A Unified Parallel CORDIC-based Hardware Architecture for LSTM Network Acceleration

Nadya A. Mohamed, *Student Member, IEEE,* and Joseph R. Cavallaro, *Fellow, IEEE*

Abstract—Deep Neural Networks (DNNs) have recently become the standard tool for solving various practical problems in a wide range of applications with state-of-the-art performance. Recurrent Neural Networks (RNNs) such as Long Short-Term Memory (LSTM) are a subset of DNNs with fully connected single or multi-layer networks. The complex neurons and internal states of LSTM networks enable them to build a memory of events, making them ideal for time series applications. Despite the great potential of LSTM networks, their heterogeneous operations and computational resource requirements create a vast gap when it comes to the fast processing time required in real-time applications using low-power, low-cost edge devices. This work proposes a novel hardware architecture that combines serial-parallel computation with matrix algebra concepts and efficient low-power computer arithmetics for LSTM network acceleration. The hardware is based on a systolic ring of outer-product-based processing elements (PEs) and a reusable single activation function block (AFB). PEs and AFB are implemented using the coordinate rotation digital computer algorithm (CORDIC) in the linear and hyperbolic modes. Unlike most approaches, the proposed hardware can be configured to perform recurrent and non-recurrent fully connected layers (FC) computations, making it suitable for various low-power edge applications. The architecture is validated on the Xilinx PYNQ-Z1 development board using an open-source time series dataset. The implemented design achieves $114\mu s$ average latency and $1.8GOPS$ throughput. The proposed design's low latency and $0.438W$ power consumption makes it suitable for resource-constrained edge platforms.

Index Terms—Recurrent neural networks, long short-term memory, serial-parallel computation, systolic, fixed-point arithmetic, CORDIC, FPGA, accelerator.

1 Introduction

The increasing advances in science and technology and the availability of a vast amount of training data and computing power have enabled deep neural network models to excel in solving various practical problems with state-of-the-art performance [1], [2], [3]. Recurrent Neural Networks (RNNs) are a subset of DNNs capable of handling long-term dependencies, making them useful for sequential data processing such as dynamical system control, speech recognition, and natural language processing [4], [5], [6]. Unlike Convolutional neural networks (CNNs), which use filters, RNNs are fully connected single or multi-layer networks with complex neurons and internal states. The prediction accuracy of RNNs is further improved by introducing gating units to let information through the network optionally and build a memory of time dependencies. Long Short-Term Memory and Gated Recurrent Unit (GRU) are the most popular gated variants of RNNs [7]. LSTM networks were around 21% of Google's tensor processing units (TPUs) deep learning training workload in 2019 [8]. Specifically, LSTM networks addressed time-series applications targeting small to medium Internet-of-Things (IoT) and edge devices with limited computing and memory resources.

The high accuracy of RNNs comes at the cost of increased computational and memory requirements due to the high dimensionality of input data and the number of computations that need to be performed. Cloud computing is a common approach to meeting the requirements of RNNs. This approach requires moving the data from the data source [e.g., IoT sensors and smartphones] to a centralized location in the cloud. Such a solution introduces several challenges. Sending data to the cloud for inference incurs additional propagation delays from the network, leading to failure to satisfy the end-to-end low-latency requirements for real-time interactive applications. In addition, unloading data from the sources to the cloud introduces scalability issues in network resource utilization, especially when not all the data from all the resources are needed. Besides, uploading sensitive information to the cloud and how the cloud or applications will use these data risks privacy.

Providing computational abilities close to the end devices through edge computing is a viable solution to meet the earlier latency, scalability, and privacy challenges. Many research efforts have focused on using CNNs for edge computing, while less has been reported for RNNs. Specialized Field-Programmable Gate Array (FPGA) [9], [10] and Application-Specific Integrated Circuit (ASIC) [11], [12], [13] accelerators for low power CNN's inference have been proposed. However, these specialized architectures cannot directly be utilized to accelerate RNNs inference. The densely connected layers with large memory footprints, the heterogenous computing patterns, and the necessity of storing and regularly updating the internal states in RNNs make their acceleration more difficult, motivating novel algorithmic and architecture solutions.

This work presents a novel unified parallel CORDIC-based solution for accelerating recurrent LSTM networks. The proposed solution combines serial-parallel computa-

• *N.A. Mohamed and J.R. Cavallaro are with the Department of Electrical and Computer Engineering, Rice University, Houston, TX, 77005.*
E-mail: nam7@rice.edu; cavallar@rice.edu
This work was supported in part by the US NSF under grants CNS-2016727, and CNS-1827940, for the "PAWR Platform POWDER-RENEW"

Manuscript received September 18, 2022.

tion with matrix algebra concepts and coordinate rotation digital computer algorithm to enhance the accuracy and throughput of LSTM inference. A systolic ring of outer-product-based processing elements and a single reusable activation function block is adopted in the architecture. The outer product generates and accumulates partial sums in parallel, eliminating data dependencies and increasing hardware utilization and system throughput. In addition, the single activation function block performs nonlinear computations for the whole network. Given the CORDIC algorithm's ability to perform various computing tasks using two fundamental operations (shift and add), it used to implement both PEs and AFB. The CORDIC implementation of the activation function block allows configurable activation functions (sigmoid/tanh) to maximize hardware utilization. In addition, the CORDIC implementation of the PEs makes the proposed solution generic for FPGA and ASIC platforms. Furthermore, the generic unified computing kernel can be configured to perform recurrent and non-recurrent fully connected computations, making it suitable for many IoTs and edge applications. The major contributions of this work are summarized as follows:

- A hardware architecture that combines serial-parallel computation with matrix algebra concepts to form a systolic ring of outer-product-based processing elements and a single reusable activation function block. The proposed hardware architecture reduces data dependency, permits pipelining, and increases hardware utilization and system throughput.
- A CORDIC-based implementation of the processing elements and activation function block. The CORDIC algorithm linear mode is employed for the PE MAC implementation. In addition, it is used together with the hyperbolic mode to implement the configurable activation function block. The CORDIC-based implementation makes the proposed solution generic for FPGA and ASIC platforms.
- A unified computing kernel with the ability to perform both recurrent and non-recurrent fully connected layer computations to improve hardware utilization and support various applications.
- Experimental validation of the proposed hardware architecture on resource-constrained Xilinx PYNQ-Z1 development board using an Autoencoder-LSTM network presented in [34], [42]

The rest of this article is organized as follows. Section II presents the background of LSTM networks and reviews the existing accelerator designs. Section III describes the proposed accelerator design. Section IV details the architecture implementation. Section V discuss experimental results, and Section VI concludes the paper.

2 Background and Preliminaries

2.1 LSTM Recurrent Neural Networks

Long short-term memory networks (LSTM) are a special kind of recurrent neural network (RNNs) capable of learning long-term dependencies, making them suitable for time series analysis. They have the form of a chain of repeating modules called LSTM cells, shown in Fig. 1. Each LSTM cell consists of layers of neural networks, internal states, and point-wise operations. The key to LSTM networks is the cell-state, c_t, which could be viewed as the extracted information from the input sequence at each time step. The LSTM cell can add or remove information to the cell state using Gate structures. Gates are ways to let information through optionally. They are composed of sigmoid and hyperbolic tangent neural net layers and point-wise operations. The LSTM cell has three main gates; forget, input, and output. Forget gate determines the fraction of history information to be forgotten by multiplying the value of the cell-state, c_t, by a number between 0 (delete) and 1 (keep everything). The multiplication value is determined by the current input, x_t, and the LSTM cell hidden-state from the previous time step, h_{t-1}. The input gate has two parts; the tanh layer, which creates a vector of new candidate values, u_t, and the sigmoid layer, which decides the amount of new candidates to be added to c_t. The LSTM cell hidden-state and also the LSTM cell's output, h_t, is a manipulated version of c_t. The cell-state, c_t, is first passed through a tanh layer to push the values between -1 and 1, then multiplied by a number between 0 (no outputs) and 1 (preserve output) generated using the output gate structure. The size of the LSTM cell is defined by the number of elements in the hidden state, h_t, and the number of input features per time step (number of features in x_t). The computations in a single LSTM cell with n hidden-state units and m-dimensional input features are described using the following set of equations:

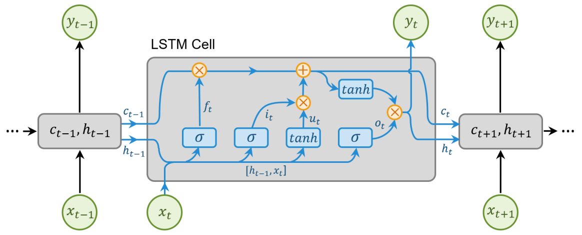

Fig. 1. LSTM layer architecture and details of a single LSTM cell. LSTM Layer has the form of a chain of repeating modules called LSTM cells. Each LSTM cell consists of layers of neural networks, internal states, and point-wise operations. The σ represents the sigmoid activation function.

$$\begin{aligned}
f_t &= sigmoid(U_f \times h_{t-1} + W_f \times x_t + b_f) \\
i_t &= sigmoid(U_i \times h_{t-1} + W_i \times x_t + b_i) \\
u_t &= tanh(U_u \times h_{t-1} + W_u \times x_t + b_u) \\
o_t &= sigmoid(U_o \times h_{t-1} + W_o \times x_t + b_o) \\
c_t &= (f_t \cdot c_{t-1}) + (i_t \cdot u_t) \\
h_t &= o_t \cdot tanh(c_t)
\end{aligned} \tag{1}$$

where $f_t, i_t, u_t, o_t \in \mathrm{I\!R}^{\mathrm{n}}$ are the outputs of the forget gate, the input gate, and the output gate, respectively. The c_t and h_t, as described earlier, are the cell-state and the hidden-state/output of the LSTM cell. They are initialized to zero and updated at each time step, demonstrating the "recurrent" nature of LSTM. $W_j \in \mathrm{I\!R}^{\mathrm{nxm}}$, $U_j \in \mathrm{I\!R}^{\mathrm{nxn}}$ and $b_j \in \mathrm{I\!R}^{\mathrm{n}}$ $(j = f, i, u, o)$ are weight and bias parameters learned during the training process.

The LSTM network could be a single or multi-layer network. The network could be layered or stacked by connecting the LSTM layer cells' hidden state to the input of the following LSTM layer cells. For final processing, the hidden state, h_t, of the last layer is often connected to a non-recurrent fully connected layer described using the following equation:

$$y_t = AF(W_y \times h_t + b_y) \tag{2}$$

where W_y is the weight matrix, b_y is a bias vector, h_t is the hidden state of the last LSTM layer, and AF is the activation function used in the layer.

2.2 RNNs Acceleration

The complex data dependencies encountered in RNN models make their acceleration more challenging than feed-forward neural networks and CNNs. Thus, CPUs and GPUs have difficulties exploiting RNN's fine-grained parallelism, and they remain underutilized [14]. Various optimization methods have been proposed to reduce the computational complexity and memory footprints and accelerate the RNN's inference using FPGA and ASIC accelerators. Proposed methods include pruning, quantization, and specialized computing units.

2.2.1 *Weight Pruning*

Weight pruning is a model compression technique proposed to reduce DNNs memory and computation costs. It removes redundant neuron connections converting the weight matrices to sparse matrices. Accordingly, converting the dense matrix-vector multiplication (MxV) to sparse matrix-vector multiplication (SpMxV) [20]. The resulting sparse matrices are stored using compressed sparse row (CSR) or compressed sparse column (CSC) format [21]. A hardware accelerator with on-chip sparse matrix decoding ability could reduce the dominant MxV operations by executing the multiply-and-accumulate (MAC) operations only on non-zero weights. However, the irregularity of the SpMxV could challenge the hardware accelerator's maximum performance, energy efficiency, and hardware utilization. Coarser-grained weight pruning methods to induce more structured sparsity patterns and a scheduler to encode and partition the compressed model into multiple processing elements were presented to overcome the limitations above [22], [23]. The authors in [24], [25] introduced a Bank-Balanced Sparsity (BBS) that splits the weight matrix row into multiple equal-sized banks. Then, a fine-grained pruning is performed to obtain a similar sparsity among the banks. These advanced methods improve RNNs' memory and computational costs with minimal performance loss; however, the training process is relatively complex and requires additional hyper-parameter to find the particular structure that would give the optimal performance. Instead of reducing the number of computations, this work focuses on reducing the hardware complexity, data dependencies, and computation patterns, thus increasing hardware utilization and throughput.

2.2.2 *Quantization*

The most used software framework for Deep learning performs inference by adopting the floating-point representation to ensure the best accuracy. However, considering hardware implementation on resource-constrained devices, floating-point arithmetic is not optimal for resource utilization. Various methods have focused on reducing numerical precision to reduce computing complexity and memory footprint. Reduced bit-width floating-point formats such as IEEE's half-precision (float16), Google's brain floating-point (bfloat16), and Nvidia's 18-bit TensorFloat format have been shown to run effectively without accuracy loss on a variety of platforms [15], [16]. Various methods were also proposed to quantize the structure of gates and interlinks in RNNs to reduce memory footprints and make it possible to use fixed-point MAC units [17], [18], [19]. These methods fall into two categories, post-training quantization, and quantization-aware training. The feedback loop in RNNs makes quantization-aware training challenging. Given that the main focus of this work is the efficient hardware accelerator architecture, post-training quantization is utilized to find the fixed-point representation that would ensure a suitable precision for the computations and low resource utilization.

2.2.3 *Specialized Computing Units*

With the breakthrough of DNN's application, the networks developed in the process showed the trend of increased computations for better accuracy. Therefore, it was vital to develop energy-efficient computing units, specifically matrix multiplication units. Bit-serial MAC that allows per-layer precision selection is presented in [26], [27]. Compared to conventional fixed-point MAC units, the introduced flexible bit-serial MAC supports various precision and is smaller in area but would require more cycles to finish the multiplication between high-bit precision operands. Approximate multipliers are another widely adopted approach that aims to achieve the best possible trade-off between accuracy and design efficiency [28]. The approximate logarithmic multipliers in [29], [30] are based on Mitchell's method [31] that utilizes binary logarithms for multiplication. The presented approaches offer a straightforward design but exhibit significantly higher computational error than bit-serial MAC. Conversely, the Booth algorithm-based approximate multipliers in [32], [33] focused on simplifying partial product generation offering a lower computational error at higher design complexity. This work proposes a CORDIC-based MAC unit that could support various precision to achieve the best possible trade-off between accuracy and design efficiency.

All previously presented methods reviewed in this section contributed to reducing the computation complexity and memory footprint of the RNNs. However, the computation kernels' configurability, stall, and resource parallelism remain challenging, leaving space for further exploration and improvement.

3 Accelerator Design

The proposed unified parallel CORDIC-based architecture accelerates the inference of the LSTM networks in addition to fully-connected layers. The main computation effort in the LSTM cells, as described earlier in section 2, comes from the gates, cell state, and hidden state computations. Despite the number of input features and the number of elements in the hidden state which may vary according to the application, the computation patterns in a single LSTM cell remain the same. This fact makes it possible to propose a versatile hardware architecture.

3.1 Systolic Outer Product-based Architecture

Optimizing the computation of the gate functions is essential to accelerate real-time LSTM network inference. With reference to Fig. 1 and equations listed in (1), each LSTM gate requires two MACs and element-wise vector additions. However, given that the weight matrices $W_j \in \mathbb{R}^{\mathrm{nxm}}$, $U_j \in \mathbb{R}^{\mathrm{nxn}}$ and the bias vector $b_j \in \mathbb{R}^{\mathrm{n}}$ in each gate share the same first dimension n, it is possible to combine them into one matrix of dimension $(n \times (n + m + 1))$. Likewise, since all the gates share the same dimensionality and input vector, it is possible to merge the gates' combined matrices into one matrix of size $((4n) \times (n + m + 1))$. Therefore, rather than optimizing four matrix-vector multiplications, each time step computations would focus on optimizing a single matrix-vector multiplication, reducing the pipeline control complexity.

In general, the matrix-vector multiplication of a matrix $W \in \mathbb{R}^{\mathrm{nxm}}$ by vector $x \in \mathbb{R}^{\mathrm{m}}$ could be optimized in two forms:

- Inner product-based: Multiply in parallel all elements of the input vector x by the matrix row vectors W_i. This would require m multipliers and an adder tree. In addition, the process should be repeated for all n rows of the matrix W as shown in Fig. 2 (a). In this case, the elements of the output vector are computed sequentially.
- Outer product-based: Multiply in parallel a single element of the input vector x by the matrix column vectors W_j. This would require n MAC units, and the process should be repeated for all m columns of the matrix W as shown in Fig. 2 (b). In this case, the output vector elements are computed in parallel.

The conventional designs of the MxV used in most of the existing LSTM network architectures are of the inner product-based option. The main drawback of such a structure is the hardware pipeline stall time imposed by the recurrent nature of the LSTM networks and the data dependencies between the output vector of the current time step and the input vector of the next time step. In such a case, the system must wait for the newly computed hidden state, h_t, before starting the subsequent step computations. This indicates that the whole system pipeline must be drained out before starting the following time step matrix-vector multiplication. Pipeline latency is critical to achieving a high throughput system. Therefore, our proposed scheme adopted the outer product-based approach to reduce the hardware stall time and the data dependencies between different time step computations. To illustrate the proposed scheme's computations, the matrix-vector multiplication in (3) shows a simplified example of an m-dimensional input vector with three hidden units. The matrix W represents the combined parameters matrix described earlier in this section. Each row in W represents the weight of one hidden unit, and the first column includes all units' biases. After $(m + 1)$ computations, the resulting vector y contains the sum of the products of each hidden unit. Using as many multipliers as hidden units working in parallel, the partial sum of all the units will be simultaneously computed for each element in the input vector; an example of a single partial sum is boldly marked in (3). Similarly, each unit's final sum of products could simultaneously be obtained using an arithmetic accumulator per unit.

$$
\begin{aligned}
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} &= \begin{bmatrix} b_0 & \mathbf{w_{00}} & w_{01} & \dots & w_{0m} \\ b_1 & \mathbf{w_{10}} & w_{11} & \dots & w_{1m} \\ b_2 & \mathbf{w_{20}} & w_{21} & \dots & w_{2m} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{x_0} \\ \vdots \\ x_m \end{bmatrix} \\
&= \begin{bmatrix} b_0 + \mathbf{w_{00}x_0} + w_{01}x_1 & \dots & w_{0m}x_m \\ b_1 + \mathbf{w_{10}x_0} + w_{11}x_1 & \dots & w_{1m}tx_m \\ b_2 + \mathbf{w_{20}x_0} + w_{21}x_1 & \dots & w_{2m}x_m \end{bmatrix}
\end{aligned} \tag{3}
$$

Given the serial processing nature of the input vector in the outer product-based scheme, a single activation function block (AFB) performing both activation and point-wise operations can serve all the hidden units forming a systolic ring topology. Fig. 3 demonstrates the computation sequence in the proposed scheme. Steps 1 through 3 compute and accumulate MxV partial products. The final results of the MxV are stored in a parallel-in serial-out shift register (PISO), step 4. The activation function block in step 5 uses the MxV results stored in the PISO register to compute the LSTM cell or the FC layer output vector, which also serves as an input to the subsequent matrix-vector multiplication. In step 6, the AFB updates the input vector with the newly computed output vector elements one at a time. Once the first element in the output vector is computed, the subsequent matrix-vector multiplication could immediately start. While the MxV kernel performs step 1 computations, the AFB would update step 2 input, and the same process repeats for the rest of the elements. The proposed architecture allows the following subsequent MxV to start without waiting for the system pipeline to be

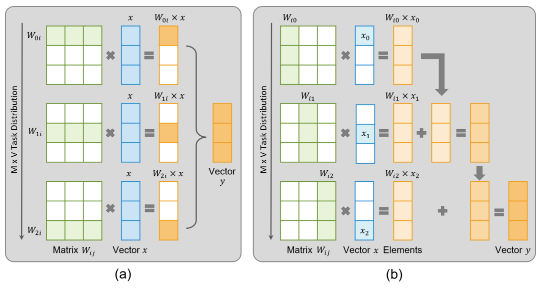

Fig. 2. Matrix-vector multiplication (MxV) optimization options: (a) Inner product-base MxV. (b) Outer product-base MxV.

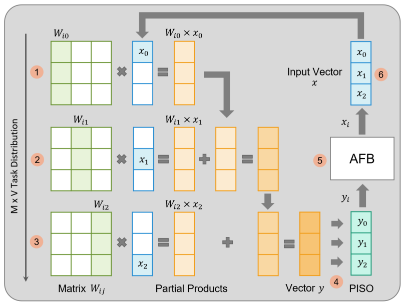

Fig. 3. Computation sequence in the proposed systolic outer product-based architecture.

drained. Additionally, the hardware is iteratively reused to compute the output of either the LSTM cell or the FC layer, reducing the hardware complexity and power consumption.

3.2 CORDIC-based Computing Units

3.2.1 Overview

The Coordinate Rotational Digital Computer algorithm is an iterative technique to evaluate elementary functions [35]. The CORDIC algorithm's basic idea is to rotate a two-dimensional vector through an angle to obtain some elementary functions such as sine, cosine, inverse tangent, hyperbolic sine, hyperbolic cosine, inverse hyperbolic tangent, multiplication, division, and square root. These functions could be further processed to obtain other functions such as tangent, logarithms, and exponential functions [36]. The CORDIC algorithm can evaluate three classes of functions: linear, circular, and hyperbolic. It computes the results using three variables: x, y, and z. The initialization of these variables depends on the implemented arithmetic operation or mathematical function. After initializing the three variables, a set of iterative equations is repeatedly applied to these variables until they converge to the results. The generalized iterative equations are:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & m\alpha_i\delta_i \\ -\alpha_i\delta_i & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad (4)$$
$$z_{i+1} = z_i + \delta_i\theta_i$$

where i is the index of iteration, m determines the class of function being evaluated: linear ($m = 0$), circular ($m = +1$), hyperbolic ($m = -1$). The value of δ_i is either -1 or $+1$ and is chosen to either drive y or z toward zero and obtain the desired function. In addition, α_i is set to 2^{-i}, this setting simplifies the computations performed in (4) to simple shift and add binary operations. The rotation angle θ_i for each of the three classes is computed using $\tanh^{-1}\alpha_i$ for ($m = -1$), α_i for ($m = 0$), and $\tan^{-1}\alpha_i$ for ($m = +1$).

This work utilizes the CORDIC algorithm linear and hyperbolic modes. The linear mode is employed for the MAC implementation. In addition, it is used together with the hyperbolic mode to implement the configurable activation function block. Table 1 summarizes the available basic CORDIC functions in linear and hyperbolic modes. In Table 1 x_{in}, y_{in}, and z_{in} represent the initial values of variables x, y, and z respectively. The factor K_h in hyperbolic mode is a constant that corrects the amplification introduced by the linearized "rotation" in x and y coordinates and is given by

$$K_h = \prod_{i=1}^{N} \sqrt{1 - (2^{-i})^2} \qquad (5)$$

The accuracy of the functional results in Table 1 depends on the convergence of the CORDIC algorithm, which is how closely the y or z variable is driven toward zero. The y or z variable can theoretically be driven to zero if the initial point (x_{in}, y_{in}) or the initial value z_{in} is within a specific range. This specific range is known as the "range of convergence" and is limited by the sum of rotation angles $\theta_N + \sum_{j=i+1}^{N} \theta_j$. Due to the incomplete representation of the hyperbolic rotation angles θ_i, some iterations must be repeated to satisfy the convergence theorem. In [36], it was recommended that every $(4, 13, 40, 121, ...)^{th}$ iteration be repeated to complete the angle representation. The numerical values of each class range of convergence are also given in Table 1. These convergence ranges are further discussed in the context of DNN in the coming sections.

3.2.2 MAC Units

Given the CORDIC algorithm's ability to perform various computational tasks using two fundamental inexpensive operations (shift and add), it is used to realize the MAC computations in the proposed architecture. As noted from Table 1, operating in the linear rotation mode yields the sum of the product $(x_{in}z_{in})$ and the input y_{in}. For the CORDIC algorithm to converge to the desired results, the initial value of z_{in} should be within $[-1, 1]$. The limited convergence range is one of the major shortcomings of the CORDIC algorithm. The authors in [37] proposed expanding the set of iteration index to $i = -M, -M+1, ..., -1, 0, 1, 2, ..., N$ to expand the convergence range to $[-2^{M+1}, 2^{M+1}]$. Such an expansion comes at the cost of increased data word length or roundoff error in a fixed-point hardware implementation.

Expanding the convergence range may not necessarily be needed for neural network implementation. Feature scaling, an essential preprocessing step in many machine learning algorithms, including deep neural networks, would scale the range of the input features. More specifically, normalization transforms feature values to a standard range,

TABLE 1
Summary of basic CORDIC algorithm in linear and hyperbolic modes.

	Hyperbolic $m = -1$ $i = 1, 2, 3, ..., N$ Repeat Iterations $(4, 13, 40, 121, ...)$	**Linear** $m = 0$ $i = 1, 2, 3, ..., N$
Vectoring Mode $(y \to 0)$ $\delta_i = \begin{cases} -1 & \text{if } x_i y_i < 0 \\ +1 & \text{if } x_i y_i \geq 0 \end{cases}$	$x_N \approx K_h\sqrt{x_{in}^2 - y_{in}^2}$ $z_N \approx z_{in} + \tanh^{-1}(\frac{y_{in}}{x_{in}})$ $\lvert\tanh^{-1}(\frac{y_{in}}{x_{in}})\rvert \leq 1.1182$	$x_N = x_{in}$ $z_N \approx z_{in} + \frac{y_{in}}{x_{in}}$ $\lvert\frac{y_{in}}{x_{in}}\rvert \leq 1$
Rotation Mode $(z \to 0)$ $\delta_i = \begin{cases} -1 & \text{if } z_i \geq 0 \\ +1 & \text{if } z_i < 0 \end{cases}$	$x_N \approx K_h[x_{in}\cosh(z_{in}) + y_{in}\sinh(z_{in})]$ $y_N \approx K_h[x_{in}\sinh(z_{in}) + y_{in}\cosh(z_{in})]$ $\lvert z_{in}\rvert \leq 1.1182$	$x_N = x_{in}$ $y_N \approx y_{in} + x_{in}z_{in}$ $\lvert z_{in}\rvert \leq 1$

usually $[0,1]$ or sometimes $[-1,1]$. Accordingly, the convergence range of the CORDIC algorithm should not be a limitation for the input features. Similarly, the output range of the most commonly used activation functions in DNN models is also within the $[-1,1]$ range. Therefore, the internal results between the hidden layers are also within the required $[-1,1]$ range. Furthermore, when the output range of the activations is not within the $[-1,1]$ range, a batch norm layer is usually inserted between a hidden layer and the next layer. The inserted batch norm layer normalizes the outputs from the first hidden layer before passing them as input to the next layer. Hence, the range of the passed input would also be within the CORDIC algorithm range. Given that the main focus of this paper is the LSTM network, the LSTM cell formulation in (1) is used to demonstrate the MAC units' input range within the cell. Assuming the LSTM cell input, x_t is normalized to zero mean and unit variance, and the h_t vector is initialized to zero. Passing x_t and h_t vector elements as z_{in} in the CORDIC MAC units would satisfy the CORDIC algorithm convergence range requirement. In addition, given that the two main activation functions used in the LSTM cell structure are the $sigmoid$ and $tanh$, the output of $f_t, i_t, u_t, o_t \in [-1,1]$. Consequently, the newly computed h_t, which would also be used as an input in the following MAC computations, would also be in the required $[-1,1]$ range. Thus, the CORDIC algorithm could efficiently be used to perform the required MAC computations within the LSTM cell and other general neural network types without expanding the convergence range.

The recursive nature of the CORDIC algorithm creates another hard lower limit on the algorithm latency. As noticed from the generalized iterative equations in (4), the direction of the subsequent microrotation depends on the current iteration result. Consequently, the standard CORDIC implementation is sequential and therefore slow. This work adopts a prediction-based approach to speed up the serial CORDIC implementation in the linear rotation mode. The main idea is to examine the binary representation of the input z_{in} and generate the bipolar values corresponding to the rotation directions δ_i. To explain the underlying idea, we will assume that the initial value of z_{in} is given in 2's complement binary notation. Using the binary-to-bipolar recoding (BBR) method, the corresponding rotation directions are obtained as follows:

$$\begin{aligned} z_{in} &= (-b_o) + \sum_{j=1}^{N-1} b_j 2^{-j} \\ &= (-b_o) + \sum_{j=1}^{N-1} [2^{-j-1} + (2b_j - 1)2^{-j-1}] \\ &= \sum_{i=1}^{N} \delta_i 2^{-i} - 2^{-N} \end{aligned} \tag{6}$$

where $b_j \in \{0,1\}$, $\delta_i \in \{-1,1\}$ and

$$\begin{cases} \delta_1 &= 2b_0 - 1 \\ \delta_i &= 1 - 2b_{i-1}, \quad i = 2,3,...,N \end{cases} \tag{7}$$

Using (6) (7), the N rotation directions (δ_1 to δ_N) could directly be derived in parallel eliminating the z-datapath, hence reducing the implementation area. In addition, the speed of the proposed algorithm could further be improved using tree-structured adders for the y microrotations. Compared to conventional fixed-point MAC units, consisting of a multiplier and an adder block, the proposed CORDIC-based MAC could be fully pipelined, eliminating the adder block idle time in the conventional MAC units. Additionally, it could support various precision to achieve the best possible trade-off between accuracy and design efficiency.

3.2.3 Activation Functions

The nonlinear activation function is one of the main components of the artificial neural network computational units. Each neuron in the LSTM cell and FC layers needs an activation function. The output of the MAC units passes through the activation function to compute the final output of each neuron. Sigmoid and hyperbolic tangent functions are the most widely used activation functions and are the primary activation functions in the LSTM cell. The sigmoid and tanh functions are mathematically defined as follows:

$$tanh(z) = \frac{sinh(z)}{cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{8}$$

$$sigmoid(z) = \frac{1}{1 + cosh(z) + sinh(z)} = \frac{1}{1 + e^{-z}} \tag{9}$$

The straightforward implementation of these functions on hardware is costly, given that both require computing the exponential and division. Various methods have been proposed to achieve high-fidelity approximations [39]. These methods generally fall into two main categories, piecewise linear approximation and look-up table-based (LUT) approaches. In both categories, the implementations tend to use more resources and latency to achieve high-accuracy approximations. This work focuses on the efficient hardware implementation of the activation functions using an optimized CORDIC algorithm that simplifies the approximation method without sacrificing the network accuracy. The CORDIC algorithm in the hyperbolic rotation mode allows the computation of the hyperbolic $sinh$ and $cosh$ functions, and in the linear vectoring mode, it could be used to perform $division$. The integration of the two modes allows the computation of the $sigmoid$ and $tanh$ activation functions using two relatively inexpensive operations.

(a) CORDIC Hyperbolic sinh and cosh

I. Expanding the Basic Range of Convergence

The CORDIC hyperbolic convergence range reported in Table 1 is not large enough to cover the range of the $sigmoid$ and $tanh$ activations where the activations are not saturated. The $tanh$ activation function saturates to -1 for inputs less than -3 and 1 for inputs greater than 3. On the other hand, the $sigmoid$ activation converges to 0 for inputs less than -5 and 1 for inputs greater than 5. The approach presented in [37] is adopted to expand the hyperbolic CORDIC convergence range. The authors in [37] proposed expanding the set of iteration index to $i = -M, -M+1, ..., -1, 0, 1, 2, ..., N$ and thus augmenting the iteration angle list with $\theta_i = \tanh^{-1}(1 - 2^{i-2})$ for $i \leq 0$. Based on the new augmented θ_i's list, the algorithm iterations presented in (4) are modified to, for i ≤ 0

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & -\delta_i(1-2^{i-2}) \\ -\delta_i(1-2^{i-2}) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{10}$$

$$z_{i+1} = z_i + \delta_i \tanh^{-1}(1-2^{i-2})$$

for i > 0

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & -\delta_i 2^{-i} \\ -\delta_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{11}$$

$$z_{i+1} = z_i + \delta_i \tanh^{-1}(2^{-i})$$

Accordingly, the new convergence range is obtained using,

$$\begin{aligned} |z_{in}| \leq & \sum_{i=-M}^{0} \tanh^{-1}(1-2^{i-2}) \quad + \\ & \tanh^{-1}(2^{-N}) + \sum_{i=1}^{N} \tanh^{-1}(2^{-i}) \qquad (12) \\ |z_{in}| \approx & \quad 3.44 \quad (\text{for } M=1) \\ |z_{in}| \approx & \quad 5.16 \quad (\text{for } M=2) \end{aligned}$$

Including negatively indexed iterations in finite word-length implementation may introduce larger arithmetic errors than those caused by only the positively indexed iterations. Therefore, the scale factor K_h introduced in (5) should be extended to accommodate the amplification introduced by the negatively indexed iterations. The K_h factor could be redefined as,

$$\begin{aligned} K_h = & \left[\prod_{i=-M}^{0} \sqrt{1-(1-2^{i-2})^2}\right] \left[\prod_{i=1}^{N} \sqrt{1-(2^{-i})^2}\right] \\ K_h \approx & \quad 0.2652 \quad (\text{for } M=1) \qquad (13) \\ K_h \approx & \quad 0.0923 \quad (\text{for } M=2) \end{aligned}$$

As noticed from (13), the constant K_h is less than one in magnitude, and the inclusion of more iterations makes K_h even smaller. In a fixed-point implementation, the hyperbolic CORDIC results computed using the smaller K_h will use less of the dynamic range of the three CORDIC variables, x, y, and z, detailed in Table 1. Because the CORDIC output's relative error depends on the LSB's weight, the smaller K_h will lead to a higher error. Therefore, the number of negatively indexed iterations should be limited to as few as possible. In our implementation of the CORDIC-based activation function, we evaluate the $sigmoid$ activation as a scaled version of the $tanh$ using the relationship detailed next.

$$\begin{aligned} tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ &= \frac{e^z + e^{-z} - 2e^{-z}}{e^z + e^{-z}} \\ &= 1 - \frac{2e^{-z}}{e^z + e^{-z}} \\ &= 1 - \frac{2}{e^{2z}+1} \end{aligned} \tag{14}$$

Since the $sigmoid$ function is symmetric around the origin,

$$1 - sigmoid(z) = sigmoid(-z) = \frac{1}{1+e^z} \tag{15}$$

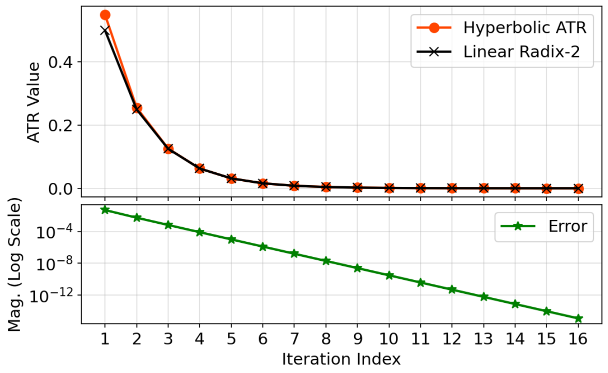

Fig. 4. Hyperbolic ATR and linear radix-2 constants. The top plot compares the actual values of the hyperbolic ATRs and the linear radix-2 constants. The bottom plot displays the approximation error assuming that radix-2 constants are the same as hyperbolic ATRs.

Using (14) and (15),

$$\begin{aligned} tanh(z) &= 1 - 2sigmoid(-2z) \\ sigmoid(z) &= \frac{1+tanh(\frac{z}{2})}{2} \end{aligned} \tag{16}$$

The above-detailed relationship enables the implementation of the two activation functions using a single configurable hardware unit that requires only two additional bit shifters and an adder. In addition, it limits the number of the negatively indexed iterations to $M = 1$, thus limiting the increase in the relative error caused by the additional negatively indexed iteration required to cover the $sigmoid$ input range.

II. *Partitioning the Radix Set*

As pointed out earlier, the sequential determination of the value δ_i limits the speed of the conventional CORDIC rotation. In this paper, we show that the computation of the positively indexed rotation directions in the hyperbolic class could be partially parallelized, thus reducing the overall latency without affecting the accuracy. The CORDIC algorithm performs rotation iteratively by breaking the rotation angle z_{in} into a set of predefined small angles that could be implemented using low hardware costs (4). For the hyperbolic class, this set is: $\{\theta_1, \theta_2, ..., \theta_N\} = \{\tanh^{-1} 2^{-1}, \tanh^{-1} 2^{-2}, ..., \tanh^{-1} 2^{-N}\}$. The θ terms are referred to as Arc Tangent Radix (ATR) constants. The hyperbolic ATR constants approach the linear radix-2 constants gradually for increasing values of the CORDIC iteration index. Fig. 4 compares the hyperbolic ATR and the linear radix-2 constants. The error in assuming that the hyperbolic ATRs are the same as the linear radix-2 constants for the large iteration index i in fixed-point representation is negligible. Therefore, if the linear radix-2 is assumed for the least significant part of the rotation directions, the precision in the least significant part will be about the same as the conventional ATRs. Accordingly, the prediction-based approach adopted in the CORDIC MAC for the parallel generation of the rotation directions could also be utilized for the least significant part of the hyperbolic rotations.

Using the earlier observation on hyperbolic ATRs approximation using linear radix-2 constants, we define the

Hybrid Hyperbolic ATR :

$$\{\overbrace{\tanh^{-1} 2^{-1}, \tanh^{-1} 2^{-2}, ..., \tanh^{-1} 2^{-n+1}}^{\text{most significant part}}, \underbrace{2^{-n}, ..., 2^{-N}}_{\text{least significant}}\}$$

The defined set is a mix of conventional hyperbolic ATRs for the most significant part and the linear radix-2 constants for the least significant part. The most significant part iterations, along with the negatively indexed iterations introduced in the previous part, are performed using the conventional sequential CORDIC iterations. On the other hand, the iterations corresponding to the least significant part are parallelized using the binary-to-bipolar recoding method as in the linear mode. Because the CORDIC iterations related to the most significant part may change the value of the bits in the least significant part, the evaluation of the least significant part rotation directions is performed after the rotations of the most significant parts to avoid errors. To identify the partitioning index that will preserve the full accuracy, the error introduced by assuming that the hyperbolic ATR is the same as the linear radix-2 constant is used.

$$\epsilon_i = \tanh^{-1} 2^{-i} - 2^{-i} \tag{17}$$

Using the Taylor series approximation of $\tanh^{-1} 2^{-i}$,

$$\begin{aligned}\epsilon_i &= \left[2^{-i} + \frac{1}{3}2^{-3i} + \frac{1}{5}2^{-5i} + ...\right] - 2^{-i} \\ &= \frac{1}{3}2^{-3i} + \frac{1}{5}2^{-5i} + ...\end{aligned} \tag{18}$$

This error should be less than the smallest representable angle, 2^{-N+1} for N bits fixed-point representation with $N-1$ bits in the fractional part. Given that $\frac{1}{3}2^{-3i} + \frac{1}{5}2^{-5i} + ... < 2^{-3i-1}$,

$$2^{-N+1} \geq 2^{-3i-1} > \epsilon_i = \frac{1}{3}2^{-3i} + \frac{1}{5}2^{-5i} + ... \tag{19}$$

Solving for i,

$$\begin{aligned}2^{-N+1} &\geq 2^{-3i-1} \\ i &\geq \frac{N-2}{3}\end{aligned} \tag{20}$$

Hence, the minimum index value i that will preserve the full accuracy is $i \simeq \frac{N-2}{3}$ and is approximately close to the minimum index reported in [38] which focused only on the CORDIC circular mode.

In order to use the binary-to-bipolar recoding method presented in subsection 3.2.2 in the parallel generation of the least significant part rotation directions, the recording method should be extended to accommodate for the hyperbolic repeated iterations. Assuming that the rotation angle is in the standard binary representation, Table 2 summarizes the cases to consider in the parallel generation of the hyperbolic repeated iteration directions. It has been noticed that the incomplete representation of the hyperbolic rotation angles may affect the binary representation of the most significant part of the z output used in the parallel generation of the rotation directions in the least significant part. More specifically, the most significant part of the z output may not be zero. This usually happens when the first iteration in the least significant part is one of the iterations to be repeated, case 1. Therefore, the bit at iteration index $i-1$ should be compared against the sign bit. If the two bits match, the prediction of the repeated iteration direction will be based on the bit at iteration index i.

TABLE 2
Summary of cases to consider in the parallel generation of hyperbolic repeated iteration directions.

Case 1: The iteration to be repeated is the first iteration in the least significant part.					
b_0 (Sign bit)	b_{i-1}	b_i	δ_i	δ_i (Repeated)	δ_{i+1}
0	0	0	-1	+1	-1
0	0	1	-1	-1	+1
0	1	0	-1	-1	-1
1	0	1	+1	+1	+1
1	1	0	+1	+1	-1
1	1	1	+1	-1	+1
Case 2: The iteration to be repeated is not the first iteration in the least significant part.					
0 or 1	0	0	+1	+1	-1
0 or 1	0	1	+1	-1	+1
0 or 1	1	0	-1	+1	-1
0 or 1	1	1	-1	-1	+1

$$\delta_{i\ (Repeated)} = (1 - 2b_i) \tag{21}$$

On the other hand, if the bit at iteration index $i-1$ did not match the sign bit, then the direction of the repeated iteration should be obtained using,

$$\delta_{i\ (Repeated)} = (1 - 2b_0) \cdot (1 - 2b_{i-1}) \cdot (1 - 2b_i) \tag{22}$$

The second case details the generation of the rotation direction when the repeated iteration is not the first in the least significant part. This case is straightforward and only requires evaluating the bit at the iteration index of the repeated iteration as in (21). For all the abovementioned cases, the rotation direction of the iteration following the repeated iteration will always depend on the inverse of the bit at the index of the repeated iteration.

$$\delta_{i+1} = (2b_i - 1) \tag{23}$$

(b) CORDIC Linear Division

Given the limited range of the sigmoid and tanh activation functions, which also fall within the CORDIC linear mode's convergence range, the hyperbolic $sinh$ and $cosh$ division is implemented using the CORDIC algorithm in linear vectoring mode. Unlike the CORDIC rotation mode, the CORDIC vectoring mode rotation direction depends on both x and y components, making it more challenging to derive the rotation directions in parallel. Therefore, the CORDIC divider in this work is implemented using the standard iterative CORDIC method. It is worth mentioning that the CORDIC divider could be fully pipelined to reduce the overall latency, making it a practical choice for low-power edge computing solutions.

4 Hardware Implementation

This section presents the FPGA implementation details of the proposed hardware architecture introduced in section 3.

The architecture was coded in C++ using the Xilinx Vivado HLS tool targeting the Xilinx PYNQ-Z1 FPGA development board. The development board consists of an XC7Z020 ZYNQ series FPGA containing a Dual ARM Cortex-A9 core processor. In addition, it is a hardware platform for the PYNQ open-source framework, which comprises software running on the ARM CPUs and a base hardware library. After verifying the functionality of our custom-designed accelerator modules, the designed hardware accelerator was exported to Xilinx Vivado Design Suite for synthesis and implementation. A simplified block diagram of the implemented hardware architecture is shown in Fig. 5. The main modules consist of an array of computationally independent processing elements, a parallel-in serial-out shift register, a single activation function block, and a control module.

Each processing element consists of two main parts: a single port on-chip Block RAM (BRAM) and a CORDIC-based MAC unit. The BRAM holds the stationary network parameters (weights, biases), and the CORDIC-MAC unit performs the corresponding MxV computations. Each PE BRAM holds the parameters of one neuron unit per layer. The on-chip BRAMs are used in the implementation to reduce the external I/O communication and to enable concurrent operation of the PEs. The AFB is a configurable CORDIC-based activation block. It could be configured to handle LSTM gates activations, cell state updates, hidden state computations, and FC layer activations. Last is the control module, which encodes instructions and controls the order of operations, data movements, and storage. Given that the LSTM layers and cells reuse the hardware, the architecture described above is generic and could be extended in terms of the number of hidden units in the LSTM gates and the input vector size as long as the hardware resources are available.

The general data flow in the proposed architecture is as follows; the network input data are sequentially introduced and concurrently multiplied by their corresponding weights in each PE. The multiplication results are added to the accumulator register in each PE. The final sums of products from all PEs are passed to the PISO shift register to be shifted through the activation function block. The outputs of the activation function block (c_t, h_t) are stored in internal memory to be used in the next-time step computations or passed to the output.

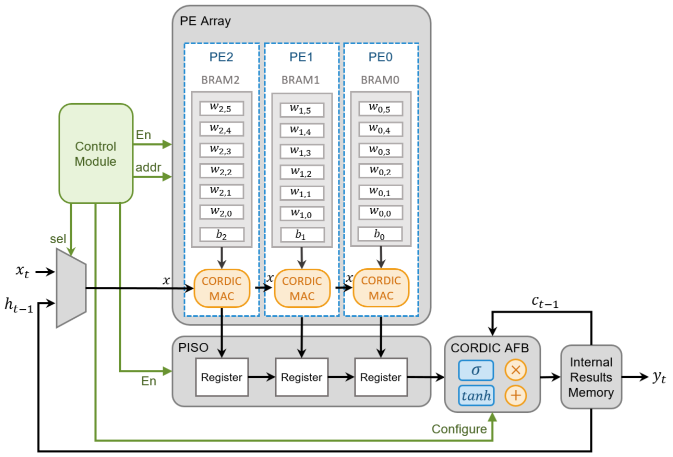

Fig. 5. The hardware architecture of the proposed unified parallel CORDIC-based accelerator. The main modules consist of an array of computationally independent Processing elements, a parallel-in serial-out shift register, a single activation function block, and a control module.

4.1 CORDIC MAC Units

As described earlier, the architecture PEs include CORDIC-based MAC units implemented using the binary-to-bipolar recoding method presented in subsection 3.2.2. Algorithm 1 describes the CORDIC-MAC computations using the BBR method. The iterations in the first loop, which corresponds to rotation direction prediction, are independent; accordingly, the loop could be unrolled to obtain the rotation directions in parallel. The obtained rotation directions are then used in the second loop implemented using bit shifting and tree-structured adders to compute the corresponding multiply and accumulate computations.

Algorithm 1 CORDIC-based MAC unit using BBR method

```
Require: $-1 \leq z_{in} \leq 1$
Input: $x_{in},\ y_{in},\ z_{in}$
Output: $y_{in}\ +\ x_{in}z_{in}$
  $n \leftarrow number\ of\ fractional\ bits$
  $dir[n] \leftarrow 0$
  $y \leftarrow y_{in}$
  if $z_{in} < 0$ then $dir[0] = +1$ else $dir[0] = -1$
  for $i = 1, i < n$ do
    if $z_{in}[n-i] == 0$ then $dir[i] = +1$ else $dir[i] = -1$
  end for
  for $j = 0,\ j < n$ do
    $x_{reg} \leftarrow x_{in} >> (j+1)$
    if $dir[j] == +1$ then $x_{reg} \leftarrow -x_{reg}$
    $y \leftarrow y + x_{reg}$
  end for
  return $y$
```

The implemented parallel CORDIC MAC unit using a 16-bit fixed-point representation, 4 bits for the signed integer part, and 12 bits for the fractional part has a latency of 3 clock cycles with single-cycle data throughput. Comparing the performance of the implemented CORDIC-based MAC unit against the Xilinx DSP48E using the same data representation and with reference to floating-point multiplication gives relative mean absolute error (MAE) results. The MAE of the CORDIC-based MAC is $\sim 5.8 \times 10^{-4}$, while it is $\sim 4.5 \times 10^{-4}$ for the DSP48E. The absolute error heat maps of the two implementations are shown in Fig. 6. The output quantization error in the CORDIC-based implementation is higher than the DSP48E. The quantization error in the CORDIC-based implementation can be split into two parts; the first part is due to the input samples quantization, while the second is caused by the truncation of x and y after shifting them to the right at each iteration. The truncation error is equivalent to adding random variables to each of the newly computed x and y in addition to the accumulated error terms from the prior stage. In section 5.5, we will show that the accuracy loss in the designed network using the proposed CORDIC-based MAC units is negligible.

4.2 CORDIC AFB

Implementing the CORDIC-based activation functions consists of cascading three CORDIC processors, as illustrated

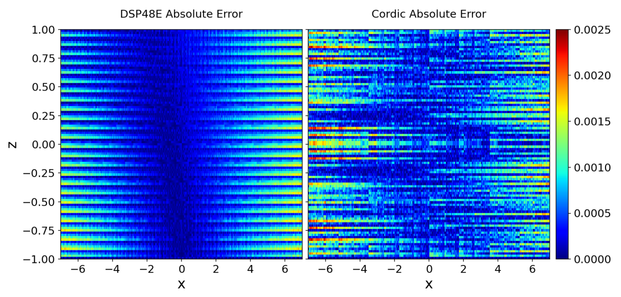

Fig. 6. The absolute error heat maps of the CORDIC-based MAC unit and the Xilinx DSP48E using 16 bits fixed-point data representation. The x and z are the multiplication operands.

in Fig. 7. The first two processors are for the hyperbolic $sinh$ and $cosh$ computations, while the third processor performs the division. The first hyperbolic processor generates and applies the rotation directions sequentially as in the conventional CORDIC algorithm using negatively indexed microrotation angles and angles defined in the most significant part of the Hybrid Hyperbolic ATR set introduced in the previous section. This first processor uses the input rotation angle z_{in} in addition to initial vector coordinates x_{in} and y_{in}, to compute z_n, x_n, and y_n at the end of first n iterations. The input rotation angle z_{in} is connected to the output of the CORDIC-MAC unit and PISO unit as shown in Fig. 5, while x_{in} is set to $1/K_h$ and $y_{in} = 0$. The second hyperbolic processor eliminates the z data path through the parallel generation of the rotation directions using z_n. The implementations of the on-the-fly converter and the second hyperbolic processor are based on the approach detailed in 3.2.3. The generated rotation directions by the on-the-fly converter are applied to x_n and y_n to compute the final value of the hyperbolic $sinh$ and $cosh$. The third CORDIC processor uses the computed $sinh$ and $cosh$ and performs the division using the conventional CORDIC algorithm in linear vectoring mode.

The implemented CORDIC-based activation functions using 16-bit fixed-point representation have a latency of 21 clock cycles. Furthermore, if we were also to pipeline the first hyperbolic and the third linear processors, then this would enable a single-cycle data throughput and reduce the latency to 13 clock cycles at the cost of additional hardware resources. Evaluating the performance of the implemented CORDIC-based activation functions gives MAE of $\sim 2.8 \times 10^{-4}$ for the $tanh$ activation and $\sim 1.6 \times 10^{-4}$ for the $sigmoid$ using (16) . Fig. 8 shows the similarity in shape between the real-valued and the CORDIC-based approximation, along with the absolute error for both activations over the range of interest. The output quantization error in the hyperbolic CORDIC implementation includes, in addition to input sample quantization and truncation errors, angle approximation error caused by the quantized representation of the CORDIC rotation angle using a finite number of elementary angles. Compared with the activation function implementations reported in [39], the proposed CORDIC-based activation function achieves higher approximation accuracy, making it suitable in applications with minimal accuracy loss requirements.

As pointed out earlier, the activation function block is a configurable activation block that handles the LSTM gates activation, the cell state update, the hidden state computations, and the dense fully-connected layer activation. To reduce the implementation complexity and enable activation function and arithmetic units reuse, the LSTM gates activation and point-wise operations are pipelined as shown

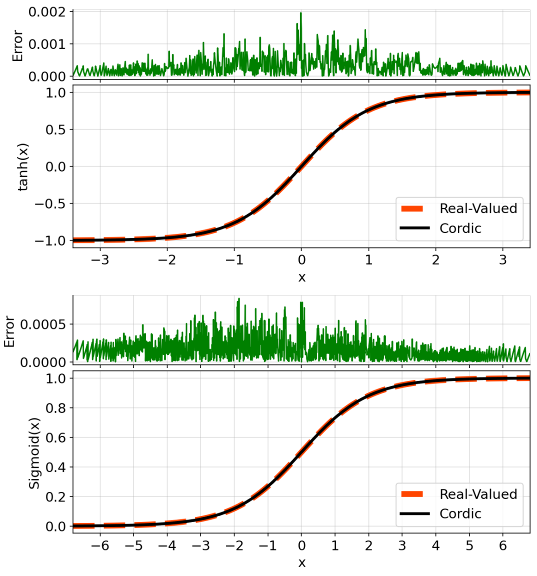

Fig. 8. CORDIC-based activation functions performance. The top plot shows the real-valued $tanh$ and the CORDIC-based approximation along with the corresponding absolute error. The bottom plot shows the same but for the $sigmoid$ activation function.

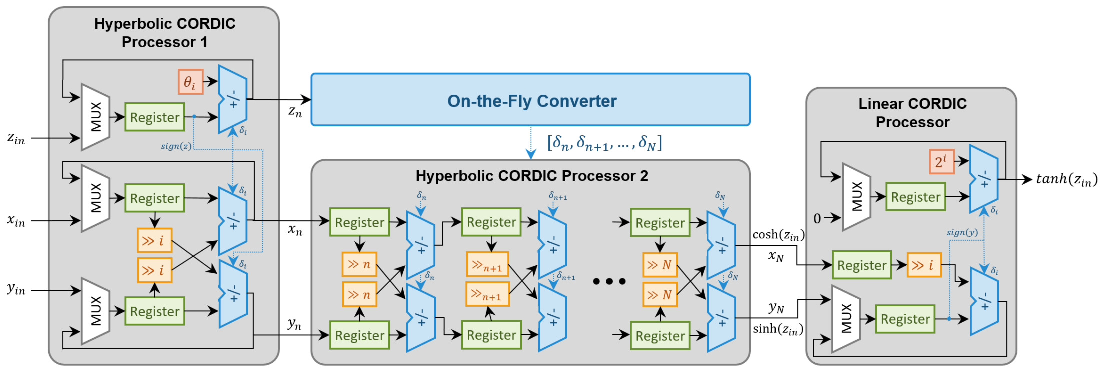

Fig. 7. The implemented CORDIC- based activation function consists of cascading three CORDIC processors. The first two processors are used to compute the hyperbolic $sinh$ and $cosh$, while the third performs the division to compute the final result of $tanh(z_{in})$. The input rotation angle z_{in} is connected to the output of the CORDIC-MAC unit, while x_{in} is set to $1/K_h$ and $y_{in} = 0$.

in Fig. 9. The LSTM cell formulation in (1) is divided into six pipeline stages $S0 - S5$. The inputs to the activation pipeline are the matrix-vector multiplication results of the forget gate, input gate, and output gate shown in Fig. 1 and expressed in the LSTM cell formulation in (1). The outputs are the cell state and the hidden state. In the cases when the layer is of a fully-connected type, only the first pipeline stage would be activated, and the computed activation result is passed to the output.

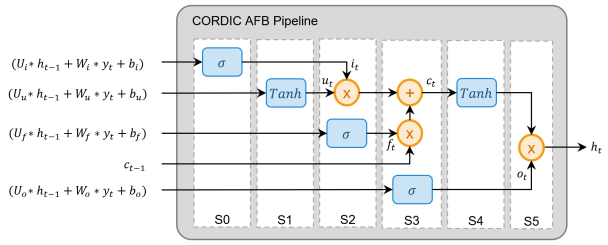
CORDIC AFB Pipeline
$(U_i * h_{t-1} + W_i * y_t + b_i)$
$(U_u * h_{t-1} + W_u * y_t + b_u)$
$(U_f * h_{t-1} + W_f * y_t + b_f)$
c_{t-1}
$(U_o * h_{t-1} + W_o * y_t + b_o)$
σ
Tanh
i_t
u_t
c_t
f_t
o_t
h_t
S0
S1
S2
S3
S4
S5

Fig. 9. CORDIC Activation function block pipeline. The inputs to the CORDIC AFB pipeline are the matrix-vector multiplication results of the forget gate, input gate, and output gate. The outputs are the cell state and the hidden state.

5 Evaluation and Analysis

5.1 Experimental Setup

The Grand St. Bernard open-source dataset [40] is used to demonstrate the performance of the designed hardware accelerator in a real-case application. The dataset consists of temperature measurements and other metrological characteristics of the environment collected for two months from multiple wireless sensor nodes deployed at the Grand St. Bernard pass, located between Switzerland and Italy. The performance and the flexibility of the designed unified computing kernel in performing recurrent and non-recurrent fully connected layers computations are validated using an Autoencoder-LSTM network trained in Keras Tensorflow using the dataset mentioned above [34], [42]. The trained network integrates Autoencoder and LSTM neural networks for real-time temperature forecasting on resource-constrained devices. Table 3 summarizes the trained Autoencoder-LSTM network model. The autoencoder consists of two FC dense layers (FC1, FC2), while the LSTM network comprises an LSTM layer followed by two fully connected dense layers (FC3, FC4). The LSTM layer has a hidden size of 40 and an input dimension (or sequence length) of 30-time steps.

The LSTM layer computations dominate the number of computations in the trained network. The number of MAC computations processed by our accelerator for the LSTM layer only is approximately $\sim 0.2M$ while it is $\sim 8K$ for the rest of the network. Therefore, to reduce the latency and increase the throughput, the number of processing elements in the implemented design is set to four times the LSTM layer hidden size to enable the parallel computation of the LSTM cell gates. Furthermore, to maintain the trained network accuracy while considering the limited hardware resources on resource-constrained devices, a 16-bit fixed-point data representation, 4 bits for the signed integer part, and 12 bits for the fractional part is adopted in the implementation. After verifying the functionality of our custom-designed accelerator modules in Xilinx's Vivado HLS tool, the designed hardware accelerator is exported to Xilinx Vivado Design Suite to build the hardware overlay. The resulting hardware overlay is loaded on the PYNQ-Z1 board. The top-level diagram of the hardware overlay on the PYNQ-Z1 board is shown in Fig. 10. A python code is developed to interface with the hardware overlay. The quantized network parameters are loaded into the programmable logic (PL) distributed BRAMs using the AXI high-performance (HP) interface. Then, sensor readings are transferred from the ARM core processing system (PS) to the accelerator using the AXI general-purpose (GP) interface. The accelerator outputs are extracted and evaluated offline to measure the performance.

TABLE 3
Summary of the trained Autoencoder-LSTM network model.

Layer	*Output Shape*	*Activation*	*Param #*
Input	(90,1)	-	0
FC1 (Dense)	(60,1)	tanh	5460
FC2 (Dense)	(30,1)	tanh	1830
LSTM	(40,1)	Sigmoid, tanh	6720
FC3 (Dense)	(20,1)	tanh	820
FC4 (Dense)	(1,1)	tanh	21
Total:			14,851

5.2 Hardware Resource Utilization

Table 4 shows the PL resource utilization of the AE-LSTM network using the proposed unified parallel CORDIC-based implementation with reference to sequential floating-point implementation and fixed-point implementation presented in [34]. In the reference sequential floating-point implementation and fixed-point implementation from [34], the network layers are stacked as presented in Table 3. Each Dense layer is implemented using one MAC unit and a single tanh activation. On the other hand, the LSTM layer consists of one reusable LSTM cell of 4 MACs, two sigmoid activations, and one tanh activation. Of the 111 DSPs used in the floating-point implementation, 35 are used for the MAC computations, while the activation functions use the rest. Each exponential computation in the floating-point activation requires 7 DSPs in addition to Flip-Flops (FF) and LUTs. In the fixed-point implementation, the DSP utilization has substantially been reduced using 20-bit fixed-point representation and piece-wise linear approximation of the activation functions.

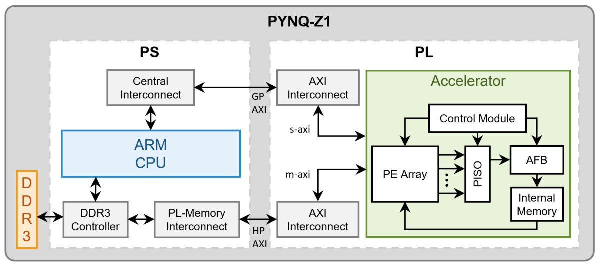

Fig. 10. Top-level diagram of the implemented accelerator on Xilinx PYNQ-Z1 FPGA development board.

TABLE 4
Resource utilization of the proposed design on Xilinx PYNQ-Z1 FPGA development board.

	LUT	*Slice Logic*	*FF*	*BRAM*	*DSP*
Avail.	53200	13300	106400	140	220
Reference Floating-Point Implementation presented in [34]					
Util.	24807	8498	22585	30	111
Util. %	46.63%	63.89 %	21.23%	21.43%	50.45%
Fixed-Point Implementation presented in [34]					
Util.	12544	4816	11922	28.50	17
Util. %	23.58%	36.21%	11.20%	20.36%	7.73%
Proposed Unified Parallel CORDIC-based Implementation					
Util.	36285	10912	24042	82.5	-
Util. %	68.20%	82.05%	22.60%	58.93%	-

Given that the proposed parallel CORDIC-based implementation is multiplier-less, DSP units are not utilized. The shift and add nature of the CORDIC-based PEs and AFB increases the LUT, slice logic, and FF utilization. Around 30% of the LUT utilization and 50% of the slice logic are caused by the PEs, and the rest are used by the activation functions, control module, and internal results storage. The outer product-based implementation of the matrix-vector multiplication in which each PE comprises an internal BRAM unit increases the utilization of distributed BRAMs compared to the reference designs in which each layer's parameters are kept together either in a single BRAM unit or multiple units.

The estimated response time of the proposed parallel CORDIC fixed-point implementation using the global $100MHz$ clock that drives the PL from the PS is $\sim 114\mu s$, achieving $\sim 1.8GOPS$ throughput. Compared to the sequential floating-point implementation with an estimated response time of $\sim 5ms$ and the fixed-point implementation with $\sim 1.13ms$ response time, the reduction is substantial considering the slight increase in the hardware resources detailed in Table 4.

5.3 Power Measurement

The Xilinx Power Analyzer is used to estimate the dynamic and static power of the proposed architecture on the PYNQ-Z1 XC7Z020 chip. Table 5 shows the analyzer breakdown. As per the breakdown, the accelerator consumes $\sim 0.438W$ on average. A USB power meter is used to verify the analyzer's total power consumption, including other board components. As shown experimentally in Fig. 11, the whole PYNQ-Z1 system board burns at most $\sim 2W$ when the accelerator is up and running. The meter reported power consumption is very close to the Zynq chip total noted in Table 5.

5.4 Efficiency Comparison

To illustrate the benefit of our proposed architecture, a comparison with an existing LSTM accelerator [41] designed using a comparable benchmark and implemented on the same FPGA board is detailed in Table 6. The table lists the chip name, target frequency, data type, number of operations, inference latency, throughput, and power consumption. It should be noted that the architecture in [41] has only accelerated the LSTM gates MxV and unloaded the results to the PS for activation. On the other hand, our proposed design

TABLE 5
Proposed architecture power consumption breakdown on PYNQ-Z1 XC7Z020 chip using Xilinx Power Analyzer.

Part		*Power (W)*	*Percentage*
Dynamic Power			
PS		1.256	67.5%
Accelerator	MAC Units	0.207	23.5%
	AFB	0.125	
	Control Module	0.002	
	BRAMs	0.004	
	Clocks	0.1	
Accelerator Total		**0.438**	
AXI Interconnects		0.011	0.5%
Static Power			
PL static		0.157	8.5%
Total		**1.862**	

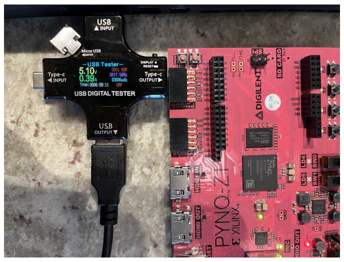

Fig. 11. Total power measurement using a USB power meter. When the accelerator is up and running, the whole PYNQ-Z1 system board burns at most $\sim 2W$.

accelerates both the LSTM MxV and activation performing all the required computations on the PL, thus reducing the I/O communication to parameters loading and final result unloading. With reference to the scheme described above, the latency and throughput reported in Table 6 for [41] are based on the MxV computation time, the I/O communication, and the activation computations delay were not reported in the paper.

In terms of design scalability, while considering the available resources on the PYNQ-Z1 FPGA, the authors in [41] reported that with 180 DSP-based PEs, the architecture utilizes around 97% of the available LUTs. Therefore, the maximum parallel MAC operations could not be extended beyond 180. On the other hand, our architecture could be expanded, given the available resources detailed in Table 4. More specifically, with each PE consuming on average 63

TABLE 6
Efficiency comparison with previous LSTM implementation on PYNQ-Z1 FPGA.

	Proposed	*[41]*
Chip	XC7Z020	XC7Z020
Frequency	100MHz	150MHz
Precision	fixed-16	fixed-16
Functions in Accelerator	Complete proposed architecture	Only MxV
Functions in ARM-core	None	Activation and point-wise operations
Operations	0.2M	0.1984M
# of PE	160	180
MxV Latency	$30.46\mu s$	$46.7\mu s$
MxV Throughput	6.5 GOPS	4.25 GOPS
Overall Latency	$114\mu s$	-
Overall Throughput	1.8 GOPS	-
Power	2 W	2.29 W

TABLE 7
Proposed architecture logic synthesis results using 45nm technology.

Total # of cells	$213,606$
Area	$870,523.8\ \mu m^2$
Cell Internal Power	$36.56\ mW$
Net Switching Power	$110.1\ mW$
Cell Leakage Power	$4.65\ mW$
Total Power	$0.151\ W$

LUT, 55 FF, and 35 slice logic, the number of PEs could be raised to 216. The utilization report of our expanded architecture on PYNQ-Z1 FPGA shows that the expansion resulted in 90.6%, 30%, and 99.8% utilization of the available LUT, FF, and Slice Logic, respectively.

To further demonstrate the proposed architecture's efficiency, the architecture's RTL is synthesized, and Design Compiler-Synopsys produced results presented in Table 7. It can be observed that the proposed design achieves sufficiently low area and power compared to the FPGA-based implementation.

5.5 Network Accuracy

The accuracy of the implemented prediction network is evaluated on the test set from the Grand St. Bernard dataset to study the impact of using CORDIC-based computing units on the overall network prediction performance. The implemented CORDIC-based prediction network achieves a mean absolute error of 20×10^{-3}. Compared with the reference floating-point version, with 16×10^{-3} MAE, the performance degradation may be acceptable in many applications given reduced bit width and CORDIC quantization error. Fig. 12 presents an example of system response collected from the PYNQ-z1 board to sensor readings from the same dataset. The results are presented in the $[-1, 1]$ scale, which is the scale the AE-LSTM network uses for prediction. The absolute difference between the sensor data and the AE-LSTM system prediction is also presented in both plots. With reference to the floating-point implementation, the effect of quantization noise on network prediction is more evident in the region where the sensor data are less noisy.

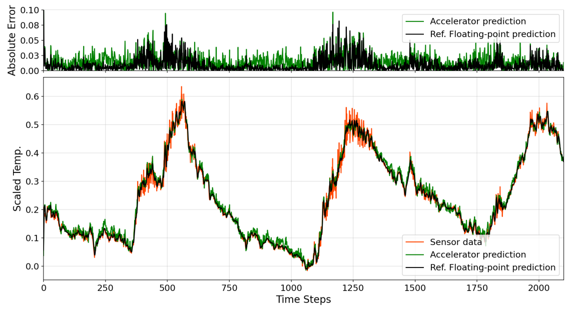

Fig. 12. Prediction network results collected from the PYNQ-Z1 board. The bottom plot shows actual sensor readings (orange), corresponding CORDIC-based accelerator predictions (green), and reference floating-point predictions (black). The top plot presents the absolute difference between sensor observations and the network prediction for the CORDIC and floating-point implementations.

6 Conclusion

This paper presented a novel unified configurable parallel CORDIC-based architecture for accelerating recurrent LSTM and non-recurrent fully connected computations in DNN architectures. The implemented solution consists of a systolic ring of outer product-based processing elements and a single reusable activation function block. The outer product generates and accumulates partial sums in parallel, eliminating data dependencies and increasing hardware utilization and system throughput. The serial processing nature of the input vector in the outer product-based scheme enables a single activation function block (AFB) to serve the whole network, decreasing overall system complexity. The CORDIC implementation of the PEs and activation function block makes the proposed solution generic for FPGA and ASIC platforms. In addition, given the CORDIC algorithm's ability to perform many elementary functions using the same shift and add scheme, the utilization of the proposed CORDIC-based computing kernels could further be extended to other functionality beyond the MAC units and activation functions. Experimental validation of the proposed hardware architecture on a resource-constrained Xilinx PYNQ-Z1 development board using an open-source time-series dataset achieves low average latency and power consumption, making the proposed solution suitable for resource-constrained IoTs and edge platforms. Future work could exploit automating the generation of efficient CORDIC-based architecture for accelerating recurrent LSTM and non-recurrent fully connected computations in DNN architectures while considering application requirements and available hardware resources.

References

[1] I. Arel, D. C. Rose and T. P. Karnowski, "Research frontier: Deep machine learning-a new frontier in artificial intelligence research, " Comp. In tell. Mag., vol. 5, no. 4, pp. 13-18, Nov. 2010.
[2] W. G. Hatcher and W. Yu, "A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends," in IEEE Access, vol. 6, pp. 24411-24432, 2018, doi: 10.1109/ACCESS.2018.2830661.
[3] S. Shamshirband, T. Rabczuk and K. -W. Chau, "A Survey of Deep Learning Techniques: Application in Wind and Solar Energy Resources," in IEEE Access, vol. 7, pp. 164650-164666, 2019, doi: 10.1109/ACCESS.2019.2951750.
[4] T. W. S. Chow, Xiao-Dong Li and Yong Fang, "A real-time learning control approach for nonlinear continuous-time system using recurrent neural networks," in IEEE Transactions on Industrial Electronics, vol. 47, no. 2, pp. 478-486, April 2000, doi: 10.1109/41.836364.
[5] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh and K. Shaalan, "Speech Recognition Using Deep Neural Networks: A Systematic Review," in IEEE Access, vol. 7, pp. 19143-19165, 2019, doi: 10.1109/ACCESS.2019.2896880.
[6] T. Young, D. Hazarika, S. Poria and E. Cambria, "Recent Trends in Deep Learning Based Natural Language Processing [Review Article]," in IEEE Computational Intelligence Magazine, vol. 13, no. 3, pp. 55-75, Aug. 2018, doi: 10.1109/MCI.2018.2840738.
[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Comput., vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco. 1997.9.8.1735.
[8] N. P. Jouppi et al., "A domain-specific supercomputer for training deep neural networks," Commun. ACM, vol. 63, no. 7, pp. 67-78, Jun. 2020.
[9] J. Wang, J. Lin and Z. Wang, "Efficient Hardware Architectures for Deep Convolutional Neural Network," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 65, no. 6, pp. 1941-1953, June 2018, doi: 10.1109/TCSI.2017.2767204.
[10] T. Yuan, W. Liu, J. Han and F. Lombardi, "High Performance CNN Accelerators Based on Hardware and Algorithm Co-Optimization," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 68, no. 1, pp. 250-263, Jan. 2021, doi: 10.1109/TCSI.2020.3030663.

[11] Y. Lin and T. S. Chang, "Data and Hardware Efficient Design for Convolutional Neural Network," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 65, no. 5, pp. 1642-1651, May 2018, doi: 10.1109/TCSI.2017.2759803.

[12] F. Conti, P. D. Schiavone and L. Benini, "XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 11, pp. 2940-2951, Nov. 2018, doi: 10.1109/TCAD.2018.2857019.

[13] R. Andri, L. Cavigelli, D. Rossi and L. Benini, "YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 1, pp. 48-60, Jan. 2018, doi: 10.1109/TCAD.2017.2682138.

[14] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," 2016 26th International Conference on Field cite and Applications (FPL), 2016, pp. 1-4, doi: 10.1109/FPL.2016.7577314.

[15] D. Kalamkar et al., "A study of BFLOAT16 for deep learning training," arXiv:1905.12322, 2019, [online] Available: http://arxiv.org/abs/1905.12322.

[16] N. Ho and W. Wong, "Exploiting half precision arithmetic in Nvidia GPUs," 2017 IEEE High Performance Extreme Computing Conference (HPEC), 2017, pp. 1-7, doi: 10.1109/HPEC.2017.8091072.

[17] Q. He et al., "Effective quantization methods for recurrent neural networks," arXiv:1611.10176, 2016, [online] Available: http://arxiv.org/abs/1611.10176.

[18] M. Z. Alom, A. T. Moody, N. Maruyama, B. C. Van Essen and T. M. Taha, "Effective Quantization Approaches for Recurrent Neural Networks," 2018 International Joint Conference on Neural Networks (IJCNN), 2018, pp. 1-8, doi: 10.1109/IJCNN.2018.8489341.

[19] L. Hou, J. Zhu, J. Kwok, F. Gao, T. Qin and T.-Y. Liu, "Normalization helps training of quantized LSTM," Proc. Adv. Neural Inf. Process. Syst., pp. 7344-7354, 2019.

[20] S. Han, J. Pool, J. Tran and W. Dally, "Learning both weights and connections for efficient neural network," Proc. Adv. Neural Inf. Process. Syst., vol. 28, pp. 1135-1143, 2015.

[21] S. Han, H. Mao and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning trained quantization and Huffman coding," Proc. Int. Conf. Learn. Represent., pp. 1-14, 2016.

[22] S. Anwar, K. Hwang and W. Sung, "Structured pruning of deep convolutional neural networks," ACM J. Emerg. Technol. Comput. Syst., vol. 13, no. 3, pp. 32, 2017.

[23] S. Han et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, pp. 75-84, Feb. 2017.

[24] S. Cao et al., "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays, pp. 63-72, Feb. 2019.

[25] D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti and J. Seo, "An 8.93 TOPS/W LSTM Recurrent Neural Network Accelerator Featuring Hierarchical Coarse-Grain Sparsity for On-Device Speech Recognition," in IEEE Journal of Solid-State Circuits, vol. 55, no. 7, pp. 1877-1887, July 2020, doi: 10.1109/JSSC.2020.2992900.

[26] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt and A. Moshovos, "Stripes: Bit-serial deep neural network computing," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783722.

[27] O. Bilaniuk, S. Wagner, Y. Savaria and J. David, "Bit-Slicing FPGA Accelerator for Quantized Neural Networks," 2019 IEEE International Symposium on Circuits and Systems (ISCAS), 2019, pp. 1-5, doi: 10.1109/ISCAS.2019.8702332.

[28] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu and J. Han, "Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications," in Proceedings of the IEEE, vol. 108, no. 12, pp. 2108-2135, Dec. 2020, doi: 10.1109/JPROC.2020.3006451.

[29] M. S. Kim, A. A. Del Barrio Garcia, H. Kim and N. Bagherzadeh, "The Effects of Approximate Multiplication on Convolutional Neural Networks," in IEEE Transactions on Emerging Topics in Computing, doi: 10.1109/TETC.2021.3050989.

[30] M. S. Ansari, B. F. Cockburn and J. Han, "An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing," in IEEE Transactions on Computers, vol. 70, no. 4, pp. 614-625, 1 April 2021, doi: 10.1109/TC.2020.2992113.

[31] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," in IRE Transactions on Electronic Computers, vol. EC-11, no. 4, pp. 512-517, Aug. 1962, doi: 10.1109/TEC.1962.5219391.

[32] Y. -J. Chang, Y. -C. Cheng, S. -C. Liao and C. -H. Hsiao, "A Low Power Radix-4 Booth Multiplier With Pre-Encoded Mechanism," in IEEE Access, vol. 8, pp. 114842-114853, 2020, doi: 10.1109/ACCESS.2020.3003684.

[33] H. Zhang, H. Xiao, H. Qu and S. -B. Ko, "FPGA-Based Approximate Multiplier for Efficient Neural Computation," 2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), 2021, pp. 1-4, doi: 10.1109/ICCE-Asia53811.2021.9641971.

[34] N. A. Mohamed and J. R. Cavallaro, "Real-time FPGA-Based Outlier Detection using Autoencoder and LSTM," 2021 55th Asilomar Conference on Signals, Systems, and Computers, 2021, pp. 1195-1199, doi: 10.1109/IEEECONF53345.2021.9723300.

[35] J. E. Volder, "The CORDIC Trigonometric Computing Technique," in IRE Transactions on Electronic Computers, vol. EC-8, no. 3, pp. 330-334, Sept. 1959, doi: 10.1109/TEC.1959.5222693.

[36] J. S. Walther, "A unified algorithm for elementary functions," Proceedings of the May 18-20, 1971, spring joint computer conference on - AFIPS '71 (Spring), 1971.

[37] X. Hu, R. G. Harber and S. C. Bass, "Expanding the range of convergence of the CORDIC algorithm," in IEEE Transactions on Computers, vol. 40, no. 1, pp. 13-21, Jan. 1991, doi: 10.1109/12.67316.

[38] S. Wang, V. Piuri and E. E. Swartzlander, "Hybrid CORDIC algorithms," in IEEE Transactions on Computers, vol. 46, no. 11, pp. 1202-1207, Nov. 1997, doi: 10.1109/12.644295.

[39] T. Yang et al., "Design Space Exploration of Neural Network Activation Function Circuits," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 10, pp. 1974-1978, Oct. 2019, doi: 10.1109/TCAD.2018.2871198.

[40] G. Barrenetxea. (2019). Sensorscope Data [Data set]. Zenodo. https://doi.org/10.5281/zenodo.2654726

[41] X. Zhang, W. Jiang and J. Hu, "Achieving Full Parallelism in LSTM via a Unified Accelerator Design," 2020 IEEE 38th International Conference on Computer Design (ICCD), 2020, pp. 469-477, doi: 10.1109/ICCD50377.2020.00086.

[42] N. A. Mohamed and J. R. Cavallaro, "Design and Implementation of Autoencoder-LSTM Accelerator for Edge Outlier Detection," 2021 IEEE Workshop on Signal Processing Systems (SiPS), 2021, pp. 134-139, doi: 10.1109/SiPS52927.2021.00032.

Nadya A. Mohamed received the B.S. degree in Information Technology Computer System Engineering major from United Arab Emirates University, Al Ain, UAE, in 2008, and the M.S. degree in Microsystems Engineering from Masdar Institute of Science and Technology, Abu Dhabi, UAE, in 2012. She is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Rice University, Houston, TX, USA. Her current research interests include deep learning hardware acceleration with a focus on time series applications and resource management in wireless sensor networks and Internet-of-Things systems.

Joseph R. Cavallaro received the B.S. degree from the University of Pennsylvania, Philadelphia, Pa, in 1981, the M.S. degree from Princeton University, Princeton, NJ, in 1982, and the Ph.D. degree from Cornell University, Ithaca, NY, in 1988, all in electrical engineering. In 1988, he joined the faculty of Rice University, Houston, TX, where he is currently a professor of electrical and computer engineering. His research interests include computer arithmetic, and DSP, GPU, FPGA, and VLSI architectures for applications in wireless communications.