An FPGA Implementation of Stochastic Computing-based LSTM

Guy Maor*
ECE Department
The University of Texas at Dallas
Richardson, U.S.
Guy.Maor@utdallas.edu

Xiaoming Zeng*
ECE Department
The University of Texas at Dallas
Richardson, U.S.
Xiaoming.Zeng@utdallas.edu

Zhendong Wang*
ECE Department
The University of Texas at Dallas
Richardson, U.S.
Zhendong.Wang@utdallas.edu

Yang Hu
ECE Department
The University of Texas at Dallas
Richardson, U.S.
Yang.Hu4@utdallas.edu

***Abstract*—As a special type of recurrent neural networks (RNN), Long Short Term Memory (LSTM) is capable of processing sequential data with a great improvement in accuracy and is widely applied in image/video recognition and speech recognition. However, LSTM typically possesses high computational complexity and may cause high hardware cost and power consumption when being implemented. With the development of Internet of Things (IoT) and mobile/edge computation, lots of mobile and edge devices with limited resources are widely deployed, which further exacerbates the situation. Recently, Stochastic Computing (SC) has been applied into neural networks (NN) (e.g., convolution neural networks, CNN) structure to improve the power efficiency. Essentially, SC can effectively simplify the fundamental arithmetic circuits (e.g., multiplication), and reduce the hardware cost and power consumption. Therefore, this paper introduces SC into LSTM and creatively proposes an SC-based LSTM architecture design to save the hardware cost and power consumption. More importantly, the paper successfully implements the design on a Field Programmable Gate Array (FPGA) and evaluates its performance on the MNIST dataset. The evaluation results show that the SC-LSTM design works smoothly and can significantly reduce power consumption by 73.24% compared to the baseline binary LSTM implementation without much accuracy loss. In the future, SC can potentially save hardware cost and reduce power consumption in a wide range of IoT and mobile/edge applications.**

***Index Terms*—LSTM, stochastic computing, mobile and edge devices, hardware resources and power efficiency, accuracy**

I. Introduction

Recurrent neural networks (RNN) [1] have shown significant competency for a wide variety of applications, such as image/video recognition and speech recognition. As a special type of RNN, Long Short Term Memory (LSTM) [2], [3] successfully tackles the issue of vanishing gradient descent for long time dependency data, which makes it great for sequential data processing. Traditionally, an LSTM model is designed to be deployed on high-performance CPU or GPU architectures located in the cloud site, since it inherently involves high computational complexity, resource consumption, and energy cost.

However, the wide-spread of Internet of Things (IoT) and concerns about energy, latency and privacy have pushed the intelligence deployment site on the verge of a major shift, from the beefy cloud system to wimpy resource/energy-restricted edge devices. We are expecting more neural networks running on embedded platforms such as FPGA and ASIC in the near future. Existing solutions of implementing hardware accelerators for LSTM mainly focus on shrinking the deep neural networks (DNN) model size via weight pruning and quantization [4]–[6]. Though they show promising results, the DNN structure is still relatively large for those resource-/power-restricted IoT/edge devices. We argue that the architecture design of edge-deployed LSTM accelerators should be explored from a unconventional angle.

In this work, we set out to explore implementing edge-based LSTM hardware accelerators in a manner of bitstream processing. Our proposal is motivated by the fact that many edge-deployed embedded accelerators require to process input raw data (e.g. sound and speech) as form of *bitstream* [7], which is naturally fit to the LSTM applications. However, a conventional end-to-end speech recognition system involves the conversion of bitstream data to LSTM recognizable binary data and resource-hungry LSTM processing steps. It may also need another conversion of processed data back to bitstream data. We can observe that a conventional binary-based LSTM architecture is not efficient for processing bitstream data.

To accommodate the bitstream data processing without additional conversion, we employ stochastic computing (SC) that enables efficient arithmetic circuits implementations (e.g., multiplication, addition, nonlinear functions) [8], [9]. The stochastic computing has been adopted to optimize DNN (e.g., convolution neural networks) [10]–[12], which significantly improves the hardware resource and power efficiency only with trivial accuracy loss.

* Authors with equal contribution

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00014

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

In this paper, we propose a stochastic computing (SC)-based LSTM inference engine, SC-LSTM, by successfully applying SC into the complex LSTM model and leveraging the high hardware efficiency of SC circuits to effectively reduce the resource and power consumption of LSTM structure. Specifically, we substitute the computation units in the conventional LSTM architecture with the computation units of SC architecture, which may inherently save the hardware cost and power consumption due to the high hardware efficiency.

First, we propose a series of hardware-efficient SC circuit implementations of the LSTM architecture. Considering that LSTM is a vector processing structure, the process of calculating each element of the output vector is assigned to an individual hardware core in our design. This will thus construct a multi-core design to achieve maximum parallelism on hardware. To avoid potential accuracy lost(representing range overflow), a single core is designed to employ a stochastic-binary hybrid method (i.e., combining the binary-circuit design and SC-circuit design methods).

Second, together with the output of the previous time-step, the input data of each core is converted to an array of stochastic values. These values are then passed into a gating module that is in charge of SC dot-product and SC activation function approximations. In order to retain the temporal behavior of LSTM, cell state computation module is cautiously designed to expand the representing range for the temporal value of cell state from [-1,1] to [-B,B]. Besides, we implement the addition functions as both APC-based [13] and MUX-based and obtain the optimized design with the best performance trade-off.

Finally, we implement our SC-LSTM design on Xilinx Zedboard FPGA and evaluate our design. We first implement a conventional binary LSTM module on an FPGA platform as a baseline. We evaluate our design with a set of experiments on the MNIST [14] dataset. We test the SC-LSTM implementation under different settings of core count and time-step window size to evaluate our design in terms of power consumption, accuracy, and runtime. In addition, we compare the performance of the SC-LSTM design with the baseline. Our test results show that the recognition accuracy increases from 66.12% to 90.75% for the APC implementation and 73.68% to 92.71% for the MUX implementation with the core-count increasing from 8 cores to 16 cores. Accordingly, the power consumption increases by 132% for the APC implementation and 85% for the MUX implementation. Notably, by increasing the window size from 2^{12} to 2^{16}, the accuracy increases by 9.43% for the APC implementation and 38.08% for the MUX implementation. Comparing to the binary LSTM baseline, both the APC-based and the MUX-based design consume much less power. The power consumption of MUX-based implementation is 73.24% less than binary baseline.

Overall, this paper makes the following contributions:

- We introduce a scalable hardware-efficient SC circuit implementation of the LSTM inference engine. To the best of our knowledge, this is the first scalable SC based LSTM architecture implementation.
- We creatively propose a SC addition design between two bit-stream with different weight, and we successfully validate the SC-LSTM design on an FPGA board with a set of experiments to evaluate the performance of the design in terms of power consumption, accuracy and runtime.
- Considering that SC can greatly simplify the fundamental arithmetic circuit, especially multiplication, which is largely involved in LSTM model, the SC-LSTM design can significantly reduce the power consumption of the complex LSTM structure without much loss of accuracy when being compared to the binary LSTM design, which would benefit the application of LSTM in many mobile and edge devices.

The remainder of this paper is organized as follows: Section II introduces background knowledge involved in the system design, including LSTM, SC, etc. Section III elaborates our design on the SC-LSTM structure. Section IV shows the evaluation results. Section V discusses the related work. Section VI concludes the paper.

II. Background

A. LSTM

LSTM is one of the mostly used RNN structure. RNN is a class of deep neural networks (DNN) where connections between nodes form a directed graph along a temporal sequence and thus, exhibits temporal dynamic behavior. RNN is greatly applicable to tasks such as unsegmented and connected handwriting recognition or speech recognition. In practice, LSTM can utilize gating mechanism to effectively mitigate the issue of the vanishing gradient during the training. Therefore, LSTM can effectively avoid catastrophic errors and typically achieve higher performance in terms of accuracy compared to the primitive RNN.

LSTM is explicitly designed to avoid the long-term dependency problem and able to remember information for long periods. Specifically, LSTM uses gating mechanisms to operate on a cell state (i.e., a vector that contains the memory of the LSTM) at time-step t, C_t, and each gate is responsible for determining how much information is allowed to be remembered or forgotten from the cell state. Typically, there are four types of gates, each of which is represented by a neuron operation, including: Input gate (I), Cell gate (C′), Forget gate (F), and Output gate (O). Each gate is the output of a matrix multiplication, a bias addition, and an activation function. The following equations represent the functions used to calculate the next hidden state, h_t [15]:

$$F = sigmoid(W_f \cdot [h_{t-1}, x_t] + B_f) \tag{1}$$

$$I = sigmoid(W_i \cdot [h_{t-1}, x_t] + B_i) \tag{2}$$

$$C^{'} = tanh(W_c \cdot [h_{t-1}, x_t] + B_c) \tag{3}$$

$$O = sigmoid(W_o \cdot [h_{t-1}, x_t] + B_o) \tag{4}$$

$$C_t = C_{t-1} * F + I * C^{'} \tag{5}$$

$$h_t = tanh(C_t) \cdot O \tag{6}$$

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

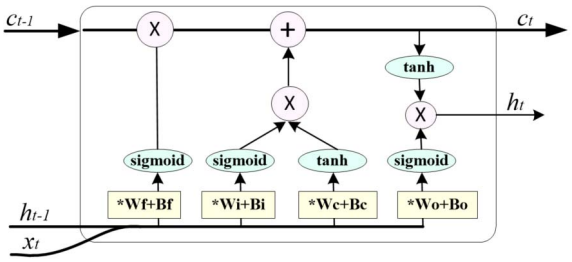

Fig. 1: The internal structure of a LSTM layer.

LSTM actually has the form of a chain of repeating modules of NN. Fig. 1 only shows the internal structure of an LSTM layer. The input to the LSTM layer is the current input, x_t, concatenated with the output of previous time step, h_{t-1}, and the output is h_t. In LSTM, there are two important non-linear activation functions that are critical to our design, sigmoid ($sigm$), and hyperbolic tangent ($tanh$). The $sigm$ function is the activation function that is a part of the forget gate, input gate, and output gate, while the $tanh$ function is the activation function of the cell gate.

B. Stochastic Computing

In conventional computing, we represent numbers using base 2 notation, i.e., each bit is represented by either 0 or 1. This notation does not change with time (i.e., no matter how many clock cycles of the system pass, the bits don't change). Meanwhile, we map n bits of a number to an n-bits bus (i.e., the 0 or 1 corresponds to the high or low level voltage in digital circuit). However, in SC, the number is represented with a probability p, [16]. The probability, p, is associated with a stochastic value, v, which is represented by using a single bit and may change with each system clock cycle. Each clock cycle, any wire storing a stochastic value has a probability, p, of being a 1 bit, and probability, $1-p$, of being a 0 bit. The probability, p, defines what that value being stored represents. For example, we are reading a stochastic value for 100 clock cycles. During those 100 cycles, we notice that 30 of those clock cycles, the bit is 1; in this case, we say the probability of that wire bit is 0.3.

There are two ways of interpreting a stochastic value, v [16]. The first way is non-polarized. Non-polarized maps the probability, p, directly to its logical value, v. If p is 0.3, we say the logical value, v, is 0.3 as well. The problem with non-polarized mapping is that we cannot represent negative numbers. The second way of interpreting a stochastic value, v, is polarized. Specifically, we map the value of p from the range [0,1] to the logical value, v, within the range [-1,1]. Thats to say, if p is 0.5, the mapped value, v, is 0. Compared to non-polarized way, polarized allows us to represent values as negative numbers. In our SC-LSTM design, we use polarized way to interpret stochastic values considering that some parameters, like weights, can be negative. The following equations represent the specific mappings between the probability, p, and the logical value, v:

$$p = (v+1)/2 \tag{7}$$

$$v = 2*p-1 \tag{8}$$

1) Multiplication: Matrix multiplications are widely used in LSTM model; however, in conventional computing, the implementation of multiplication on hardware is very expensive, which indispensably involves complex arithmetic circuit combinations. In contrast, the multiplier in SC can be efficiently implemented merely with a simple XNOR gate [16]. Since the output of an XNOR gate is 1 if only both the inputs are the same values (i.e., either 1 or 0 simultaneously), the output probability is the probability of both inputs being the same. In the following equations, p_1 and p_2 represent the probabilities of inputs 1 and 2 being 1s, and p_{out} represents the output probability. The values v_1, v_2, and v_{out} represent the stochastic values of the probabilities p1, p2, and p_{out}, respectively.

$$p_{out} = P(input_1 == input_2) \tag{9}$$

$$\begin{aligned} p_{out} = {} & P(input_1 == 1 \; and \; input_2 == 1) \\ & + P(input_1 == 0 \; and \; input_2 == 0) \end{aligned} \tag{10}$$

$$p_{out} = p_1 * p_2 + (1-p_1)*(1-p_2) \tag{11}$$

Based on equation 7, we can substitute the probability values, p, in equation 11 above with the stochastic values, v, as follows:

$$\begin{aligned} (v_{out}+1)/2 = {} & (v_1+1)/2 * (v_2+1)/2 \\ & +(1-(v_1+1)/2)*(1-(v_2+1)/2) \end{aligned} \tag{12}$$

$$v_{out} = v_1 * v_2 \tag{13}$$

2) Addition: In stochastic computing, there are multiple approaches to implement addition operation [17]. Even so, chances are that adding two stochastic values together can produce a value outside of the allowable range. In this case, we have no choice but to resort to a stochastic-binary hybrid circuit design to achieve an addition operation. To perform addition, intuitively, we use an Approximate Parallel Counter (APC) [13], which takes in N inputs and produces a binary output value that equals the number of 1 bit at the input. The downside of doing so is that the APC has to involve a large number of adders and may consume significant power. Alternatively, instead of an APC, a MUX can be leveraged to achieve the addition functionality as well, which can effectively mitigate the issue of high power consumption at the expense of higher runtime.

III. Design

A. Top Level Design

In this section, we discuss the top-level design of the whole system architecture. Fig. 2 shows the main modules involved in the architecture, which is mainly composed of two RAM modules, a memory writer and stochastic memory, the SC-LSTM module, and output port. The two RAM modules mainly contain the data (i.e., parameter and input data) to be written to Stochastic Memory (SM). Specifically, Data RAM

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

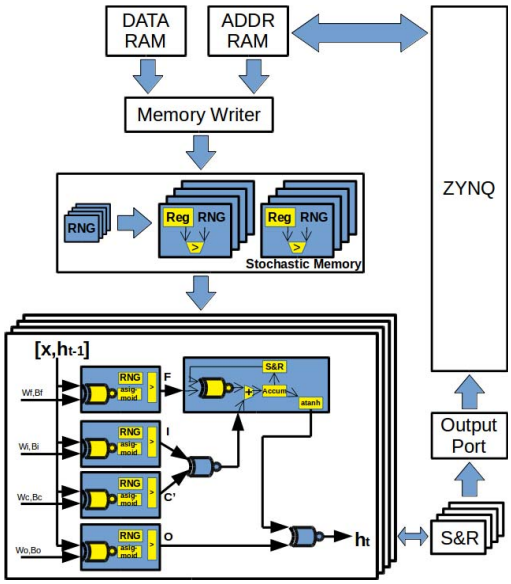

Fig. 2: The top level design of the SC-LSTM architecture.

contains the binary values of the parameter and input while address RAM (Addr RAM) contains the locations where the corresponding values will be written into SM. The Memory Writer (Mem Writer) sequentially reads data from both block RAMs and stores the values in SM where their addresses are specified in Addr RAM. The primary objective of SM is to convert binary data into stochastic values. It contains stochastic registers for the parameter and input data. Once the binary data is converted into stochastic value through SM, the value goes into the SC-LSTM module, which is constructed by utilizing a multi-core design structure to perform the main SC-LSTM algorithm. Finally, the output port allows the user to read the output values of the SC-LSTM module.

B. SC-LSTM module

1) multi-core Design: This section discusses how the scalable SC-LSTM module is constructed through a multi-core design approach. In essence, LSTM is a function going from a vector to another vector. It begins by performing matrix multiplication on the input vector concatenated with the previous output vector. After that, all the other functions in the LSTM model are achieved based on element-wise operations (i.e., element-wise sigmoid, element-wise tanh, element-wise multiplication, and element-wise addition). In this case, when these operations are implemented on hardware, the operation of each element can be assigned to a core, which drives us to construct the module through a multi-core design. That's to say, each hidden output of the SC-LSTM is assigned to a single core to be processed. Fig. 3 (a) illustrates the multi-core design with an example of the output dimension being m, which indicates that the m cores are required to process the computation. Since our HDL implementation is all parameterized, which makes our design scalable with the model size changes.

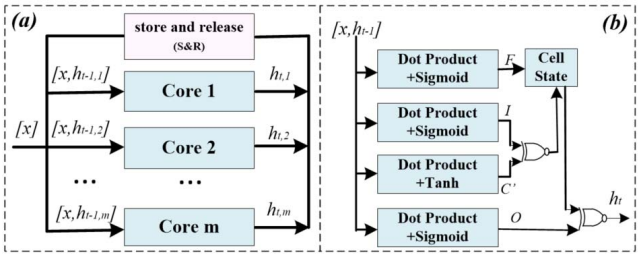

Fig. 3: (a) the multi-core design with an m-dimension output; (b) the internal structure of each core.

Next, we further explore the internal structure of each core in the multi-core design. In short, the single core is designed by using a stochastic-binary hybrid method (i.e., combining the binary-circuit design and SC-circuit design methods). The input to the core is an array of stochastic values. The stochastic values are passed into $DotProduct + Activation$ neuron, which is shown in Fig. 3 (b). These functions perform the matrix multiplications and activation functions of different gates. The outputs of the Input and Cell gates are multiplied with each other using element-wise multiplication. Both the result of the multiplication and the Forget gate go into the cell state, which is in charge of maintaining the cell state value as well as applying a $tanh$ to the cell state before passing it to the output where its multiplied with the Output gate using an XNOR gate.

2) Neuron: The entire design of a Neuron cannot easily be implemented using only SC. Instead, we use a Stochastic-Binary hybrid. As we mentioned before, either an APC or a MUX can be adopted to construct a Neuron. Fig. 4 shows the internal structure of the Neuron [18]. The inputs to this module are composed of the input vector and the weights. The input vector is the x vector concatenated with the previous time-step output, h_{t-1}. The input vector is multiplied element-wise with the weight vector, then, the result is passed into the APC or MUX to complete the matrix multiplication.

Because the result of APC-addition is non-deterministic, the output result will not always be consistent. We would rather average the result over 2^M (number of data to be accumulated) clock cycles. The more clock cycles, the more consistent the result will be. We use an Accumulator to sample the results for 2^M samples. Rather than dividing the Accumulator result, we leave the result undivided to retain information. If the result is L bits, the Accumulator result will be M+L bits. After the Accumulator step, the next step is to perform the activation function. Theoretically, there are two types of activation functions: $Tanh$ and $Sigmoid$. In practice, both activation functions are approximated by using Binary Piecewise Functions. Depending on which function, different Piecewise functions are used. A RNG with two types of different action functions are described in equations (14) and (15). Equation(16)-(17) and (18) depicts the comparator with function $Tanh$ and $Sigmoid$, respectively.

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

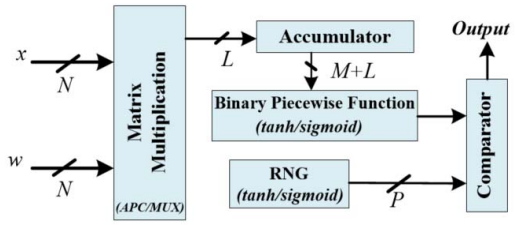

Fig. 4: The internal structure of Dot Product with Activation Neuron within each core.

$$RNG_{sigmoid} : [-n\_window\_size * 4, \; n\_window\_size * 4] \tag{14}$$

$$RNG_{tanh} : [-n\_window\_size, \; n\_window\_size] \tag{15}$$

$$Sigmoid_{temp} = Accumulator * 2 - N * n\_window\_size + n\_window\_size * 2 \tag{16}$$

$$Sigmoid = [max(Sigmoid_{temp}, 0) > RNG_{sigmoid}] \; ? \; 1 \; : \; 0 \tag{17}$$

$$Tanh = [(Accumulator * 2 - N * n\_window\_size) > RNG_{tanh}] \; ? \; 1 \; : \; 0 \tag{18}$$

3) Store And Release: In this section, we discuss the concept of a store and release module (S&R), which is critical to the implementation of the SC-LSTM module. The module's goal is to read in a stochastic input for one time window, which is defined by the number of clock cycles, and use it as the output for the next window. The motivation behind this is that we can emulate the temporal behavior of a cell state and hidden state of the LSTM model. Fig. 5 shows the structure of the (S&R) module, which is composed of two components, the store and the release. The store component's job is to count the number of 1 bits from the input and accumulate it till the next time window. At the end of the window, the accumulated value is passed to the release component and the store component is reset to 0. In the release component, we utilize the stored value to generate a bit-stream of random values proportional to the stored value. We do so by leveraging a random number generator (RNG) to generate a random integer, which will be compared with the stored value. If the random integer is less than or equal to the stored value, a 1 is outputted. Otherwise, its a 0.

For example, lets assume that during the store phase, an input bit-stream has a probability 0.3 of being a 1 bit. With a window size of 1000 bits, the expected value to be stored is 300 (3 out of every 10 bits is a 1). During the release phase, a number between 1 and 1000 is generated at random for multiple times. The probability that the number will be less than 300 is 0.3. This indicates that the output will be a bit-stream with a probability of 0.3 which is the same as the input.

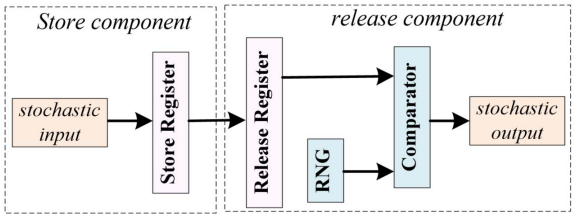

Fig. 5: The structure of store and release component.

In order for the store and release module to work properly, we have to create an RNG that produces a random integer between 0 and the window size, W. Any other range and the output probability will not match the input probability. To implement an RNG, we use Linear Feedback Shift Registers (LFSR) [19]. Using a prime polynomial of degree 32, we can construct a 32 bit random number generator. If we want fewer bits, we can mask out the other bits we do not need. This, however, means we can only produce an RNG with a range that is a power of 2. This means that we must restrict the window size, W, to a power of 2 as well.

4) Cell State: In this section, we discuss how the cell state is implemented and the issues we had with implementing it. The biggest issue with implementing the cell state is the fact that it is an unbounded value. The value of the cell state can theoretically approach infinity as the number of time-steps approaches infinity. As we discussed before, with stochastic computing, we can only represent stochastic values within the range of [-1,1]. To solve this issue, we have to come up with a way of representing stochastic values in a wider range. Rather than map the probability value, p, from [0,1] to [-1,1], we select a bound value, B, and map p from [0,1] to [-B, B]. For instance, the probability, p = 0.75, maps to the logical value, $B/2$; once the cell state reaches B or $-B$, it saturates. If the value, B, is too large, it will decrease the accuracy, while if B is too small, the cell state may saturate too early, leading to computational errors. Therefore, it's critical to select a proper B value that is neither too large nor too small. Fig. 6 shows the internal structure of the cell state module. We modify the S&R and use it to implement the temporal behavior of the cell state. The previous cell state value (c_{t-1}) is the output of the $S\&R$ module. We multiply (c_{t-1}) with the forget gate (F) as described in equation 5. Next, we add the resulting value with $I * C^{'}$ ($I^{'}$) in binary (1 if the input is 1, -1 if the input is 0). Considering that the cell state's range is B times larger than the range of $I^{'}$, the cell state is weighted B times as $I^{'}$ is weighted. Finally, the output binary value is accumulated back into the S&R. The final step is to use a tanh function on the cell state. Similar to a neuron, we use linear approximation to approximate the tanh function. We pass the current window cell state value into an accumulator. If the current window cell state is L bits, the accumulator value will be L+M bits as the number of samples is 2^M. An M bit RNG produces a signed number which is compared to the Accumulator value. The output is an aproximation of the tanh function on the cell state.

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

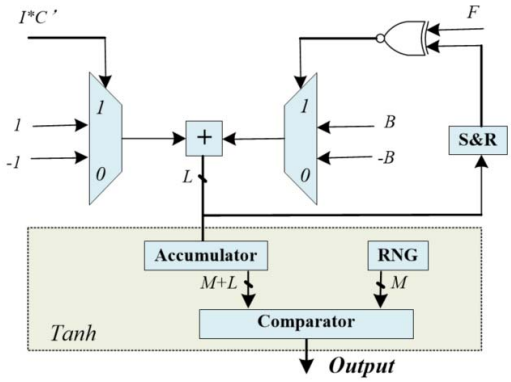

Fig. 6: The structure of cell state.

C. Stochastic Memory

In order to implement an NN on a stochastic architecture, the parameters and input data have to be stored as stochastic values. As we stated in the top level design, the SM module mainly works to convert a binary value into a stochastic value. Stochastic Memory is composed of stochastic registers and random number generators. Each stochastic register contains a register for storing a binary value, while the RNG and comparator are critical to achieving the converting mechanism. Each stochastic register contains a register for storing a single binary value and a comparator for converting the stored value to a stochastic value. Specifically, the comparator reads in both the binary value and the random number generated by a RNG. If the random number is less than the binary value, it outputs a 1, otherwise, it outputs a 0. In order to generate a stochastic value correctly, the size of the RNG, in terms of bits, must be as the same as that of the binary value. In other words, the largest possible binary value, generated by the RNG is mapped to a stochastic 1, while the smallest possible binary value generated by the RNG, is mapped to a stochastic -1.

Due to the large structure of the SM, the SM consumes a lot of power. Therefore, we adopt a couple of approaches to save resources and power. First, we quantify the input data and parameters. Rather than store all 32 bits of a binary value, we limit the number of bits we used to represent a parameter or data value. This not only reduces the number of flip-flops used to store a value, but also halves the precision with each bit we remove. Second, we reduce the number of RNGs we use by sharing them among stochastic registers. Generally, sharing RNG may cause interference to the computation results and is supposed to be avoided at all costs. However, in our case, if two stochastic values produced by stochastic registers do not go into the same Neuron, it is acceptable for those two stochastic registers to share an RNG. This is because when the stochastic values go into the Neuron, the data is converted into binary and the random sequences are lost. Once the random sequences are lost, it is impossible to tell if the random sequences were produced by the same RNG.

In order to access the address locations we want as easily as possible, we created an easy addressing scheme for the stochastic memory, which is shown Table I. Specifically, each stochastic register has an address composed of 4 components, which includes the register domain, gate, core, and index in the core. The domain component determines whether the register belongs to the input weight matrix, the hidden state weight matrix, the input, or the bias; the gate determine which gate the parameter of the register belongs to; the core determines which core the parameter goes to, and the index determines which vector index the parameter or data value belongs to should that parameter or data value be a part of a vector (i.e. for a weight matrix, the parameter column represents the core while the parameter row represents the index). For a weight matrix, there are two possible domains: W_I and W_H (i.e., input weight matrix and hidden state matrix). Each component of the weight parameter address needs to be populated because it could be in any possible gate, core, and index. For the input values of the LSTM, the index component is the index of the vector it belongs to. However, every input goes into every core, meaning we populate the index component of the address but leave the core component blank. Also, we leave the gate component unpopulated due to the fact that each input vector goes into each gate. In contrast, every bias vector element has a unique core, but does not belong to a single index. We populate the core and gate components but leave the index component unpopulated.

Matrix	Domain	Gate	Core	Index
Input Weight	$W_I(0)$	$F, I, C, or O(0,1,2,3)$	$[0 : corecount-1]$	$[0 : inputdimension-1]$
Hidden State	$W_H(1)$	$F, I, C, or O(0,1,2,3)$	$[0 : corecount-1]$	$[0 : corecount-1]$
Input Vector	$X(2)$	$NA(0)$	$NA(0)$	$[0 : inputdimension-1]$
Bias	$B(3)$	$F, I, C, or O(0,1,2,3)$	$[0 : corecount-1]$	$NA(0)$

TABLE I: Easy addressing scheme for stochastic memory.

IV. Evaluation

In this section, the SC-LSTM design is verified and its performance is evaluated. First, we introduce the environment setup of our experiments. Then, we evaluate the performance of the SC-LSTM system design in terms of power consumption and recognition accuracy as well as runtime. Finally, we compare the SC-LSTM design with the binary LSTM design.

A. Experimental Setup

Platform. Zedboard FPGA (Xilinx Zynq-7000 AP SoC, Dual-core ARM Cortex-A9, 512 MB DDR3, 256 MB Quad-SPI Flash, etc.) and Xilinx Design Suite software, which is shown in Fig. 7.

Dataset. MNIST [14] which are widely applied in image recognition tasks. With MNIST, each 28x28 image can be treated as a sequence of rows which needs 28 time-step to process. The LSTM weights are constrained between the range [-1,1] in order to be possible to represent them as stochastic values. We train the LSTM model using standard 32-bit floating point or FP32 via Keras on Ubuntu 18.04.

Settings. After acquiring the trained LSTM model, We test the SC-LSTM inference engine under different settings of core count and time-step window size to evaluate the designs in terms of power consumption, recognition accuracy,

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

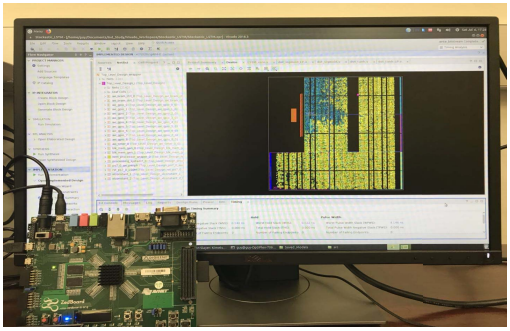

Fig. 7: The real-case implementation of SC-LSTM on Zedboard.

and runtime. In addition, we compare the performance of the SC-LSTM design with the baseline (i.e., binary LSTM implementation on the same Zedboard) which has been implemented with the structure configuration of 28-16. For the APC implementation, the cell state window size, cell state bound and activation function window size as well as stochastic data size are set 256, 8, 256, and 11, respectively. For the MUX implementation, these values are 1024, 8, 64, and 11. The reason why these parameters are different is that MUX-based design suffer more from latency than APC-based design does.Both of them running at 100MHZ clock.

Performance metrics. We choose the following performance metrics to evaluate our implementation. *a) power consumption*: The amount of power consumed by the LSTM module, which is reported by power estimator in Vivado. *b) accuracy*: The ratio of datapoints where the prediction correctly matches the label to the total number of datapoints. *c) runtime*: The amount of time used to make a prediction on a single datapoint. This is measured by measuring the time to execute 1000 datapoints and dividing that time by 1000.

B. Evaluation Results

Fig. 8 shows the results of evaluation on the performance of the SC-LSTM system. Fig. 8.(a)-(c) illustrate the effect of increasing the core count on the performance of the system in terms of power, accuracy and runtime. With the core count increasing from 8 cores to 16 cores, the power consumption increases by 132% for the APC implementation and 85% for the MUX implementation. Obviously, more cores, more power consumption. In comparison, with the core count increasing, the runtime hardly increases at all. This is because all cores work in parallel with each other and the runtime is controlled by the window size. Notably, with the MNIST dataset, the accuracy increases from 66.12% to 90.75% for the APC implementation and 73.68% to 92.71% for the MUX implementation with the cores count increasing. This is due to the fact that, as we trained the LSTM model, increasing the LSTM hidden layer dimension increased the accuracy. Fig. 8.(d)-(f) illustrate the effect of increasing the window size on the performance of the SC-LSTM system. The increase of the window size almost has no effect on the power consumption. This is because changing the window size does not change

Design	Power(mw)	Accuracy	Runtime(ms)
$baseline\ binary$	142	94.00%	0.17
$APC_{based}\ SC$	72	90.75%	18.58
$MUX_{based}\ SC$	38	92.31%	18.58

TABLE II: Performance Comparison between baseline and SC-LSTM design.

Design	LUT	FF	DSP	RAM
$baseline\ binary$	7741	2412	1	3.58KB
$APC_{based}\ SC$	9529	8456	0	0
$MUX_{based}\ SC$	6763	5928	0	0

TABLE III: Resource utilization comparison between baseline and SC-LSTM design.

the architecture, only the amount of time it runs per timestep. We see that increasing the window size also increases the accuracy. By increasing the window size from 2^{12} to 2^{16}, the accuracy increases by 9.43% for the APC implementation and 38.08% for the MUX implementation. This is due to the fact that the entire system is non-deterministic, the more bits we sample, the more consistent the computation results will be. Meanwhile, the runtime increases from 1.392 ms to 18.590 ms. Obviously, as the window size increases, the amount of time it consumes to run a single timestep increases. We can safely conclude that there exists a trade-off between the accuracy and runtime. Considering that our proposed SC-LSTM design is configurable in terms of latency, a proper configuration can be fully investigated and applied to deal with various scenarios.

Table II shows the comparison of the baseline and SC-LSTM designs in terms of power, accuracy and runtime. The baseline architecture is a single LSTM layer with input dimension of 28 , hidden and output dimension of 16. We use a window size of 2^{16} cycles on both the APC and MUX implementation. Obviously, either the APC-based or the MUX-based design consumes much less power than baseline does; APC-based only consumes half of the power the baseline does and the MUX-based even consumes less than 1/3 of the power baseline power does. Meanwhile, the results indicate that, despite the system being non-deterministic, the reduction in accuracy is at most 3.25% and 1.69% for APC-based SC-LSTM and MUX-based SC-LSTM, respectively. In order to achieve best inference accuracy, window-size of both APC-based and MUX-based design is configured as 65535 (i.e., 2^{16}) cycles. Therefore, the runtime of baseline is a few of magnitudes less than that of the APC-/MUX-based design. Table III shows the resource utilization under the three different designs, including the baseline and the two SC-LSTM designs. It appears that the SC design consumes more LUTs and FFs compared to the baseline. However, the fact is that, the whole SC design is merely made up of LUTs and FFs while the baseline design utilizes one more DSP Macro and 3.58KB RAM besides the regular LUTs and FFs. The DSP Macro works for MAC operation and RAM is for activation function look-up table, which greatly contributes to the reduction of LUTs and FFs consumption.

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

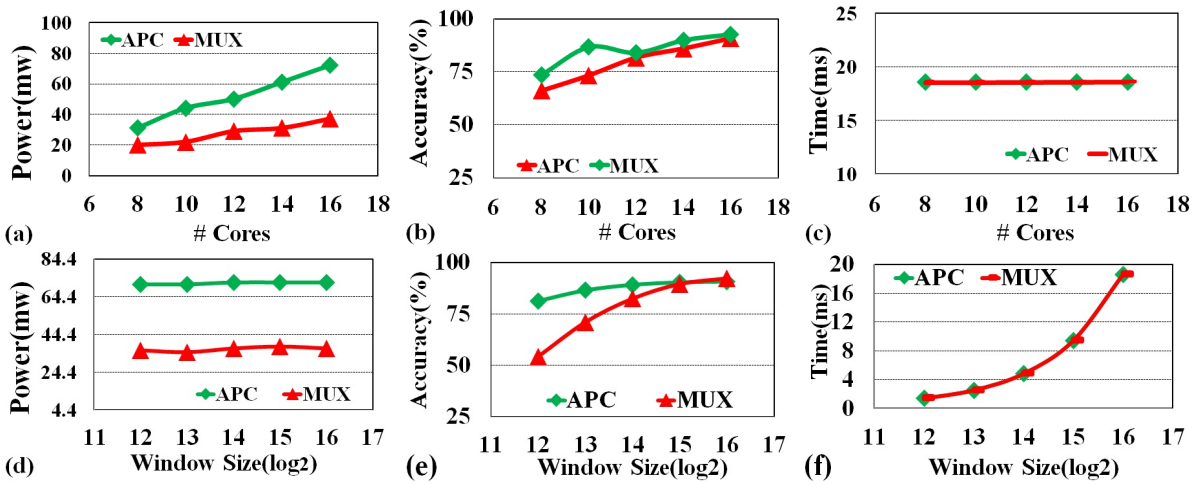

Fig. 8: Performance of the SC-LSTM design in terms of power consumption and accuracy as well as runtime, including APC-based and MUX-based.

V. Related Work

Considering that LSTM can improve recognition accuracy significantly especially for sequential data at the cost of increased computational complexity, many designs have been proposed to improve the hardware efficiency of LSTM-RNN. [4] proposed a balance-aware pruning algorithm to improve the parallel processing efficiency. [20] utilized Fast Fourier Transform (FFT) and inverse FFT to reduce the complexity of matrix multiplication of LSTM. [5] developed a structured compression technique to compress the weight matrices of LSTM. However, it's still challenging to implement a LSTM model on resource-limited mobile or edge devices.

Generally speaking, NN model inherently involve complex architecture and the high computational cost, some highly-parallel and specialized hardware has been designed to accelerate its execution and reduce its hardware cost, enabling its applications in the mobile and edge devices. [21], [22] used GPGPU to accelerate the CNN implementation; [23], [24] explored the optimization on CNN using FPGA [12]. Even so, due to the inherent inefficiency of conventional computing methods or general-purpose computing devices in implementing complex NN, there still exists a large margin to improve the hardware efficiency. On the other hand, SC has been introduced to implement neural networks earlier as a low-cost alternative to conventional binary computing [25]. [13], [26], [27] have done a lot of research on designing elementary stochastic computational elements, such as APC and approximate activation functions. [28] proposed a hardware implementation of a radial basis function neural network by leveraging stochastic logic. [18] and [29] developed a deep belief network using stochastic computational. [30] explored the design space for hardware-efficient stochastic computing using discrete cosine transformation as a case study. [16], [31] tried to explore the trade-off between energy efficiency and accuracy when applying SC in DNN. [12] presented a highly efficient SC-based inference framework of the large-scale DCNNs that achieves high energy efficiency and low area/hardware cost. Further more, [32] has proposed an end-to-end stochastic system to further reduce power for the whole structure.

VI. Acknowledgment

We thank all the anonymous reviewers for invaluable and insightful comments to make this paper better. This work is supported in part by NSF grant CCF-1822985. The corresponding author is Yang Hu.

VII. Conclusion

In this paper, we propose an general scalable SC-LSTM design, which effectively integrates the stochastic computing with complex LSTM model. We successfully implement the design on the ZedBoard platform, and evaluate the performance of design in terms of power efficiency, recognition and runtime using MNIST dataset. Both APC and MUX based SC neuron design is implemented and experimented. Meanwhile, the baseline binary LSTM is constructed on the same FPGA platform. With the hidden-size being 16 and window-size 2^{16}, in the best case, the power consumption of SC-LSTM is 73.24% less than the baseline LSTM implementation; on the other hand, the accuracy of SC-LSTM is also competitive to the baseline LSTM implementation.

References

[1] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019, 2015.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[3] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[4] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 75–84. ACM, 2017.

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.

[5] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 11–20. ACM, 2018.

[6] Zhe Li, Caiwen Ding, Siyue Wang, Wujie Wen, Youwei Zhuo, Chang Liu, Qinru Qiu, Wenyao Xu, Xue Lin, Xuehai Qian, et al. E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 69–80. IEEE, 2019.

[7] Kyle Daruwalla, Heng Zhuo, Carly Schulz, and Mikko Lipasti. Bitbench: A benchmark for bitstream computing. In Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, pages 177–187, New York, NY, USA, 2019. ACM.

[8] Brian R Gaines. Stochastic computing systems. In Advances in information systems science, pages 37–172. Springer, 1969.

[9] Di Wu and Joshua San Miguel. In-stream stochastic division and square root via correlation. In Proceedings of the 56th Annual Design Automation Conference 2019, page 162. ACM, 2019.

[10] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Yanzhi Wang, and Bo Yuan. Dscnn: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks. In 2016 IEEE 34th International Conference on Computer Design (ICCD), pages 678–681. IEEE, 2016.

[11] Hyeonuk Sim, Saken Kenzhegulov, and Jongeun Lee. Dps: Dynamic precision scaling for stochastic computing-based deep neural networks. In Proceedings of the 55th Annual Design Automation Conference, page 13. ACM, 2018.

[12] Zhe Li, Ji Li, Ao Ren, Ruizhe Cai, Caiwen Ding, Xuehai Qian, Jeffrey Draper, Bo Yuan, Jian Tang, Qinru Qiu, et al. Heif: Highly efficient stochastic computing based inference framework for deep neural networks. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018.

[13] Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. Approximate derandomizer for stochastic circuits. In 2015 International SoC Design Conference (ISOCC), pages 123–124. IEEE, 2015.

[14] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits.

[15] Understanding lstm networks, 2015.

[16] Bradley D Brown and Howard C Card. Stochastic neural computation. i. computational elements. IEEE Transactions on computers, 50(9):891–905, 2001.

[17] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. ACM SIGOPS Operating Systems Review, 51(2):405–418, 2017.

[18] Yidong Liu, Yanzhi Wang, Fabrizio Lombardi, and Jie Han. An energy-efficient stochastic computational deep belief network. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1175–1178. IEEE, 2018.

[19] Pong P Chu and Robert E Jones. Design techniques of fpga based random number generator. In Military and Aerospace Applications of Programmable Devices and Technologies Conference, volume 1, pages 28–30. Citeseer, 1999.

[20] Zhe Li, Shuo Wang, Caiwen Ding, Qinru Qiu, Yanzhi Wang, and Yun Liang. Efficient recurrent neural networks using structured matrices in fpgas. arXiv preprint arXiv:1803.07661, 2018.

[21] Endre László, Péter Szolgay, and Zoltán Nagy. Analysis of a gpu based cnn implementation. In 2012 13th International Workshop on Cellular Nanoscale Networks and their Applications, pages 1–5. IEEE, 2012.

[22] GEORGE VALENTIN STOICA, RADU DOGARU, and C Stoica. High performance cuda based cnn image processor, 2015.

[23] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 161–170. ACM, 2015.

[24] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), pages 575–580. IEEE, 2016.

[25] Armin Alaghi and John P Hayes. Survey of stochastic computing. ACM Transactions on Embedded computing systems (TECS), 12(2s):92, 2013.

[26] Behraoz Parhami and Chi-Hsiang Yeh. Accumulative parallel counters. In Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, volume 2, pages 966–970. IEEE, 1995.

[27] Bingzhe Li, Yaobin Qin, Bo Yuan, and David J Lilja. Neural network classifiers using stochastic computing with a hardware-oriented approximate activation function. In 2017 IEEE International Conference on Computer Design (ICCD), pages 97–104. IEEE, 2017.

[28] Yuan Ji, Feng Ran, Cong Ma, and David J Lilja. A hardware implementation of a radial basis function neural network using stochastic logic. In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pages 880–883. EDA Consortium, 2015.

[29] Kayode Sanni, Guillaume Garreau, Jamal Lottier Molin, and Andreas G Andreou. Fpga implementation of a deep belief network architecture for character recognition using stochastic computation. In 2015 49th Annual Conference on Information Sciences and Systems (CISS), pages 1–5. IEEE, 2015.

[30] Bo Yuan, Chuan Zhang, and Zhongfeng Wang. Design space exploration for hardware-efficient stochastic computing: A case study on discrete cosine transformation. In 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 6555–6559. IEEE, 2016.

[31] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), pages 1–6. IEEE, 2016.

[32] Mikko Lipasti and Carly Schulz. End-to-end stochastic computing, 2017.

Authorized licensed use limited to: University of Florida. Downloaded on July 28,2020 at 20:49:36 UTC from IEEE Xplore. Restrictions apply.