Memristor Overwrite Logic (MOL) for In-Memory DNN

Khaled Alhaj Ali, Mostafa Rizk, Amer Baghdadi, Jean-Philippe Diguet, Jalal Jomaah

Memristor Overwrite Logic (MOL) for In-Memory DNN

Khaled Alhaj Ali[1,4], Mostafa Rizk[1,3,4], Amer Baghdadi[1], Jean-Philippe Diguet[2], and Jalal Jomaah[4]

[1]Institut Mines-Telecom, IMT Atlantique, Lab-STICC CNRS UMR 6285, Brest, France.
[2]Lab-STICC CNRS UMR 6285, Université de Bretagne-Sud, Lorient, France.
[3]International University of Beirut, Beirut, Lebanon.
[4]Physics Department, Faculty of Sciences, Lebanese University, Beirut, Lebanon.

***Abstract*—In-memory computing is a promising solution to address the memory wall challenges in future processing systems. Substantial improvement in performance and energy efficiency is expected, in particular for data intensive applications. A typical use case is neural network applications, where large amount of data should be processed and moved between memory and processing cores. Although several recent works tried to accelerate processing through dedicated parallel hardware designs, data movement cost is still a critical technical challenge. In this context, we propose a novel programmable architecture design for in-memory deep neural networks (DNN) computation. Based on a new logic design style, namely Memristor Overwrite Logic (MOL), specialized computational memory is designed. The original architecture of the proposed computational memory allows to execute multiply-accumulate operations between stored words using MOL. Outstanding features are demonstrated with respect to other recent logic design styles based on emerging non-volatile memory technologies.**

***Index Terms*—In-memory computing, Memristor, Deep neural network (DNN), Memristor Overwrite Logic (MOL).**

I. Introduction

Emerging and future IoT devices are expected to analyse raw data by running machine learning algorithms such as neural networks. Acquired data will be typically sent to the cloud for processing. However, this doesn't guarantee the required real-time response. Edge computing aims to get rid of the network latency by processing data locally. Yet, running an intensive workloads such as deep neural networks (DNN) on traditional cores (CPUs and GPUs) results in slow processing speed and high energy consumption. This is due to the intensive data movement between memory and processing cores. The idea of in-memory computing emerges to address this issue. Instead of sending large amount of data to the processing cores, in-memory computing allows the computation of a part of the tasks inside the memory. It reduces the memory accesses bottleneck and can result in significant performance increase. Several existing works have proposed the use of in-memory computing for accelerating the performance of neural network. This can be achieved by keeping the trained weights and the input data inside the memory [1]. The emerging memristor devices have recently shown its potential in the field of in-memory computing. Memristors are usually employed to build non-volatile memories (NVM) such as resistive RAMs. These memories allow for processing and storage within the same cells. Several techniques for in-memory computing in NVM have been investigated. In this context, the memristor aided logic (MAGIC) gate [2] and the memristor-based material implication (IMPLY) [3] have been recently explored to allow in-memory computations. In fact, IMPLY and MAGIC allow the execution of universal Boolean functions (Implication and NOR respectively) within the memristive elements. Based on these gates, storage and processing are both allowed within the same cells keeping the same topology of the memristive memory array. However, it is shown that the performance of such operations inside memory is highly affected by the parameters of the adopted memristive devices and other design constraints [3–5].

In this paper, we propose a novel programmable architecture design for in-memory DNN computation. The proposed architecture allows to execute any sequence of arithmetic tasks inside a specialized computational memory. In-memory addition and multiplication operations associated with the execution of a DNN weighted accumulation process is illustrated as a relevant case study. The computations are based on a new logic design style, namely Memristor Overwrite Logic (MOL) [6]. MOL operations are independent from memristor technology parameters and thus it is considered eligible for highly reliable applications.

We use the terms memristor and memristive device interchangeably for simplicity.

II. In-memory computing using MOL

A. Computing inside memristive crossbars

The non-volatile internal resistance state of a memristive device is dependent on the polarity and duration of the applied bias across its terminals [7]. Normally, a sufficient magnitude and duration of the bias lead to a full switching of the device to its boundary resistance state (R_{ON} or R_{OFF}). Otherwise, a partial switching occurs leading to a resistance state R where $R_{ON} < R < R_{OFF}$. This intermediate state is undesired in most digital memristive systems. However, if we succeed to guarantee a sufficient magnitude and duration of the applied bias, the resistance state of the memristive device can be considered as binary ($R \in \{R_{ON}, R_{OFF}\}$). Based on these considerations, we represent the current internal state Q_n of a memristive device in digital domain. The terminals A and B are modeled as binary input ports. The new internal state Q_{n+1} is function of the logical states at the terminals A, B and the previous internal state Q_n. By taking all the possible combinations of A, B and Q_n, the finite state machine (FSM) of a memristive device is proposed and demonstrated in Fig. 1. Based on this FSM representation, the state equation of a memristive device is expressed as follows:

$$Q_{n+1} = Q_nA + Q_n\overline{B} + A\overline{B} \tag{1}$$

The state representation expressed in (1) describes the computational as well as storage capability of a single memristive device. Six possible computational cases have been derived from (1) and are expressed in (2).

$$Q_{n+1} = \begin{cases} Q_n + A\ , & B = 0, \quad case: 1 \\ Q_nA\ , & B = 1, \quad case: 2 \\ Q_n + \overline{B}\ , & A = 0, \quad case: 3 \\ Q_n\overline{B}\ , & A = 1, \quad case: 4 \\ A\overline{B}\ , & Q_n = 0, \quad case: 5 \\ A + \overline{B}\ , & Q_n = 1, \quad case: 6 \end{cases} \tag{2}$$

The cases in (2) are split into two groups. The first group includes cases 1 to 4 which correspond to Memristor Overwrite Logic (MOL). In these 4 cases, a memristor acts as logic accumulator. The previously stored bit Q_n is subjected to OR/AND with the new input $A/\overline{B}$ while the other terminal of the memristor is set to logic "0" or logic "1" depending on the desired function. The obtained output is simultaneously saved in the form of new internal state Q_{n+1}. The rest cases (i.e. 5 and 6) are achieved by initializing the memristor to a specific state (logic "0" or logic "1"). These cases are considered as the conventional combinational logic between its terminals (A and B) of the memristor. The result of this combination is saved as the new internal state (Q_{n+1}) of the memristor.

The same concept applies inside memristive crossbar array. Fig. 2(a) illustrates cases 1 and 2. These cases are achieved within two steps. In step 1, the input vector $I = [I_{N-1}\ I_{N-2}...\ I_1\ I_0]$ is written into the N memristors by mapping logic "0" and logic "1" to the normalized voltage levels -1V and 1V respectively while the common horizontal line is set to 0V. At the end of this step, the resulting state of a given memristor M_k is $Q_k = I_k$. In step 2, the same N memristors are overwritten with the input vector $A = [A_{N-1}\ A_{N-2}...\ A_1\ A_0]$. However the input voltage level on the common horizontal line is set to 0V or 1V depending on the desired operation. For the case of MOL-OR, the horizontal line is set to 0V and the result stored in a given memristor M_k is $Q'_k = A_k + I_k$. For the case of MOL-AND, the horizontal line is set to 1V and the result which is stored in a given memristor M_k is $Q'_k = A_k I_k$.

Fig. 2(b) illustrates the use of case 5 inside crossbar array. The first step corresponds to initializing the crossbar to zeros. In the second step, an input vector A and an inverted vector B are fed to the columns and rows of the crossbar respectively. This combination results in a partial product of the two input vectors. The result is achieved in a single computational step.

In fact, the operations presented in Fig. 2 require specialized peripheral drivers to execute the process. Fig. 3 presents our proposed memristive crossbar architecture which can be configured to support the operations presented in 2. The architecture is presented in two models:

(i) 1M crossbar model: Each cell is formed of a single memristive device sandwiched between each horizontal (bitline BL) and vertical (wordline WL) nanowires.
(ii) 1T1M crossbar model: A single MOSFET transistor is added in series with each memristive device. The transistor is used as a selector to prevent undesired sneak paths which are encountered in the 1M memory models [8][9][10].

These architectures act similarly and could be configured in five modes:

(1) Write mode: An input N-bit vector is supplied to the crossbar memory by mapping logic "0" and "1" to the normalized voltage levels of $-1V$ and $1V$ respectively. The bitline driver block (BLD) takes over this mapping role, while the isolation block (ISO) acts as a connecting switch. Simultaneously, the address decoder selects a single WL. The selected WL is shared with $V_{SEL} = 0V$, while the unselected WLs are kept floating.
(2) Overwrite mode: The memory is configured to perform MOL-OR/MOL-AND inside its crossbar array. BLD maps the input data bits to the normalized voltage levels $0V$ and $1V$ corresponding to logic "0" and "1" respectively. The ISO block is kept in the connecting state. The address decoder selects a single WL. For the case of MOL-OR, the selected WL is shared with $V_{SEL} = 0V$, whereas for the case of MOL-AND, V_{SEL} is set to $1V$.
(3) Partial product mode: BLD maps the input data bits to the normalized voltage levels $0V$ and $1V$ corresponding to logic "0" and "1" respectively. The ISO block is kept in the connecting state. The MUX block passes the input vector to the WLs of the crossbar. Thus, the address decoder output is ignored.
(4) Read mode: The ISO block acts as an open switch keeping all BLs isolated from the BLD block. A single WL is selected by the address decoder to sense the individual states of its memristors. The sensing voltage is supplied by V_{SEL} and it is set to $0.5V$ (normalized).
(5) Idle mode: The crossbar array of the memory is totally isolated from BL side as well as WL side. The address decoder is disabled. Hence, the memory is not active.

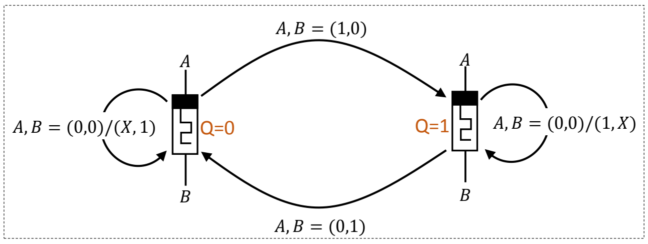


Fig. 1. Finite State Machine (FSM) of a memristive device.

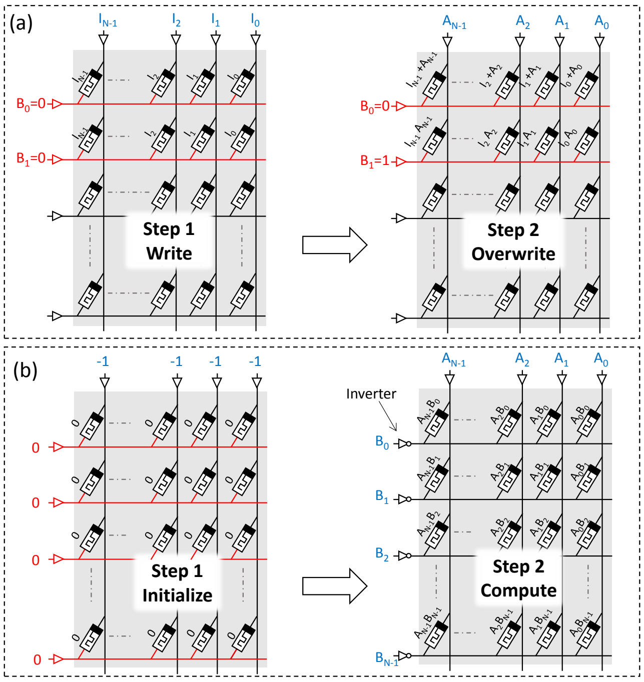


Fig. 2. Computation inside memristive crossbar: (a) MOL-OR or/and MOL-AND (b) Partial product.

B. Proposed computational memory architecture

The proposed MOL-memory architectures presented in Section II-A act as logic accumulators for the newly arriving bits. In

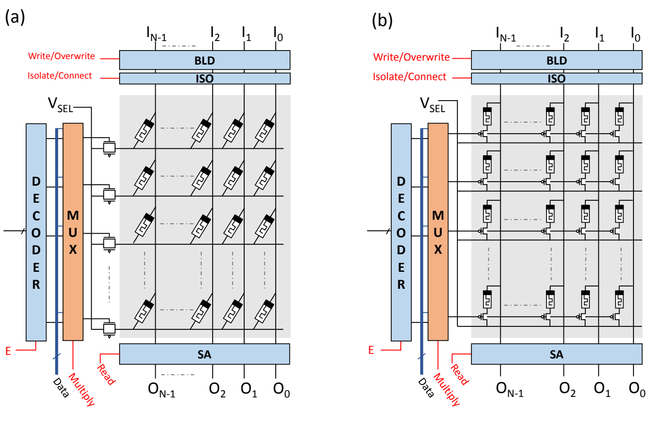


Fig. 3. MOL memory architecture: (a) 1M model and (b) 1T1M model.

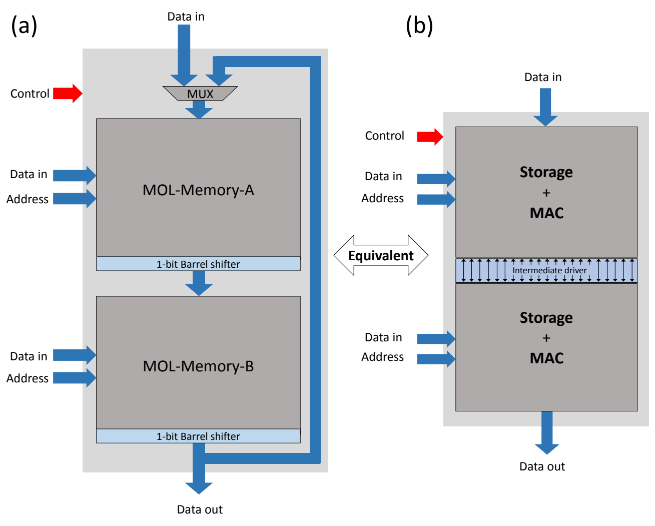


Fig. 4. MOL-based computational memory: (a) Architecture (b) Simplified diagram.

other words, computation in such memory is restricted for logic accumulation. Accordingly, performing general boolean functions in this memory requires an additional process to load the stored data bits outside the memory. These additional load operations are at odd with the concept of computation inside the memory. To overcome these load operations, we propose to use two coupled MOL memories (MOL-memory-A and MOL-memory-B), that work in complementary manner. At each time step, one of these memories acts as source of input data bits of the second memory. The second memory performs MOL with the previously stored bits in its memristive crossbar. Fig. 4(a) illustrates our proposed novel computational memory architecture. The architectures of MOL-memory-A and MOL-memory-B are identical. A controlled 1-bit barrel shift driver (BSD) is added after the sensing stages of the two memories to enable bit-level operations in addition to vector-level operations. Fig. 4(b) is a simplified diagram for the proposed architecture.

The proposed architecture presented in Fig. 4 has been verified by simulation using Cadence Virtuoso toolset based on the CMOS $65nm$ process. An 8×8 memristive crossbar is designed. The crossbar uses the Spin Transfer Torque Magnetic Memory (STT-MRAM) [11] which is based on the Magnetic Tunnel Junction (MTJ) memristive cell. The physical model describing the static, dynamic and stochastic behaviors of the adopted MTJ device is presented in [12][13]. An 8-bit addition has been carried out successfully on two arbitrary vectors inside the memory. The resulting sum is obtained after $6N+1$ computational steps which is equal to 49 for $N = 8$. Moreover, a partial product multiplication is performed between two arbitrary 8-bit vectors. The result is obtained within one computational step.

III. In-memory DNN computation

Deep neural network is a propular category of machine learning algorithms. Generally, it is presented as a network of interconnected neurons, containing an input layer, an output layer and one or more hidden layers. Fig. 6(a) presents an example of neural network with an input layer of M neurons, an output layer of three neurons and no hidden layers for simplicity. As illustrated in the figure, each output neuron executes a weighted accumulation of the input vector. Fig. 6(b) presents the multiply-accumulate (MAC) operation in a matrix form. The total number of weighted accumulations is proportional to the size of the network, i.e. size of input vectors and number of hidden layers. For large DNN, this represents a major challenge as it implies intensive data movement

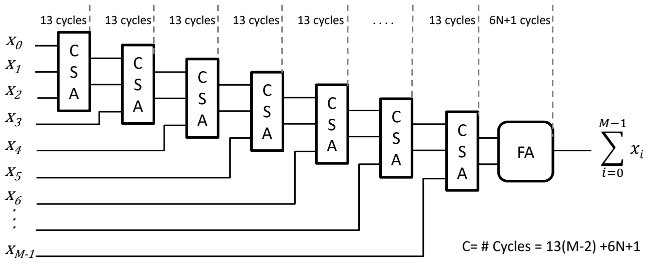


Fig. 5. Addition of M operands using tree-like CSA blocks.

between memory and processing cores. In fact, the cost of the multiply-accumulate operation and read/write memory accesses becomes considerable in terms of time and energy consumption [14].

In order to reduce this cost, we investigated the use of the proposed computational memory architecture to perform in-memory multiply-accumulate operations.

In-memory multiplication is achieved in two phases: (i) Phase 1 corresponds to the one-step partial product which is discussed in Section II-B and (ii) Phase 2 corresponds to the addition of the obtained partial product vectors inside the memory. Normally, an $N \times M$ multiplication requires addition of M partial products, each of size N bits, to generate a $(N+M)$-bit products. Knowing that each addition operation inside our proposed computational memory requires $6N + 1$ cycles, the total number of cycles to obtain the final product is $(M - 1)(6N + 1)$.

In order to reduce the required number of cycles, we use the method of carry save adder (CSA) to add multiple numbers together in tree structure inside the memory. CSA provides a 3:2 operands reduction, with a fixed latency irrespective of the size N of the operands. Fig. 5 represents a diagram showing the addition of M operands using tree-like CSA blocks. A single 3:2 addition inside the memory requires a latency of 13 cycles. The last stage is a classical 2:1 adder and requires $6N + 1$ cycles. Therefore, the overall latency required to add M operands is estimated as $C = 13(M-2)+6N+1$ cycles. By including the first two cycles required to obtain the partial products, a $N \times M$ multiplications inside the memory takes $C + 2$ cycles in total. By considering the example of the network presented in Fig. 6, it is possible to increase the level of parallelism to compute each output neuron separately, as they have no dependency to each others.

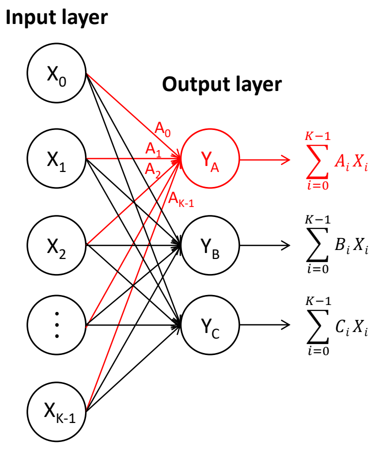
Fig. 7 presents the proposed architecture design for in-memory DNN computation, illustrated for the simplified neural network of Fig. 6. It is composed of interconnected computational memory blocks. Each block is in charge of the computation of one output neuron. The convolution of the input vector $[X_i]$ with the weight matrix $[A_i\ B_i\ C_i]$ is performed as follows. The weight matrix $[A_i\ B_i\ C_i]$ is supplied from the classical write path of the memory blocks while the input data vector $[X_i]$ is shared to all blocks along their rows. The outputs Y_A, Y_B and Y_C are executed simultaneously. The estimated total latency for the convolution operation is equal to $(C + 15)K + 6(N + M) + 1$ cycles.

IV. Discussions

MOL technique: Compared to the other existing in-memory computing techniques such as IMPLY [3] and MAGIC [2], our proposed MOL logic design style doesn't have any constraints on the technology of the adopted memristive device. In contrast, MAGIC and IMPLY require memristive devices that have sufficient margin between its low resistance states (LRS) and high resistance states (HRS).

Latency: When comparing a single addition operation with the works carried out in [3][4][15][16][17], MOL based in-memory

(a)



(b)

Input Data **Neuron Weights** **Output Equations**

$$[X_0 \;\; X_1 \dots \;\; X_{K-1}] * \begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ \vdots & \vdots & \vdots \\ A_{K-1} & B_{K-1} & C_{K-1} \end{bmatrix} = \begin{bmatrix} Y_A = X_0A_0 + X_1A_1 + \cdots X_{K-1}A_{K-1} \\ Y_B = X_0B_0 + X_1B_1 + \cdots X_{K-1}B_{K-1} \\ Y_C = X_0C_0 + X_1C_1 + \cdots X_{K-1}C_{K-1} \end{bmatrix}$$

Fig. 6. Example of a simplified neural network: (a) Network diagram (b) Matrix form representation.

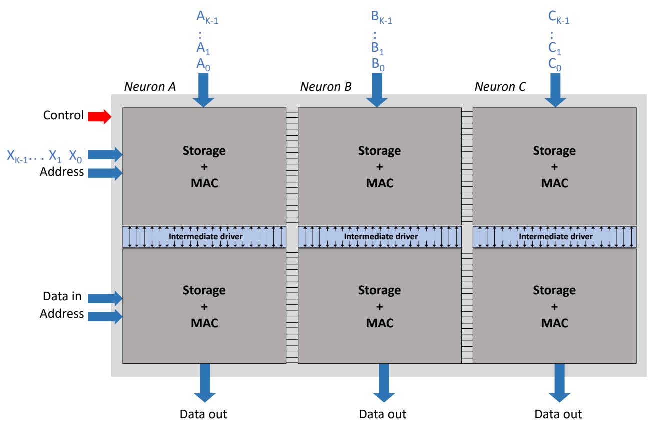


Fig. 7. Proposed design for in-memory DNN computation, illustrated for the simplified neural network of Fig. 6.

addition requires $6N+1$ cycles compared to the best case $(10N+3)$ achieved in [4].

Area overhead: In fact, the interconnected blocks in our proposed in-memory DNN computation architecture shares the same address decoder, MUX block and control signals. Therefore, the proposed design efficiently utilizes the peripheral circuits by sharing them to all sub-memory blocks on one hand, and between storage and computation operations on the other hand. This implies significant reduction in area overhead and simplifies the control. Moreover, the proposed architecture uses direct digital-based computation without any analog-to-digital conversion. This ensures a scalable design approach.

In addition, we remove the necessity of using the unreliable multi-level memristive devices. In fact, our design is based on binary memristive devices only.

V. Conclusion

This paper proposed a novel architecture design for in-memory DNN applications. The architecture is composed of interconnected specialized computational memory blocks. The design efficiently utilizes peripheral driving circuits which are shared between all memory blocks. The proposed architecture is programmable, allowing to execute any sequence of arithmetic tasks. A fully digital in-memory addition and multiplication operations associated with the execution of the DNN weighted accumulation process is illustrated as a relevant case study. This addresses the inefficiency of moving data between memory and processing cores which is time and energy consuming. For verification purpose, the architecture is simulated in Cadence virtuoso toolset based on CMOS $65nm$ process and a realistic model for the adopted MTJ device. Computation inside memory blocks are performed using MOL design style. MOL operations are independent from memristor technology parameters and thus it is considered eligible for highly reliable applications.

References

[1] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 27–39.

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

[3] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2014.

[4] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[5] X. Fang and Y. Tang, "Circuit analysis of the memristive stateful implication gate," *Electronics Letters*, vol. 49, no. 20, pp. 1282–1283, 2013.

[6] K. Ali, M. Rizk, A. Baghdadi, J.-P. Diguet, and J. Jomaah, "Crossbar Memory Architecture Performing Memristor Overwrite Logic," in *proc. of the International Conference on Electronics Circuits and Systems (ICECS)*, 2019.

[7] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, no. 1, p. 13, 2013.

[8] M. A. Zidan, H. A. H. Fahmy, M. M. Hussain, and K. N. Salama, "Memristor-based memory: The sneak paths problem and solutions," *Microelectronics Journal*, vol. 44, no. 2, pp. 176–183, 2013.

[9] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Sneak-path constraints in memristor crossbar arrays," in *proc. of the IEEE International Symposium on Information Theory (ISIT)*, 2013, pp. 156–160.

[10] S. Shin, K. Kim, and S.-M. Kang, "Analysis of passive memristive devices array: Data-dependent statistical model and self-adaptable sense resistance for RRAMs," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2021–2032, 2012.

[11] S. Ikeda, K. Miura, H. Yamamoto, K. Mizunuma, H. Gan, M. Endo, S. Kanai, J. Hayakawa, F. Matsukura, and H. Ohno, "A perpendicular-anisotropy CoFeB–MgO magnetic tunnel junction," *Nature materials*, vol. 9, no. 9, p. 721, 2010.

[12] Y. Wang, Y. Zhang, E. Deng, J.-O. Klein, L. A. Naviner, and W. Zhao, "Compact model of magnetic tunnel junction with stochastic spin transfer torque switching for reliability analyses," *Microelectronics Reliability*, vol. 54, no. 9-10, pp. 1774–1778, 2014.

[13] Y. Wang, H. Cai, L. A. Naviner, Y. Zhang, J.-O. Klein, and W. Zhao, "Compact thermal modeling of spin transfer torque magnetic tunnel junction," *Microelectronics Reliability*, vol. 55, no. 9-10, pp. 1649–1653, 2015.

[14] L. Fick and D. Fick, "Introduction to compute-in-memory," in *2019 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2019, pp. 1–65.

[15] P. Thangkhiew, R. Gharpinde, D. N. Yadav, K. Datta, and I. Sengupta, "Efficient implementation of adder circuits in memristive crossbar array," in *the proc. of the IEEE Region 10 Conference (TENCON)*, 2017, pp. 207–212.

[16] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta, "A scalable in-memory logic synthesis approach using memristor crossbar," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 2, pp. 355–366, 2018.

[17] P. L. Thangkhiew, R. Gharpinde, P. V. Chowdhary, K. Datta, and I. Sengupta, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *the proc. of the International Design & Test Symposium (IDT)*, 2016, pp. 142–147.