System-on-a-chip (SoC)-based Hardware Acceleration for Extreme Learning Machine

Amin Safaei, Q. M. Jonathan Wu (Senior Member, IEEE), Yimin Yang, Thangarajah Akilan
Department of Electrical and Computer Engineering
University of Windsor
Windsor, ON, Canada
Email: {safaeia,jwu,eyyang,thangara}@uwindsor.ca

***Abstract*—Extreme learning machine (ELM) is a popular class of supervised models in machine learning that is used in a wide range of applications, such as image object classification, video content analysis (VCA) and human action recognition. However, ELM classification is a computationally demanding task, and the existing hardware implementations are not efficient for embedded systems. This work addresses the implementation of extreme learning machine (ELM) in a system on a chip field-programmable gate-array (SoC FPGA)-based customized architecture to efficiently utilize hardware accelerator. The optimization process consists of parallelism extraction, algorithm tuning and deep pipelining.**

***Index Terms*—Extreme learning machine, system on chip field-programmable gate array (SoC FPGA), hardware (HW) neural networks (NNs).**

I. Introduction

Neural networks (NNs) are a powerful set of machine learning algorithms, where the complexity of algorithm is defined based on the number of samples, that have been used in a wide range of applications. As a result, for applications that require high accuracy, the computational operation load is increased. Therefore, when the application is embedded, such as embedded vision, automotive or security systems (surveillance systems) that are required to work in real-time, and power management, the computational operations and the number of utilized resources (cells) become more important.

One of the active research area in NNs is extreme learning machines (ELMs), which are known for computational efficiency and performance for large data processing. ELMs are based on the theory of random vector functional link (RVFL) networks [1]; however, ELMs outperform RVFL networks [1]. ELM was proposed as a single-hidden-layer feedforward neural classifier in which the hidden layer does not need to be neuron-like. Fig 1 represents the structure of an ELM that consists of a random hidden layer with nonlinear feature mapping for the output, which has the advantages of low training complexity, fast learning speed, ability to use different types of activation functions [1] and well-known representation ability. Currently, many researchers are aiming to develop machine learning algorithms with better rates and performance; however, few researchers have focused on developing a hardware implementation to execute the whole algorithm. This has motivated substantial research to develop a system to utilize parallel computing platforms, such as field-programmable gate arrays (FPGAs) [2] and graphic processing units (GPUs), [3] or specialized multi-core microprocessors with architecture optimized for multiple computational operations. Implementations of ELMs on GPU platforms have been discussed recently [4]; however, GPUs face challenges with regard to power consumption [5] and are thus difficult to deploy in embedded environments. Hence, a new generation of FPGA-based SoC that is composed of both a microprocessor and FPGA on a single chip [6], consumes less power and can be built into small systems offers an attractive platform for embedded applications.

In this work, we propose a specialized SoC hardware implementation and design approaches for embedded ELM classification applications, such as human action recognition, where classification needs to be performed with low power and often with limited available resources. The presented design is part of research into developing embedded systems for recognizing human actions. The parts related to pre-processing [7] [8] [9] and feature extraction [10] [11] were previously reported, and in this work, we mainly focus on the final step, which is classification.

This paper is organized as follows. Section II provides the background on ELM classifiers and related work. The design of the module for implementing ELM on an embedded platform is presented in section III. Section IV presents SoC FPGA experimental and implementation results. Finally, conclusions are drawn in Section V.

II. Related Work

The fixed training time, possibility of parallel operation and temporal requirements make ELM a good candidate for hardware implementation. Because of the specific structure of NNs, general processors are not efficient for the implementation of NNs that are designed for applications that require online training and classification. To cope with this problem, different hardware accelerators based on FPGA, GPU, VLSI/ASIC or SoC FPGA have been proposed recently to improve the performance of NN designs. The difficulty of the design procedure and the cost of production make the VLSI/ASIC approach unpopular. Several works tried to improve the classification speed by accelerating data processing in the GPU [3], where, as we discussed before, power

978-1-5386-1911-7/17/$31.00 ©2017 IEEE

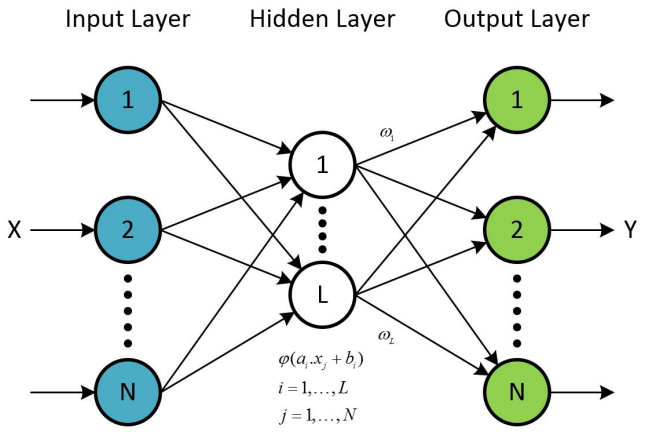

Fig. 1. System illustration

consumption is a problem for embedded platforms. Among these approaches, configurable hardware, such as FPGAs and complex programmable logic devices (CPLDs), have attracted increasing attention from researchers because they have the advantages of good performance, high energy efficiency, fast development and simple coding.

A. Extreme Learning Machine (ELM)

ELM was mainly proposed as a learning theory for single-hidden-layer feedforward neural networks (SLFNs), where the hidden layer need not be neuron-like. Traditionally, the gradient-descent-based method was used for feedforward neutral networks and required all the parameters to be tuned, which resulted in long processing times. By contrast, ELM requires one hidden layer in which all the parameters of the layer, weights and bias are randomly defined. Inverse operation can be used to find output weights to link the hidden layer to the output. In this part, we briefly discuss the fundamental theory of the ELM. More detail has been provided in [1]. Let consider X represent a training data set of N arbitrary distinct sample pairs $(x_i, y_i), i = 1 \ldots N$, where $x_i = [x_{i1}, x_{i2}, \ldots, x_{in}]^T \in R^n$ is the i_{th} input vector and $y_i = [y_{i1}, y_{i2}, \ldots, y_{im}]^T \in R^m$ is the target vector. Then, the SLFN with L nodes in the hidden layer, the activation function $\varphi(x)$ and the output function $f(x)$ are defined as follow:

$$f_L(x) = \sum_{i=1}^{L} \omega_i \varphi_i(x) = \varphi(x)\omega \tag{1}$$

$$y_j = \sum_{i=1}^{L} \omega_i \varphi(a_i.x_j + b_i) \quad j = 1, \ldots, N, \tag{2}$$

where ω_i is the weight vector connecting the i_{th} hidden node and the output nodes, and $\varphi(x) = [\varphi_1(x), \ldots, \varphi_L]$ is the ELM nonlinear feature mapping. The vector $\varphi_i(x)$ is the i_{th} hidden node output, where a_i connects the input layer to the hidden node through a set of weights $a_i = [a_{i1}, a_{i2}, \ldots, a_{in}]^T$, and $b_i = [b_{i1}, b_{i2}, \ldots, b_{in}]^T$ is a biased term. The training procedure of the ELM consists of two steps. First, hidden node parameters (a, b) are randomly defined to map input data to the feature space. The mapping function can be any activation function [1], and the sigmoid function is commonly used.

$$\varphi_i(x) = g(a_i, b_i, x) \tag{3}$$

$$g(a, b, x) = \frac{1}{1 + e^{-a.x+b}} \tag{4}$$

The second step is finding the values of weights that connect the hidden node to the output node ω_i, which is achieved by minimizing the convex cost,

$$\min_{\omega \in R^{L \times m}} \|\varphi\omega - y\|^2 \tag{5}$$

$$\varphi = \begin{bmatrix} \varphi(x_1) \\ \vdots \\ \varphi(x_N) \end{bmatrix}_{N \times L} = \begin{bmatrix} \varphi_1(x_1) & \cdots & \varphi_L(x_1) \\ \vdots & \ddots & \vdots \\ \varphi_1(x_N) & \cdots & \varphi_L(x_N) \end{bmatrix}_{N \times L} \tag{6}$$

$$\omega = \begin{bmatrix} \omega_1^T \\ \vdots \\ \omega_L^T \end{bmatrix}_{L \times m} = \begin{bmatrix} \omega_{11} & \cdots & \omega_{1m} \\ \vdots & \ddots & \vdots \\ \omega_{L1} & \cdots & \omega_{Lm} \end{bmatrix}_{L \times m} \tag{7}$$

$$y = \begin{bmatrix} y_1^T \\ \vdots \\ y_N^T \end{bmatrix}_{N \times m} = \begin{bmatrix} y_{11} & \cdots & y_{1m} \\ \vdots & \ddots & \vdots \\ y_{N1} & \cdots & y_{Nm} \end{bmatrix}_{N \times m} \tag{8}$$

where y and φ are the training data and hidden layer output matrix, respectively, and $\|.\|$ is the Euclidean norm. Each row of the matrix φ is the hidden layer feature mapping with respect to the i_{th} input x_i, and each column of φ is the i_{th} hidden node output with respect to inputs $x_1, x_2, \cdots, x_N$. To solve equation 5, consider that the number of the hidden neurons is equal to the number of samples; then, matrix φ is square and invertible. Output ω is obtained by inverting matrix φ; however, in practice, when the number of samples is greater than the number of hidden nodes, matrix φ is not square and equation 5 is not resolvable. To overcome this problem, since a and b are fixed, equation 5 can be considered to be a linear system that can be rewritten as:

$$\omega = \varphi^{\dagger} y, \tag{9}$$

where $\varphi^{\dagger}$ is the Moore–Penrose pseudoinverse of matrix φ, which can be obtained as follows:

$$\varphi^{\dagger} = (\varphi^T \varphi)^{-1} \varphi^T \rightarrow \text{If } \varphi^T\varphi \text{ is nonsingular} \tag{10}$$

$$\varphi^{\dagger} = \varphi^T (\varphi^T \varphi)^{-1} \rightarrow \text{If } \varphi^T\varphi \text{ is singular.} \tag{11}$$

Since the training procedure only requires three calculation steps, the training time can be extremely fast, which is necessary for applications that require real-time training.

B. ELM Computation

As discussed in equation 9, matrix inversion, transposition and multiplication are required to obtain matrix $\varphi^{\dagger}$. Of these three operations, inversion is the most challenging. Different approaches have been proposed for matrix decomposition, and $Cholesky$, LU and QR are the most common methods. Cholesky and LU are mainly used for positive and non-singular square matrices, whereas QR can be used for any type

of matrix. QR decomposes a given square matrix φ into an orthogonal Q and a triangular matrix R, $\varphi = QR$, and the inverted matrix can be computed as $\varphi^{-1} = (Q.R)^{-1} = R^{-1}.Q^t$. For nonsquare matrices, the pseudoinverse of φ is found as $\varphi^{\dagger} = R^{-1}.Q^t$. Different approaches have been proposed to perform QR decomposition: *Gram-Schmidt*, *Householder Transformation* and *Givens Rotations*. Among of them, *Givens Rotations* is most popular because of its stability, and *Gram-Schmidt* is not efficient when run on hardware [12]. The problem of stability is solved in the modified model of *Gram-Schmidt (MGS)*. In addition, in contrast to the original algorithm, which requires a square root operation in the final step, it requires fewer operations and resources [13].

$$\varphi = [\varphi_1|\varphi_2|\cdots|\varphi_n] \tag{12}$$

$$u_1 = \varphi_1 \tag{13}$$

$$u_2 = \varphi_2 - \frac{\langle \varphi_2, u_1 \rangle}{\langle u_1, u_1 \rangle} u_1 \tag{14}$$

$$u_n = \varphi_n - \frac{\langle \varphi_n, u_1 \rangle}{\langle u_1, u_1 \rangle} u_1 - \ldots \frac{\langle \varphi_n, u_{n-1} \rangle}{\langle u_{n-1}, u_{n-1} \rangle} u_{n-1} \tag{15}$$

$$e_1 = \frac{u_1}{\| u_1 \|} \quad e_2 = \frac{u_2}{\| u_2 \|} \quad \ldots\ldots\ldots \quad e_n = \frac{u_n}{\| u_n \|} \tag{16}$$

$$\varphi = \underbrace{[e_1|e_2|\cdots|e_n]}_{\mathbf{Q}} \underbrace{\begin{bmatrix} \varphi_1 \cdot e_1 & \varphi_2 \cdot e_1 & \ldots & \varphi_n \cdot e_1 \\ 0 & \varphi_2 \cdot e_2 & \ldots & \varphi_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \varphi_n \cdot e_n \end{bmatrix}}_{\mathbf{R}} \tag{17}$$

$$\varphi^{-1} = (Q.R)^{-1} = R^{-1}.Q^t \quad \text{If } \varphi \text{ is a square matrix} \tag{18}$$

$$\varphi^{\dagger} = (Q.R)^{-1} = R^{-1}.Q^t \quad \text{If } \varphi \text{ is a non-square matrix} \tag{19}$$

III. Hardware Implementation

In this work, the proposed implementation is synthesized and run on a Xilinx Zynq SoC 7000. The advantage of the Xilinx Zynq SoC is that it has two core ARM processors (PS unit) with a programmable logic block (PL unit) in a single chip. In this implementation, we used the first processor to control the sequence of operations consisting of training and prediction. Also, all the control signals of the modules in the PL block are controlled by the processor.

The AMBA AXI version 4 protocol is the main interface for communication between the processor and programmable logic. The AXI interface consists of four 64/32-bit high-performance AXI (HP AXI) slave interfaces, two 32-bit AXI master interfaces, two 32-bit AXI slave interfaces, a 64-bit AXI accelerator coherency port (ACP) and an extended multiplexed I/O (EMIO) interface.

In the traditional accelerating system, separate chips, such as DSP, FPGA and GPU, work as co-processors for the host processor. The host processor manages control signals and transfers data to shared memory. The co-processor is triggered by the host processor to execute the operation on the shared data. The output of the algorithm is then sent back to shared memory. In the final stage, the host processor reads the content of the shared memory and transfers it to the output port. All transactions are controlled and monitored by the host processor, which is inefficient and leads to high overhead. In the new architecture that we use in this study, the shared memory between the host processor and accelerator is an L2 cache with low latency. The ACP is the interface between the AXI4-Stream interconnect and the host processor (ARM CPU). The ACP is a 64-bit AXI slave interface on the snoop control unit (SCU), which provides an asynchronous cache-coherent access point directly from the PL to the host processor subsystem. The ACP provides a low-latency path between the PS and the accelerator implemented in the PL. The host processor needs only to initialize the cache, after which the input data are transferred in a completely independent manner to the DMA in the PL. As a result, more cycles remain available on the host processor for higher-level processing. The implementation of ELM operates in a two-clock domain. The first clock domain is used to train the ELM, and after the input data are transferred from shared memory to the DMA, the circuit illustrated in figure 2 is used to execute the MGS algorithm to find Q and the R matrix. The circuits consist of three shift registers that store the results of operations u_i, $< u_i.u_i >$ and $< u_i.\varphi_n >$. Each cell of the shift registers update in parallel, which saves more clock cycles for the rest of the operation. After updating, the content of each cell is passed to the temp register for division and subtraction. The obtained u_i is sent back to the second shift register to compute u_{i+1}. The MUX_1 is used to control the input of the second shift register since in the initialization it is necessary for φ_1 to pass directly to the second SHR_2. The second MUX_2 controls the input for the subtraction operation. φ_n is first subtracted from $\frac{\langle \varphi_n, u_1 \rangle}{\langle u_1, u_1 \rangle} u_1$, and then it is subtracted from the rest of the items (Equ 15).The output of the circuit in figure 2 generates $\{u_1, u_2, ..., u_n\}$, which can be used to find $[e_1|e_2|...|e_n]$. A square root operation is required to find e_i, and it is implemented by an arithmetic unit IP in Vivado. The output of the arithmetic unit generates each element of matrix Q. The matrix Q is transfered to the shared memory L2 cache, and the host processor computes matrix R and finally $\varphi^{\dagger}$. After training and obtaining all the parameters, the circuit in figure 3 is used for prediction, which operates in the second clock domain. It consist of four main modules: *Pre-Computation*, *Control-Input*, *Neuron Computation* and *Output*. The first stage consists of multiple shift registers, where each shift register generates a vector for the i_{th} neuron:

$$[(a_i.x_1 + b_i), (a_i.x_2 + b_i), \ldots, (a_i.x_N + b_i)]. \tag{20}$$

The input of each neuron is selected by MUX, and the output of each neuron is computed as $\varphi_i = g(a_i, b_i, x_{j=1\ldots N})$ and passes to the output stage to compute $y_j = \sum_{i=1}^{L} \omega_i \varphi(a_i.x_j + b_i) \quad j = 1, \ldots, N$.

IV. Performance of the FPGA Implementation

The proposed FPGA implementation of the ELM algorithm is performed on a Zynq-7000 (XC7Z020) Xilinx SoC with a Vivado 2015.2 synthesizer.

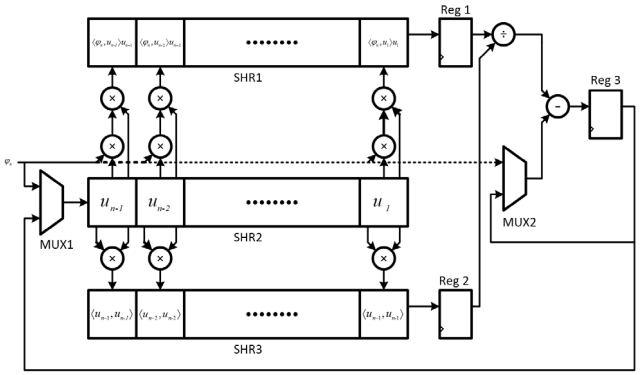

Fig. 2. Schematic diagram of the MGS-based Q and R matrix computation.

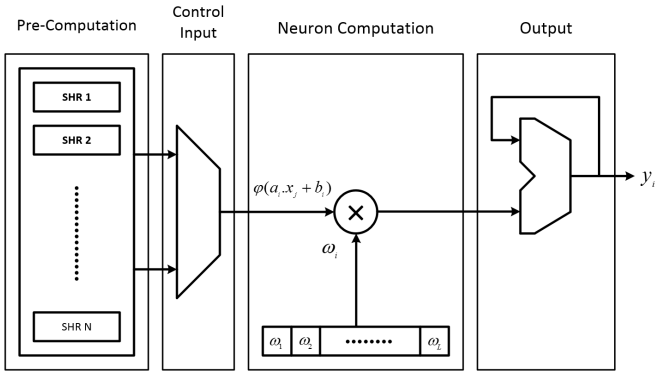

Fig. 3. Prediction circuit.

Comparison of the implementation results with those of other works is difficult since all preivous works consider different testing datasets, different numbers of hidden layers, and different types of variables (fixed point or floating point). Also, different solutions are defined based on the varying applications. As mentioned before, this research mainly focuses on embedded systems for human action recognition, which requires the system to be capable of both training and prediction. In this work, we test our algorithm on two recent challenging datasets: 3D Hollywood [14] and HON4D [15]. The data format for this implementation is fixed point with 64-bit width. The accuracy of the classification was 12.2 % for the 3D Hollywood dataset and 72.40% for HON4D. For both datasets, the highest accuracy achieved was 13.3% for 3D Hollywood and 88.89% for HON4D. The highest accuracy was reached on a PC with floating-point variables and operations, whereas in this work, we used fixed-point variables. Table 1 shows the resources utilized for implementation.

TABLE I
FPGA CONSUMED RESOURCES

Target FPGA	LUT	DSP48E	BRAM
Proposed System (Zynq)	15488	81	96

V. CONCLUSION

In this work, an SoC FPGA-based ELM algorithm is presented. The proposed work discussed the appropriate circuits for both training and prediction on an embedded system. Future work could test the proposed system with more datasets and evaluate the performance of the proposed circuit under different conditions, including fixed-point variables with different widths and floating-point variables.

ACKNOWLEDGMENT

This study was supported in part by the Canada Research Chair Program, AUTO21, Networks of Centers of Excellence, the Natural Sciences and Engineering Research Council of Canada, and the Xilinx University Program.

REFERENCES

[1] G. Huang, G.-B. Huang, S. Song, and K. You, "Trends in extreme learning machines: A review," *Neural Networks*, vol. 61, pp. 32–48, 2015.

[2] J. V. Frances-Villora, A. Rosado-Muñoz, J. M. Martínez-Villena, M. Bataller-Mompean, J. F. Guerrero, and M. Wegrzyn, "Hardware implementation of real-time extreme learning machine in fpga: Analysis of precision, resource occupation and performance," *Computers & Electrical Engineering*, vol. 51, pp. 139–156, 2016.

[3] M. Van Heeswijk, Y. Miche, E. Oja, and A. Lendasse, "Gpu-accelerated and parallelized elm ensembles for large-scale regression," *Neurocomputing*, vol. 74, no. 16, pp. 2430–2437, 2011.

[4] S. Li, X. Niu, Y. Dou, Q. Lv, and Y. Wang, "Heterogeneous blocked cpu-gpu accelerate scheme for large scale extreme learning machine," *Neurocomputing*, 2017.

[5] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.

[6] Xilinx. (2017, Aug.) Expanding the all programmable soc portfolio. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc.html

[7] A. Safaei and Q. M. J. Wu, "A system-level design for foreground and background identification in 3d scenes," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 2571–2574.

[8] A. Safaei, Q. M. J. Wu, and Y. Yang, "System-on-a-chip (soc)-based hardware acceleration for foreground and background identification," *Journal of the Franklin Institute*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0016003217303666

[9] A. Safaei, Q. M. J. Wu, and T. Akilan, "System-on-chip-based hardware acceleration for human detection in 2d/3d scenes," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct 2017, Accepted.

[10] A. Safaei and Q. M. J. Wu, "Fpga implementation of the histogram of oriented 4d surface for real-time human activity recognition," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 531–536.

[11] A. Safaei, Q. M. J. Wu, and T. Akilan, "System-level design for human action recognition in 3d scenes," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct 2017, Accepted.

[12] J. E. Gentle, *Matrix algebra: theory, computations, and applications in statistics*. Springer Science & Business Media, 2007.

[13] Å. Björck, "Solving linear least squares problems by gram-schmidt orthogonalization," *BIT Numerical Mathematics*, vol. 7, no. 1, pp. 1–21, 1967.

[14] S. Hadfield, K. Lebeda, and R. Bowden, "Hollywood 3d: What are the best 3d features for action recognition?" *International Journal of Computer Vision*, pp. 1–16, 2016.

[15] O. Oreifej and Z. Liu, "Hon4d: Histogram of oriented 4d normals for activity recognition from depth sequences," in *Proc. Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2013, pp. 716–723.