An Artificial Neural Network Processor with a Custom Instruction Set Architecture for Embedded Applications

Daniel Valencia, Saeed F. Fard, and Amir Alimohammad
Department of Electrical and Computer Engineering
San Diego State University, San Diego, U.S.A.

***Abstract*—This article presents the design and implementation of an embedded programmable processor with a custom instruction set architecture for efficient realization of artificial neural networks (ANNs). The ANN processor architecture is scalable, supporting an arbitrary number of layers and number of artificial neurons (ANs) per layer. Moreover, the processor supports ANNs with arbitrary interconnect structures among ANs to realize both feed-forward and dynamic recurrent networks. The processor architecture is customizable in which the numerical representation of inputs, outputs, and signals among ANs can be parameterized to an arbitrary fixed-point format. An ASIC implementation of the designed programmable ANN processor for networks with up to 512 ANs and 262,000 interconnects is presented and is estimated to occupy 2.23 mm^2 of silicon area and consume 1.25 mW of power from a 1.6 V supply while operating at 74 MHz in a standard 32-nm CMOS technology. In order to assess and compare the efficiency of the designed ANN processor, we have designed and implemented a dedicated reconfigurable hardware architecture for the direct realization of ANNs. Characteristics and implementation results of the designed programmable ANN processor and the dedicated ANN hardware on a Xilinx Artix-7 field-programmable gate array (FPGA) are presented and compared using two benchmarks, the MNIST benchmark using a feed-forward ANN and a movie review sentiment analysis benchmark using a recurrent neural network.**

I. Introduction

A biological brain consists of billions of relatively slow elements called neurons, each of which is connected to thousands of other neurons with which they communicate by sending messages in the form of voltage spikes [1]. An artificial neural network (ANN) is an information processing paradigm that is inspired by the way a biological brain processes information. An ANN is composed of interconnected processing elements referred to as artificial neurons (ANs), which loosely model the neurons in a biological brain. In an ANN structure, the interconnected ANs are organized among input, hidden, and output layers. An ANN stores representations in the interconnections between ANs (like the synapses in the biological brain), each of which contains a value known as the weight. Similarly to biological brains, ANNs learn by example. An ANN is configured for a specific application through the learning process. The learning mechanism involves adjustments to the weights of the interconnects based on the input patterns. Therefore, instead of being programmed as in microprocessors, ANNs learn what weights to use through a process called training. After observing enough examples, neural networks can categorize new objects they have never experienced before, or at least offer a prediction. During operation, a pattern is applied to the input layer. Each AN reacts to the input data. Using a set of weighted interconnects, particular ANs in the network react the strongest when they sense a matching pattern. The response is broadcasted to the other ANs in the hidden layers and finally, the prediction is produced at the output layer.

The application domain of ANNs is broad and diverse, including pattern recognition, image classification [2], autonomous vehicles [3], and language translation with recurrent neural networks [4]. Some recent research has been focusing on the digital hardware implementation of relatively large network models for high-performance and accelerated computing, such as AlexNet [2], VGG-16 [5], and GoogLeNet [6]. Also, hardware realizations of convolutional neural networks (CNNs) have received interest [7]–[9]. Various processor-based architectures for the realization of deep neural networks (DNNs) have also been reported [10]–[16]. For example, Google's neural network processor, the Tensor Processing Unit (TPU) [17], was designed to process computationally-intensive workloads of DNNs on server farms. There has also been work presenting custom instruction set architectures for neural network processors [18]–[20].

Recent advances in memristor technology has shown promising analog implementation of ANNs [21]. Memristors can exhibit a range of programmable resistance values and are the basis for multiply-accumulate (MAC) operations employed in ANNs. The weights are encoded as memristor resistance values and the dot-products are performed automatically as currents flow through the memristor-crossbar. While more area and energy-efficient than their digital counterparts, the memristor technology is not yet mature compared to the standard CMOS, precluding their applications in practical implementations [22].

While high-performance general-purpose processors or application-specific processors for high-throughput realization of large neural networks have received considerable attention, our focus in this work is on designing a compact and programmable processor architecture with a custom instruction set architecture for area- and power-constrained embedded applications utilizing moderately-sized ANNs. Moderately-sized neural networks consist of a relatively small number of neurons, in the order of hundreds to thousands, requiring a relatively small number of parameters (in the order of a few hundred thousand) compared to large network models with hundreds of thousands to millions of parameters. The designed processor architecture supports arbitrary interconnect structures among ANs to realize both feed-forward and

The authors have made their design description publicly available at: https://github.com/dlvalencia/ANN-Processor.

dynamic recurrent neural networks (RNNs) and is scalable, i.e., supporting ANNs with arbitrary number of layers and number of ANs per layer, limited by the number of available configurable resources on the device. Moreover, the processor architecture is customizable in which the numerical representation of inputs, outputs, and signals among ANs can be parameterized to an arbitrary fixed-point format.

One of the applications of ANN models is in brain-computer interfaces (BCIs) [23], [24]. In [25], an ANN is utilized for mapping neural activities from one region of the brain and produce neural stimulation to another region of the brain, in which the ANN behaves as an artificial pathway for restoring and augmenting functions. ANNs have been utilized for training spiking neural networks (SNNs) [26], which closely resemble the spiking dynamics of biological neurons, and decoding of neural signals for prosthesis control [27], [28]. SNNs pass information among neurons by emitting action potentials, or spikes. Because SNNs simulate the voltages found in biological neurons, they are considered a prime candidate for modeling biological neurons. There have been advances in both the computational models [29], [30] as well as hardware implementation of SNNs [31], [32]. Compared to ANNs in which the precise timing of spiking activity is not inherently part of the model [33], the timing resolution of spiking neurons greatly increases the computational complexity of SNNs. The computational complexity of the SNNs can be reduced using an event-driven methodology [31], in which a global clock signal is not required and the input and output behavior of the neurons are emulated without being strictly bio-physically accurate.

Recently published neuromorphic processors implemented in analog [34] and digital [35] are based on SNNs. These processors are not tuned to one specific application and often employ online learning methods, such as spike time-dependent plasticity [36]. Compared to the mature field of ANN training, the training algorithms for SNNs are still an active area of research [37], [38]. While neuromorphic processors may be ideal for systems where real-time learning or adaptation to signal changes is required, certain applications may not be well-suited for SNNs, such as classifications of frame-based data (i.e., data that is not inherently time-dependent) [39]. The widely-employed gradient descent-based learning schemes for training ANNs make them an attractive model when pre-processing is required.

This work focuses on the design and implements of a programmable processor architecture for realizing various ANN topologies and network specifications. The rest of this article is organized as follows: The design and implementation of the programmable ANN processor is presented in Section II. Section III details the design of a dedicated reconfigurable hardware architecture for the direct implementations of ANNs. The dedicated hardware architecture is used to assess and compare the efficiency of the designed ANN processor. In Section IV, two ANN benchmarks are employed, MNIST digit recognition [40] and epileptic seizure detection [41], to quantify and compare the implementation characteristics of the designed programmable ANN processor and the dedicated reconfigurable ANN hardware on a Xilinx Artix-7 field-programmable gate array (FPGA). Finally, Section V makes some concluding remarks.

II. The Embedded Programmable ANN Processor

An ANN is based on a collection of interconnected ANs. Each connection can transmit a signal from one AN to another. Typically, ANs are aggregated into layers and signals traverse from the first (input) layer to the last (output) layer, possibly through some middle (hidden) layers. The model of the AN and an example three-layer neural network consisting of an input layer, a hidden layer, and an output layer with 2, 3, and 1 ANs, are shown in Figs. 1(a) and 1(b), respectively. The number of hidden layers, the number of ANs in each layer, and the interconnect structure among ANs can vary greatly among various ANN models. The output z_n of the n–th AN is computed by some non-linear activation function $f(y_n)$ of the sum of the weighted inputs and a bias as:

$$y_n = \sum_{i=1}^{M_n} \left[w_{in} x_i \right] + b_n, \tag{1}$$

where x_i denotes the i-th input, w_{in} denotes the interconnection weight between the i–th AN of the previous layer and the n–th AN of the current layer, M_n denotes the number of inputs to the n–th AN, and b_n denotes the bias for the n–th AN.

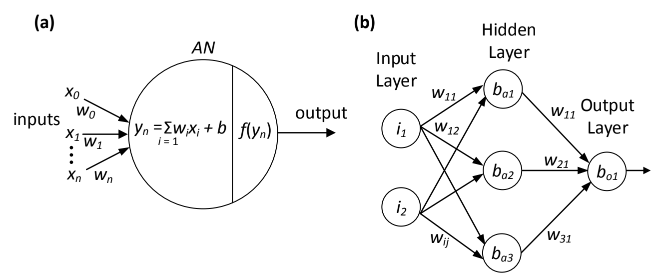

Fig. 1. (a) An artificial neuron and (b) a three-layer ANN.

Activation functions are used to model the excitation properties of biological brain neurons. By shifting the activation function to the left or right by a bias b_n, an ANN can fine tune how easy (or difficult) it is for particular ANs to exhibit an excited output state. Non-linear activation functions have been widely used in ANNs [42]. Two commonly used sigmoidal activation functions (SAFs) are the logistic sigmoid and the hyperbolic tangent (tanh) functions [43], which are defined as $f_l(y_n) = (1+e^{-y_n})^{-1}$ and $f_t(y_n) = \left(2 \times (1+e^{-2y_n})^{-1}\right) - 1$, respectively. The sigmoid and the tanh functions have bounded outputs within [0, 1] and [-1, 1], respectively. The rectified linear unit (ReLU) activation function

$$f_r(y_n) = \begin{cases} 0 & \text{if } y_n \leq 0 \\ y_n & \text{if } y_n > 0 \end{cases}$$

has also received applications in the hardware implementation of ANNs as it only requires a comparator and lookup tables (LUTs). However, because of the fixed wordlength of signals and the unbounded nature of ReLU, the output of $f_r(y_n)$ may overflow, causing erroneous values to propagate to subsequent layers of the network.

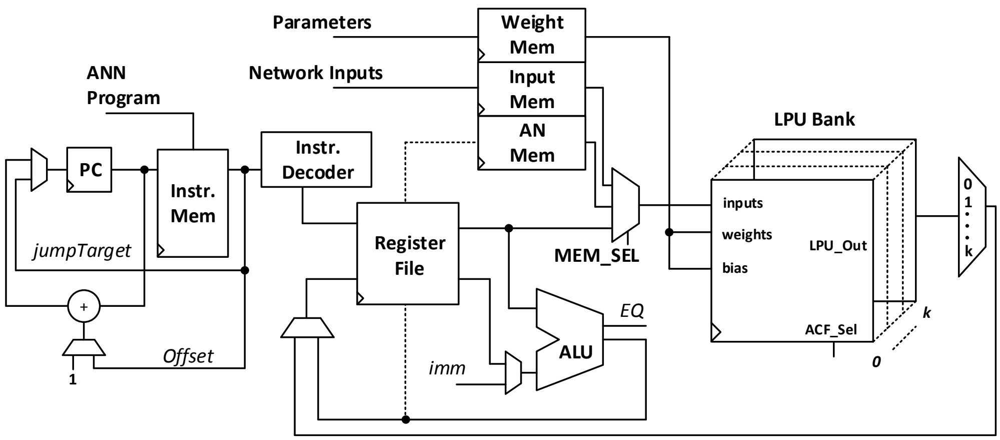

Fig. 2. The top-level microarchitecture of the proposed programmable ANN processor.

The top-level microarchitecture of the designed programmable ANN processor is shown in Fig. 2. It consists of memory units, an instruction decoder, a register file, and a layer processing unit (LPU) bank, which consists of k LPUs to perform the computation of ANs in a layer. The LPU's datapath is shown in Fig. 3, which consists of a LUT RAM to store weight values, a MAC unit, and an activation function *ACF* unit. The end-user first loads a set of instructions, weights, biases, and network inputs to the *Instr Mem*, *Weight Mem*, and *Input Mem*, respectively, after which the processor can begin executing instructions. The artificial neuron memory *AN Mem* is used to store the output of each AN in the ANN. The depth of the *AN Mem* is directly defined by the total number of ANs. The instructions, which are addressed by the program counter (PC) register, are decoded by the *Decoder* to generate the appropriate control signals for the multiplexers and registers. The *ACF* unit is realized by implementing two sigmoidal functions, sigmoid and tanh functions, and the ReLU function. For the two SAFs, we have utilized the piecewise linear approximation (PLA) technique [44]. The work in [45] has also employed PLA for approximating the non-linear activation functions, however, it utilizes a fixed activation function and a fixed number of segments and hence, used a direct transformation of y_n to $f(y_n)$ by shifting and adding constants to the values. Since for relatively accurate approximation, the number of uniform segments varies among applications and different ANN models, we design and implement a generalized realization of the PLA by parameterizing the activation functions and ranges of uniform segments. In our approach, the two SAFs are uniformly segmented within $y_n \in [-5, 5]$. As the functions are symmetric, the coefficients of the piecewise linear approximation are calculated for the input range [0, 5]. For an input $y_n < 0$, the output for the sigmoid and tanh functions are given as $f_l(y_n) = 1 - f_l(|y_n|)$ and $f_t(y_n) = -f_t(|y_n|)$, respectively. The logtanh function can be derived from the logistic sigmoid function by first squaring the exponential term e^{-y_n}, then scaling by two, and finally subtracting one from the scaled and squared quotient. However, this approach would require two approximations, one for e^{-x} and one for $\frac{1}{x}$, as opposed to only one. Moreover, because the hyperbolic tangent requires squaring the exponential term, the piecewise approximation would require more LUTs for relatively accurate representation of the coefficients. Fig. 4 shows the mean squared error (MSE) of the PLA for the logsig and logtanh functions compared to their real-valued functions over various number of uniform segments. Since the MSE decreases as the number of segments approaches eight, we choose to implement the PLA of the logsig and logtanh functions utilizing eight uniform segments. The number of chosen segments may differ for other applications.

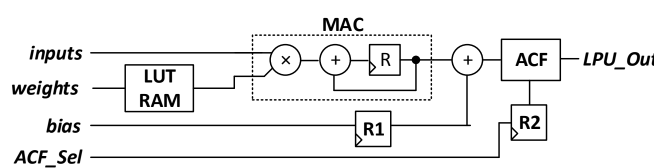

Fig. 3. The LPU datapath.

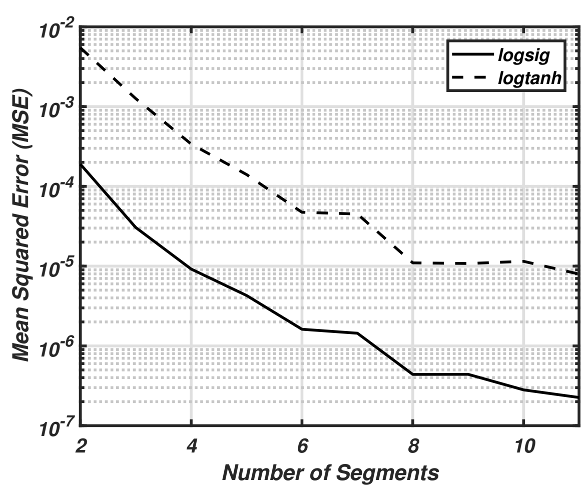

Fig. 4. The mean squared error of the piecewise linear approximation of the logistic sigmoid and hyperbolic tangent functions over varying numbers of uniform segments.

Fig. 5 shows the *ACF's* datapath. The module *Abs* passes the two's complement of input y_n if $y_n < 0$. The output of the

TABLE I
INSTRUCTION SET OF THE DESIGNED ANN PROCESSOR

Instruction	*Assembly format*	*Description*	*Instruction type*
Add	**add** `$s1 $s2 $dst`	`$dst = $s1 + $s2`	V-type
Add immediate	**addi** `$s1 $dst imm`	`$dst = $s1 + imm`	V-type
Subtract	**sub** `$s1 $s2 $dst`	`$dst = $s1 - $s2`	V-type
Subtract immediate	**subi** `$s1 $dst imm`	`$dst = $s1 - imm`	V-type
Set source	**source** `code`	`MEM_SEL = code`	N-type
Set function	**sfunc** `code`	`ACF_SEL = code`	N-type
Load weights	**lw** `$raddr $node`	`LPU_Bank[$node].weights = WeightMem[$raddr]`	N-type
Load all	**la** `$raddr`	`LPU_Bank.inputs = Mem[$raddr]`	N-type
Load all except	**lax** `$raddr $LPU`	`LPU_Bank[!$LPU].inputs = Mem[$raddr]`	N-type
Load single	**ls** `$raddr $LPU`	`LPU_Bank[$LPU].inputs = Mem[$raddr]`	N-type
Write to memory	**wm** `$LPU $waddr`	`NodeMem[$waddr] = LPU_Bank[$LPU].LPU_Out`	N-type
Write to register file	**wrf** `$LPU $waddr`	`RegFile[$waddr] = LPU_Bank[$LPU].LPU_Out`	N-type
Branch on equal	**beq** `$s1 $s2 offset`	`If $s1 == $s2, PC = PC + offset, else PC = PC + 1`	C-type
Jump	**jump** `jumpTarget`	`PC = jumpTarget`	C-type
No operation	**nop**	`No Operation`	C-type

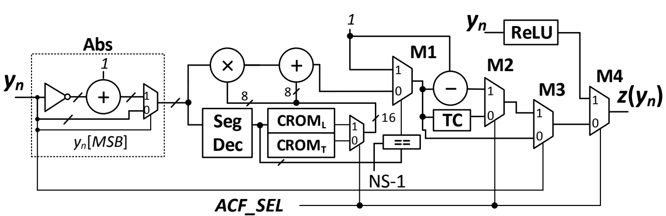

Fig. 5. The ACF datapath.

module *Abs* is then passed to the segment decoder module *SegDec*, which uses a series of comparators to determine in which segment the input signal's value $|y_n|$ lies. The output of the segment decoder is then used as an address for the read-only memories $\mathrm{CROM_T}$ and $\mathrm{CROM_L}$, which store the coefficients of the piecewise linear approximations of the tanh and sigmoid activation functions for each segment, respectively. Because both SAFs are symmetric, only the coefficients within [0, 5] are stored in the read-only memories (ROMs). Due to the relatively small number of coefficients, the ROMs are implemented using LUTs. The multiplexer on the output of the ROMs is used to select which SAF to compute, with the input select line *ACF_SEL*. If *ACF_SEL* is determined to be fixed during run-time, the synthesis tool will remove the unused ROM. The multiplier and adder compute the linear approximation $a|y_n| + b$, where the a and b coefficients are obtained using the PLA of the activation functions. If the input $|y_n|$ lies outside the range of the selected number of segments *NS*, then the function has reached saturation, either at 1 or -1 for tanh, or at 0 or 1 for sigmoid. Thus, the output of the segment decoder is compared to *NS*. If the saturation condition is not true, the saturation multiplexer *M1* passes $a|y_n| + b$. Note that for an input $y_n < 0$ for tanh, the output is $-f(|y_n|)$. Thus, the output of the saturation multiplexer is passed to the two's complement module *TC*, which is implemented using an inverter and an adder, as shown in the *Abs* block. For the sigmoid function, the output $f_l(y_n)$ for $y_n < 0$ is $1 - f_l(|y_n|)$. The output of the saturation multiplexer is thus passed to a subtractor. The output of the *TC* module or the subtractor is selected using the multiplexer *M2*. For both sigmoidal activation functions, the output depends on the sign bit of the original input y_n, which is used as the select line for multiplexer *M3*. Ultimately the input *ACF_SEL* is used to select which of the activation functions' outputs should be passed as the output.

We define a set of instruction types and formats, as shown in Fig. 6 for executing ANN operations using our application-specific programmable processor architecture. Each instruction is 28 bits: 4 bits are reserved for the operation code (*opcode*), and the remaining 24 bits are reserved for different fields, depending on the instruction type. The processor supports three different instruction types: variable instructions (V-type), network instructions (N-type), and control instructions (C-type instructions). The V-type instructions are used to add or subtract variables. The N-type instructions can be used to interact with the LPU bank, such as providing input data to LPUs and/or storing the output of the LPU bank into the *AN Mem*. Finally, the C-type instructions support conditional and unconditional branches. Table I gives the list of defined custom instructions along with their assembly formats. The `$R` symbol denotes the value stored in register `R` in the register file. The immediate values are encoded in the instructions.

The programs for the processor use the instruction set for loading weights into LPUs, applying inputs to LPUs simultaneously, and writing the LPUs' outputs into the AN memory. Conditional and unconditional branches are used to iteratively compute and store the AN outputs of each layer. The ISA also supports RNNs. The key difference is that the hidden layer outputs of the previous time step are applied to the input of the hidden layer during the current time step. As seen in Program 1, weight values are first loaded into the LPUs. For the first time step, there are no previous hidden layer outputs and the current hidden layer outputs are stored in the *AN Mem*. For the second time step and beyond, the input source is switched with the *AN Mem* after the network inputs have been loaded into the LPUs. The *LPU Bank* accumulates the weighted hidden layer outputs of the previous time step. For the example RNN in Program 1, the output layer of the RNN is only computed for the final time step, but this can be done for every time step if required. Because RNNs require values from the previous time steps, the total number of neurons that can be implemented is half of those for feed-forward ANNs. While

LPUs store the current time step values into the *AN Mem*, the previous values must also be maintained. Note that the ANNs that can be implemented on a single FPGA are limited by the amount of configurable resources, including on-chip memory blocks and the operations supported by the LPUs.

Program 1. An example RNN program using our custom-developed ISA.

```
 addi $0, $a0, 3          #0 -> 3 time steps
 add $0, $0, $a1          #time step counter
 addi $0, $c0, 1          #increment var.
 sfunc 2                  #use ReLU
 addi $0, $b0, 10         #output neuron
 addi $0, $b1, 9          #HL size
rnnStart:
 source 0                 #source is input mem.
 add $0, $0, $b2          #HL counter
HLWL:
 lw $b2, $b2              #WeightMem.b2 -> LPUBank.b2
 beq $b2, $b1, loadIMem
 add $c0, $b2, $b2        #inc. b2
 jump HLWL
loadIMem:
 la $a1                   #InMem.a1 -> LPUBank.all
 add $0, $0, $b2          #HL counter
 beq $a1, $0, writeHL
 source 1                 #Prev. HL outputs
loadPrevHL:
 la $b2                   #ANMem.b2 -> LPUBank.all
 beq $b2, $b1, zeroHL
 add $c0, $b2, $b2        #inc. b2
 jump loadPrevHL
zeroHL:
 add $0, $0, $b2          #HL counter
writeHL:
 wm $b2, $b2              #LPUBank.b2 -> ANMem.b2
 beq $b2, $b1, OLCheck
 add $c0, $b2, $b2        #inc. b2
 jump writeHL
OLCheck:
 beq $a1, $a0, OL
 add $c0, $a1, $a1        #inc. a1
 jump rnnStart
OL:
 add $0, $0, $b2          #HL counter
 lw $b0, $0               #WeightMem.b0 -> LPUBank.0
OLInputs:
 ls $b2, $0               #ANMem.b2 -> LPUBank.0
 beq $b2, $b1, writeOL
 add $c0, $b2, $b2        #inc. b2
 jump OLInputs
writeOL:
 wm $0, $b0               #LPUBank.0 -> ANMem.b0
```

Implementation of programmable processors with custom ISAs for neural network applications have been reported previously in [15], [18]–[20]. The work in [18] presents a 16-bit reduced instruction set (RISC) processor. The processor operates using a linear array of an undisclosed number of processing units (PUs). Depending on the available memory, each PU can support a number of PEs, with each PE supporting up to 64K of virtual interconnects. The defined 16 instructions provide the processing required by ANN algorithms. All instructions are one-word long with the same format consisting of a 4-bit operation code field and a 12-bit immediate/address field. Each of the supported 16 instructions require a different number of memory accesses and hence, cannot maintain single-cycle execution of instructions. To maintain single-cycle execution for most of the instructions, the authors have adopted a dual-phase, non-overlapping clocking scheme with memory accesses occurring at each clock phase. Our processor, however, does not need to leverage dual-phase clocking to maintain a single-cycle execution of instructions. Moreover, while our target application places an upper bound on the number of PEs supported, the bound with regards to the number of interconnects is limited by the available memory space, not the number of employed PEs. Unfortunately, the authors in [18] have not implemented their design on an actual device, so we cannot compare our implementation results. The ISA in [19] supports similar instructions as in our designed ISA, however, more complex instructions are supported for performing convolution layers, pooling layers, and activation functions required for acceleration of DNNs, which makes their processor architecture more complex. Note that the target application of [19] is FPGA acceleration of DNNs, whereas our ANN processor utilizes a simpler ISA for area and power-constrained embedded system applications.

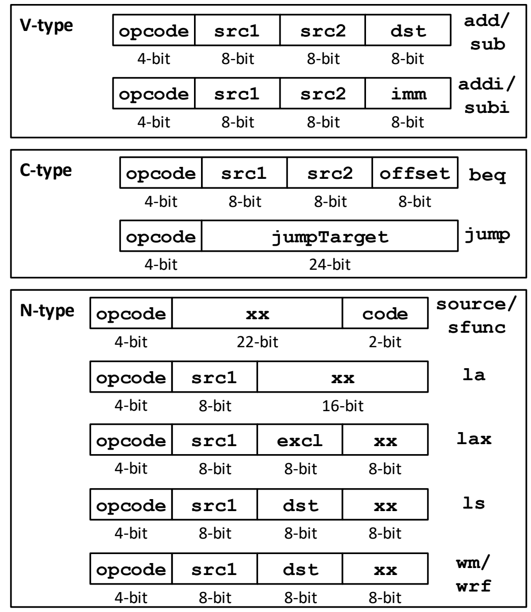

Fig. 6. Instruction types and their formats for the designed ANN processor.

The applicability of ANNs toward area- and power-constrained embedded applications thus leads to the need for such application-specific integrated processors (ASIPs) for ANN computation. The two previously reported ASIPs in [15], [20] utilize dedicated custom instruction set architectures for ANN operations. In [20], the authors utilize an architectural description language (ADL) to automate the generation of synthesizable descriptions of their hardware architectures and the required software tools (assembler, linker, and compiler) for their designed ASIP. Their ASIP is a 4-stage pipelined processor and their main processing elements are 32 multiply-and-accumulate units. Their design focuses on the implementation of multi-layer perceptrons on a Zynq FPGA using a user-defined non-linear activation function. The ANN ASIP presented in [15] contains a central processing unit for fetching and decoding the instructions from the memory. The supported activation functions are implemented using non-uniform linear approximation. A 16-word register file is utilized for both

TABLE II
THE ASIC CHARACTERISTICS AND IMPLEMENTATION RESULTS OF VARIOUS NEURAL NETWORK PROCESSORS

Work	Network	Technology	Voltage (V)	Clock (MHz)	Area (mm^2)	Throughput	Power (mW)	Power Density
[10]	DNN	65–nm	1.2	200	4.4	51.2 GOPS	141.4	32 mW/mm^2
[11]	DNN	65–nm	1.2	200	3.52	51.2 GOPS	126	35 mW/mm^2
[12]	DNN	28–nm	$0.6 \sim 0.9$	$20 \sim 400$	4.8	$410 \sim 3270$ GOPS	$3.4 \sim 20.8$	$0.66 \sim 4.33$ mW/mm^2
[13]	CNN/RNN	65–nm	$0.77 \sim 1.1$	$50 \sim 400$	16	-	$34.6 \sim 279$	$2.16 \sim 17$ mW/mm^2
[15]	FF	130–nm	–	4	0.000384	5.665 kOPS	0.153	398 mW/mm^2
Ours	FF/RNN	32–nm	1.6	74	2.23	74 MOPS	1.25	0.56 mW/mm^2

general purpose registers as well as control registers.

We have implemented our designed ANN processor for moderately-sized ANNs (up to 512 ANs with 262,000 interconnections) using the ASIC design flow. The ASIC has been implemented using the Synopsys design kit for a 32-nm CMOS process with a 1.6V supply voltage. Synthesis is performed using Synopsys Design Compiler and place-and-route is done with Synopsys IC Compiler. The memory units are implemented using the SRAM standard cells available in the Synopsys design kit. For supporting an arbitrary interconnect structure, the potential fanout of a neuron can be relatively large and hence, increasing the maximum number of supported ANs would directly increase the memory requirement and thus, the silicon area of the ANN ASIC. The chip layout of the designed ANN processor is shown in Fig. 7. The ANN processor ASIC layout is estimated to occupy an area of 2.23 mm^2 and consume 1.25 mW of power while operating at 74 MHz. Even though the maximum number of neurons and synapses, the activation functions, and the neural network operations are fixed after the chip fabrication, the ANN machine program that is loaded into the instruction memory of the processor can be readily updated to realize various neural networks. As can be seen in Fig. 7, the weight memories consume a significant portion of the silicon area, due to the processor's support for arbitrary interconnect structures. Comparing our ASIC implementation results with recently implemented neural recording and/or stimulation brain implants [46]–[48] suggests that the designed ANN processor architecture can be used for in-vivo processing of brain neural signals, such as decoding motor signals to control a prosthesis [49]. The brain-implantable recording system presented in [46] was implemented using 180-nm CMOS technology. It consumes 10.57 mm^2 of silicon area and 1.45 mW of power while the internal digital controller is operated at 60 MHz. Another neural recording system is presented in [47], and it was implemented using 130-nm CMOS technology. It consumes 45.7 mm^2 of silicon area and 13.54 mW of power while the internal digital controller is operated at 93.6 MHz. Finally, the recording and stimulation neural interface presented in [48] was implemented in 350-nm CMOS. It performs optogenetic stimulation and consumes 4.21 mm^2 of silicon area and 13.4 mW of power while operating the digital control unit at 12 MHz. It can be seen that our implemented design fits within the safe brain-implantable margins in regards to power consumption and die area. While strict power dissipation constraints limit the amount of in-vivo processing, our designed ASIC meets the tissue-safe requirements with a power density of $0.56\ \mathrm{mW/mm^2}$ [50].

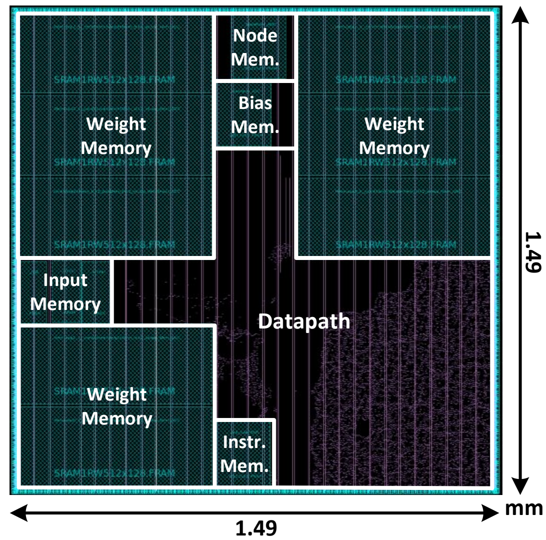

Fig. 7. ASIC layout of the designed ANN processor.

Table II gives the ASIC characteristics and implementation results of various state-of-the-art programmable neural network processors. The work in [10] presents a DNN training processor for real-time object tracking. The processor was specifically designed for high-throughput applications. The work in [11] also presents a DNN processor for object tracking with on-chip learning. Their design was optimized for processing convolutional layers and their processing elements perform MAC operations using matrix-vector multipliers and adder trees. The work in [12] implemented a reconfigurable processor for a DNN using binary and ternary weights so the architecture does not require any multipliers. The work in [13] presents a processor-implementation for CNNs and RNNs for high performance computation. The processor in [13] has been optimized for hardware acceleration of large ANNs. The work in [14] presents a DNN processor for datacenter-based applications. The DNN processor, named Project Brainwave (BW) utilizes a neural processing unit (NPU) to perform vector- and matrix-based operations. One of the key features of the BW NPU is what the authors refer to as instruction chaining, i.e., most instructions expect (and produce) input (and output) arguments. This allows a sequence of instructions in which the output of the current instruction is passed to the next instruction. This enables the micro-architecture to avoid costly memory read and write operations, thus optimizing their design for high-performance computing.

Their target platform was the Intel Stratix 10 280 FPGA, which consumes 125 W of power with a peak throughput of 35.875 TFLOPS. While these processors focused on high-throughput applications utilizing relatively large neural networks, such as CNNs and DNNs, our focus is on the design of an efficient programmable ANN processor for realizing moderately-sized ANNs used in area- and power-constrained embedded applications. The ASIP presented in [15] was also implemented in a standard 130-nm CMOS technology. Their power consumption is directly related to a significantly lower operating frequency of 4 MHz compared to our 74 MHz operating frequency. Moreover, their reported throughput is significantly less than our design's throughput. As given in Table II, our designed and implemented programmable ANN processor is ideal for relatively small to moderately-sized neural networks, commonly employed in the area- and power-constrained embedded applications, such as battery-powered wearable devices and mobile handsets. We presented one such application for the in-vivo real-time processing of the brain's neural signals in [51]. Because the designed ANN processor meets the brain tissue's stringent limitations of the silicon area and power consumption for implantable devices, it can be utilized for the in-vivo real-time processing of neural signals, which can then be used to control a prosthetic arm [52]. While the other ANN realizations given in Table II offer greater computational throughputs, their high power consumptions make them infeasible for power-constrained embedded applications.

III. Dedicated reconfigurable ANN hardware

In order to verify and assess the efficiency of the designed programmable ANN processor, we have designed and implemented a dedicated hardware architecture for the direct implementation of ANNs. Using our custom-developed MATLAB interpreter, a given ANN specification is directly converted into its equivalent Verilog HDL description. This allows a dedicated hardware architecture to be readily developed for an arbitrary ANN topology. While the generated Verilog description is for the realization of the specified ANN only, the designed and implemented programmable processor can be utilized for realizing an arbitrary ANN specification by updating a new ANN program and a new set of values for the weight and activation function parameters. The dedicated hardware architecture can be reconfigured to support an arbitrary ANN configuration by specifying the number of layers, the number of ANs per layer, the interconnect structure, and the fixed-point format of signals. The principal processing elements (PEs) of an ANN are the ANs. An AN calculates the sum of the weighted inputs according to Equation (1) and computes the activation function value based on the calculated weighted sum. Because the number of inputs to a particular AN may vary among applications, we have designed a fully-parameterizable datapath for the AN. Since the target embedded devices typically have limited computational resources, for a compact implementation of the ANs, we employ the resource sharing technique, as shown in Fig. 8, to greatly reduce the number of PEs required to compute the sum of weighted inputs. Two shift registers, which support parameterizable depths, receive the inputs and pass an input-weight pair to the registered multiply-and-accumulate *MAC* unit serially. The control unit *CU* is implemented using a finite state machine (FSM) and counts the number of inputs that have been given to *MAC*. Once all of the weighted inputs have been accumulated, the register *MR* is enabled to pass the weighted inputs to the bias adder. Finally, the biased and weighted sum is passed as an input to the activation function module *ACF*, which can be configured to support the ReLU, sigmoid, or tanh activation functions, and its output is written into the output register *OR*. Utilizing resource sharing for a compact realization of ANNs, the output will be ready after a latency of $M_n + 2$ clock cycles, where M_n denotes the number of inputs to the n–th AN. The control unit *CU* asserts the hand-shaking signal *Ready*, which informs the main controller that the output of an AN is available. The control unit also receives a control signal *start* from the main controller, which initializes the process of accumulating the weighted inputs, as well as resetting the output register of the *MAC* unit.

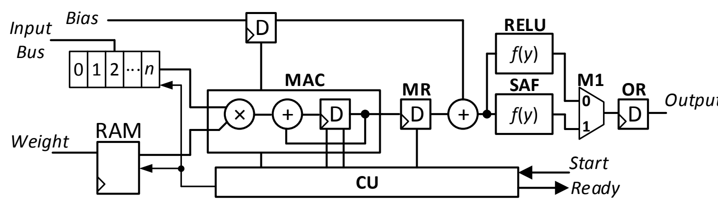

Fig. 8. Datapath of an artificial neuron utilizing resource-sharing for compact implementation.

Table III gives the characteristics and implementation results of a 20-input AN using 20-bit inputs, 12-bit weights, and a 32-bit accumulator when utilizing either a SAF or the ReLU on a Xilinx Artix-7 FPGA. Each of the SAFs is implemented using 8 segments. The coefficients of the piecewise linear approximations are stored in the (*WI*, *WF*) = (1, 7) fixed-point format using one sign bit for the integer part and 7 bits for the fractional part. The output of the multiplier uses the larger number of the *WI* and *WF* bits to avoid overflow errors. For example, if the input format is (6, 14) and the coefficients are stored in the (1, 7) format, the intermediate signals would be represented in the (6, 14) format. For the sigmoidal activation functions, the output signal is always bounded between -1 and 1 and thus, the output's *WI* can be represented by using only two bits.

TABLE III
Characteristics and implementation results of a 20-input artificial neuron using 20-bit inputs, 12-bit weights, and a 32-bit accumulator on a Xilinx Artix-7 FPGA

ACF	Regs. (%)	LUTs. (%)	DSP48s. (%)	Freq. (MHz)
Sigmoidal	240 (0.09)	198 (0.15)	3 (0.71)	173
ReLU	104 (0.04)	80 (0.06)	1 (0.14)	394

Fig. 9 shows the block diagram of the designed dedicated ANN hardware, which supports a parameterizable number of ANs in the input and output layers, variable number of hidden layers, and variable number of ANs per hidden layer. The number of network inputs and network outputs is also

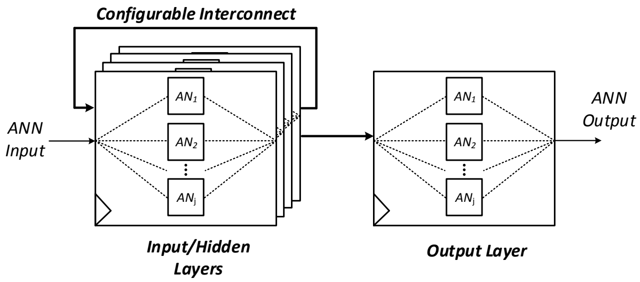

Fig. 9. Block diagram of the dedicated reconfigurable ANN architecture.

parameterizable and can be specified by the user. The fixed-point numerical representation of input, intermediate, and output signals of an ANN and also its weight and bias values can be parameterized. The dedicated ANN hardware can be reconfigured to support an arbitrary interconnect structure among ANs, which allows modeling both classical feed-forward neural networks as well as dynamic recurrent networks. The values of weights and biases are stored in a register bank. The activation function of each individual layer can be chosen from the three designed activation functions. While our dedicated ANN hardware architecture is fully parameterizable, after synthesizing the design and implementing the ANN hardware on a target device, the architecture cannot be altered to realize a different ANN and is hence referred to as a dedicated hardware. Note that in our dedicated ANN hardware, the ANN parameters, i.e., weight and bias values, are stored in on-chip memory units (i.e., using BRAMs and LUTs on FPGA devices and SRAM macro cells on ASICs) rather than in off-chip memory modules. Therefore, the size of ANNs that can be realized is directly proportional to the total number of configurable resources and storage elements available on the target device. Utilizing on-chip storage elements, however, removes the need for off-chip memory and eliminates the memory bandwidth bottleneck.

IV. Design Verification and Benchmark Results

The design flow for the implementation of the programmable ANN processor is as follows: (1) The instruction set architecture, which includes the required operations, the register set, and the assembly and machine instruction sets, is defined; (2) The microarchitecture of the processor is designed and described in Verilog HDL. The functional verification of the custom microarchitecture is performed using the Xilinx Vivado design suite; (3) For a given ANN, an assembly program is written and translated into its equivalent machine-level instructions using a custom developed interpreter. For fully-connected feed-forward ANNs, an interpreter is developed to translate the ANN description into its equivalent assembly code and subsequently, its machine-level instructions; (4) Functional verification is performed by simulating the programmable ANN processor on a Xilinx Artix-7 FPGA using the Vivado design suite. The ANN outputs are verified using a custom-developed graphical user interface for various benchmarks; (5) After functional verification, the ASIC flow begins with synthesizing the ANN processor description in Verilog HDL using Synopsys Design Compiler; (6) Synopsys IC Compiler is then used to perform placement and routing of the synthesized netlist; (7) After static timing analysis of the post-placed and routed design, the netlist is again simulated using Synopsys VCS for verification. To verify the designed programmable ANN processor and the dedicated reconfigurable ANN hardware and compare their characteristics and implementation results on a Xilinx Artix-7 FPGA, we utilize two ANN benchmarks.

We implemented the handwritten digit recognition benchmark using the MNIST dataset [40]. Fig. 10 shows the laboratory setup for verifying the functionality of the designed ANN architectures. The dataset is provided as 28×28 pixel grayscale

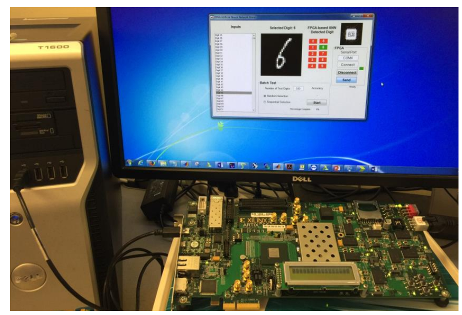

Fig. 10. The laboratory setting for testing the programmable ANN processor and the dedicated reconfigurable ANN architecture.

images and the target outputs are provided as digits denoting the number shown in the image. To provide the network with the entire image, the rows of the image are concatenated to form a 1×784 image vector. The target data matrix included in the dataset provides the correct digit, represented in decimal, for each corresponding column test vector. However, due to the nature of the bounded ACFs, we have redefined the target data as 10-row column vectors. Rows 1 through 9 of the target matrix columns refer to a recognized digit 1 through 9, respectively, and the tenth row refers to a recognized digit 0. Thus, the columns of the target data matrix have a single 1 and the rest are 0s. For the network implemented using the tanh SAF, the 0s are mapped to -1 to match the lower bound of the activation function output. Some pre-processing on the input data is also performed, such as mapping the input test vectors to values between 0 and 1 for the sigmoid, and between -1 and 1 for the tanh activation function, to reduce input wordlengths and avoid using overly large weight and bias values. We have chosen a four layer model given as 784–20–15–10, where 784 is the number of inputs, and 20, 15, and 10 are the number of ANs per layer, respectively. To obtain the optimal values of weights and biases, we use MATLAB's neural network toolbox [53] to train the handwritten digit recognition ANN. Since the ANN training for calculating the values of weights and biases is performed offline, one can utilize various training algorithms or machine learning frameworks, such as Caffe [54], TensorFlow [55], and PyTorch [56]. There have been considerable improvements in neural network training methodologies, from reducing the memory requirements of learned parameters via binarization [57], [58] to normalization

TABLE IV
CHARACTERISTICS AND IMPLEMENTATION RESULTS OF THE DESIGNED ANN ARCHITECTURES ON A XILINX ARTIX-7 FPGA

Design	Benchmark/ Network topology	LUTs. (%)	Regs. (%)	BRAMs. (%)	DSP48s. (%)	Frequency (MHz)	Latency (Clock cycles)
Dedicated ANN Hardware	MNIST 784–20–15–10	62695 (46.58)	206405 (76.67)	10 (2.74)	135 (18.24)	133	819
Dedicated RNN Hardware	Sentiment analysis (SA) 16–40–1 (50 time steps)	17547 (12.97)	35894 (13.33)	1.5 (0.41)	122 (16.49)	156	2800
ANN Processor	General-purpose FF/RNN (1024 ANs)	5064 (3.76)	553 (0.21)	20 (5.34)	30 (4.05)	62	MNIST – 1728 SA – 17285

approaches that significantly reduce the convergence time for training [59]. The weights are represented in the (5,11) fixed-point format. The processor's instruction memory is loaded with an 82-instruction program to execute the MNIST operations. Fig. 11 shows the testing accuracy of the MNIST ANN over various number of SAF segments. One can see that using the logsig SAF results in a greater testing accuracy. As stated in Section II, the preferred activation function can vary among applications. Nevertheless, the designed programmable ANN processor supports the logsig, logtanh, and the ReLU activation functions.

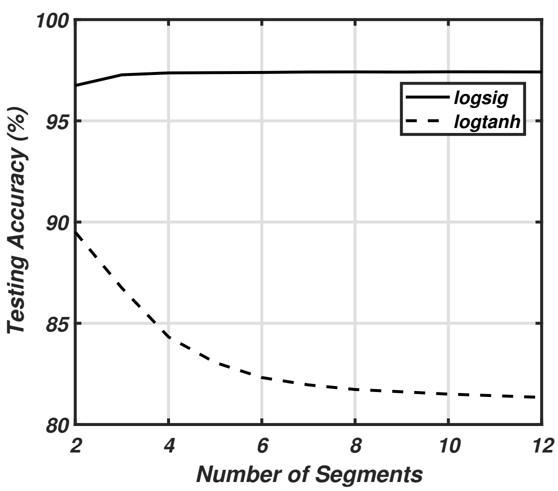

Fig. 11. The testing accuracy of the MNIST ANN using the logsig activation function and tanh activation functions over various number of SAF segments.

We have also implemented a RNN for sentiment analysis of reviews from the internet movie database [60]. The RNN attempts to predict whether a review was positive or negative for a particular movie. We utilized a three-layer RNN model with 40 recurrent ANs with tanh activation functions and one output AN with the sigmoid activation function. The input to the RNN is a word vector of 16 elements and each review consists of 50 word vectors. After processing the last word vector, the sentiment is predicted by the output layer neuron. The network parameters, such as the word embeddings, which are numerical vectors that encode each word in the dataset vocabulary, the weights, and the bias values, were obtained using Tensorflow and Python and then converted into the fixed-point numerical representation. The assembly program to execute the sentiment analysis RNN consists of 79 instructions to predict the sentiment of each review. Despite a relatively small RNN of only 41 neurons, the network correctly predicts 80.66% of the 25,000 testing reviews.

Table IV gives the characteristics and implementation results of the dedicated reconfigurable ANN and RNN hardware and the processor architecture on a Xilinx Artix-7 FPGA. Because the dedicated hardware architecture is tied to a specific ANN/RNN, we have listed the benchmark network as well as the network topologies. The designed ANN processor supports up to 1024 arbitrarily connected ANs and has an LPU Bank with 10 LPUs. The latency in Table IV refers to the number of clock cycles required to execute the given benchmark. Dedicated hardware architectures for the MNIST and sentiment analysis (SA) benchmarks compute the network outputs in 6.1 μs and 17.9 μs, respectively. For our ANN processor, the latency is directly related to the number of operations executed in the program. For a fully connected feed-forward ANN, the number of operations can be given as $\sum_{i=2}^{\eta}\left[S_{i-1}(\lceil(S_i/k)\rceil)+2(S_i)+\lceil(S_i/k)\rceil\right]+S_\eta$, where η denotes the number of layers in the network, S_i denotes the number of inputs or ANs in the i-th layer, k denotes the number of LPUs being used, and $\lceil\cdot\rceil$ denotes the ceiling operator. For RNNs, the latency can be given as $\sum_{i=2}^{\tau}\left[N_{i-1}(\lceil(S_i/k)\rceil)+2(S_i)+\lceil(S_i/k)\rceil\right]+S_\eta$, where N denotes the number of inputs to the recurrent layer at a particular time step i, and τ denotes the number of time steps. The ANN processor computes the MNIST and SA network outputs after 27.8 μs and 270 μs, respectively. For a fair comparison between the MNIST and SA dedicated ANN hardware architectures, we compare their power and energy consumptions while running at 100 MHz. The MNIST and SA dedicated architectures consume 806 mW and 225 mW of power, respectively. Given their execution times, the MNIST and SA architectures have energy consumption rates of 6.6 μJ and 6.3 μJ, respectively. The processor consumes 235 mW of power while running at 62 MHz, and has energy consumption rates of 6.53 μJ and 65.5 μJ for the MNIST and SA programs, respectively. While the dedicated ANN hardware architectures can execute their respective benchmarks considerably faster, the processor architecture can support arbitrary ANNs by changing their programs, while using significantly smaller silicon area. For very small ANNs, the dedicated ANN hardware may be more energy efficient, however, this requires a larger silicon area for a dedicated hardware that is fixed for a specific network after implementation.

Recent research has also focused on the FPGA implementation of SNNs [61], [62]. While the flexibility of SNNs make them an attractive design choice for realization on FPGAs, the employed neuron models, depending on their level of biophysical accuracy, can result in a greater reconfigurable resource utilization. For example, [61] and [62] both present SNN hardware architectures supporting 1024 and 1440 neurons, respectively, on Virtex FPGAs. However, their reconfigurable resource utilization is significantly larger than that of our proposed ANN processor. The design in [61] consumes 19397 (9%) LUTs, 32420 (15%) registers, 264 (81%) BRAMs, and 16 (8%) DSP48s. The design in [62] consumes 55884 (37%) LUTs, 48502 (16%) registers, 392 (91%) BRAMs, and 408 (53%) DSP48s. While some applications, which require time-insensitive processing, may employ SNNs, tasks such as classification or pattern recognition can be efficiently realized using the designed ANN processor with significantly fewer resources.

V. Conclusion

This article presented a programmable processor with a custom instruction set architecture for the efficient realization of artificial neural networks (ANNs). A dedicated reconfigurable hardware for the direct implementation of ANNs was also presented. The ANN processor and the dedicated ANN hardware both support various ANNs with an arbitrary number of layers, number of artificial neurons (ANs) per layer, and arbitrary interconnect structures, including feed-forward and recurrent networks. The functionality and implementation results of both designs on a Xilinx field-programmable gate array (FPGA) were assessed and compared using two ANN benchmarks. The ASIC implementation results of the designed ANN processor confirms that our processor occupies smaller silicon area compared to the other state-of-the-art processors and consumes significantly less power. The designed programmable processor can be effectively used in area- and power-constrained embedded applications utilizing moderately-sized ANNs.

Acknowledgment

This work was supported by the Center for Neurotechnology (CNT), a National Science Foundation Engineering Research Center (EEC-1028725).

References

[1] S. Herculano-Houzel, "The human brain in numbers: a linearly scaled-up primate brain," *Frontiers in Human Neuroscience*, vol. 3, p. 31, 2009.

[2] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[3] M. Bojarski, et al., "End to end learning for self-driving cars," *Computing Research Repository*, 2016.

[4] I. Sutskever, O. Vinyals, Q. V. Le, "Sequence to sequence learning with neural networks," in *Neural Information Processing Systems*, 2014, pp. 3104–3112.

[5] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the devil in the details: Delving deep into convolutional nets," *arXiv preprint arXiv:1405.3531*, 2014.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1 – 9.

[7] L. Huimin, X. Fan, L. Jiao, W. Cao, X. Zhou, L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *International Conference on Field Programmable Logic and Applications*, 2016, pp. 1 – 9.

[8] S. Venieris, C. Bouganis, "FPGAconvnet: A framework for mapping convolutional neural networks on FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines*, 2016, pp. 40 – 47.

[9] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, and S. Song, "Going deeper with embedded FPGA platform for convolutional neural network," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26 – 35.

[10] D. Han, J. Lee, J. Lee, S. Choi, and H.-J. Yoo, "A 141.4 mw low-power online deep neural network training processor for real-time object tracking in mobile devices," in *Proceedings of International Symposium on Circuits and Systems*, 2018.

[11] D. Han, J. Lee, J. Lee, and H. Yoo, "A low-power deep neural network online learning processor for real-time object tracking application," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–11, 2018.

[12] S. Yin, P. Ouyang, J. Yang, T. Lu, X. Li, L. Liu, and S. Wei, "An energy-efficient reconfigurable processor for binary- and ternary-weight neural networks with flexible data bit width," *IEEE Journal of Solid-State Circuits*, pp. 1–17, 2018.

[13] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE International Solid-State Circuits Conference*, 2017, pp. 240–241.

[14] J. Fowers et al, "A configurable cloud-scale dnn processor for real-time AI," in *International Symposium on Computer Architecture*, 2018, pp. 1–14.

[15] J. Rust and S. Paul, "Design and implementation of a neurocomputing ASIP for environmental monitoring in WSN," in *IEEE International Conference on Electronics, Circuits, and Systems*, 2012, pp. 129–132.

[16] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[17] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[18] P. Treleaven, M. Pacheco, and M. Vellasco, "VLSI architectures for neural networks," *IEEE Micro*, vol. 9, no. 6, pp. 8–27, 1989.

[19] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *ACM/IEEE International Symposium on Microarchitecture*, 2016, pp. 1 – 12.

[20] D. Rakanovic and R. Struharik, "Implementation of application specific instruction-set processor for the artificial neural network acceleration using LISA ADL," in *IEEE East-West Design & Test Symposium*, 2017, pp. 1–6.

[21] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[22] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels, "Memristor for computing: Myth or reality?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 722–731.

[23] C. Pandarinath et al., "High performance communication by people with paralysis using an intracortical brain-computer interface," *eLife*, vol. 6, pp. 1–27, 2017.

[24] J. R. Wolpaw and D. J. McFarland, "Control of a two-dimensional movement signal by a noninvasive brain-computer interface in humans," in *Proceedings of the National Academy of Sciences*, 2004, pp. 17 849–17 854.

[25] R. P. Rao, "Towards neural co-processors for the brain: combining decoding and encoding in brain-computer interfaces," *Current Opinion in Neurobiology*, vol. 55, pp. 142 – 151, 2019.

[26] E. Fetz, "Dynamic neural network models of sensorimotor behavior," in *The Neurobiology of Neural Networks*. MIT Press, 1993, ch. 7, pp. 165–190.

[27] M. Velliste, S. Perel, M. C. Spalding, A. S. Whitford, and A. B. Schwartz, "Cortical control of a prosthetic arm for self-feeding," *Nature*, vol. 453, no. 7198, p. 1098, 2008.

[28] L. R. Hochberg et al., "Reach and grasp by people with tetraplegia using a neurally controlled robotic arm," *Nature*, vol. 485, no. 7398, p. 372, 2012.

[29] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[30] ——, "Which model to use for cortical spiking neurons?" *IEEE Transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[31] P. A. Merolla et al., "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[32] C. Frenkel, M. Lefebvre, J. Legat, and D. Bol, "A 0.086-mm^2 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 1, p. 145, 2019.

[33] R. Brette et al., "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, no. 3, pp. 349–398, 2007.

[34] G. Indiveri, F. Corradi, and N. Qiao, "Neuromorphic architectures for spiking deep neural networks," in *2015 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2015, pp. 4–2.

[35] C. Frenkel, J.-D. Legat, and D. Bol, "Morphic: A 65-nm 738k-synapse/mm^2 quad-core binary-weight digital neuromorphic processor with stochastic spike-driven online learning," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 5, pp. 999–1010, 2019.

[36] G.-q. Bi and M.-m. Poo, "Synaptic modification by correlated activity: Hebb's postulate revisited," *Annual review of neuroscience*, vol. 24, no. 1, pp. 139–166, 2001.

[37] S. R. Kheradpisheh and T. Masquelier, "S4nn: temporal backpropagation for spiking neural networks with one spike per neuron," *arXiv preprint arXiv:1910.09495*, 2019.

[38] J. Göltz, A. Baumbach, S. Billaudelle, O. Breitwieser, D. Dold, L. Kriener, A. F. Kungl, W. Senn, J. Schemmel, K. Meier *et al.*, "Fast and deep neuromorphic learning with time-to-first-spike coding," *arXiv preprint arXiv:1912.11443*, 2019.

[39] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: opportunities and challenges," *Frontiers in neuroscience*, vol. 12, p. 774, 2018.

[40] Y. Lecun, C. Cortes, C. Burges, *The MNIST Database*, available at `yann.lecun.com/exdb/mnist/`.

[41] 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," *Physical Review E*, vol. 64, no. 6, pp. 1–8, 2001.

[42] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303–314, 1989.

[43] N. Gershenfeld, *The Nature of Mathematical Modeling*. Cambridge University Press, 1999.

[44] J. M. Muller, *Elementary Functions: Algorithms and Implementation*. Birkhauser, 2006.

[45] H. Amin, K. M. Curtis, B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," in *Proceedings of IEEE Circuits, Devices, and Systems*, 1997, pp. 313 – 317.

[46] C. M. Lopez et al., "An implantable 455-active-electrode 52-channel CMOS neural probe," *IEEE Journal of Solid State Circuits*, vol. 49, no. 1, pp. 248–261, 2014.

[47] ——, "A neural probe with up to 966 electrodes and up to 384 configurable channels in 0.13-μm SOI CMOS," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 3, pp. 510–522, 2017.

[48] R. Ramezani et al., "On-probe neural interface ASIC for combined electrical recording and optogenetic stimulation," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, no. 3, pp. 576–588, 2018.

[49] S. Micera et al., "Decoding of grasping information from neural signals recorded using peripheral intrafascicular interfaces," *Journal of NeuroEngineering and Rehabilitation*, vol. 8, no. 1, pp. 1–10, 2011.

[50] S. Kim, P. Tathireddy, R. A. Normann, and F. Solzbacher, "Thermal impact of an active 3-D microelectrode array implanted in the brain," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 15, no. 4, pp. 493–501, 2007.

[51] D. Valencia, J. Thies, and A. Alimohammad, "Frameworks for efficient brain-computer interfacing," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 6, pp. 1714–1722, 2019.

[52] M. Kocaturk, H. O. Gulcur, and R. Canbeyli, "Toward building hybrid biological/in silico neural networks for motor neuroprosthetic control," *Frontiers in Neurorobotics*, vol. 9, p. 8, 2015.

[53] M. Beale, M. Hagan, H. Demuth, *Neural Network Toolbox User's Guide*, MathWorks, 2017.

[54] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM international conference on Multimedia*, 2014, pp. 675–678.

[55] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, "Tensorflow: a system for large-scale machine learning." in *Symposium on Operating Systems Design and Implementation*, vol. 16, 2016, pp. 265–283.

[56] N. Ketkar, "Introduction to pytorch," in *Deep learning with python*. Springer, 2017, pp. 195–208.

[57] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[58] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 4107–4115.

[59] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[60] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*. Association for Computational Linguistics, 2011, pp. 142–150.

[61] W. Luk, D. Thomas, "FPGA accelerated simulation of biologically plausible spiking neural networks," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 45–52.

[62] D. Pani *et al.*, "An FPGA platform for real-time simulation of spiking neuronal networks," *Frontiers in Neuroscience*, vol. 11, no. 90, pp. 1–13, 2017.

Daniel Valencia is a Research Assistant working in the VLSI Design and Test Laboratory in the Department of Electrical and Computer Engineering at the San Diego State University. He is currently pursuing the Ph.D. degree in Electrical and Computer Engineering at the University of California, San Diego, and the San Diego State University. His research interests include field-programmable gate arrays, brain-computer interfacing, and VLSI architectures for neural signal processing.

Saeed Fouladi Fard received the M.Sc. degree in Electrical Engineering from the University of Tehran, Iran, in 2003, and the Ph.D. degree in Electrical and Computer Engineering from the University of Alberta, Canada, in 2009. Currently he is a Principal Engineer at Eidetic Communications Inc., a company that he co-founded in 2016. Since 2008, he has been working as a digital design engineer at Ukalta Engineering Inc., Rad3 Communications, PMC-Sierra (now Microchip) and Eidetic Communications Inc. His work on SerDes and error control codes are parts of several VLSI chipsets used by major Internet companies. His research interests include data compression and encryption, machine learning, high-performance computing, error control coding, high-speed SerDes, digital communications, and efficient hardware computation techniques.

Amir Alimohammad is an Associate Professor in the Electrical and Computer Engineering Department at the San Diego State University. He was the Co-Founder and Chief Technology Officer of Ukalta Engineering in Edmonton, Canada, from 2009-2011. He obtained a Ph.D. degree in Electrical and Computer Engineering from the University of Alberta in Canada. His research interests include digital VLSI design, brain-computer interfacing, and wireless communication.