

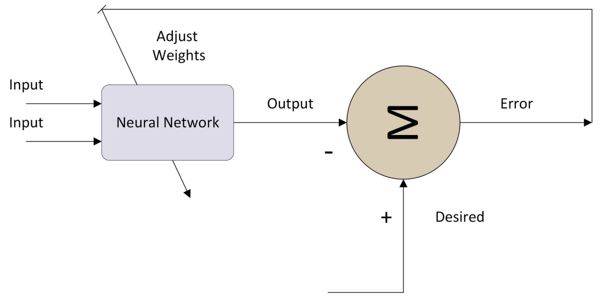


FIGURE 3. **Training of a deep neural network.**

the training data is fed to carry out the learning on the training set through the network. It is an iterative process, where the outputs, produced against each input from the training set, are compared with the reference or true values and the network parameters are adjusted accordingly [5]. The network is considered to be trained after achieving the target performance on the training data.

The training of DNN involves the recursive use of feed-forward and backward propagations, as shown in Fig. 3. In each iteration, the feed-forward computation is used to predict the output. The predicted output is then compared with the expected output to calculate the corresponding difference. On the other hand, the backward propagation is used to propagate the errors through all the layers. In this article, 11500 samples are used for the training such that the size of each samples is 178.

III. MATERIALS AND METHODS

The previous sections of this article set the stage for a scalable DNN hardware acceleration methodology. This section describes the employed materials and methods. First, the overall system design hierarchy and its testing mechanism are presented in Section III-A and Section III-B respectively. Then, an overview of different tools and software, used in different phases of the design, is provided in Section III-C. Finally, the various parameters of the target case study are presented in Section III-D.

A. SYSTEM DESIGN HIERARCHY

The design process starts by constructing (designing) a DNN in Python, followed by the training of the network for the given data (epilepsy data in our case). Subsequently, the accuracy of the trained DNN is tested. Once the training of DNN is accomplished and the final weights are obtained, the system implementation phase is started. The proposed system consists of two major modules: Processor System Module (Section III-A1) and Accelerator System Module (Section III-A2). These two modules are combined to make a complete system (HADL). The proposed system employs a hardware software co-design technique to execute the DNN application for a scalable architecture size.

1) PROCESSOR SYSTEM MODULE

It consists of a Hard Processing System (ARM A-9 Processor)), BRAMs, DMAs and GPIO (General Purpose

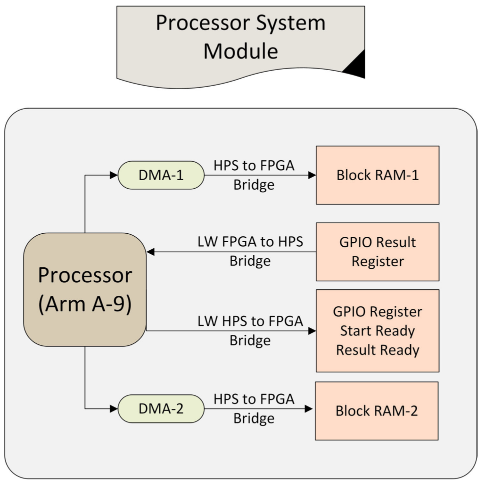


FIGURE 4. **Components of processor system module.**

Input/Output) registers, as shown in Fig. 4. A C-program is executed on the processor to perform less computational intensive tasks (such as reading test samples and weights data). Furthermore, a virtual mapping of all the components is also performed. Moreover, the DMAs are configured which are used to shift the data towards FPGA for fast processing. Finally, the result from DNN model is stored into a GPIO register, which is further used for computing the overall accuracy.

2) ACCELERATOR SYSTEM MODULE

It consists of four major blocks: Read-Block-RAMs (RBR), Tiled Matrix Multiplication Unit (TMMU), Rectified Linear Unit (RELU) and Sigmoid component. The data processing in the accelerator system module starts when it gets permission from the Processor System Module. Once the permission is received, the RBR component in the accelerator system module reads the data from FPGA-on-Chip RAMs. Subsequently, this data is assigned to TMMU module as its input. The TMMU module multiplies all the tiled inputs with their respective weights. Moreover, all the intermediate results are added to generate the final result for the RELU component. The RELU component is an activation function which approximates the output between 0 and X i.e max(0,X). If this activation function is not applied, the output is simply a linear function which is easy to solve and has low complexity. In other words, the system has less power to learn complex functional mappings from the data. The aforementioned three steps are used iteratively to calculate the hidden layer's node values. Once all the hidden layer nodes are calculated, a sigmoid approximation is applied at the output layer using Sigmoid component to calculate the final output of DNN.

B. TESTING MECHANISM

In order to test the overall functionality of the proposed system, designed and implemented on DE1-SoC board, it is important to understand the sequence of entire operations. Therefore, a system flow-diagram is presented in Fig. 5 to illustrate the working mechanism of the HADL system.

The system test mechanism is executed on a dual core Arm cortex A-9 Processor of DE1-SoC board. The first major step is the memory mapping of all the IP (Intellectual Property) components for the communication of Processor. Furthermore, FPGA-to-HPS and HPS-to-FPGA bridges are dedicated to perform communication between Processor and other custom IP components. Subsequently, the test samples and weight data are read and transferred into FPGA-on-Chip RAM-1 and RAM-2 using DMA-1 and DMA-2 respectively. It is followed by a for-loop to test the N number of samples. At the same time, a timer is started in order to calculate the total time for predicting all the test samples outputs.

After transferring the data, it writes 1 onto the start-ready GPIO register. This start ready signal is used to inform the Accelerator System that the data has been written to BRAMs. In response, the accelerator starts its processing. Next, the Accelerator System Module performs its functions to calculate the final output and write 1 at result ready signal to update the Processor. Subsequently, the final result on GPIO register is read by the Processor.

The For-loop (for testing the samples) is incremented by one and then the next input sample is shifted towards FPGA-on-Chip-RAM-1 (component of Processor System Module) to predict the 2nd sample output. This step continues until for-loop is completed and all the samples have been predicted. The timer is stopped after the completion of for-loop. Consequently, the total test execution time is calculated and printed on the screen. At the end of the C-program, there is a function to calculate the accuracy of the entire system. This function compares the predicted outputs with the original labels of test samples and calculates the total accuracy percentage.

C. OVERVIEW OF DESIGN TOOLS

1) DESIGN AND TRAINING IN PYTHON

The anaconda platform [28], which is an open-source distribution for scientific computing and machine learning, has been used for constructing, training and testing the DNN python model. The Anaconda version 2019 (Linux version) provides 32-bit and 64 bit installers for the Python 2.7 and 3.7 respectively. As our target hardware platform (DE1-SoC development board) is a 32-bit architecture, the 32-bit installer for Python-3.7 has been used. Particularly, the Spyder development environment in anaconda platform has been employed to facilitate the design process.

2) IMPLEMENTATION IN C LANGUAGE

There are two ways to compile and execute a Linux program (execution of C-programs) for the Arm Processor of the DE1-SoC hardware. The first method is to write a code on the command line interface of Linux running on the SoC board, whereas the second method gives the option to write and compile the code at host computer and then transfers the executable file in DE1-SoC board. In this article, we have adopted the first approach as it is more simple and easy to implement.

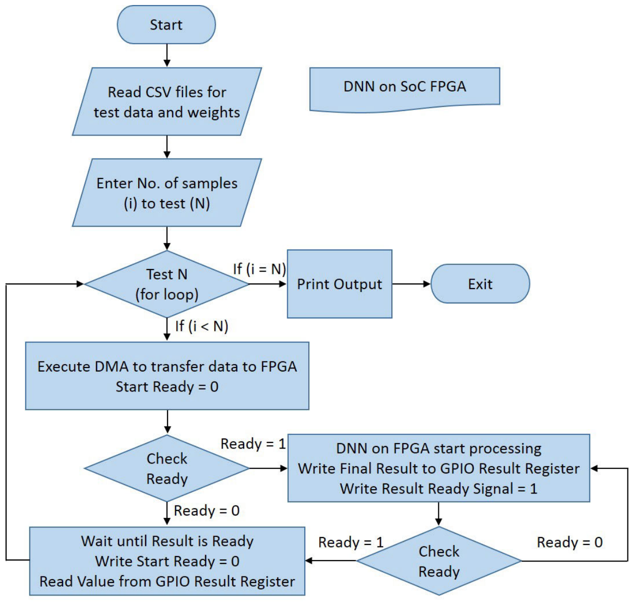

Start
Read CSV files for test data and weights
DNN on SoC FPGA
Enter No. of samples (i) to test (N)
Test N (for loop)
If (i = N)
Print Output
Exit
If (i < N)
Execute DMA to transfer data to FPGA Start Ready = 0
Check Ready
Ready = 1
DNN on FPGA start processing Write Final Result to GPIO Result Register Write Result Ready Signal = 1
Ready = 0
Wait until Result is Ready Write Start Ready = 0 Read Value from GPIO Result Register
Ready = 1
Check Ready
Ready = 0

FIGURE 5. Flow chart for testing.

3) IMPLEMENTATION IN C AND VHDL ON SoC

The Processor System Module of the proposed HADL has been realized using platform designer tool (formerly called Q-sys) [29] which is the next-generation system integration tool in the Intel Quartus Prime software [30]. The idea of platform designer is to generate the interconnect logic in order to connect different IP components, functions and subsystems. In other words, it allows designers to select the desired IP components (such as DMA, On-Chip-RAM, Processor etc) from its library and edit the corresponding specifications as per particular requirements. Finally, the Quartus prime lite edition [30] is used to synthesize the overall design and perform timing analysis.

D. DESIGN PARAMETERS

The epileptic seizure dataset has been selected as an application for this article [22]. The selected dataset consists of 5 folders with 100 files each. Moreover, each file represents the data of one person for his brain activity for 23.6 seconds. The aforementioned data is sampled into 4097 data points. These 4097 points are divided into 23 chunks such that each chunk contains 178 points presenting the brain activity for 1 second. Consequently, the target dataset contains 11500 (23×500) pieces of information (or rows). In addition to various data points, it contains 5 different classes as labels. Class A and Class B have an EEG recording from healthy persons with their eyes open and close respectively. Similarly, Class C and Class D contain the recording of epileptic patients without seizure. Finally, Class E corresponds to the data of epileptic patient during the seizure.

The two important steps, that must be taken into consideration before training, are: (1) splitting the data into training set and test set and (2) feature scaling. It is important to note that 80% of the data set is used for training while the remaining 20% data (2300 samples) is used for the testing. Similarly, feature scaling or standardization is a data pre-processing method which is applied to features or independent variables in the dataset to cluster them between the given minimum and maximum values. Consequently, the feature scaling method normalizes the data over a particular range which helps in speeding up the computations. In this context, the Standard Scaler from Sklearn library has been used in this article to scale the data, which standardizes the data features by removing the means and scaling it to a unit variance [31].

IV. CONSTRUCTION, TRAINING AND EVALUATION OF DNN MODEL

This section explains the design and training of DNN using Python language for the target application (detection of epileptic seizures). The target application requires 178 nodes in the input layer and 1 node at the output layer. The size of hidden layer can be varied as per requirements. In this article, we have used two different sizes for the hidden layers i.e 32×32 and 64×64. Consequently, the final architectures, used for epileptic detection, are $178 \times 32 \times 32 \times 1$ and $178 \times 64 \times 64 \times 1$. Libraries and packages from Keras-Pandas [32] are used in Python to create and train the DNN architecture. The following steps are involved in the construction, training and evaluation of DNN: (1) Reading the data set, (2) Splitting the data set into training and test sets, (3) feature scaling, (4) Add layers (Input, Hidden-1, Hidden-2, Output), (5) Training of neural network (6) Save the final weights after training, (7) Predict the outputs for test set, (8) Calculate the accuracy

The target data set consists of 179 columns. The first 178 columns contain EEG features for patients while the column 179 describes whether this data belongs to the epileptic seizure class or non-epileptic class which is also termed as its label. The data is partitioned into two sets named as *Training Set* and *Test Set*. The training set is composed of 80% of total samples, whereas the remaining 20% are placed into the test set. A standard scaler function from the Sklearn library is used to scale the data in between the minimum and maximum values (0 and 1). This normalization of the data over a particular range of 0 to 1 helps in speeding up the calculations inside the deep learning algorithm.

The next three steps involve the creation of network layers, training of network and storing of the final weights. To do this, the Dense and Sequential functions are imported from Keras library to create an ANN. Size of the layers can be selected according to the designers choice. The dimensions for 64×64 architecture are selected. The Relu function is used for activating the neurons inside the input and hidden layers, but for the output layer sigmoid function is applied. The sigmoid function is used specially when there is a need to predict the probability as an output. Since our goal is to classify the test samples into seizure and non-seizure class which denotes to the class of binary representation. Probability can exists only between 0 and 1, so sigmoid function is the good choice to use in output layer as an activation function. Finally, the test set is applied to trained network to check the output behaviour using the predict function from Keras library. Predicted outputs are compared with the original labels of test samples. Number of correct and wrong predictions are counted in order to calculate the accuracy using the following formula:

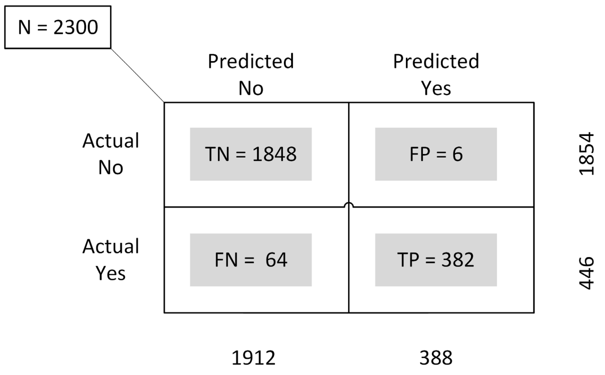


FIGURE 6. Confusion matrix output.

$$Accuracy = \frac{\text{corrects}}{\text{corrects} + wrongs} \tag{3}$$

A. EVALUATION OF DNN

For the evaluation of DNN, a confusion matrix is employed to measure the performance of classification for any test data where the true values are already known. The classifier has made 2300 predictions in total. Out of these 2300 cases, the classifier predicted *Yes* for 388 times, and *No* for 1912 times. Whereas in reality, 446 patients have disease and 1854 have no disease. The testing results, as summarized in Fig. 6, show that the *True Positives (TP)* are the cases in which model has predicted *Yes* (which means they have the disease), and they do have the disease in actual. Similarly, *True Negatives (TN)* implies that the model has predicted *No*, and patients don't have the disease. Moreover, *False Positives (FP)* means the model has predicted *Yes* but in real patients don't have the disease. Furthermore, the *False Negatives (FN)* show that the model has predicted *No* but in actual patients do have the disease. Consequently, the accuracy (how often the predictions of the classifier are correct) is formulated in Equation 4. Similarly, the error rate shows the frequency of wrong predictions, as formulated in Equation 5. Accuracy and error rate for our designed DNN model, calculated through Equation 4 and Equation 5 are 96.95% (1848+382)/2300) and 3.04% (1848+382)/2300) respectively.

$$Accuracy = \frac{\text{TP} + \text{TN}}{\text{N}} \tag{4}$$

$$ErrorRate = \frac{\text{FP} + \text{FN}}{\text{N}} \tag{5}$$

V. SYSTEM DESIGN

Once the DNN model is designed, trained and tested in Python, as described in Section IV, the next step is to present the design of proposed HADL system-on-chip solution which integrates a Cortex A-9 processor with Cyclone-V FPGA. The C-Program, running on the processor, is responsible to read the input samples (described in Section III-D) and weights of the DNN (computed in Section IV). Subsequently, it transfers input samples and weights to the FPGA side, using a DMA controller. Once the data is transferred, it informs FPGA to start the required processing. Subsequently, the output from FPGA is saved in a register. The processor reads the result from the corresponding register and transfers the next sample of data to the FPGA side.

A. PROCESSOR SYSTEM DESIGN

It has been mentioned in Section III-C that the Platform Designer tool has been used to design/configure the Processor System Module, as per requirements of the application. The complete Processor System Module consists of the following components: a hard processing system (HPS), a phase-lock loop (PLL), two block RAMs (SRAM-1 and SRAM-2), four DMA controllers (DMA-1, DMA-2, DMA-3 and DMA-4) and three GPIO registers. Once the components along with their specifications are selected, the connections are made accordingly. The PLL is responsible to provide the clock and reset signals to all other components. Similarly, DMA-1 and DMA-2 are used to copy all the data from HPS to SRAM-1 and back respectively. Furthermore, DMA-3 and DMA-4 a are employed to copy all the data from HPS to SRAM-2 and back respectively. For all the aforementioned components, there is a unique start-address and end-address, assigned by the the Platform Designer tool, which defines the span of the component's address range. This address range is ultimately used for the memory mapping in C-code, running on the processor, in order to access various components of the Processor System Module.

B. ACCELERATOR SYSTEM DESIGN

It consists of four main components: RBR, TMMU, RELU, and Sigmoid. The RBR component reads the corresponding data from BRAMs and delivers it to the TMMU module. The TMMU unit is then used recursively in every clock cycle until all the values from input layer are multiplied with their associated weights to generate the intermediate output value. The RELU approximation of the intermediate output goes into one of the corresponding nodes inside the hidden-layer-1. As a result, all the nodes of hidden layer-1 are calculated. Subsequently, the nodes of hidden layer-1 are multiplied with their associated weights to calculate the values for hidden layer-2. Similarly, the nodes of hidden layer-2 are processed to feed the nodes of output layer. Finally the Sigmoid approximation function is applied to calculate the final output.

1) RBR COMPONENT

Two factors are important in the design of RBR component: the data width of accelerator input and the number of multipliers in TMMU. In the proposed design, there are 32 multipliers and data width for each multiplier input is 16 bits. In order to compute the result of TMMU in a single clock cycle, it requires that 32 values (16-bit each) should be available in each clock cycle. These 16-bit inputs of 32 multipliers can only be read in a single clock cycle provided that 512 (16×32) bits are present at one address location of the BRAM. Therefore, the data width of BRAM memory is selected to 512.

2) TMMU COMPONENT

The parallelism can be achieved using a large number of hardware resources which leads to more speedup. However, the size of TMMU (tile size) is critical due to the limitation of available hardware resources. The cost of TMMU mainly depends upon the tile size (number of multipliers) and there is always a trade-off between the hardware resources and speedup of the system. In our design, the tile size depends upon the availability of hardware resources in the target DE1-SoC board. Therefore, the total number of multipliers are 32. Each multiplier has two 16-bit inputs and one 32-bit output. Finally, the outputs of all the multipliers are added in a pipeline strategy, as shown in Fig. 7.

3) RELU COMPONENT

It is used for the activation of neurons in the hidden layers of DNN. It performs a non-linear transformation to the output neuron, making it capable to learn more and perform complicated tasks. The output of TMMU is an input to RELU component to calculate the approximation value and feed the output to the corresponding node of hidden layer. Mathematically this function can be described by Equation 6:

$$F(x) = max(0, x) \tag{6}$$

4) SIGMOID COMPONENT

It is used to determine the output of a neural network (Yes or No). It maps the resulting values in between 0 to 1. The purpose of Sigmoid function in our case is to approximate the inputs (ranging from -8 to $+8$) to the outputs (ranging from 0 to $+1$). The Sigmoid function is particularly useful for all those scenarios where we one has to predict probability as an output. Since the probability of anything ranges from 0 to 1, Sigmoid function is the right choice for such kind of approximation. Mathematically, Sigmoid function can be described by Equation 7 which includes an exponent operator. However, the exponent factor is not included in VHDL library. Therefore, the mathematical function of sigmoid is implemented using look-up tables. To do this, Sigmoid function is first implemented in MATLAB with step-size of 0.5 with a range of (-8 to $+8$). Subsequently, all the results are stored in a lookup table defined in the VHDL component of sigmoid.

$$F(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

The communication between the Accelerator System Module and the Processor System Module is provided in Appendix VII. Furthermore, the functional Verification of

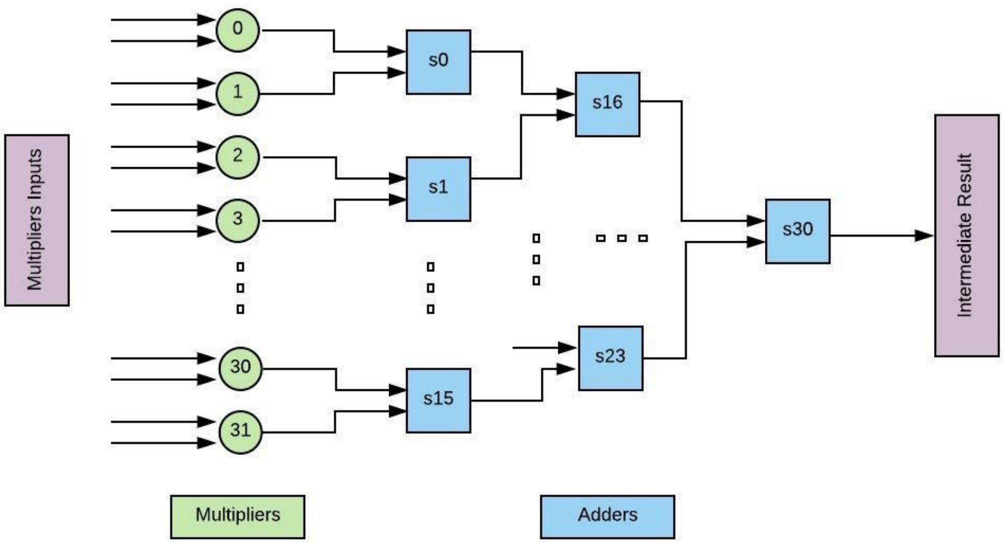


FIGURE 7. Block diagram for tiled matrix multiplication unit.

Processor System Module and Accelerator System Module is provided in Appendix VII and Appendix VII respectively. Section VII describes the implementation of HADL system on DE1-SoC development board. Consequently, the next section provides the Simulation results of Accelerator System on the target SoC board (DE1-SoC board).

VI. EVALUATION RESULTS

This section evaluates the performance of proposed HADL system for various architecture sizes in terms of execution time, accuracy, clock frequency, power consumption and the required hardware resources. Particularly, the results are provided for two different architecture sizes ($178 \times 32 \times 32 \times 1$ and $178 \times 64 \times 64 \times 1$). The input sample size for the target case study is is 178. The numbers 32×32 and 64×64 show that the number of nodes in hidden layers (tile size or number of multipliers) are 32 and 64 respectively. First, the experimental results for native implementation (C code only) on ARM 9 processor of DE1-SoC board are presented in Section VI-A. Subsequently, the results for complete HADL system (C + VHDL) are provided and benchmarked with the pure C-implementation in Section VI-B and Section VI-C respectively. A generalized formula for the computation of total execution time is presented in Section VI-D. Similarly, the area utilization results are provided in Section VI-E. Finally, the proposed system is compared with various state-of-the-art solutions in terms of different performance attributes in Section VI-F.

A. RESULTS FOR NATIVE IMPLEMENTATION

The execution time and accuracy results for the native implementation (C code only) on Arm Cortex-A9 Processor, for the architecture sizes of 32×32 and 64×64, are shown in Table 1. It can be observed from Table 1 that the accuracy is increased with an increase in the architecture size but at the expanse of more execution time.

TABLE 1. Execution time and accuracy results for the native implementation on Arm cortex-A9 processor.

Architecture Size	Number of Input Samples	Accuracy ((%))	Execution Time (Seconds)
32X32	2300	95.56 %	1.06 Sec
64X64	2300	96.13 %	2.55 Sec

TABLE 2. Shifting and testing time for one sample in HADL system.

Architecture Size	Shifting Time	Time to Check Accuracy	Accelerator Time	Testing Time
32X32	30μsec	0.5μsec	7μsec	37.5μsec
64X64	30μsec	0.5μsec	19μsec	49.5μsec

B. RESULTS FOR THE PROPOSED HADL SYSTEM

Before analysing the results of proposed HADL on the target SoC board, it is important to compute the required shifting time for transferring one sample of data using DMA from HPS memory to the FPGA block memory, as shown in Table 2. Column 1 of Table 2 shows the corresponding architecture size. Similarly, column 2 shows the total shifting time which is 30μsec for both architecture sizes. The total shifting time is composed of two parts: The first part is the time required for writing one sample to HPS-on-Chip memories (28μsec). The second part is the DMA time for copying the data from HPS-on-Chip memories to FPGA (2μsec). Similarly, the third column shows the required time to check the accuracy of the input samples (size is 2300). The fourth column shows the accelerator time for two different architecture sizes. Finally, the last column shows the total testing time for one complete sample of input data. Total testing time is the sum of shifting time, time to check accuracy and the accelerator time.

The time and accuracy results for the proposed HADL system (C + VHDL) on DE1-SoC development board, for the architecture sizes of 32×32 and 64×64, are shown in Table 3.

TABLE 3. Time and accuracy results for the proposed HADL system (C + VHDL) on DE1-SoC.

Architecture Size	Number of Input Samples	Accuracy ((%))	Execution Time (Seconds)
32X32	2300	95.5 %	0.0861 Sec
64X64	2300	95.9 %	0.1146 Sec

TABLE 4. Comparison of native implementation with the proposed HADL system.

Arch. Size	Time(Sec) (Native)	Time(Sec) (HADL)	Speed-up	Accuracy (Native)	Accuracy (HADL)
32X32	1.06	0.0861	12.3	95.56 %	95.5%
64X64	2.55	0.1146	22.25	96.13 %	95.9%

C. COMPARISON OF NATIVE IMPLEMENTATION WITH THE PROPOSED HADL SYSTEM

Table 4 shows a comparison between the native implementation and the proposed HADL system, in terms of accuracy and execution time for two different architecture sizes. It can be observed from Table 4 that the total testing time for the proposed HADL system is much less as compared to the native implementation, with a very small effect on the accuracy. The minor decrease in accuracy is due to the scaling factor, selected during the shifting of input and weight data from Processor towards FPGA. After getting the final result from FPGA, it is de-scaled with the same scaling factor. For experimental results in Table 4, a scale of 2^{10} is used. This scaling factor can be increased to improve the accuracy, but at the same time, more memory and data width of registers are required. Consequently, there is a trade-off between hardware resources and accuracy which can be selected, depending upon the requirements of a specific application.

It can be observed from Table 4 that if we increase the architecture size, the total testing time in native as well as in HADL system increases. Increasing the architecture size implies that the number of nodes in the hidden layers are increased which consequently increases the number of computations involved, and therefore, more amount of time is required to compute the final result. In case of native implementation, the architecture size 64×64 takes more than the double time as compared to 32×32 size. On the other hand, the execution time is not doubled for the SoC-implementation case. The reason is the shifting time, which is same for both the architecture sizes, as explained in Section VI-B. It implies that one can expect more speedup as the architecture size is increased.

D. A GENERALIZED FORMULA FOR TOTAL EXECUTION TIME

While the execution time results for two different architecture sizes are shown in Table 4, it is important to compute the execution time for other architecture sizes also. However, the computation of execution time for a larger architecture size is time consuming. Therefore, a generalized formula for the calculation of total execution time, without performing simulations, is critical. The total execution time depends upon the number of total clock cycles for computing the accelerator output. Therefore, we have formulated the total number of required clock cycles in Equation 8:

$$\begin{aligned} CC = & \left[\left(\frac{X}{Ts} + 3\right) * HL_{(1)}\right] \\ & + \sum_{i=1}^{h-1}\left[\left(\frac{HL_{(i)}}{Ts} + 3\right) * HL_{(i+1)}\right] \\ & + \left[\left(\frac{HL_{(h)}}{Ts} + 3\right) * Y\right] \end{aligned} \tag{8}$$

where, CC is the total number of clock cycles, h denotes the total number of hidden layers, X presents the number of input nodes, Y shows the number of output nodes, $HL(i)$ is the size of corresponding i_{th} hidden layer, Ts is the tile size which shows the number of multipliers in the proposed accelerator unit.

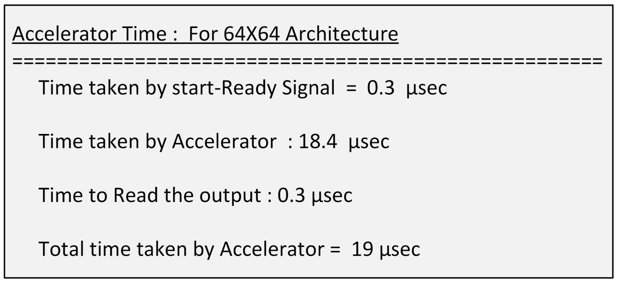
In addition to the aforementioned factors, it is important to note that a factor of 3 is added in Equation 8, as 3 additional clock cycles are required for the following reasons: the 1st clock cycle is required for updating the address signal of RAMs. When any value is written on the address, it is available only in the next clock cycle. Similarly, the 2nd clock cycle is needed for reading the input data from RAMs. The 3rd clock cycle is required for updating the output, as signals always update their values in the next clock cycle.

In order to validate the proposed formula, the following values are used: $Ts = 32$, $h = 2$, $X = 192$, $Y = 1$, $HL(1) = 64$, $HL(2) = 64$. If we put these values in Equation 8, We find that the total number of clock cycles are 901. Now, if 1 clock cycle takes 20 nano seconds, (i.e 50 MHZ clock is used) then the total execution time is 18.02 μsec. It is important to mention that the time obtained from Equation 8 is slightly different from the actual time obtained from the DE1-SoC hardware. For example, in the above example, the time obtained from Equation 8 is 18.02 μsec while the actual time obtained from the DE1-SoC hardware is 19 μsec, as shown in Fig. 8.

This small difference in execution time is due to some extra parameters which have not been included in the formula of Equation 8. For example, the time taken by HPS to FPGA bridge for informing the FPGA via start ready signal is not included. Similarly, the time taken by FPGA-to-HPS bridge to inform processor about the final result is also not included. Both the aforementioned bridges require 0.3 μsec for sending the control signal information. To summarize, if we include all these minor factors in Equation 8, the total execution time obtained from simulation will be equal to the total execution time obtained from the formula.

E. RESOURCE UTILIZATION

Table 5 and Table 6 provide a brief summary of resources, consumed by the proposed hardware accelerator, for architecture sizes 32×32 and 64×64. It can be analysed from these tables that the architecture size 64×64 consumes slightly more amount of resources as compared to 32×32 architecture but provides a relatively high speedup factor when benchmarked with the native C-implementation.



```
Accelerator Time :  For 64X64 Architecture
====================================================
   Time taken by start-Ready Signal  =  0.3  µsec

   Time taken by Accelerator  : 18.4  µsec

   Time to Read the output : 0.3 µsec

   Total time taken by Accelerator =  19 µsec
```

FIGURE 8. Accelerator time for 64 × 64 architecture.

TABLE 5. Resource utilization on DE1-SoC hardware for architecture size 32 × 32.

Resource	Utilization	Used / Available
Logic Utilization in (ALMs)	15%	4970 / 32070
Total Pins	28%	129 / 457
Total Registers	5.53%	7094/128300
Total Block Memory bits	7%	274432 / 4065280 bits
Total DSP blocks	18%	16 / 87
Total PLL	17%	1 / 6
Total DLL	25%	1 / 4

TABLE 6. Resource utilization on DE1-SoC hardware for architecture size 64 × 64.

Resource	Utilization	Used / Available
Logic Utilization in (ALMs)	16%	5060 / 32070
Total Pins	28%	129 / 457
Total Registers	6.045%	7392/128300
Total Block Memory bits	13%	536576/ 4065280 bits
Total DSP blocks	18%	16 / 87
Total PLL	17%	1 / 6
Total DLL	25%	1 / 4

The 32 × 32 architecture uses 7094 registers while 64 × 64 architecture consumes 7392 registers. It implies that the difference is not very high in terms of total registers used. However, 64 × 64 architecture uses 13% of block memory whereas 32 × 32 size uses only 7%.

F. COMPARISON WITH STATE-OF-THE-ART

Table 7 presents a comparison of proposed work with state-of-the-art in terms of speedup, network architecture size, clock frequency and power consumption. The work in [19] provides a speedup of 5 at 100 MHZ, for a network architecture size of 64 × 64. The power consumption results are not reported in [19] (shown as N.R in Table 7). On the other hand, for the same network architecture size (64 × 64), the proposed HADL system achieves a speedup of 22.3 at a relatively lower clock frequency (50 MHZ) and consumes 272 mW. The lower clock frequency ensures the lower power consumption. Similarly, the work in [20] obtains a speedup of 25 at 200 MHZ. However, the architecture size for the speedup in [20] is 256 × 256 which consumes relatively more hardware resources. The work in this article achieves almost the same speedup (22.3) with a significantly lower network size (64 × 64 network size) and reduced clock frequency (50 MHZ). Furthermore, the works in [19] and [20] provide their results for RBM algorithms only, whereas the proposed HADL system is much more scalable and flexible.

It can be observed from Table 7 that the work in [18] provides more speedup (117.87) as compared to the proposed solution. However, the higher speedup in [18] is achieved with a clock speed of 1 GHZ which may result in a very high power consumption (not reported in the article). Furthermore, the work in [18] is not implemented using a reconfigurable hardware, and therefore, it cannot adapt to different application demands. In other words, it is hard wired instead of implemented on an FPGA platform, and therefore, it cannot efficiently adapt to different sizes for network architecture.

TABLE 7. Comparison of HADL with state-of-the-art.

Work	Network Architecture	Clock Frequency	Speedup	Power Consumption
Chow [19]	64X64	100 MHZ	5	N.R
Kim [20]	256X256	200 MHZ	25	N.R
Dian [18]	General	0.98 GHZ	117.87	N.R
DLAU [9]	64X64	200 MHZ	19.41	1814 mW
HADL	**64X64**	**50 MHZ**	**22.3**	**272 mW**

A scalable and flexible solution is presented in [9] which obtains a speedup of 19.4, with a network architecture size of 64 × 64, at 200 MHZ clock with a power consumption of 1814 mW. In comparison, the proposed HADL system is able to achieve a speedup of 22.3 with the same network size (64 × 64), but with a reduced clock frequency of 50 MHZ. Moreover, the power consumption is reduced to 272 mW. In other words, although the work in [9] is scalable and flexible, but in comparison, the proposed HADL provides same scalability with high speedup and low power consumption. The reason for this performance difference is that the BRAMs are not used in [9] to store input samples. Instead, the authors in [9] have used an array of 32 registers at FPGA side to store 32 instances of a sample. Consequently, more number of DMA operations are required to shift one complete sample. Subsequently, the processing is performed on 32 instances of the sample. Once the processing is finished, the DMA is executed again to shift next 32 instances of the input sample. The proposed HADL design, on the other hand, uses an additional 2KB memory to store input sample data instead of an array of registers. This approach significantly reduces the number of DMA transfer operations. In [9], the DMA is executed 6 times to transfer the data for one input sample, whereas in HADL, it is executed only one time to transfer one input sample. As a result, the DMA transfer operations is 5 times less for a specific case of tile size (32). Similarly, the proposed HADL system uses 50 MHZ clock as compared to the work in [9] which uses 200 MHZ. Therefore, the proposed solution is also more efficient in terms of power consumption as compared to [9].

Experimental results in Table 7 demonstrate that the proposed scalable architecture provides a relatively high speedup and reduced clock frequency as compared to existing solutions. At this point, one can argue that if the input sample size is very large and the employed memory (2 KB) in the proposed HADL system is not sufficient to store it, then the input sample can not be shifted (transferred) in one DMA operation. In other words, the number of required DMA operations for the shifting of one complete input sample will increase. To avoid this situation, the memory size can be

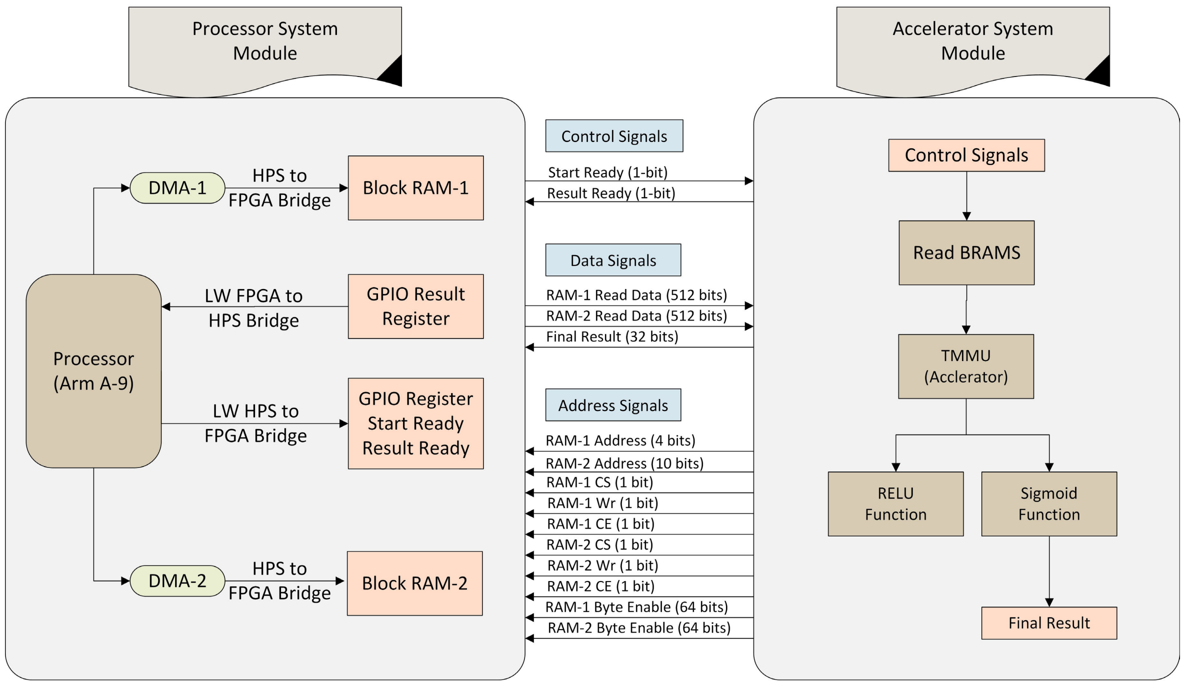


FIGURE 9. Communication between processor system and accelerator system.

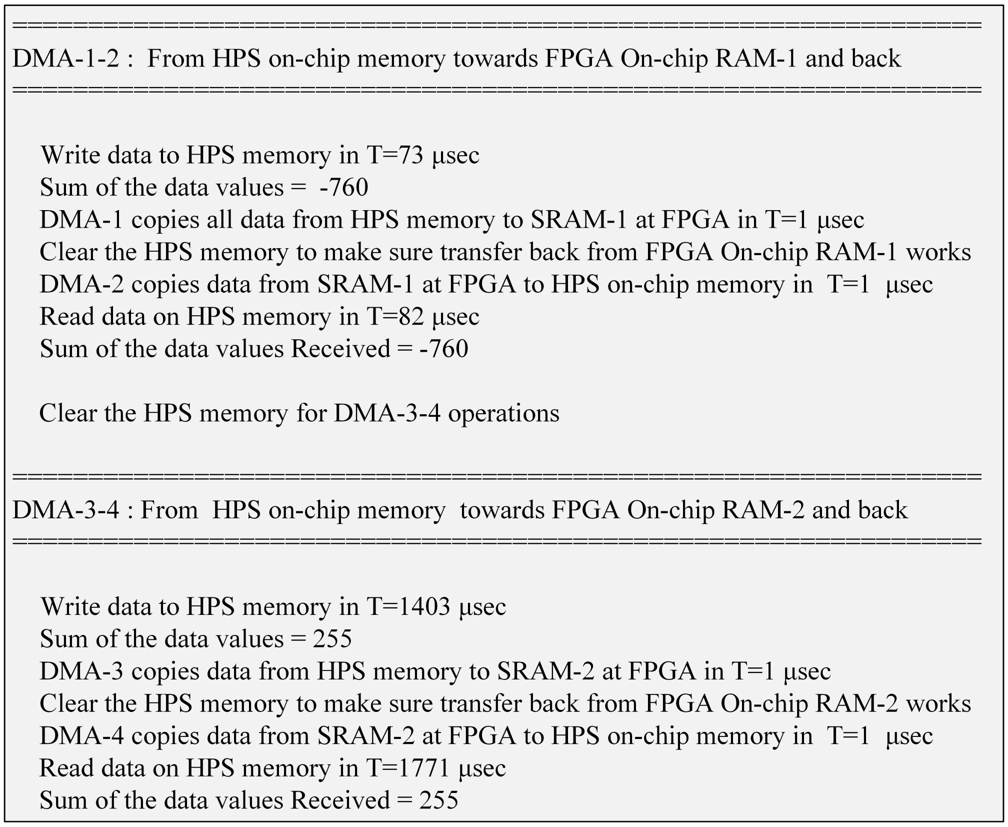
increased so that a complete input sample can be stored in a single DMA operation.

It is important to note that the proposed HADL system can be configured to operate with different architecture sizes, and therefore, exhibits a trade-off between speedup and hardware cost. Consequently, the proposed solution is more flexible to accommodate various machine learning applications. For example, the computational speed can be further increased by using a tile size of more than 32. However, some additional hardware resources are required to achieve an increase in the corresponding computational speed. Similarly, the total shifting time for data from HPS towards FPGA or from FPGA towards HPS can be further decreased. Again, some additional hardware resources are needed to obtain a decrease in data shifting time. Therefore, it can be concluded from aforementioned results and discussion that the proposed HADL system best suits those application where the speed and power consumption are more important parameters in comparison to hardware resources and cost.

To summarize, this section has evaluated the performance of proposed HADL system on DE1-SoC development board. For benchmarking, the DNN implementation in C is executed on the dual core Arm cortex A-9 processor. The tile size of 32 is used according to the availability of hardware resources in DE1-SoC development board. The clock frequency is 50 MHZ which is much less as compared to state-of-the approaches [9], [18], [20]. Although, only two network sizes (32 × 32 and 64 × 64) are tested (simulated) to compare the results with state-of-the-art approaches, however, a general-purpose formula is proposed to calculate the total number of required clock cycles for any architecture size, without performing simulation on the target development board. Finally, the comparison of HADL with state-of-the-art is provided in terms of execution time, accuracy, power consumption and speed-up.

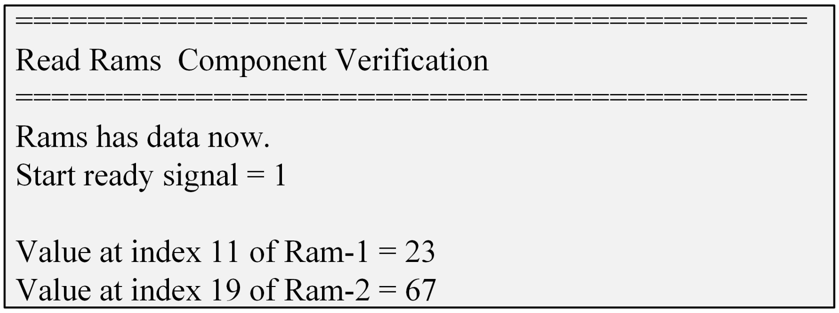
VII. CONCLUSION

This article has presented a co-design approach, for the classification of epileptic and non-epileptic samples, using Deep Neural Network (DNN). The proposed co-design approach starts with the construction of a DNN for the target case study. The construction of DNN is followed by the system implementation phase of the design process. In system implementation phase, a hardware accelerator is developed for computational intensive parts of the DNN. The developed hardware employs a tile technique to split the input data into smaller sets and compute the arithmetic logic iteratively. Consequently, a hardware-software co-simulation is performed on a System-on-Chip (SoC) development board. In order to validate the performance of proposed solution, the developed hardware accelerator is first benchmarked with a pure software implementation running on the ARM processor of the target SoC development board. Two different network architectures, with hidden layer sizes equal to 32 × 32 and 64 × 64, are used to explore the impact of architecture size on speed and accuracy. The performance of proposed solution is also compared with state-of-the-art in terms of execution time, clock frequency, power consumption and architecture size. The performance comparison show that the proposed solution achieves a higher speedup with a relatively lower clock frequency and power consumption.



```
=====================================================================
DMA-1-2 :  From HPS on-chip memory towards FPGA On-chip RAM-1 and back
=====================================================================

   Write data to HPS memory in T=73 μsec
   Sum of the data values =  -760
   DMA-1 copies all data from HPS memory to SRAM-1 at FPGA in T=1 μsec
   Clear the HPS memory to make sure transfer back from FPGA On-chip RAM-1 works
   DMA-2 copies data from SRAM-1 at FPGA to HPS on-chip memory in  T=1  μsec
   Read data on HPS memory in T=82 μsec
   Sum of the data values Received = -760

   Clear the HPS memory for DMA-3-4 operations

=====================================================================
DMA-3-4 : From  HPS on-chip memory  towards FPGA On-chip RAM-2 and back
=====================================================================

   Write data to HPS memory in T=1403 μsec
   Sum of the data values = 255
   DMA-3 copies data from HPS memory to SRAM-2 at FPGA in T=1 μsec
   Clear the HPS memory to make sure transfer back from FPGA On-chip RAM-2 works
   DMA-4 copies data from SRAM-2 at FPGA to HPS on-chip memory in  T=1  μsec
   Read data on HPS memory in T=1771 μsec
   Sum of the data values Received = 255
```

FIGURE 10. Output verification for processor system module.



```
============================================
Read Rams  Component Verification
============================================
Rams has data now.
Start ready signal = 1

Value at index 11 of Ram-1 = 23
Value at index 19 of Ram-2 = 67
```

FIGURE 11. Simulation results for RBR component.

APPENDIX A
COMMUNICATION BETWEEN MODULES

The communication between the Accelerator System and the Processor System modules is shown in Fig. 9. There are two control signals (single bit each) which are responsible to start and stop the accelerator. The *Address Signals* in Fig. 9 are dedicated for various configuration settings (e.g. read/write the data from/to the BRAMs). The width of read signals is 512, which is further divided into 32 signals (16 bit each). These signals are then assigned to the inputs of multipliers. Finally, the width of final result register is selected as 32 bits.

APPENDIX B
VERIFICATION OF PROCESSOR SYSTEM

To verify the functionality of Processor System Module, some known values are transferred from Processor to FPGA and then read them back, as shown in Fig. 10.

APPENDIX C
VERIFICATION OF ACCELERATOR SYSTEM

The synthesizeable code for each of the four components (RBR, TMMU, RELU and Sigmoid) is written in VHDL and simulated. The RBR component is used to read the data from memories and assigns the data to the next unit for further processing. As a test example, some random numbers (23 and 67) are copied from Processor to FPGA at index number 11 and 19 respectively. The output of RBR component assigned to the final system output, as shown in Fig. 11. Once the inputs of the Accelerator System Module are read correctly, the data is provided as an input to TMMU component. At first, the TMMU is simulated independently

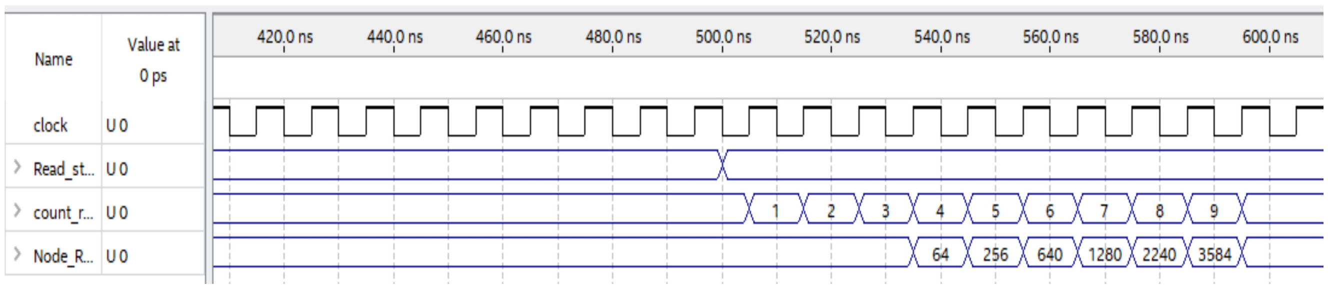


FIGURE 12. **Behaviour of TMMU inside the accelerator system module.**

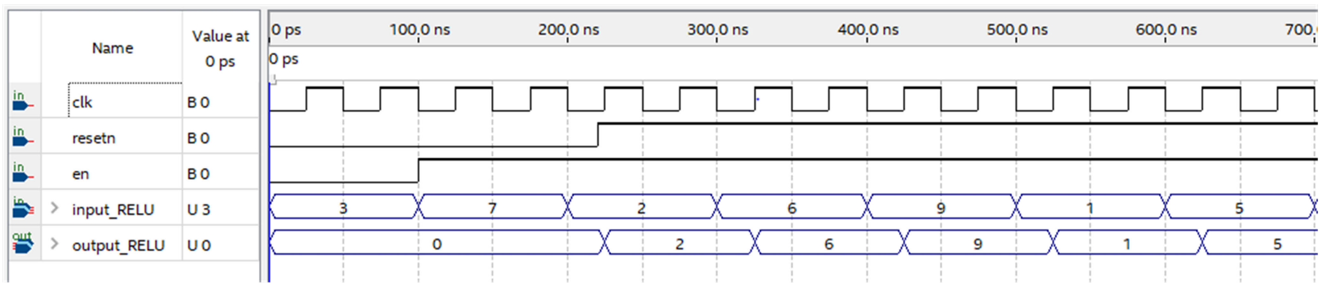


FIGURE 13. **Simulation results for RELU component.**

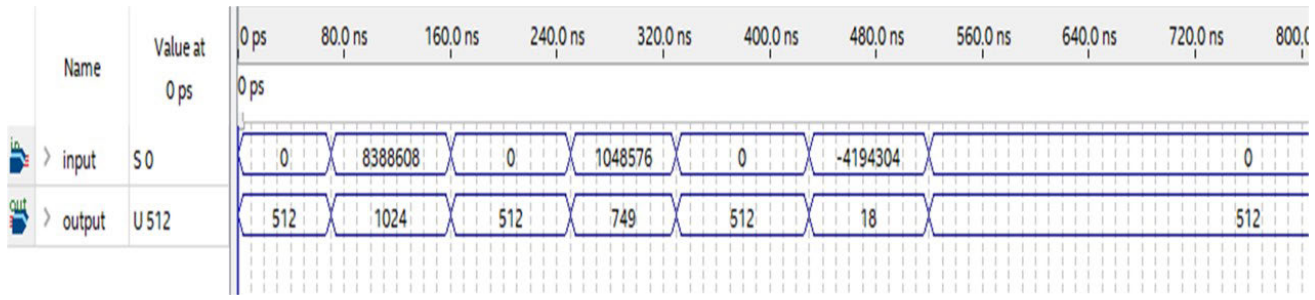


FIGURE 14. **Simulation results for sigmoid component.**

to verify the output and then the component is instantiated inside the Accelerator System Module to verify the combined behaviour, as shown in Fig. 12. As soon as the read start signal is available, the processing begins and the counter is updated. The first result is achieved after three clock cycles. The delay of three clock cycles is due to the update of block RAM's addresses and the final result. In order to check the functionality of RELU component, some random values are assigned to the input of the component. The corresponding response is verified accordingly, as shown in in Fig. 13. The node values, calculated for the last hidden layer, are provided to the sigmoid component and the corresponding output is verified, as shown in Fig. 14. The output ranges between 0 and 1, with a scaling of 2^{10}. The scaling factor is required to convert decimal values (between 0 and 1) to some large numbers so that the processor and FPGA can communicate with each other. Without this scaling factor, the processor and FPGA can send/receive only zeros as the decimal parts are skipped. The data type of output entity is unsigned, as the output is always positive (ranging from 0 to 1). However, the data type of input is signed (ranging from -8 to +8), with a scaling factor of 2^{10}.

REFERENCES

[1] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. W. M. van der Laak, B. van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Med. Image Anal.*, vol. 42, pp. 60–88, Dec. 2017.

[2] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep learning for generic object detection: A survey," *Int. J. Comput. Vis.*, vol. 128, pp. 261–318, Feb. 2020.

[3] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M. Shyu, S. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, p. 92, 2018.

[4] S. Dargan, M. Kumar, M. R. Ayyagari, and G. Kumar, "A survey of deep learning and its applications: A new paradigm to machine learning," *Arch. Comput. Methods Eng.*, vol. 27, no. 4, pp. 1071–1092, Sep. 2020.

[5] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, Nov. 2018, Art. no. e00938.

[6] S. A. B. Shah, M. Rashid, and M. Arif, "Estimating WCET using prediction models to compute fitness function of a genetic algorithm," *Real-Time Syst.*, vol. 56, no. 1, pp. 28–63, Jan. 2020.

[7] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53040–53065, 2019.

[8] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digit. Signal Process.*, vol. 73, pp. 1–15, Feb. 2018.

[9] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 3, pp. 513–517, Mar. 2017.

[10] M. Rashid, S. A. B. Shah, M. Arif, and M. Kashif, "Determination of worst-case data using an adaptive surrogate model for real-time system," *J. Circuits, Syst. Comput.*, vol. 29, no. 1, Jan. 2020, Art. no. 2050005.

[11] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on GPUs," *J. Syst. Archit.*, vol. 99, Oct. 2019, Art. no. 101635.

[12] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, Mar. 2020.

[13] A. G. Blaiech, K. Ben Khalifa, C. Valderrama, M. A. C. Fernandes, and M. H. Bedoui, "A survey and taxonomy of FPGA-based deep learning accelerators," *J. Syst. Archit.*, vol. 98, pp. 331–345, Sep. 2019.

[14] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–37, May 2020.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and R. Boyle, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annual Int. Symp. Comput. Archit.*, New York, NY, USA, 2017, pp. 1–12.

[16] *DE1-SoC Board User Manual*. ALTERA Corporation, ALTERA Univ. Programme, Terasic Technol., San Jose, CA, USA, 2019, pp. 1–2. Accessed: May 2021. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836&PartNo=4

[17] F. Pajuelo-Holguera, J. A. Gómez-Pulido, F. Ortega, and J. M. Granado-Criado, "Recommender system implementations for embedded collaborative filtering applications," *Microprocessors Microsyst.*, vol. 73, Mar. 2020, Art. no. 102997.

[18] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, Apr. 2014.

[19] D. L. Ly and P. Chow, "A high-performance FPGA architecture for restricted Boltzmann machines," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2009, pp. 73–82.

[20] S. K. Kim, L. C. McAfee, P. L. Mcmahon, and K. Olukotun, "A highly scalable restricted Boltzmann machine FPGA implementation," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2009, pp. 367–372.

[21] P. Boonyakitanont, A. Lek-uthai, K. Chomtho, and J. Songsiri, "A review of feature extraction and performance evaluation in epileptic seizure detection using EEG," *Biomed. Signal Process. Control*, vol. 57, Mar. 2020, Art. no. 101702.

[22] R. G. Andrzejak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. E. Elger, "Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 64, no. 6, Nov. 2001, Art. no. 061907. Accessed: May 2021. [Online]. Available: https://repositori.upf.edu/handle/10230/43637

[23] World Health Organization. *Epilepsy: A Public Health Imperative*. Accessed: May 2021. [Online]. Available: https://www.who.int/publications-detail/epilepsy-a-public-health-imperative

[24] Z. Chen, G. Lu, Z. Xie, and W. Shang, "A unified framework and method for EEG-based early epileptic seizure detection and epilepsy diagnosis," *IEEE Access*, vol. 8, pp. 20080–20092, 2020.

[25] M. Rebsamen, Y. Suter, R. Wiest, M. Reyes, and C. Rummel, "Brain morphometry estimation: From hours to seconds using deep learning," *Frontiers Neurol.*, vol. 11, p. 244, Apr. 2020.

[26] Q. Wang, X. Gao, K. Wan, F. Li, and Z. Hu, "A novel restricted Boltzmann machine training algorithm with fast Gibbs sampling policy," *Math. Problems Eng.*, vol. 2020, pp. 1–19, Mar. 2020.

[27] H. Shen and H. Li, "A gradient approximation algorithm based weight momentum for restricted Boltzmann machine," *Neurocomputing*, vol. 361, pp. 40–49, Oct. 2019.

[28] Anaconda. *Solutions for Data Science Practitioners and Enterprise Machine Learning*. Accessed: May 2021. [Online]. Available: https://www.anaconda.com/

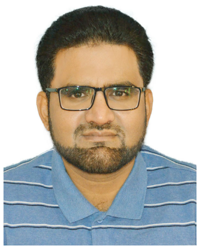
[29] *Platform Designer (Formerly Qsys)*. Accessed: May 2021. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/features/qts-platform-designer.html

[30] *Quartus Prime Lite Edition*. Accessed: May 2021. [Online]. Available: https://fpgasoftware.intel.com/?edition=lite

[31] *Scikit-Learn: Machine Learning in Python*. May 2021. [Online]. Available: https://scikit-learn.org/

[32] *Keras-Pandas 3.1.0*. Accessed: May 2021. [Online]. Available: https://pypi.org/project/keras-pandas/

FAISAL SHEHZAD received the M.S. degree in microelectronics and microsystems from the University of Bremen, Germany, in 2020. During his M.S. thesis, he has worked on system-on-chip implementation of deep neural networks. Previously, he has also served as an Assistant Manager of the Advanced Engineering Research Organization, Pakistan, for five years. His research interests include digital system design, embedded programming, and microelectronics.



MUHAMMAD RASHID received the Ph.D. degree from the University of Bretagne Occidentale, France, in 2009. He has been a Faculty Member with the Computer Engineering Department, Umm Al-Qura University (UQU), Mecca, Saudi Arabia, since last ten years. He worked with the Thomson Research and Development, for three years, and the Advanced Engineering Research Organization, Pakistan, for six years. He possesses an experience of more than 20 years in academia and industry for the design automation of modern embedded systems. He has published more than 100 articles in reputed international journals and conferences. His achievements in the field can be found at: https://uqu.edu.sa/en/mfelahi.

MOHAMMED H. SINKY received the Ph.D. degree from Oregon State University, USA, in 2015. He is currently working with the Computer Engineering Department, Umm Al-Qura University (UQU), Mecca, Saudi Arabia. His research interests include computer architecture, embedded programming, and smart systems.

SAUD S. ALOTAIBI (Member, IEEE) received the Doctorate degree in computer science from Colorado State University, USA, in 2015. He is currently an Associate Professor with the Department of Information Systems, Umm Al-Qura University (UQU), Mecca, Saudi Arabia. His research interests include emotional intelligence, data mining, natural language processing, machine learning, deep learning, computer networks, and network security.

MUHAMMAD YOUSUF IRFAN ZIA received the Doctorate degree in telecommunication engineering from the University of Malaga, Spain, in 2021. For more than 20 years, he has served as a faculty member at various institutes. His research interests include embedded systems, cost-effective and energy-efficient designs, and wireless communication systems in oceanic engineering.