AN FPGA IMPLEMENTATION OF STOCHASTIC COMPUTING-BASED LSTM

by

Guy A. Maor

APPROVED BY SUPERVISORY COMMITTEE:

Yang Hu, Chair

Carl Sechen

Mehrdad Nourani

This thesis is dedicated to
my loving parents who have
supported me throughout my
entire life.

AN FPGA IMPLEMENTATION OF STOCHASTIC COMPUTING-BASED LSTM

by

GUY A. MAOR, BS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTERS OF SCIENCE IN

COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

August 2019

ACKNOWLEDGMENTS

To start, I would like to thank my thesis advisor, Dr. Yang Hu, for assisting me throughout my entire thesis. He has encouraged me to achieve goals I never would have imagined I could have accomplished.

I would also like to thank Dr. Carl Sechen and Dr. Mehrdad Nourani for taking the time out of their busy schedule to be my committee members.

I would like to thank Xiaoming Zeng and Zhendong Wang for guiding me throughout the research process.

July 2019

AN FPGA IMPLEMENTATION OF STOCHASTIC COMPUTING-BASED LSTM

Guy A. Maor, MS
The University of Texas at Dallas, 2019

Supervising Professor: Yang Hu, Chair

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 thesis introduces SC into LSTM and creatively proposes an SC-based LSTM architecture design to save the hardware cost and power consumption. More importantly, the thesis successfully implements the design on an 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.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Recurrent neural networks (RNN) are widely used in applications such as image prediction, speech recognition (Lipton et al., 2015). As a representative structure of RNN, Long Short Term Memory (LSTM) neural networks can tackle the issue of vanishing gradient descent, which is common during the training of conventional RNNs, and further improve recognition accuracy (Hochreiter and Schmidhuber, 1997; Gers et al., 1999). Therefore, both industrial and academia show increasing interest in the development and deployment of LSTM. In convention, an LSTM model is constructed with software methods, and is implemented on high-performance servers. However, considering that LSTM typically involves high computational complexity, the execution of software-based LSTM on the servers typically cause high power consumption and large hardware cost, though multiple designs have been proposed to improve its hardware efficiency (Han et al., 2017; Li et al., 2018; Wang et al., 2018). Due to this fact, it has become increasingly challenging to apply LSTM in mobile and edge devices, which typically have limited resources, under the background of Internet of Things (IoT) and 5G deployment.

Recently, Stochastic Computing (SC) has been quickly developed to represent continuous values by streams of random bits and implement complex computations by using simple bit-wise operations on the streams. Therefore, SC can effectively reduce the hardware cost of some fundamental arithmetic circuits (Gaines, 1969; Brown and Card, 2001), such as multiplication, adders, and even nonlinear functions, and is considered a great opportunity to simplify the implementation of LSTM on hardware and save its hardware cost. As a matter of fact, SC designs have been introduced into neural networks (NN) before, such as convolutional neural networks (CNN)(Li et al., 2016; Sim et al., 2018; Li et al., 2018), the multi-layers perceptrons(Rosselló et al., 2012), etc. However, the application of SC in RNN, especially in LSTM, has never been effectively explored. In such applications, SC is

expected to significantly improve the hardware efficiency of LSTM, and further promote its deployment in mobile and edge computation devices.

To enable the SC-based NN in practice with low power and hardware cost, customized hardware platform has been developed, including Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC).

Among the numerous platforms, FPGAs can provide customized hardware performance as well as flexible reconfigurability.

Therefore, this thesis designs an SC-based LSTM architecture, SC-LSTM, by successfully applying SC into the complex LSTM model. To the best of our knowledge, this is the first work to introduce SC into the RNN and leverage the high hardware efficiency of SC circuits to effectively reduce the resource and power cost of RNN 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.

To verify the low-power feature of SC-LSTM design, we first implement a conventional binary LSTM module on an FPGA platform as a baseline. Then, we successfully construct the SC-LSTM design on the same platform. We adopt MNIST dataset to make a comparison between these two designs, and observe that, the SC-LSTM reduces power consumption by 73.24% compared to the baseline, though the recognition accuracy decreases by 1.69%. The results show that the SC-LSTM can significantly reduce power consumption without much loss on accuracy.

In the future work, in order to further reduce the system power consumption, low-power ASIC techniques based on FPGA design can be effectively adopted.

Overall, this thesis makes the following contributions:

- We introduce a hardware-efficient SC circuit implementation of an LSTM architecture. To the best of our knowledge, this is the first time such an architecture has been implemented.

- We successfully validate the SC-LSTM design on an FPGA board and implement 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.

- Currently, we mainly focus on the SC-LSTM design as well as its optimization. By leveraging the design, we expect to further optimize the stochastic memory structure, another module consuming much power, in the future such that the power cost of the entire system can be significantly reduced.

The remainder of this thesis 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 thesis.

CHAPTER 2

BACKGROUND

2.1 Supervised Machine Learning

Of all the topics most heavily researched in Artificial Intelligence, nothing comes close to Machine Learning, specifically Supervised Machine Learning (Murphy, 2012). Rather than hard code an intelligent system to make human-like decisions, systems that use Supervised Machine Learning are trained using thousands of training data points. Each data point describes two detail: A possible scenario the system is expected to observe (i.e., input data), and the expected prediction the system is suppose to make (i.e., label). Input data and labels are commonly in the form of vectors; however, it's very common to see data in the form of sequences, tensors, or two or more different forms (i.e., a sentence can be treated as a sequence of vectors). Once these thousands of data points have been collected, the system trains a model to take in input data and guess their labels. A system that can guess a significantly large majority of labels correctly, we say that system is very accurate. For example, given a dataset that contains a picture of either a shoe or a shirt, the label to each data point in the dataset can be a 0 for shoe and 1 for shirt. Once the model is trained, the model can take in a photo of a shoe or shirt it's never seen before and correctly guess if it's a shoe or a shirt.

Supervised Machine Learning has two phases, the training phase, and the inference phase. The training takes in a training dataset and trains the model so that the predictions of all the data points match their labels for as many data points as possible. Once the training phase is complete, the inference phase, also known as the testing phase, tests the performance of the model. In order to test performance, a completely new dataset of the same type as the training dataset must be used. This is to ensure the model can perform on data it's never seen before. Performance can be measured in many ways, but is commonly measured as

accuracy, which is the ratio of correctly predicted labels of the testing dataset to the total number of testing data points.

2.2 Neural Networks

One of the most popular Supervised Machine Learning models is the Neural Network (Feldman, Feldman). What makes Neural Networks so special is that they are very versatile and can be used for a large variety of tasks, from Natural Language Processing, to Image Recognition. Each Neural Network is described by Linear Algebra equations. Most Neural Networks contain smaller equations in the form $a(X * W + B)$ where X is an input vector, W is a weight matrix, B is a bias vector, and $a()$ is a non-linear activation function such as hyperbolic tangent.

In order to train these models, we use a concept known as Gradient Descent. Gradient Descent begins by defining a loss function. The loss function is any function that describes how far the predictions are from their labels. For example, given input data, X, the label to the data, Y, and the function that describes the model we want to train, $f(X)$, we can define the loss function, $E(X, Y)$, as described in equation 2.1.

$$E(X, Y) = (Y - f(X))^2 \tag{2.1}$$

With gradient descent, we first treat the input data as constants. When then find the gradient of the loss function with respect to each parameter value in the Neural Network (i.e., every weight matrix and bias vector that is not the input data). Once we find the gradient of each parameter, we subtract the gradients from the parameters scaled by a learning rate, l, as described in equations 2.2 and 2.3.

$$W <= W - l * W' \tag{2.2}$$

$$B <= B - l * B' \tag{2.3}$$

We repeat this process until we are satisfied with the accuracy of a validation dataset. The validation dataset can be the training dataset or a completely new dataset.

2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNN) is a subclass of Neural Networks that are most commonly used for processing sequential data (Nguyen, 2018). What's unique about RNNs, compared to traditional Feed-Forward Neural Networks, is that at each timestep, the output to the previous calculation also becomes part of the input to the next calculation. For example, at timestep, t, given input data, X_t, the output from the previous timestep, H_{t-1}, and the model, $f()$, we make the prediction for the current timestep, H_t, as described in equation 2.4.

$$h_t = f([x_t, h_{t-1}]) \tag{2.4}$$

In order to train an RNN, we must unravel the RNN. We unravel the RNN by expressing each timestep as its own Neural Network layer where the output of one layer is the input to the layer that corresponds to the next time step. The layers, despite being separate, share their weights. Fig. 2.1 illustrates how we unravel an RNN. Originally, we expressed the equation in terms of x_t and h_{t-1}. Once we unravel the RNN, we express the model in terms of x_1, x_2, x_3, h_1, and h_2. When defining the loss function, we express it in terms of x_1, x_2, x_3, h_1, h_2, and h_3.

One big issue associated with RNNs is the issue of vanishing gradient (Yoshua Bengio and Frasconi, 1994). As we discussed earlier, we use Gradient Descent to train Neural Networks. The issue is that as the sequence increases in length, gradients early in the sequence become smaller. This makes it very difficult to train because small gradients have hardly any effect on the results. Later, we will discuss how LSTM reduces the severity of this issue.

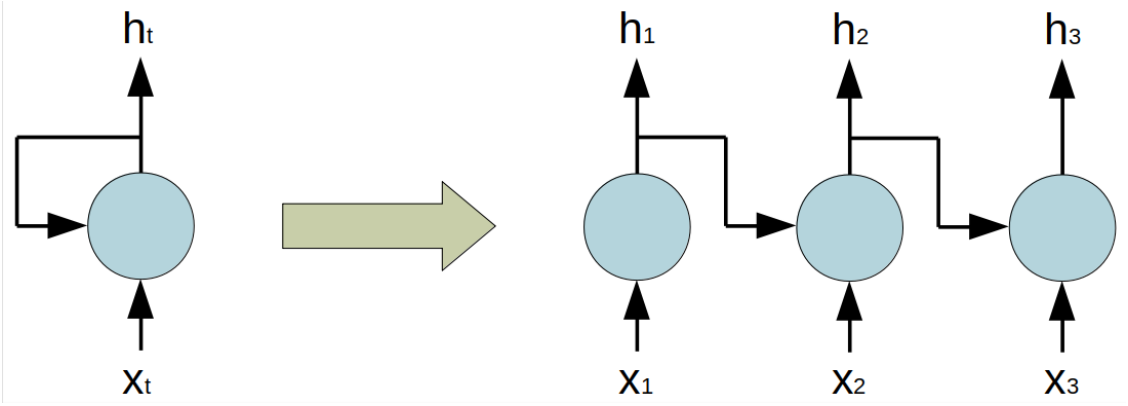

Figure 2.1. Unraveling An RNN With A Sequence Length of 3

2.4 LSTM

LSTM is one of the most commonly used RNN model. Unlike primitive RNNs, LSTM utilizes gating mechanisms to mitigate the issue of Vanishing Gradient that occurs when training primitive RNNs. Therefore, LSTM can effectively avoid catastrophic errors and typically achieve higher performance in terms of accuracy compared to primitive RNNs. The following equations represent the functions used to calculate the next hidden state, h_t (Olah, 2015):

$$F = sigmoid(W_f \cdot [h_{t-1}, x_t] + B_f) \tag{2.5}$$

$$I = sigmoid(W_i \cdot [h_{t-1}, x_t] + B_i) \tag{2.6}$$

$$C^{'} = tanh(W_c \cdot [h_{t-1}, x_t] + B_c) \tag{2.7}$$

$$O = sigmoid(W_o \cdot [h_{t-1}, x_t] + B_o) \tag{2.8}$$

$$C_t = C_{t-1} * F + I * C^{'} \tag{2.9}$$

$$h_t = tanh(C_t) \cdot O \tag{2.10}$$

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. The cell state, C_t, is a vector that acts as

the internal memory of the LSTM layer. Gates operate on the cell state to determine what information should be remembered or forgotten. Typically, there are four types of gates, each of which is represented by a neuron operation, including the Forget gate (F), Input gate (I), Cell gate (C′), and Output gate (O).

Fig. 2.2 shows the internal structure of an LSTM layer. The Forget gate is responsible for determining what elements in the cell state are remembered or forgotten. It does so by multiplying the cell state by a masking vector. The masking vector multiplies elements by 0 in order to forget them, while multiplying elements by 1 to maintain them in the cell state's memory. We use a sigmoid activation function with the Forget gate because the sigmoid function works really well to generate masking vectors. The Cell gate generates a vector that determines whether to increase or decrease the values of the cell state. We use a hyperbolic tangent function (tanh) to limit the values of the Cell gate between the range of -1 and 1. Similar to the Forget gate, the Input gate is responsible for masking the results of the Cell gate by using a sigmoid function to generate a masking vector and multiplying it with the Cell gate output. After adding the result of the Cell gate to the cell state, the new current cell state is complete. In order to generate the output, H_t, the cell state is passed through a tanh function to restrict its elements between -1 and 1. Finally, the result is masked once more by the output gate which determines what values are visible at the output.

2.5 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, (Brown and Card, 2001). The probability, p, is associated with a stochastic value, v, which is represented

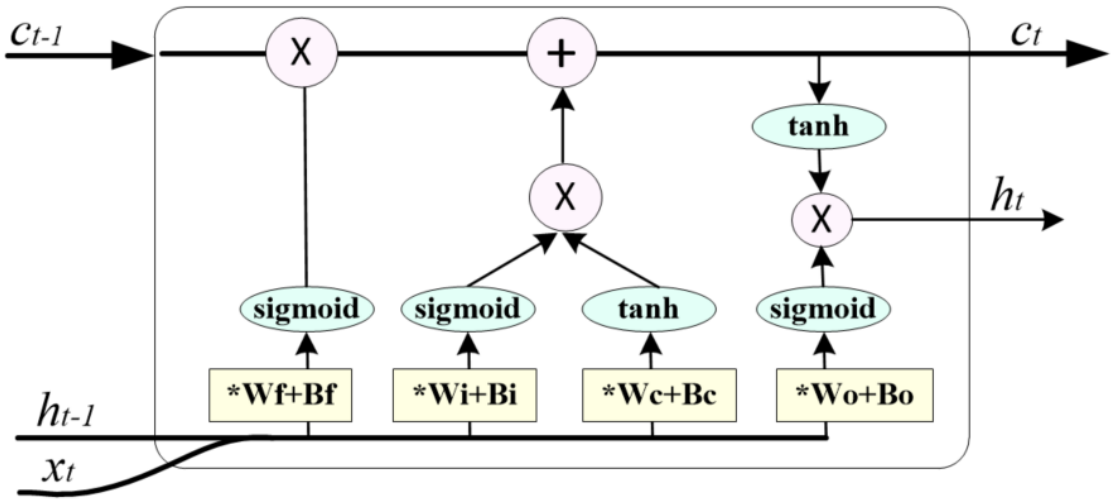

Figure 2.2. The Internal Structure of An LSTM Layer

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 (Brown and Card, 2001). 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:

$$p = (v + 1)/2 \tag{2.11}$$

$$v = 2 * p - 1 \tag{2.12}$$

2.5.1 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 (Brown and Card, 2001). 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.

$$p_{out} = P(input_1 == input_2) \tag{2.13}$$

$$\begin{aligned} p_{out} = {} & P(input_1 == 1 \; and \; input_2 == 1) \\ & + P(input_1 == 0 \; and \; input_2 == 0) \end{aligned} \tag{2.14}$$

$$p_{out} = p_1 * p_2 + (1 - p_1) * (1 - p_2) \tag{2.15}$$

Based on equation 2.11, we can substitute the probability values, p, in equation 2.15 above with the stochastic values, v, as follows:

$$\begin{aligned} (v_{out} + 1)/2 = {} & (v_1 + 1)/2 * (v_2 + 1)/2 \\ & + (1 - (v_1 + 1)/2) * (1 - (v_2 + 1)/2) \end{aligned} \tag{2.16}$$

$$v_{out} = v_1 * v_2 \tag{2.17}$$

2.5.2 Addition

In stochastic computing, there are multiple approaches to implement addition operation (Ren et al., 2017). 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. There are two implementations we explored when implementing addition, APC-Based and MUX-Based. Both provide unique advantages. Despite this, generating a stochastic value equal to the sum of stochastic values is not always possible. Instead, we either represent values in binary, or represent values as the average of the addition operation. Farther down the circuit, we address how to convert the values back into their appropriate stochastic values.

APC-Based Approach

The first approach to implementing addition is by the use of an Approximate Parallel Counter (APC) (Kim et al., 2015). An APC is a combinatorial circuit that takes an array of N bits as the input and counts the number of bits in the array that are set to high. The output to the APC is a binary number that is equal to the number of 1-bits at the input. When taking in an array of stochastic inputs, because the entire system is non-deterministic, the output is also deterministic. However, if we were to calculate the expected value of the output of the APC, we would find that it is equal to the sum of the expected values of the input, which are effectively the probabilities of the input. In other words, the expected value of the output is equal to the sum of the probabilities. This is important because, although the output is a binary value, the binary value is a linear function of the sum of the stochastic values of the inputs. To obtain a more consistent output, we would average the APC results over multiple samples. In practice, we accumulate the samples rather than averaging them since the result is still proportional to the sum of the probabilities.

MUX-Based Approach

The second approach is to implement addition using a MUX (Brown and Card, 2001). Similar to the APC, the MUX takes in an array of N bits. Unlike the APC, the output is a stochastic value rather than a binary value. Connected to the select port is a random sequence that randomly selects an input to be routed as the output. The stochastic value of the output is proportional to the sum of the stochastic values of the inputs in the array. Similar to the APC, when we use an accumulator on the output, we can get a more consistent output. The benefit to the MUX-based approach is that it consumes significantly less power than the APC-Based approach as we will demonstrate in our test results. However, the MUX-based approach is far less accurate as an APC can accumulate N bits at a time while the MUX accumulates 1 bit at a time.

2.6 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. (Han et al., 2017) proposed a balance-aware pruning algorithm to improve the parallel processing efficiency. (Li et al., 2018) utilized Fast Fourier Transform (FFT) and inverse FFT to reduce the complexity of matrix multiplication of LSTM. (Wang et al., 2018) 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. (László et al., 2012; George Valentin Stoica, 2015) used GPGPU to

accelerate the CNN implementation; (Zhang et al., 2015; Motamedi et al., 2016) explored the optimization on CNN using FPGA (Li et al., 2018). 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 (Alaghi and Hayes, 2013). (Kim et al., 2015; Parhami and Yeh, 1995; Li et al., 2017) have done a lot of research on designing elementary stochastic computational elements, such as APC and approximate activation functions. (Ji et al., 2015) proposed a hardware implementation of a radial basis function neural network by leveraging stochastic logic. (Liu et al., 2018) and (Sanni et al., 2015) developed a deep belief network using stochastic computational. (Yuan et al., 2016) explored the design space for hardware-efficient stochastic computing using discrete cosine transformation as a case study. (Kim et al., 2016; Brown and Card, 2001) tried to explore the trade-off between energy efficiency and accuracy when applying SC in DNN. (Li et al., 2018) 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, (Lipasti and Schulz, 2017) has proposed an end-to-end stochastic system to further reduce power for the whole structure.

CHAPTER 3

DESIGN

3.1 Top Level Design

In this section, we discuss the top-level design of the whole system architecture. Fig. 3.1 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 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.

3.2 SC-LSTM module

3.2.1 Multi-Cores Design

This section discusses how the SC-LSTM module is constructed through a multi-cores 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

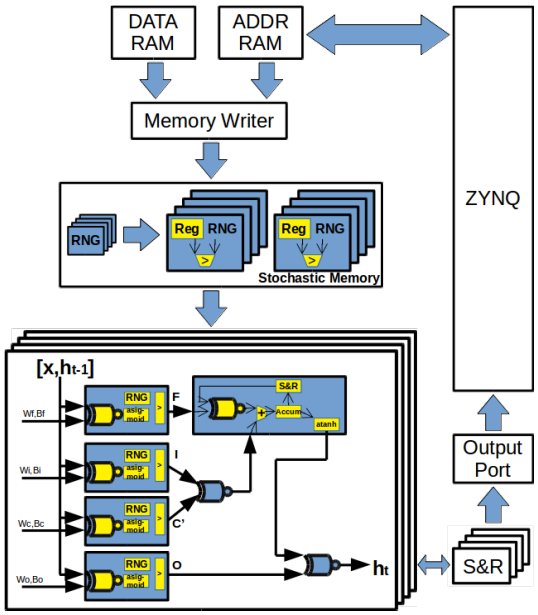

Figure 3.1. The Top Level Design of The SC-LSTM Architecture

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-cores design. That's to say, each output element of the SC-LSTM is assigned to a single core to be processed. Fig. 3.2 (a) illustrates the multi-cores design with an example of the output dimension being m, which indicates that the m cores are required to process the computation.

Next, we further explore the internal structure of each core in the multi-cores 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. 3.2 (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

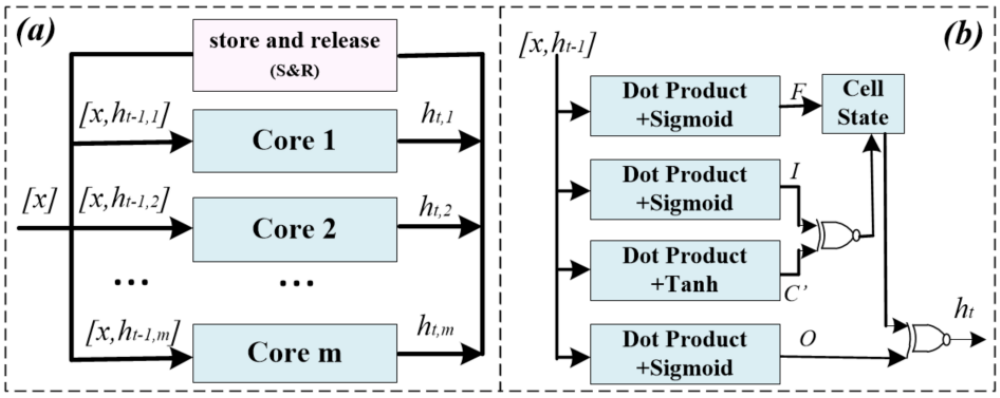

Figure 3.2. (a) The Multi-Cores Design With An M-Dimension Output; (b) The Internal Structure of Each Core.

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.

3.2.2 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 the addition operation of a Neuron. Fig. 3.3 shows the internal structure of the Neuron (Liu et al., 2018). 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 and MUX addition is non-deterministic, the output result will not always be consistent. We would rather average the result of the APC or MUX outputs over 2^M (number of data to be accumulated) samples. The more clock cycles, the more consistent the result will be. We use an accumulator to sample the results for 2^M samples. The accumulator contains two registers, a storing register, and a holding register. The storing register reads in 2^M samples at the input of the accumulator. After the 2^M samples, the

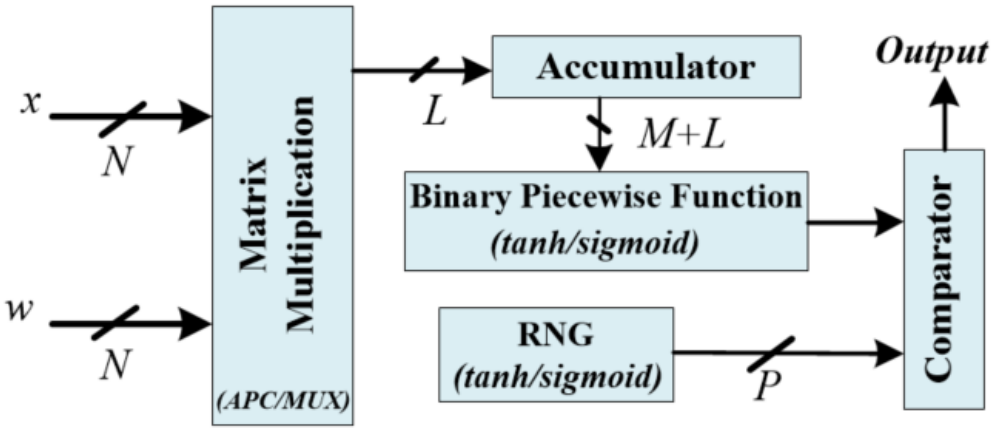

Figure 3.3. The Internal Structure of A Neuron

storing register writes its value to the holding register before the storing register resets itself for another 2^M samples. The holding register value is the output to the accumulator.

As we mentioned earlier when discussing the MUX implementation, in practice, we have to use a random sequence at the select port; however, because we use an accumulator to make the results more consistent, we use a simple counter that counts from 0 to N-1 to produce the sequence as the results provided acceptable accuracy. Also, because MUX accumulates accumulates 1 bit at a time rather than all the input bits in a single clock cycle, the neuron window size will be N times larger in order for the accumulator to produce results similar to that of the APC.

After the Accumulator step, the next step is to perform the activation function. Theoretically, there are two types of activation functions: *Sigmoid* and *Tanh*. 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 3.1 and 3.2. Equation 3.3 - 3.4 and 3.5 describe the comparator operations with functions *Sigmoid* and *Tanh*, respectively. The variable *n_window_size* defines the number of samples the Accumulator reads before storing the value. The variable *Accumulator* simply represents the output the Accumulator.

$$RNG_{sigmoid} : [-n\_window\_size * 4,\ n\_window\_size * 4] \tag{3.1}$$

$$RNG_{tanh} : [-n\_window\_size,\ n\_window\_size] \tag{3.2}$$

$$Sigmoid_{temp} = Accumulator * 2 - N * n\_window\_size + n\_window\_size * 2 \tag{3.3}$$

$$Sigmoid(x) = [max(Sigmoid_{temp}, 0) > RNG_{sigmoid}] \ ?\ 1\ :\ 0 \tag{3.4}$$

$$Tanh(x) = [(Accumulator * 2 - N * n\_window\_size) > RNG_{tanh}] \ ?\ 1\ :\ 0 \tag{3.5}$$

3.2.3 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. 3.4 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

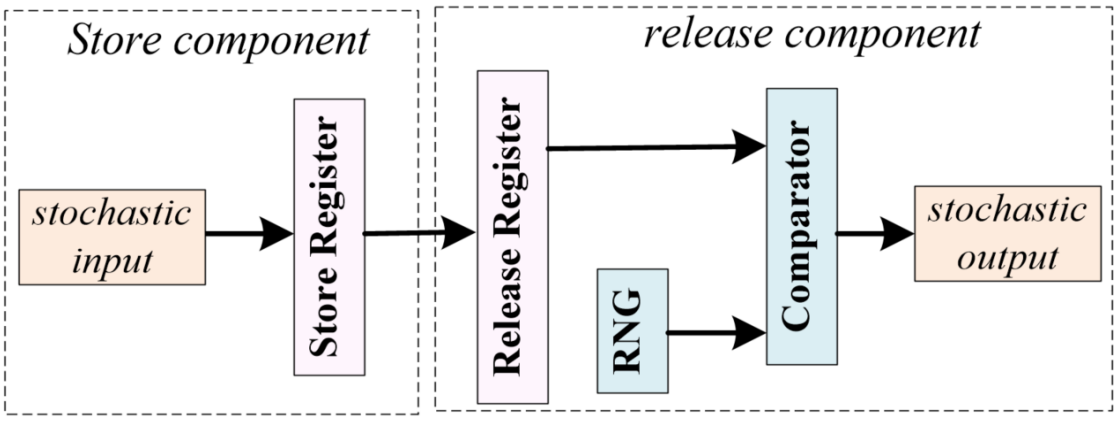

Figure 3.4. The Structure of The Store And Release Component

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.

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) (Chu and Jones, 1999). 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.

3.2.4 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. 3.5 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 2.9. 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 approximation of the tanh function on the cell state.

3.3 Stochastic Memory

In order to implement a 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

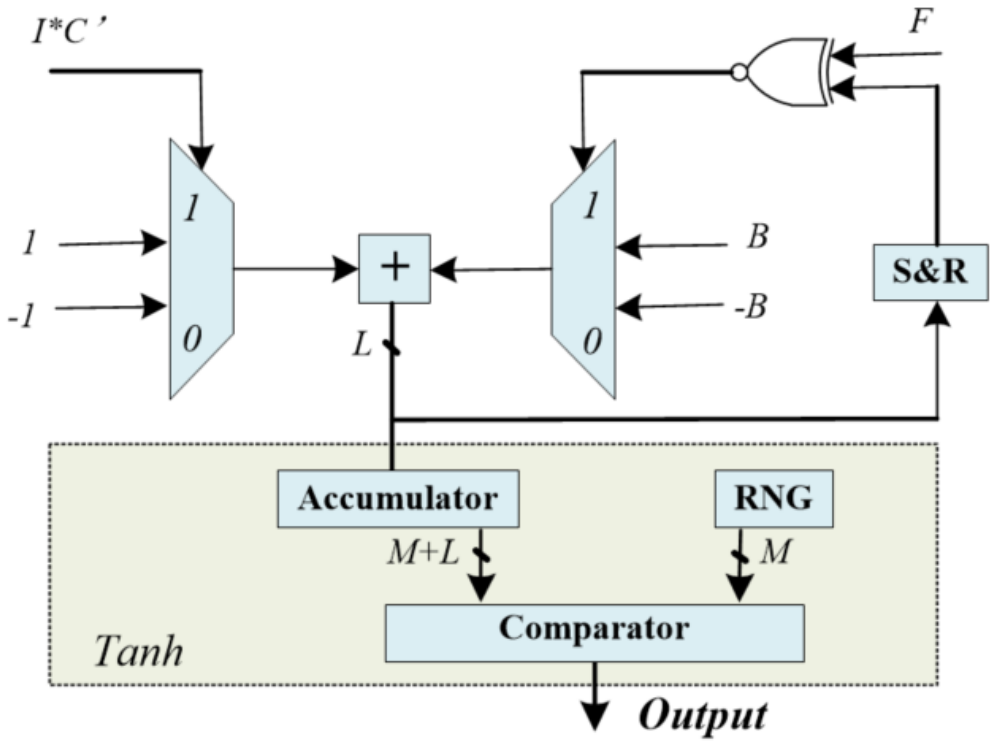

Figure 3.5. The Structure of The Cell State

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 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 **??**. 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

Table 3.1. Easy Addressing Scheme For Stochastic Memory.

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

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.

CHAPTER 4

EVALUATION

In this chapter, 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.

4.1 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. 4.1.

Dataset. MNIST (Yann LeCun, Yann LeCun) 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 for it to be possible to represent them as stochastic values. We train the network using Keras on Ubuntu 18.04.

Settings. 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. We test the SC-LSTM implementation under different settings of core count and time-step window size to evaluate the designs in terms of power consumption, recognition accuracy, 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 28-16-10. Both of them running at 100MHZ clock.

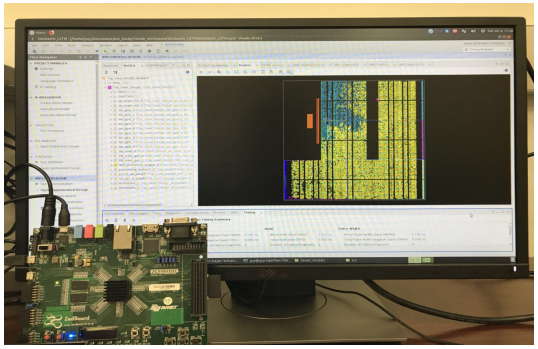

Figure 4.1. The Real-Case Implementation of SC-LSTM On The Zedboard

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 data points where the prediction correctly matches the label to the total number of data points. *c) runtime*: The amount of time used to make a prediction on a single data point. This is measured by measuring the time to execute 1000 data points and dividing that time by 1000. We also measured the number of LUTs and Flip-Flops utilized by each design.

4.2 Test results

Fig. 4.2 shows the results of evaluation on the performance of the SC-LSTM system. Fig. 4.2.(a)-(e) 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

Table 4.1. Comparison Between Baseline And SC-LSTM Design

Design	**Power (mW)**	**Accuracy**	**Runtime (ms)**	**LUTs**	**FFs**	**RAMB18**	**DSP48**
baseline binary	142	94.00%	0.164	2676	1847	2	16
APC_{based} *SC*	72	90.75%	18.58	9529	8456	0	0
MUX_{based} *SC*	38	92.31%	18.58	6763	5928	0	0

LSTM hidden layer dimension increased the accuracy. As we increase the cores from 8 to 16, the number of LUTs increase from 4582 to 9529 for the APC implementation, and 3517 to 6763 for the MUX implementation. Similarly, the number of Flip-Flops increase from 4072 to 8456 for the APC and 3000 to 5928 for the MUX implementation. This increase in resources is due to the fact that as we increase the number of cores, we increase the hardware, which increases resources.

Fig. 4.2.(f)-(j) 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 only the amount of time it runs per timestep, which requires very little extra hardware. 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 takes to run a single timestep increases. LUTs and Flip-Flops remain about the same with a difference of about 200 LUTs and 250 Flip-Flops. As mentioned earlier, changing the window size has very little effect on the architecture.

Table 4.1 shows the comparison of performance between the baseline and SC-LSTM design. The baseline architecture is a single LSTM layer with input dimension of 28 and

output dimension of 16 followed by a fully connected layer with output dimension of 10 and a softmax activation function. We used 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. The computation of both APC-based and MUX-based design last 65535 (i.e., 2^{16}) cycles while baseline only costs one cycle for one computation; therefore, the runtime of baseline is a few of magnitudes less than that of the APC-/MUX-based design. In terms of resources, the number of LUTs and Flip-Flops are less in the baseline; however, the baseline uses 2 RAMB18s and 16 DSP48s while neither the APC nor the MUX implementation uses a single RAMB18 or DSP48.

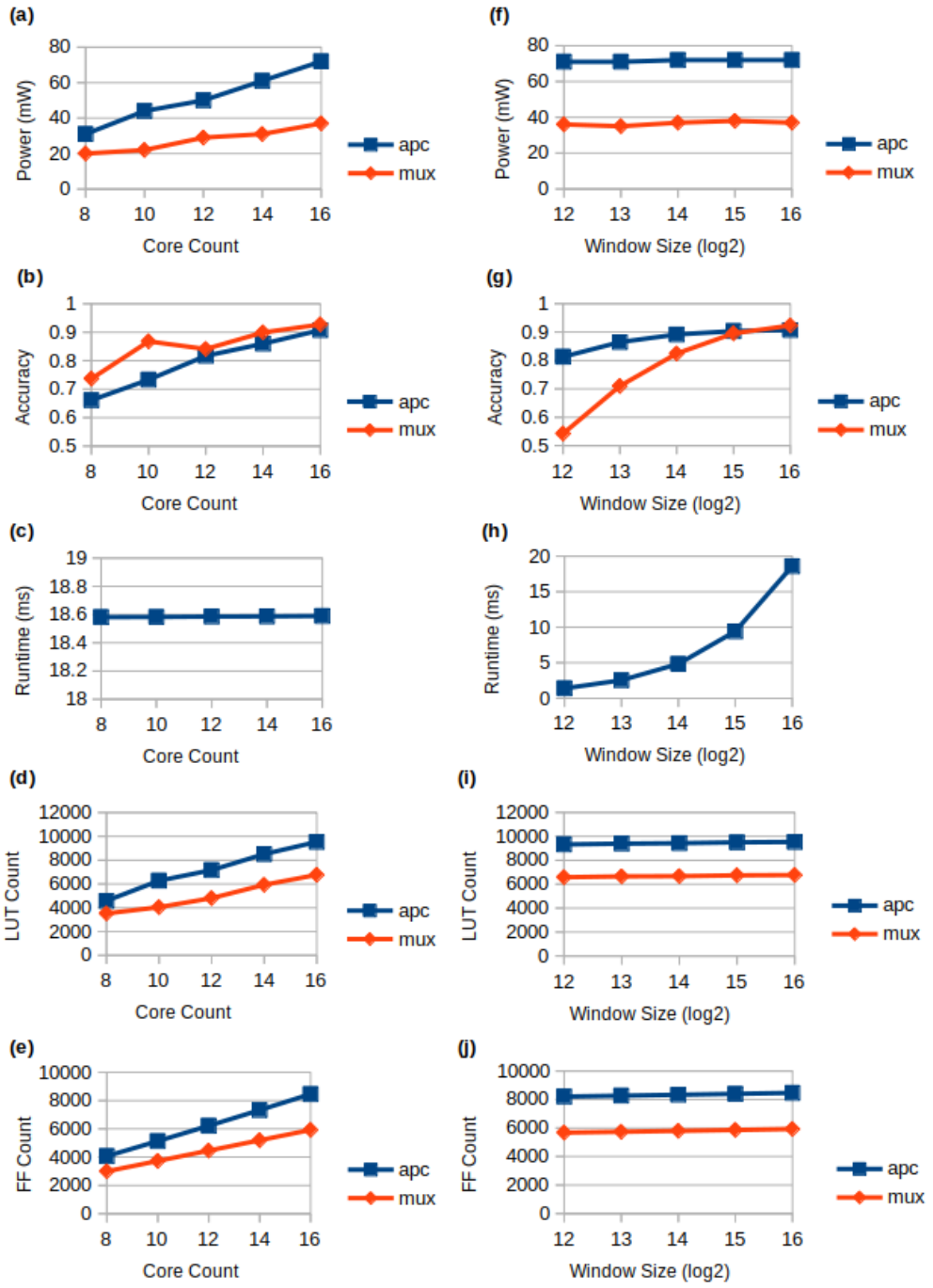

Figure 4.2. Performance of The SC-LSTM Design In Terms of LUTs, Flip-Flops, Power Consumption, Accuracy, And Runtime, Including APC-Based And MUX-Based

CHAPTER 5

CONCLUSION

In this thesis, we propose an SC-LSTM design, which effectively integrates the stochastic computing with complex LSTM model for the first time. 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 0.036W, which is nearly 1/4 of the baseline LSTM implementation; on the other hand, the accuracy of SC-LSTM is also competitive to the baseline LSTM implementation.

REFERENCES

Alaghi, A. and J. P. Hayes (2013). Survey of stochastic computing. *ACM Transactions on Embedded computing systems (TECS) 12*(2s), 92.

Brown, B. D. and H. C. Card (2001). Stochastic neural computation. i. computational elements. *IEEE Transactions on computers 50*(9), 891–905.

Chu, P. P. and R. E. Jones (1999). Design techniques of fpga based random number generator. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference*, Volume 1, pp. 28–30. Citeseer.

Feldman, J. *Neural Networks: A Systematic Introduction*.

Gaines, B. R. (1969). Stochastic computing systems. In *Advances in information systems science*, pp. 37–172. Springer.

George Valentin Stoica, Radu Dogaru, E. C. S. (2015). High performance cuda based cnn image processor.

Gers, F. A., J. Schmidhuber, and F. Cummins (1999). Learning to forget: Continual prediction with lstm.

Han, S., J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al. (2017). Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84. ACM.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural computation 9*(8), 1735–1780.

Ji, Y., F. Ran, C. Ma, and D. J. Lilja (2015). 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*, pp. 880–883. EDA Consortium.

Kim, K., J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi (2016). Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE.

Kim, K., J. Lee, and K. Choi (2015). Approximate de-randomizer for stochastic circuits. In *2015 International SoC Design Conference (ISOCC)*, pp. 123–124. IEEE.

László, E., P. Szolgay, and Z. Nagy (2012). Analysis of a gpu based cnn implementation. In *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*, pp. 1–5. IEEE.

Li, B., Y. Qin, B. Yuan, and D. J. Lilja (2017). Neural network classifiers using stochastic computing with a hardware-oriented approximate activation function. In *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 97–104. IEEE.

Li, Z., J. Li, A. Ren, R. Cai, C. Ding, X. Qian, J. Draper, B. Yuan, J. Tang, Q. Qiu, et al. (2018). Heif: Highly efficient stochastic computing based inference framework for deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Li, Z., A. Ren, J. Li, Q. Qiu, Y. Wang, and B. Yuan (2016). Dscnn: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 678–681. IEEE.

Li, Z., S. Wang, C. Ding, Q. Qiu, Y. Wang, and Y. Liang (2018). Efficient recurrent neural networks using structured matrices in fpgas. *arXiv preprint arXiv:1803.07661*.

Lipasti, M. and C. Schulz (2017). End-to-end stochastic computing.

Lipton, Z. C., J. Berkowitz, and C. Elkan (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

Liu, Y., Y. Wang, F. Lombardi, and J. Han (2018). An energy-efficient stochastic computational deep belief network. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1175–1178. IEEE.

Motamedi, M., P. Gysel, V. Akella, and S. Ghiasi (2016). Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 575–580. IEEE.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Nguyen, M. (2018). Illustrated guide to recurrent neural networks.

Olah, C. (2015). Understanding lstm networks.

Parhami, B. and C.-H. Yeh (1995). Accumulative parallel counters. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, Volume 2, pp. 966–970. IEEE.

Ren, A., Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan (2017). Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. *ACM SIGOPS Operating Systems Review 51*(2), 405–418.

Rosselló, J. L., V. Canals, and A. Morro (2012). Probabilistic-based neural network implementation. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7. IEEE.

Sanni, K., G. Garreau, J. L. Molin, and A. G. Andreou (2015). 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)*, pp. 1–5. IEEE.

Sim, H., S. Kenzhegulov, and J. Lee (2018). Dps: Dynamic precision scaling for stochastic computing-based deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, pp. 13. ACM.

Wang, S., Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang (2018). 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*, pp. 11–20. ACM.

Yann LeCun, Corinna Cortes, C. J. B. The mnist database of handwritten digits.

Yoshua Bengio, P. S. and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. In *IEEE Transactions On Neural Networks*, Volume 5. IEEE.

Yuan, B., C. Zhang, and Z. Wang (2016). 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)*, pp. 6555–6559. IEEE.

Zhang, C., P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170. ACM.

BIOGRAPHICAL SKETCH

Guy A. Maor's interest in computer engineering began during in his junior year of high school when he took a computer science class. Ever since then he has been interested in programming and understanding how computers work. He took a year and a half to complete his Associate of Science at Collin College before transferring to The University of Texas at Dallas where he majored in Computer Engineering for his undergraduate studies. There, he continued his education to earn his master's degree and interned at Texas Instruments for the summer of 2018, where he gained working experience in digital design. Throughout his master's program, he developed an interest in Embedded Systems, FPGA design, and Machine Learning which inspired him to write a thesis involving all three subjects.

CURRICULUM VITAE

Guy A. Maor

July 24, 2019

Contact Information:

Email: gxm130030@utdallas.edu

Educational History:

Associate of Science, Collin College, 2015
B.S., Computer Engineering, The University of Texas At Dallas, 2017
M.S., Computer Engineering, The University of Texas At Dallas, 2019

Thesis Advisor: Dr. Yang Hu

Employment History:

Digital Design Internship, Texas Instruments, June 2018 – August 2018