VLSI Implementation of Deep Neural Networks Using Integral Stochastic Computing

Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross

Abstract—The hardware implementation of deep neural networks (DNNs) has recently received tremendous attention since many applications require high-speed operations. However, numerous processing elements and complex interconnections are usually required, leading to a large area occupation and a high power consumption. Stochastic computing has shown promising results for area-efficient hardware implementations, even though existing stochastic algorithms require long streams that exhibit long latency. In this paper, we propose an integer form of stochastic computation and introduce some elementary circuits. We then propose an efficient implementation of a DNN based on integral stochastic computing. The proposed architecture uses integer stochastic streams and a modified Finite State Machine-based tanh function to improve the performance and reduce the latency compared to existing stochastic architectures for DNN. The simulation results show the negligible performance loss of the proposed integer stochastic DNN for different network sizes compared to their floating point versions.

I. INTRODUCTION

Recently, deep neural networks (DNNs), especially Deep Belief Networks (DBNs), have shown state-of-the-art results on various computer vision and recognition tasks [1]. The implementation of biologically-inspired artificial neural networks such as the Restricted Boltzmann Machine (RBM) has aroused great interest due to their excellent performance in performing complex machine learning tasks. They can be split in two phases, which are referred to as learning and inference phases [1]. The learning engine finds a proper configuration to map learning input data into their associated outputs, while the inference engine uses the extracted configuration to compute outputs for new data.

DBNs are constructed of multiple layers of RBMs and a classification layer at the end. The main computation kernel consists of millions of vector-matrix multiplications followed by non-linear functions in each layer. In the past few years, general purpose processors have been mainly used for software realization of both training and inference engines of DNN. However, large power consumption and high resource utilization are the main drawbacks of this approach. Since multiplications are costly to implement in hardware, existing parallel or semi-parallel VLSI implementations of such a network also suffer from high silicon area and power consumption. Moreover, hardware implementation of this network results in large silicon area: this is caused by the connections between layers, that lead to severe routing congestion. Therefore, an efficient VLSI implementation of DNN is still an open problem.

Recently, Stochastic Computing (SC) has shown promising results for ultra low-cost and fault-tolerant hardware implementation for various applications [2]. For instance, using unipolar SC, the multiplication and scaled addition are implemented using an AND gate and a multiplexer, respectively [2]. However, the multiplexer-based adder introduces a scaling factor that can cause precision loss [2], resulting in poor performance when using SC for deep neural networks that require many additions. In this paper, integral stochastic computation is introduced to solve the precision loss issue of conventional scaled adders. A novel finite-state machine (FSM)-based tanh function is then proposed as the non-linear function used in DNN. Finally, an efficient stochastic implementation of DNN based on the aforementioned techniques with a low misclassification error is proposed.

II. STOCHASTIC COMPUTING AND ITS COMPUTATIONAL ELEMENTS

In stochastic computation, numbers are represented as sequences of random bits. The information content of the sequence does not depend on the particular value of each bit, but rather on their statistics. Let us denote by \( X \in \{0, 1\} \) a bit in the random sequence. To represent a real number \( x \in [0, 1] \), we generate a sequence such that:

\[
E[X] = x,
\]

where \( E[X] \) denotes the expected value of the random variable \( X \). This is known as the unipolar format. The bipolar format is another commonly used format where \( x \in [-1, 1] \) is represented by setting:

\[
E[X] = (x + 1)/2.
\]

Note that any real number can be represented in one of these two formats by scaling it down to fit within the appropriate interval. In this paper, we use upper case letters to represent elements of a stochastic stream, while lower case letters represent the real value associated with that stream.

Multiplication of two stochastic streams is performed using AND and XNOR gates in unipolar and bipolar encoding formats, respectively. In unipolar format, the multiplication of two input stochastic streams \( A \) and \( B \) is computed as:

\[
Y = \text{AND} (A, B),
\]

and if the input sequences are independent, we have:

\[
y = E[Y] = a \times b.
\]

Multiplications in bipolar format can be also derived in a similar way.

Additions in SC are usually performed by using either scaled adders or OR gates [2]. The scaled adder uses a
multiplexer (MUX) to perform addition. The output $Y$ of a MUX with inputs $A$, $B$ and select line $S$ is given by:

$$Y = A \cdot S + B \cdot (1 - S).$$

(5)

If $S$ is independent from $A$ and $B$, and $\mathbb{E}[S] = 1/2$, the output of the MUX represents $y = (a + b)/2$. This 2-input scaled adder ensures that its output is in the legitimate range of each encoding format by scaling it down by factor of 2. For more than 2 inputs, scaled addition can be performed using a tree of 2-input MUXs. The result of such an adder is scaled down multiple times, which can decrease the precision of the stream. To achieve the desired accuracy, longer bit-streams must be used, resulting in larger latency.

OR gates can also be used as approximate adders. The output $Y$ of an OR gate with independent inputs $A$, $B$ can be expressed as

$$Y = A + B - A \cdot B.$$  

(6)

OR gates function as adders only if $\mathbb{E}[AB] \ll 1$ and $\mathbb{E}[A] + \mathbb{E}[B] < 1$. Therefore, the inputs should first be scaled down to ensure that the aforementioned conditions are met. This type of adder still requires long bit-streams to overcome a precision loss incurred by the scaling factor.

To improve the efficiency, the Accumulative Parallel Counter (APC) was proposed in [2]. The APC takes $N$ parallel bits as inputs and adds them to a counter in each clock cycle of the system. Therefore, this adder preserves all the information and results in lower latency due to its small variance of the sum. It is also worth mentioning that this adder converts the stochastic stream to binary form [2]. Therefore, this adder is restricted to cases where additions are performed to obtain the final result, or requiring an intermediate result in binary format.

The hyperbolic tangent function is a computation required by many applications. This function is implemented in the stochastic domain by using a FSM [3]. We denote the number of states in the FSM by $n$. For a FSM taking an input $X$ and producing an output $Y$, we define a function $\text{Stanh}(n, X)$ that gives the stochastic stream $Y$ when $\mathbb{E}[X] = x$. The FSM is constructed such that

$$\mathbb{E}[Y] = \mathbb{E}[\text{Stanh}(n, X)] \approx \tanh \left( \frac{nx}{2} \right),$$  

(7)

where $n$ denotes the number of states in the FSM. It is worth mentioning that both input and output of the Stanh function are in bipolar format.

III. PROPOSED INTEGRAL STOCHASTIC COMPUTING

A. Generation of Integer Stochastic Stream

An integer stochastic stream of degree $m$ is a sequence of integers $S_i \in \{0, 1, \ldots, m\}$ that are represented in the usual binary numeral system (e.g. 2’s complement). The average value of this stream is a real number $s \in [0, m]$ for unipolar format and $s \in [-m, m]$ for bipolar format, where $m \in \{1, 2, \ldots\}$. In other words, the real value $s$ is the summation of $m$ stream probabilities. For instance, 1.5 can be expressed as 0.75 + 0.75. Each of these probabilities can be represented by a conventional binary stochastic stream as shown in Fig. 1(a). Therefore, the integer stochastic representation of 1.5 can be readily achieved as a summation of generated binary stochastic streams as illustrated in Fig. 1(b). In general, the integer stochastic stream $S$ representing the real value $s$ is a sequence with elements $S_i, i \in \{1, 2, \ldots, N\}$:

$$S_i = \sum_{j=1}^{m} X_i^j,$$  

(8)

where $X_i^j$ denotes each element of a binary stochastic sequence representing a real value $x^j$. The expected value of the integer stochastic stream is then given by

$$s = \mathbb{E}[S_i] = \sum_{j=1}^{m} x^j.$$  

(9)

We can also generate integer stochastic streams in the bipolar format. In that case, the elements $S_i$ of the stream are given by:

$$S_i = 2 \times \sum_{j=1}^{m} X_i^j - m,$$  

(10)

and the value represented by the stream is

$$s = \mathbb{E}[S_i] = 2 \times \sum_{j=1}^{m} \mathbb{E}[X_i^j] - m = 2 \times \sum_{j=1}^{m} x^j - m.$$  

(11)

Any real number can be approximated by using an integer stochastic stream without prior scaling, as opposed to a conventional stochastic stream which is restricted only to the [-1, 1] interval. In integral SC, computation on two streams with different effective lengths is also possible while conventional SC fails to provide this property. For instance, representation of 0.875 and 0.5625 require effective bit-stream lengths of 8 and 16, respectively, using conventional SC. Therefore, effective bit-stream lengths of 16 is used to generate the conventional stochastic bit-stream of these two numbers for operations. However, the second number which requires higher effective length, i.e., 0.5625 in this example, can be generated
by using integral SC with \( m = 2 \) as shown in Fig. 2. In this case, the bit-stream length of 8 is used for both numbers and operations can be performed by using smaller lengths compared to conventional SC. This technique potentially reduces the latency brought by stochastic computations, making integral SC suitable for throughput-intensive applications.

B. Multiplication In Integral SC

The main advantage of SC compared to the binary radix format is the low complexity implementation of mathematical operations. It is shown that multiplication can be implemented by using AND or XNOR gates depending on the coding format. However, integer stochastic multipliers make use of binary radix multipliers (see Fig. 3(a)). The multiplication of an integer stream \( S^1 \) of degree \( m^1 \) with a stream \( S^2 \) of degree \( m^2 \) can be performed by evaluating \( Y = S^1 \times S^2 \) using a binary radix multiplier. If the streams are independent, then \( y = s^1 \times s^2 \). The implementation cost of this multiplier strongly depends on \( m^1 \) and \( m^2 \). However, an important special case is when either \( m^1 = 1 \) or \( m^2 = 1 \). In that case, the multiplication can be implemented using bit-wise AND gate or a MUX as depicted in Fig. 3(b). The range of \( y \) is \([0, m \times m']\) in the unipolar case, and \([-m \times m', m \times m']\) in the bipolar case.

C. Addition In Integral SC

Conventional SC suffers from precision loss when scaled adders are used, making it inappropriate for applications which require many additions. On the other hand, integer SC uses binary radix adders to perform additions without loss of precision. Using (9), addition in unipolar format is performed as follows:

\[
y = s^1 + s^2 = \mathbb{E}[s^1 + s^2] = \mathbb{E}[S^1_1] + \mathbb{E}[S^2_1],
\]

since the expected value operator is linear.

Equation (12) remains valid also in the bipolar case, while the range of \( y \) is \([0, m + m']\) and \([-m + m'], m + m']\) for unipolar and bipolar formats respectively. This adder provides some advantages similar to APC. First of all, due to the fact that it retains all precision provided as inputs, it reduces the variance of the sum. Secondly, it potentially reduces the bit-stream length required for computations compared to conventional SC [2]. Moreover, the output of this adder is still an integer stochastic stream, which can be used by subsequent stochastic computational units, as opposed to APC.

D. FSM-Based Functions In Integral SC

In binary SC, the inputs of stochastic FSM-based \( \tanh \) function is restricted to real values in the \([-1, 1]\) interval. Therefore, a desired \( \tanh \) function can be achieved by scaling down the inputs and adjusting the term \( n \) in (7), which potentially increases bit-stream length and results in long latency. The transition between each state of FSM is performed according to the input value in bipolar format, which is either 1 or 0.

The integer stochastic \( \tanh \) function is proposed by generalizing conventional FSM. In integral SC, each element of a stochastic stream is represented using 2’s complement or sign-magnitude representations in \([-m, \ldots, m]\) for bipolar format. A state counter is increased or decreased according to the integer input value \( S_i \in \{-m, \ldots, m\} \) where \( i \in \{1, 2, \ldots, N\} \). Therefore, the state counter is incremented or decremented by up to \( m \) in each clock cycle, as opposed to conventional FSM-based functions which are restricted to one-step transitions. In integral SC, the counter of FSM-based function is saturated in \([0, n \cdot (m - 1)]\) and incremented by \( S_i \) in each clock cycle.

The output of the proposed integer FSM-based functions in integral SC domain and its encoding format are similar to the conventional FSM-based functions. For instance, the output of the integer \( \tanh \) function is in bipolar format. Moreover, the integer FSM-based functions require \( m \) times more states compared to its conventional counterpart. Therefore, the approximate function of integer \( \tanh \) function, which is referred to as NStanh, is:

\[
\mathbb{E}[	ext{NStanh}(m \times n, S)] \approx \tanh \left( \frac{ns}{2} \right).
\]
IV. INTEGER STOCHASTIC IMPLEMENTATION OF DNN

A. A Review on the DNN Algorithm

DBNs are the hierarchical graphical models obtained by stacking RBMs on top of each other and then training them in a greedy unsupervised manner [4]. DNNs take low-level inputs and construct higher-level abstractions through the composition of layers. Adjusting the number of layers and their sizes provides different levels of abstractions.

In this paper, a two-layer DNN followed by a classification layer at the end is used for handwritten digit recognition, which is implemented using an integer stochastic architecture. As a benchmark, we use the Mixed National Institute of Standards and Technology (MNIST) data set. This data set provides thousands of $28 \times 28$ pixel images for both training and testing procedures. Each pixel is represented by an integer number between 0 to 255, requiring 8 bits for digital representation. The training procedure can be performed on central processing units (CPUs) or graphics processing units (GPUs). Therefore, the extracted weights are stored in a memory for the hardware inference engine to classify the input images in real-time.

The implemented network in this paper is constructed using two layers of RBM, which are also called hidden layers, followed by a classification layer. Inputs of DNN and outputs of a hidden layer are hereafter referred to as visible nodes and hidden nodes, respectively. The hierarchical computations of each hidden layer are performed as follows:

$$z_j = \sum_{i=1}^{M} W_{ij} v_i + b_j,$$

(14)

$$h_j = \frac{1}{1 + \exp(-z_j)} = \sigma(z_j),$$

(15)

where $M$ denotes the number of visible nodes, $v_j$ the value of visible nodes, $W_{ij}$ the extracted weights, $b_j$ the bias term, $z_j$ intermediate value, $h_j$ the output value of each hidden node and $j$ an index to each hidden node. The nonlinearity function used in DNN, i.e., equation (15), is called a sigmoid function. The classification layer does not require a sigmoid function as it is only used for quantization. In other words, the maximum value of the output denotes the recognized label.

B. The Proposed Stochastic Architecture of a DNN

VLSI implementations of a DNN network in binary form are computationally expensive since they require many matrix multiplications. Moreover, there is no straightforward way to implement the sigmoid function in hardware. Therefore, this unit is normally implemented by LUTs, which requires extra memory in addition to the memory used for storing weights. Considering 10 bits for weights, $78400 \times 8b$ multipliers are required to do the matrix multiplications of the first hidden layer for a parallel implementation of a network with configuration of $784-100-200-10$, meaning 784 visible nodes, 100 first-layer hidden nodes, 200 second-layer hidden nodes and 10 output nodes.

Stochastic implementation of DNN is a promising approach to perform the mentioned complex arithmetic operations using simple and low-cost elements. In order to find the output value of the first hidden node, 784 multiplications are required, which can be easily performed by using AND gates in unipolar format. Then, addition of multipliers output should be performed by using a scaled adder or an OR gate. Using a scaled adder to sum 784 numbers requires an extremely long bit-stream due to the fact that the output result of this adder is scaled down by 784 times, a very small number to be represented by short stream length. In [5], an OR gate is used as an adder to perform this computation while the inputs first are scaled down to make the term $A \cdot B$ close to 0 in (6), which potentially increases the required stream length for computations.

We have shown in Section III-A that the integer stochastic stream can be generated by adding conventional stochastic streams. Considering that the multiplications of the first layer of a DNN are performed in conventional stochastic domain, the nature of the algorithm is to add the multiplication results together. Exploiting a binary tree adder, the addition result remains in integer-stochastic form without any precision loss. The sigmoid function can also be implemented in the integer stochastic domain.

It is well-known that the sigmoid function can be computed through tanh function as follows:

$$\sigma(x) = \frac{1 + \tanh \left( \frac{x}{2} \right)}{2}.$$  

(16)

The tanh function can also be implemented by NStanh function (see (13)) in integer stochastic domain. The output of NStanh is in bipolar format in conventional stochastic domain. Therefore, considering its output in unipolar format according to (16) and (2), the output of NStanh is equivalent to the sigmoid function in stochastic domain.

Fig. 5 shows the proposed integer stochastic architecture of a single neuron. The input signal stream is generated by...
TABLE I

THE MISCLASSIFICATION ERROR OF THE PROPOSED ARCHITECTURES FOR DIFFERENT NETWORK SIZES AND STREAM LENGTHS

| Implementation Type: Floating Point [6] | SC [5] | Integral SC (Proposed) |
|-----------------------------------------|--------|------------------------|
| m:                                      | 2      | 1                      |
| Stream Length:                          | 1024   | 512                    |
| 784-100-200-10                          | 2.29   | 2.40                   |
| 784-300-600-10                          | 1.82   | 2.01                   |

TABLE II

THE AVERAGE ADP OF NEURONS FOR A 784-100-200-10 NETWORK @400 MHZ IN A 65 NM CMOS TECHNOLOGY

| The Proposed Integral SC                  |
|------------------------------------------|
| m:                                       |
| 1                                       |
| 2                                       |
| 4                                       |
| Stream Length                            |
| 1024                                    |
| 512                                     |
| 256                                     |
| ADP [mm^2 * ns]                          |
| 58.88                                   |
| 56.32                                   |
| 54.4                                    |

using conventional stochastic domain: however, the weights are represented by 2’s complement format in integer stochastic domain with range of \( m \), which requires \( \log_2 (m) + 1 \) bits for representation. The multiplications are performed bit-wise by AND gates. A tree adder and an NStanh unit are used to perform the additions and nonlinearity function, respectively. The output of the integer stochastic sigmoid function is represented by a single wire in unipolar format. Therefore, the input and output formats are the same. Integer stochastic architecture of DNN is formed by stacking the proposed single neuron architecture. In the simulations, we used 10 bits to represent the weights in binary radix format. Each pixel in the input images requires a minimum bit-stream length of 256, but the weights require a minimum bit-stream length of 1024 in conventional stochastic domain. Therefore, the latency of the proposed integer-stochastic implementation of the DNN is equal to 1024 for \( m = 1 \).

The input range of the NStanh function as the last parameter of the proposed architecture is selected through simulation. The simulation results show that the probabilities of inputs of the NStanh units are mostly focused between either -4 to 4 or -6 to 6 depending on the network size. The degree of NStanh units can be determined by taking a window which covers 95% of data. This range strongly depends on the correlations among the stochastic inputs. The range would be a bigger number as the correlation increases. In this paper, \( m' = 6 \) and \( m' = 4 \) (see Fig. 5), depending on the network size, are used for the simulations.

V. SIMULATION AND IMPLEMENTATION RESULTS

The misclassification error rate of DNNs plays a crucial role in the performance of the system. In this part, the misclassification errors of the proposed integer stochastic architectures of DNNs with different configurations are summarized in Table I. Simulation results have been obtained by using MATLAB on 10000 MNIST handwritten test digits [7] for both floating point code and the proposed architecture. The method proposed in [6] is used as our training core to extract the network weights. A 10-bit precision is used to represent the weights, which requires a stream length of 1024. For instance, stream length of 1024 is used for generation of both integer stochastic stream of weights with \( m = 1 \) and stochastic stream of input images in this case. In a second scenario, stream length of 512 is used for generation of both integer stochastic stream of weights with \( m = 2 \) and the binary stochastic of input pixels. The 10-bit precision is kept by increasing the value of \( m \) for weights while the precision of input images is decreased since it only requires 8 bits for representation in binary form. Therefore, the stream length can be at most reduced to 256 while using AND gates for multiplications. The simulation results show the negligible performance loss of the proposed integer stochastic DNN for different sizes compared to their floating point versions as opposed to the previously proposed implementation of deep neural network based on SC.

The average area-delay product (ADP) of neurons with different degrees measured in a 65 nm CMOS Technology is provided in Table II. These results shows that the average ADP of neurons remains roughly constant as the degree is increased while the latency is reduced. Therefore, the integral SC can be used as a technique reducing the latency without changing the ADP.

VI. CONCLUSION

Integral SC makes the hardware implementation of precision-intensive applications feasible in the stochastic domain, and allows a computation to be performed on streams of different lengths, which can improve the latency of the system. An efficient stochastic implementation of a deep neural network is then proposed using integral SC. The simulation results show that the proposed architecture can perform the classification task with the negligible performance loss with stream length of 256 as opposed to the existing stochastic architecture of DNN.

REFERENCES

[1] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, “A 4.6 A1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications,” in IEEE Int. Solid-State Circuits Conference - (ISSCC), Feb 2015, pp. 1–3.
[2] P.-S. Ting and J. Hayes, “Stochastic Logic Realization of Matrix Operations,” in 17th Euromicro Conf. on Digital System Design (DSD), Aug 2014, pp. 356–364.
[3] B. Brown and H. Card, “Stochastic neural computation. I. Computational elements,” IEEE Trans. on Computers, vol. 50, no. 9, pp. 891–905, Sep 2001.
[4] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” Science, vol. 313, no. 5786, pp. 504–507, July 2006.
[5] B. Li, M. Najafi, and D. J. Lilja, “An FPGA implementation of a Restricted Boltzmann Machine classifier using stochastic bit streams,” in IEEE 25th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP), July 2015, pp. 68–69.
[6] M. Tanaka and M. Okutomi, “A Novel Inference of a Restricted Boltzmann Machine,” in 22nd Int. Conf. on Pattern Recognition (ICPR), Aug 2014, pp. 1526–1531.
[7] Y. Lecun and C. Cortes, “The MNIST database of handwritten digits.” [Online]. Available: http://yann.lecun.com/exdb/mnist/