# Efficient Algorithms for In-Memory Fixed Point Multiplication Using MAGIC

Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, and Shahar Kvatinsky
Andrew and Erna Viterbi Faculty of Electrical Engineering
Technion - Israel Institute of Technology, Haifa, Israel 3200003
Email: {ameerh, rotembenhur, nimrodw}@campus.technion.ac.il, shahar@ee.technion.ac.il

*Abstract*— **The growing disparity between processor and memory performance poses significant limits on system performance and energy efficiency. To address this widely investigated problem, modern computing systems attempt to minimize data transfer by means of a memory hierarchy. Yet the benefit from such a solution for data-intensive applications is limited. Emerging non-volatile resistive memory technologies (memristors) offer the ability to both store and process data within the memristive memory cells, with almost no data transfer. In this paper, we propose algorithms for performing fixed point multiplication within the memristive memory using Memristor Aided Logic (MAGIC) gates and execute them in a cycle-accurate simulator to verify and evaluate them. Previously proposed implementations were not feasible for execution within memory because the required number of memory cells for the computation was too large to fit the size-limited memristive memory arrays. The algorithms proposed in this paper not only improve the latency as compared to previously proposed algorithms by 1.8× on average, but their significantly better area efficiency now makes it possible to perform numerous fixed point multiplications simultaneously within memristive memory arrays.**

## I. Introduction

Conventional computing systems are based on von Neumann architecture, where the data is stored in a memory but processed in a separate processing unit. Transferring the data between these different units is several orders of magnitude more expensive in terms of both energy and performance as compared to the computation itself [1], which is the primary bottleneck in data intensive applications. One promising approach to reduce the amount of data transfer is moving computation into the memory unit [2]. Real in-memory processing can be performed using novel memory structures based on emerging memristive technologies, such as Resistive RAM (RRAM) [3]. Memristive memory cells consist of resistive switches (namely, *memristors*), which change their resistance according to the voltage across them. Memristive technologies are considered as alternatives to DRAM and Flash, due to their high density, low power consumption, and good scalability [4]–[10].

A unique property of memristive memory cells is their ability to be used for both memory and logic [11]–[14]; no additional computation elements are needed and the data movement is minimal. One promising technique for executing in-memory computations is Memristor Aided loGIC (MAGIC) [15]. MAGIC can be used to execute NOR and NOT operations within a memristive memory array, in which the resistance of specific memory cells represents the inputs and outputs of logic gates at different stages of the computation. Since data is stored in RRAM as resistance, information can be stored and processed using the same cells, with no need for conversion, sensing or moving of data. These advantages have been the driving force behind many recent works on MAGIC and similar techniques [16]–[20]. An important feature of MAGIC is that when the inputs and outputs of different gates are located in the same row/column, the operation of all gates can be executed simultaneously in a single cycle. Applications that require the same instruction on multiple data in parallel are thus likely to benefit greatly from using MAGIC.

Digital image processing, fast Fourier transform [21], convolutional neural networks [22], and matrix multiplication are examples of data intensive applications that should benefit naturally from MAGIC since many data inputs are processed similarly in parallel. To simplify complicated multiplication operations, most of these applications depend on fixed-point (FiP) multiplication [22]–[33], which is unfortunately not properly supported by MAGIC yet. Support and optimization of FiP multiplication is therefore a crucial step in realizing the potential of MAGIC in these applications.

A previous attempt to implement FiP multiplication using MAGIC [34] concluded that its excessive latency and area preclude supporting it in size-limited memory arrays. Thus, the authors implemented FiP multiplication using standard CMOS logic in the periphery. This implementation requires reading the data from the memory array to the periphery, processing it, and writing it back to the memory, which involves data movement [2], one of the very problems that MAGIC is designed to solve.

In this paper, we argue that the MAGIC based FiP multiplication can be significantly improved to fit memristive memory arrays. This paper makes the following contributions:

- We propose two algorithms for efficient execution of FiP multiplication using MAGIC gates.
- We execute the proposed algorithms in a cycle-accurate simulator to verify and evaluate them.
- We show that our algorithms achieve on average 1.8× better latency and 23× better area efficiency than the previously proposed implementation [34], making it possible to perform numerous FiP multiplications simultaneously within acceptably sized memristive memory arrays.

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

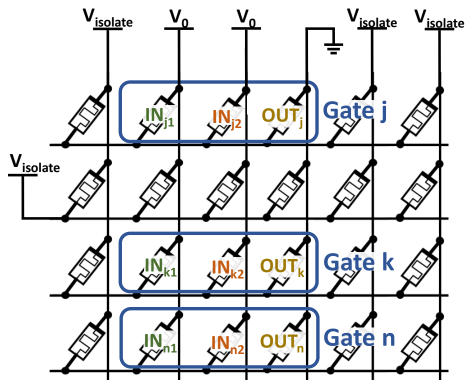

Fig. 1. Performing a MAGIC NOR operation within a memristive memory array. Three independent MAGIC NOR operations are executed in parallel on the first, third, and fourth rows (gates $j, k$ and $n$) by applying voltages as presented. All of the other cells are unselected and isolated.

## II. Background and Related Work

### A. Memristor Aided loGIC (MAGIC)

The logical state of each memristive memory cell is represented by resistance, where high and low resistances ($R_{OFF}$ and $R_{ON}$) are considered, respectively, as '0' and '1'. In stateful logic techniques [2] such as MAGIC, the inputs of the gates are the initially stored logical states of the input memristors, and the output is the logical state of the output memristor at the end of the computation. In MAGIC, NOR and NOT logic operations can be executed within the memory by applying specific voltages (*i.e.*, $V_0$ and *ground*) to the input(s) and output memristors, as shown in Figure 1. Note that MAGIC requires initializing the output memristors to logical '1' ($R_{ON}$) before the execution [15]. The MAGIC operation of all gates can be executed simultaneously in a single cycle when the inputs and outputs of different gates are co-located on the same row (wordline) or column (bitline). Any row on which we do not wish to perform the operation can be excluded by applying an isolation voltage to the corresponding row [17].

### B. Related Work

FiP multiplication is similar to integer multiplication but with an implied decimal point which allows having fractional results. It could thus be implemented using the partial products algorithm [35]. Previously, we proposed an algorithm to execute an $N$-bit adder using MAGIC [17] in $12N+1$ cycles. Imani *et al.* [34] implemented FiP multiplication by serializing similar adders after generating the partial products, requiring $15N^2 - 11N - 1$ cycles and $15N^2 - 9N - 1$ memristors. For the rest of the paper, we consider this algorithm to be the baseline.

The baseline algorithm requires numerous memristors for relatively small tasks. Particularly, the large number of required memristors does not permit the execution of FiP multiplication on a single row, where even 16-bit FiP multiplication requires more than 3700 consecutive memristors, much more than the number available in any reasonable memory array. Hence, Imani *et al.* concluded that in-memory FiP multiplication with MAGIC is impractical.

## III. The Proposed FiP Multiplication Algorithms

In this section, we describe two algorithms for efficient in-memory execution of FiP multiplication that offer substantial improvements over the baseline.

The proposed algorithms improve the latency of the execution (in terms of the number of cycles for the execution sequence), but more importantly, the area (determined as the number of memristors participating in the execution) is linearly dependent on the size (number of bits) of the inputs rather than the quadratic dependency in the baseline algorithm. The improvement in area makes it feasible to execute FiP multiplication in a single row, enabling massive parallelism within the memristive memory array.

### A. Full Precision FiP Multiplication

To multiply two numbers, we use the partial products multiplication algorithm and reuse the memristive cells during execution. For simplicity and without loss of generality, we assume two $N$-bit numbers, $A$ and $B$, stored in the same row ($A$ and $B$ are located in memristors 0 to $2N-1$) in the memristive memory array. The algorithm starts by initializing the memristors participating in the computation to $R_{ON}$. $A$ and $B$ are then negated to memristors at locations $2N$ to $4N-1$. After that, the partial products are generated and accumulated (using the latency optimized adder proposed in [17]) one by one in a repeated multiply-accumulate (MAC) manner using the same memristors. The entire computation is summarized in Algorithm 1. Figure $2(a,b,c)$ shows an example of this algorithm where $A = 010$ and $B = 001$.

The latency of the proposed algorithm is composed of $2N$ cycles to generate negated versions of $A$ and $B$, and $N-1$ MAC operations. Each MAC operation takes O($N$) cycles to complete [17], bringing the total number of cycles to O($N^2$). The area required for all the MAC stages is similar to the area required for a single add operation and a single partial product (due to the repeated use of the same memristors for computation), which is O($N$) [17]; together with the $4N$ memristors for storing $A$, $A'$, $B$ and $B'$, and $2N$ memristors for storing the final result, the total number of memristors is O($N$). The exact latency and area are summarized in Table I.

The numbers ($A$ and $B$) inside the memory array are assumed to be in the same row. However, if the two numbers are stored in different rows, they should be brought to the same row by negating each one in 1 cycle to the exact row (all the bits of each number are negated simultaneously). Note that while this adds up to 2 cycles to the latency, $2N$ cycles are actually saved by removing steps $2-4$ in the algorithm, which serially negate the two numbers bit after bit. Therefore, the expressions listed in Table I include the worst case scenario. Note that the two numbers might be located in different memory arrays and thus external data movement must be considered [36].

**Algorithm 1** Full Precision FiP Multiplication

```
// M_i = Memristor at location i
// M_{0 to N-1} = A, M_{N to 2N-1} = B
// M_{4N to 6N-1} = Final Result
1: $M_{2N \text{ to } 20N-5} \leftarrow R_{ON}$
   // Generate $A'$ and $B'$:
2: for $i = 0$ to $2N - 1$ do
3:     $M_{i+2N} \leftarrow NOT(M_i)$
4: end for
5: $M_{6N-1} \leftarrow$ NOR($M_{3N-1}$, $M_{4N-1}$)
   // Final Result_LSB ← NOR($A'_{LSB}$, $B'_{LSB}$)
6: for $j = 1$ to $N - 1$ do
7:     $M_{8N-j} \leftarrow$ NOR($M_{3N-1-j}$, $M_{4N-1}$)
       // First partial product_{j-1} ← NOR($A'_j$, $B'_{LSB}$)
8: end for
9: INTERMEDIATE_RESULT $\triangleq M_{7N+1 \text{ to } 8N-1}$
   /* INTERMEDIATE_RESULT refers to First partial product */
   // Perform $N-1$ MAC operations:
10: for $i = 1$ to $N - 1$ do
11:     for $j = 0$ to $N - 1$ do
12:         $M_{7N-1-j} \leftarrow$ NOR($M_{3N-1-j}$, $M_{4N-1-i}$)
            // $i^{th}$ partial product_j ← NOR($A'_j$, $B'_i$)
13:     end for
14:     if $i < N - 1$ then
15:         if $i$ mod $2 == 1$ then
16:             ($M_{8N \text{ to } 9N-1}$, $M_{6N-1-i}$) ←
                SUM($M_{6N \text{ to } 7N-1}$, INTERMEDIATE_RESULT)
                /* SUM($i^{th}$ partial product, INTERMEDIATE_RESULT) */
17:             INTERMEDIATE_RESULT $\triangleq M_{8N \text{ to } 9N-1}$
                /* INTERMEDIATE_RESULT refers to the new intermediate result */
18:             ($M_{6N \text{ to } 8N-1}$, $M_{9N \text{ to } 20N-5}$) ← $R_{ON}$
19:         else
20:             ($M_{7N \text{ to } 8N-1}$, $M_{6N-1-i}$) ←
                SUM($M_{6N \text{ to } 7N-1}$, INTERMEDIATE_RESULT)
                /* SUM($i^{th}$ partial product, INTERMEDIATE_RESULT) */
21:             INTERMEDIATE_RESULT $\triangleq M_{7N \text{ to } 8N-1}$
                /* INTERMEDIATE_RESULT refers to the new intermediate result */
22:             ($M_{6N \text{ to } 7N-1}$, $M_{8N \text{ to } 20N-5}$) ← $R_{ON}$
23:         end if
24:     end if
25: end for
26: $M_{4N \text{ to } 5N} \leftarrow$ SUM($M_{6N \text{ to } 7N-1}$,
    INTERMEDIATE_RESULT)
    /* Final Result_MSBs ← SUM(Final partial product, INTERMEDIATE_RESULT) */
```

## B. Limited Precision FiP Multiplication

The algorithm proposed in the previous subsection generates a result with twice the precision of the inputs ($2N$). However, in conventional systems, especially in digital signal processors (DSPs) [32], [33], the inputs and outputs are from the same type (precision). Hence, it is inefficient and unnecessary to generate a $2N$-bit result.

To limit the precision of the result to $N$ bits, we propose to perform a limited precision FiP multiplication, which modifies the previous algorithm by generating and accumulating only the necessary partial products, as illustrated in the example in Figure 2($d$). To generate only the necessary partial products, the algorithm decreases the size of partial product $i$ to $N - i$ bits by skipping the most significant $i$ bits when this partial product is generated. The reduced size partial products are accumulated in a MAC manner similarly to Algorithm 1. The new algorithm improves the latency by approximately 2×.

The exact latency and area are summarized in Table I. The benefits in latency come from decreasing the size of the partial products throughout the computation ($N, N-1, ..., 1$) rather than using a constant $N$-bits for each partial product, which reduces the total number of bits accumulated and generated in Algorithm 1 to half.

TABLE I
LATENCY AND AREA OF THE PROPOSED FIP MULTIPLICATION ALGORITHMS. $N$ IS THE NUMBER OF BITS IN EACH NUMBER.

| **Algorithm** | **Latency (Cycles)** | **Area (# of memristors)** |
|---|---|---|
| Full Precision | $13N^2 - 14N + 6$ | $20N - 5$ |
| Limited Precision | $6.5N^2 - 7.5N - 2$ | $19N - 19$ |

## IV. SIMULATION RESULTS

To verify the correctness of the proposed FiP multiplication algorithms, we implemented a functional simulator written in MATLAB that accurately performs the operations proposed in the algorithms cycle by cycle. The results confirmed the theoretical results.

To evaluate the latency and area (number of memristors participating in the computation) of the proposed FiP multiplication algorithms, we compare them to the baseline algorithm by Imani *et al.* [34]. Table II lists the latency and area results for FiP multiplication as a function of different numbers of bits (commonly used $N$-bit precisions) generated by the cycle-accurate simulator. The average improvement in latency is 1.2× for full precision and 2.4× for limited precision. This improvement is mainly attributed to the algorithm's ability to avoid adding unnecessary zeros before each addition operation of partial products as well as to its ability to generate two negated versions of the input numbers. The latter makes it possible to generate each partial product in $N$ cycles rather than $3N$. Additionally, the limited precision algorithm reduces the number of bits generated as partial products and accumulated to half. The benefits of this reduction are observed in the 2× improvement over the full precision algorithm.

Most of the savings in area are due to the proposed MAC operation, which allows the same memristors to be reused for all the partial products and add operations. The average improvement in area is 22× in full precision and 24× in limited precision; more importantly, area is linearly dependent

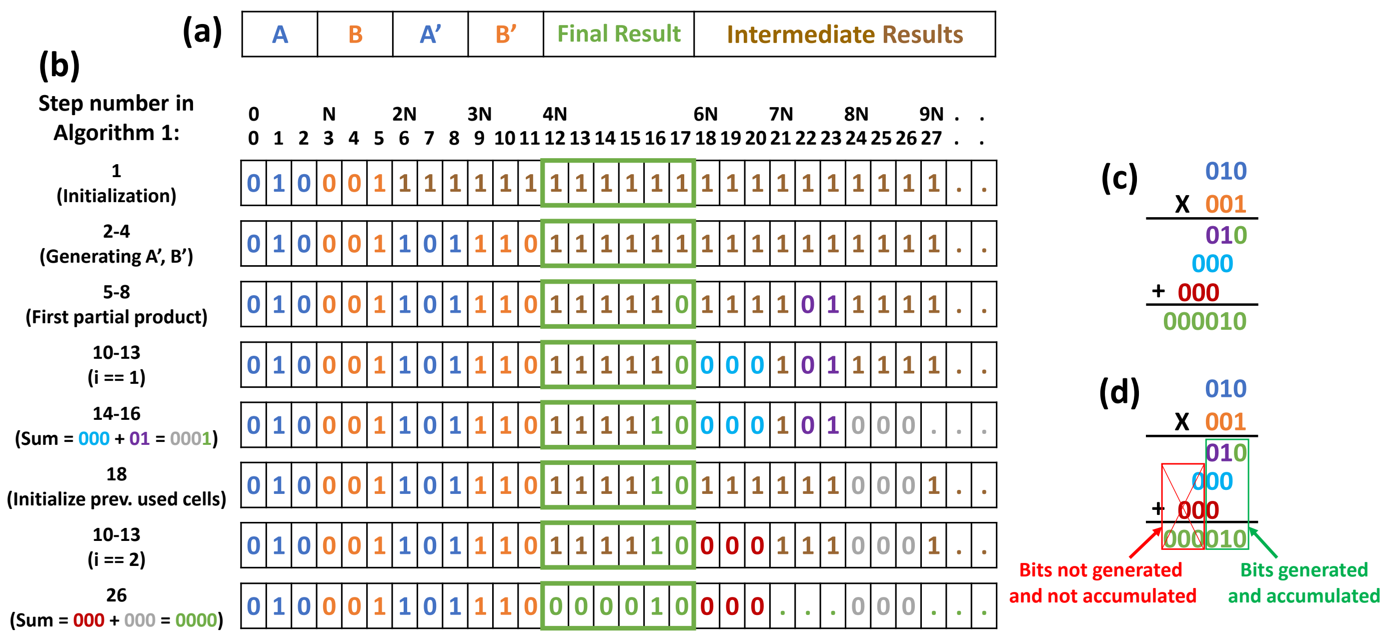
(a)
A
B
A'
B'
Final Result
Intermediate Results
(b)
Step number in Algorithm 1:
1 (Initialization)
2-4 (Generating A', B')
5-8 (First partial product)
10-13 (i == 1)
14-16 (Sum = 000 + 01 = 0001)
18 (Initialize prev. used cells)
10-13 (i == 2)
26 (Sum = 000 + 000 = 0000)
(c)
010
X 001
010
000
+ 000
000010
(d)
010
X 001
010
000
+ 000
000010
Bits not generated and not accumulated
Bits generated and accumulated

Fig. 2. Example of full precision FiP multiplication described in Algorithm 1 where $A = 010$ and $B = 001$. (a) The general structure of the processed row, (b) the same row during different steps of the execution, and (c) the multiplication of $A$ and $B$, and (d) its execution in limited precision FiP multiplication. Brown dots mean more logical '1's ($R_{ON}$), other dots mean the memristors participating in the SUM computations, and the colors of the bits illustrate what is being computed from the multiplication of $A$ and $B$.

TABLE II
LATENCY AND AREA RESULTS GENERATED BY A CYCLE-ACCURATE SIMULATOR FOR THE PROPOSED FIP MULTIPLICATION ALGORITHMS AS FUNCTION OF NUMBER OF BITS ($N$) COMPARED TO THE BASELINE ALGORITHM [34].

| | Latency (Cycles) | | | Area (# of memristors) | | |
|---|---|---|---|---|---|---|
| **N** | **Imani *et al.* [34]** | **Full Prec.** | **Limited Prec.** | **Imani *et al.* [34]** | **Full Prec.** | **Limited Prec.** |
| **8** | 871 | 726 | 354 | 887 | 155 | 133 |
| **16** | 3663 | 3110 | 1542 | 3695 | 315 | 285 |
| **32** | 15007 | 12870 | 6414 | 15071 | 635 | 589 |
| **64** | 60735 | 52358 | 26142 | 60863 | 1275 | 1197 |

on the number of bits rather than the quadratic dependency in the baseline algorithm.

The proposed algorithms assume the array size is sufficiently large to execute any given $N$-bit number multiplication. Nevertheless, the size of a memristive array is limited in practice, typically to $512 \times 512$ [37]. Since the prevalent FiP precision is 16-bit [32], [33], the required number of rows/columns in the memristive array (in the worst case algorithm) is therefore 315 rows/columns, which is compatible with state-of-the-art memristive memory arrays. This is in contrast to the baseline algorithm, where 3700 rows/columns are required. Hence, we conclude that the proposed algorithms enable in-memory FiP multiplication using MAGIC.

## V. Exploiting the Parallelism of MAGIC

In the proposed algorithms, the computation is done in a single row, which seemingly hinders reaching the full potential in terms of latency. Nevertheless, this approach has been chosen because aligning multiple inputs in multiple rows enables vector operations to be realized (multiplying multiple inputs simultaneously) with the same latency as a single multiplication. For example, for 16-bit limited precision FiP multiplication the latency is approximately 1500 cycles. While this value for a single multiplication is high, in memristive arrays of size $512 \times 512$, 512 multiplications could be performed simultaneously, effectively decreasing this latency to 3 cycles for a single multiplication and substantially improving throughput (number of executions per cycle).

## VI. Conclusions

In this paper, we propose novel algorithms that enable the efficient execution of FiP multiplication within a single row in memristive memory arrays using MAGIC. We envision that parallel execution of the algorithms will enable the efficient execution of more sophisticated data intensive applications such as the *Hadamard* product [27] and image convolution [26], which we seek to implement and evaluate in future work.

## Acknowledgment

This research is partially supported by the European Research Council under the European Union's Horizon 2020 Research and Innovation Programme (grant agreement no. 757259), by the Viterbi Fellowship at the Technion Computer Engineering Center, by the EU ICT COST Action IC1401, and by the Israel Science Foundation grant no. 1514/17.

## References

[1] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era," *IEEE Design Test*, vol. 34, no. 2, pp. 39–50, Apr. 2017.

[2] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. Haj-Ali, P.-E. Gaillardon, and S. Kvatinsky, "Memristive Logic: A Framework for Evaluation and Comparison," *27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2017.

[3] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, Feb. 2015.

[4] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "The Desired Memristor for Circuit Designers," *IEEE Circuits and Systems Magazine*, vol. 13, no. 2, pp. 17–22, May 2013.

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

[6] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[7] W. Woods, M. M. A. Taha, S. J. D. Tran, J. Burger, and C. Teuscher, "Memristor Panic - A Survey of Different Device Models in Crossbar Architectures," *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 106–111, Jul. 2015.

[8] J. Lee, M. Jo, D. Seong, J. Shin, and H. Hwang, "Materials and Process Aspect of Cross-Point RRAM," *Microelectronic Engineering*, vol. 88, no. 7, pp. 1113–1118, Jul. 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-addressable, Persistent Memory," *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp. 133–146, Oct. 2009.

[10] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, Mar. 2011.

[11] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-based imply logic design procedure," *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pp. 142–147, Oct. 2011.

[12] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor Based Computation-In-Memory Architecture for Data-Intensive Applications," *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1718–1725, Mar. 2015.

[13] L. Amaru, P. E. Gaillardon, and G. D. Micheli, "Majority-Inverter Graph: A New Paradigm for Logic Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, May 2016.

[14] E. Lehtonen, J. H. Poikonen, and M. Laiho, "Two Memristors Suffice to Compute All Boolean Functions," *Electronics Letters*, vol. 46, no. 3, pp. 239–240, Feb. 2010.

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

[16] R. Ben-Hur, N. Wald, N. Talati, and S. Kvatinsky, "SIMPLE MAGIC: Synthesis and In-memory MaPping of Logic Execution for Memristor-Aided loGIC," *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2017.

[17] 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, Jul. 2016.

[18] R. Ben-Hur and S. Kvatinsky, "Memory Processing Unit for In-Memory Processing," *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 171–172, Jul. 2016.

[19] 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, Oct. 2014.

[20] A. Raghuvanshi and M. Perkowski, "Logic Synthesis and a Generalized Notation for Memristor-realized Material Implication Gates," *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pp. 470–477, Nov. 2014.

[21] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*. Springer Science & Business Media, Dec. 2012, vol. 2.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25*, pp. 1097–1105, Dec. 2012.

[23] O. Catrina and A. Saxena, "Secure Computation with Fixed-Point Numbers," *Financial Cryptography*, vol. 6052, pp. 35–50, Jan. 2010.

[24] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 14–26, Jun. 2016.

[25] 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," *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, Jun. 2016.

[26] R. Jain, R. Kasturi, and B. G. Schunck, "Machine vision," vol. 5, Mar. 1995.

[27] R. A. Horn, "The Hadamard product," *Matrix Theory and Applications*, vol. 40, pp. 87–169, 1990.

[28] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 36–58, Sep. 2001.

[29] R. Zhen and R. L. Stevenson, "Image Demosaicing." Springer, 2015, pp. 13–54.

[30] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui, "Visual object tracking using adaptive correlation filters," *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2544–2550, Jun. 2010.

[31] B. Shen, I. K. Sethi, and V. Bhaskaran, "Dct convolution and its application in compressed domain," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 8, pp. 947–952, Dec. 1998.

[32] H. Choi, "Fixed Point Arithmetic," Aug. 2005. [Online]. Available: http://www.cpplab.net/web/dsp/Choi_Fixed-Point.pdf

[33] E. Bocchieri, "Fixed-point arithmetic," *Automatic Speech Recognition on Mobile Devices and over Communication Networks*, pp. 255–275, 2008.

[34] M. Imani, S. Gupta, and T. Rosing, "Ultra-Efficient Processing In-Memory for Data Intensive Applications," *Proceedings of the 54th Annual Design Automation Conference 2017*, Jun. 2017.

[35] C. R. Baugh and B. A. Wooley, "A Twoś Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, Dec. 1973.

[36] N. Talati, A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, P.-E. Gaillardon, and S. Kvatinsky, "Practical challenges in delivering the promises of real processing-in-memory machines," *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018.

[37] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, "Design Implications of Memristor-Based RRAM Cross-Point Structures," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, Mar. 2011.