COSMO: Computing with Stochastic Numbers in Memory

SARANSH GUPTA, University of California, San Diego
MOHSEN IMANI, University of California, Irvine
JOONSEOP SIM, ANDREW HUANG, FAN WU, and JAEYOUNG KANG,
University of California, San Diego
YESEONG KIM, Daegu Gyeongbuk Institue of Science and Technology
TAJANA ŠIMUNIĆ ROSING, University of California, San Diego

Stochastic computing (SC) reduces the complexity of computation by representing numbers with long streams of independent bits. However, increasing performance in SC comes with either an increase in area or a loss in accuracy. Processing in memory (PIM) computes data in-place while having high memory density and supporting bit-parallel operations with low energy consumption. In this article, we propose COSMO, an architecture for computing with stochastic numbers in memory, which enables SC in memory. The proposed architecture is general and can be used for a wide range of applications. It is a highly dense and parallel architecture that supports most SC encodings and operations in memory. It maximizes the performance and energy efficiency of SC by introducing several innovations: (i) in-memory parallel stochastic number generation, (ii) efficient implication-based logic in memory, (iii) novel memory bit line segmenting, (iv) a new memory-compatible SC addition operation, and (v) enabling flexible block allocation. To show the generality and efficiency of our stochastic architecture, we implement image processing, deep neural networks (DNNs), and hyperdimensional (HD) computing on the proposed hardware. Our evaluations show that running DNN inference on COSMO is 141× faster and 80× more energy efficient as compared to GPU.

CCS Concepts: • **Hardware** → **Emerging architectures**; *Non-volatile memory*; • **Computer systems organization** → *Neural networks;*

Additional Key Words and Phrases: Stochastic computing, computing in memory, processing in memory, memristors, reram, neural networks, hyperdimensional computing, image processing

ACM Reference format:
Saransh Gupta, Mohsen Imani, Joonseop Sim, Andrew Huang, Fan Wu, Jaeyoung Kang, Yeseong Kim, and Tajana Šimunić Rosing. 2022. COSMO: Computing with Stochastic Numbers in Memory. *J. Emerg. Technol. Comput. Syst.* 18, 2, Article 37 (January 2022), 25 pages.
https://doi.org/10.1145/3484731

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grant, Office of Naval Research grant # N00014-21-1-2225, and also NSF grants # 1527034, # 1730158, # 1826967, # 1911095, and # 2127780.
Authors' addresses: S. Gupta, University of California, San Diego, La Jolla, CA 92093; email: sgupta@ucsd.edu; M. Imani, University of California, Irvine, CA 92697; email: m.imani@uci.edu; J. Sim, A. Huang, F. Wu, J. Kang, and T. Š. Rosing, University of California, San Diego, La Jolla, CA 92093; emails: {j7sim, anh162, f2wu}@ucsd.edu, j5kang@eng.ucsd.edu, tajana@ucsd.edu; Y. Kim, Daegu Gyeongbuk Institue of Science and Technology, Daegu 333, Republic of Korea; email: yeseongkim@dgist.ac.kr.

1550-4832/2022/01-ART37 $15.00
https://doi.org/10.1145/3484731

1 INTRODUCTION

The era of the **Internet of Things** (**IoT**) brings together billions of inter-connected devices, which are expected to double every year [5, 63]. This results in generating a huge amount of raw data, which is processed by algorithms such as machine learning, most of which are computationally expensive [8, 71]. To ensure network scalability, security, and system efficiency, the majority of IoT data processing needs to run at least partly on the devices at the edge of the internet [12, 76, 90]. However, running data intensive workloads on traditional cores results in high energy consumption and slow processing speed due to a large amount of data movement between memory and processing units. Although processor technology has evolved to serve computationally complex tasks in a more efficient way, data transfers between processor and memory still hinder the efficiency of application performance [11, 61]. This has changed the metrics used to describe a system from being performance focused (OPS/sec) to being efficiency focused ($OPS/sec/mm^2$, $OPS/sec/W$). Restricted by these metrics, IoT devices either implement extremely simple versions of these complex algorithms, while trading a lot of accuracy, or transmit data to cloud for computation while incurring huge communication latency costs.

Interestingly, new computing paradigms have shown the capability to perform complex computations at lower area and power costs [30, 42, 92]. **Stochastic Computing** (**SC**) [24] is one such paradigm, which represents each data point in the form of a bit-stream, where the probability of having "1"s corresponds to the value of the data [6, 7, 75]. Representing data in such a format increases the size of data, with SC requiring 2^n bits to precisely represent an n-bit number. However, it comes with the benefit of extremely simplified computations and tolerance to noise [7, 31]. For example, a multiplication operation in SC requires a single logic gate as opposed to the huge and complex multiplier in integer domain. This simplification provides low area footprint and power consumption. However, with all its positives, SC comes with some disadvantages. (i) Generating stochastic numbers is expensive and is a key bottleneck in SC designs, consuming as much as 80% [69] of the total design area. (ii) Increasing the accuracy of SC requires increasing the bit-stream length, resulting in higher latency and area. (iii) Increasing the speed of SC comes at the expense of more logic gates, resulting in larger area. These pose a big challenge that cannot be solved with today's CMOS technology.

Processing In-Memory (**PIM**) is an implementation approach that uses high-density memory cells as computing elements [4, 21, 26, 60, 77]. Specifically, PIM with **non-volatile memories** (**NVMs**) like **resistive random accessible memory** (**ReRAM**) has shown great potential for performing in-place computations and hence, achieving huge benefits over conventional computing architectures [21, 37, 77]. ReRAM boasts of (i) small cell sizes, making it suitable to store and process large bit-streams, (ii) low energy consumption for binary computations, making it suitable for a huge number of bitwise operations in SC, (iii) high bit-level parallelism, making it suitable for bit-independent operations in memory, and (iv) stochastic nature at sub-threshold level, making it suitable for generating stochastic numbers.

The basic motivation behind this article is to combine the benefits of SC and PIM to obtain a system which not only has high computational ability but also meets the area and energy constraints of IoT devices. We propose COSMO, an architecture for computing with stochastic numbers in memory. The main contributions of the article are as follows:

— To the best of the authors' knowledge, COSMO is the first generalized ReRAM processing in memory architecture which brings together the benefits of both ReRAM devices and SC to efficiently support various SC encoding techniques and operations. It can accelerate a wide range of tasks.
— It is a highly parallel architecture which efficiently scales with the size of SC computations.

— COSMO combines the basic properties of ReRAM and SC to make implementation of SC on PIM highly efficient. First, COSMO exploits the stochastic nature of ReRAM devices to propose a new stochastic number generation scheme. This completely eliminates the use of stochastic number generators which can consume up to 80% area on a SC chip. Second, COSMO uses the analog nature of ReRAM PIM to present a novel SC addition that is highly compatible with memory. It is low in complexity but delivers high accuracy.
— It implements, for the first time, implication logic in regular crossbars. This enables COSMO to combine various logic families to execute logic operations more efficiently. COSMO implementation of basic SC operators using implication logic are faster and more efficient than state-of-the-art.
— The article presents detailed implementations of two learning algorithms, **deep neural networks** (**DNNs**), and **hyperdimensional** (**HD**) computing, on the proposed COSMO architecture. COSMO is highly suitable and scalable for both of them.

We evaluate COSMO over six general image processing applications, DNNs, and HD computing to show the generality of COSMO. Our evaluations show that running DNNs on COSMO is 141× faster and 80× more energy efficient as compared to GPU.

2 BACKGROUND AND RELATED WORK

2.1 Stochastic Computing

SC represents numbers in terms of probabilities using long independent bit-streams. For example, a sequence $x = 0010110001$ represents 0.4 since the probability of a random bit in the sequence to be "1", p_x is 0.4. Various encodings like unipolar, bipolar, extended stochastic logic [15], **sign-magnitude stochastic computing** (**SM-SC**) [91], and so on have been proposed, which allow converting both unsigned and signed binary number to a stochastic representation. To represent numbers beyond the range [0,1] for signed and [−1,1] for unsigned number, a pre-scaling operation is performed. Arithmetic operations in stochastic representation involve simple logic operations on uncorrelated and independently generated input bit-streams. For example, multiplication for unipolar and SM-SC encodings is implemented by ANDing the two input bit-streams x_1 and x_2 bitwise [24]. Here, all bitwise operations are independent of each other. The output bit-stream represents the product $p_{x_1} \times p_{x_2}$. For bipolar numbers, multiplication is performed using XNOR operation.

Unlike multiplication, stochastic addition, or accumulation, is not a simple operation. Several methods have been proposed which involve a direct tradeoff between the accuracy and complexity of operation. The simplest way is to OR x_1 and x_2 bitwise. Since the output is "1" in all but one case, it incurs high error, which increases with the number of inputs. The most common stochastic addition passes N input bit-streams through a **multiplexer** (**MUX**) [14]. The MUX uses a randomly generated number in range 1 to N to select one of the N input bits at a time. The output, given by $(p_{x_1}+p_{x_2}+\cdots+p_{x_N})/N$ represents the scaled sum of the inputs. It has better accuracy due to random selection. The most accurate way is to count, sometimes approximately, the N bits at any bit position to generate a stream of binary numbers [44, 67, 87]. It introduces large area and latency bottleneck due to addition at each bit position. For subtraction, which is only required for bipolar encoding, the subtrahend is first inverted bitwise. Any additional technique can then be used. Many arithmetic functions like trigonometric, logarithmic, and exponential functions can be approximated in stochastic domain with acceptable accuracy [14, 58, 68, 69]. A function $f(x)$ can be implemented using a Bernstein polynomial [69], based on the Bernstein coefficients, b_i. The work in [14] implements $f(x)$ using FSMs. For example, $tanh\frac{K}{2}x$ can be implemented using K-state FSM in SC domain, where x is the stochastic input. Another work [68] uses truncated Maclaurin series representation of $f(x)$, converting them into a series of stochastic multiplications and additions.

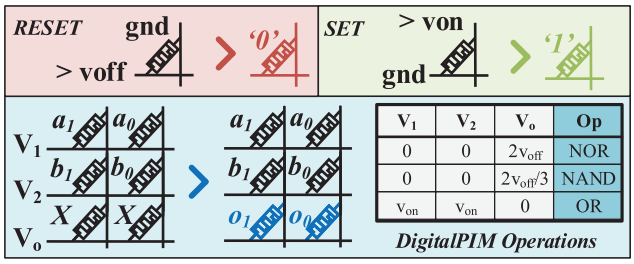


Fig. 1. Digital PIM operations. The red (green) section shows the voltage applied across RRAM device for RESET (SET) operations. The blue section shows voltages applied to obtain outputs o_i for bitwise operations on inputs a_i and b_i. The arrows (>) point to the state of the same memory cells after voltage application. The memory cells with modified states are highlighted in color.

SC is enabled by **stochastic number generators** (**SNGs**), which perform binary to stochastic conversion. It compares the input with a random, or pseudo random, number generated every cycle using a comparator. The output of the comparator is a bit-stream representing the input. The random number generator, generally a counter or a **linear feedback shift register** (**LFSR**) [27], and comparator have large area footprint, using as much as 80% of the total chip resources [69].

SC re-emerged as an active area of research with the introduction of IoT, where devices are small, less complex, and need low latency. There are recent work in multiple directions. Some try to improve the efficiency of SC operations by proposing new approximate implementations [44, 67, 87]. The work in [15, 59, 91] propose new encoding schemes for SC which are more accurate than traditional encoding. Some work also optimize SC for different applications [43, 54, 59, 75, 81, 91].

2.2 Digital Processing in Memory

A large number of recent designs enabling PIM in ReRAM are based on analog computing [21, 23, 70, 77, 83, 86]. Each element of array is a programmable multi-bit ReRAM device. In computing mode, digital input data is transferred to analog domain using **digital to analog converters** (**DACs**) and passed through a crossbar memory. The accumulated current in each memory bitlines is sensed by a **analog to digital converter** (**ADC**), which outputs a digital number. The ADCs based designs have high power and area requirements. For example, for the accelerators ISAAC [77] and IMP [23], the ReRAM crossbar consumes just takes 8.7% (1.5%) and 19.0% (1.3%) of the total power (area) of the chip. Moreover, these designs cannot support many bitwise operations, restricting their use for SC.

However, some recent work has demonstrated ways to implement logic using ReRAM switching [13, 28, 39, 48, 85]. Digital PIM exploits variable switching of memristor to implement a variety of logic functions inside memory [28, 48]. Figure 1 details the execution of digital PIM operations in RRAM. It works on the principle that a memory device switches whenever the voltage across it exceeds a threshold [49]. The red (green) section of Figure 1 shows the voltage applied across RRAM device for RESET (SET) operations. For example, in case of a RESET operation, a voltage equivalent to logical *gnd* (generally 0V) is applied at the column/bitline of the memory device, whereas a voltage greater than the device off-threshold voltage, v_{off}, is applied at the row/wordline. The blue section of Figure 1 shows the implementation of bitwise operations in digital PIM. A voltage V_1 (V_2) is applied at the wordlines with inputs a_i (b_i), while voltage V_o is applied at the output wordline. As shown, these operations can be implemented in parallel over multi-bits ($[o_0 = a_0\ Op\ b_0]$, $[o_1 = a_1\ Op\ b_1]$), even the entire row of memory. The output of operation changes with the applied voltage [28]. The table shows voltages applied to obtain outputs o_i for bitwise operations (NOR,

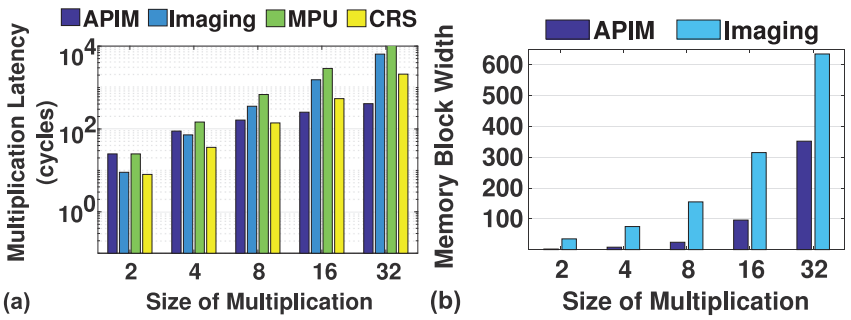


Fig. 2. (a) Change in latency for binary multiplication with the size of inputs in state-of-the-art PIM techniques. (b) The increasing block size requirement in binary multiplication.

NAND, and OR) on inputs a_i and b_i. Digital PIM allows high-density operations within memory without reading out the data. In this article, we utilize digital PIM to implement a majority of stochastic operations. In addition, we also introduce, for the first time, support for an entire class of digital logic, i.e., implication logic, in regular crossbar memory using digital PIM.

3 WHY STOCHASTIC ON PIM?

In this section, we present the motivation behind COSMO. We discuss the challenges faced by the current ReRAM-based PIM techniques. We then explore how the properties of ReRAM are suitable for SC. We also show some inefficiencies associated with the direct adoption of SC to the PIM and show how the memory compatible operations in COSMO can help alleviate them.

PIM on Conventional or Stochastic Data: The digital PIM designs, discussed in Section 2.2, use the conditional switching of ReRAM devices to implement logic functions in memory. A series of these switching operations is used to implement more complex functions like addition and multiplication. This results in large latency, which increases with the size of the inputs. Figure 2(a) shows the way the latency increases with the bit-length of inputs for binary multiplication in current PIM techniques [29, 37, 80, 85]. There is an approximately exponential increase in the latency, consuming at least 164 (254) cycles for 8-bit (16-bit) multiplication. As shown in the previous section, multiplication in stochastic domain just requires bitwise AND/XOR of the inputs. With stochastic bits being independent from each other, increasing the bit-length in stochastic domain does not change the latency of operation, requiring just two cycles for both 8-bit and 16-bit multiplications.

PIM-based SNG: Another major bottleneck or overhead in most SC based designs is the SNG. SNGs may take up to 80% of the total area of a stochastic design [69]. This poses a big issue for energy aware applications. The work in [46] shows that the stochastic nature of ReRAM inherently supports generations of stochastic numbers. Moreover, this does not require any major addition to the memory periphery since it uses the trivial circuits like voltage controllers, pulse generation circuits, and so on, most of which are already present.

PIM Parallelism: Digital PIM is often limited by the size of the memory block, particularly the width. Figure 2(b) shows how the size of operands increases the demand for larger memory blocks in binary multiplication. SC is uniquely able to overcome this issue. Since each bit in stochastic domain is independent, the bits may be stored over different blocks without changing the logical perspective.

Although the stochastic operations are simple, parallelizing stochastic computation in conventional (CMOS) implementations comes at the cost of a direct, sometimes linear, increase in the hardware requirement. However, the independence between stochastic bits allows for extensive bit-level parallelism which many PIM techniques support.

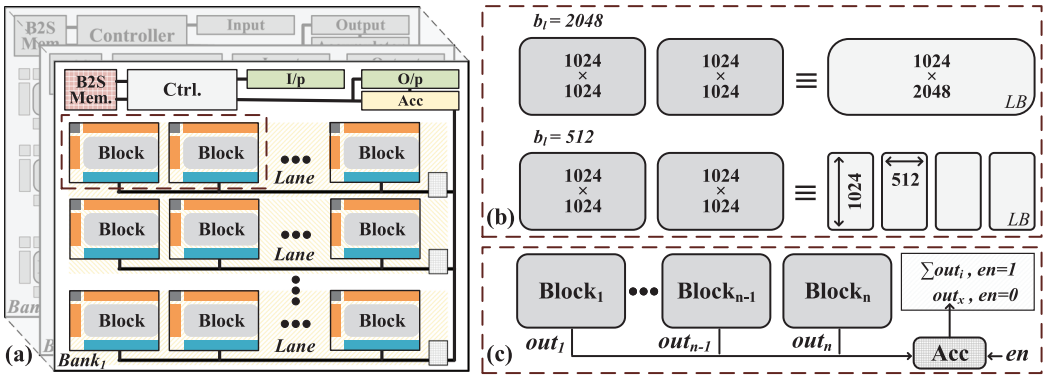


Fig. 3. COSMO overview.

4 COSMO OVERVIEW

In this article, we present COSMO, a digital ReRAM-PIM based architecture for SC. It is a general stochastic platform, which supports all SC computing techniques. It combines the complementary properties of ReRAM-PIM and SC as discussed in Section 3. COSMO architecture consists of multiple ReRAM crossbar memory blocks grouped into multiple banks (Figure 3(a)). A memory block is the basic processing element in COSMO, which performs stochastic operations using digital ReRAM-PIM. The feature with enables COSMO to perform stochastic operations is the support for *flexible block allocation*.

Now, due to the limitations of crossbar memory [64, 89], the size of each block is restricted to, say, $1,024 \times 1,024$ ReRAM cells in our case. In a conventional architecture, if the length of stochastic bit-streams, b_l, is less than 1,024, it results in under-utilization of memory. Lengths of $b_l > 1,024$ could not be supported. COSMO on the other hand allocates blocks dynamically. It divides a memory block (if $b_l < 1,024$), or groups multiple blocks together (if $b_l > 1,024$) to form a *logical block* (Figure 3(b)). A logical block has a logical row-size of b_l cells. This logical division and grouping is done dynamically by the bank controller. All the blocks in a bank work with the same bit-stream length, b_l. However, different banks can configure logical blocks independently, enabling simultaneous multi-b_l support. In addition, COSMO uses a within-a-block partitioning approach where a memory block is divided into 32 smaller partitions by segmenting the memory bitlines. The segmentation is performed by a novel buried switch isolation technique. The switches, when turned-off, isolate the segments from each other. This results in 32 smaller partitions, each of which *behaves* like a block of size $32 \times 1,024$. This increases the intra-block parallelism in COSMO by up to $32\times$.

Any stochastic application has three major phases, (i) binary to stochastic conversion (B2S), (ii) stochastic logic computation, (iii) stochastic to binary (S2B) conversion. COSMO follows bank level division where all the blocks in a bank work in the same phase at a time. The stochastic nature of ReRAM cells allows COSMO memory blocks to inherently support B2S conversion. Digital PIM techniques combined with memory peripherals enable logic computation in memory and S2B conversion in COSMO. S2B conversion over multiple physical blocks is enabled by accumulator-enabled bus architecture of COSMO (Figure 3(c)).

5 STOCHASTIC PIM

In this section, we present the hardware innovations which make COSMO efficient for SC. First, we present a PIM-B2S conversion technique. Then, we propose a new way to compute logic in memory. Next, we show how we bypass the physical limitations of previous PIM designs to achieve a highly parallel architecture. Last, we show the implementation of different SC operations in COSMO.

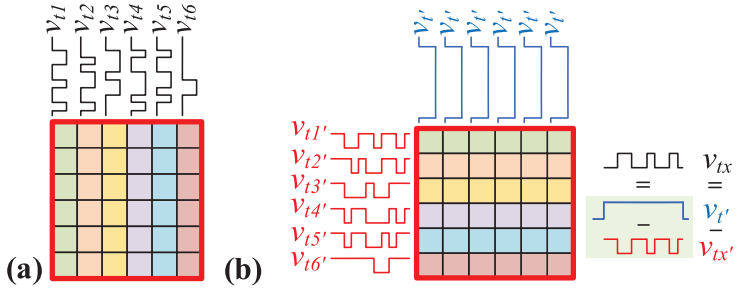


Fig. 4. Generation of stochastic numbers using (a) group write [46], (b) COSMO row-parallel generation.

5.1 Stochastic Number Generation

The ReRAM device switching is probabilistic at sub-threshold voltages, with the switching time following a Poisson distribution [40]. For a fixed voltage, the switching probability of a memristor can be controlled by varying the width of the programming pulse. The group writes technique presented in [46] showed that stochastic numbers of large sizes can be generated over multiple bits of a column in parallel. It first deterministically programs all the memory cells to zero (RESET) and then stochastically, based on the input number, programs them to one (SET). However, since digital PIM is row-parallel; it is desirable to generate such a number over a row. This can be achieved in two ways:

ON→OFF Group Write: To generate a stochastic number over a row, we need to apply the same programming pulse to the row. As shown before in Figure 1, the bipolar nature of memristor allows it to switch only to "0" by applying a voltage at the wordline. Hence, a ON→OFF group write is needed. Stochastic numbers can be generated over rows by applying stochastic programming pulses at word lines instead of bitlines. However, a successful stochastic number generation requires us to SET all the rows initially. This results in a large number of SET operations. The SET phase is both slower as well as more energy consuming than the RESET phase, making this approach very inefficient. Hence, we propose a new generation method.

COSMO Row-Parallel Generation: The switching of memristor is based on the effective voltage across its terminals. In order to achieve low static energy for initialization, we RESET all the rows like the original group write. However, instead of applying different voltage pulses, $v_{t1}, v_{t2}, \ldots, v_{tn}$, to different bitlines, we apply a common pulse, $v_{t'}$, to all the bitlines. A pulse, v_{tx}, applies a voltage v with a time width of tx. Now, we apply pulses, $v_{t1'}, v_{t2'}, \ldots, v_{tn'}$, to different wordlines such that $v_{tx} = v_{t'} - v_{tx'}$. It generates stochastic numbers over multiple rows in parallel, as shown in Figure 4(b).

COSMO's stochastic number generation alleviates the need for special resource allocation for SNG by enabling it in a standard memory crossbar. Hence, each memory crossbar in COSMO can generate stochastic numbers independently without having an effect on the latency of other parts of the memory. This is in contrast with CMOS-based SC accelerators, where single or limited number of SNGs may introduce a latency bottleneck wherein all inputs utilize the same SNG(s). In CMOS-based SC designs, there exists a tradeoff between amortizing the area overhead of SNG by sharing it for multiple inputs and increasing the parallelism in design.

5.2 Efficient PIM Operations

SC multiplication with bipolar (unipolar, SM-SC) numbers involves XNOR (AND). While the digital PIM discussed in Section 2.2 implements these functions; they are inefficient in terms of latency, energy consumption, memory requirement, number of device switches. We propose to use

Table 1. Comparison of the Proposed XNOR and and with State-of-the-art Techniques

	Latency (cycles)		Energy (fJ)		Memory Req. (# of cells)		Device Sw. (# of cells)	
	XNOR	AND	XNOR	AND	XNOR	AND	XNOR	AND
COSMO	2	2	37.1	45.2	1	2	≤2	≤2
FELIX [28]	3	2	53.7	48.8	2	2	≤3	≤2
MAGIC [48]	5	3	120.29	64.1	5	3	≤5	≤5

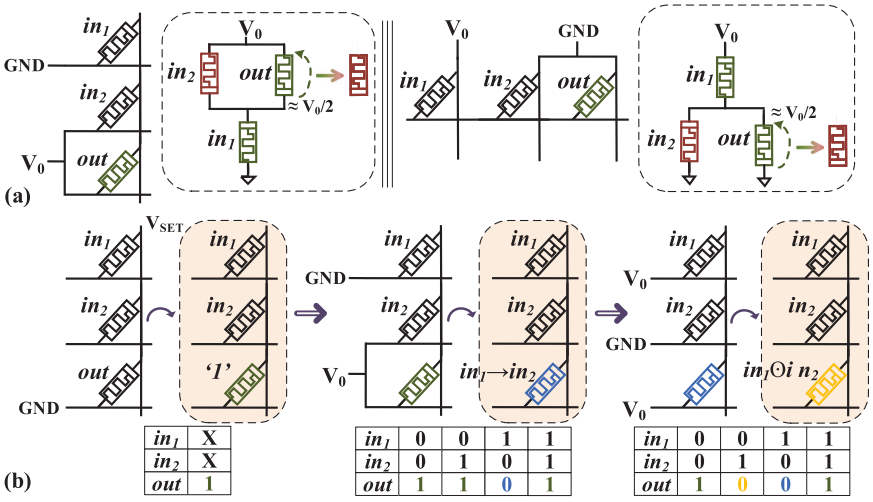


Fig. 5. (a) Implication in a column/row, (b) XNOR in a column.

an implication-based logic. Implication ($\rightarrow$, where $A \rightarrow B = A' + B$) combined with false (always zero) presents a complete logic family. XNOR and AND are implemented using implication very efficiently, as described in Table 1. Some previous work implemented implications in ReRAM [13, 50, 55]. However, they required additional resistors of specific value to be added to the memory crossbar. Instead, COSMO enables, for the first time, implication-based logic in conventional crossbar memory, with the same components as the basic digital PIM. Hence, COSMO supports both implication and basic digital PIM operations.

COSMO Implication in-memory: As discussed in Section 2.2, a memristor requires a voltage greater than v_{off} ($-v_{on}$) to switch from "1" ("0") to "0" ("1") to **high resistive state** (**HRS**) (HRS, logical "0"). We exploit this switching mechanism to implement implication logic in memory. Consider three cells, two input cells, and an output cell, in a row of crossbar memory as shown in Figure 5(a). We apply an execution voltage, V_0, at the bitline corresponding to one of the inputs (in_1) while ground the other input (in_2) and the output cell (out). Let out be initialized to "1". In this configuration, out switches to "0" only when the voltage across it is greater or equal to v_{off}. For all the cases when in_1 is "0", most of the voltage drop is across in_1, resulting in a negligible voltage across out. In case in_1 is "1", the voltage across out is ~V_0/3 and ~V_0/2 when in_2 is "1" and "0", respectively. If $2 * v_{off} \leq V_0 < 3 * v_{off}$, then out switches only when in_1 is "1" and in_2 is "0". This results in the truth table shown in Figure 5(a), corresponding to $in_1 \rightarrow in_2$. To execute $in_2 \rightarrow in_1$, V_0 is applied to in_2 while in_1 and out are grounded.

COSMO XNOR in-memory: XNOR ($\odot$) can be represented as, $A \odot B = (A \rightarrow B).(B \rightarrow A)$. Instead of calculating $in_1 \rightarrow in_2$ and $in_2 \rightarrow in_1$ separately and then ANDing them, we first calculate

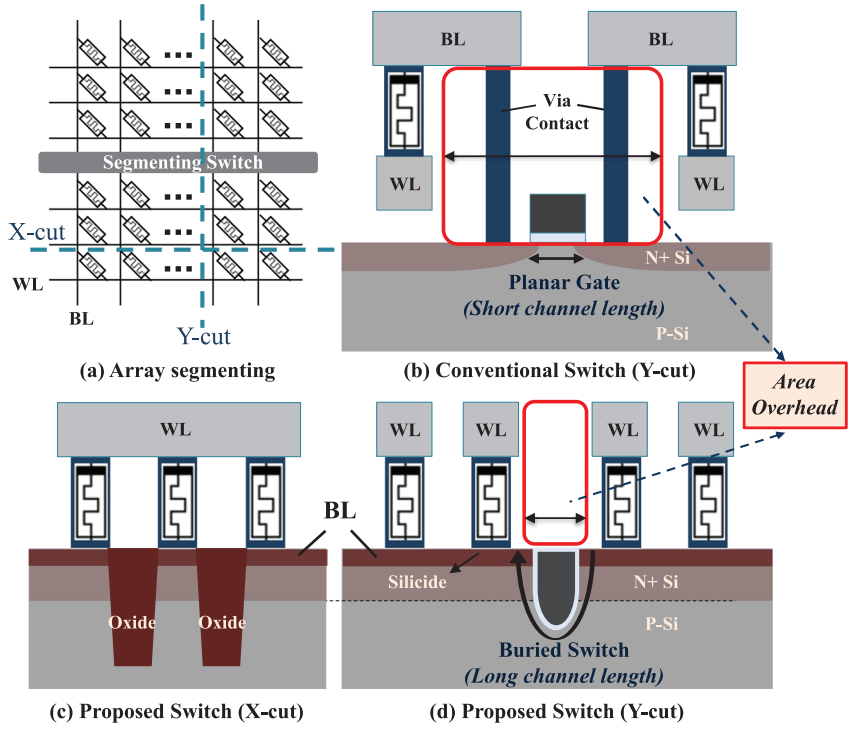


Fig. 6. Buried switch technique for array segmenting.

$in_1 \rightarrow in_2$ and then use its output cell to implement $in_2 \rightarrow in_1$ as shown in Figure 5(b). In this way, we eliminate separate execution of AND operation.

COSMO AND in-memory: AND (.) is represented as, $A.B = (A \rightarrow B')'$. The inversion uses NOT presented in [50]. Similar to the implementation presented in Section 2.2, all bitwise operations proposed in this section can be executed for all the columns (rows) of a row (column) in parallel.

5.3 Memory Bitline Segmentation

As discussed in Section 2.2, digital PIM supports row-level parallelism, where an operation can be applied between two or threes row for the entire row-width in parallel. However, parallelism between multiple sets of rows is not possible. We enable multiple-row parallelism in COSMO by segmenting memory blocks. As shown in Figure 6(a), a row of segmenting switches physically divides two segments of memory block, preventing the interference of currents from different segments. Prior works have segmented the array blocks using conventional transistors for the same purpose [25, 28, 53, 74, 79], which utilizes a planar-type transistor. This type of structure has mainly two drawbacks: (1) large area overhead and (2) off-leakage due to short channel length. As shown in Figure 6(b), the area of a single transistor with the planar type structure is impacted by gate length, via contact area, gate to via space, via to adjacent-WL space in pairs for side of gate. On the other hand, we design a novel bitline isolation technique using *buried switches*. Figure 6(c) and (d) describe the cross sectional view of X-cut and Y-cut of the proposed design, respectively. COSMO utilizes a conductor on silicon, called *silicide*, to design the switch. This allows COSMO to fabricate the switch using a simple trench process [34]. Here, instead of independent switches for different bitlines, we have a single row-width wide switch.

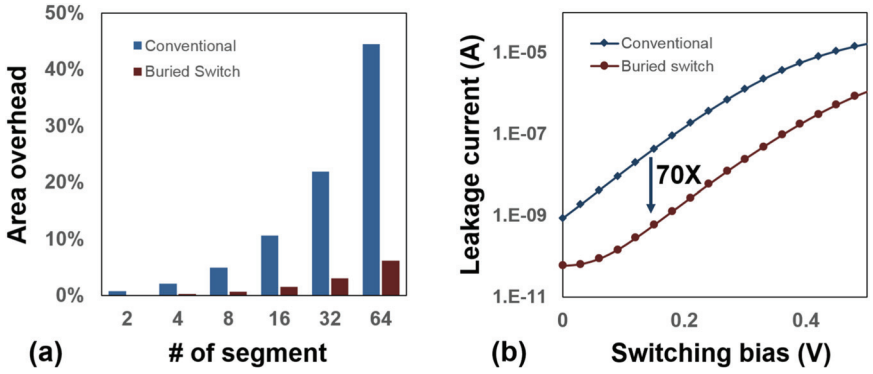


Fig. 7. (a) Area overhead and (b) leakage current comparison of proposed segmenting switch to the conventional design.

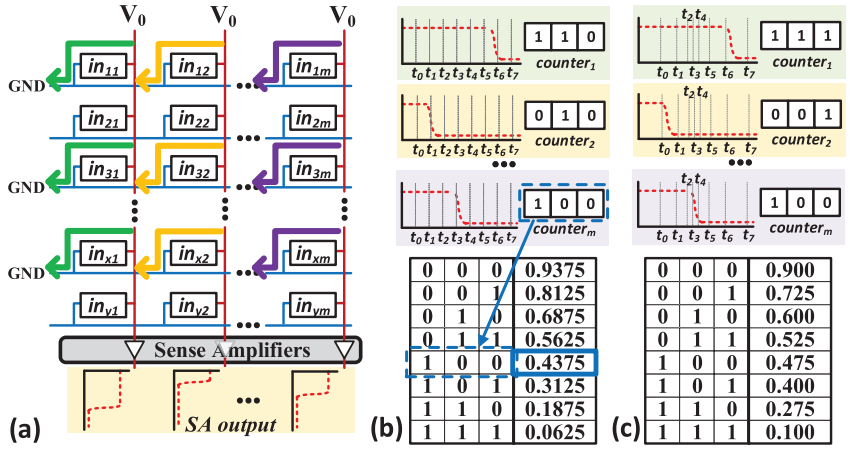
From the perspective of area, COSMO segmenting needs only a trench width in lateral direction. Figure 7(a) shows the change in area overhead due to segmentation as the number of segments increases. The estimated area from Cadence p-cell data with 45 *nm* process shows that COSMO has 7.2× less silicon footprint as compared to conventional MOSFET-based isolation [53, 74]. Due to its highly efficient segmentation, COSMO with 32 partitions results in just 3% crossbar area overhead. In addition, the buried switch makes channel length longer than the conventional switch, as shown in Figure 6(d). This suppresses the *short channel effect* of conventional switches. As a result, COSMO achieves 70× lower leakage current in the subthreshold region (Figure 7(b)), enabling robust isolation. Switches can be selectively turned-off or on to achieve the required configuration. For example, alternate switches can be turned-on to have 16 partitions of size 64×1024 each.

5.4 SC Arithmetic Operations in COSMO

Here, we explain how COSMO implements SC operations. The operands are either generated using the B2S conversion technique in Section 5.1 or are pre-stored in memory as outputs of previous operations. They are present in different rows of the memory, with their bits aligned. The output is generated in the output row, bit-aligned with the inputs.

Multiplication: As explained in Section 2, multiplication of two numbers in stochastic domain involves a bitwise XNOR (AND) between bipolar (unipolar, SM-SC) numbers across the bit-stream length. This is implemented in COSMO using the PIM technique explained in Section 5.2.

Conventional Addition/Subtraction/Accumulation: Implementations of different stochastic N-input accumulation techniques (OR, MUX, and count-based) discussed in Section 2 can be generalized to addition by setting the number of inputs to two. In case of subtraction, the subtrahend is first inverted using a single digital PIM NOT cycle. Then, any addition technique can be used. The OR operation is supported by COSMO using the digital PIM operations [28], generating OR of N bits in single cycle. The operation can be executed in parallel for the entire bit-stream, b_l, and takes just one cycle to compute the final output. To implement MUX-based addition in memory, we first stochastically generate b_l random numbers between 1 to N using B2S conversion in Section 5.1. Each random number selects one of the N inputs for a bit position. The selected input bit is read using the memory sense amplifiers and stored in the output register. Hence, MUX-based addition takes one cycle to generate one output bit, consuming b_l cycles for all the output bits. To implement **parallel count** (**PC**)-based addition in memory, one input bit-stream (b_l bits) is read out by the sense amplifier every cycle and sent to counters. This is done for N inputs sequentially, consuming N cycles. In the end, the counters store the total number of ones at each bit position in the inputs.



0	0	0	0.9375
0	0	1	0.8125
0	1	0	0.6875
0	1	1	0.5625
1	0	0	0.4375
1	0	1	0.3125
1	1	0	0.1875
1	1	1	0.0625

0	0	0	0.900
0	0	1	0.725
0	1	0	0.600
0	1	1	0.525
1	0	0	0.475
1	0	1	0.400
1	1	0	0.275
1	1	1	0.100

Fig. 8. COSMO addition and accumulation in parallel across bit-stream. (a) Discharging of bitlines through multiple rows (rows $1, 3, \ldots, x$ here), (b) linear COSMO addition with counter value to output relation, and (c) non-linear COSMO addition centered around 0.5.

COSMO Addition: Count-based addition is the most accurate but also the slowest of the previously proposed methods for stochastic addition. Instead, we use the analog characteristics of memory to generate a stream of b_l binary numbers representing the sum of the inputs. As shown in Figure 8(a), the execution of addition in COSMO takes place in two phases. In the first phase, all the bitlines are pre-charged. In the second phase, only those wordlines or rows which contain the inputs of addition are grounded, while the rest of the wordlines are kept floating. This results in discharging of the bitlines. However, the speed of discharging depends upon the number of low resistive paths, i.e., the number of "1"s. Hence, more is the number of "1"s in a bitline, faster is the discharging. To detect the discharge edges, we use the traditional 1-bit sense amplifier, along with a time-controlled latch. We set the circuit to latch at eight different time steps, generating 3-bit ($log_2 8$) outputs (Figure 8(b)). The accuracy of this addition can be increased by having non-linear time steps as in Figure 8(c). We tested the accuracy of the proposed COSMO addition by generating 1M random binary numbers. We then calculated the average absolute errors. The accuracy of the COSMO addition is very close to the count-based addition (accuracy loss of 1.92% for $b_l = 16$, which decreases to 0.34% for $b_l = 128$). Increasing the number of time steps further increases the accuracy but at the cost of significant area overheads of increased latching and counting circuit.

Other Arithmetic Operations: COSMO supports trigonometric, logarithmic, and exponential functions using truncated *Maclaurin Series* expansion [68]. The expansion approximates these functions using a series of multiplications and additions. With just 2–5 expansion terms, it has been shown to produce more accurate results [68] than most other stochastic methods [14, 69].

6 COSMO ARCHITECTURE

A COSMO chip is divided into 128 banks, each consisting of 1,024 ReRAM memory blocks. A memory block is the basic processing element in COSMO. Each block is a $1,024 \times 1,024$-cell ReRAM crossbar. A block has a set of row and column drivers, which are responsible for applying appropriate voltages across the memory cells to read, write, and process the data. They are controlled by the memory controller.

COSMO Bank: A bank has 1,024 memory blocks arranged in 32 lanes with 32 blocks each. A bank controller issues command to the memory blocks. It also performs the logical block allocation

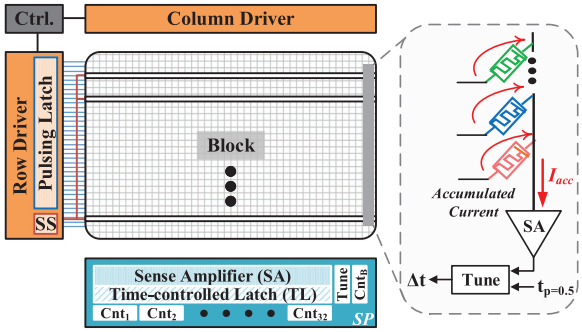


Fig. 9. A COSMO block.

in COSMO. Each bank has a small memory that decodes the programming time for B2S conversions. Using this memory, the bank controller sets the time corresponding to an input binary number for a logical block. The memory blocks in a bank lane are connected with a bus. Each lane bus has an accumulator to add the results from different physical blocks.

COSMO Block: Each block is a crossbar memory of $1,024 \times 1,024$ cells (Figure 9). Each block can be segmented into up to 32 partitions using the buried switch isolation of Section 5.3. In addition to the memory sense amplifiers, block peripheral circuits include a 10-bit ($log_2 1024$) counter per 32 columns to implement accumulation. Each block also use an additional 10-bit counter to support pop count across rows/columns.

Variation-Aware Design: ReRAM device properties show variations with time, temperature, and endurance, most of which change the device resistance. To make COSMO more resilient to resistance drift, we use only single-level RRAM cells (RRAM-SLCs) whose large off/on resistance ratio makes them distinguishable even under large resistance drifts [88]. Moreover, the probabilistic nature of SC makes it resilient to small noise/variations. To deal with large variations due to overtime decay or major temperature variations, COSMO implements a simple feedback enabled timing mechanism, as shown in Figure 9. One dummy column in a memory block is allocated to implement this approach. The cells in the designated column are activated and the total current through them is fed to a tuner circuit. The circuit outputs Δt, which is used to change the pulse widths for input generation and sense amplifier operations.

COSMO parallelism with bit-stream length: COSMO implements operations using digital PIM logic, where computations across the bit-stream can be performed in parallel. This results in proportional increase in performance, while consuming similar energy and area as bit-serial implementation. In contrast, the traditional CMOS implementations scale linearly with the bit-stream length, incurring large area overheads. Moreover, the dynamic logical block allocation allows the parallelism to extend beyond the size of block.

COSMO parallelism with number of inputs: COSMO can operate on multiple inputs and execute multiple operations in parallel within the same memory block. This is enabled by COSMO memory segmentation. When the segmented switches are turned-off, the current generated flowing through a bitline of a partition is isolated from currents of any other partition. Hence, COSMO can execute operations in different partitions in parallel.

7 LEARNING ON COSMO

Here, we study the implementation of two types of learning algorithms, DNNs and brain-inspired **hyper-dimensional** (**HD**) computing, on COSMO. The goal is to show the generality of COSMO

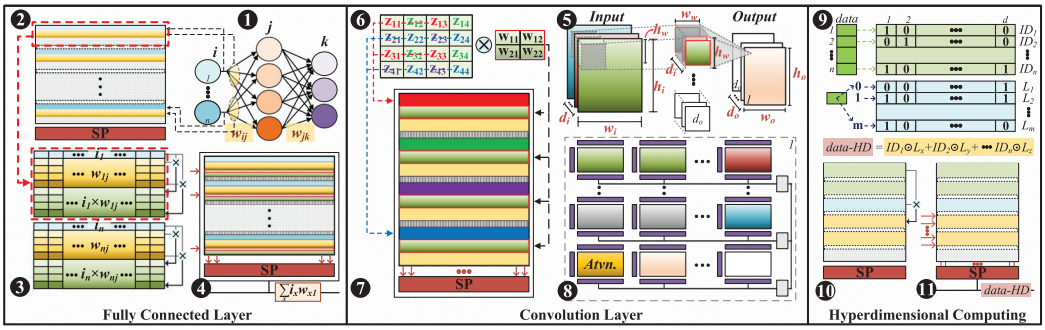


Fig. 10. Implementing fully connected layer, convolution layer, and HD computing on COSMO.

by implementing a compute-intensive algorithm like DNN as well as more hardware-friendly algorithm, HD computing. Figure 10 summarizes the discussion in this section.

7.1 Deep Neural Networks Inference

There has been some interest in implementing DNNs using SC [10, 43, 56, 59, 75, 91]. They provide high performance but that performance comes at the cost of huge silicon area. Here, we show the way COSMO can be used to accelerate DNN applications. We describe COSMO implementation of different layers of neural networks, namely fully connected, convolution, and pooling layers.

Fully-Connected (FC) Layer: A FC layer with n inputs and p outputs is made up of p neurons. Each neuron has a weighted connection to all the n inputs generated by the previous layer. All the weighted inputs for a neuron are then added together and passed through an activation function, generating the final result. COSMO distributes weights and inputs over different partitions of one or more blocks. Say, a neural network layer, j, receives input from layer i. The weighted connections between them can be represented with a matrix, w_{ij} (❶). Each input (i_x) and its corresponding weights (w_{xj}) are stored in a partition (❷). Inputs are multiplied with their corresponding weights using XNOR and the outputs are stored in the respective partition (❸). Multiplication happens serially within a partition but in parallel across multiple partitions and blocks. Then, all the products corresponding to a neuron are selected and accumulated using COSMO addition (❹). If $2p$+1 (one input, p weights, p products) is less than the rows in a partition, the partition is shared by multiple inputs. If $2p$+1 is greater than the number of rows, then w_{xj} are distributed across multiple partitions.

Activation: Activation function brings non-linearity to the system and generally consists of non-linear functions like *tanh*, *sigmoid*, *ReLU*, and so on. Of these, *ReLU* is the most widely used operation. It is a threshold-based function, where all numbers below a threshold value (v_T) are set to v_T.The output of FC accumulation is pop counter and compared to v_T. All the numbers to be thresholded are replaced with stochastic v_T. Other activation functions like *tanh* and *sigmoid*, if needed, are implemented using the Maclaurin series-based operations discussed in Section 5.4.

Convolution Layer: Unlike FC layer, instead of a single big set of weights, convolution has multiple smaller sets called weight kernels. A kernel moves through the input layer, processing a same-sized subset (*window*) of the input at a time and generates one output data point for each (❻). A **multiply and accumulate** (**MAC**) operation is applied between a kernel and a *window* of the input at a time. Say, a convolution layer convolves an input layer (of width w_i, height h_i and depth d_i) with d_o weight kernels (of width w_w, height h_w, and depth d_i) and generates an output with depth d_o.

We first show how a convolution between a single depth of input and weight kernel maps to COSMO. In ❺, a 4×4 input is convolved with a 2×2 weight kernel. To completely parallelize

multiplication in a convolution window, we distribute input such that all input elements in any window are stored in separate partitions. Same colored inputs in ❺ go to the same partitions as shown in ❼. Each partition further has all the weights. To generalize, COSMO splits input into $h_w \times w_w$ partitions ($part_{11}, part_{12}, \ldots, part_{h_w w_w}$). A partition, $part_{ij}$, has all the weights in the kernel and the input elements at every h_wth column and w_wth row starting from (i, j). A partition may be distributed over multiple physical COSMO segments as described in Section 5.3.

The MAC operation in a window is similar to the fully connected layer explained before. Since all the inputs in a window are mapped to different partitions (❼), all multiplication operations for one window happen in parallel. The rows corresponding to all the products for a window are then activated and accumulated. The accumulated results undergo activation function (*Atvn.* in ❽) and then, are written to the blocks for next layer. While all the windows for a unit depth of input are processed serially, different input depth levels and weight kernel depth levels are evaluated in parallel in different blocks ❽. Further, computations for d_o weight kernels are also parallelized over different blocks.

Pooling: A pooling window of size $h_p \times w_p$ is moved through the previous layer, processing one subset of input at a time. MAX, MIN, and average pooling are the three most commonly used pooling techniques. While average pooling is same as applying COSMO addition over a subset of the inputs in pooling window, MAX/MIN operations are implemented using the discharging concept used in COSMO addition. The input in the subset discharging the first (last) corresponds to the maximum (minimum) number.

Batch Normalization: During inference, all parameters of batch normalization are fixed and known in advance. This reduces batch normalization to a simpler linear transformation operation, where the mean of the batch is subtracted using COSMO subtraction, followed by multiplication with the inverse of the variance. The remaining scaling and addition operations are performed with COSMO multiplication and addition.

7.2 Hyperdimensional Computing

HD computing tries to mimic human brain and computes with *patterns of numbers* rather than the numbers themselves [41]. It is a hardware efficient algorithm that provides extremely high parallelism. HD represents data in the form of high-dimension (thousands) vectors, where the dimensions are independent of each other. The long bit-stream representation and dimension-wise independence make HD very similar to SC. HD computing consists of two main phases: encoding and similarity check [36, 72, 73].

Encoding: The encoding uses a set of orthogonal hypervectors, called *base hypervectors*, to map each data point into the HD space with d dimensions. As shown in ❾, each feature of a data point (feature vector) has two base hypervectors associated with it: identity hypervector, *ID*, and level hypervector, *L*v [38]. Each feature in the data point has a corresponding *ID* hypervector. The different values which each feature can take have corresponding *L* hypervectors. The *ID* and *L* hypervector for a data point are XNORed together. Then, the XNORs for all the features are accumulated to get the final hypervector for the data point [28].

The value of d is usually large, 1,000s to 10,000s, which makes conventional architectures inefficient for HD computing. COSMO, being build for SC, presents the perfect platform for HD computing. The base hypervectors are generated just once. COSMO creates the orthogonal base hypervectors by generating d-bit long vectors with 50% probability, as described in Section 5.1. It is based on the fact that randomly generated hypervectors are orthogonal. For each feature in data, the corresponding *ID* and *L* are selected and then XNORed using COSMO XNOR (❿). The outputs of XNOR for different features are stored together. Then, all the XNOR results are selected and accumulated using COSMO addition (⓫) and further sent for similarity check.

Similarity Check: HD computes the similarity of the unseen test data point with pre-stored hypervectors. The pre-stored hypervectors may represent different classes in case of classification applications. COSMO computes the similarity of a test hypervector with k class hypervectors by performing k dot products between vectors in d dimensions. The hypervector with the highest dot product is selected as the output. To implement the dot product, the encoded d-dimension feature vector is first bitwise XNORed, using COSMO XNOR, with k d-dimension class hypervectors. It generates k product vectors of length d. COSMO then finds the maximum of the product hypervectors using the discharging mechanism of COSMO addition.

8 EVALUATION

8.1 Experimental Setup

We develop C++-based cycle-accurate simulator which emulates the functionalities of COSMO. The simulator uses the performance and energy characteristics of the hardware are obtained from circuit level simulations for a 45 nm CMOS process technology using Cadence Virtuoso. We use VTEAM memristor model [49] for our memory design simulation with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively and a switching delay of 1.1 ns. This forms the COSMO's design cycle. All COSMO operations have been designed to meet the memory cycle constraint of 1.1 ns. We implement logic operations using digital PIM operations [28] discussed in brief in Section 2.2 and shown in Figure 1. In doing so, some operations, like NOR, take one cycle while others take multiple cycles. For example, implementing XOR operation in COSMO takes three cycles. For other operations, like COSMO-addition, we take the simulator estimate of sense-amplifier read time as 1.1 ns even though it is much faster than that. We simulate a memory block under various conditions for different operations in Virtuoso. We record the latency and energy consumption at different settings and use them in our simulator. Our simulator maps the application over many memory blocks and utilizes the results obtained from memory block simulation to calculate the latency and the average power consumption of our design.

We compare the efficiency of the proposed COSMO with state-of-the-art processor NVIDIA GPU GTX 1080 Ti. While reporting the execution time for GPU, we preload the data onto GPU and report only the GPU execution time. We utilize *nvprof* with Nvidia Visual Profiler (NVVP) to get the execution time for individual kernels and consider only the kernels corresponding to application computations while comparing GPU performance with COSMO. However, GPU has certain GPU-side overheads like context switching, hardware scheduling, and so on, which may increase its execution time.

We consider COSMO efficiency on several image processing and learning applications. For image processing, we used four general applications, including: *Sobel, Robert, Prewitt*, and *BoxSharp*. We use random images from *Caltech 101* [57] library. For learning application, evaluate COSMO efficiency on DNN and HD computing applications. As Table 2 shows, we test COSMO efficiency and accuracy on four popular networks running on large-scaled ImageNet dataset [47]. The GPU evaluations for DNNs were done using their PyTorch [1] implementations. We used Brevitas library from Xilinx to obtain their integer models [66]. For HD computing, we evaluated COSMO accuracy and efficiency on four practical applications including speech recognition (ISOLET), face detection (FACE), activity recognition (UCIHAR), and security detection (SECURITY). To compare COSMO with GPUs, we developed a GPU-optimized version of the CPU implementation presented in [35]. HD similarity check-in GPU implements dot product between two 10,000 dimension vectors. This operation is the same for all datasets because the encoded input hypervector has dimensionality of 10,000 for each dataset. For an inference input, we instantiate k such operations, where k is the number of classes. Table 2 compares the baseline accuracy (32-bit integer values) and the quality

Table 2. DNN and HD Computing Workloads

Deep Neural Networks			Hyperdimensional Computing (10,000 dimensions)			
ImageNet Networks	*Base Top-5 Accuracy*	*Quality Loss*	*Apps*	*# Classes*	*Base Top-1 Accuracy*	*Quality Loss*
AlexNet [16, 47]	80.3%	1.32%	**ISOLET** [22]	26	96.4%	0.72%
ResNet-18 [18, 33]	87.6%	0.76%	**FACE** [45]	2	95.3%	0.29%
VGG-16 [19, 82]	89.3%	1.18%	**UCIHAR** [9]	12	93.1%	0.49%
GoogleNet [17, 32]	90.8%	1.61%	**SECURITY** [62]	10	93.2%	1.20%

loss of the applications running on COSMO using 32-bit SM-SC [91] encoding. Our evaluation shows that COSMO can result only about 1.5% and 1% quality loss on DNN and HD computing.

8.2 COSMO Tradeoffs

COSMO, and SC in general, depends on the length of bit-stream. The greater the length, higher is the accuracy. However, this increase in accuracy comes at the cost of increased area and energy consumption. As the length is increased, more area (both memory and CMOS) is required to store and process the data, requiring more energy. It may also result in higher latency for some operations like MUX-based additions, for which the latency increases linearly with bit-stream length. To evaluate the effect of bit-stream length at operation level, we generate random inputs for each operation and take the average of 100 such samples. Each accumulation operation inputs 100 stochastic numbers. Moreover, the results here correspond to unipolar encoding. However, all other encodings have similar behavior with slight change in accuracy. An increase in bit-stream length has a direct impact on the accuracy, area, and energy at operation level. While the latency of the design remains same for all operations except MUX-based addition, Bernstein polynomial, and FSM-based operations. It happens because these operations process each bit sequentially. While COSMO supports MUX-based addition, it uses the proposed COSMO-addition (Section 5.4) by default, which does not scale linearly with latency. When implemented with a bit-stream length of 256, all operations have on an average 4× improvement in area and energy consumption as compared to the corresponding implementation with a bit-stream length of 1,024 and incur 3.6% quality loss. For the same change in bit-stream length, the latency of MUX-based addition, Bernstein polynomial, and FSM-based operations differ on an average by 3.95×.

To evaluate the effect at application level, we implement the general applications listed above using COSMO with an input dataset of size 1 kB. The results shown here use unipolar encoding with AND-based multiplication, COSMO addition, and Maclaurin series-based other arithmetic functions. Since all these operations are scalable with the bit-stream length, the latency of the operations does not change. The minor increase in the latency at application level with the length is due to the time taken by stochastic-to-binary conversion circuits. However, this change is negligible. Figure 11 shows the impact of bit-stream length on different applications. On an average, both the area and energy consumption of the applications increase by 8×, when the bit-stream length increases from 512 to 4,096, with an average 6.1 dB PSNR gain. As shown in Figure 12, with a PSNR of 29 dB, the output of Sobel filter with bit-stream length of 4,096 is visibly similar to that of the exact computation.

8.3 Learning on COSMO and GPU

COSMO Configurations: We compare COSMO with GPU for the DNNs and HD computing workloads detailed in Table 2. We use SM-SC encoding with a bit-stream length of 32 [91] to represent the inputs and weights in DNNs and value of each dimension in HD computing on COSMO. Also, evaluation is performed while keeping the COSMO area and technology node the same as GPU. We analyze COSMO in five different configurations to evaluate the impact

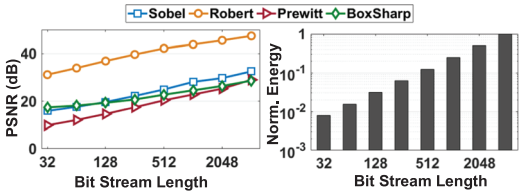


Fig. 11. Effect of bit-stream length on the accuracy and energy consumption for different applications.

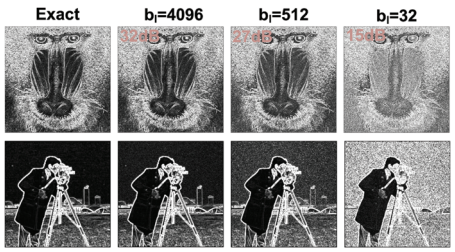


Fig. 12. Visualization of quality of computation in Sobel application, using different bit-stream lengths.

of the various techniques proposed in this work at application level. Of these configurations, COSMO-ALL is the best configuration and applies all the stochastic PIM techniques proposed in this work. As compared to COSMO-ALL, COSMO-PC, and COSMO-MUX do not implement the new addition technique proposed in Section 5.4 but use the conventional PC and MUX-based addition/accumulation, respectively. COSMO-NP implements all the techniques except the memory bitline segmentation, which eliminates block partitioning. Finally, COSMO-FX replaces the XNOR operation in COSMO-ALL with the XNOR implementation of [28].

Comparison of different COSMO Configurations: Comparing different COSMO configurations in Figure 13, we observe that COSMO-PC addition is on an average 240× and 647× slower than COSMO-ALL for DNNs and HD computing. This happens since COSMO-PC reads each and every data sequentially for accumulation as compared to COSMO-ALL which performs a highly parallel single cycle accumulation. This effect is seen very clearly in case of DNNs, where COSMO-ALL is 810× and 140× faster than COSMO-PC for AlexNet and VGG-16, both of which have large FC layers. On the other hand, COSMO-ALL is just 3.8× and 7.7× better than COSMO-PC for ResNet-18 and GoogleNet which have one fairly small FC layer each accumulating $\{512 \times 1,000\}$ and $\{1,024 \times 1,000\}$ data points. The latency of COSMO-MUX scales linearly with the bit-stream length. For our 32-bit DNN implemenatation, COSMO-ALL is 5.1× faster than COSMO-MUX. COSMO-ALL really shines over COSMO-MUX in the case of HD computing and is 188× faster. COSMO-MUX becomes a bottleneck in similarity check phase when the products for all dimensions need to be accumulated. COSMO-ALL provides the maximum theoretical speedup of 32× over COSMO-NP. In practice, COSMO-ALL is on an average 11.9× faster than COSMO-NP for DNNs. Further, COSMO-ALL is 20% faster and 30% more energy efficient than COSMO-FX for DNNs. This shows the benefits of COSMO over previous digital PIM operations.

Comparison with GPU for DNNs: COSMO benefits from three factors: simpler computations due to SC, high density storage and processing architecture, and less data movement between processor and memory due to PIM. From Figure 13, we observe that COSMO-ALL is on an average 141× faster than GPU for DNNs. COSMO latency majorly depends upon the convolution operations in a network. As discussed before, while COSMO parallelizes computations over input channels and weights depth in a convolution layer, the convolution of a weight window over an individual input channel still serializes the sliding of windows through the input. This means that the latency of a convolution layer in COSMO is directly proportional to its output size. It is reflected in the results where COSMO achieves higher acceleration in case of AlexNet (362×) and ResNet-18 (130×) as compared to VGG-16 (29×) and GoogleNet (41×). Here, even though ResNet-18 is deeper than VGG-16, its execution is faster because it reduces the size of the output of its convolution significantly faster than VGG-16. Also, COSMO-ALL is 80× more energy efficient than GPU. This is due

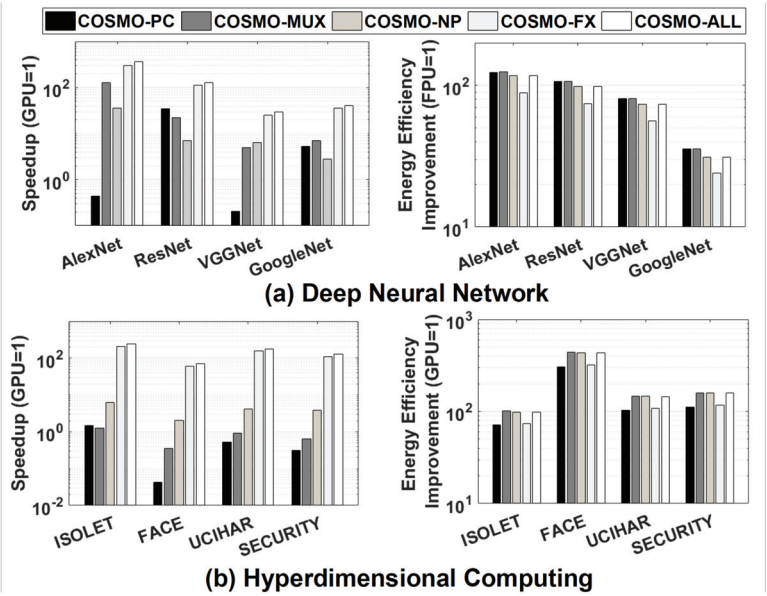


Fig. 13. Speedup and energy efficiency improvement of COSMO running (a) DNNs, (b) HD computing.

to the low-cost SC operations and the reduced data movement in COSMO, where the DNN models are pre-stored in memory.

To justify our experimental results, we also present a qualitative comparison between GPU and COSMO. Here, we consider the ideal performance of 11.34 TOPS/s for NVIDIA GTX 1080 Ti [2]. We account only for the core TDP of 250W of the GPU and not the entire system power. This translates to an ideal computational (power) efficiency of 10 GOPS/s/mm^2 (46 GOPS/s/W). In contrast, COSMO-ALL (in 45 nm process node) has computational (power) efficiency of 525 GOPS/s/mm^2 (908 GOPS/s/W). For a fair comparison, we normalized COSMO-ALL to 16 nm process node, same as GTX 1080 Ti. While scaling down, we consider area and power changes [84]. We do not reduce COSMO's latency since it depends on the switching behavior of ReRAM which may not scale sufficiently. We observe that the COSMO's computational and power efficiency increase to 4151 GOPS/s/mm^2 and 3681 GOPS/s/W, respectively. However, COSMO has on average 1.2% accuracy loss as compared to GPUs running in 32-bit integer representation, as shown in Table 2. Moreover, COSMO is not meant for DNN training, owing to its stochastic nature. Also, COSMO cannot perform online model re-training as GPU. Instead, we train DNN models on GPUs and load the trained and quantized model to COSMO. However, the data loading happens just once and is amortized over several test inputs. The time for loading inference model is common to both GPU and COSMO, and is excluded from our performance estimates.

Comparison with GPU for HD Computing: COSMO-ALL is on an average 156× faster than GPU for HD classification tasks. The computation in a HD classification task is directly proportional to the number of output classes. However, computation for different classes are independent from each other. The high parallelism (due to the dense architecture and configurable partitioning structure) provided by COSMO makes the execution time of different applications less dependent on the number of classes. However, in the case of GPU, the restricted parallelism (4,000 cores in GPU vs 10,000 dimensions in HD) makes the latency directly dependent on the number of classes. The energy consumption of COSMO-ALL scales linearly with classes while being on an average

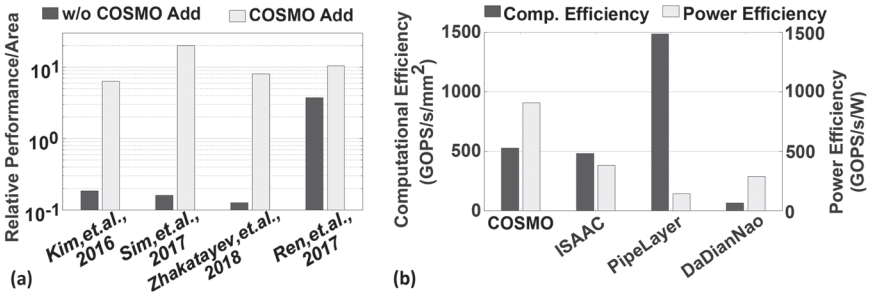


Fig. 14. (a) Relative performance per area of COSMO as compared to different SC accelerators with and without COSMO addition and (b) comparison of computational and power efficiency of running DNNs on COSMO and previously proposed DNN accelerators.

2,090× more energy efficient than GPU. This is majorly due to the reduced data movement in COSMO, where the huge class models are pre-stored in the memory. Moreover, HD computing consists of simple bitwise and addition operations. Unlike COSMO, GPU is not able to fully exploit the simplicity of operations that HD provides. GPU performs hundreds of thousands of MAC operations to implement HD dot product, whereas COSMO just uses simple XNORs and highly efficient accumulation.

8.4 COSMO vs Previous Accelerators

Stochastic Accelerators: We first compare COSMO with four state-of-the-art SC accelerators [43, 75, 81, 91]. To demonstrate the benefits of COSMO, we first implement the designs proposed by them on COSMO hardware. Figure 14(a) shows the relative performance per area of COSMO as compared to them. We present the results for two configurations. In the first, we use the same bitstream length and logic for multiplication, addition, and other functions that the corresponding accelerators use. In the second, we replace the additions and accumulations in all the designs with COSMO addition. Irrespective of the configuration, COSMO consumes 7.9×, 1,134×, 474×, 2,999× less area as compared to [43, 75, 81, 91], respectively. While comparing with previous designs in their original configuration, we observe that COSMO does not perform better than three of the designs [43, 81, 91]. The high area benefits provided by COSMO are overshadowed by the high latency addition used in these designs. It requires popcounting each data point either exactly or approximately, both of which require reading out data. Unlike previous accelerators, COSMO uses memory block as processing elements. Multiple data read-outs from a memory block need to be done sequentially, resulting in high execution times, with COSMO being on an average 6.3× and maximum 7.9× less efficient. Moreover, the baseline performance figures for these accelerators used to compare COSMO are optimized for small workloads which do not scale with the complexity and size of operations (a 200-input neuron for [43] and a $8 \times 8 \times 8$ MAC unit for [81, 91], while ignoring the overhead of SNGs). However, when COSMO addition is used for these accelerators, COSMO is on an average 11.5× and maximum 20.1× more efficient than these designs.

On the other hand, when the workload size and complexity is increased, as in case of SC-DNN [75] which implements LeNet-5 [52] neural network for MNIST dataset [51], COSMO is better than SC-DNN even in their original configuration, being 3.7× more efficient for the most accurate SC-DNN design. Further, when COSMO addition and accumulation is used on the same design, COSMO becomes 10.4× more efficient.

DNN Accelerators: We also compare the computational ($GOPS/s/mm^2$) and power efficiency ($GOPS/s/W$) of COSMO with state-of-the-art DNN accelerators [20, 77, 83]. Here, DaDianNao

[20] is a CMOS-based ASIC design, while ISAAC [77] and PipeLayer [83] are ReRAM based PIM designs. Unlike these designs, which have a fixed processing element (PE) size, the high flexibility of COSMO allows it to change the size of its PE according to the workload and operation to be performed. For example, a 3×3 convolution (2000×100 FC layer) is spread over 9 (2000) logical partitions, each of which may further be split into multiple physical partitions as discussed in Section 7.1. As a result, COSMO does not have theoretical figures for computational and power efficiency. However, to compare COSMO with these accelerators, we run the four neural networks shown in Table 2 on COSMO and report their average efficiency in Figure 14(b). We observe that COSMO is more power efficient than all DNN accelerators, being 3.2×, 2.4×, and 6.3× better than DaDianNao, ISAAC, and PipeLayer, respectively. This is due to three main reasons, reducing the complexity of each operation, reducing the number of intermediate reads and writes to the memory, and eliminating the use of power hungry conversions between analog and digital domains.

We also observe that COSMO is computationally more efficient than DaDianNao and ISAAC, being 8.3× and 1.1× better respectively. This is due to the high parallelism that COSMO provides, processing different input and outputs channels in parallel. COSMO is still 2.8× computationally efficient as compared to PipeLayer. It happens because even though COSMO parallelizes computation within a convolution window, it serializes sliding of a window over the convolution operation. On the other hand, PipeLayer makes a large number of copies of weights to parallelize computation within the entire convolution operation. However, computational efficiency is inversely effected by the size of accelerator, which makes the comparatively old technology node of COSMO an invisible overhead in computational efficiency. To give an intuition for the benefits which COSMO can provide, we scale all the accelerators to the same technology, i.e., 28 nm. DaDianNao and PipeLayer are already reported at 28 nm node. On scaling ISAAC and COSMO to 28nm, their computational efficiency increase to 625 $GOPS/s/mm^2$ and 1,355 $GOPS/s/mm^2$ respectively. This shows that COSMO can be as computational efficient as the best DNN accelerator while providing significantly better power efficiency.

Embedded Devices: We compare the power efficiency of COSMO with the state-of-the-art implementations of DNN inference on NVIDIA Tegra Jetson X1 GPU [65], FPGA [78], and Edge-TPU [3]. For the inference task on AlexNet, Tegra X1 (FPGA) achieves a power efficiency of 45 images/s/W (16 images/s/W), while COSMO achieves 506 images/s/W. For VGG-16, COSMO achieves power efficiency of 1,112 images/s/W as opposed to 66 images/s/W for Edge-TPU.

8.5 COSMO and Memory Non-Idealities

Bit-Flips: SC is inherently immune to singular bit-flips in data. COSMO, being based on it, enjoys the same immunity. Here, we evaluate the quality loss in COSMO with increase in the number of bit-flips. We evaluate the general applications with the same configuration as in Section 8.2 with a bit-stream length of 1024. The quality loss is measured as the difference between accuracy with and without bit-flips. Figure 15(a) shows that with 10% bit-flips, the average quality loss is meagre 0.27%. When the bit-flips increase to 25%, applications lose only 0.66% in accuracy.

Memory Lifetime: COSMO uses the switching of ReRAM cells, which are known to have low endurance. Higher switching per cell may result in reduced memory lifetime and increased unreliability. Previous work [29, 37, 50] uses an iterative process to implement multiplication and other complex operations. The more the iterations, higher is the number of operations and so is the per cell switching count. COSMO reduces this complex iterative process to just one logic gate, in case of multiplication, while it breaks down other complex operations into a series of simple operations. Hence, achieving less switching count per cell. Figure 15(b) shows that for multiplication, COSMO increases the lifetime of memory by 5.9× and 6.6× on an average as compared to APIM [37] and Imaging [29], respectively.

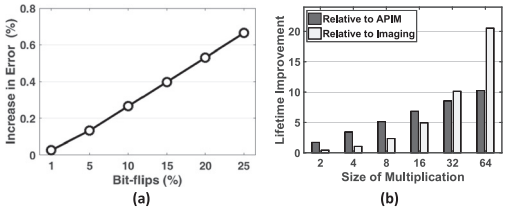


Fig. 15. COSMO's resilience to (a) memory bit-flips and (b) endurance.

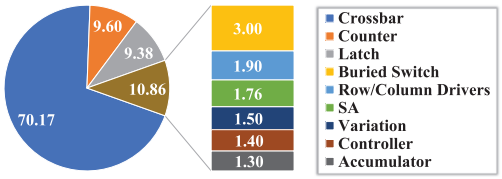


Fig. 16. COSMO's area breakdown.

8.6 Area Breakdown

COSMO completely eliminates the overhead of SNGs, which typically consume 80% of the total area in a SC system. However, the COSMO addition, which significantly accelerates SC addition and accumulation and overcomes the effect of slow PIM operations, requires significant changes to the memory peripheral circuits. Adding SC capabilities to the crossbar incurs ~26% area overhead to the design, as shown in Figure 16. This comes in the form of 3-bit counters (9.6%), 1-bit latches (9.38%), modified SAs (1.76%), and accumulators (1.3%). We use buried switches to physically partition a memory block, which contributes 3% to the total area overhead. Our variation aware SA tuning mechanism costs an additional 1.5% overhead. The remaining 73.4% of COSMO area is consumed by traditional memory components.

9 CONCLUSION

In this article, we proposed COSMO, a general in-memory processing architecture for SC on ReRAM. COSMO is a highly parallel architecture which scales with the size of SC. To achieve this, COSMO proposes a variety of novel techniques including, flexible block allocation, in-memory stochastic number generation, implication in memory, memory bitline segmenting, a new SC addition. It supports all SC encoding schemes and operations fully in memory. We implemented image processing, DNNs, and HD computing to show the generality of COSMO. Our evaluations over six general applications show that COSMO is 141× faster and 80× more energy efficient than GPU. COSMO illustrated that error-tolerant applications can benefit from the fast but not so precise SC.

REFERENCES

[1] 2016. Pytorch. Retrieved 1 Dec., 2020 from https://github.com/pytorch/pytorch.
[2] 2017. NVIDIA GTX 1080 ti specifications. Retrieved 1 Dec., 2020 from https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877.
[3] 2020. Edge TPU performance benchmarks. Retrieved 1 Dec., 2020 from https://coral.ai/docs/edgetpu/benchmarks.
[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 336–348.
[5] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* 17, 4 (2015), 2347–2376.
[6] Armin Alaghi and John P. Hayes. 2013. Survey of stochastic computing. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 2s (2013), 92.
[7] Armin Alaghi, Weikang Qian, and John P. Hayes. 2018. The promise and challenge of stochastic computing. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 37, 8 (2018), 1515–1531.
[8] Mohammad Abu Alsheikh, Shaowei Lin, Dusit Niyato, and Hwee-Pink Tan. 2014. Machine learning in wireless sensor networks: Algorithms, strategies, and applications. *IEEE Communications Surveys & Tutorials* 16, 4 (2014), 1996–2018.

[9] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. 2012. Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine. In *Proceedings of the International Workshop on Ambient Assisted Living*. Springer, 216–223.

[10] Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross. 2017. VLSI implementation of deep neural network using integral stochastic computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2688–2699.

[11] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.

[12] D. Blaauw, D. Sylvester, P. Dutta, Y. Lee, I. Lee, S. Bang, Y. Kim, G. Kim, P. Pannuto, Y.-S. Kuo, and D. Yoon. 2014. Iot design space challenges: Circuits and systems. In *Proceedings of the 2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*. IEEE, 1–2.

[13] Julien Borghetti, Gregory S. Snider, Philip J. Kuekes, J. Joshua Yang, Duncan R. Stewart, and R. Stanley Williams. 2010. Memristive switches enable stateful logic operations via material implication. *Nature* 464, 7290 (2010), 873–876.

[14] Bradley D. Brown and Howard C. Card. 2001. Stochastic neural computation. I. Computational elements. *IEEE Transactions on Computers* 50, 9 (2001), 891–905.

[15] Vincent Canals, Antoni Morro, Antoni Oliver, Miquel L. Alomar, and Josep L. Rosselló. 2016. A new stochastic computing methodology for efficient neural network implementation. *IEEE Transactions on Neural Networks and Learning Systems* 27, 3 (2016), 551–564.

[16] Liu Changyu. 2020. Retrieved from Alexnet-pytorch. https://pypi.org/project/alexnet-pytorch/.

[17] Liu Changyu. 2020. Retrieved from Googlenet-pytorch. https://pypi.org/project/googlenet-pytorch/.

[18] Liu Changyu. 2020. Retrieved from Resnet-pytorch. https://pypi.org/project/resnet-pytorch/.

[19] Liu Changyu. 2020. Retrieved from VGG-PyTorch. https://pypi.org/project/vgg-pytorch/.

[20] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[21] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[22] Ron Cole and Mark Fanty. 1994. *ISOLET - Data Set*. UCI Machine Learning Repository. Retrieved 1 Dec., 2020 from https://archive.ics.uci.edu/ml/datasets/ISOLET.

[23] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-memory data parallel processor. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1–14.

[24] Brian R. Gaines. 1967. Stochastic computing. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. ACM, 149–156.

[25] Kanad Ghose and Milind B Kamble. 1999. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*. ACM, 70–75.

[26] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The terasys massively parallel PIM array. *Computer* 28, 4 (1995), 23–31.

[27] Solomon W. Golomb. 1981. *Shift Register Sequences* (Revised ed.). Aegean Park Press, Laguna Hills.

[28] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7.

[29] Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, and Shahar Kvatinsky. 2018. Efficient algorithms for in-memory fixed point multiplication using MAGIC. In *Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[30] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the 2013 18th IEEE European Test Symposium (ETS)*. IEEE, 1–6.

[31] John P. Hayes. 2015. Introduction to stochastic computing and its challenges. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 59.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*. IEEE, 1026–1034.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[34] Sang-Min Hwang, Srinivasa Banna, Cathy Tang, Sunil Bhardwaj, Mayank Gupta, Tim Thurgate, David Kim, Jungtae Kwon, Joong-Sik Kim, Seung-Hwan Lee, and J. Y. Lee. 2011. Offset buried metal gate vertical floating body memory

technology with excellent retention time for DRAM application. In *Proceedings of the 2011 Symposium on VLSI Technology (VLSIT)*. IEEE, 172–173.
[35] Mohsen Imani. 2018. HD-IDHV. Retrieved 1 Dec., 2020 from https://Github.Com/Moimani/Hd-Idhv.
[36] M. Imani, J. Messerly, F. Wu, W. Pi, T. Rosing. 2019. A binary learning framework for hyperdimensional computing. In *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
[37] Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2017. Ultra-efficient processing in-memory for data intensive applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 6.
[38] Mohsen Imani, Deqian Kong, Abbas Rahimi, and Tajana Rosing. 2017. Voicehd: Hyperdimensional computing for efficient speech recognition. In *Proceedings of the 2017 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 1–8.
[39] Byung Chul Jang, Yunyong Nam, Beom Jun Koo, Junhwan Choi, Sung Gap Im, Sang-Hee Ko Park, and Sung-Yool Choi. 2018. Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics. *Advanced Functional Materials* 28, 2 (2018), 1704725.
[40] Sung Hyun Jo, Kuk-Hwan Kim, and Wei Lu. 2008. Programmable resistance switching in nanoscale two-terminal devices. *Nano Letters* 9, 1 (2008), 496–500.
[41] Pentti Kanerva. 2009. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation* 1, 2 (2009), 139–159.
[42] Arpan Kumar Kar. 2016. Bio inspired computing–a review of algorithms and scope of applications. *Expert Systems with Applications* 59 (2016), 20–32. https://www.sciencedirect.com/journal/expert-systems-with-applications/issues.
[43] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. 2016. Dynamic energy-accuracy tradeoff using stochastic computing in deep neural networks. In *Proceedings of the 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[44] Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. 2015. Approximate de-randomizer for stochastic circuits. In *Proceedings of the 2015 International SoC Design Conference (ISOCC)*. IEEE, 123–124.
[45] Yeseong Kim, Mohsen Imani, and Tajana Rosing. 2017. Orchard: Visual object recognition accelerator based on approximate in-memory processing. In *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 25–32.
[46] Phil Knag, Wei Lu, and Zhengya Zhang. 2014. A native stochastic computing architecture enabled by memristors. *IEEE Transactions on Nanotechnology* 13, 2 (2014), 283–293.
[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in Neural Information Processing Systems*. 1097–1105.
[48] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. 2014. Magic – memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.
[49] Shahar Kvatinsky, Misbah Ramadan, Eby G. Friedman, and Avinoam Kolodny. 2015. VTEAM: A general model for voltage-controlled memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 8 (2015), 786–790.
[50] Shahar Kvatinsky, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. 2014. Memristor-based material implication (IMPLY) logic: Design principles and methodologies. *Tvlsi* 22, 10 (2014), 2054–2066.
[51] Yann LeCun. 1998. The MNIST database of handwritten digits. Retrieved 1 Dec., 2020 from http://yann.Lecun.Com/exdb/mnist/.
[52] Yann LeCun, L. D. Jackel, Leon Bottou, A. Brunot, Corinna Cortes, J. S. Denker, Harris Drucker, I. Guyon, U. A. Muller, Eduard Sackinger, and P. Simard. 1995. Comparison of learning algorithms for handwritten digit recognition. In *Proceedings of the International Conference on Artificial Neural Networks*, Vol. 60. Perth, 53–60.
[53] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. 2013. Tiered-latency DRAM: A low latency and low cost DRAM architecture. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE, 615–626.
[54] Vincent T. Lee, Armin Alaghi, John P. Hayes, Visvesh Sathe, and Luis Ceze. 2017. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 13–18.
[55] Eero Lehtonen and Mika Laiho. 2009. Stateful implication logic with memristors. In *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures*. IEEE, 33–36.
[56] Bingzhe Li, M. Hassan Najafi, and David J. Lilja. 2016. Using stochastic computing to reduce the hardware requirements for a restricted boltzmann machine classifier. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 36–41.
[57] Fei-Fei Li, Marco Andreetto, Marc Aurelio Ranzato, and Pietro Perona. 2003. *Caltech101 Library*. Retrieved 1 Dec., 2020 from http://www.vision.caltech.edu/Image_Datasets/Caltech101.

[58] Peng Li, David J. Lilja, Weikang Qian, Kia Bazargan, and Marc Riedel. 2012. The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 480–487.

[59] Shuangchen Li, Alvin Oliver Glova, Xing Hu, Peng Gu, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2018. SCOPE: A stochastic computing engine for DRAM-Based in-situ accelerator. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 696–709.

[60] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[61] Gabriel H. Loh, Nuwan Jayasena, M. Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. 2013. A processing in memory taxonomy and a case for studying fixed-function pim. In *Proceedings of the Workshop on Near-Data Processing (WoNDP)*. 1–4.

[62] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Dominik Breitenbacher, Asaf Shabtai, and Yuval Elovici. 2018. *UCI Machine Learning Repository*. UCI Machine Learning Repository. Retrieved 1 Dec., 2020 from https://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT.

[63] Subhas Chandra Mukhopadhyay and Nagender K. Suryadevara. 2014. Internet of things: Challenges and opportunities. In *Proceedings of the Internet of Things*. Springer, 1–17.

[64] Anirban Nag, Rajeev Balasubramonian, Vivek Srikumar, Ross Walker, Ali Shafiee, John Paul Strachan, and Naveen Muralimanohar. 2018. Newton: Gravitating towards the physical limits of crossbar acceleration. *IEEE Micro* 38, 5 (2018), 41–49.

[65] Nvidia. 2015. GPU-based deep learning inference: A performance and power analysis. *Nvidia Whitepaper*.

[66] Alessandro Pappalardo. 2019. *Xilinx/brevitas*. Github.com. DOI:https://doi.org/10.5281/zenodo.3333552

[67] Behraoz Parhami and Chi-Hsiang Yeh. 1995. Accumulative parallel counters. In *Proceedings of the Conference Record of the 29th Asilomar Conference on Signals, Systems and Computers*, Vol. 2. IEEE, 966–970.

[68] Keshab Parhi and Yin Liu. 2016. Computing arithmetic functions using stochastic logic by series expansion. *IEEE Transactions on Emerging Topics in Computing* 7, 1 (2016), 44–59.

[69] Weikang Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja. 2011. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers* 60, 1 (2011), 93–105.

[70] Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, and Hai Li. 2018. Atomlayer: A universal reRAM-based CNN accelerator with atomic layer computation. In *Proceedings of the 55th Annual Design Automation Conference*. ACM, 103.

[71] Junfei Qiu, Qihui Wu, Guoru Ding, Yuhua Xu, and Shuo Feng. 2016. A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing* 2016, 1 (2016), 67.

[72] Abbas Rahimi, Sohum Datta, Denis Kleyko, Edward Paxon Frady, Bruno Olshausen, Pentti Kanerva, and Jan M. Rabaey. 2017. High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 9 (2017), 2508–2521.

[73] Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. 2016. A robust and energy-efficient classifier using brain-inspired hyperdimensional computing. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 64–69.

[74] Ravishankar Rao, J. Wence, Diana Franklin, Rajeevan Amirtharajah, and Venkatesh Akella. 2006. Exploiting non-uniform memory access patterns through bitline segmentation. In *Proceedings of the Workshop on Memory Performance Issues, in Conjunction with High Performance Computer Architecture (HPCA)*.

[75] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. 2017. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 405–418.

[76] Philipp Schulz, Maximilian Matthe, Henrik Klessig, Meryem Simsek, Gerhard Fettweis, Junaid Ansari, Shehzad Ali Ashraf, Bjoern Almeroth, Jens Voigt, Ines Riedel, and A. Puschmann. 2017. Latency critical iot applications in 5G: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine* 55, 2 (2017), 70–78.

[77] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[78] Masoud Shahshahani, Bahareh Khabbazan, Mohammad Sabri, and Dinesh Bhatia. 2020. A framework for modeling, optimizing, and implementing dnns on fpga using hls. In *Proceedings of the 2020 IEEE 14th Dallas Circuits and Systems Conference (DCAS)*. IEEE, 1–6.

[79] James M. Sibigtroth, George L. Espinor, and Bruce L. Morton. 2006. Memory bit line segment isolation. *US Patent 7,042,765*.

[80] Anne Siemon, Stephan Menzel, Rainer Waser, and Eike Linn. 2015. A complementary resistive switch-based crossbar array adder. *Jetcas* 5, 1 (2015), 64–74.

[81] Hyeonuk Sim, Dong Nguyen, Jongeun Lee, and Kiyoung Choi. 2017. Scalable stochastic-computing accelerator for convolutional neural networks. In *Proceedings of the Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 696–701.

[82] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[83] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 541–552.

[84] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81. https://www.sciencedirect.com/journal/integration/issues.

[85] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology* 15, 4 (2016), 635–650.

[86] Shibin Tang, Shouyi Yin, Shixuan Zheng, Peng Ouyang, Fengbin Tu, Leiyue Yao, JinZhou Wu, Wenming Cheng, Leibo Liu, and Shaojun Wei. 2017. AEPE: An area and power efficient RRAM crossbar-based accelerator for deep CNNs. In *Proceedings of the 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[87] Pai-Shun Ting and John Patrick Hayes. 2014. Stochastic logic realization of matrix operations. In *Proceedings of the 2014 17th Euromicro Conference on Digital System Design*. IEEE, 356–364.

[88] Christian Walczyk, Damian Walczyk, Thomas Schroeder, Thomas Bertaud, Małgorzata Sowinska, Mindaugas Lukosius, Mirko Fraschke, Dirk Wolansky, Bernd Tillack, Enrique Miranda, C. Wenger. 2011. Impact of temperature on the resistive switching behavior of embedded HfO_2-Based RRAM devices. *IEEE Transactions on Electron Devices* 58, 9 (2011), 3124–3131.

[89] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–488.

[90] Teng Xu, James B. Wendt, and Miodrag Potkonjak. 2014. Security of iot systems: Design challenges and opportunities. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 417–423.

[91] Aidyn Zhakatayev, Sugil Lee, Hyeonuk Sim, and Jongeun Lee. 2018. Sign-magnitude SC: Getting 10X accuracy for free in stochastic computing for deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. ACM, 158.

[92] Yaoxue Zhang, Ju Ren, Jiagang Liu, Chugui Xu, Hui Guo, and Yaping Liu. 2017. A survey on emerging computing paradigms for big data. *Chinese Journal of Electronics* 26, 1 (2017), 1–12.

Received August 2020; revised May 2021; accepted August 2021