Logic Synthesis for Digital In-Memory Computing

Muhammad Rashedul Haq Rashed
Department of ECE
University of Central Florida
Orlando, FL, USA
rashed09@knights.ucf.edu

Sumit Kumar Jha
CS Department
University of Texas at San Antonio
San Antonio, TX, USA
sumit.jha@utsa.edu

Rickard Ewetz
Department of ECE
University of Central Florida
Orlando, FL, USA
rickard.ewetz@ucf.edu

ABSTRACT

Processing in-memory is a promising solution strategy for accelerating data-intensive applications. While analog in-memory computing is extremely efficient, the limited precision is only acceptable for approximate computing applications. Digital in-memory computing provides the deterministic precision required to accelerate high assurance applications. State-of-the-art digital in-memory computing schemes rely on manually decomposing arithmetic operations into in-memory compute kernels. In contrast, traditional digital circuits are synthesized using complex and automated design flows. In this paper, we propose a logic synthesis framework called LOGIC for mapping high-level applications into digital in-memory compute kernels that can be executed using non-volatile memory. We first propose techniques to decompose element-wise arithmetic operations into in-memory kernels while minimizing the number of in-memory operations. Next, the sequence of the in-memory operation is optimized to minimize non-volatile memory utilization. Lastly, data layout re-organization is used to efficiently accelerate applications dominated by sparse matrix-vector multiplication operations. The experimental evaluations show that the proposed synthesis approach improves the area and latency of fixed-point multiplication by 77% and 20% over the state-of-the-art, respectively. On scientific computing applications from Suite Sparse Matrix Collection, the proposed design improves the area, latency and, energy by 3.6X, 2.6X, and 8.3X, respectively.

ACM Reference Format:
Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2022. Logic Synthesis for Digital In-Memory Computing. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549348

1 INTRODUCTION

Scientific simulation of complex physical models is essential for application domains such as finance [5], biology [24], and climate modeling [22]. The simulation of these models often requires massive amount of computational resources over periods of weeks or months [17]. To further scale up the size of such simulations, substantial computing efficiency improvements are required. Unfortunately, with the slow down of Moore's law and the von-Neumann bottleneck, only limited performance gains are expected from further technology scaling [30]. This has resulted in a broad exploration of alternative computing paradigms such as quantum computing [26], optical computing [23], and in-memory computing using non-volatile memory [12]. Processing in-memory computing has attracted an increasing amount of interest due to the energy-efficient computation and ability to break the von-Neumann barrier. Promising non-volatile memory technology include phase change memory (PCM) [3], spin-transfer torque magnetic random-access memory (STT-RAM) [11], and Resistive random access memory (RRAM) or memristor [27, 31].

Processing in-memory can be divided into analog and digital in-memory computing. Analog in-memory computing is extremely efficient but the precision is approximate by nature [10], which is unacceptable for scientific simulation [13]. On the other hand, digital in-memory computing is capable of evaluating logic with deterministic precision. Digital in-memory computing has been explored using logic families such as IMPLY [2], MAGIC [18], Bitwise-in-Bulk [20], and FLOW [15]. The logic styles have been used to accelerate digital circuits and arithmetic operations such as addition, multiplication, and matrix-vector multiplication. The Bit-wise-in-bulk and MAGIC paradigms are capable of executing bitwise operations in parallel, which enables efficient acceleration of arithmetic operations. The difference between the two logic styles is that bitwise-in-bulk performs processing using the peripheral circuitry while MAGIC is completely in-memory, i.e., both the inputs and outputs are stored in memory. Many recent architecture level works focus on MAGIC due to the smaller hardware requirements [13].

The MAGIC logic style is centered on performing NOR operations between data stored in parallel bitlines (or wordlines). To execute elementwise multiplication operations, the elementwise multiplication is required to be decomposed into NOR operations. Manually designed template decompositions have been explored in [8]. The multiplication operations were first decomposed into partial products and multi-bit addition operations. Next, the multi-bit addition operations were decomposed into single bit-addition

Table 1: Area and Latency of 32-bit Arithmetic Operations.

Arithmetic Operation	Work in	Approach	Area (# of Intermediate Storage)	Latency (Time Steps)
Addition	[29]	manual	351	385
Addition	(**proposed**)	synthesis	62	322
Improvement			**82%**	**16%**
Multiplication	[8]	manual	507	12870
Multiplication	(**proposed**)	synthesis	126	10046
Improvement			**75%**	**22%**

ICCAD '22, October 30-November 3, 2022, San Diego,CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549348

and subsequently into NOR operations. The memory utilization was reduced from square to linear by adding the partial products in order. These templates have been used as the underlying building block for numerous architecture level studies [9, 13, 14].

Digital circuits are almost exclusively designed using automated synthesis flows with hundreds of design steps [21]. The synthesis tools map logic into a netlist of connected Boolean gates while minimizing the number of gates and interconnections. While custom synthesis approaches for digital in-memory computing have been explored, they are limited to rather simple circuits. In this paper, we propose to leverage recent advancements within logic synthesis to map arithmetic operations into digital in-memory compute kernels. The use of automated synthesis instead of manually designed templates can potentially lead to remarkable improvements in terms of power, latency, and area. We show the improvements in area and time steps in Table 1.

In this paper, we propose a logic synthesis framework called LOGIC for mapping arithmetic operations to digital in-memory compute kernels. The framework approaches the mapping problem by formulating and solving mathematical optimization problems combined with software/hardware co-design. The contributions of LOGIC can be summarized, as follows:

(1) We propose techniques to decompose element-wise arithmetic operations into in-memory kernels while minimizing the number of in-memory operations. Our experimental investigations show that our in-memory compute kernel synthesis can reduce in-memory operations by 20% for fixed-point multiplication.
(2) We present a new graph-based approach to optimize the sequence of the in-memory operation in order to minimize non-volatile memory utilization. Our experimental studies demonstrate that we reduce the size of the required non-volatile memory by 36% for fixed-point adders.
(3) We present a data layout re-organization algorithm to convert sparse matrix-vector-multiplication (MVM) operations into dense blocks, to improve hardware utilization and reduce inter-crossbar data transfer for in-memory MVM operations.
(4) The LOGIC framework is evaluated on the Suite Sparse Matrix Collection. Compared with state-of-the-art in [1], LOGIC improves area, latency and, energy by 3.6X, 2.6X, and 8.3X, respectively when accelerating scientific computing applications.

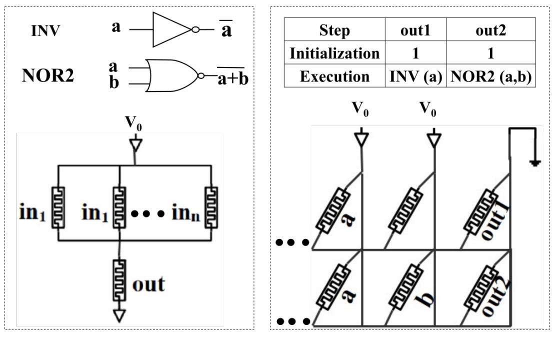
The remainder of this paper is organized as follows: Preliminaries in Section 2. An overview of the framework is given in Section 3. The synthesis steps are detailed in Section 4 and 5. Section 6 describes data layout re-organization algorithm for sparse matrices. The architecture and experimental results are presented in Section 7. The paper is concluded in Section 8.

2 PRELIMINARIES

In this section, we first briefly review basic concepts related to digital in-memory computing. Next, we discuss the state-of-the-art approach for decomposing arithmetic operation into in-memory kernels using manually designed templates.

2.1 Digital In-Memory Computing

MAGIC [18] is one of the most prominent approaches to in-memory computing. In this subsection, we discuss how MAGIC can be used to perform logic NOR operations. Since the NOR logical operation is functionally complete, any Boolean function can be represented using a NOR-only netlist [7]. The MAGIC circuit that can perform INV and NOR operations is shown in Figure 1(a). The memristors in_1 to in_n represent inputs of the NOR gate, and the *out* memristor realizes the output of the NOR operation. The MAGIC operation is performed in two steps: one initialization step, and one execution step, which are shown on top of Figure 1(b). In the initialization step, the n input memristors are set either to high resistance state (HRS) which corresponds to logic "0", or to low resistance state (LRS) which corresponds to logic "1". The *out* memristor is set to LRS. In the execution step, a controlled voltage V_0 is applied to the input memristors, and the output of NOR operation is realized in the *out* memristor. Parallel NOR operations can be performed when the memristors are arranged in crossbars. The execution of a parallel INV and NOR2 operation is shown at the bottom of Figure 1(b).



Step	out1	out2
Initialization	1	1
Execution	INV (a)	NOR2 (a,b)

(a) (b)

Figure 1: (a) Digital in-memory computing using MAGIC and, (b) parallel MAGIC operations in a crossbar.

2.2 Manually Designed Templates for Arithmetic Operations

In this subsection, we review the standard approach of decomposing arithmetic operations into digital in-memory compute kernels. The state-of-the-art in-memory computing paradigms use a manually crafted template based approach. For instance, a 1-bit full adder can be decomposed using NOR-only operations. The carry-out bit can for example be written as three 2-input NORs followed by one 3-input NOR.

$$C_{out} = \overline{\overline{(A+B)} + \overline{(A+C_{in})} + \overline{(B+C_{in})}}$$

$$Sum = \overline{\overline{\overline{\overline{(\overline{A}+\overline{B}+\overline{C_{in}})} + \{\overline{\overline{(A+B+C_{in})} + C_{out}}\}}}}$$

Here, A and B are the input operands and C_{in} and C_{out} are the carry-in and carry-out bits respectively. The in-memory adder operation is performed in a single row of crossbar as shown in Figure 2(a). This approach partitions the memristors cells in a row into three groups [29]. The input and the output cells store the input and the output of the adder respectively. The functional cells generate the *Sum* and C_{out} through sequential NOR operations.

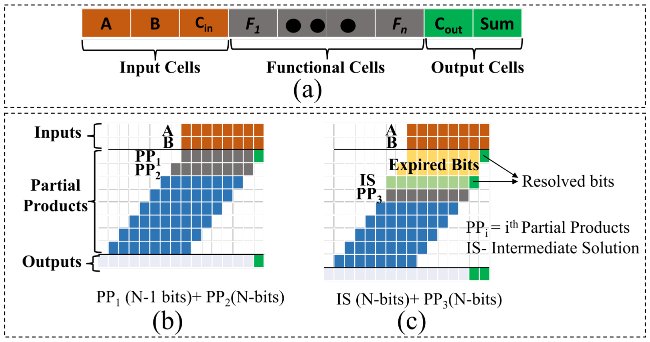


Figure 2: State-of-the-art schemes for (a) MAGIC arithmetic and (b)-(c) addition of partial products of multiplication.

The adder operation can be extended to n-input multiplication operation [8] in a straightforward manner. This is done by applying the adder operation on the partial products of multiplication. For example, the workflow of multiplication of two multi-bit operands A and B is shown in (b) and (c) of Figure 2. The addition of first two partial products PP_1 and PP_2 is shown in Figure 2(b). The addition leads to the first intermediate solution, IS. In the next step, the intermediate solution IS is added to the third partial product PP_3 which is shown in Figure 2(c). The process is continued until all the partial products are covered.

3 THE LOGIC FRAMEWORK

In this section, we introduce the LOGIC framework. A high-level overview of the framework is shown in Figure 3.

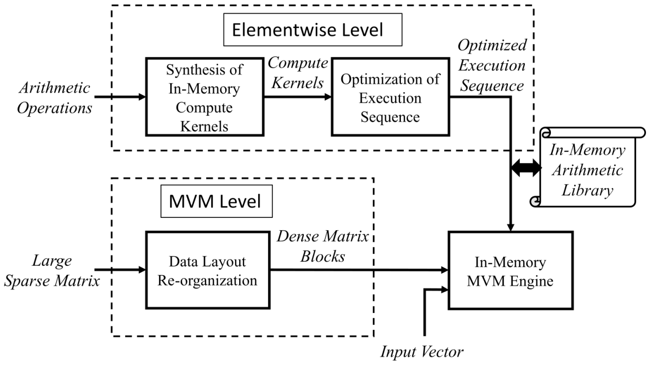

Elementwise Level
Arithmetic Operations
Synthesis of In-Memory Compute Kernels
Compute Kernels
Optimization of Execution Sequence
Optimized Execution Sequence
In-Memory Arithmetic Library
MVM Level
Large Sparse Matrix
Data Layout Re-organization
Dense Matrix Blocks
In-Memory MVM Engine
Input Vector

Figure 3: Overview of the LOGIC framework

The objective of the framework is to accelerate sparse matrix-vector multiplication operations using digital in-memory computing. The framework consists of three parts: (a) synthesis of in-memory compute kernels, (b) execution sequence optimization, and (c) data layout re-organization. The synthesis of in-memory compute kernels step takes an arithmetic operation as input and synthesises it to in-memory kernels, which is equivalent to a traditional netlist. The objective of this step is to minimize the number of in-memory operations. The execution sequence optimization takes the netlist of in-memory kernels as input and provides an execution sequence as the output. The objective of the optimization is to minimize the required amount of non-volatile memory needed for intermediate storage. The synthesis of in-memory compute kernels together with the execution sequence optimization is used to build a library of in-memory execution sequences for arithmetic operations of different bitwidths. The data layout re-organization takes a large sparse MVM operations as input and re-organizes the computation into dense blocks that can be efficiently accelerated with the pre-characterized library of arithmetic operations.

The components of the LOGIC framework are described in Section 4, Section 5, and Section 6, respectively. The architecture of the in-memory computing platform is given in Section 7.

4 SYNTHESIS OF IN-MEMORY COMPUTE KERNELS

In this section, we explain how arithmetic operations such as elementwise addition and multiplication are compiled into INV and NOR operations. The user specifies the maximum number of inputs that the NOR operations support based on the specific technology. The synthesis consists of a (i) hierarchical multiplier and adder decomposition step, a (ii) traditional logic synthesis step, and a (iii) technology mapping step. The flow is shown in Figure 4. The first step hierarchically decomposes the elementwise multiplication into partial product computations and adder operations. The second step converts each hierarchical component into in-memory compute kernels using traditional logic synthesis using ABC [21]. (We omit the details of this step as it is performed directly using ABC). The last step is to convert the initial netlist to an optimized netlist with high fan-in gates.

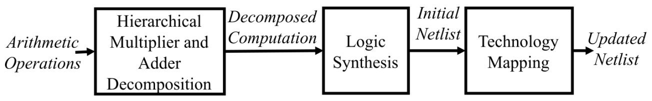


Figure 4: Synthesis of in-memory compute kernels.

4.1 Hierarchical Multiplier and Adder Decomposition

In this section, we hierarchically decompose elementwise multiplications into smaller components. Ideally, we would pass the entire multiplication into a traditional logic synthesis tool such as ABC in the next step. However, netlists of arithmetic operations scale exponentially in size with the bit-width. Therefore, a hierarchical decomposition approach is needed.

We decompose an n-bit elementwise multiplication operation into partial products and m-bit additions, as shown in Figure 2. The exact value of m is dependent on the number of partial products that have been added. However, instead of decomposing each

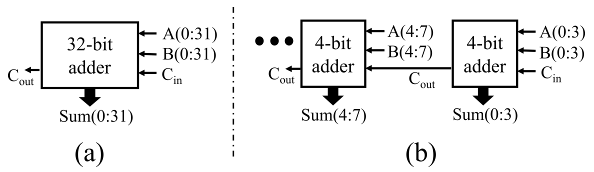


Figure 5: Decomposing arithmetic operations. (a) a 32-bit full adder and, (b) 4-bit full adder chain.

m-bit addition into m 1-bit additions, we explore using multi-bit adders, which is shown in Figure 5. Moreover, instead of manually decomposing the multi-bit adders into INV and NOR operations, we utilize traditional logic synthesis such as ABC [21].

We compare the intermediate memory utilization/bit and number of operations/bit with respect to adders with different bit-width in Figure 6. It can be observed that the intermediate memory utilization grows linearly with the adder bit-width. On the other hand, the number of operations/bit decreases until a bit-width of 8, next the number of operations start to increase again with respect to the bit-width. The noise in the plot is a result of that the ABC tool is based on heuristics. Based on these observations, we select the target adder bit-width to 8 bits, in order to minimize expected energy and delay.

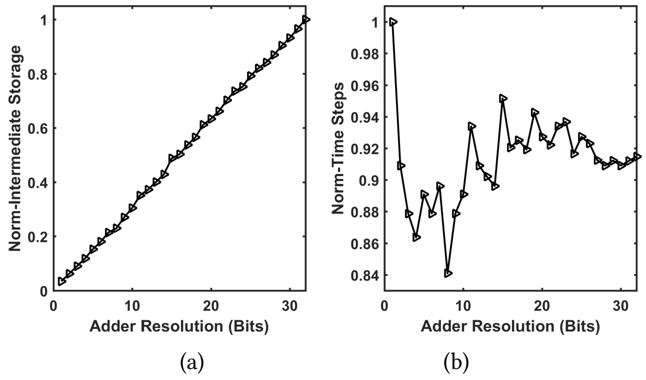


Figure 6: Intermediate storage and time steps with respect to adder bit-width.

4.2 Technology Mapping

In this subsection, we describe the technology mapping step. In this step, the synthesis tool converts the low fan-in NOR-gates of the initial netlist into higher fan-in NOR-gates with the goal of reducing area and latency.

The technology mapping step is explained with an example netlist in Figure 7. First, the NOR-only netlist is converted into a directed acyclic graph (DAG), $G = (V, E)$, which is shown in Figure 7(a). The corresponding gate encoding of nodes of the DAG are shown on the right of Figure 7(a). Next, a cell library of higher fan-in NOR-gates are generated. Each cell of this library is a superset of the rudimentary NOR-gates of the initial netlist. A subset of the cell library is shown in Figure 7(b). Next, the DAGON [16] algorithm is used to cover the DAG using cells from the cell library. An example cover is shown in Figure 7(c). The updated netlist after technology mapping is shown in Figure 7(d). In this example, the total number of nodes of the initial netlist is reduced from 16 to only 4 nodes in the updated netlist. Our approach translates into improvement in latency due to the choice of our bit-width resolutions and the reduction of the number of nodes in the netlist.

5 EXECUTION SEQUENCE OPTIMIZATION

In this section, we explain the execution sequence optimization step. Next, we outline an optimal algorithm based on enumeration. Lastly, we provide a practical greedy approach to solve the sequence ordering.

5.1 Problem Formulation

The input is a netlist represented using a DAG, $G = (V, E)$, where the nodes V correspond to in-memory operations and the edges E correspond to predecessor constraints. The objective is to process the operations while minimizing the required intermediate storage. Let S denote a sequence of the $|V|$ operations. This can be formalized, as follows:

$$\min_{S} \max_{i=\{0,\ldots,|V|\}} cost(S_i) \quad (1)$$
$$s.t. \quad n_i \leq n_j \quad \forall (i,j) \in E,\ n_i, n_j \in V$$

where S_i is the first i nodes in S. The $cost(S_i)$ is the intermediate storage required after i nodes have been processed. The $cost(S_i)$ is computed by forming a set $U_i = V \setminus S_i$. Next, let W_i be the set of nodes in S_i that have at least one edge connected to a node in U_i. The nodes in S_i with no connections to a node in U_i have been consumed and are not required to be stored. The $cost(S_i)$ is finally defined to be $|W_i|$.

5.2 Enumeration of Execution Sequences

In this step, we outline a solution to the execution sequencing problem based on enumeration. The DAG representation of a half adder is shown in Figure 8(a). All possible topological sortings of the nodes are shown in Figure 8(b). $\boxed{a}$ and $\boxed{b}$ represents the primary inputs and nodes c−g represents the gates of the netlist. We show the $cost(S_i)$ from Eq (1) for the left-most and the right-most sequences in the figure. Note that, after a node n is processed, the inputs of n that are no longer needed for future processing are expired and the corresponding functional memristors are released and ready

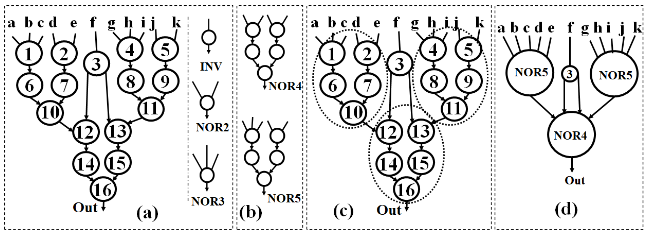


Figure 7: Technology mapping: (a) initial netlist with gate encoding, (b) library of in-memory compute kernels, (c) cover of subject graph, and (d) optimized netlist.

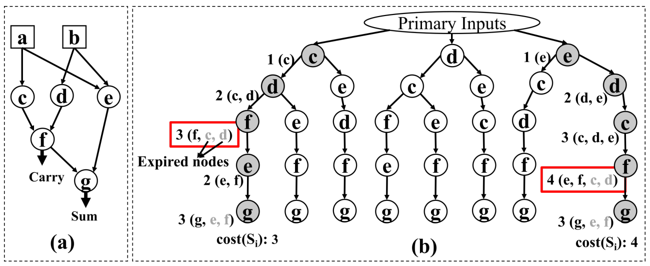


Figure 8: (a) DAG representation of a half adder and, (b) enumeration of all feasible execution sequences.

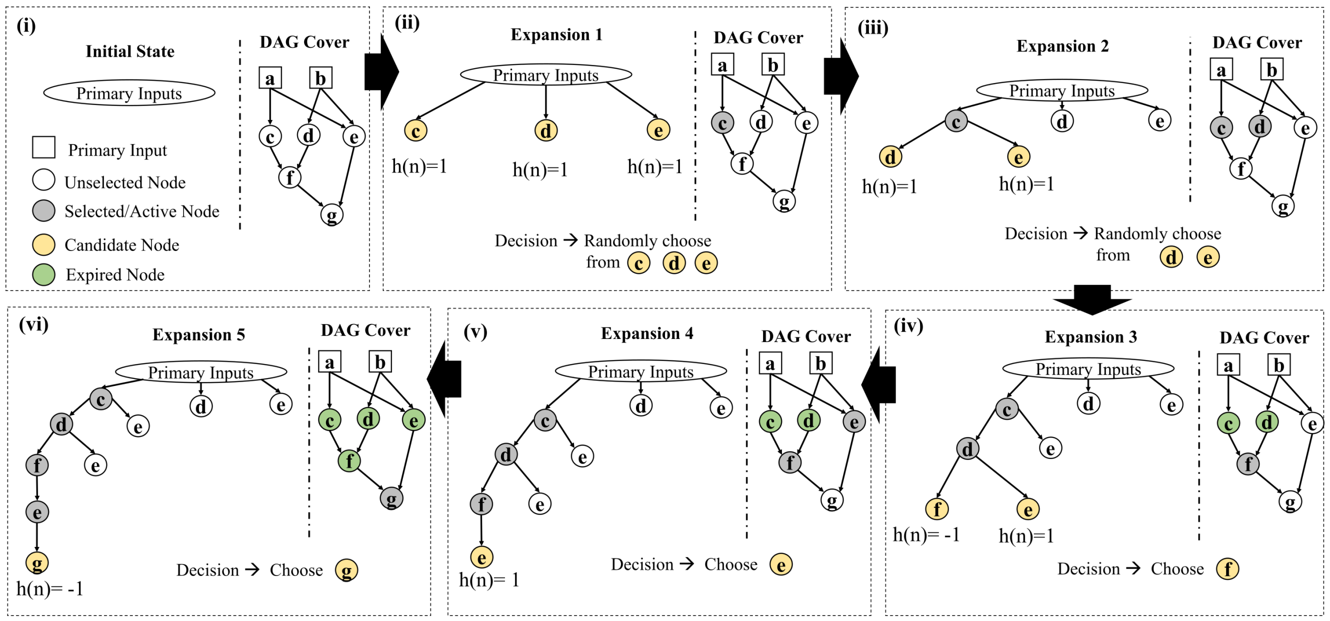


Figure 9: Greedy algorithm for execution sequence ordering.

for reuse. In the figure, the $cost(S_i)$ for sequence c⟶d⟶f⟶e⟶g is 3 and the $cost(S_i)$ for sequence e⟶d⟶c⟶f⟶g is 4. The synthesis tool selects the sequence that minimizes $cost(S_i)$.

While enumeration of all the sequences of a DAG would ensure that an optimum cost sequence is selected, it becomes computationally impractical for DAGs with many nodes. For example, the number of total sequences for a half adder, a 1-bit full adder and, a 2-bit full adder are 8, 856 and, 26795 respectively. Therefore, for adders with higher bit-width resolutions, we implement a greedy algorithm to generate a relatively small number of sequences.

5.3 Greedy Algorithm

In this step, we propose a greedy algorithm for generating in-memory execution sequences. To generate a sequence that has the best chance of incurring minimum cost, the greedy algorithm makes informed decision before including a node into the sequence. To achieve this, we define a new parameter called *effective inclusion cost (EIC), h(n)*. Here, $h(n)$ = *(memory cost of including n)–(memory released after processing n)*. The algorithm is presented in Algorithm 1. For each expansion of a sequence, the algorithm first evaluates the $h(n)$ for each of the candidate nodes. Here the candidate nodes are the nodes that satisfy the predecessor constraint. The algorithm selects the candidate node that yields the minimum $h(n)$. If there are multiple candidate nodes with minimum $h(n)$, the algorithm selects one of these nodes in random. The algorithm concludes when all the nodes within the DAG are covered. We illustrate the algorithm with an example in Figure 9. The figure shows the generation of a greedy sequence for the DAG of a half adder. The synthesis tool memoizes the $cost(S_i)$ of the generated sequence and applies a branch-and-bound concept for the future sequence generations [4]. Whenever the $cost(n_{ij})$ of a sequence exceeds $min(cost(S_i))$, the expansion of the sequence is terminated and the algorithm starts a new sequence generation.

Algorithm 1: Algorithm for Priority Traversal of the DAG for Generating Executions

```
Inputs: DAG, $G = (V, E)$
Output: Greedy execution sequence, $\mathcal{S}$;
main {
$\mathcal{S} \leftarrow \phi$; \\initializing
$Memo(0) \leftarrow \infty$; \\for memoization of sequence cost
$N \leftarrow$ total nodes in G;\\excluding primary inputs
for $i=0$ to $M$ do
    $\mathcal{P}, C \leftarrow \phi$; \\initializing
    while $size(\mathcal{P}) \neq N$ do
        $\mathcal{H}, \mathcal{R} \leftarrow \phi$; \\initializing
        $C \leftarrow$ all candidate nodes;
        for all $j \in C$ do
            $\mathcal{H}(j) \leftarrow h(j)$;\\calculating EIC
        end
        $\mathcal{R} \leftarrow$ all $k \in C$ with $min(\mathcal{H})$;\\pruning
        $n \leftarrow$ rand(R);\\informed random selection
        $\mathcal{P} \leftarrow \mathcal{P} \cup n$;\\ordered execution set
        if $cost(n) \geq min(Memo)$ then
            break; \\branch and bound
        end
    end
    if $cost(\mathcal{P}) < min(Memo)$ then
        $\mathcal{S} \leftarrow \mathcal{P}$;
    end
    $Memo(i+1) \leftarrow cost(\mathcal{P})$; \\memoization
end
return $\mathcal{S}$;
}
```

6 DATA LAYOUT RE-ORGANIZATION

In this section, we develop a data layout re-organization algorithm to decompose in-memory matrix-vector-multiplication (MVM) of sparse matrices. The aim of the algorithm is to improve the hardware utilization and to reduce inter-crossbar data transfer for in-memory MVM operations.

State-of-the-art approach to decompose MVM into crossbar is to arrange the matrix operands row-wise into the crossbar and perform row-parallel arithmetic operations [1, 13]. This row-wise layout ensures that all the arithmetic operands of an output vector element are aligned in the same crossbar row and therefore no inter-row copy operation is required. This approach works very well with dense matrices like the *Journals* matrix shown in Figure 10(a).

However, most matrices within physical systems are sparse. A sparse matrix *eris*1176 is shown in Figure 10(b). We make the observation that a naive row-wise data alignment of sparse matrix leads to hardware under-utilization. For efficient utilization of hardware, a denser data layout is desired. For instance, a row-wise shifting of the operands of the *eris*1176 matrix is shown in Figure 10(c). This shifted alignment ensures that the matrix operands will achieve a denser assignment in crossbars which would translate as less hardware requirement and fewer inter-crossbar data communications.

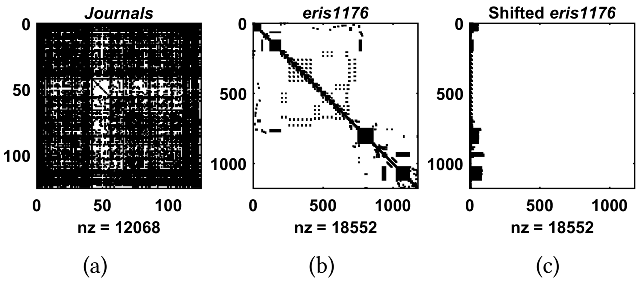


Figure 10: Matrices from the Suitsparse matrix collection [6]. (a) Dense matrix *Journals*, (b) sparse matrix *eris*1176 and, (c) row-wise shifted *eris*1176 matrix

In this new approach with potentially extreme shifting, the routing of corresponding input vector operands become an interesting problem as each row of the shifted matrix is multiplied with an unique subset of the original input vector. Therefore, unlike a regular dense matrix, the routing resources cannot be shared among the rows of a crossbar.

To improve the routability of the input vectors, we propose a data layout re-organization algorithm. We explain the algorithm with the example matrix segment of Figure 11(i). The aim of the algorithm is to reorganize the matrix operands in a fashion that all the rows of a crossbar can share the same set of routing resources for the incoming input vectors. To achieve this, we first do a *padding* operation on the columns of the matrix that contain at least one non-zero element. The *padding* operation is shown in Figure 11(ii). Next we perform a shifting operation on the matrix columns with non-zero elements which is shown in Figure 11(iii). Finally, we clean the padded elements from the matrix as shown in Figure 11(iv). Note that, the set of shifted column indices for both rows is {2,4,6}. Therefore, both the shifted rows are multiplied with 2nd, 4th and

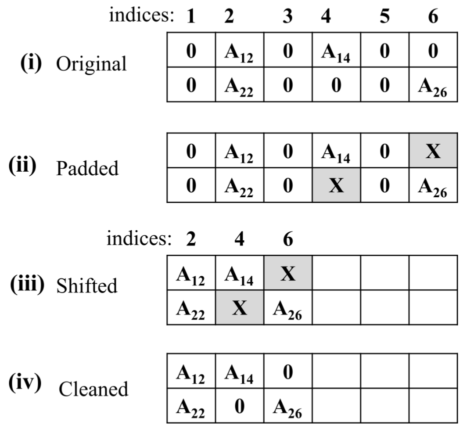


Figure 11: Sparse matrix data layout reorganization

6th elements of input the vector. For a crossbar dimension of $P \times Q$, the algorithm ensures that all P rows of the crossbar share the same subset of the input vector.

The data layout re-organization on the sparse matrix *eris*1176 is shown in Figure 12. Figure 12(a) shows the *padding* operation on the matrix. The padding increases the number of nonzeros to 224,992 from the original number of nonzeros of 18,552. Next, the *shifting* and *cleaning* operations are shown in Figure 12(b) and (c) respectively. After cleaning, the number of non-zeros are equal to the number of non-zeros in the original matrix. Finally, a blocking of the shifted non-zero operands is shown in Figure 12(d). In our architecture, we consider crossbars of 128×128 dimension. In Figure 12(d), each block has 128 rows which is equal to the number of rows in the crossbar. Each of the block is multiplied with an unique subset of input vector elements which are parallelly routed to all the rows of the corresponding crossbars.

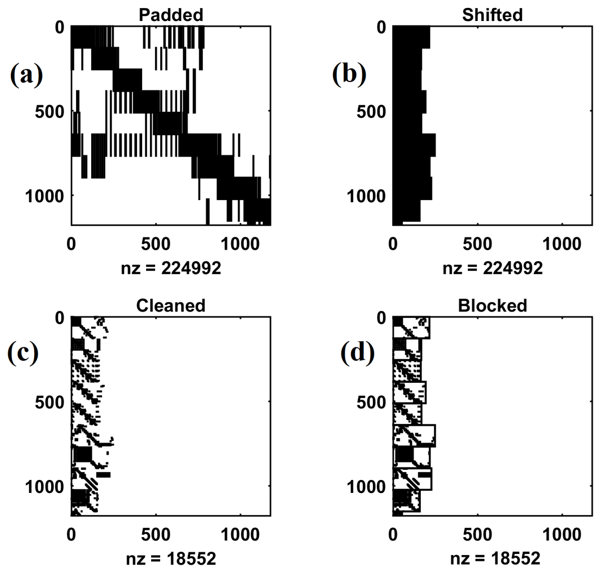


Figure 12: Data layout re-organization of *eris*1176 matrix.

7 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of the LOGIC framework. We first present the architecture for the LOGIC in-memory computing platform. Next, we experimentally evaluate the performance of LOGIC.

7.1 Architecture

An overview of the architecture of the LOGIC framework is shown in Figure 13(a). The overall architecture consists of several accelerator tiles. Each accelerator tile is equipped with several processing elements (PE). Input registers (IR) and output registers (OR) are used to convey input and output operands respectively. EDRAM buffer is used to store intermediate results and routing indices of MVM operands. High speed bus is used to communicate among the PEs, the storage units and I/O interface. The components of a PE are detailed in Figure 13(b). A PE consists of many memristor crossbars arranged in a row-parallel fashion. The dimension of each crossbar is 128 × 128. Drivers and routing blocks (RB) are used to program the crossbars. Sense amplifiers (SA) and row-parallel-copying (RPC) circuits are used to achieve inter-crossbar data transfer [13]. The cross-section of a routing block is shown in Figure 13(c). Each intersection of nanowire within the RB contains a memristor. The memristors work as routing switches within the RB.

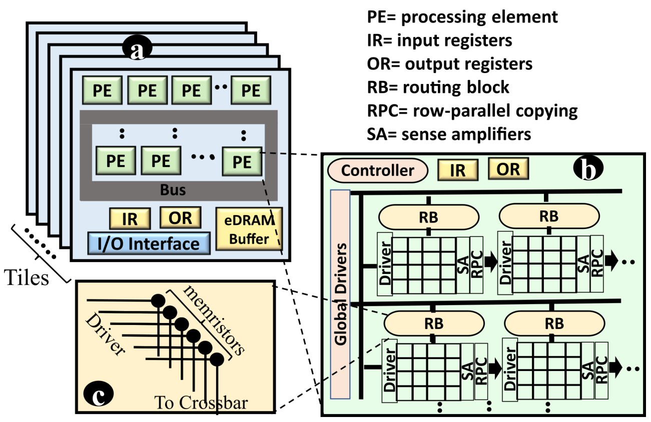


Figure 13: (a) Overview of the architecture, (b) components of a processing element and, (c) routing block architecture

The architecture offers high order of parallelism for arithmetic operations. For instance, all of the decomposed blocks of a MVM operation can be performed in parallel where each of the blocks is assigned to a set of row-parallel crossabrs.

7.2 Experimental Evaluation

In this section, we experimentally evaluate the performance of the LOGIC framework. We use an octa core 3.60 GHz Intel Core i9 processor with NVIDIA RTX 2070 and 64 GB RAM for our experiments. The synthesis tool is developed using a combination of C++ and MATLAB scripts. For converting decomposed arithmetic operations into initial Boolean netlist, we use the ABC tool [21].

The per-unit cost of different architectural components are summarized in Table 2. The area and power costs are appropriately adapted from previous works [13, 14, 19, 25]. The cross architecture

Table 2: Area-Power Cost of Architectural Components

Component	Parameter	Specs	Area	Power
Crossbar	size	128 × 128	25 μm^2	0.30 mW
Sense Amp.	# unit	128	7.14 μm^2	0.29 mW
Controller	# unit	1	400 μm^2	0.65 mW
RB	# unit	1	12 μm^2	0.01 mW
eDRAM Buffer	size	128 KB	0.17 mm^2	41.40 mW
IR	size	16 B	16.80 μm^2	0.01 mW
OR	size	16 B	46.88 μm^2	0.02 mW
Bus	bandwidth	128-bits	15.70 mm^2	13 mW
Local Bus	#wires	128	0.03 mm^2	2.33 mW

data communication cost between crossbars is adapted from the case study of [28].

We first evaluate the performance of the LOGIC framework for elementwise operations in Section 7.2.1. Next, we evaluate the performance of the framework for scientific computing application on benchmarks of Suite Sparse Matrix Collection [6] in Section 7.2.2. The overview of the benchmarks are listed in Table 3.

Table 3: Overview of benchmarks from the SuiteSparse Matrix Collection.

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

7.2.1 *Elementwise Arithmetic.*

A performance comparison between the proposed framework and the state-of-the-art in-memory arithmetic paradigm SIMPLER [1] is shown in Figure 14. We compare the area-latency overhead of the two paradigms for elementwise arithmetic operation. The results show, the framework improves

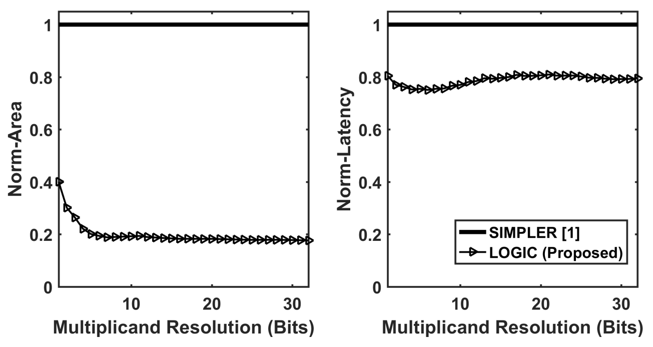


Figure 14: Area-latency overhead comparison of the proposed synthesis approach with the state-of-the-art in-memory computing approach SIMPLER [1].

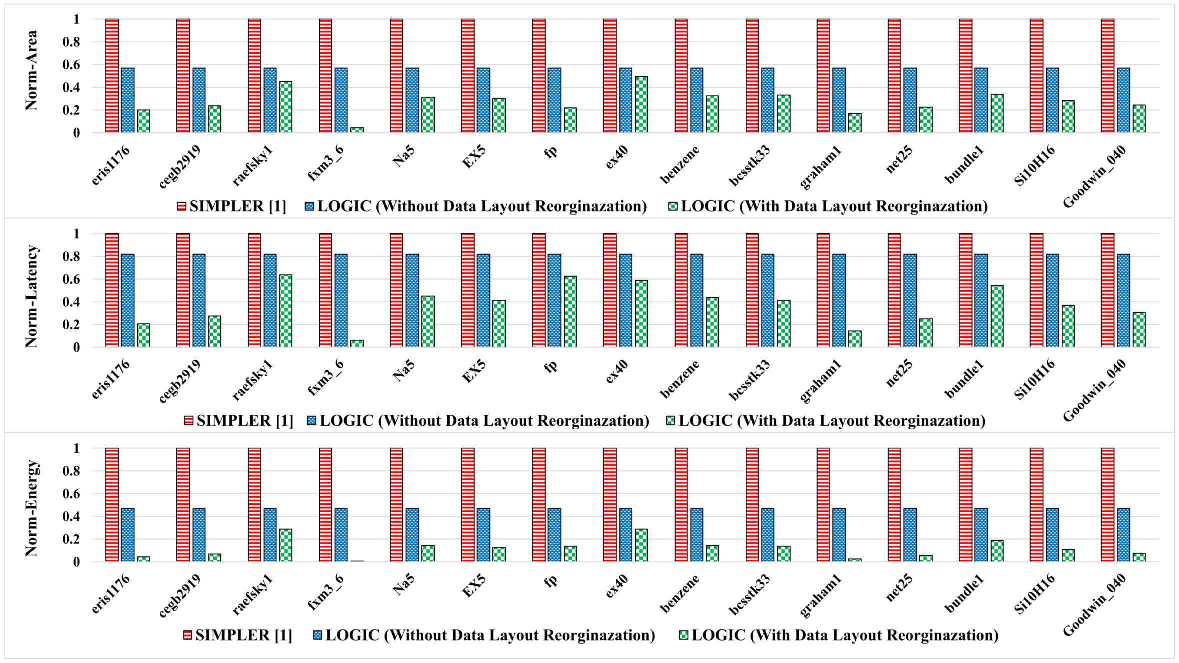


Figure 15: Area-latency-energy overhead evaluation of LOGIC and the state-of-the-art in-memory computing approach SIMPLER [1] on 15 applications from the suite sparse matrix collection [6].

the area by 77% on average for fixed-point multiplication with different bit-resolutions. The framework also achieves a latency improvement of 20% on average. The improvements are achieved in different steps of the synthesis. For instance, the technology mapping step reduces the number of total gates on average by 4%. This reduction in gates translates into reduction of MAGIC operation and therefore a reduction in latency. Also, compared with random execution sequence, the greedy algorithm generated execution sequence reduces the intermediate storage cost by 36%. The experimental results validate that the proposed synthesis approach provides superior performance over manual decomposition based in-memory computing paradigms for elementwise arithmetic operations.

7.2.2 Evaluation of Scientific Computing Application. In this section, we evaluate the LOGIC framework using scientific computing applications dominated by sparse MVM operations. For this evaluation, we select 15 sparse matrices in Table 3 from different domains and sparsity patterns from the Suite Sparse Matrix collection [6]. We evaluate the area-latency-energy cost of the LOGIC framework before and after data-layout reorganization. We compare the results with state-of-the-art paradigm for MVM operations [1]. The comparative performance in terms of area, latency, and energy is provided in Figure 15.

Compared with SIMPLER [1], LOGIC (without data layout reorganization) improves the area, latency, and energy by 1.8X, 1.2X, and 2.1X respectively. These improvements stem from that each elementwise multiplication operation is performed using fewer operations and with smaller area. Compared with LOGIC (without data layout re-organization), LOGIC (with data layout organization) improves area, latency, and energy with 2X, 2.2X, and 3.9X, respectively. The improvements come form more efficiently aligning the matrix data and input vectors within the in-memory computing platform.

In summary, the proposed logic synthesis based approaches demonstrate significant advantages over the state-of-the-art approach SIMPLER [1], which is based on manually designed templates. The improvements come from leveraging the advancements within logic synthesis from the past decades.

8 SUMMARY AND FUTURE WORK

In this paper, we have proposed a framework called LOGIC for mapping high-level applications to digital in-memory compute kernels. The framework includes automated techniques to minimize the number of in-memory computing operations. The ordering of the execution of the in-memory operations are next optimized to minimize the utilization of non-volatile memory for storage of intermediate data. Finally, a data layout re-organization technique is proposed to compress sparse MVM operations into dense MVM operations to minimize inter crossbar communication. Compared with manually designed template based approaches, the area, latency, and energy is improved with 3.6X, 2.6X, and 8.3X, respectively. In our future work, we plan to extend LOGIC to automatically decompose entire multiplication without a hierarchical approach. We also plan to combine LOGIC with other digital in-memory computing logic styles.

ACKNOWLEDGMENTS

This work was in part supported by NSF awards #1755825, #1908471, #2008339, #2113307, DARPA cooperative agreement #HR00112020002, and ONR grant #N000142112332.

REFERENCES

[1] R. Ben-Hur et al. Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput. *TCAD*, 39(10):2434–2447, 2019.

[2] J. Borghetti et al. 'memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464(7290):873–876, 2010.

[3] G. W. Burr et al. Phase change memory technology. *JVSTB*, 28(2):223–262, 2010.

[4] J. Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[5] J. S. Dagpunar. *Simulation and Monte Carlo: With applications in finance and MCMC*. John Wiley & Sons, 2007.

[6] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *TOMS*, 38(1):1–25, 2011.

[7] H. B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[8] 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)*, pages 1–5. IEEE, 2018.

[9] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky. Not in name alone: A memristive memory processing unit for real in-memory processing. *IEEE Micro*, 38(5):13–21, 2018.

[10] 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*, pages 1–6. IEEE, 2016.

[11] Y. Huai et al. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS bulletin*, 18(6):33–40, 2008.

[12] D. Ielmini and H.-S. P. Wong. In-memory computing with resistive switching devices. *Nature electronics*, 1(6):333–343, 2018.

[13] M. Imani, S. Gupta, Y. Kim, and T. Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA*, pages 802–815. IEEE, 2019.

[14] 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)*, pages 356–371. IEEE, 2020.

[15] S. K. Jha, D. E. Rodriguez, J. E. Van Nostrand, and A. Velasquez. Computation of boolean formulas using sneak paths in crossbar computing, 2016. US Patent 9,319,047.

[16] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC*, pages 341–347, 1987.

[17] S. Koziel, L. Leifsson, and X.-S. Yang. *Solving computationally expensive engineering problems: methods and applications*, volume 97. Springer, 2014.

[18] S. Kvatinsky et al. Magic—memristor-aided logic. *TCAS-II: Express Briefs*, 61(11):895–899, 2014.

[19] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny. Vteam: A general model for voltage-controlled memristors. *TCAS-II: Express Briefss*, 62(8):786–790, 2015.

[20] S. Li et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *DAC*, pages 1–6. IEEE, 2016.

[21] A. Mishchenko et al. Abc: A system for sequential synthesis and verification. "http://www.eecs.berkeley.edu/alanmi/abc".

[22] R. A. Pielke Sr. *Mesoscale meteorological modeling*. Academic press, 2013.

[23] A. A. Sawchuk and T. C. Strand. Digital optical computing. *Proceedings of the IEEE*, 72(7):758–779, 1984.

[24] B. Schölkopf, K. Tsuda, and J.-P. Vert. *Kernel methods in computational biology*. MIT press, 2004.

[25] A. Shafiee et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *SIGARCH*, 44(3):14–26, 2016.

[26] A. Steane. Quantum computing. *Reports on Progress in Physics*, 61(2):117, 1998.

[27] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[28] 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)*, pages 1628–1633. IEEE, 2018.

[29] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky. Logic design within memristive memories using memristor-aided logic (magic). *IEEE Transactions on Nanotechnology*, 15(4):635–650, 2016.

[30] M. V. Wilkes. The memory wall and the cmos end-point. *SIGARCH*, 23(4):4–6, 1995.

[31] S. Yu. Resistive random access memory (rram). *Synthesis Lectures on Emerging Engineering Technologies*, 2(5):1–79, 2016.