IMAGING: In-Memory AlGorithms for Image processiNG

Ameer Haj-Ali, *Student Member, IEEE*, Rotem Ben-Hur, Nimrod Wald, Ronny Ronen, *Fellow, IEEE*, and Shahar Kvatinsky, *Senior Member, IEEE*

***Abstract*— Data-intensive applications such as image processing suffer from massive data movement between memory and processing units. The severe limitations on system performance and energy efficiency imposed by this data movement are further exacerbated with any increase in the distance the data must travel. This data transfer and its associated obstacles could be eliminated by the use of emerging non-volatile resistive memory technologies (memristors) that make it possible to both store and process data within the same memory cells. In this paper, we propose four in-memory algorithms for efficient execution of fixed point multiplication using MAGIC gates. These algorithms achieve much better latency and throughput than a previous work and significantly reduce the area cost. They can thus be feasibly implemented inside the size-limited memory arrays. We use these fixed point multiplication algorithms to efficiently perform more complex in-memory operations such as image convolution and further show how to partition large images to multiple memory arrays so as to maximize the parallelism. All the proposed algorithms are evaluated and verified using a cycle-accurate and functional simulator. Our algorithms provide on average 200× better performance over state-of-the-art APIM, a processing in-memory architecture for data intensive applications.**

***Index Terms*— von Neumann bottleneck, memristors, MAGIC, algorithms, processing in memory.**

I. Introduction

IN MODERN von Neumann systems, the data is stored in a memory but processed in a separate processing unit. Data transfer between these units incurs energy and delay several orders of magnitude greater than the energy and delay incurred by computation itself [1]. If the application is data intensive, the data transfer requirement will severely limit performance and energy efficiency. Often called the *memory wall*, this problem, already the primary bottleneck in modern computer systems, worsens with any increase in the distance data has to traverse. Minimizing data transfer and travel distance are thus crucial to improve performance and energy. Processing the data and storing it in a single unit by moving the computation to the memory itself is one promising avenue towards achieving this goal [2], [3].

Previous works with conventional memory technologies [4]–[6] added computational elements to the memory or modified the memory cells and periphery to add limited processing capability within conventional memories. However, true in-memory processing can be achieved using emerging memristive technologies, such as Resistive RAM (RRAM) [7]. RRAM cells consist of resistive switches (namely, *memristors*), which change their resistance according to the voltage across them. These memristive technologies are being explored as a replacement for DRAM and Flash, as they offer low power consumption, high density, non-volatility and good scalability [8], [9].

Memristors can be used for both memory and logic [10]–[12], thus eliminating the data movement. Memristor Aided loGIC (MAGIC) [13] is a promising logic-in-memory approach for executing in-memory computations. MAGIC enables the execution of NOR and NOT operations within a memristive memory array, where the inputs and outputs of logic gates at different stages of the computations are represented by the resistance of specific memory cells. Storing the data in RRAM as resistance allows information to be stored and processed using the same cells, with no need for conversion, sensing or moving of data. Much recent research on MAGIC and similar techniques has been conducted so to exploit this advantage [2], [14]–[17]. An important feature of MAGIC is its ability to execute numerous different gates simultaneously, when their inputs and outputs are located in the same row/column. MAGIC will thus greatly boost the performance of applications that require executing the same instruction on multiple data in parallel (SIMD).

This paper investigates one attractive set of applications for in-memory computation — digital image processing. Digital image processing is the field of analyzing and manipulating digitized images to obtain enhanced images or to extract useful information from them [18]–[26]. Manipulation of images requires data intensive computations, often in real-time, and the demand for data movement is only increasing as image resolution becomes higher. Therefore, image processing applications suffer from high energy consumption and a long processing time [27]. Digital image processing would benefit

Manuscript received March 3, 2018; revised April 22, 2018, May 18, 2018, and June 4, 2018; accepted June 5, 2018. Date of publication June 27, 2018; date of current version October 23, 2018. This work was supported in part by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259, in part by the Viterbi Fellowship at the Technion Computer Engineering Center, in part by the EU ICT COST Action IC1401, and in part by the Israel Science Foundation under Grant 1514/17. This paper was recommended by Associate Editor D. Comminiello. *(Corresponding author: Ameer Haj-Ali.)*

The authors are with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology, Haifa 32000, Israel (e-mail: ameerh@campus.technion.ac.il; rotembenhur@campus.technion.ac.il; nimrodw@campus.technion.ac.il; ronny.ronen@technion.ac.il; shahar@ee.technion.ac.il).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2018.2846699

naturally from MAGIC since many pixels are processed in parallel and the parallelism improves as the dimensions of the image increase. Computation with minimal data movement and massive parallelism within the memory is fundamentally different than standard computation. Hence, to boost the performance and energy efficiency in digital image processing tasks, new algorithms must be developed to efficiently map the data to the assigned memory cells and process the data within these cells. These algorithms must reshape the data to maximize the parallelism and perform the nontrivial task of converting a complicated application into an efficient sequence of NOR/NOT gates (*i.e.*, minimize the number of NOR/NOT operations and concurrently execute them).

Most digital image processing applications depend on fixed point (FiP) multiplication for its simplicity. Unfortunately, FiP multiplication is not properly supported by MAGIC yet. A previous work tried to implement FiP multiplication using MAGIC [28] but concluded that its excessive latency and area preclude supporting it in size-limited memory arrays. Thus, the authors proposed a processing in-memory architecture called APIM, which uses standard CMOS logic in the periphery to support FiP multiplication. This implementation requires reading the data from the memory array to the periphery, processing it, and writing it back to the memory; that is, it involves data movement, one of the very problems that MAGIC is designed to solve [29]. This functionality also makes the computation serial rather than utilizing the parallel execution ability that MAGIC is designed to support.

This paper makes the following contributions:

- We extend a previous work [30] and propose four algorithms for efficient execution of FiP multiplication using MAGIC gates. These algorithms enable FiP multiplication to be performed within acceptably sized memristive memory arrays.
- We demonstrate how to leverage the high throughput of these FiP multiplication algorithms to accelerate three common image processing applications: FiP dot product (FiPDP) [18], the *Hadamard* product [21], and image convolution [20]. For these three applications, we have developed algorithms that map and perform the desired tasks within the memristive memory array, exploiting the massive parallelism that MAGIC offers and minimizing the data transfer.
- We show how to scale the execution of the proposed image processing algorithms to efficiently process large images in the size-limited memory arrays, demonstrating on average 200× better performance over state-of-the-art APIM using a cycle-accurate, functional simulator.

The rest of the paper is organized as follows. Section II gives a short introduction to MAGIC, briefly discusses prior image processing works with memristors, and describes a previous, unsuccessful attempt to use it to implement FiP multiplication. In Section III, we propose four algorithms for efficient execution of FiP multiplication using MAGIC gates and evaluate them. We then demonstrate in Section IV how to use these FiP multiplication algorithms to efficiently implement three common image processing algorithms and further show how to scale the processing of large images. We evaluate these image processing algorithms in Section V and compare them to state-of-the-art APIM. The paper is concluded in Section VI.

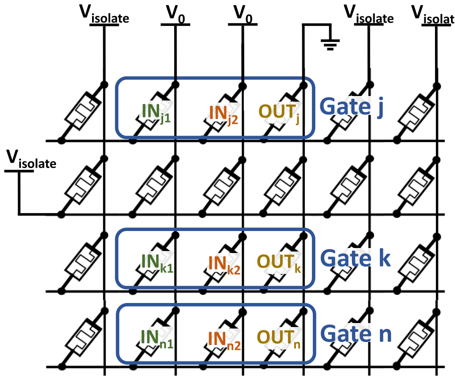


Fig. 1. Performing a MAGIC NOR operation within a memristive memory array. Three independent MAGIC NOR operations are executed in parallel on the first, third, and fourth rows (gates j, k and n) by applying voltages as presented. All of the other cells are unselected by applying isolation voltages.

II. Background

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 logical '0' and '1'. In stateful logic techniques [29] such as MAGIC [13], 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 completely within the memory without sensing or moving the data outside the memory array and without adding any additional, complicated logic to the periphery. This is done by applying specific voltages (*i.e.*, V_0 and *ground*) to the input and output memristors, as shown in Figure 1. Note that MAGIC requires a single cycle to initialize the output memristors to a low resistive state (logical '1', R_{ON}) before the execution begins. 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), thus allowing a high throughput of NOR operations inside the memory array.

B. Image Processing With Memristors

Previous attempts to accelerate image processing tasks with memristors [18], [19], [31] have relied on analog based computation using the accumulation of currents in analog-to-digital converters (ADCs) to perform sum of products. The input multipliers are represented as voltage while the multiplicands are stored as resistances in multi-level cells (MLC) [32]–[35]. While these approaches are efficient, they suffer from limited precision and reliability as compared to digital computation since the number of levels in MLC is limited and ADCs have limited accuracy. Therefore, digital computation is often necessary when precision is a concern. For example, digital image convolution is used in most digital camera image sensors to implement demosaicing of Bayer color arrays [23], [36]. In addition, because these approaches are restricted to predefined tasks that are based on the sum of products operation, they cannot be programmable. By contrast, since MAGIC NOR is functionally complete, any task could be mapped to a sequence of MAGIC NOR operations. Furthermore, applying different voltage levels requires large digital-to-analog converters, and sensing different current levels requires huge ADCs. This complex and considerable area overhead significantly restricts the area efficiency of the memory. Pinatubo [37] implements bit-wise operations in memristive memory arrays using standard CMOS logic in the periphery and thus requires moving the data serially to and from the periphery for every operation, which is impractical for image processing applications.

C. Fixed Point Multiplication With MAGIC

Fixed Point (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 [38]. Previously, we proposed an algorithm to execute an N-bit adder using MAGIC [15] in $12N + 1$ cycles (N is the number of bits required to represent a FiP number). Imani *et al.* [28] implemented FiP multiplication by serializing similar adders after generating the partial products, requiring $15N^2 - 11N - 1$ cycles and $15N^2 - 9N - 1$ memristors.

This implementation 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, far more than the number available in any reasonable memory array. This limitation makes parallelization of FiP multiplication impossible, rendering worthless one of the main advantages of MAGIC. Hence, Imani *et al.* concluded that in-memory FiP multiplication with MAGIC is impractical.

Instead, they implemented APIM, which uses standard CMOS logic in the periphery. APIM accelerates the FiP multiplication by using the periphery to speed up the generation of partial products and then applying a MAGIC-based fast carry-save adder. Two N-bit numbers are multiplied by reading the multiplier bit after bit (serially) in N cycles. After each bit whose value was ‘1’ is read, the multiplicand is copied in two MAGIC NOT cycles. When summed, the latency of generating partial products is N read cycles and $2N$ MAGIC cycles. Finally, the N N-bit partial products generated are accumulated and simultaneously added using the fast carry-save adder.

The latency of this adder is 285, 551, and 1057 MAGIC cycles for 8, 16, and 32-bit numbers, respectively. Although beneficial for performing a single FiP multiplication, APIM suffers from increased data movement between the memory array and its periphery. Moreover, multiple FiP multiplications can only be performed serially, since reading the data to the periphery is serial (for each FiP multiplication) and the fast carry-save adder uses multiple rows to compute. Thus, multiple fast carry-save additions are also performed serially (for each addition). These limitations make parallelization of FiP multiplication in APIM impossible.

In this paper, we show that it is not only possible to execute FiP multiplication inside an acceptably sized memory array but that by doing so it is also possible to realize complex image processing applications such as FiPDP, the Hadamard Product, and image convolution. We demonstrate a substantial improvement to the FiP multiplication algorithm of Imani *et al.* in terms of latency and area cost, and propose efficient and novel algorithms that take advantage of MAGIC’s innate parallelism to execute digital image processing tasks. These algorithms outperform APIM, proposed as an alternative to MAGIC.

III. Proposed Algorithms for Fixed Point Multiplication

In this section, we describe four algorithms for efficient in-memory execution of FiP multiplication [30] that offer substantial improvements over that of Imani *et al.* The proposed algorithms improve the latency of the execution (in terms of the number of cycles for the execution sequence). More importantly, they reduce the area (determined as the number of memristors participating in the execution) so that it 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 permits the execution of FiP multiplication in a single row, enabling massive parallelism within the memristive memory array.

A. Full Precision FiP Multiplication (FPFiPM)

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 [15]) one by one in a repeated multiply-accumulate (MAC) manner using the same memristors. The entire computation is summarized in Algorithm 1. A detailed example of the execution flow of this algorithm and further explanations are available in [30].

The latency of the proposed algorithm is composed of $2N$ cycles to generate negated versions of A and B, $N - 1$ initialization cycles, and $N - 1$ MAC operations. Each MAC operation takes $O(N)$ cycles to complete [15], bringing the total number of cycles to $O(N^2)$. The area required for 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 $O(N)$ [15]; together with the $4N$ memristors for storing A, A', B and B', and $2N$ memristors for storing the final result, the total

Algorithm 1 Full Precision FiP Multiplication (FPFiPM)

```
// $M_i$ = Memristor at location $i$
// $M_{0 \text{ to } N-1}$ = $A$, $M_{N \text{ to } 2N-1}$ = $B$
// $M_{4N \text{ to } 6N-1}$ = 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$_{\text{LSB}}$ ← 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 $i \bmod 2 == 1$ 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$_{\text{MSBs}}$ ← SUM(Final partial product,
   INTERMEDIATE_RESULT)                                   */
```

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 a shared row by negating each one in a single cycle to that 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.

TABLE I

LATENCY AND AREA OF THE PROPOSED FIP MULTIPLICATION ALGORITHMS. N IS THE NUMBER OF BITS IN EACH NUMBER. THE AREA REPRESENTS THE MINIMUM NUMBER OF MEMRISTORS REQUIRED IN A SINGLE ROW/COLUMN

Algorithm	**Latency (Cycles)**	**Area (# of memristors)**
FPFiPM	$13N^2 - 14N + 6$	$20N - 5$
LPFiPM	$6.5N^2 - 7.5N - 2$	$19N - 19$
Area Opt. FPFiPM	$16N^2 - 14N + 6$	$9N + 5$
Area Opt. LPFiPM	$8N^2 - 7.5N - 2$	$8N + 2$

B. Limited Precision FiP Multiplication (LPFiPM)

The algorithm proposed in the previous subsection generates a result with twice the precision of the inputs ($2N$). However, in classical integer multiplication, only the N least significant bits of the result are needed. Hence, generating only these N-bits will lead to more efficient execution without losing the required accuracy.

To limit the precision of the result to N bits, we propose to perform LPFiPM, which modifies the previous algorithm by generating and accumulating only the necessary partial products. 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 partial products are accumulated in a MAC manner similarly to Algorithm 1. The new algorithm improves the latency by approximately 2× at the cost of reduced precision. The exact latency and area are summarized in Table I. The benefits in latency come from the smaller size of the partial products throughout the computation ($N, N-1, ..., 1$), which reduces the total number of bits accumulated and generated in Algorithm 1 to half. This is in contrast to using a constant N-bits for each partial product.

C. More Reuse

The FPFiPM and LPFiPM algorithms rely on memristor reuse to optimize the area efficiency. In the current configuration, these algorithms reuse the same memristors to compute all the sums of partial products, which requires $13N - 5$ memristors. Increasing memristor reuse can further optimize the area efficiency. For example, every two partial products are added by serializing N single bit full adders. Therefore, we can minimize the area by optimizing the area efficiency of each such full adder and using the same memristors to compute all the full adders. In [15], an area-optimized full adder was proposed that uses five memristors but requires three additional cycles for repeated initialization. By deploying this full adder and using the same memristors to compute all the full adders, the number of reused memristors can be reduced to $2N + 5$ (rather than $13N - 5$). This would reduce the area of FPFiPM to $9N + 5$ memristors and the area of LPFiPM to $8N + 2$ memristors. The additional required initialization cycles would, however, increase the latency of FPFiPM to $16N^2 - 14N + 6$ and that of LPFiPM to $8N^2 - 7.5N - 2$. The exact latency and area of the area-optimized FiP multiplication algorithms are summarized in Table I. In Section III-E.3, we show how increasing memristor reuse decreases the lifetime of the memory.

TABLE II

LATENCY OF FIXED-POINT MULTIPLICATION ALGORITHMS FOR DIFFERENT NUMBERS OF BITS N. THE LATENCY FOR THE BASELINE ALGORITHM [28] IS COMPARED TO TO THE PROPOSED ALGORITHMS. THE VALUE IN PARENTHESES IS THE SPEEDUP AS COMPARED TO THE BASELINE

	Latency (Cycles)				
N	**Imani *et al.* [28]**	**FPFiPM**	**LPFiPM**	**Area Opt. FPFiPM**	**Area Opt. LPFiPM**
8	871	726 (1.20×)	354 (2.46×)	918 (0.95×)	450 (1.94×)
16	3663	3110 (1.18×)	1542 (2.38×)	3878 (0.95×)	1926 (1.90×)
32	15007	12870 (1.17×)	6414 (2.34×)	15942 (0.94×)	7950 (1.89×)
64	60735	52358 (1.16×)	26142 (2.32×)	64646 (0.94×)	32286 (1.88×)

TABLE III

AREA AS A FUNCTION OF THE NUMBER OF BITS N. THE AREA FOR THE BASELINE ALGORITHM [28] IS COMPARED TO THE PROPOSED FIP MULTIPLICATION ALGORITHMS. THE VALUE IN PARENTHESES IS THE IMPROVEMENT AS COMPARED TO THE BASELINE

	Area (# of Memristors)				
N	**Imani *et al.* [28]**	**FPFiPM**	**LPFiPM**	**Area Opt. FPFiPM**	**Area Opt. LPFiPM**
8	887	155 (5.7×)	133 (6.7×)	77 (11.5×)	66 (13.4×)
16	3695	315 (11.7×)	285 (13.0×)	149 (24.8×)	130 (28.4×)
32	15071	635 (23.7×)	589 (25.6×)	293 (51.5×)	258 (58.4×)
64	60863	1275 (47.7×)	1197 (50.8×)	581 (104.8×)	514 (118.4×)

TABLE IV

THROUGHPUT OF FIP MULTIPLICATIONS IN 1000 CYCLES AS A FUNCTION OF THE NUMBER OF BITS N, CONSIDERING A SINGLE MEMORY ARRAY OF SIZE 512 × 512. THE VALUE IN PARENTHESES IS THE SPEEDUP AS COMPARED TO APIM

	Throughput ($\frac{Multiplications}{1000\ Cycles}$)				
N	**APIM**	**FPFiPM**	**LPFiPM**	**Area Opt. FPFiPM**	**Area Opt. LPFiPM**
8	3.3	714 (216×)	1428 (433×)	558 (169×)	1138 (345×)
16	1.7	167 (98×)	333 (196×)	132 (82×)	266 (156×)
32	0.89	does not fit	does not fit	32 (36×)	64 (72×)

D. Single Row Execution

In the proposed multiplication algorithms, the computation is done in a single row, which seemingly may hinder reaching the full potential in terms of latency. Nevertheless, this approach is 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. The throughput is thus significantly improved, allowing more complex parallel applications to be realized. In Section IV, we leverage these vector operations to implement FiPDP, the Hadamard product, and image convolution.

E. Simulation Results

To verify the correctness of the proposed algorithms, we implemented a functional simulator written in MATLAB that accurately performs the logical flow of each algorithm cycle by cycle. Furthermore, we evaluated the latency, area, and throughput of the algorithms and compared them to previous works We likewise evaluated the endurance limitation of the memory array.

1) Latency and Area: 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.* [28]. Tables II and III list the latency and area results, respectively, for FiP multiplication as a function of different numbers of bits (N-bit commonly used precision) generated by the cycle-accurate simulator. The results show that the proposed algorithms not only improve the execution time of FiP multiplication but also substantially decrease the area relative to the baseline.

RRAM array sizes are limited due to sneak paths, IR drop across the parasitic resistances of the wires, parasitic capacitance across the wires, and the peak current the voltage drivers can drive. Reasonable sizes of RRAM arrays are 512 × 512 and 1024 × 1024 [39], [40]. The significant improvement in the area efficiency enables all 16-bit and 32-bit FiP multiplication algorithms to fit, respectively, in 512 × 512 and 1024 × 1024 arrays.

2) Throughput: The feasibility of performing FiP multiplication using MAGIC within a single row substantially increases the throughput. We compare the throughput of the proposed FiP multiplication algorithms with APIM. For an apples-to-apples comparison, we exclude the read latency in APIM's multiplier and consider only the MAGIC cycles, since the read latency in RRAM is usually smaller than the latency for performing MAGIC. To this end, for a single FiP multiplication we include only the $2N$ MAGIC cycles for generating the partial products and the delay of the MAGIC-based fast carry-save adder described in Section II-C. Note that the real latency of APIM is higher than that considered here, and thus the results shown here are more conservative than they would be if the read latency was included. To evaluate the throughput, we count the number of multiplications that could be executed in a single 512 × 512 array during 1000 cycles. Table IV summarizes the throughput of the proposed FiP multiplication algorithms as compared to APIM. The throughput is two to

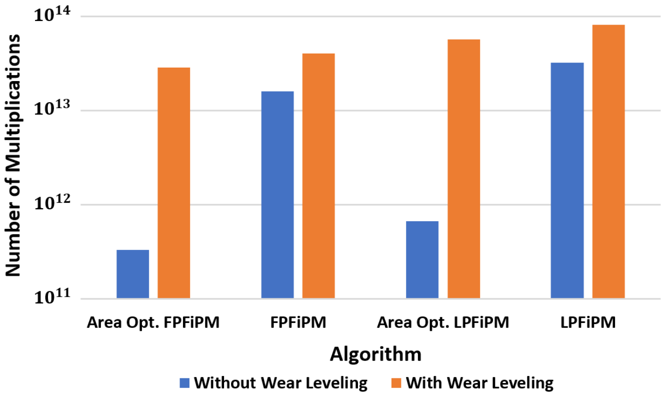


Fig. 2. Number of 16-bit FiP multiplications that could be executed in a 512 × 512 array before it wears out.

TABLE V

EXPRESSIONS FOR LATENCY AND AREA OF THE PROPOSED ALGORITHMS. N IS THE NUMBER OF BITS IN EACH NUMBER. H AND W ARE, RESPECTIVELY, THE HEIGHT AND WIDTH OF THE IMAGES ($H \times W$), AND P IS THE SIZE OF THE KERNEL IN CONVOLUTIONS ($P \times P$). *#Colors* IS 3 FOR RGB AND 1 FOR GRAY-SCALE IMAGES

Algorithm	Latency (Cycles)	Area ($\#rows \times \#columns$)
FiPDP	$13N^2 - 16N + 6 + \lceil LOG_2(H) \rceil \cdot (26N - 5) + H$	$(28N - 5) \times H$
Hadamard product	$W \cdot (13N^2 - 16N + 6)$	$(4NW + 16N - 5) \times H$
Image convolution	$WP \cdot (13N^2 + 32N - 4) - W \cdot (46N - 10) + \#colors \cdot H \cdot (P - 1)$	$(P \cdot (H + P - 1) \cdot \#colors) \times (5WN + PN + 21N - 5)$

three orders of magnitude better than that of APIM. This significant improvement is due to MAGIC's natural parallelism, which allows 512 (the number of rows in the array) multiplications to be performed simultaneously. In contrast, the throughput of APIM is limited, as it operates serially on the inputs. Increasing the array size will linearly increase the throughput of MAGIC while the throughput of APIM will remain constant.

3) Endurance Limitation: Using MAGIC NOR operations causes periodic changes in the output and intermediate memristor values, wearing out and stressing the memristors, and thus decreasing the lifetime of the memory. The endurance of state-of-the-art RRAM is currently limited to 10^{12} writes [41], [42]. The lifetime of the memory array is limited to the lifetime of the memristors used most frequently.

To precisely evaluate the usage of each memristive cell, we count the number of times each cell is written inside the cycle-accurate simulator. The memristors written most frequently are the ones used to repeatedly store the partial products or perform the add operations. There are $13N - 5$ such memristors in FPFiPM and LPFiPM, each used $2N$ times, and 5 such memristors in the area optimized versions proposed in Section III-C, each used $6N^2$ times. To maximize the number of multiplications before the memory array wears out, we leverage an approach similar to wear leveling [43], where the mapping of memory elements is changed periodically until most of the elements are near their end of life. To this end, we continuously remap the area used for storing inputs and intermediate results during the computation to different regions inside the memory array. This remapping does not add any performance overhead since it only changes the address (column/row) of the output when performing MAGIC.

Figure 2 shows the number of 16-bit FiP multiplications that can be performed inside a single 512 × 512 array before it wears out, for all the proposed algorithms, with and without wear leveling. This number is lower in the area-optimized algorithms even after considering wear leveling because these algorithms reuse area heavily to optimize area efficiency.

The numbers in the figure could be also used to determine the lifetime of the memory array in years. For example, if we assume a 100ns MAGIC gate delay of the memristor [18], and a continuous serial execution of $4 \cdot 10^{13}$ 16-bit FPFiPMs (3110 MAGIC cycles for a single FPFiPM, one row at a time) with wear leveling, memory wear-out could take 394.5 years. In contrast, when performing the same operations with full parallelism (3110 MAGIC cycles for a vector of 512 simultaneous FPFiPMs), memory wear-out will occur after only 0.8 years.

The endurance limitation of RRAM is a well-known challenge. In fact, the industry is working on improving the currently endurance-limited technology to increase the endurance and capacity of RRAM, and thus the lifetime of the memory. Some believe that the endurance will increase to at least 10^{15} [44]. This improvement will directly increase the lifetime of the memory array and similarly the number of operations it could perform before wear out. RRAM could ultimately replace DRAM as a result. With that being said, a direct solution to increase the lifetime of the memory array could currently be achieved by decreasing the parallelism or the utilization. The parallelism could be decreased by performing fewer MAGIC gates simultaneously at the cost of a linear decrease in performance. The utilization of each memory array could be decreased by performing MAGIC gates less frequently, for instance by mapping the task periodically to different memory arrays. In practice, memory array utilization will be less than 100% since the data will have to be stored before the computation starts and computing in all the arrays simultaneously is not always necessary since it depends on the data size.

IV. Image Processing Using MAGIC

Using the proposed vector FiP algorithms, we develop algorithms for efficient in-memory execution of three common image processing applications: fixed point dot product, Hadamard product, and image convolution. These image processing algorithms were chosen as they are data intensive and require element-wise operations that could be performed similarly and independently on all the pixels and thus could substantially benefit from the natural parallelism offered by MAGIC. The latency and area for all the proposed algorithms are summarized in Table V.

In all the algorithms, we assume a preprocessing stage (Initialization) where all data elements are stored properly for optimization purposes, and all the input data elements

Algorithm 2 Fixed Point Dot Product (FiPDP)

```
// Defines:
// M$_{(i,\ j)}$ = Memristor in row $i$ and column $j$
// N = Bit precision
// H = Height(input vector)
// Initialization:
// M$_{(0 \text{ to } H-1,\ 0 \text{ to } N-1)}$ = NOT(input vector 1)
// M$_{(0 \text{ to } H-1,\ N \text{ to } 2N-1)}$ = NOT(input vector 2)
1: M$_{(0 \text{ to } H-1, 2N \text{ to } 4N-1)}$ ←
     FPFiPM(M$_{(0 \text{ to } H-1, 0 \text{ to } N-1)}$, M$_{(0 \text{ to } H-1, N \text{ to } 2N-1)}$)
     // Multiply (element-wise) the two input vectors
2: $h = H/2$
3: for $i = 1$ to $\lceil LOG_2(H) \rceil$ do
4:     M$_{(h \text{ to } 2h-1, 4N \text{ to } 6N-1)}$ ←
       NOT(M$_{(h \text{ to } 2h-1, 2N \text{ to } 4N-1)}$)
       /* Negate bottom half of the result vector
       right                                          */
5:     M$_{(0 \text{ to } h-1, 4N \text{ to } 6N-1)}$ ←
       NOT(M$_{(h \text{ to } 2h-1, 4N \text{ to } 6N-1)}$)
       /* Negate the negated bottom half of the result
       vector up so it becomes aligned with the top
       half                                           */
6:     M$_{(0 \text{ to } h-1, 2N \text{ to } 4N-1)}$ ←
       SUM(M$_{(0 \text{ to } h-1, 2N \text{ to } 4N-1)}$, M$_{(0 \text{ to } h-1, 4N \text{ to } 6N-1)}$)
       /* Sum the top half and bottom half of the
       result vector                                  */
7:     $h = h/2$
8: end for
```

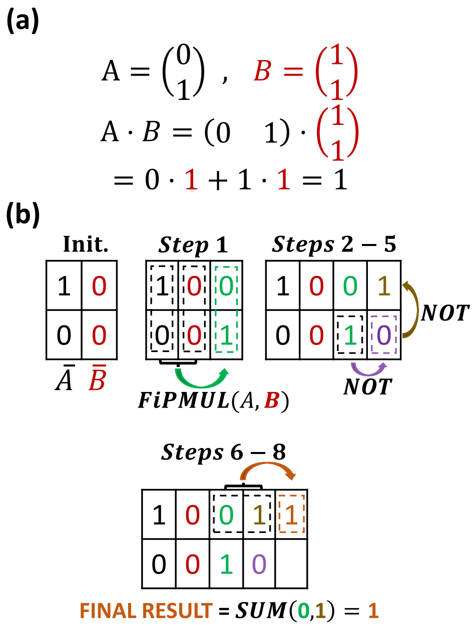


Fig. 3. Example of FiPDP as described in Algorithm 3 for two single-bit vectors $A = (0\ 1)$ and $B = (1\ 1)$. (a) The arithmetic result of the vector dot product multiplication, and (b) the different steps of the execution.

are stored negated. This assumption saves significant area and latency overhead in negating the inputs inside the FiP multiplication algorithms. Furthermore, we assume the data is properly aligned, *i.e.*, the multiplicands and multipliers are stored in a juxtaposed position (in the same row). In a system dedicated to image processing, these assumptions might be valid. Nevertheless, in Section IV-D we explore this preprocessing overhead in case these assumptions are not valid. For simplicity, we assume that the memory array size is not limited and can fit any input size. We address the memory array size limitation in Section IV-E. For the worst case scenario (in terms of area and latency), we assume that FiPDP, the Hadamard product and image convolution use FPFiPM. The other multiplication algorithms could be used for further optimization. Additionally, for adding two numbers we use the latency optimized adder [15]. Using the area-optimized adder [15] could further optimize the area efficiency.

A. Fixed Point Dot Product (FiPDP)

FiP dot product (FiPDP) is the fundamental operation in multiplying two matrices and the driving force behind many previous works on memristive logic for image processing [18], [19], [31]. In FiPDP, two vectors are multiplied, element by element, and the results are accumulated. Parallel multiplication and addition on all the rows is used to perform FiPDP between two $H \times 1$ vectors. The proposed method for calculating the FiPDP of two vectors is described in Algorithm 2. The algorithm starts by performing vector element-wise FPFiPM operations and then accumulates all the elements of the resulting vector by repeatedly splitting it into two similarly sized vectors, aligning the two vectors adjacently and summing them. Figure 3 shows an example of this algorithm being used to multiply two vectors of size 2×1 containing single-bit numbers.

The number of cycles required to complete the algorithm is the number of cycles required for a single FPFiPM operation (step 1) along with the $\lceil LOG_2(H) \rceil$ align (steps $3-5$) and sum operations (steps $6-8$). The number of cycles required for all the alignments is $O(H + N \cdot \lceil LOG_2(H) \rceil)$, since the size of the vector is halved in each iteration of the 'for' loop. The number of cycles required for all the sum operations is $O(N \cdot \lceil LOG_2(H) \rceil)$.

The complexity of the algorithm is therefore $O(N^2 + N \cdot \lceil LOG_2(H) \rceil + H)$, and the area consists of the area needed to store negated versions of the vectors and the area needed to store intermediate values before reaching the final result. The same intermediate memristors are reused for the FiP multiplication and all the align and add operations, bringing the total number of memristors to $O(HN)$.

B. Hadamard Product

The Hadamard product is frequently used in image processing tasks, *e.g.,* visual tracking, data compression and fast Fourier transform [25], [26]. In the Hadamard product, two images of the same dimensions are multiplied, element by element, producing another image of the same dimension as the original images.

Parallel multiplication on all the rows is used to perform the Hadamard product between two $H \times W$ images. The proposed method for calculating the Hadamard product of two images is implemented in Algorithm 3. The algorithm iterates over the equivalent columns in the input images and performs vector FiP multiplication between these columns. In Figure 3 (b)-step 1, two vectors of size $H = 2$ and $W = 1$ are

Algorithm 3 Hadamard Product

```
// Defines:
// $M_{(i,j)}$ = Memristor in row $i$ and column $j$
// N = Bit Precision
// H = Height(input image)
// W = Width(input image)
// Initialization:
// $M_{(0 \text{ to } H-1,\ 0 \text{ to } N \cdot W-1)}$ = NOT(input image 1)
// $M_{(0 \text{ to } H-1,\ N \cdot W \text{ to } 2N \cdot W-1)}$ = NOT(input image
   2)
1: for $i = 0$ to $W - 1$ do
2:    $M_{(0 \text{ to } H-1,\ 2N \cdot (W+i) \text{ to } 2N \cdot (W+i+1)-1)} \leftarrow$
      FPFiPM($M_{(0 \text{ to } H-1,\ N \cdot i \text{ to } N \cdot (i+1)-1)}$,
      $M_{(0 \text{ to } H-1,\ N \cdot (W+i) \text{ to } N \cdot (W+i+1)-1)}$) /* Multiply
      (element-wise) two matching input vectors    */
3: end for
```

Algorithm 4 Image Convolution

```
// Defines:
// $M_{(i,j)}$ = Memristor in row $i$ and column $j$
// N = Bit Precision
// H = Height(input image), Hp = H + P - 1
// W = Width(input image), Wp = W + P - 1
// P = height(input kernel) = width(input kernel)
// $(R, G, B) = (0, 1, 2)$
// c = column, e = element, clr = color, img =
   image
// Initialization:
/* P negated copies of each input color image are
   stored one below the other:
   $M_{(0 \text{ to } P \cdot Hp-1,\ 0 \text{ to } N \cdot Wp-1)}$ = NOT(P copies of
   input red image)
   $M_{(P \cdot Hp \text{ to } 2P \cdot Hp-1,\ 0 \text{ to } N \cdot Wp-1)}$ = NOT(P copies
   of input green image)
   $M_{(2P \cdot Hp \text{ to } 3P \cdot Hp-1,\ 0 \text{ to } N \cdot Wp-1)}$ = NOT(P copies
   of input blue image)                              */
/* 3Hp negated kernels are centered across unique
   lines in the image:
   $M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot Wp \text{ to } N \cdot (Wp+P)-1)}$ = NOT(3Hp
   copies of input kernel)                           */
1: for $c = \lfloor P/2 \rfloor$ to $W + \lfloor P/2 \rfloor$ do
2:    for $j = -\lfloor P/2 \rfloor$ to $\lfloor P/2 \rfloor$ do
3:       $M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+2c) \text{ to } N \cdot (Wp+2c+2)-1)} \leftarrow$
         FPFiPMAC($M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (c+j) \text{ to } N \cdot (c+j+1)-1)}$, $M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+\lfloor P/2 \rfloor+j) \text{ to } N \cdot (Wp+\lfloor P/2 \rfloor+j+1)-1)}$)
4:    end for
5: end for
   // Negate the result image horizontally:
6: for $c = 0$ to $2N \cdot W - 1$ do
7:    $M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+P+2W)+c)} \leftarrow$
      NOT($M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+P)+c)}$)
8: end for
9: for $e = 1$ to $P - 1$ do
10:   for $clr = R, G, B$ do
11:      for $img = 0$ to $P - 1$ do
12:         for $row = img + e;\ row < Hp;\ row \mathrel{+}= P$ do
13:            $M_{((clr+img) \cdot Hp+row-e,\ N \cdot (Wp+P+2W) \text{ to } N \cdot (Wp+P+4W)-1)}$
               $\leftarrow$ NOT($M_{((clr+img) \cdot Hp+row,\ N \cdot (Wp+P+2W) \text{ to } N \cdot (Wp+P+4W)-1)}$)
               // Negate one negated result row up
14:         end for
15:      end for
16:   end for
    // Accumulate the matching vectors:
17:   for $c = 0$ to $W - 1$ do
18:      $M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+P+2c) \text{ to } N \cdot (Wp+P+2c+2)-1)} \leftarrow$
         SUM($M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+P+2c) \text{ to } N \cdot (Wp+P+2c+2)-1)}$,
         $M_{(0 \text{ to } 3P \cdot Hp-1,\ N \cdot (Wp+P+2W+2c) \text{ to } N \cdot (Wp+P+2W+2c+2)-1)}$
19:   end for
20: end for
```

multiplied (element-wise); thus it is equivalent to performing the Hadamard product between these two vectors.

The number of cycles required to complete the algorithm is the number of cycles required for a single FiP multiplication (step 2) multiplied by the number of columns (in the 'for' loop). The complexity of the algorithm is therefore O(N^2W), and the area consists of the area required for storing negated versions of the images and the area required for storing intermediate values before the final results are reached. The same intermediate memristors are reused for all the FiP multiplications, bringing the total number of memristors to O(HWN).

C. Image Convolution

Image convolution is a popular operation in different image processing tasks such as smoothing, filtering, and edge detection. It is useful in image recognition and manipulation, medical imaging and convolutional neural networks [22]–[24]. In image convolution, a kernel is slided over an image and its values are multiplied by the corresponding pixel values of the image.

Algorithm 4 performs the convolution of an $H \times W$ image represented in RGB (by an image for each color, with numbers represented with $N = 8$ bits [45]) and a $P \times P$ filtering kernel. The proposed algorithm is simultaneously executed for all color images (*i.e.*, red, blue and green), either in the same array or in different arrays, thus efficiently exploiting the parallelism of MAGIC NOR gates. The initial stage (Initialization), where negated kernels and negated images are aligned, is shown in Figure 4(a). In steps $1-5$, all pixels in each column participate simultaneously in a FiP multiply and accumulate (FiPMAC) operation with the equivalent column in the kernels. This is repeated for all the columns. In steps $6-8$, the remaining results for each pixel are negated to the closest available memristors (*e.g.*, the entire column is negated to the right). Finally, in steps $9-20$, the remaining results are negated up (steps $9-16$) and accumulated (steps $17-20$) to generate the final result. An example of the execution of the algorithm is shown in Figure 4 (for simplicity it is only shown for a red image).

The number of cycles required to execute the algorithm is determined by the number of cycles it takes to (1) perform parallel multiplications and accumulations of columns in the data structure presented in Figure 4(a), (2) align the remaining results, and (3) accumulate them. Therefore, the number of cycles required is O($N^2PW + PH$), which means that the latency is linear with the largest dimension of the image. Most of the memristors are used for storing the negated versions of the image and kernel. The memristors used for processing and storing intermediate values occupy a much smaller percentage of the total area as the algorithms become more complex. When summed, the number of required memristors

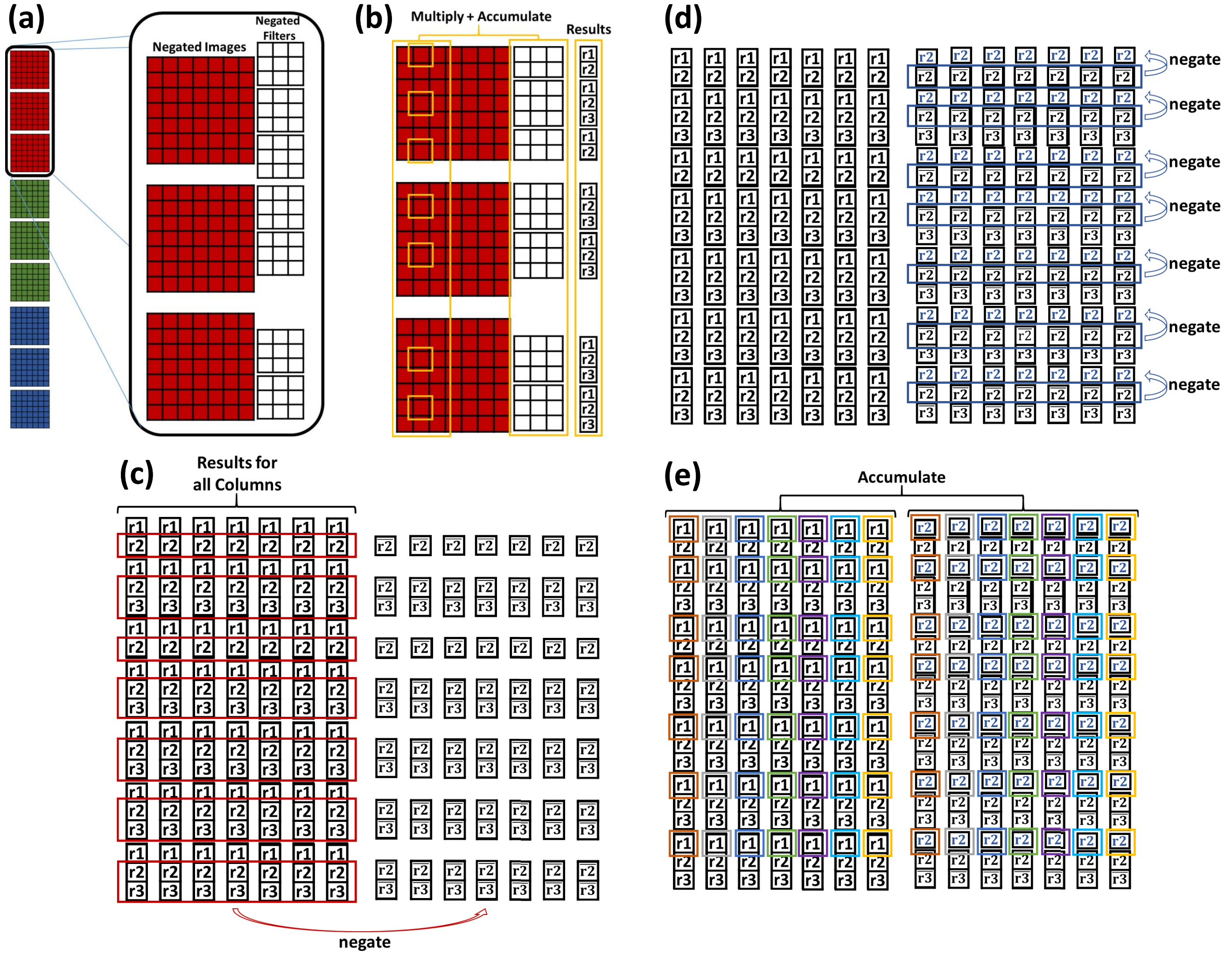


Fig. 4. Example of executing Algorithm 4 on a 7 × 7 image convolution with a 3 × 3 kernel. (a) The initial stage (Initialization). The negated kernel is duplicated 7 times to cover a unique row of the image, such that all the rows are covered. Kernels are negated opposite to the 1st, 4th and 7th rows of the top image, the 2nd and 5th rows of the middle image, and the 3rd and 6th rows of the bottom image. This duplication is repeated for the green and blue images as well. (b) A single iteration of steps 1 − 5. (c) The results are negated to the right columns (steps 6 − 8). (d) The results are negated up to align them with the equivalent numbers that (e) are eventually summed. The steps of (d) and (e) are repeated until all the results are accumulated to generate the output image.

is O($PNHW$). In other words, the area is proportional to the size of the image.

In some cases, *e.g.*, MNIST [46], images are presented in gray scale rather than RGB. In such cases, the area cost is reduced by 3×. However, the improvement in latency is negligible since the proposed algorithm processes all color images simultaneously.

D. Initialization Overhead

The proposed algorithms assume that data is located within the memory in the desired location before the beginning of the execution (Initialization stage). For example, in the Hadamard product the images are stored negated and aligned in the same rows and columns. In a system dedicated to image processing, the data might indeed be stored to meet this requirement. However, if the data is not stored appropriately (*e.g.*, scattered), we propose the following in-memory techniques (using MAGIC) to internally organize/move the data by considering the cost of such data movement.

In FiPDP, generating two negated vectors and aligning them requires $2N$ NOT operations. In the Hadamard product, generating two negated images and aligning them requires $2H$ if the images do not share any column, or $H + WN$ NOT operations if they do. In image convolution, generating P negated images and H negated kernels and aligning them requires $2PH$ for each color. Note that these techniques are applicable only when all data exist within the same array. When external data movement between different arrays is required, different techniques should be used [17]. The initial data organization overheads (Initialization) are summarized in Table VI. In Section V-A, we show that the initial data organization overhead is negligible.

E. Memory Array Limitations

The algorithms above assume the array size is not limited and can fit the required area for any given image. However, this is not the case in practice: processing large images (larger than the memory array) requires splitting into multiple arrays,

TABLE VI

THE INITIAL DATA ORGANIZATION LATENCY OVERHEAD (INITIALIZATION) FOR EACH ALGORITHM

Latency (Cycles)		
FiPDP	**Hadamard product**	**Image convolution**
$2N$	$MAX(2H, H+WN)$	$2P \cdot (H+P-1) \cdot \#colors$

TABLE VII

THE MAXIMUM SPLIT SIZE FOR EACH ALGORITHM IN A SINGLE 512 × 512 ARRAY WITH 8-BIT PRECISION

Split Size (Dimensions)		
FiPDP	**Hadamard product**	**Image convolution**
512	512×12	$\lfloor \frac{512}{P} \rfloor \times 8$

where each part needs to be processed independently using the same algorithm. If the image consists of multiple color images, each image will be considered alone, as would be done for multiple single colored images (*e.g.*, gray-scale). Furthermore, using multiple arrays enables multiple images to be processed with roughly the same latency as a single image.

For 8-bit precision and an array of size 512×512, the maximum length of the vector for FiPDP is 512. However, the area used for storing the inputs and performing the computation limits the width of the image in the Hadamard product to 12, and to 8 in image convolution. While the height of the image in Hadamard product is 512, the height in image convolution with a $P \times P$ kernel is limited to $\lfloor \frac{512}{P} \rfloor$ due to the P required copies of the original input image. The maximum split size for each algorithm is listed in Table VII. In Section V-D, we demonstrate how these sizes could be increased.

V. Simulation Results

We have extensively evaluated our proposed image processing algorithms, including the data organization overhead, execution time, throughput, comparison with APIM, and further possible optimizations. To this end, we built a cycle-accurate, functional simulator in MATLAB that accurately performs the logical flow exactly as shown in each algorithm. The simulator considers the array size, the precision (N), the dimensions, the locations and the organization of the inputs, and the different optimizations proposed in Section V-D.

Figure 5 shows the results generated from the simulator for the proposed image convolution algorithm after applying four different 3 × 3 filters used for sharpening and edge detection, one per quarter of the image. The value of each pixel is identical to the value of the equivalent pixel generated when executing a conventional C code implementation of image convolution with the same filters. This sanity check was performed for each of the implemented algorithms.

For the evaluation, we consider arrays of size 512 × 512, where each array can perform FiPDP, the Hadamard product, and image convolution with a 3 × 3 kernel (which is the most commonly used kernel [47]) on images of size up to 512 × 1, 512×12 and 170×8, respectively, with 8-bit precision for the pixels (using the values in Table VII). Larger images (*e.g.*, $K \times K$ where K is 800) are split to multiple smaller images with the exact same size that is bounded by the sizes mentioned previously so that each image could fit in arrays of size 512 × 512. Furthermore, we assume the data is stored organized and aligned prior to execution, except in Section V-A, where we evaluate the overhead due to data organization and show that the initial preprocessing overhead is negligible.

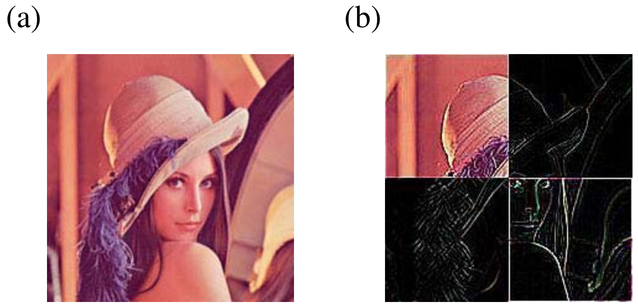


Fig. 5. Image convolution results generated from the cycle-accurate and functional simulator of image convolution proposed in this paper. (a) The original image and a (b) mosaic of the results after applying four different 3 × 3 filters used for sharpening and edge detection.

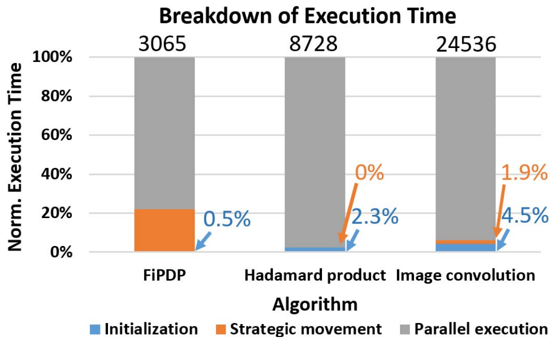


Fig. 6. Breakdown of the execution time of FiPDP, the Hadamard product and image convolution with a 3×3 kernel on images of size 512×1, 512×12 and 170×8, respectively, with 8-bit precision (using the values in Table VII). The values on the top of each bar are the overall number of cycles.

A. Data Organization Overhead

We break the execution time into three parts: (1) worst case preprocessing overhead before the algorithm begins the execution (Initialization, Section IV-D) in case the data is not properly stored (*e.g.*, scattered), (2) strategic data movement between different stages of the algorithm, for the purpose of reshaping the data to maximize the parallelism in the later stages, and (3) parallel execution of FiP multiplication/addition. Figure 6 shows the breakdown of the execution time of FiPDP, the Hadamard product, and image convolution with a 3 × 3 kernel on images of size 512 × 1, 512 × 12 and 170 × 8, respectively, with 8-bit precision (using the values in Table VII). In all three applications, the majority of the execution time is spent efficiently on parallel execution of FiP multiplications/additions. In the Hadamard product there is no strategic data movement since it is necessary only when the algorithm includes stages that perform operations on elements located in different rows during the execution; these elements must be aligned to the same row in case doing so improves the parallelism in the later stages. For FiPDP, 21% of the execution time is spent on strategic data movement, and 1.9% for image convolution, as both algorithms include such operations. Both FiPDP and image convolution require summing intermediate results located in different rows (after all the FiP multiplication operations are executed); instead of serially adding each value, each algorithm spends a small fraction of the execution time to reshape the data into one aligned vector so that addition of data elements can ultimately be performed in parallel.

The worst case preprocessing overhead is 0.5%, 2.3%, and 4.5%, respectively, in FiPDP, the Hadamard product, and image convolution. Image convolution requires multiple (P) copies of the image and thus its preprocessing overhead is the highest. In a system dedicated to image processing, the data will be stored properly and thus there will be no such overhead. With that being said, the significantly low initial preprocessing overhead is further justification for our decision to ignore it in the rest of our evaluation.

B. Execution Time

Our execution time evaluation focuses on the Hadamard product and image convolution. We leave performing FiPDP on large vectors for future work as it requires external data movement between the arrays for the final accumulation of the results. To evaluate the execution time of the Hadamard product and image convolution, we simulate the execution of 8-bit square images ($K \times K$) for different values of K. Note that the execution time of multiple small images is identical to processing multiple splits of a large image. The baseline assumes the memory array is sufficiently large to perform both algorithms within a single array for any desired image size. Then, we limit the size of the array to 512×512 and consider different numbers of arrays. To maximize the parallelism in the latter case, we ensure that all arrays have exactly the same split size, that the width of the split is the lowest possible, and that its height is the highest possible. To this end, the width of the split is chosen to be

$$split_{WIDTH} = \lceil \frac{K^2}{num\_arrays \cdot max\_height} \rceil, \qquad (1)$$

and the height to be

$$split_{HEIGHT} = \lceil \frac{K^2}{num\_arrays \cdot split_{WIDTH}} \rceil, \qquad (2)$$

where *max_height* is 512 in the Hadamard product and 170 in image convolution. These values are based on the maximum dimensions in Table VII for $P = 3$.

Figure 7 shows the execution time (in cycles) for the Hadamard product and image convolution with a 3×3 filter as a function of image size ($K \times K$) for different numbers of available arrays or a single sufficiently large array capable of performing the original baseline algorithm.

The execution time in the single sufficiently large array is linear with the dimension of the image thanks to the natural parallelism of MAGIC, which computes on all the rows simultaneously. However, it gives the worst execution time (and requires unfeasible array sizes for most K values). Partitioning to multiple arrays improves the execution time by three orders of magnitude as it extends the parallelism to multiple columns (the different columns are processed serially in the original algorithm without partitioning). Furthermore, because a large array incurs a longer MAGIC delay, the improvement in the execution time when using multiple smaller arrays is even higher in practice. The jumps in the execution time in multiple arrays occur only when $split_{WIDTH}$ increases. However, when $split_{HEIGHT}$ increases, the execution time is roughly constant. This is why we chose to minimize $split_{WIDTH}$ and maximize $split_{HEIGHT}$.

The largest image that can be fit in 512 arrays (each sized 512×512) is 1773×1773 for the Hadamard product versus 834×834 in image convolution. In image convolution, multiple copies of the image are required to maximize the parallelism and align the data (Figure 4(c)), lowering the area efficiency. These copies make the maximum split size much lower than in the Hadamard product, and the ratio between the split sizes determines the ratio between the maximum size of the images these algorithms can support.

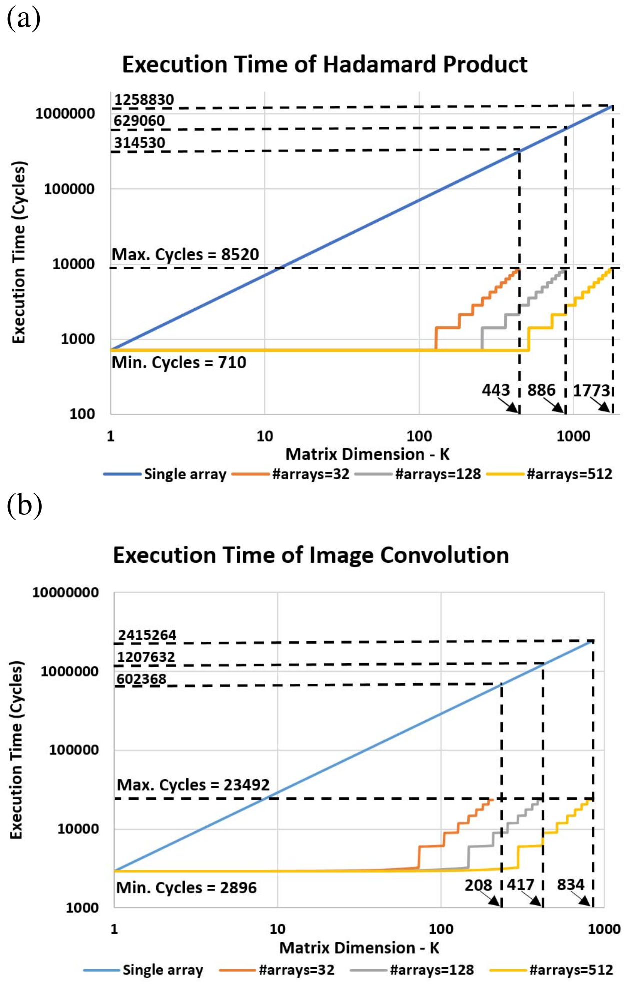


Fig. 7. Execution time (in cycles) of (a) the Hadamard product and (b) image convolution for different numbers of active memory arrays. The baseline considers a single sufficiently large array capable of performing the relevant algorithm without partitioning.

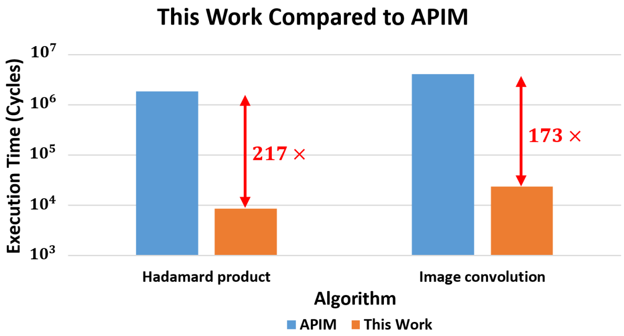


Fig. 8. Execution time (in cycles) of the Hadamard product and image convolution with a 3 × 3 kernel on images of size 512 × 12 and 170 × 8, respectively, with 8-bit precision (based on Table VII) as compared to APIM.

C. Comparison With APIM

We compare the execution of the proposed algorithms for the Hadamard product and image convolution with the APIM execution. We use a 3 × 3 kernel on images of size 170 × 8 for image convolution, images of size 512 × 12 for the Hadamard product, and 8-bit precision (using the values from Table VII). In APIM, image convolution is implemented by using the FiP multiplier and the MAGIC-based fast carry-save adder described in Section II-C. We use the same procedure to implement the Hadamard product in APIM. We compare the execution in a single 512 × 512 array only, since the results scale linearly for multiple arrays. Note that in APIM as well, the data is assumed to be stored properly organized before the execution begins.

1) Performance: For an apples-to-apples comparison, we exclude the read latency in APIM's multiplier and consider only the MAGIC cycles. Note that the real latency of APIM (when the read latency is included) is higher than the one considered here, and thus the results shown here are more conservative.

Figure 8 shows the execution time of the proposed algorithms for the Hadamard product and image convolution as compared to APIM. For the Hadamard product and image convolution, the execution time improves by 217× and 173×, respectively, thanks to the parallelism MAGIC enables. By contrast, APIM computes in the periphery, rendering its computation serial. Furthermore, the logic added by APIM to every single memory array complicates the periphery and lowers the capacity of the memory. Finally, the additional arrays used to compute with APIM require additional peripheral circuits, not required in pure MAGIC based computation.

2) Energy: Using a methodology similar to the one used in APIM, the energy results were obtained from circuit level simulations for a 45nm CMOS process technology using Cadence Virtuoso. We used the VTEAM memristor model [48], where the device parameters fit the HfOx based bipolar memristor [42], with R_{ON} of $10k\Omega$, R_{OFF} of $10M\Omega$, V_{SET} of $2V$ and V_{RESET} of $1V$.

The energy consumption improves by 217× for the Hadamard product and by 173× for image convolution. Interestingly, in APIM, the energy dissipated in the periphery when reading the data is dwarfed by the massive computation energy of performing MAGIC. To minimize data movement, most of the execution time in APIM is spent on performing MAGIC. That explains why the energy improvement of the proposed algorithms has the same trend as the latency improvement.

D. Further Possible Optimizations

1) Increasing the Split Size in Image Convolution: The maximum split size in image convolution is relatively small: 170 × 8 (Table VII). When executing the strategic data movement, the result image is negated horizontally (Figure 4(c)), so that the elements can later be negated up (Figure 4(d)) in parallel. The granularity of this parallelism is $O(W)$. However, since the width ($W = 8$) and the strategic data organization overhead are both relatively low (1.9% for the strategic data organization; see Figure 6), this part of the algorithm (steps $6 - 16$) could be executed by repeatedly negating horizontally one column at a time, then negating up (serially), and then adding the results. Doing this would increase the maximum split size to 170 × 13, and the maximum square image that could be convoluted would increase to 1064 × 1064, at the cost of an 11% increase in the execution time.

2) Lowering the Precision: Using lower precision is a direct solution to improve both the area efficiency and execution time. For example, 1-bit precision is very common in image processing applications [49]. Multiplying two 1-bit numbers is basically an AND operation. Since these two numbers are already stored negated, performing a single NOR operation gives the result of the AND ($AND(a, b) = NOR(a', b')$). Using 1-bit rather than 8-bit precision improves the execution time by 710× for the Hadamard product (this improvement is in accordance with the execution time ratios of 8-bit versus 1-bit FPFiPM) and by 25× for image convolution. Furthermore, the maximum split size becomes 512 × 170 for the Hadamard product and 170 × 98 for image convolution, since less area is needed for computation and the precision of both inputs and outputs decreased.

3) Using LPFiPM: Using LPFiPM rather than FPFiPM improves the execution time by 2.1× for the Hadamard product and 1.8× for image convolution. Additionally, since the precision of the outputs is lower in LPFiPM, more space could be given to the inputs, which thus increases the maximum split size to 512 × 16 for the Hadamard product and 170 × 17 for image convolution.

4) Using the Area Optimized Multipliers/Adders: Using the area optimized versions of FPFiPM and LPFiPM along with the area optimized adder algorithm [15] would increase the area efficiency of the Hadamard product and image convolution. For example, deploying the area optimized FPFiPM and adder would increase the maximum split size to 512 × 14 for the Hadamard product and 170 × 12 for image convolution, increasing the execution time by 27% and 10%, respectively.

5) Destroying the Inputs: The proposed algorithms could be modified to write the results in the memristors that store the input data (once the computation is finished and the computed data is no longer needed for other computations). While this would not affect the execution time, it would increase the maximum split size to 512 × 23 in Hadamard product and 170 × 10 in image convolution.

VI. Conclusions

In this paper, we show the feasibility of executing FiP multiplication in a single row of size-limited memory arrays using MAGIC. We also show how single row execution provides substantially higher throughput when the parallelism of MAGIC is utilized. We use vector FiP multiplication to realize more advanced algorithms: FiPDP, the Hadmard product, and image convolution. Our algorithms efficiently map and perform the desired task within the memristive memory array and among different arrays, exploit the massive parallelism that MAGIC offers, and minimize data transfer. We envision that these algorithms and others that are based on single row execution will provide the foundation for efficient memristive Memory Processing Unit (mMPU) that will help mitigate the von Neumann bottleneck. The next step is to build such a system and conduct a systematic comparison between a memory capable of performing MAGIC based operations versus alternative architecture points.

Acknowledgment

The authors would like to thank Saransh Gupta for sharing his methodology and data about APIM.

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] R. Ben-Hur and S. Kvatinsky, "Memory processing unit for in-memory processing," in *Proc. Int. Symp. Nanosc. Archit. (NANOARCH)*, Jul. 2016, pp. 171–172.

[3] R. Ben-Hur and S. Kvatinsky, "Memristive memory processing unit (MPU) controller for in-memory processing," in *Proc. IEEE Int. Conf. Sci. Elect. Eng. (ICSEE)*, Nov. 2016, pp. 1–5.

[4] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 273–287.

[5] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 481–492.

[6] S. F. Yitbarek, T. Yang, R. Das, and T. Austin, "Exploring specialized near-memory processing for data intensive operations," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2016, pp. 1449–1452.

[7] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 476–488.

[8] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "The desired memristor for circuit designers," *IEEE Circuits Syst. Mag.*, vol. 13, no. 2, pp. 17–22, 2nd Quart., 2013.

[9] J. Lee, M. Jo, D. Seong, J. Shin, and H. Hwang, "Materials and process aspect of cross-point RRAM," *Microelectron. Eng.*, vol. 88, no. 7, pp. 1113–1118, Jul. 2011.

[10] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-based IMPLY logic design procedure," in *Proc. IEEE 29th Int. Conf. Comput. Design (ICCD)*, Oct. 2011, pp. 142–147.

[11] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2015, pp. 1718–1725.

[12] L. Amaru, P. E. Gaillardon, and G. D. Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016.

[13] S. Kvatinsky *et al.*, "MAGIC—Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[14] R. Ben-Hur, N. Wald, N. Talati, and S. Kvatinsky, "SIMPLE MAGIC: Synthesis and in-memory MaP-ping of logic execution for memristor-aided loGIC," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2017, pp. 1–8.

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

[16] 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 Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.

[17] N. Talati *et al.*, "Practical challenges in delivering the promises of real processing-in-memory machines," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1628–1633.

[18] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with *in-situ* analog arithmetic in crossbars," in *Proc. ACM/IEEE Annu. 43rd Int. Symp. Comput. Archit.*, Jun. 2016, pp. 14–26.

[19] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 27–39.

[20] R. Jain, R. Kasturi, and B. G. Schunck, *Machine Vision*, vol. 5. New York, NY, USA: McGraw-Hill, Mar. 1995.

[21] R. A. Horn, "The Hadamard product," in *Matrices: Theory and Applications*, vol. 40. Providence, RI, USA: AMS, 1990, pp. 87–169.

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

[23] R. Zhen and R. L. Stevenson, "Image demosaicing," in *Color Image and Video Enhancement*. Cham, Switzerland: Springer, 2015, pp. 13–54.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2012, pp. 1097–1105.

[25] D. Bolme, J. Beveridge, B. Draper, and Y. Lui, "Visual object tracking using adaptive correlation filters," in *Proc. Int. Conf. Comput. Vis. Pattern Recognit.*, Sep. 2010, pp. 2544–2550.

[26] B. Shen, I. K. Sethi, and V. Bhaskaran, "DCT convolution and its application in compressed domain," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, no. 8, pp. 947–952, Dec. 1998.

[27] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[28] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[29] J. Reuben *et al.*, "Memristive logic: A framework for evaluation and comparison," in *Proc. 27th Int. Symp. Power Timing Modeling, Optim. Simulation (PATMOS)*, Sep. 2017, pp. 1–8.

[30] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using MAGIC," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[31] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 541–552.

[32] S.-S. Sheu *et al.*, "A 5ns fast write multi-level non-volatile 1 K bits RRAM memory with advance write scheme," in *Proc. Symp. VLSI Circuits*, Jun. 2009, pp. 82–83.

[33] M.-C. Wu, W.-Y. Jang, C.-H. Lin, and T.-Y. Tseng, "A study on low-power, nanosecond operation and multilevel bipolar resistance switching in Ti/ZrO$_2$/Pt nonvolatile memory with 1T1R architecture," *Semicond. Sci. Technol.*, vol. 27, no. 6, p. 065010, May 2012.

[34] L. Zhang, D. Strukov, H. Saadeldeen, D. Fan, M. Zhang, and D. Franklin, "SpongeDirectory: Flexible sparse directories utilizing multi-level memristors," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation*, Aug. 2014, pp. 61–74.

[35] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, p. 075201, 2012.

[36] H. S. Malvar, L.-W. He, and R. Cutler, "High-quality linear interpolation for demosaicing of Bayer-patterned color images," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, vol. 3, May 2004, pp. iii-485–iii-488.

[37] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. 53nd ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[38] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C-22, no. 12, pp. 1045–1047, Dec. 1973.

[39] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, "Design implications of memristor-based RRAM cross-point structures," in *Proc. Design, Autom., Test Eur. Conf. Exhib. (DATE)*, Mar. 2011, pp. 1–6.

[40] L. Zhang, S. Cosemans, D. J. Wouters, G. Groeseneken, M. Jurczak, and B. Govoreanu, "Selector design considerations and requirements for 1 SIR RRAM crossbar array," in *Proc. IEEE 6th Int. Memory Workshop (IMW)*, May 2014, pp. 1–4.

[41] J. J. Yang *et al.*, "High switching endurance in TaO$_x$ memristive devices," *Appl. Phys. Lett.*, vol. 97, no. 23, p. 232102, Nov. 2010.

[42] H. Y. Lee *et al.*, "Evidence and solution of over-RESET problem for HfOX based resistive memory with sub-ns switching speed and high endurance," in *IEEE IEDM Tech. Dig.*, Dec. 2010, pp. 19.7.1–19.7.4.

[43] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2009, pp. 14–23.

[44] J. Nickel, "Memristor materials engineering: From flash replacement towards a universal memory," in *Proc. IEEE IEDM Adv. Memory Technol. Workshop*, Dec. 2011, pp. 1–3.

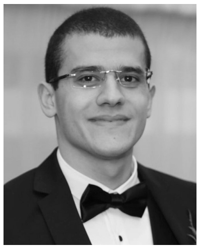
[45] P. J. Norman *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[46] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[47] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Comput. Sci.*, Sep. 2014, pp. 1–14.

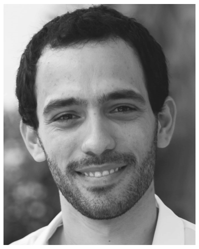
[48] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM: A general model for voltage-controlled memristors," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 8, pp. 786–790, Aug. 2015.

[49] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4107–4115.

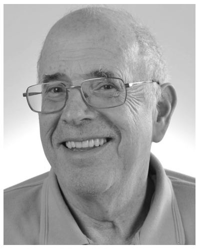


Ameer Haj-Ali received the B.Sc. degree *(summa cum laude)* in computer engineering from the Technion–Israel Institute of Technology in 2017. He is currently a Graduate Student with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology. From 2015 to 2016, he was with Mellanox Technologies as a Chip Designer. His current research is focused on novel computer architectures with emerging memory technologies and the design of energy efficient and parallel architectures.

Rotem Ben-Hur received the B.Sc. degree in electrical engineering from the Technion–Israel Institute of Technology in 2014, where she is currently pursuing the Ph.D. degree (direct path) with the Andrew and Erna Viterbi Faculty of Electrical Engineering. In 2012, she joined Elbit Systems as an FPGA Designer. Her current research is focused on novel architectures for logic with emerging memory technologies.



Nimrod Wald received the B.Sc. degree in electrical engineering and physics from the Technion–Israel Institute of Technology, Haifa, in 2013. Since 2015, he has been a Graduate Student with the Technion–Israel Institute of Technology, where he was involved in novel circuits and architectures for logic with memristors. In 2011, he joined Qualcomm Inc., as a Hardware Design Student. In 2013, he took a hardware architecture position in the area of performance analysis.



Ronny Ronen (F'08) received the B.Sc. and M.Sc. degrees in computer science from the Technion–Israel Institute of Technology in 1978 and 1979, respectively. From 1980 to 2017, he was with Intel where he was involved in various technical and managerial positions. He led the Intel Collaborative Research Institute for Computational Intelligence. He led the development of several system software products and tools including the Intel Pentium processor performance simulator and several compiler efforts. Until 2011, he was the Director of the Microarchitecture Research and then a Senior Staff Computer Architect with the Intel Development Center, Haifa. In these roles, he led/involved in the initial definition and pathfinding of major leading edge Intel processors. He was an Intel Senior Principal Engineer. He is currently a Senior Researcher with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology. He holds over 70 issued patents and has published over 20 papers.

Shahar Kvatinsky received the B.Sc. degree in computer engineering and applied physics and the M.B.A. degree from the Hebrew University of Jerusalem, in 2009 and 2010, respectively, and the Ph.D. degree in electrical engineering from the Technion–Israel Institute of Technology in 2014. From 2006 to 2009, he was with Intel as a Circuit Designer and was a Post-Doctoral Research Fellow with Stanford University from 2014 to 2015. He is currently an Assistant Professor with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology. His current research is focused on circuits and architectures with emerging memory technologies and the design of energy efficient architectures. He was a recipient of the 2010 Benin Prize, the 2013 Sanford Kaplan Prize for Creative Management in High Tech, the 2014 and 2017 Hershel Rich Technion Innovation Awards, the 2015 IEEE Guillemin-Cauer Best Paper Award, the 2015 Best Paper of Computer Architecture Letters, the Viterbi Fellowship, the Jacobs Fellowship, the ERC starting grant, the 2017 Pazy Memorial Award, and six Technion excellence teaching awards. He is an Editor of *Microelectronics Journal*.