Stochastic Processors on FPGAs to Compute Sensor Data Towards Fault-Tolerant IoT

***Abstract*—The continuous increase in the amounts of data received at the edges of the Grid is making necessary to pre-compute data at the IoT device before communicating it over the network. Moreover, in the IoT context the devices are often required to operate under heavy power and area constraints and subjected to harsh environments. However, in this context, traditional computing paradigms struggle to provide high availability and fault-tolerance. Stochastic Computing has emerged as a competitive computing paradigm that produces fast and compact implementations of arithmetic operations, while offering high levels of parallelism, and graceful degradation when in the presence of errors. Stochastic Computing is based on the computation of pseudo-random sequences of bits, hence requiring only a single bit per signal. In virtue of the granularity of the bitstreams, the bit-level specification of circuits, high-performance characteristics and reconfigurable capabilities, FPGAs are often adopted to implement and test such systems. The proposed framework takes a high-level specification and automatically creates a complete Stochastic Computing systems capable of interfacing sensors directly on the FPGA, and perform computations over the stochastic bitstreams. Moreover, the presented framework is also able to generate custom stochastic processing units, perform fault-tolerance tests, and report estimates on performance, resources and power. As proof-of-concept, this paper presents two Machine Learning applications typical in IoT, implementing Karhunen-Loeve Transforms and Neural Networks, and compares them against typical implementations.**

***Index Terms*—IoT, Fault-Tolerance, Stochastic Computing, Stochastic Bitstreams, Approximate Computing, FPGA, Evaluation Framework, Karhunen-Loeve Transform, Neural Networks.**

I. Introduction

Edge computing intends to compute vast amounts of data, constantly generated by IoT devices, before communicating the resulting data to the Grid. Moreover, to alliviate the Edge servers from computing basic, but essential, Digital Signal Processing (DSP) and Machine Learning (ML) functions, there is interest in delegating such computing to the IoT device. However, in the IoT context devices are often required to operate under heavy power and area constraints and subjected to harsh environments, struggle to provide high availability and fault-tolerance. To overcome such limitations, this work proposes to make use of a different computing paradigm that blends well with the IoT context, and offers direct analog sensor interface without Analog-to-Digital Converters (ADCs), fault-tolerance and savings in resources and power.

Stochastic arithmetic has emerged as an alternative computational paradigm able to provide approximate computations requiring less hardware, towards a circuit design with simpler but massively parallel components, trading off precision for computation time [1].

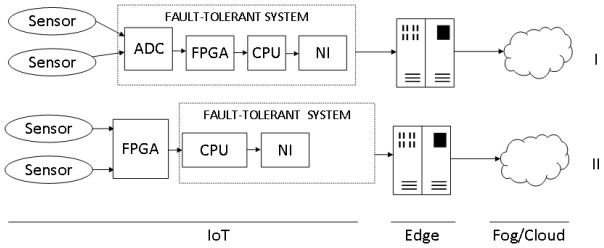

Fig. 1. Illustration of two scenarios for a typical IoT application, with (I) and without (II) stochastic computing.

Applications such as neuromorphic systems [2], [3], [4], bio-inspired systems [5], neural networks [6], [7], high-throughput Bayesian inference [8], image and video processing [9], Finite Impulse Response (FIR) [10] and Infinite Impulse Response (IIR) [11] digital filters, and autonomous cyber-physical systems [12] are characterized for their regularity in their data-path. Their computations are mainly based on multiple multiplications followed by accumulations. Moreover, many of these applications do not require exact results and can tolerate some deviations in their computations.

The operation of Stochastic Computing (SC) closely resembles biological neural systems, which are known to excel in robustness, power-efficiency, and massively parallel neural elements. Moreover, they have highly reconfigurable and plastic connections, a feature which can be emulated by reconfiguration capabilities of Field-Programmable Gate Arrays (FPGAs). In addition, the bit level specification of stochastic bitstreams makes it favorable for implementation on these devices.

Figure 1 illustrates the typical scenario (I) where a dependable IoT system requires the implementation of fault-tolerance mechanisms at all levels of the system, even though it only acquires data from the sensor and communicates the data to the Edge servers. The figure also presents the new approach (II) using SC, which performs computations directly over he acquired stochastic bitstream, thus alleviating the computational load at the Edge. Moreover, by adopting SC, the proposed approach also reduces the require fault-tolerance mechanisms at the IoT and Edge levels.

Insofar, the majority of research conducted on SC is confined to a set of applications, which are highly customized,

specific to certain applications and difficult to extend its adoption. Furthermore, the benefits of SC are not always clear due to the resources of the supporting elements and the clock latency to process long bitstreams. Often, the benefit of SC is shadowed by the latency and resources required to interface a traditional computing systems.

As a motivational example, the Bayesian inference system presented in [13] requires 597 Logic Elements (LEs) to be implemented, of which, only 42 are spent on the datapath for the Bayesian Machine. The remaining 555 LEs are spent on conversion of 13 stochastic bitstreams. The contribution in [14] presents a comparison of parallel binary versus stochastic implementations for neural networks on reconfigurable hardware. The authors concluded that even though stochastic bitstreams require more clock cycles to compute than binary, the advantage of compact realizations in hardware surpasses that through comparison of geometric mean of the two metrics. Therefore there is a need for a methodology to make this assement at an earlier stage of the design process.

The main challenge addressed in this paper is to ease the definition and evaluation of a Stochastic Processor (SP) to compute, at the IoT-level, mathematical expressions as alternative to other time consuming and prone-to-error design approaches, and without having to delve into the technicalities of High-Level Synthesis (HLS).

This contribution proposes a highly customizable and scalable framework that given a problem's specification as mathematical expression, it generates the corresponding SP, and its supporting blocks, targeting reconfigurable logic. This work is intended to facilitate automated architectural changes via unified and regular interfaces, and design-space exploration often sought in research due to the long execution times. Moreover, this work provides an estimate of resources, power and performance metrics. This enables the usage of the SC in stand-alone stochastic systems or accelerators for heterogeneous and System-on-a-Chip (SoC) platforms.

The main challenge is to find a quick and easy method to describe the datapath of the system which is also facilitates the implementation of applications.

The flow of the proposed framework is illustrated in Fig. 2. It starts with a high-level specification from the user as a mathematical expressions described in Python, which are translated into a computational stack, as sequences of inter-connected operands and operators. The framework then generates the Register Transfer Level (RTL) in Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) for custom SC arithmetic units, the SP which implements the desired functionality, and the supporting blocks.

The main contribution of this work is a framework for fast prototyping os SC systems on FPGAs, and its demonstration generating circuit designs for algorithms common in IoT. Moreover, the Karhunen-Loeve Transform (KLT) algorithm can be implemented as inner product, which is the same for an FIR filter, hence demonstrating its wide range of applicability of the proposed work.

This paper is organized as follows: section II is devoted to introduce SC and presents the most relevant research contributions incorporated in the proposed framework. Section III introduces the inner workings of the proposed framework, to generate VHDL entities and the data-path for the mathematical expression to be implemented. Section IV presents the details about the proposed framework. A demonstration of a large-scale stochastic system with the first implementation of the KLT implemented on an FPGA. Conclusions and final remarks are in section VIII.

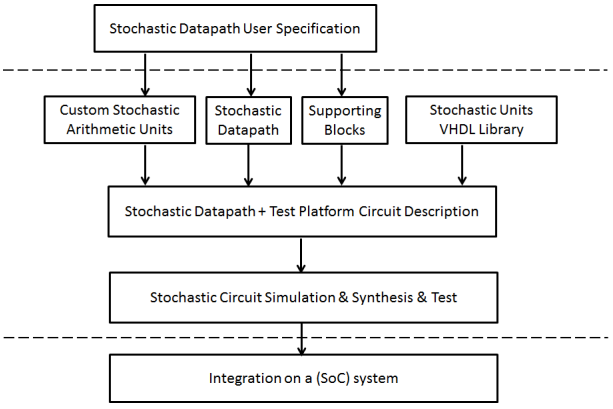

Fig. 2. Flow of the proposed framework to simulate, synthesize and evaluate Stochastic Computing systems on FPGAs.

II. Background on Stochastic Computing

J. Von Neumann introduced SC in [15] as a method to design probabilistic logic circuits and synthesize robust systems from unreliable components. In [16], Gaines has introduced the use of stochastic bitstreams to represent operators with high levels of error tolerance.

A. Stochastic Bitstreams

By definition, a stochastic signal is the result of a continuous-time stochastic process which produces two values: 0 and 1. (bipolar). According to [1], a unipolar stochastic bitstream is a sequence of stochastic signals over time whose value is within $[0; 1]$ and defined as the number of ones (o) over the total number of bits (t). In bipolar representation the value is within $[-1; 1]$ and is also encoded as a ratio but followed by a negative bias and a scale factor of 2. On stochastic bitstreams there are no weights in the representation, as in typical binary-radix representation, thus all bits have the same contribution for the encoded value. For example, the same sequence of 8 bits 10110110 represents $5/8 = 0.625$ in unipolar and $2 * (5/8 - 0.5) = 0.25$ in bipolar. Fig. 3 illustrates the aforementioned stochastic bitstream. On top there's the clock signal, to ensure synchronism; and on bottom the encoded value.

B. Arithmetic Units

To perform arithmetic computations several stochastic arithmetic units have been proposed, including an adder, and a multiplier, as illustrated in Fig. 4.More details on stochastic

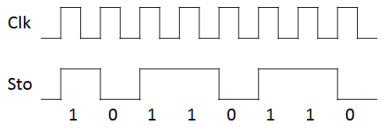

Fig. 3. Example of a stochastic bitstream encoding 0.625 and 0.25 in unipolar nd bipolar encodings, respectively.

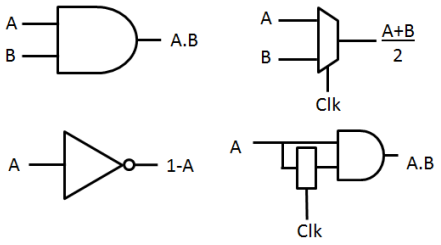

Fig. 4. Block diagram of the unipolar stochastic units: a) multiplier (top-left), b) adder (top-right), c) negation (bottom-left) and d) squarer (bottom-right).

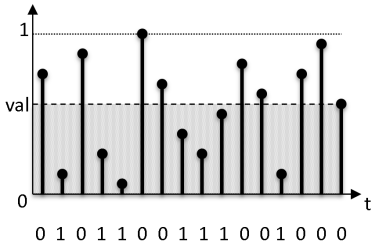

Fig. 5. Detail on process of generating a pseudo-random bitstream for a given binary-radix value between 0 and 1.

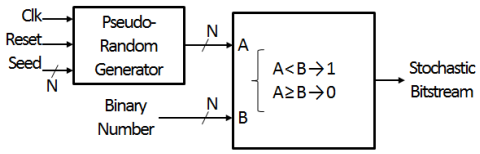

Fig. 6. Block diagram of a binary-to-stochastic unit.

arithmetic units can be found in the survey presented in [17] which covers the most common arithmetic units.

The unipolar stochastic multiplication is only the result of a logic AND of its stochastic inputs. The complement is the negation of the bitstream. Bipolar multiplication is achieved through an XNOR operation. Addition, or more precisely average, is obtained via a round-robin multiplexation of the stochastic inputs, which depends on a N-module counter corresponding to N inputs in the multiplexer. The squarer of a stochastic bitstream is the equivalent to a multiplication of a bitstream by itself, de-phased by a clock cycle. The clock cycle makes the pseudo-random bitstreams to be uncorrelated. The multiplication is now of two independent streams but with the same value, resembling the power of two operation.

The implementation of n-ary add or multiply operators is achieved by adding additional inputs to the logic circuits. In terms of FPGA implementation it means that the number of inputs of a LE can be evaluated simultaneously.

For more complex operators, such as exp, $tanh$ and abs, there are realizations of stochastic operators using Finite State Machines (FSMs). Implementation of such units can be found in [18], [19].

C. Supporting Blocks

Since most systems usually adopt parallel binary-radix representation, a converter from/to stochastic bitstreams is therefore required to ensure its inter-operability. The process of generating the stochastic bitstream is illustrated in Fig. 5, where a specific binary-radix value (*val*) is compared with the output of a uniform pseudo-random generator, usually an Linear Feedback Shift-Register (LFSR) [20]. Whenever the pseudo-random number is smaller, it produces a 1 and 0 otherwise. After each pseudo-random sample the ratio between the number of ones and the total number of bits will be towards *val*. In this example, the encoded value is $9/16 = 0,5625$. The conversion from stochastic-to-binary is based on the integration of the 1s on a bitstream, which is accomplished using a binary-radix counter. A second counter is required to count the total number of bits. Fig. 6 and 7 show the details of the conversion units.

To improve the statistical quality of the stochastic bitstreams on the datapath, this work adopts the Self-Timed Ring-Oscillator (STRO) proposed in [13]. The main reasons to consider a STRO instead of a global clock source are: different clock signal for each stochastic unit; all generated clock signals with the variation of voltage, temperature, location on the device and its degradation. All synchronous stochastic units have an instance of this unit to generate its clock signal.

D. Graceful Degradation

The graceful degradation of stochastic bitstreams is referred to the impact of bit-flips on the bitstream. In such occurrences, and regardless of the position of the bit on the bitstream, the value of the error associated with each bit-flip is the same as the least significant bit, in binary-radix. On this account, [9] has applied the concept of stochastic logic to a reconfigurable architecture that implements image processing operations on a simulated data-path. The authors show that the quality of the results degrades gracefully with the increase of errors on the bitstream.

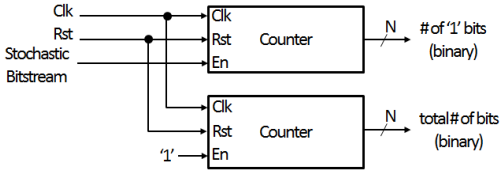

Fig. 7. Block diagram of a stochastic-to-binary unit.

E. Main Limitations

The main fragilities of SC are: a linear increase in the precision of typical binary representations, for stochastic computations it imposes an exponential increase in the length of the bitstream; sensitivity to temporal correlations; and the supporting blocks are usually the performance bottleneck, rather than the arithmetic units.

III. From Mathematical Expressions to Reverse Polish Notation to Stochastic Datapaths

Like any datapath, SP implements a chain of computations, but over a bitstream. This work proposes a method to specify it as a mathematical expression, defined as a list of operands and operators, organized in a stack to resemble Reverse Polish Notation (RPN), or postfix notation. The advantages of such representation are: simplified representation without parenthesis, hence fewer operations are needed, faster introduction by the user and with fewer mistakes [21], [22]. In RPN the operators follow the operands. The strength of this notation is the support of n-ary operators, which is compatible with the aforementioned stochastic operators. Example: $3 \ - \ 4 \times \ 5$ is defined in RPN as $3 \ 4 \ 5 \times -$. Essentially, the framework recognizes the different operands and operators of a mathematical expression, and then generates the corresponding VHDL. From this data structure it is possible to identify the requirements for a system, namely: the number of input, internal and output signals; the different types, number of input operands and data dependencies of the operators used. The data structure is organized as a tree of computations which maintains the data dependencies in the data-path. These mathematical expressions can be variable in size and type of operations.

Considering the following motivational example of a function to be implemented:

$$f = \frac{1}{N}\left(in_0 \times in_1 + in_2 \times in_3 \times in_4\right) \tag{1}$$

it has the corresponding RPN stack representation:

$$f = in_0 \ in_1 \ \times \ in_2 \ in_3 \ in_4 \ \times \ \times \ + \ N \ / \tag{2}$$

To facilitate the stack manipulation, it is split into a set of partial computations stored in intermediate variables:

$$tmp_0 = in_0 \times in_1 \tag{3}$$

$$tmp_1 = in_2 \times in_3 \times in_4 \tag{4}$$

which the original expression can be replaced with:

$$f = \frac{1}{N}\left(tmp_0 + tmp_1\right) \tag{5}$$

and to facilitate the generation of the VHDL source file describing the data-path to implement this expression, it is expressed as a list of operands and operators in Python, e.g. sums and multiplications, resembling RPN. This regular form is easily extracted and can be efficiently mapped into an RTL specification, exploiting the parallelism offered by FPGAs.

The user input for equation 2 in Python can be described by the following list of computations, which itself can be comprised of other lists, or operands, and operators:

```
t1 = ['in0', 'in1', '*'];
t2 = ['in2', 'in3', 'in4', '*'];
f = [t1, t2, '+'];
```

which results in the following Python variable:

```
>>> f
[['in0', 'in1', '*'], ['in2', 'in3', 'in4',
'*'], '+']
```

Variables `t1` and `t2` are lists of strings, which represent partial computations. These variables can be of any size. The last element, or tail, of the list holds the representation of the operation. In this example, the operands are:d `+` or `*`. The remaining elements are the operands. It is also possible to define operations which depend on the results of previous computations, e.g. `f` is defined as the sum, or average, of `t1`, and `t2` . Table I lists the stochastic combinatorial and sequential operators supported so far.

TABLE I
List of the stochastic operators supported

Operator	Nomenclature
Sum (average)	`+`
Multiplication	`*`
Negation	`-`
Square	`pow2`

The inputs and outputs of the SP correspond to the number of variables and are automatically determined. To complete the specification of a datapath it is necessary to indicate the length and type (unipolar/bipolar) of the bitstream. To serve this purpose there is a variable in Python which holds this configuration. The architecture of the SP is independent of the bitstream's length.

IV. Stochastic Framework

One of the key strengths of the proposed framework is that given any mathematical expression, regardless the complexity of the mathematical expressions, the system maintains its regularity.

The framework integrates the translation of the expression of the SP into a stack. Apart from the core of the SP, which is different for all expressions, all other supporting units have the same architecture, such as data sources and sinks for the stochastic bitstreams, varying only the number of bits, or the length of the bitstreams supported.

The generated SP was planned to be autonomous or part of a larger system, as illustrated in Fig. 8. The SP is in the middle and the rest of the circuit is formed by the supporting units to do the computations. The system is interfaced via the input and output bitstreams, and also the FSM's control signals, namely *Clk*, *Enable* and *Reset*. In particular, the FSM is responsible for the generation of the control signals for all units in the design. It also controls the burn-in period to compensate the clock cycles required by the FSM-based stochastic arithmetic units.

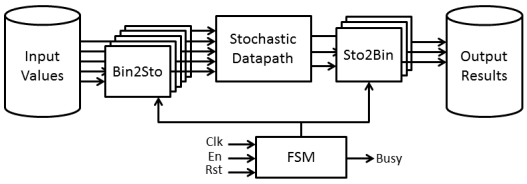

Fig. 8. Top-level architecture of the circuit design to test the Stochastic Datapaths, including the supporting units.

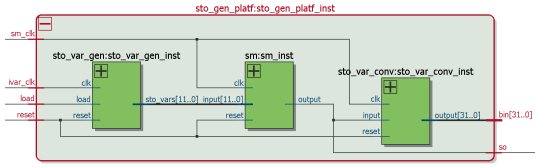

Fig. 9. RTL of a test circuit for a Stochastic Datapath.

A. Generation of Custom Arithmetic Units From Templates

In typical binary-radix representation all basic operators are either unary or binary, with 1 or 2 inputs, respectively. However, has n-ary SC operators support for more than two operands. Therefore, it is required to create the required custom components, as it is difficult to account for all possible operators with any number of operands in advance. Therefore, the framework determines the number of arguments for multiplications and sums and then generates the required stochastic arithmetic components. In more detail, it iterates over the aforementioned list of computations to retrieve the different operators and then generates the VHDL entity matching the operation and the number of inputs.

In SC, each operator can have a diverse number of operands. Therefore, it is necessary to generate custom arithmetic units according to the mathematical expression. Moreover, the number of variables considered is unknown, so it is also necessary to create the necessary interfaces to support any number of inputs and outputs. The VHDL source files are created, and used to synthesize the design and generate the FPGA configuration file, or to simulate the system. This process has been automated through the use of Python and Tool Command Language (TCL) scripts.

B. Interfaces

Conversion between binary-radix and stochastic bitstreams is the major limitation in interfacing typical digital systems. Even tough it offers many parallel operators there are not many inputs available.

Connecting the SP from the rest of the supporting elements allows to integrate it in other systems, capable of interfacing with stochastic bitstreams, such as [12]. In this work the authors have created a cyber-physical system which interfaces analog sensors and actuators without the need to have either analog to digital and binary-to-stochastic converters, to acquire input data; and stochastic-to-binary and digital to analog converters to drive the actuators.

In essence, the generation of a bitstream from a binary-radix value requires more resources than an analog interface, but the analog interface requires a dedicated input pin.

C. Advanced Features

Keep up with the novel advancements in SC. In this direction the framework already includes a few research novelties to demonstrate its scalability. The incorporated features which mitigate some of the limitations in SC and improve the quality of the results

1) Burn-in Period: The values at the output are not ready in the same instant the computation starts. Therefore, a counter in included to accommodate sufficient latency to have the SP producing valid results. Once the counter reaches the threshold value, the outputs are computed.

2) Independent and Uncorrelated Clock Sources: In [23], the authors claim to reduce the correlation between bitstreams by using STROs to generate spread-spectrum, individual and uncorrelated clock sources for each synchronous stochastic unit. Moreover, the authors also claim reduction in the power dissipated in the clock trees, without the penalty of introducing synchronizers, or alternative components, typical of asynchronous circuit designs [24]. This feature, which consists of a configurable length ring oscillator can be instantiated to provide the clock signal to all synchronous components in a SP.

D. Test Platform

The proposed framework provides a test platform to run any SP generated by it on an FPGA. It creates a fully functional autonomous stochastic system, containing the SP derived from the mathematical expression. The system supports SP of any size, being limited by the resources available on the FPGA device. This test platform manages the input and output signals required by the SP, along with the required conversions to be accessed by the host computer. Fig. 9 depicts the system to be implemented on the FPGA. On the edges there are the conversion blocks, and in the middle the unit corresponding to the SP.

1) Circuit: The test platform circuit is constituted by the circuit under test (i.e. a simple arithmetic unit or a SP), the bitstream generators, and the output calculators.

It includes the units for the generation of the stochastic bitstreams from binary values previously stored in Block Random Access Memorys (BRAMs), and the result converters back to binary and its storage in other BRAMs. In more detail, each of these units supports many parallel bitstreams.

2) Operation: Once the FPGA is configured with the test circuit, via its Joint Test Action Group (JTAG) port, it is ready to exchange data with the host computer. To automate the process on the host computer a TCL script has been created to download the data also via the JTAG interface. The FSM controls the test process. It waits for the indication from the host computer to start the generation of the input bitstreams and starts counting the burn-in period, from binary values

stored in BRAMs. After the burn-in period is over, the FSM starts the convertion of the output bitstreams.

V. Performance, Power and Resources Estimates

The framework relies on the FPGA vendor tools to synthesize the test circuit and to produce a configuration bitstream. When this process is launched it instructs the synthesis tool to produce results for resource consumption, and timing and power estimates.

VI. Case Study: KarhunenLove Transform

A. Background

The KLT, also known as Principal Component Analysis (PCA), is an algorithm widely used in Machine Learning to reduce the dimensionality of data sets of many correlated variables, and is formulated as follows. Given a set of N data $x^i \in R^P$, where $i \in [1, N]$ an orthogonal basis described by a matrix Λ with dimensions $P \times K$ can be estimated that projects these data to a lower dimensional space of K dimensions. The projected data points are related to the original data through the formula in (6), written in matrix notation, where $X = [x^1, x^2, ..., x^N]$ and $F = [f^1, f^2, ..., f^N]$, where $f^i \in R^K$ denote the factor coefficients.

$$F = \Lambda^T X. \tag{6}$$

The original data is described from the lower dimensional space via (7):

$$X = \Lambda F + D \tag{7}$$

where D is the error of the approximation. The objective of the transform is to find a matrix Λ that has the Mean-Square Error (MSE) of the data approximation minimized. A standard technique is to evaluate the matrix Λ iteratively as described in steps (8) and (9), where λ_j denotes the j^{th} column of the Λ matrix.

$$\lambda_j = arg\ max\ \mathrm{E}\{(\lambda_j^T X_{j-1})^2\} \tag{8}$$

$$X_j = X - \sum_{k=1}^{j-1} \lambda_k \lambda_k^T X \tag{9}$$

where $X = [x^1 x^2 ... x^N]$, $X_0 = X$, $\|\lambda_j\| = 1$ and $\mathrm{E}\{.\}$ refers to expectation.

B. Hardware Implementations

The KLT algorithm is based on the dot-product operation, which can be implemented using different circuits. Fig. 10 shows the datapath for: a) rolled and b) unrolled architectures of a dot-product based circuit, to implement the datapath of one projection vector from a Z^p to Z^k KLT.

The circuit receives data from the input stream, identified with X. The samples, from the input stream, for each dimension p, are multiplied by the corresponding projection vector λ_{pk}. The output of the multiplier is connected to an adder to do the accumulation. The final result is placed in the output stream, identified with f_k.

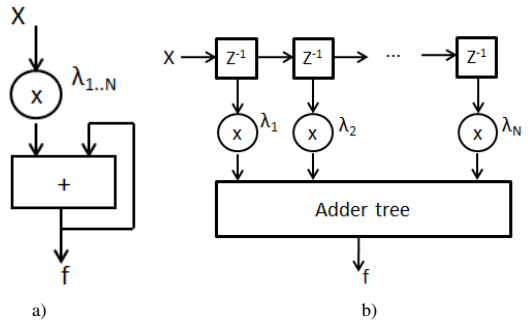

Fig. 10. Schematics of the datapath of a dot-product, for one projection vector of a KLT circuit: a) rolled and b) unrolled architectures.

C. Experimental Results

In this experiment, to compare binary-radix against SC, it was considered the unfolded architecture, which is the one that maximizes the parallelism offered by FPGAs. Considering also 9-bit binary-radix representation, it corresponds to a 512-bit bitstream. However, the length of the bitstream has no influence on the SP.

The implementation of the aforementioned KLT example in a complete parallelized would require to compute 100 streams with 500 multiply-accumulate operations, thus it is necessary to evaluate what is the maximum level of parallelism, and determine if the adoption of the SC is the most favorable approach.

The introduction of the expression for the KLT is generated in Python to explore different possible implementations for the problem via the following script:

```
[frame=single]
inp = 0
outp = []
expr = []
mac_inputs = 2
num_streams = 10

for i in range(0,num_streams):
    for j in range(0,10,mac_inputs):
        for k in range(0,mac_inputs):
     outp.append("in" + str(inp))
     inp = inp + 1
        outp.append("*")
    outp.append("+")
expr.append(outp)
```

Fig. 11 presents the comparison of the resources required by both types of implementation using different numbers of inputs and parallel streams. For the SP implementation,on the left, the inputs are associated with converters, which penalizes the solution by requiring 18% extra resources. The plot on the right shows the results for the same implementation but without considering the conversion of the inputs, leading a SP solution which in the worst case requires 10% of the resources for the binary-radix solution. The results for the synthesis, in terms of resources, of the binary-radix and SC were modeled, using a linear approximation, to reduce the

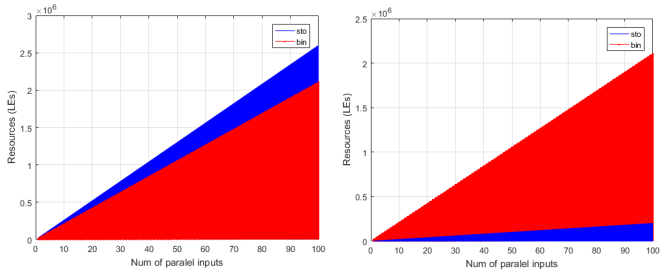

Fig. 11. Comparison of the resources required to implement KLT using binary-radix (red) and SC (blue) for different numbers of input and parallel streams, with radix conversion units (left) and without (right).

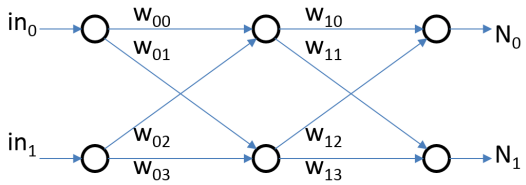

Fig. 12. Representation of a simple neural network.

number of synthesis required to perform the evaluation.

Other applications such as neural networks can also be implemented by the proposed framework. An example of a very simple neural network is illustrated in Figure 12, and would be given by the mathematical expression for each neuron:

$$N_0 = SN((in_0 \times w_{00} + in_1 \times w_{02}) \times \frac{w_{10}}{2} + (in_0 \times w_{01} + in_1 \times w_{03}) \times \frac{w_{12}}{2})$$

$$N_1 = SN((in_0 \times w_{00} + in_1 \times w_{02}) \times \frac{w_{11}}{2} + (in_0 \times w_{01} + in_1 \times w_{03}) \times \frac{w_{13}}{2})$$

In this example, SN is the operator corresponding to a Stochastic Neuron, which implements the activation function, usually tanh(n), and N_x is the output of each neuron. w_x are the weights for each connection of the network.

In the current version of the framework, such expression would generate replicated operations in the VHDL specification, nevertheless the synthesis tool is instructed to eliminated them in the project options before being synthesized.

VII. Discussion

A. Benefits in Datapath Design

The present work could be of benefit for the engineer that is not familiarized with SC and is considering adopting it to use the processing power of a sensor node and improve the reliability of the system. By automatically producing and evaluating traditional datapaths in their SC implementations, not only is possible to compare resources but also evaluate it's performance when operating variation of the operating conditions (power, temperature).

B. Time Overhead

In SC each value is encoded as a bitstream over time instead of a parallel set of bits at one. Therefore, in the case-study presented the latency to produce a valid computation is given by the clock cycles to reach the length of the projection vector plus the time to perform multiplication and go through the adder tree, which is given by equation 10:

$$T = P_{ProjLen} + T_{Mult} + T_{AdderTree} \quad (10)$$

For both cases $P_{ProjLen}$ is the same as the number of inputs. However for the particular case of SC, T_{Mult} requires 2^{WL} clock cycle and so does $T_{AdderTree}$, but delayed by one clock cycle. For typical parallel binary T_{Mult} is 1, considering a fully combinatorial multipliers, and $T_{AdderTree}$ is $log_2(ProjLen)$.

The tradeoff is given in terms of the size of the projection vector and the wordlength (WL) adopted. Thus, the latency SC implementation would only produce results faster if $log_2(ProjSize)$ is greater than 2^{WL}. A parallel binary system is able to produce a result per clock cycle, whereas an SC system requires 2^{WL} clock cycles.

VIII. Conclusions and Future Work

This work introduces a framework to do an early assessment of the cost and performance of SC. It receives a specification for a datapath as a mathematical expression in Python and synthesizes the corresponding SP on an FPGA. Future work involves including support for automatic evaluation of tradeoffs in terms of errors introduced by the representation on the datapath. The framework, along with its source files and a tutorial, are available at https://www.removed-for-blind-review.com, under an open-source license.

References

[1] B. R. Gaines, "Techniques of identification with the stochastic computer," in *in "Proc. International Federation of Automatic Control Symposium on Identification, Prague*, 1967.

[2] M. Suri, O. Bichler, D. Querlioz, G. Palma, E. Vianello, D. Vuillaume, C. Gamrat, and B. DeSalvo, "Cbram devices as binary synapses for low-power stochastic neuromorphic systems: Auditory (cochlea) and visual (retina) cognitive processing applications," in *Electron Devices Meeting (IEDM), 2012 IEEE International*, Dec 2012, pp. 10.3.1–10.3.4.

[3] H. Li, D. Zhang, and S. Foo, "A stochastic digital implementation of a neural network controller for small wind turbine systems," *Power Electronics, IEEE Transactions on*, vol. 21, no. 5, pp. 1502–1507, Sept 2006.

[4] N. L. Zhang and D. Poole, "Exploiting causal independence in bayesian network inference," *Journal of Artificial Intelligence Research*, vol. 5, pp. 301–328, 1996.

[5] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, Sept 2011, pp. 1–4.

[6] F. Zhou, J. Liu, Y. Yu, X. Tian, H. Liu, Y. Hao, S. Zhang, W. Chen, J. Dai, and X. Zheng, "Field-programmable gate array implementation of a probabilistic neural network for motor cortical decoding in rats," *Journal of Neuroscience Methods*, vol. 185, no. 2, pp. 299 – 306, 2010.

[7] J. Zhao, "Stochastic bit stream neural networks," PhD thesis, London University, 1995.

[8] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput bayesian computing machine with reconfigurable hardware," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 73–82.

[9] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *Computers, IEEE Transactions on*, vol. 60, no. 1, pp. 93–105, Jan 2011.

[10] Y.-N. Chang and K. Parhi, "Architectures for digital filters using stochastic computing," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 2697–2701.

[11] N. Saraf, K. Bazargan, D. Lilja, and M. Riedel, "IIR filters using stochastic arithmetic," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

[12] R. P. Duarte, M. Vestias, and H. Neto, "Xtokaxtikox: A stochastic computing-based autonomous cyber-physical system," in *Proceedings of the 1st IEEE International Conference on Rebooting Computing (ICRC)*, 2016.

[13] R. P. Duarte, J. Lobo, J. F. Ferreira, and J. Dias, "Synthesis of bayesian machines on FPGAs using stochastic arithmetic," 2nd International Workshop on Neuromorphic and Brain-Based Computing Systems (NeuComp 2015), associated with DATE2015, Design Automation Test Europe 2015, March 2015.

[14] N. Nedjah and L. de Macedo Mourelle, "Reconfigurable hardware for neural networks: binary versus stochastic," *Neural Computing and Applications*, vol. 16, no. 3, pp. 249–255, May 2007. [Online]. Available: https://doi.org/10.1007/s00521-007-0086-x

[15] J. von Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 43–98.

[16] B. Gaines, "Stochastic computing systems," A. in Information Systems Science, Ed., vol. 2, 1965, p. 37.

[17] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 92:1–92:19, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465794

[18] B. Brown and H. Card, "Stochastic neural computation. i. computational elements," *Computers, IEEE Transactions on*, vol. 50, no. 9, pp. 891–905, Sep 2001.

[19] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan, "Logical computation on stochastic bit streams with linear finite-state machines," *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1474–1486, June 2014.

[20] P. Alfke, "Efficient shift registers, lfsr counters, and long pseudo-random sequence generators," July 1996.

[21] D. M. KASPRZYK, C. G. DRURY, and W. F. BIALAS, "Human behaviour and performance in calculator use with algebraic and reverse polish notation," *Ergonomics*, vol. 22, no. 9, pp. 1011–1019, 1979. [Online]. Available: https://doi.org/10.1080/00140137908924675

[22] S. Agate and C. Drury, "Electronic calculators: which notation is the better?" *Applied Ergonomics*, vol. 11, no. 1, pp. 2 – 6, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0003687080901143

[23] R. P. Duarte, M. Vestias, and H. Neto, "Enhancing stochastic computations via process variation," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Aug 2015, pp. 519–522.

[24] A. Martin and M. Nystrom, "Asynchronous techniques for system-on-chip design," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, June 2006.