All-in-Memory Stochastic Computing using ReRAM

João Paulo C. de Lima[1,2], Mehran Shoushtari Moghadam[3],
Sercan Aygun[4], Jeronimo Castrillon[1,2,5], M. Hassan Najafi[3], and Asif Ali Khan[1]
[1]Chair for Compiler Construction, Dresden University of Technology, Dresden, Germany
[2]Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), Dresden, Germany
[3]Electrical, Computer, and Systems Engineering Department, Case Western Reserve University, Cleveland, OH, USA
[4]School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA, USA
[5]Barkhausen Institut, Dresden, Germany
Corresponding authors: {joao.lima, asif_ali.khan}@tu-dresden.de

***Abstract*—As the demand for efficient, low-power computing in embedded and edge devices grows, traditional computing methods are becoming less effective for handling complex tasks. Stochastic computing (SC) offers a promising alternative by approximating complex arithmetic operations, such as addition and multiplication, using simple bitwise operations, like `majority` or `AND`, on random bit-streams. While SC operations are inherently fault-tolerant, their accuracy largely depends on the length and quality of the stochastic bit-streams (SBS). These bit-streams are typically generated by CMOS-based stochastic bit-stream generators that consume over 80% of the SC system's power and area. Current SC solutions focus on optimizing the logic gates but often neglect the high cost of moving the bit-streams between memory and processor. This work leverages the physics of emerging ReRAM devices to implement the entire SC flow in place: ❶ generating low-cost true random numbers and SBSs, ❷ conducting SC operations, and ❸ converting SBSs back to binary. Considering the low reliability of ReRAM cells, we demonstrate how SC's robustness to errors copes with ReRAM's variability. Our evaluation shows significant improvements in throughput (1.39×, 2.16×) and energy consumption (1.15×, 2.8×) over state-of-the-art (CMOS- and ReRAM-based) solutions, respectively, with an average image quality drop of 5% across multiple SBS lengths and image processing tasks.**

I. Introduction

The growing prevalence of embedded and edge devices has driven the demand for low-cost but efficient computing solutions. These devices, which often run complex applications like computer vision tasks in real-world environments, are constrained by computational resources and power budget, making traditional computing methods less effective. Stochastic computing (SC) and non-von Neumann paradigms have emerged as promising alternatives, offering trade-offs in computational density, energy efficiency, and error tolerance [1, 2, 3, 4].

In SC, data is represented by random bit-streams, where a value $x \in [0, 1]$ is encoded by the probability (P_x) of a '1' appearing in the stream. For example, the bit-stream '10101' represents the value $\frac{3}{5}$, where 5 is the bit-stream length (N). This unconventional representation enables complex computations like *multiplication* and *addition* to be approximated with simple logic operations such as `AND` and `majority`, respectively, reducing area and power consumption substantially without taking a high toll on computational accuracy. Additionally, since all bits in the bit-streams carry equal weights – no *most*- or *least*-significant bits – SC is naturally tolerant to noise, including bit flips and inaccuracies in the input data and computations. This makes SC particularly advantageous for a range of applications, including image processing [5], signal processing [6], and neural networks [7].

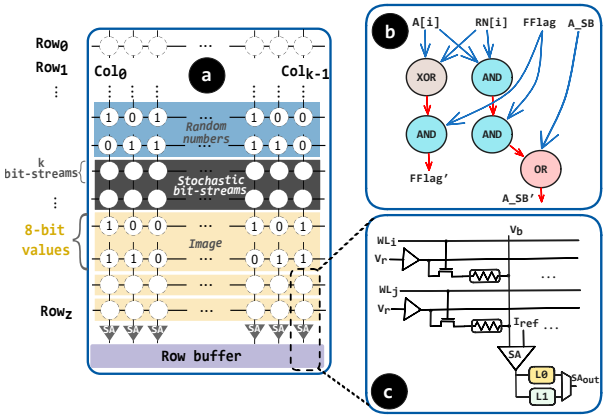

Fig. 1. A high-level overview of our proposed in-memory SC solution: (a) ReRAM array, (b) Greater-than operation using basic logic gates, (c) Write latches in the peripheral circuitry.

Stochastic bit-streams (SBSs) are conventionally generated using a CMOS-based structure, called stochastic bit-stream generator, built from a *pseudo random* (or more recently, *quasi-random)* [8] *number generator*, and a *binary comparator*. The accuracy and cost efficiency of SC systems highly depend on this bit-stream generation unit. Presently, CMOS-based bit-stream generation consumes up to 80% of the system's total hardware cost and energy consumption [4, 9]. Additionally, SC implementations on classic von Neumann systems require extensive movement of bit-streams *from/to* memory, which is often overlooked in evaluations but can easily offset the benefits of the simpler SC operations. This has motivated significant research into non-von Neumann computing for SC, using different memory technologies [10, 11, 12, 13, 14, 15, 16, 17].

Most compute-in-memory (CIM) designs for SC utilize ReRAM, a nonvolatile memory technology that stores data in the resistance state of the devices [18]. ReRAM offers DRAM-comparable read latency, higher density, and significantly reduced read energy consumption but incurs expensive write operations, limited write endurance, and suffers from non-linearities [19]. SC in ReRAM benefits from ReRAM's high density for efficient storage and in-place processing of long bit-streams, while SC's inherent robustness helps mitigate the effects of ReRAM's non-linearities. In existing ReRAM-based CIM designs for SC, conventional CMOS-based random

number generators (RNGs) are used for bit-stream generation, while ReRAM arrays handled in-place logic operations [10, 11, 12]. This increases parallelism and reduces data movement overhead, though the high cost of random bit-stream and SBSs generation remains a bottleneck. Techniques like leveraging ReRAM's inherent write noise [13] and employing DRAM-based lookup tables [20] have been explored to improve the performance of random bit-stream generation. However, SBSs generation continues to face challenges, including the high energy cost of ReRAM write operations and scalability issues with DRAM-based methods.

To address these challenges, we propose a CIM accelerator that implements *all steps* of SC using ReRAM. We decouple RNG from SBS generation, allowing compatibility with any RNG type, including general-purpose true RNGs (TRNGs) based on ReRAM [21]. This approach ensures (1) *accurate SBS generation* with target probabilities and (2) *correlation control*, despite ReRAM cell noise. We perform in-memory logic and comparison operations to convert true random numbers into SBS (❶), conduct SC operations (❷), and convert the resulting values back into a binary representation (❸). Concretely, this work makes the following novel contributions:

- We propose a ReRAM-based accelerator for SC that implements all steps in place, including those often overlooked by current SC designs.
- We develop a novel in-memory method for converting true random binary sequences (50% *ones*) into SBSs with desired probabilities. To the best of our knowledge, this is the first such method reported in the literature.
- Our SBS generation approach is RNG-agnostic, leveraging in-memory comparison to produce SBS even under substantial CIM failures caused by ReRAM variability.
- For SC operations typically implemented with `MUX`s, we propose novel alternatives that are CIM-friendly and achieve comparable accuracy.

Compared to the state-of-the-art CMOS-based solutions, the proposed design, while requiring minimal changes to the memory periphery, on average, reduces energy consumption by $1.15\times$ and improves throughput by $1.39\times$ across multiple image processing applications. Our design is also more robust than traditional arithmetic for CIM, with only a 5% average quality drop in the presence of faults compared to a 47% drop. It eliminates the need for protection schemes on unreliable ReRAM devices and provides better correlation control than previous in-memory SC designs.

II. Background and Related Work

A. ReRAM-based Computing

Resistive RAM (ReRAM) is a type of nonvolatile memory where each cell, typically a metal oxide, and being referred to as a memristor, is programmed to different resistance states using an electric voltage [22]. Data is represented through resistance levels, such as a high resistance state (HRS) for '0' and a low resistance state (LRS) for '1'. Organized in a conventional memories like 2D grid of rows (wordlines, WL) and columns (bitlines, BL), ReRAM promises DRAM-comparable read performance but has costly write operations that impact both energy consumption and the write endurance [19]. For CIM using ReRAM, the 1T1R (one transistor, one resistor) crossbars are extensively used in machine learning and other domains to perform analog matrix-vector multiplication in constant time [3]. Similarly, stateful and non-stateful logic techniques, such as MAGIC [23] and scouting logic (SL) [24], respectively, have been demonstrated for implementing logic operations using ReRAM. ReRAM cells have inherent stochasticity and noise; which have also been investigated to generate true random numbers [21, 25].

B. Stochastic Computing (SC)

SC is an alternate computing approach offering simple execution of complex arithmetic operations and high tolerance to soft errors. Unlike traditional binary radix, SC operates on random bit-streams of '0's and '1's, with no bit-significance. SC systems include three primary components: ❶ Bit-stream generator that converts data from traditional binary to stochastic bit-stream, ❷ computation logic that performs bit-wise operations on the bit-streams, and ❸ bit-stream to binary converter to convert data back to binary format.

Bit-stream Generation: The accuracy of SC operations highly depends on the quality of bit-streams. To convert a binary number X to an SBS of size N, an RNG is used to generate N random numbers. The SBS is generated by comparing each of these N random numbers with X. A '1' is produced if the random number is less than X, and a '0' is produced otherwise. Conventionally, SC systems employ CMOS-based pseudo-RNGs (PRNGs) such as linear-feedback shift registers (LFSRs) to generate the needed random numbers [26]. However, this can lead to suboptimal performance as very long SBSs are needed to achieve acceptable accuracy. Recent works leverage quasi-RNGs (QRNGs) for better accuracy [27] but at the cost of a higher area and power [8, 9]. The high cost of CMOS-based SBS generation offsets the gains made with simple computation circuits.

SC Operations: Basic arithmetic operations – multiplication, addition, subtraction, and division – are implemented in SC using minimal components: an `AND` gate, a multiplexer (`MUX`) unit, an `XOR` gate, and a `MUX+D`-flip-flop, respectively (Fig. 2) [4, 28]. For N-bit-long SBSs, the logic operations are often performed serially, producing the output SBS in N clock cycles. Parallel execution of the operations is also feasible by trading off time with space. This approach is particularly attractive for SC with CIM as it enables fast and independent execution of all bit-wise operations. For correct functionality of the aforementioned operations, the input bit-streams must provide the desired correlation level, i.e., *uncorrelated* for the multiplication and addition, and *correlated* for the subtraction and division operations. The independence (i.e., uncorrelation) requirement is typically satisfied by using different RNGs while the desired amount of correlation is guaranteed by using shared RNGs when generating SBSs.

C. State-of-the-art In-memory SC Solutions

Existing CIM-SC designs are mostly hybrid, i.e., either memristive arrays are used to generate random numbers and CMOS logic to perform computations, or vice versa. For instance, Knag et al. [10] proposed generating SBSs using memristors and off-memory computations using CMOS

stochastic circuits. A similar design is proposed in [16] that exploits the switching stochasticity of probabilistic Conductive Bridging RAM (CBRAM) devices to generate SBSs in memory efficiently. The generated bit-streams are then used to optimize deep learning parameters using a hybrid CMOS-memristor stochastic processor. ReRAM-based SBS generation has also been proposed for SC. However, these solutions primarily use the probabilistic switching, i.e., the write operation in ReRAM [29]. Riahi Alam et al. [17] developed an accurate method for in-memory SC multiplication by performing a deterministic binary-to-SBSs conversion. Sun et al. [15] employed unary coding, using multi-level memristor cells, for weight representation in a ReRAM-based neural network accelerator. The most relevant work to our design is SCRIMP [13], which also proposes SBS generation and computation using ReRAM. However, similar to other prior works, it generates SBSs using the stochasticity in the write operation, which is not only extremely slow but also affects write endurance. Existing methods can generate SBSs with target probabilities but *lack correlation control*, which limits their applicability for SC operations that require correlated inputs. We propose a novel ReRAM-based solution to convert true random binary sequences (50% *ones*) into SBSs with desired probabilities using bulk-bitwise operations.

III. Proposed In-ReRAM Stochastic Computing

We exploit the physical properties of ReRAM arrays to implement all stages of the SC flow (see Sec. II-B). In practice, we use multiple arrays to parallelize and pipeline the different stages. However, for simplicity, we show a single array in Fig. 1ⓐ, consisting of dedicated rows to store input binary data (yellow), random numbers (blue), and in-memory generated stochastic bit-streams (grey). In the following, we explain the in-memory implementation of these different operations and the data flow in the different stages of the SC flow.

A. Stochastic Number Generation (SNG)

The switching stochasticity of ReRAM devices has been exploited to generate true random numbers (see Sec. II-A). We build upon this prior work and consider TRNG as a single-step operation that stores random sequences directly in ReRAM arrays. To generate SBSs from these random sequences (referred to as in-memory SNG or IMSNG), we compare them with n-bit input binary operands using in-memory bitwise operations (see Sec. II-B). Concretely, for comparing two binary numbers A and RN in memory, starting from the most significant bit (MSB) to the least significant bit (LSB), we perform bitwise comparison and stop at the first non-equal bit position, i.e., where $A_i \oplus RN_i$ is 1. This is achieved by implementing the `greater-than` operation, i.e., $A_i > RN_i$, using in-ReRAM bitwise `XOR` and `AND` operation, together with a flag bit (`FFlag`), as illustrated in the Boolean network in Fig. 1ⓑ. The result of this comparison is a row A_{SB} representing the SBS of A. This network is converted into data structures like `XOR-AND`-Inverter graph (XAG) for manipulation and optimization using logic synthesis tools [30].

Considering the SL approach (see Sec. II-A), implementing this network requires $5n$ operations, as each logic gate requires one sensing step (cf. [24]). This also requires writing the intermediate signals (e.g., that of `XOR`) back to the ReRAM array at least 4 times (red arrows in Fig. 1ⓑ). These intermediate writes can be avoided by forwarding the output of one operation directly as input voltage to the bitline for the next operation. This requires the periphery to incorporate a simple feedback mechanism that converts the latched signal to adjust the bitline voltage (V_b) (see Fig. 1ⓒ) to mimic the voltage drop across the ReRAM that would have been written. This approach (referred to as *IMSNG-naive*) reduces the number of ReRAM writes to $2n$.

IMSNG-opt: As an alternative approach, we are exploiting the latches in the ReRAM peripheral circuitry, shown in Fig. 1ⓒ (L0 and L1), to minimize the write overhead. ReRAM and other nonvolatile memories, typically employ double latches and a write driver to conduct differential writes [31]. For each write driver, a latch stores the data to be written to the cells and a second latch stores whether the cell should be modified (in case the new data is different than the already stored data). We leverage this mechanism to directly implement the `AND` operations involving `FFlag` as a predicated sensing, hence eliminating the need to write the intermediate result to the memory cells. This approach, which uses existing latches and drivers to eliminate write operations deriving from intermediate results, is referred to as *IMSNG-opt*.

Table I compares the mean squared error (MSE) of our IMSNG to a software-based (SW), PRNG-based, and QRNG-based SBS generator. SW uses MATLAB's RNG (`rand`) for bit-stream generation. PRNG and QRNG use 8-bit LFSR and Sobol sequence generators. The comparison in IMSNG is between the target input and N random binary sequences

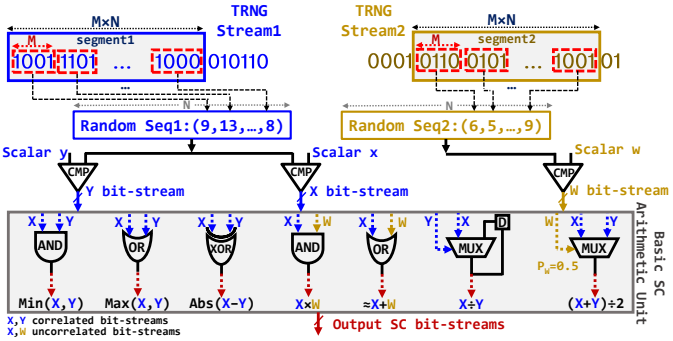

Fig. 2. In-ReRAM SBS generation and SC arithmetic operations. CORDIV [28] is considered for SC division with $X \leqslant Y$. Addition with `OR`; the inputs are in the $[0, 0.5]$ interval to not exceed 1.0 in the output.

TABLE I
MSE(%) COMPARISON OF GENERATING SBSS UTILIZING DIFFERENT RNG SOURCES (M: BLOCK SIZE, N: BIT-STREAM LENGTH).

RNG Source		Bit-stream length (N)				
	Segment size (M)	32	64	128	256	512
IMSNG [21]	5	0.567	0.321	0.189	0.134	0.103
	6	0.562	0.302	0.177	0.114	0.084
	7	0.534	0.279	0.157	0.095	0.064
	8	0.557	0.300	0.177	0.107	0.074
	9	0.520	0.282	0.159	0.090	0.060
Software - MATLAB		0.529	0.264	0.131	0.065	0.032
PRNG (8-Bit LFSR)		1.069	0.554	0.288	0.137	0.071
QRNG (8-Bit Sobol)		0.033	0.008	0.002	5.05×10^{-4}	1.25×10^{-4}

For PRNG, a Maximal length LFSR with polynomial $x^8 + x^5 + x^3 + 1$ is used.

TABLE II
MSE (%) COMPARISON OF SC ARITHMETIC OPERATIONS UTILIZING DIFFERENT RNGS WITH $M = 8$.

SC Operations	IMSNG [21]					Software - MATLAB					PRNG (LFSR)					QRNG (Sobol)				
	N:32	64	128	256	512	N:32	64	128	256	512	N:32	64	128	256	512	N:32	64	128	256	512
Multiplication	0.473	0.255	0.147	0.091	0.061	0.444	0.219	0.108	0.054	0.027	0.851	0.476	0.221	0.093	0.060	0.058	0.017	0.005	0.001	2.9×10^{-4}
Scaled Addition	0.690	0.356	0.193	0.109	0.062	0.648	0.328	0.159	0.082	0.041	1.117	0.607	0.289	0.157	0.065	0.102	0.013	0.003	0.002	2.1×10^{-4}
Approx. Addition	1.548	1.186	1.024	0.927	0.886	1.379	1.055	0.897	0.789	0.751	2.654	1.702	1.180	0.914	0.842	0.463	0.586	0.670	0.662	0.689
Abs. Subtraction	0.641	0.354	0.136	0.144	0.107	0.514	0.263	0.129	0.064	0.034	0.559	0.281	0.136	0.058	0.026	0.016	0.004	0.001	2.5×10^{-4}	6.5×10^{-5}
Division	1.614	0.895	0.518	0.295	0.187	1.454	0.789	0.392	0.196	0.106	2.760	2.140	1.688	1.630	1.477	0.251	0.164	0.129	0.126	0.128
Minimum	0.572	0.307	0.177	0.106	0.064	0.514	0.265	0.130	0.066	0.032	1.493	0.811	0.394	0.199	0.085	0.033	0.008	0.002	5.1×10^{-4}	1.3×10^{-4}
Maximum	0.572	0.302	0.186	0.117	0.077	0.543	0.259	0.132	0.064	0.033	0.481	0.263	0.123	0.073	0.027	0.032	0.008	0.002	5.0×10^{-4}	1.3×10^{-4}

(generated in-memory) of M bits, where $M=5, 6, ..., 9$ (see Fig. 2). The data is based on 1,000,000 samples extracted from a uniform distribution. The results highlight that IMSNG, despite its random fluctuations and true randomness, provides comparable accuracy to other methods. Notably, for bit-stream lengths of 32, 64, and greater than 128, MSEs of approximately 0.5%, 0.3%, and 0.1% are measured, respectively.

B. Stochastic Circuits using Scouting Logic (SL)

SL implements boolean logic using ReRAM read operations with a modified sense amplifier (SA) [24, 32]. During a logic operation, two or more rows are simultaneously activated and the resulting current through the cells in each bitline is compared with a reference current I_{ref} by the SA, whose output is the desired result of the Boolean operation (see Fig. 1c). All basic logic operations such as (N)AND, (N)OR, X(N)OR, and NOT, are realized in a single cycle [33].

Bulk bitwise in-memory operations are performed on large vectors. When operating on traditional binary-radix numbers, we can only exploit the single instruction, multiple data (SIMD) parallelism, since these algorithms are sequential by nature due to carry propagation. In contrast, SC handles basic arithmetic operations (addition, subtraction, multiplication, and division) using simple, low-cost logic units such as AND, NOT, XOR, MUXs, and flip-flops. Each bit is computed independently, allowing for in-memory SC to exploit bulk-bitwise logic and massive word-level parallelism, significantly reducing latency for basic arithmetic operations. In the following, we explain how these primary arithmetic operations are implemented using bulk bitwise logic schemes.

Multiplication is implemented by performing bitwise AND on two independent bit-streams, representing probabilities p and q. The probability of observing a '1' in the output stream equals $p \wedge q$, which aligns well with the principles of SL with the time complexity of $\mathcal{O}(1)$. This contrasts with conventional bulk-bitwise implementations of binary radix multiplication, which exhibit a time complexity of $\mathcal{O}(n^2)$, where n represents the number of bits. This complexity arises from the iterative nature of traditional methods, which rely on bit shifts and additions to compute the product.

Scaled addition is implemented in SC using a 2-to-1 MUX. In SL, a 2-to-1 MUX can be approximated by a CIM-friendly 3-input majority gate (MAJ) [34] that can be computed in a single cycle. For in-place MAJ, a reference current corresponding to the majority of the inputs is required. For instance, to conduct a 3-input MAJ gate operation, we use the same reference current used for the 2-input AND gate, as this detects when at least two out of three inputs are high. The time complexity of our MAJ-based addition is $\mathcal{O}(1)$, which is a significant improvement over both traditional ripple-carry additions in the binary domain and MUX-based addition in existing SC that has a time complexity of $\mathcal{O}(N)$.

Division in prior SC works is implemented using CMOS-based flip-flops and MUXs, and *correlated* bit-streams to approximate $y = \frac{x_1}{x_2}$. In SL, the JK flip-flop's truth table can be implemented using the existing latch-based circuitry (Fig. 1c). The intermediate values from the flip-flop are stored in the existing latch (write driver) and forwarded to the bitline as voltage inputs, eliminating the intermediate write operations and improving energy efficiency and write endurance. This method has a time complexity of $\mathcal{O}(N)$, while existing CIM division methods on integer data [35] require $\mathcal{O}(n^2)$ write cycles.

For **other operations**, such as approximate addition, absolute subtraction, minimum, and maximum, we employ bulk-bitwise operations like OR, XOR, AND, and OR, respectively. In stochastic logic, the reference current for the OR operation is set to detect when at least one of the operands is high.

Table II compares the accuracy of these stochastic operations across different SNG sources. Compared to PRNG, QRNG, and SW-based SNG methods, our IMSNG approach achieves comparable accuracy, and even in some cases (e.g., compared to PRNG-based division) a lower MSE. Still, the important advantage of our method is that it is executed completely in memory, eliminating the overheads of transferring SBSs between memory and processing circuits.

C. Stochastic to Binary Conversion

As a last step in the SC flow, the output of the SC operation needs to be converted back to binary. Existing methods use CMOS counters for stochastic-to-binary (S-to-B) conversion, sequentially counting the '1's in the output bit-stream. In contrast, our approach achieves the count in a single step by using bitline current accumulation. The output bit-stream is applied as input voltages (V_r) to a designated reference column in which all cells have been pre-programmed to low resistance states (see Fig. 1c). The total current through the bitline, representing the population count of the bit-stream, is then measured and digitized using analog-to-digital converters (ADCs).

IV. Evaluation Results and Analysis

The custom SA and the proposed hardware modifications, including the feedback mechanism and latch-based optimizations, were validated with SPICE simulations. Energy consumption and latency metrics of the in-memory design were

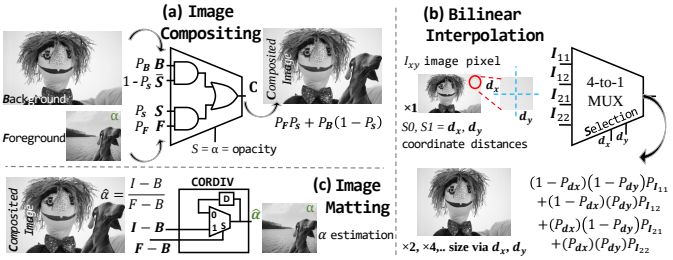

Fig. 3. SC image processing applications: (a) *Image Compositing*, merging background and foreground images with α channel. (b) *Bilinear Interpolation*, up-scaling input images. (c) *Image Matting*, parsing α channel for the foreground object to separate the background.

extracted from [24] and integrated into NVMain [36]. For S-to-B conversion, we consider a single 8-bit ADC from [37] per mat. SL output is prone to failures when deciding on the bitwise operation output due to the intrinsic variability of ReRAM, as described in [38]. We conduct simulations with the VCM-based ReRAM model [39] to determine the distribution of LRS and HRS that leads to the probability of obtaining incorrect outputs in CIM operation. For NVMain simulation, we generate traces for the SBS generation, the SC circuits in Table II, and image processing applications. The derived failure rates are used to simulate fault injections and we report the average results of 1,000 runs. For the CMOS-based SC circuits, we synthesized the designs using the Synopsys Design Compiler with the $45nm$ gate library.

A. Applications

To evaluate our CIM-SC design, we use image processing applications, including (a) image compositing, (b) bilinear interpolation, and (c) image matting.

Image Compositing. In traditional computer vision, two images — background (B) and foreground (F) — are merged using a linear formula that incorporates pixel information from both images, along with an additional channel, α, which represents the opacity of the foreground. This process allows merging two scenes, typically when one of them has a green background. The α-channel defines the object of interest, its refined edges, and the remaining background area in the final composite image. The compositing formula used, is $C = F \times \alpha + B \times (1-\alpha)$, which corresponds to the `MUX` in the SC domain. If the bit-streams of B, F, and α are provided to the first, second, and selection inputs of the `MUX`, respectively, the output becomes the composite image, C. This process is illustrated in Fig. 3(a).

Bilinear Interpolation. Image up-scaling can be achieved through bilinear interpolation involving two linear interpolations along the `x`- and `y`-axes. For any 4-pixel neighboring group, I_{11}, I_{12}, I_{21}, and I_{22}, the input image I can be used to create a larger image by estimating new pixel values between them. The intensity of the new pixel, $I(\mathrm{x},\mathrm{y})$, is calculated using the intensities of the four neighboring pixels and their relative distance (dx, dy) from the new point. The conventional formula is $I(\mathrm{x},\mathrm{y}) = (1-dx)(1-dy)I_{11} + (1-dx)(dy)I_{12} + (dx)(1-dy)I_{21} + (dx)(dy)I_{22}$. This corresponds to a 4-to-1 `MUX` in the SC domain. Here, the bit-streams of the four neighboring pixels serve as inputs to the `MUX`, while the dx and dy bit-streams are fed into the selection ports. The output bit-stream provides the intensity estimate for the new pixel in the up-scaled image. This application is shown in Fig. 3(b).

Image Matting. Image compositing can be reversed to separate the background and foreground images by estimating the α-channel. By rearranging the composite image formula and solving for α, the estimated alpha $\hat{\alpha} = \frac{I-B}{F-B}$ reveals the foreground object and helps refine the edges for a more natural composite image. This estimation is often repeated multiple times with different B and F information, especially when foreground details are missing. Given that this formula involves division operation [40], this work employs the CORDIV design [28], as shown in Fig. 3(c).

B. Performance and Energy Comparison

Table III compares the hardware costs of CMOS-based (✢) and ReRAM-based (✦) SC arithmetic designs. The analysis focuses on the breakdown of SC logic, excluding memory movement costs for CMOS-based designs that further increase their total latency and energy consumption. CMOS implementations employ LFSR and Sobol generators as SNG. Regarding latency, CMOS-based designs are significantly slower than ReRAM-based designs due to the sequential processing of bit-streams. Our ReRAM-based design reduces latency by 38%, compared to CMOS, due to the row-parallel execution of most operations. For the division operation, due to the sequential nature of `flip-flops`, CIM-based CORDIV exhibits higher latency, but it is offset by increased throughput enabled by SIMD parallelism. CORDIV is still compatible with the ADC-based S-to-B conversion, where the SBS is written as resistance values in a column, enabling the ADC to sense the bitline current that represents the number of '1's in the SBS. Other ReRAM-based implementations use the SBS as voltage input, thus requiring a reference column.

If considered in isolation, our ReRAM-based design consumes more energy than CMOS mainly due to multiple read operations ($5N$) for SBS generation and storing SBSs (at least one write operation). In this section, we only report results for the *IMSNG-opt* configuration. For comparison, *IMSNG-naive* requires 395.4 ns and consumes 10.23 nJ per conversion, whereas *IMSNG-opt* completes the same process in 78.2 ns while using only 3.42 nJ.

TABLE III
HARDWARE COST EVALUATION FOR *CMOS-based* (✢) AND *ReRAM-based* (✦) SC TECHNOLOGIES.

CMOS-based Design (✢)			**Total Latency** ❇ (ns)	**Total Energy** (nJ)
Binary→SC ❶	**SC arithmetic operations ❷**	**SC→Binary ❸**		
LFSR + Comparator	Multiplication	$log_2 N$-bit counter	122.88	0.23
	Addition		130.56	0.26
	Subtraction		133.12	0.16
	Division		133.12	0.18
Sobol + Comparator	Multiplication		125.44	0.30
	Addition		130.56	0.30
	Subtraction		133.12	0.12
	Division		130.56	0.14
ReRAM-based Design (✦)				
IMSNG-opt	Multiplication	8-bit ADC [37]	80.8	3.50
	Addition		80.8	3.50
	Subtraction		81.6	3.51
	Division		12544.0	4.48

❇: Total latency=Critical Path Latency × N. Bit-stream Length (N) is 256.

TABLE IV
SSIM (%)/PSNR (dB) COMPARISON FREE OF (✗) OR UNDER (✓) CIM FAULTS ACROSS DIFFERENT BIT-STREAM LENGTHS.

Design	Image Compositing		Bilinear Interpolation		Image Matting	
	✗	✓	✗	✓	✗	✓
✧ [35]	99.9/91.8	82.9/42.4	95.6/39.0	64.4/37.6	99.9/50.3	4.8/-18.2
✦ 32	99.9/23.4	99.9/22.2	82.0/28.5	79.4/28.6	95.3/31.5	88.2/30.0
✦ 64	99.9/26.7	99.9/25.6	87.7/29.5	86.5/29.7	98.7/37.8	93.0/36.7
✦ 128	99.9/28.2	99.9/27.6	91.4/30.2	90.0/29.7	99.4/41.7	94.6/38.5
✦ 256	99.9/32.3	99.9/30.9	93.0/31.1	92.9/31.5	99.7/44.9	96.7/44.5

PSNR: Peak Signal-to-Noise Ratio, SSIM: Structural Similarity. ↑ is better. For the ReRAM-based SC design(✦), different bit-stream lengths (N=32,64,128,256) are used.

Fig. 4 and Fig. 5 compare the SC designs (✢ and ✦) to the binary CIM (✧) (also considering memory transfers). Regarding energy savings, on average, our design reduces energy by 2.8× and 1.15×, compared to the binary CIM and and CMOS designs, respectively.

Only for larger resolutions (N=256), our solution performs poorly compared to CMOS, but these resolutions also considerably increases latency of CMOS-based designs and are typically avoided. In terms of throughput, our design achieves, on average, 2.16× and 1.39× higher throughput compared to the binary CIM and CMOS designs respectively.

The off-chip communication in CMOS-based designs – specifically loading images and storing outputs to the same ReRAM setup – significantly increases total energy consumption. ReRAM-based designs outperform CMOS-based ones for smaller SBSs (32 and 64) across all applications and for all SBS lengths in Bilinear Interpolation. However, for larger SBSs, the cost of writing SBSs outweighs CIM's benefits to the extent that transferring data to the SC logic is more efficient. Nonetheless, implementing SC on general-purpose CIM hardware is a key advantage, as it requires no additional components beyond those common in other CIM designs.

C. Reliability through Stochastic Computing

In digital CIM, a fault is a bit flip, where results invert from the expected value. Table IV presents the quality of our selected applications *with* CIM faults (realistic scenario, ✓) and *without* CIM faults (ideal scenario, ✗). For image compositing and bilinear interpolation, we compare the outputs of *ReRAM-based SC* (✦) and *Binary CIM* (✧) [35] against the SW implementation. For image matting, we compare the blended images obtained using the original α (I in Fig. 3(c)) and the estimated $\hat{\alpha}$. Our design shows an average quality drop of 5% under realistic scenarios. Traditional arithmetic [35] exhibits 47% drop in quality in the presence of faults (as high as 95.2%, in Image Matting), as errors at higher bit positions can lead to more severe and widespread inaccuracies. Among these, image matting relying on integer division is particularly vulnerable to faults, often rendering unacceptable outputs.

Comparing different SBS resolutions shows that some algorithms require only smaller bit-streams (no noticeable drop in accuracy). This aligns well with the performance and energy-efficiency trends presented in Fig. 4, where energy savings increase as bit-stream size decreases. The impact of faults is highly algorithm-dependent. For instance, image compositing and bilinear interpolation, both relying on the MAJ operation,

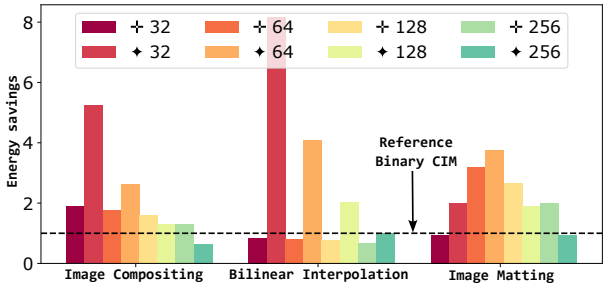

Fig. 4. Normalized energy savings for CMOS (✢) and ReRAM (✦) designs.

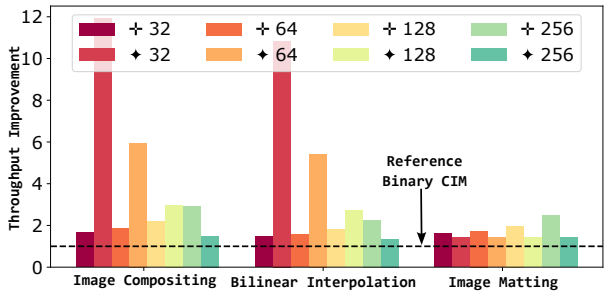

Fig. 5. Normalized throughput for CMOS (✢) and ReRAM (✦) designs.

are more tolerant than CORDIV to CIM faults. This highlights the strength of SC on unreliable CIM, offering more robustness to CIM faults and not requiring dedicated fault protection hardware, making it both energy-efficient and accurate.

Memory protection schemes exist but are costly and traditional error correction codes cannot protect CIM operations (AND, OR, and MAJ). A recent work [41] proposes a parity-based fault-tolerance scheme for NVM-CIM, which adds a significant area overhead for parity storage and syndrome generation, as well as added latency due to critical path dependencies. SC is fault-tolerant, suitable for unreliable devices, and does not add extra overhead to protect CIM operations.

V. Conclusion

We presented a ReRAM-based, in-memory implementation of stochastic computing (SC). Leveraging existing in-ReRAM true RNGs, we produce SBSs, perform stochastic operations, and convert them back to binary – all within the memory array. Compared to the state-of-the-art CMOS-based SC solution, our results across multiple image processing kernels show similar accuracy, 1.39× higher throughput and 1.15× less energy consumption, all without requiring specialized logic for SC.

Acknowledgments

This work is partially funded by National Science Foundation (NSF) under grant No. 2019511, 2339701, the German Research Council (DFG) through the HetCIM project (502388442), the AI competence center ScaDS.AI Dresden/Leipzig (01IS18026A-D), the CRC/TRR 404-Active 3D, and generous gifts from NVIDIA. We thank Stefan Wiefels from Forschungszentrum Jülich for our insightful discussions on the earlier version of this paper.

REFERENCES

[1] C. F. Frasser and et al., "Fully parallel stochastic computing hardware implementation of convolutional neural networks for edge computing applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 12, pp. 10 408–10 418, 2022.

[2] M. W. Daniels, A. Madhavan, P. Talatchian, A. Mizrahi, and M. D. Stiles, "Energy-efficient stochastic computing with superparamagnetic tunnel junctions," *Physical rev. appl.*, vol. 13, no. 3, p. 034016, 2020.

[3] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[4] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *IEEE TCAD*, vol. 37, no. 8, 2018.

[5] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. D. Riedel, "Computation on stochastic bit streams digital image processing case studies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 3, pp. 449–462, 2013.

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

[7] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," *ACM Sigplan Notices*, vol. 52, no. 4, pp. 405–418, 2017.

[8] S. Liu and J. Han, "Toward energy-efficient stochastic circuits using parallel sobol sequences," *IEEE TVLSI*, vol. 26, no. 7, pp. 1326–1339, 2018.

[9] M. H. Najafi, D. Jenson, D. J. Lilja, and M. D. Riedel, "Performing stochastic computation deterministically," *IEEE TVLSI*, vol. 27, no. 12, pp. 2925–2938, 2019.

[10] P. Knag, W. Lu, and Z. Zhang, "A Native Stochastic Computing Architecture Enabled by Memristors," *IEEE Trans. on Nanotechnology*, vol. 13, no. 2, March 2014.

[11] S. Gaba, P. Knag, Z. Zhang, and W. Lu, "Memristive devices for stochastic computing," in *2014 IEEE Intern. Symp. on Circuits and Systems (ISCAS)*, June 2014.

[12] S. Raj, D. Chakraborty, and S. K. Jha, "In-memory flow-based stochastic computing on memristor crossbars using bit-vector stochastic streams," in *2017 IEEE 17th Intern. Conf. on Nanotechnology (IEEE-NANO)*, July 2017, pp. 855–860.

[13] S. Gupta, M. Imani, J. Sim, A. Huang, F. Wu, M. H. Najafi, and T. Rosing, "Scrimp: A general stochastic computing architecture using reram in-memory processing," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1598–1601.

[14] C. Ma, Y. Sun, W. Qian, Z. Meng, R. Yang, and L. Jiang, "Go unary: A novel synapse coding and mapping scheme for reliable reram-based neuromorphic computing," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1432–1437.

[15] Y. Sun, C. Ma, Z. Li, Z. Li, Z. Li, Y. Zhao, J. Jiang, W. Qian, W. Qian, R. Yang, Z. He, L. Jiang, and L. Jiang, "Unary coding and variation-aware optimal mapping scheme for reliable reram-based neuromorphic computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[16] C. Lammie, J. K. Eshraghian, W. D. Lu, and M. R. Azghadi, "Memristive stochastic computing for deep learning parameter optimization," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 5, pp. 1650–1654, 2021.

[17] M. Riahi Alam, M. Hassan Najafi, and N. TaheriNejad, "Exact stochastic computing multiplication in memristive memory," *IEEE Design Test*, pp. 1–6, 2021.

[18] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[19] V. Gupta, S. Kapur, S. Saurabh, and A. Grover, "Resistive random access memory: a review of device challenges," *IETE Technical Review*, vol. 37, no. 4, pp. 377–390, 2020.

[20] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Scope: A stochastic computing engine for dram-based in-situ accelerator," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 696–709.

[21] K. S. Woo, Y. Wang, J. Kim, Y. Kim, Y. J. Kwon, J. H. Yoon, W. Kim, and C. S. Hwang, "A True Random Number Generator Using Threshold-Switching-Based Memristors in an Efficient Circuit Design," *Advanced Electronic Materials*, vol. 5, no. 2, p. 1800543, 2019.

[22] R. Dittmann, S. Menzel, and R. Waser, "Nanoionic memristive phenomena in metal oxides: the valence change mechanism," *Advances in Physics*, vol. 70, no. 2, pp. 155–349, 2021.

[23] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[24] L. Xie, H. A. Du Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. AlFailakawi, and S. Hamdioui, "Scouting logic: A novel memristor-based logic design for resistive computing," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2017, pp. 176–181.

[25] K. Schnieders, P. Bai, Y. Wang, T. Kempen, D. Wouters, V. Rana, R. Waser, S. Menzel, and S. Wiefels, "Exploiting read noise of filamentary VCM ReRAM for efficient TRNG," aug 2024.

[26] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An Architecture for Fault-Tolerant Computation with Stochastic Logic," *Computers, IEEE Trans. on*, vol. 60, no. 1, pp. 93–105, Jan 2011.

[27] M. S. Moghadam, S. Aygun, M. R. Alam, and M. H. Najafi, "P2lsg: Powers-of-2 low-discrepancy sequence generator for stochastic computing," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 38–45.

[28] T.-H. Chen and J. P. Hayes, "Design of division circuits for stochastic computing," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016, pp. 116–121.

[29] Y. Zhao, W. Shen, P. Huang, W. Xu, M. Fan, X. Liu, and J. Kang, "A physics-based model of rram probabilistic switching for generating stable and accurate stochastic bit-streams," in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 32.4.1–32.4.1.

[30] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage *et al.*, "The epfl logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.

[31] C. J. Chevallier, C. H. Siau, S. F. Lim, S. R. Namala, M. Matsuoka, B. L. Bateman, and D. Rinerson, "A 0.13 μm 64mb multi-layered conductive metal-oxide memory," in *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2010, pp. 260–261.

[32] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[33] J. Yu, H. A. Du Nguyen, M. A. Lebdeh, M. Taouil, and S. Hamdioui, "Enhanced scouting logic: A robust memristive logic design scheme," in *2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 2019, pp. 1–6.

[34] T.-H. Chen and J. P. Hayes, "Equivalence among stochastic logic circuits and its application," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[35] O. Leitersdorf, D. Leitersdorf, J. Gal, M. Dahan, R. Ronen, and S. Kvatinsky, "Aritpim: High-throughput in-memory arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 3, pp. 720–735, 2023.

[36] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

[37] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[38] H. Farzaneh, J. P. De Lima, A. Nezhadi Khelejani, A. A. Khan, M. Mayahinia, M. Tahoori, and J. Castrillon, "Sherlock: Scheduling efficient and reliable bulk bitwise operations in nvms," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[39] S. Wiefels, C. Bengel, N. Kopperberg, K. Zhang, R. Waser, and S. Menzel, "Hrs instability in oxide-based bipolar resistive switching cells," *IEEE Transactions on Electron Devices*, vol. 67, no. 10, pp. 4208–4215, 2020.

[40] M. S. Moghadam, S. Aygun, S. Asadi, and M. H. Najafi, "Low-cost and highly-efficient bit-stream generator for stochastic computing division," *IEEE Transactions on Nanotechnology*, vol. 23, pp. 195–202, 2024.

[41] H. Cılasun, S. Resch, Z. I. Chowdhury, M. Zabihi, Y. Lv, B. Zink, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "On error correction for nonvolatile processing-in-memory," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 678–692.