Automated Synthesis for In-Memory Computing

Muhammad Rashedul Haq Rashed*, Sven Thijssen†, Sumit Kumar Jha‡, and Rickard Ewetz*
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, USA
†Department of Computer Science, University of Central Florida, Orlando, USA
‡Computer Science Department, Florida International University, Miami, USA
{muhammad.rashed, sven.thijssen, rickard.ewetz}@ucf.edu, jha@cs.fiu.edu

***Abstract*—Processing in-memory has the potential to break von-Neumann based design principles and unleash exascale computing capabilities. A rudimentary problem for in-memory paradigms is to decompose mathematical operations into in-memory compute kernels. In this paper, we propose the AUTO framework that automatically maps arithmetic operations into in-memory compute kernels that can be executed using non-volatile memory. The AUTO framework is based on defining semantically complete custom adders optimized for in-memory computing. Using a library of such adders and a projection of the partial product space, we discover decomposition that enable fixed-point multiplication to be executed with fewer steps. The framework also directly applies the technique to dot-product operations to further improve performance. Compared with state-of-the-art, the experimental results demonstrate that AUTO can perform fixed-point multiplication and dot-product operations with 16% and 19% fewer steps, respectively. For a library of scientific computing applications, this translates into energy and latency improvements of 15% and 17%, respectively.**

I. Introduction

The next wave of scientific discovery will be predicated on self-directed and constructivistic learning using scientific simulation [1]. Scientific simulation has led to breakthroughs within Bose-Einstein modeling [2], dendritic growth in alloys [3], and improved x-ray adsorption [4]. However, scientific simulations are extremely computationally expensive and are pushing today's high performance computing systems to the breaking point. Unfortunately, no further gains are expected from technology scaling due to the slowdown of Moore's law [5] and the von-Neumann bottleneck [6]. This has forced the industry to invest in non-conventional computing paradigms such as quantum computing [7], optical computing [8], and in-memory computing [9]. Processing in-memory using emerging non-volatile memory is a promising contender to accelerate data-intensive applications in future computing systems.

Despite that design automation is a core pillar of the highly successful semiconductor industry, most studies on in-memory computing are focused on the fabrication of novel devices [10–13], innovative circuit design [14–17], and non-Von-Neumann architectures [18–20]. State-of-the-art in-memory computing schemes rely on manually decomposing computation into in-memory compute kernels. This lack of design space exploration leads to sub-optimal results and that many in-memory technologies cannot live up to their full potential.

This work was in part supported by NSF awards # 2319399, # 2113307, the University of Central Florida, and Texas STARs award to Sumit Jha.

TABLE I: Performance Comparison for 32-bit Multiplication.

Work in	Partial Product Addition Approach	# of In-Memory Operations	Reduction in Operations
[21]	full adders	15007	–
[22]	full adders	12870	–
[23]	full adders	10046	–
This work	**custom adders**	**8462**	**16%**

Processing in-memory can be divided into analog [18, 24] and digital paradigms [25, 26]. Analog in-memory processors exploit the natural multiply-and-accumulate feature of in-memory computing platforms [27, 28]. While analog computation promises high energy efficiency, the resultant precision is fundamentally limited due to various device-level and circuit-level non-idealities [29, 30]. This paper focuses on digital paradigms due to the deterministic precision required by scientific simulation. The most effective digital paradigms are based on performing parallel bitwise operations between data stored in adjacent columns or rows within a crossbar of non-volatile memory. This includes paradigms such as bitwise-in-bulk [31], IMPLY [32], and MAGIC [33]. Of these paradigms, MAGIC has recently shown the most promising results due to that the entire computation is performed within memory, which minimizes data movement and the use of costly peripheral circuitry.

The MAGIC logic style is based on performing NOR operations between data stored in adjacent bitlines (or columns). The dominating computation within scientific simulation is matrix-vector multiplication (MVM). State-of-the-art solutions using MAGIC are based on decomposing the MVM operations into element-wise multiplication and addition operations. The multiplication operations are further decomposed into full adder operations of the partial products. Previous work utilized full adders of 1-bit [22] and 8-bits [23]. Lastly, the full-adder operations are decomposed into NOR in-memory compute kernels. In contrast, we propose to leverage automated synthesis to discover custom adders that lead to remarkable improvements, which is shown in Table I. These improved decompositions can directly be used in numerous architectural level studies focused on accelerating scientific computing, deep learning, and encryption applications [19, 34–37].

In this paper, we propose the AUTO framework for mapping arithmetic operations into in-memory compute kernels. The framework is centered on defining a library of semantically complete custom adders. Custom adders sum an

irregular number of bits of different values (more details in Section III). Next, automated synthesis is used to cover the partial products of fixed-point multiplication or dot-product operations. In the AUTO framework, the covering is performed using a projected version of the partial product space, to reduce the size of the custom adder library and the complexity of the covering problem.

The main contributions of the AUTO framework are outlined, as follows:

- The introduction of the concept of a projected partial product space and custom adders with semantically complete carry generation.
- A synthesis algorithm for decomposing arithmetic operations into in-memory compute kernels. The tool is used to synthesize a library for arithmetic operations such as element-wise addition, element-wise multiplication, and dot-product operations.
- A light-weight mapping algorithm for binding matrix-vector multiplication operations to in-memory compute kernels at runtime using the library.
- Compared with state-of-the-art approaches, AUTO is capable of performing fixed-point multiplication and dot-product operations using 16% and 19% fewer in-memory operations.
- The framework is evaluated using 15 scientific computing applications from the SuiteSparse matrix collection [38]. Compared with state-of-the-art approaches, the evaluation demonstrate energy and latency improvements of 15% and 17%, respectively.

The remainder of the paper is organized as follows: preliminaries in Section II. The motivation of the AUTO framework is discussed in Section III. The synthesis methodology of the framework is explained in Section IV. The architecture and the experimental evaluations are discussed in Section V. The paper is concluded in Section VI.

II. Preliminaries

In this section, we first explain the operating principles of the digital in-memory computing within MAGIC. Next, we discuss the state-of-the-art approaches to perform fixed-point (FiP) multiplication using digital in-memory computing.

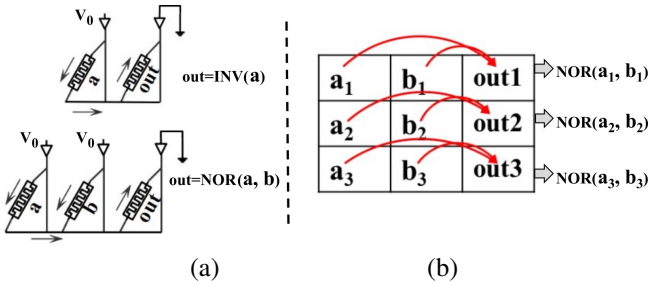


Fig. 1: (a) MAGIC based in-memory INV-NOR operations and (b) Parallel in-memory NOR-operations in NVM crossbar architecture.

A. Digital In-Memory Computing using MAGIC

MAGIC is an *in-situ* logic style where both the input and the output operands are stored inside memory. For execution of bitwise NOR/INV operations, controlled voltages and ground are applied to the memristor crossbar which is shown in Figure 1(a). Control voltages are applied to the columns with the input operands (a,b) and the column used to store the results (out) is grounded. The output memristors are switched between high resistance state (logic '0') and low resistance state (logic '1') depending on the output of the logic NOR/INV operations. The INV operations are performed by only applying the control voltage to a single bitline (or column). MAGIC is one of the more attractive logic styles for in-memory computing because it supports high-density logic operations and high-order parallelism in a crossbar architecture. The operating principle of parallel logic NOR/INV operations using MAGIC within memristor crossbar is shown in Figure 1(b). The figure shows how a NOR operation is evaluated along each wordline (or row) in a single cycle.

B. In-Memory Fixed-Point Multiplication

Acceleration of fixed-point (FiP) multiplication using the digital in-memory computing paradigms has been an active field of investigations in recent years. Majority of the works are on manually designed template-based approaches to decompose the FiP multiplication into an addition of partial products [21, 22]. Figure 2(a) shows the decomposition of a FiP multiplication into sequential row-wise partial product additions. In *Step 1*, the first two partial products are added and an intermediate result is generated. The intermediate result is then added to the next partial product. In the manual decomposition approach, the multi-bit addition is further decomposed into one-bit additions using the following NOR/INV expression.

$$C_{out} = ((a+b)' + (b+C_{in})' + (C_{in}+a)')'$$

$$Sum = ((a'+b'+C'_{in})' + ((a+b+C_{in})' + C_{out})')'$$

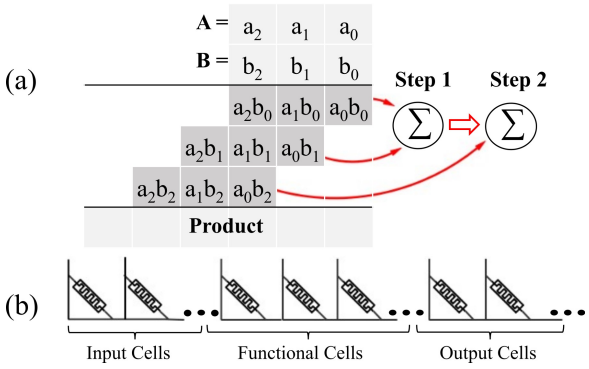


Fig. 2: (a) Decomposition of a FiP multiplication into sequential row-wise partial product additions and, (b) in-memory execution of full adder netlist using a row of memristor crossbar.

Here, C_{in} and C_{out} are the carry-in and carry-out bits respectively. The sequential NOR/INV operations are next performed using a single crossbar row as shown in Figure 2(b). The *input* and the *output* cells store the input (a_{0-i}, b_{0-i}) and output operands, respectively. The *functional cells* store the intermediate results of the sequential NOR/INV operations. Parallel crossbars rows can perform parallel FiP multiplication operations to accelerate data-intensive dot-product operations.

The use of multi-bit full adders for summing two rows of partial products was proposed in [23]. By converting an 8-bit addition directly into NOR/INV operations, the number of in-memory operations was reduced compared with using the decomposition of a single-bit full adder eight times. However, the solution approach was still manually restricted to row-wise summing two adjacent partial product terms [23]. An automated and comprehensive exploration of partial product cover patterns (row or non-row wise) is still missing.

III. Motivation

In this section, we introduce an alternate perspective to the partial product space cover problem. Next, we discuss a concept of semantically-constrained carry bit generation for decomposition of the partial products of FiP multiplication.

A. Rethinking the Partial Product Cover Problem

In this section, we introduce an alternate perspective to the partial product cover problem of FiP multiplication.

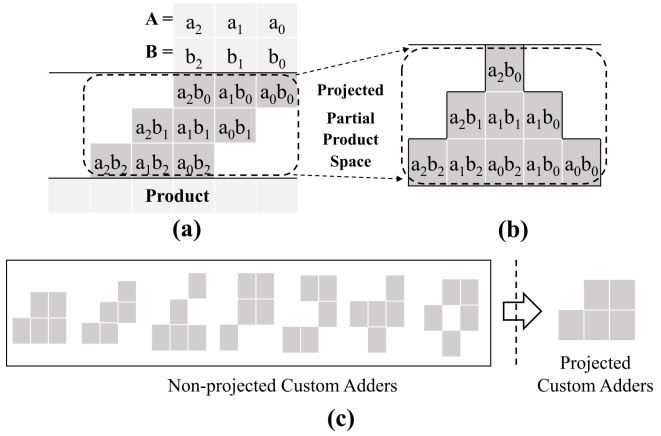


Fig. 3: (a) Traditional *non-projected* partial product space for a 3-bit FiP multiplication, (b) proposed *projected* partial product space for FiP multiplication, and (c) pruning of redundant non-projected custom adders.

We show the traditional representation of the partial product space for FiP multiplication of two 3-bit operands A and B in Figure 3(a). The multiplication results in three rows of partial products. As discussed earlier, the state-of-the-art approach to in-memory FiP multiplication is to sequentially cover these partial products row-wise using full adders. We speculate that it might be beneficial to explore arbitrary partial product cover patterns (e.g. *custom adders*) to cover the partial product space. We observe that the placement of all the bits in a column is interchangeable as addition is commutative. Hence, we present a projected view of the partial product space in Figure 3(b). The partial product space is now a pyramid of bit-wise multiplications. Re-shaping the partial product space into this pyramid structure allows us to obtain a standard structural form, which facilitates easy use of custom adders whose shapes are not strongly coupled to the choice of the bits being covered. One additional importance of the projected view of partial product space is that it helps us to avoid exploration of redundant adders. The concept is explained with an example in Figure 3(c). On the left of Figure 3(c), we present seven non-projected custom adders. Each gray box within the adder kernels represents a partial product bit. Using the projected partial product space, the seven kernels can be projected into a single projected custom adder, as shown to the right in Figure 3(c). We note that the resulting kernel exists in both projected and non-projected form. Based on this example, we deduce that the projection of partial product space allows us to prune out redundant custom adders, which both reduces the number of possible adders and simplifies the covering problem.

B. Semantically Complete Carry Generation

Before we begin the exploration of arbitrary cover patterns for the partial product addition, we hypothesize that the synthesis of custom adders should avoid generating redundant carry bits that will never be non-zero due to the semantic constraints imposed by choice of the kernel. Such a semantically-constrained synthesis does not yield unnecessary in-memory operations. We explain the concept with an example in Figure 4.

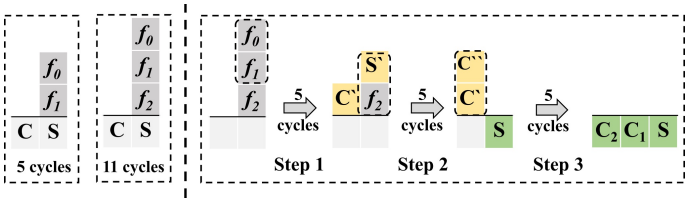


Fig. 4: Redundant carry generations. (a) Addition of 2 and 3-bits , and (b) decomposition of the kernel of addition of 3-bits into a series of addition of 2-bits.

Figure 4(a) shows the kernels for the addition of 2-bits (f_{0-1}) and 3-bits (f_{0-2}), respectively. The former kernel can be synthesized into a netlist of 5 NOR/INV operations and the latter kernel can be synthesized into a netlist of 11 NOR/INV operations. Now, we show how the kernel of addition of 3-bits can be executed using the kernel of addition of 2-bits in Figure 4(b).

In *Step 1*, we add the bits f_0 and f_1 which results in an intermediate sum bit S' and a carry bit C'. In *Step 2*, S' is added with bit f_2 and yields the sum bit S and carry bit C''. Finally, C' is added with C'' and results in two carry-out bits C_1 and C_2. However, the maximum limit on the sum of the addition of three bits is $''11''$ and the C_2 bit, therefore, is a theoretical impossibility. Due to the sequential nature of the

cover approach, the synthesis tool cannot speculate the redundant carry generations. As a result, opposed to the 11 NOR/INV operations in the netlist of addition of 3-bits, this naive cover yields 15 NOR/INV operations. Therefore, we should select to cover kernels that result in semantically complete carry bits.

Every arbitrary partial product cover corresponds to a custom adder. A custom adder is defined to be semantically complete if the maximum sum of the covered partial products is equal to $(2^k - 1)$ for any k. It is desirable to create custom adders that cover partial products with a value of 3, 7, 15, 31, 63, etc. However, in the AUTO framework, we will also leverage custom adders that are close to semantically complete to enable the use of custom adders with large covering patterns.

IV. The AUTO Framework

In this section, we first present the AUTO framework, which is shown in Figure 5. The framework consists of three steps: synthesis of custom adders, synthesis of arithmetic operations, and fast binding to hardware. The first two synthesis steps are only performed one-time, while the fast binding performed once per matrix.

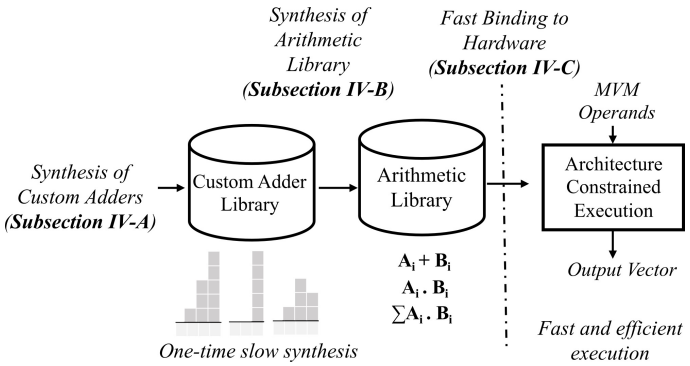


Fig. 5: Overview of the AUTO framework.

In the synthesis of custom adders step, we exhaustively synthesize all possible custom adders into NOR/INV operations. The details are provided in Section IV-A. Second, we automatically synthesize a library of arithmetic operations such as multiplication and dot-product operations with different precision. The step is outlined in Section IV-B. Third, given an MVM operation, the computation is bound to crossbar hardware by decomposing it into operations in the arithmetic library while considering the constraints of the architecture. This step is discussed in Section IV-C. Lastly, MVM operations are performed using in-memory computing for different input vectors.

A. Synthesis of Custom Adders

In this section, we describe how we synthesize a library of custom adders. The synthesis flow of custom adders is illustrated in Figure 6. The synthesis is performed by first exhaustively enumerating all custom cover patterns $\mathcal{L} = \{l_1, \cdots, l_M\}$ with a total value equal to or less than a desired threshold. Each partial product term in a cover has a value with respect to the right most column in the cover. The

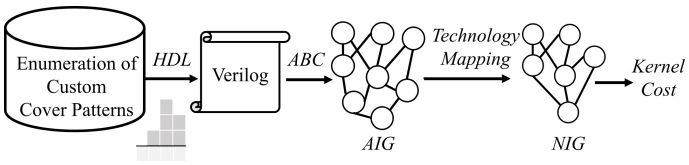


Fig. 6: Flow for synthesizing a custom adder library.

partial products in different columns are valued 1, 2, 4, 8, etc. Let's consider n is the total number of columns in a cover kernel and s_i denotes the number of partial products in the i^{th} column from the right. Now, for each $l_j \in \mathcal{L}$,

$$l_j = \sum_{i=1}^{n} s_i \times 2^{i-1} \leq 2^k - 1$$
$$\text{s.t.} \qquad s_1 \geq 2, \qquad \forall (i,j,k) \in Z^+$$

For instance, Figure 7 shows all the custom adders for a kernel weight limit of 7. We define the weight of a custom adder kernel as the maximum partial sum that the adder can generate. The weight of a kernel can be calculated using the formula $\sum_{i=1}^{n} s_i \times 2^{i-1}, \forall i$. The figure shows that there can be 12 different custom adder kernels with kernel weight ≤ 7.

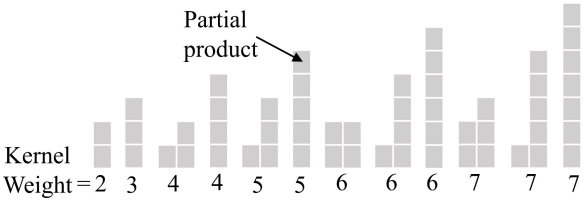


Fig. 7: Library of custom adders for kernel weight limit of 7. The gray blocks each represents a partial product.

For higher kernel limits, we get richer kernel library with many custom adders. The number of available kernels with respect to different kernel weight thresholds is shown in Table II. Richer pattern libraries with many different column and row configurations are expected to translate into improved performance.

TABLE II: Size of the Kernel Cover Library

	Kernel Weight Limit $(2^k - 1)$					
	3	7	15	31	63	127
# of Kernels	2	12	82	812	12754	318664

Next, we synthesize each of these kernels into NOR and INV operations. We first automatically generate a verilog description of the custom adder. Next, ABC [39] synthesizes the adder into an AND-Inverter Graph (AIG) which is the internal data-structure of ABC. Lastly, technology mapping is performed with respect to a technology library of INV-gates, NOR2-gates, and NOR3-gates. NOR operations with up to three inputs is the default for MAGIC. The total number of NOR/INV operations is stored in the custom adder library.

B. Synthesis of Arithmetic Operations

In this section, we first define the synthesis problem for arithmetic operations. Next, we outline our synthesis solution.

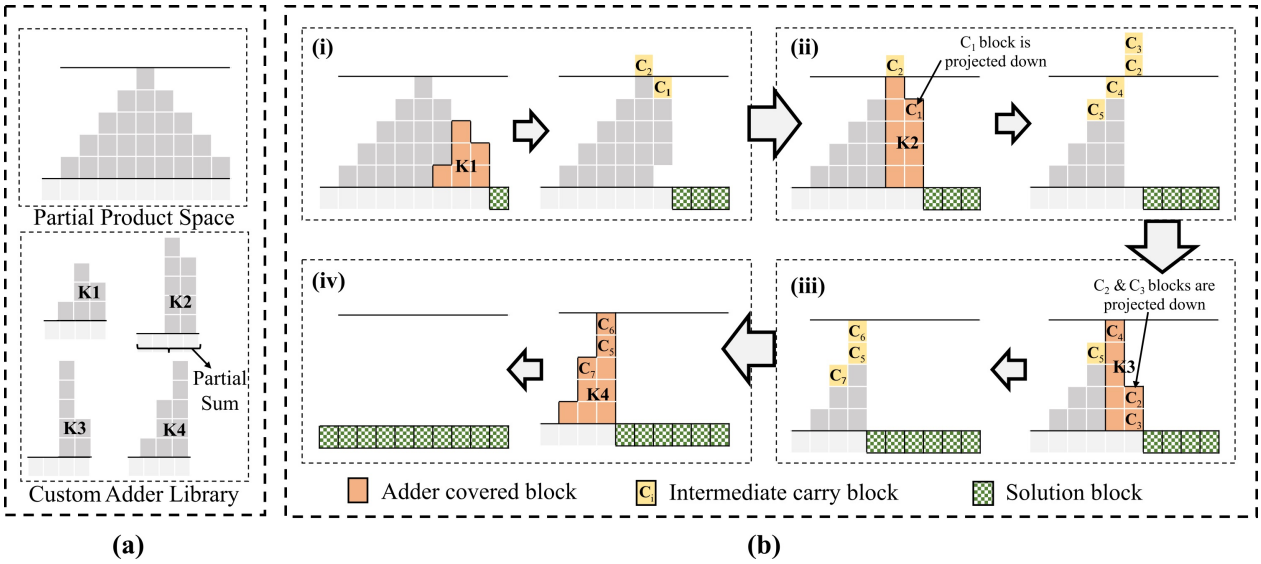


Fig. 8: Our kernel-based approach to covering the partial products of FiP multiplication. (a) On the top: the target partial product space for 5-bit FiP multiplication. At the bottom: a library of representative custom adder kernels to cover the partial product space. (b) The workflow of sequentially covering the partial product space using the kernels from the library.

Lastly, we describe how to extend the synthesis solution to dot-product operations.

Problem formulation: The input to the synthesis is the partial products of an arithmetic operation $\mathcal{S} = \{s_1, \cdots, s_N\}$, a library of custom adders $\mathcal{T} = \{t_1, \cdots, t_M\}$. The output is a sequence of K custom adder operations $\mathcal{X} = \{(x_1^a, x_1^l), \cdots, (x_K^a, x_K^l)\}$ that consume the partial products. The objective is to minimize the total number of in-memory operations, which can be formulated, as follows:

$$\min \quad \sum_{k=1}^{K} cost_{x_k^a} \tag{1}$$

$$\text{s.t.} \quad s_i + \sum_{k=1}^{K} p_{x_k^a}^{i-x_k^l+1} - c_{x_k^a}^{i-x_k^l+1} \leq 1, \forall i$$

Here, $cost_m$ is the number of operations that are required to execute the adder m, s_i is the number of partial products in column i, x_k^a is the id m of the adder executed at k in the execution sequence $\mathcal{X}$, and x_l^a is the location in the partial product space where the adder is executed (right edge) in the execution sequence $\mathcal{X}$.

1) Arithmetic Synthesis Algorithm: In this section, we explain how we synthesize the arithmetic operations in the arithmetic library. The operations are synthesized using Algorithm 1.

The algorithm iteratively selects a custom adder, processes the adder and updates the temporary partial product space. This process in continued until there is at most a single partial product in each column. The algorithm considers all custom adders as candidates. First, the algorithm prunes all adders that do not cover the right-most column of the partial product space. Second, another round of pruning is done based on the semantic completeness of the previously selected adders. We use the distance to $(2^k - 1)$ for any k to measure the completeness. Lastly, we select the adder that requires the least number of in-memory cycles to be executed.

Algorithm 1: Synthesis of Arithmetic Operation

Input: Non-traditional adder library $\mathcal{T}$; Partial product space $\mathcal{S}$.
Output: Selected series of non-traditional adders $\mathcal{X}$.
$\mathcal{X} \leftarrow \emptyset$; $p \leftarrow \mathcal{S}(LSB)$; \\pointer to the LSB position
while $\mathcal{S} \neq \phi$ **do**
  $\mathcal{C} \leftarrow \mathcal{T}$ \\$\mathcal{C}$ is a list of candidate adders
  $\mathcal{C} \leftarrow$ Prune $\mathcal{C}$ s.t. right most column in $\mathcal{S}$ is covered;
  $\mathcal{C} \leftarrow$ Prune $\mathcal{C}$ based on semantical completeness;
  $m \leftarrow$ Select remaining adder with smallest *cost* in $\mathcal{C}$;
  $\mathcal{X} \leftarrow \mathcal{X} \cup \{m\}$;
  $\mathcal{S} \leftarrow (\mathcal{S} \backslash \{m\})$; $p \leftarrow \mathcal{S}(LSB)$; \\Update $\mathcal{S}$ and p
end
return $\mathcal{X}$;

We illustrate the algorithm using an example in Figure 8. On the top of Figure 8(a), we show a target partial product space for a 5-bit FiP multiplication. Each gray box represents a partial product bit. At the bottom of Figure 8(a), we show a representative subset ($K_1 - K_4$) of the custom adder library. Note that, these kernels are not an exhaustive list of custom adder kernels. Each of the kernels $K_1 - K_4$ can cover a certain shape of the partial product space of FiP multiplication and generate several partial sum bits. The goal is to use the adders in the library to cover the target partial product space. The kernel cover workflow is illustrated in Figure 8(b). In the left of Figure 8(b)-(i), we show the placement of adder $K1$ in the partial product space. The adder resolves the covered bits from the problem space and generates two final solution bits in the least significant bit (LSB) position 2 and 3 which is shown in the right side of Figure 8(b)-(i). Note that the LSB position 1 is already resolved and need not be covered by any adder. The adder $K1$ cover also generates two carry bits C_1 and C_2 which

dynamically update the problem space. Next, the placement of adder $K2$ is shown in the left of Figure 8(b)-(ii). The C_1 bit is projected down in the figure. Note again that the placement of all the bits in a column are interchangeable as addition is commutative. After the processing of $K2$, LSB position 4 is resolved and three carry bits $C_3 - C_5$ are generated as shown in the right side of Figure 8(b)-(ii). This process is continued in Figure 8(iii) and (iv) where the adder $K3$ and $K4$ cover the remaining partial products. In each step, the problem space is dynamically updated with the removal of resolved bits and inclusion of incoming carry bits.

2) Dot-Product Operations: In this section, we extend the kernel cover approach of FiP multiplication to dot-product operations. Figure 9 illustrates the concept of generating the partial-product space of dot-product operations. The left side of the figure shows the partial product space of two FiP multiplications $A_1.B_1$ and $A_2.B_2$. The partial product space of the sum of the two FiP multiplications $A_1.B_1 + A_2.B_2$ is shown in the middle of the figure. An equivalent dot product representation of this addition of multiplications is shown in the right side of the figure. It can be observed that the partial products of dot product operation assumes a steeper pyramid shape. Fortunately, the same lookup table of adder library in the AUTO framework can cover this partial product space using the same workflow as the FiP multiplication. The AUTO framework decomposes the MVM operations into a series of dot product operations and performs a unique dot product operation, in parallel, in each row of the crossbar. As discussed in the next section, the architectural constraint of crossbar dictates how many dot-products can be processes in each row of the crossbar.

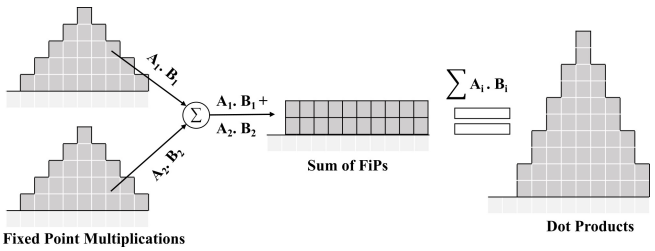


Fig. 9: Merging FiP multiplications to generate the partial product space for dot product operation.

C. Fast Binding to Hardware

In this section we discuss the architecture constrained hardware binding of matrix-vector multiplication (MVM) operations into memristor crossbars. The hardware binding is done using the Algorithm 2.

For an MVM operation of a matrix W and vector Y, the algorithm aims to bind a subset of dot-product $W_i.Y$ into a single row of crossbar. Here, W_i is the i^{th} row of the matrix. The number of dot-product operands that will fit in a crossbar row is determined by (a) the bit-precision of dot-product operation, (b) intermediate functional memristor cost (acquired from adder sequence library $\mathcal{X}$) and, (c) the dimension of the memristor crossbar. In the algorithm, the *temp* variable iteratively tracks the total number of memristor

Algorithm 2: Architecture Constrained Hardware Binding

Input: Arithmetic library $\mathcal{X}$; Crossbar dimension $r \times q$; Bit precision d; Matrix W; Vector Y;
Output: Hardware assignment of MVM.

```
u ← 0; v ← 0; temp ← 0; \\initializing
while r ≥ temp do
    v ← v+1; \\incrementing #of dot-product arguments
    temp ← (2×d×v) + cost(X,v)
           + ((2×d)+bit_length(log2(v)));
    \\total cost of input cell+functional cell+output cell
    if (r ≥ temp) then
        u ← v; \\#of architecture supported arguments
    else
        break;
    end
end
Bind(Partition((Wi.Y),u)),∀i; \\Partitioning& Binding
return;
```

requirement to bind v number of dot-product arguments. When an argument threshold u is reached, the dot-product $W_i.Y$ is partitioned into a series of dot-products, each with u arguments. Next, all of the partitioned dot-products are bound to different crossbars but each at the same row/wordline index. This condition is implemented to enable parallel data transfer between adjacent crossbars. The algorithm aims to use the dot-product operations with the maximum number of arguments, since it is observed in the experimental evaluation that this improves efficiency (see Figure 13 in Section V).

V. Experimental Evaluation

In this section, we evaluate the performance of the AUTO framework. The experiments are performed on an octa-core 3.60 GHz Intel Core i9 processor with NVIDIA RTX 2070 and 64 GB RAM. For synthesizing the in-memory computing kernels, we use the ABC tool [39] with custom cell libraries for MAGIC. The AUTO framework is developed using a blend of C++ and MATLAB scripts.

We first present the architecture of the AUTO framework. Next, we evaluate the synthesis complexity of the AUTO framework. Subsequently, we present the performance of AUTO on FiP operations and compare it with the SOTA. Finally, we evaluate the AUTO framework for MVM-based applications within scientific simulation.

A. Architecture

In this section, we present the architecture of the AUTO framework. An overview of the architecture of the AUTO framework is shown in Figure 10.

Each rank of the architecture consists of several in-memory computing chips which are shown in Figure 10(a). The micro-architecture of the chips is shown in Figure 10(b). Each bank of the chip contains an array of crossbar mats arranged in a row-parallel fashion to perform row-wise MAGIC operations [34]. The cross-section of a crossbar mat is shown in Figure 10(c). Row and column drivers are used to steer the

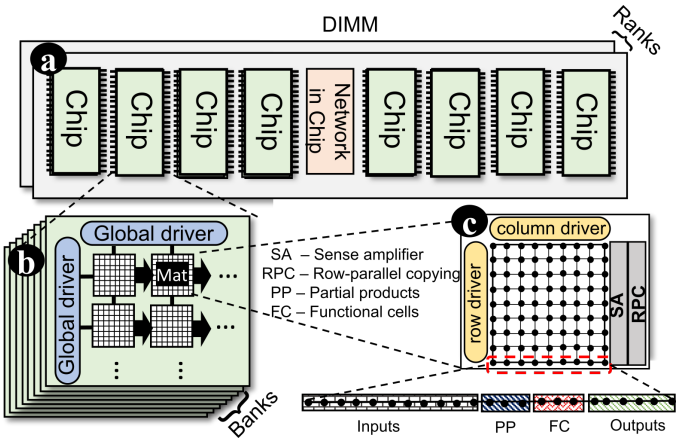


Fig. 10: Architecture of the proposed framework.

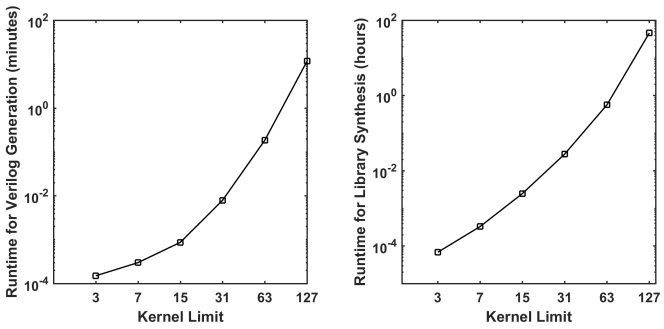


Fig. 11: Runtime complexity of adder library synthesis steps. (a) Runtime (in minutes) for verilog script generation, and (b) runtime (in hours) for adder library synthesis.

in-memory operations. Sense amplifiers (SA) and row-parallel copying units (RPCs) [19] are used to enable parallel data transfer into the neighbouring mats. Each wordline of the crossbar is partitioned into four *task-specific* segments, which is shown at the bottom of Figure 10(c). The input and the output segments store the input and output operands of dot product, respectively. The partial product segment stores the partial products of the current kernel-covered region. The functional segment performs the sequential in-memory operations within a kernel netlist using the algorithm in [23].

The per-unit cost of different architectural components are summarized in Table III. The per unit area and power costs are adapted from previous works [18, 19]. The cross architecture data communication cost between crossbar mats is adapted from the case study of [40].

TABLE III: Area-Power Cost of Architectural Components

Component	Parameter	Specs	Area	Power
Crossbar Mat	size	256×256	100 μm^2	1.20 mW
Driver	# unit	1	400 μm^2	0.65 mW
Sense Amp.	# unit	256	14.28 μm^2	0.58 mW
Bus	bandwidth	32B	31.40 mm^2	26 mW
Local Bus	#wires	256	0.06 mm^2	4.66 mW

B. Evaluation of Synthesis Complexity

The synthesis of the AUTO framework consist of a one-time expensive task of synthesizing the entire custom adder library. As shown in the Table II, the size of the adder kernel library grows exponentially with the increase of kernel weight limit $2^k - 1$ where $k \in Z^+$. We show the runtime of the library synthesis steps for different kernel weight limit in Figure 11. Figure 11(a) shows the runtime for verilog script generation and Figure 11(b) shows the runtime to synthesizing the custom adder netlists using the verilog files. The figure shows that the verilog script generation is very fast and take up to 12 minutes for kernel limit of 127. The figure also shows that the adder library synthesis is reasonably fast (from less than a second to less than an hour) for kernel limit up to 63. However, for the kernel limit of 127, the total number of adders are 318664 and the total synthesis time is 46 hours. Note again that this is a one time-effort[1] and is general purpose for any arithmetic operations.

The next steps in the AUTO framework are the synthesis of arithmetic operations using the Algorithm 1 and fast hardware binding using Algorithm 2, respectively. These steps are almost instantaneous and generally take less than a second.

C. Evaluation of FiP Multiplication and Dot-Product

The performance of the AUTO framework on FiP multiplications of different precision is shown in Table IV. The table shows that the AUTO framework requires at least 8-16% fewer in-memory operations compared to the the previous frameworks. It can also be observed that the performance improvement is greater for higher precision FiP multiplications.

TABLE IV: Performance Comparison for FiP Multiplication.

Work in	Multiplicand Precision		
	8-bit (steps)	16-bit (steps)	32-bit (steps)
ULTRA [21]	871	3663	15007
SIMPLER [22]	726	3110	12870
LOGIC [23]	518	2310	10046
AUTO **(This work)**	**478**	**2024**	**8462**
Minimum Improvement	**8%**	**12%**	**16%**

We present the results for the evaluation of dot-products within the AUTO framework in Figure 13. Figure 13(a) shows the % improvement (in terms of reduction of in-memory operations) achieved by dot-product operations over their equivalent sequential FiP multiplication and addition operations. The figure shows that for 8-bit dot products, the framework reduces the number of in-memory operations by 7-12% for variable number of dot-product arguments. The framework achieves up to 4% reduction in in-memory operations for dot product operations on 32-bit multiplicands. Figure 13(b) shows the maximum number of dot-product arguments for crossbars of different dimensions. Intuitively, dot product operations of higher precision require larger crossbars.

[1]The entire custom adder library is publicly available on GitHub (*github.com/mrhrashed/AUTO*).

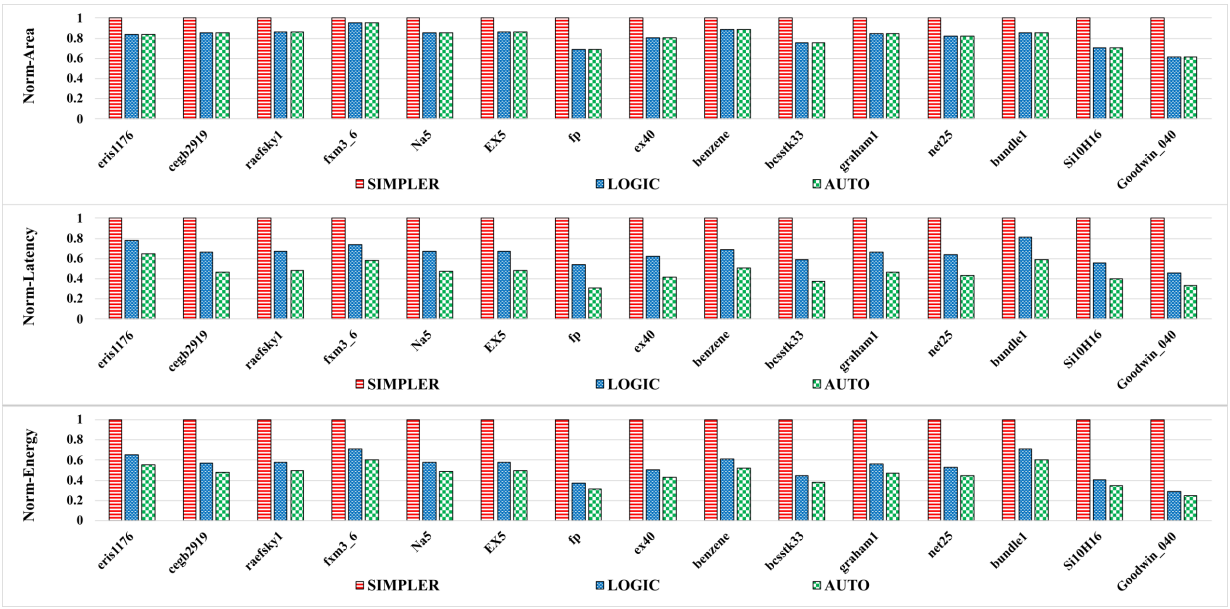
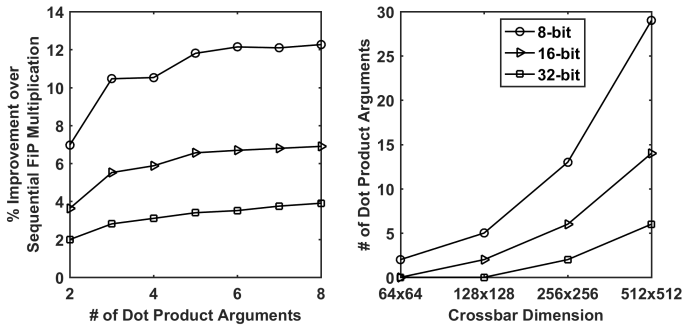


Fig. 12: Area-latency-energy overhead evaluation of AUTO compared with SIMPLER [34] and LOGIC [23], respectively.



(a) (b)

Fig. 13: Evaluation on dot product operations on 8, 16 and 32-bit multiplicands. (a) % improvement over sequential FiP multiplications for different order of dot product operations and, (b) # of dot-products for different crossbar dimensions.

D. Evaluation with MVM Applications

In this section, we evaluate the performance of the AUTO framework for the matrix-vector-multiplication (MVM) operation. MVM is the dominating computation within scientific simulation. These systems of linear equations can be solved using the conjugate gradient (CG) method [41]. In the CG method, an iterative MVM is performed which refines the system solution in each iteration. The aim of the AUTO framework is to accelerate this MVM operation. For the evaluation, we select 15 matrices from the SuiteSparse matrix collection [38]. An overview of the benchmarks is shown in Table V. We compare the performance of the AUTO framework with the state-of-the-art manual template-based in-memory computing paradigm SIMPLER [34] and with the synthesis-based in-memory computing paradigm LOGIC [23]. We consider a bit-width of 32-bits for the MVM operands.

In Figure 12, we present the area, latency and energy consumption of the three frameworks. The experimental results show that the AUTO framework improves area, latency and energy consumption by 1.4x, 2.25x and 2.5x, respectively, over SIMPLER. Additionally, the AUTO framework achieves 17% and 15% improvements in latency and energy consumption over the LOGIC framework. The improvements stem from the fact that AUTO is able to decompose the MVM operations into fewer in-memory compute kernels as shown in Table IV. The area of AUTO and LOGIC are comparable since a similar in-memory architecture is used.

TABLE V: Overview of benchmarks from the SuiteSparse Matrix Collection [38].

Applications	Systems	Matrix Dimensions	#Non-zeros
eris1176	Power Network Problem	1176×1176	18552
cegb2919	Structural Problem	2919×2919	321543
raefsky1	Computational Fluid Dynamics	3242×3242	293409
fxm3_6	Optimization Problem	5026×5026	94026
Na5	Theoretical/Quantum Chemistry	5832×5832	305630
EX5	Combinatorial Problem	6545×6545	295680
fp	Electromagnetics Problem	7548×7548	834222
ex40	Computational Fluid Dynamics	7740×7740	456188
benzene	Theoretical/Quantum Chemistry	8219×8219	242669
bcsstk33	Structural Problem	8738×8738	591904
graham1	Computational Fluid Dynamics	9035×9035	335472
net25	Optimization Problem	9520×9520	401200
bundle1	Computer Graphics/Vision	10581×10581	770811
Si10H16	Theoretical/Quantum Chemistry	17077×17077	875923
Goodwin_040	Computational Fluid Dynamics	17922×17922	561677

VI. Summary and Future Work

In this paper, we propose a framework called AUTO for mapping arithmetic operations into in-memory compute kernels that can be executed using non-volatile memory. The framework is based on defining custom adders with semantically complete carry generation. The experimental results show that AUTO reduces the number of operations needed to perform fixed-point multiplication and dot-product operations with 16% and 19%, respectively. This translates into improved simulation performances for a whole host of scientific computing applications. In our future work, we plan to extend the AUTO framework to the acceleration of a wide range of data-intensive applications.

REFERENCES

[1] K. Cranmer, J. Brehmer, and G. Louppe, "The frontier of simulation-based inference," *Proceedings of the National Academy of Sciences*, vol. 117, no. 48, pp. 30055–30062, 2020.

[2] B. Jackson and E. Zaremba, "Modeling bose-einstein condensed gases at finite temperatures with n-body simulations," *Physical Review A*, vol. 66, no. 3, p. 033606, 2002.

[3] M. A. Jaafar, D. R. Rousse, S. Gibout, and J.-P. Bédécarrats, "A review of dendritic growth during solidification: Mathematical modeling and numerical simulations," *Renewable and Sustainable Energy Reviews*, vol. 74, pp. 1064–1079, 2017.

[4] J. S. Sims, W. L. George, S. G. Satterfield, H. K. Hung, J. G. Hagedorn, P. M. Ketcham, T. J. Griffin, S. A. Hagstrom, J. C. Franiatte, *et al.*, "Accelerating scientific discovery through computation and visualization ii," *Journal of Research of the National Institute of Standards and Technology*, vol. 107, no. 3, p. 223, 2002.

[5] L. Eeckhout, "Is moore's law slowing down? what's next?," *IEEE Micro*, vol. 37, no. 04, pp. 4–5, 2017.

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[7] L. Gyongyosi and S. Imre, "A survey on quantum computing technology," *Computer Science Review*, vol. 31, pp. 51–71, 2019.

[8] R. Xu, P. Lv, F. Xu, and Y. Shi, "A survey of approaches for implementing optical neural networks," *Optics & Laser Technology*, vol. 136, p. 106787, 2021.

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

[10] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[11] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, *et al.*, "Spin-transfer torque magnetic random access memory (stt-mram)," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, pp. 1–35, 2013.

[12] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[13] E. Ambrosi, A. Bricalli, M. Laudato, and D. Ielmini, "Impact of oxide and electrode materials on the switching characteristics of oxide reram devices," *Faraday discussions*, vol. 213, pp. 87–98, 2019.

[14] Z. Lu, M. T. Arafin, and G. Qu, "Rime: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pp. 120–125, 2021.

[15] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, IEEE, 2018.

[16] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Multpim: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1647–1651, 2021.

[17] A. Velasquez and S. K. Jha, "Parallel boolean matrix multiplication in linear time using rectifying memristors," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1874–1877, IEEE, 2016.

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

[19] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 802–815, 2019.

[20] M. H. I. Chowdhuryy, M. R. H. Rashed, A. Awad, R. Ewetz, and F. Yao, "Ladder: Architecting content and location-aware writes for crossbar resistive memories," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 117–130, 2021.

[21] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[22] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2018.

[23] M. R. H. Rashed, S. K. Jha, and R. Ewetz, "Logic synthesis for digital in-memory computing," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[24] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *2016 53nd ACM/EDAC/IEEE DAC*, pp. 1–6, IEEE, 2016.

[25] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *2009 IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 33–36, IEEE, 2009.

[26] A. U. Hassen, D. Chakraborty, and S. K. Jha, "Free binary decision diagram-based synthesis of compact crossbars for in-memory computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 622–626, 2018.

[27] C. Li, M. Hu, Y. Li, H. Jiang, N. Ge, E. Montgomery, J. Zhang, W. Song, N. Dávila, C. E. Graves, *et al.*, "Analogue signal and image processing with large memristor crossbars," *Nature Electronics*, vol. 1, no. 1, p. 52, 2018.

[28] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–382, IEEE, 2018.

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

[30] M. Le Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tuma, C. Bekas, A. Curioni, and E. Eleftheriou, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, no. 4, pp. 246–253, 2018.

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

[32] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive'switches enable 'stateful'logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

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

[34] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2019.

[35] N. Talati, R. Ben-Hur, N. Wald, A. Haj-Ali, J. Reuben, and S. Kvatinsky, "mmpu—a real processing-in-memory architecture to combat the von neumann bottleneck," *Applications of Emerging Memory Technology: Beyond Storage*, pp. 191–213, 2020.

[36] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "Cryptopim: In-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[37] M. Imani, S. Pampana, S. Gupta, M. Zhou, Y. Kim, and T. Rosing, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 356–371, IEEE, 2020.

[38] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[39] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*, vol. 17, 2007.

[40] N. Talati, A. H. Ali, R. B. Hur, N. Wald, R. Ronen, P.-E. Gaillardon, and S. Kvatinsky, "Practical challenges in delivering the promises of real processing-in-memory machines," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1628–1633, IEEE, 2018.

[41] M. R. Hestenes, E. Stiefel, *et al.*, *Methods of conjugate gradients for solving linear systems*, vol. 49. NBS Washington, DC, 1952.