LOGIC: Logic Synthesis for Digital In-Memory Computing

MUHAMMAD RASHEDUL HAQ RASHED, University of Texas at Arlington, USA
SVEN THIJSSEN, Florida Atlantic University, USA
SUMIT KUMAR JHA, Florida International University, USA
RICKARD EWETZ, University of Florida, USA

In-memory processing offers a promising solution for enhancing the performance of data-intensive applications. While analog in-memory computing demonstrates remarkable efficiency, its limited precision is suitable only for approximate computing tasks. In contrast, digital in-memory computing delivers the deterministic precision necessary to accelerate high-assurance applications. Current digital in-memory computing methods typically involve manually breaking down arithmetic operations into in-memory compute kernels. In contrast, traditional digital circuits are synthesized through intricate and automated design workflows. In this paper, we introduce a logic synthesis framework called LOGIC, which facilitates the translation of high-level applications into digital in-memory compute kernels that can be executed using non-volatile memory. We propose techniques for decomposing element-wise arithmetic operations into in-memory kernels while minimizing the number of in-memory operations. Additionally, we optimize the sequence of in-memory operations to reduce non-volatile memory utilization. To address the NP-hard execution sequencing optimization problem, we have developed two *look-ahead* algorithms that offer practical solutions. Additionally, we leverage data layout re-organization to efficiently accelerate applications that heavily rely on sparse matrix-vector multiplication operations. Our experimental evaluations demonstrate that our proposed synthesis approach improves the area and latency of fixed-point multiplication by 84% and 20% compared to the state-of-the-art, respectively. Moreover, when applied to scientific computing applications sourced from the SuiteSparse Matrix Collection, our design achieves remarkable improvements in area, latency, and energy efficiency by factors of 4.8×, 2.6×, and 11×, respectively.

CCS Concepts: • **Hardware** → **Emerging technologies**; **Electronic design automation**; • **Computing methodologies**;

Additional Key Words and Phrases: In-Memory Computing, Execution Sequence Optimization

ACM Reference Format:
Muhammad Rashedul Haq Rashed, Sven Thijssen, Sumit Kumar Jha, and Rickard Ewetz. 2024. LOGIC: Logic Synthesis for Digital In-Memory Computing. 1, 1 (December 2024), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In the past decade, there has been an enormous surge in the volume of digital data worldwide [14, 49, 60]. This abundance of data has given rise to data-driven applications like large language models (LLMs) [7], computer vision [64], metaverse [2] and digital twin [28]. These new applications are fueling innovations across various fields, including military applications [5, 39, 54], intelligent transportation systems [38, 48], medical applications [59, 65], and industries [27, 29].

Authors' Contact Information: Muhammad Rashedul Haq Rashed, Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas, USA, muhammad.rashed@uta.edu; Sven Thijssen, Department of Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, Florida, USA, sthijssen@fau.edu; Sumit Kumar Jha, Knight Foundation School of Computing and Information Sciences, Florida International University, Miami, Florida, USA, sumit.jha@fiu.edu; Rickard Ewetz, Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida, USA, rewetz@ufl.edu.

Manuscript submitted to ACM

Table 1. Area-Latency costs for 32-bit arithmetic operations for manual design based approach and the proposed synthesis based approach. $LOGIC_{baseline}$ is the preliminary version of the proposed framework which was published in [45].

Arithmetic Operation	Work in	Approach	Area (# of Intermediate Storage)	Latency (Time Steps)
Addition	[57]	manual	351	385
Addition	$LOGIC_{baseline}$ [45]	synthesis	62	322
Addition	**LOGIC (this work)**	synthesis	42	322
Improvement over manual approach			**88%**	**16%**
Improvement over synthesis approach			**32%**	0%
Multiplication	[16]	manual	507	12870
Multiplication	$LOGIC_{baseline}$ [45]	synthesis	126	10046
Multiplication	**LOGIC (this work)**	synthesis	106	10046
Improvement over manual approach			**79%**	**22%**
Improvement over synthesis approach			**16%**	0%

However, these data-intensive applications place immense demands on computational resources, creating an unprecedented burden on modern computing systems [19, 43, 55]. This challenge is exacerbated by the stagnation in technology scaling, marked by the end of Moore's Law and Dennard scaling [12, 62].

As a response to this pressing issue of technological stagnation, new computing systems are being developed. Emerging computing domains such as quantum computing [42], hyper-dimensional computing [18], optical computing [61], and in-memory computing [63] are ushering in a new era of computational capabilities. Among these, in-memory computing is gaining significant attention from the research community due to its potential to overcome the von Neumann bottleneck [67, 68]. In-memory computing has been investigated using both conventional [15, 41] and novel memory devices [47]. The emerging non-volatile memory technologies are particularly appealing because they allow for complete in-place computation. Promising non-volatile memory technology include resistive random access memory (RRAM) or memristor [53, 70], spin-transfer torque magnetic random-access memory (STT-MRAM) [22], phase change memory (PCM) [6], and ferroelectric hafnium oxide-based FET (FeFET) [1].

In-memory processing can be categorized into two main subdomains: analog and digital in-memory computing [44, 71]. Analog in-memory computing stands out for its exceptional efficiency [13, 20, 36, 52]. However, its analog nature is inherently limited to approximate results, rendering it unsuitable for high precision demanding scientific simulations [33, 69]. In contrast, digital in-memory computing offers the ability to carry out logical operations with deterministic precision [8, 66]. Various logic families like IMPLY [4], MAGIC [31], Bitwise-in-Bulk [37], FLOW [26], STREAM [46] and PATH [58] have been explored within the domain of digital in-memory computing. These logic styles are capable of accelerating digital circuits and arithmetic operations, including addition, multiplication, and matrix-vector multiplication (MVM). Notably, the Bit-wise-in-bulk and MAGIC paradigms are particularly attractive for their capacity to perform parallel execution of bitwise operations. This facilitates the efficient acceleration of highly parallelizable arithmetic tasks such as MVM [35, 51]. The key distinction between these two logic styles lies in their operational methodology: bitwise-in-bulk engages peripheral circuitry for processing, whereas MAGIC operates entirely in-memory, storing both inputs and outputs within the memory. Due to its reduced hardware cost, MAGIC has been focused on a number of recent architecture-level studies [3, 17, 24, 25, 72].

The core operation within the MAGIC logic style is the execution of NOR operations among data stored in parallel bitlines (or wordlines). In order to carry out fixed-point multiplication, it is required to break down the computation into NOR operations. Decomposition of arithmetic operations into MAGIC netlists have been explored using manually

crafted templates in [16, 57]. In manual approach, the multiplication operation is first decomposed into partial products and multi-bit addition operations. Subsequently, the multi-bit addition operations are further decomposed into single-bit additions and eventually into NOR operations. By sequentially adding the partial products, memory utilization was transformed from quadratic to linear. These manually designed templates have served as the foundational components for many architectural-level studies [17, 24, 25].

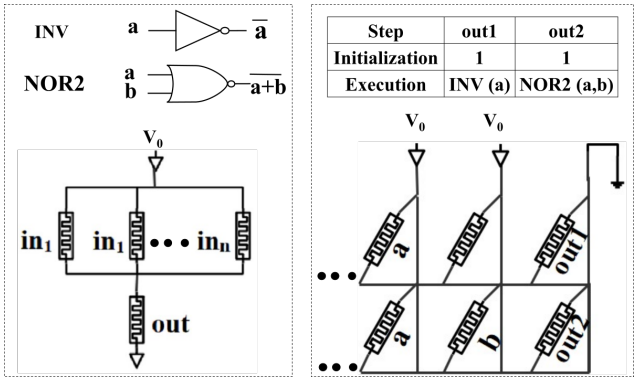
Most digital circuits are designed using automated synthesis workflows comprising numerous intricate design stages [40]. These synthesis tools convert logical operations into a network of interconnected Boolean gates, with an emphasis on minimizing both the count of gates and connections. Although custom synthesis methods for digital in-memory computing have been explored, their applicability has largely been limited to relatively simple circuits [21, 23, 34, 58]. In this paper, we utilize recent advancements in logic synthesis to map arithmetic operations into digital in-memory computing kernels.

In this paper, we introduce LOGIC, a logic synthesis framework aimed at mapping arithmetic operations onto digital in-memory compute kernels. Our framework addresses this mapping challenge through a combination of mathematical optimization problems and software/hardware co-design principles. The use of automated synthesis instead of manually designed templates can potentially lead to remarkable improvements in terms of power, latency, and area. We present a case study showing potential improvements in terms of area and time-steps for arithmetic operations in Table 1. The table shows a comparison between manual design and synthesis based design for in-memory fixed-point arithmetic operations. The results in the table shows that the proposed synthesis based approach improves the area (in terms of intermediate memory storage) by 88% and 79% respectively for 32-bit fixed-point addition and multiplication. It can also be observed that the latency is improved by 16% and 22% respectively for fixed-point addition and multiplication operations.

The key contributions of LOGIC can be summarized as follows:

(1) First, we propose techniques for breaking down element-wise arithmetic operations into in-memory kernels while minimizing the total number of required in-memory operations. Our experimental investigations demonstrate that our approach to in-memory compute kernel synthesis can achieve a 20% reduction in the number of in-memory operations for fixed-point multiplication.
(2) Second, we present two graph-based methods for optimizing the sequence of in-memory operations to minimize the utilization of non-volatile memory. In the preliminary version of this work, a greedy algorithm was used to optimize the sequencing problem [45]. In this work, we develop a node look-ahead and a cone look-ahead algorithm which achieves further performance improvement. Our experimental studies demonstrate that these algorithms can reduce the size of the required intermediate non-volatile memory by 84% for fixed-point multiplication.
(3) Third, we introduce an algorithm for reorganizing data layouts, which transforms sparse matrix-vector-multiplication (MVM) operations into dense blocks. This re-organization enhances hardware utilization and reduces the data transfer across inter-crossbars for in-memory MVM operations.
(4) We evaluate the LOGIC framework using 15 scientific computing applications from the Suite Sparse Matrix Collection [10]. Compared to the state-of-the-art approach presented in [3], LOGIC achieves substantial improvements, enhancing area, latency, and energy efficiency by factors of 4.8×, 2.6×, and 11×, respectively.

The organization of this manuscript is as follows: Section 2 presents the preliminaries. The LOGIC framework is introduced in Section 3. Section 4 and 5 outline the synthesis steps. A data layout re-organization algorithm is proposed in Section 6. The experimental evaluations are presented in Section 7. Summary and future research are discussed in Section 8.



(a) (b)

Fig. 1. (a) MAGIC-style in-memory logic operation and, (b) parallel logic operations in parallel crossbar rows.

2 Preliminaries

In this section, we first provide a concise overview of the fundamental principles associated with digital in-memory computing. Next, we describe the state-of-the-art approach of breaking down arithmetic operations into in-memory kernels using manually crafted templates.

2.1 Digital In-Memory Computing using MAGIC

MAGIC [31] is one of the most widely adopted logic styles for digital in-memory computing. In this section, we explain the utilization of MAGIC for carrying out logic NOR operations. Note that NOR is a universal gate, meaning any Boolean function can be represented using a netlist exclusively composed of NOR gates [11]. Figure 1(a) illustrates a circuit capable of executing MAGIC INV and NOR2 operations. The memristors labeled as $in_{1:n}$ serve as inputs to the NOR gate, and the memristor labeled as *out* generates the output of the NOR operation. A MAGIC operation consists of two key steps. In the initialization step, the input memristors are set to either the high resistance state (HRS), or the low resistance state (LRS). At the same time the *out* memristor is set to LRS. The HRS and LRS corresponds to logic “0", and logic “1", respectively. In the execution step, a controlled voltage V_0 is applied to the input memristors, and the output memristor is grounded. This results in the evaluation of the output of the NOR operation in the *out* memristor. These steps are shown at the top of Figure 1(b). When memristors are organized in crossbars, parallel NOR operations can be executed. The bottom part of Figure 1(b) illustrates the execution of a parallel INV and NOR2 operation.

2.2 Manually Designed Templates for Arithmetic Operations

In this section, we review the standard approach of decomposing arithmetic operation into digital in-memory compute kernels. The state-of-the-art in-memory computing paradigms use a manually crafted template based approach for this decomposition [16, 57]. We explain the decomposition of addition and fixed-point multiplication into in-memory computation kernels in the following sections.

2.2.1 Addition. A 1-bit full adder can be decomposed using NOR-only operations. The NOR-only netlist for a 1-bit full adder operation can be written as follows,

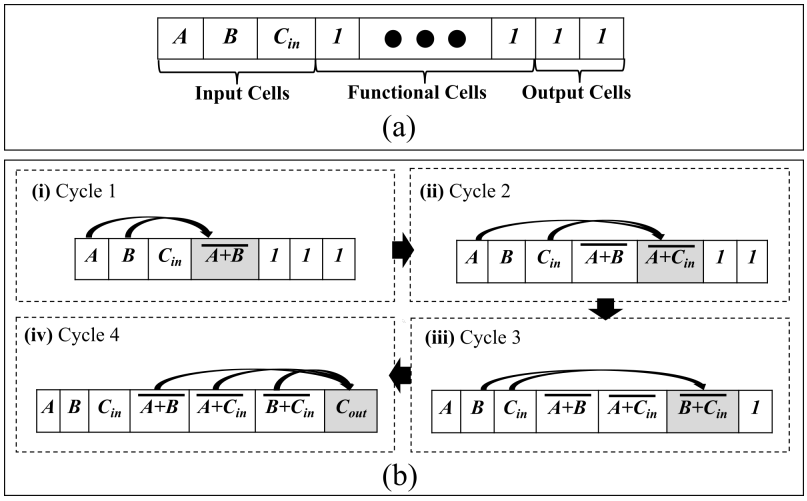


Fig. 2. Decomposition of addition operation into in-memory kernels. (a) Partitioning of a crossbar row into *task-specific* cell groups and, (b) execution flow of the NOR-only netlist of C_{out}.

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

Here, A and B are the input operands and C_{in} and C_{out} are the carry-in and carry-out bits respectively. Figure 2 illustrates the in-memory adder operation where a single row of crossbar is used to perform the computation. In this approach, the crossbar row is partitioned into three groups of memristors cells [57]: the input storing cells, the functional cells, and the output storing cells, as shown in Figure 2(a). The functional cells sequentially perform the intermediate NOR operations to generate the *Sum* and C_{out}. For instance, the execution flow of the sequential NOR operations within the C_{out} netlist is shown in the Figure 2(b). The C_{out} bit is evaluated using three 2-input NOR operations shown in Figure 2(b) (i)-(iii) followed by one 3-input NOR operation shown in Figure 2(b) (iv).

2.2.2 Fixed-Point Multiplication. An n-bit fixed-point multiplication can be decomposed into sequential adder operations of the partial products [16]. For instance, Figure 3 illustrates the multiplication of two multi-bit operands, A and B. Figure 3(a) shows the addition of the top two rows of partial products, PP_1 and PP_2. The multi-bit addition can be further decomposed into the 1-bit adder operation discussed in the previous section. The addition of PP_1 and PP_2 generates an intermediate solution, IS. Next, IS is added to the third partial product row, PP_3, as shown in Figure 3(b). This process is continued until all rows of partial products are sequentially processed.

3 The LOGIC framework

In this section, we present the LOGIC framework. An overview of the framework is shown in Figure 4.

The aim of the LOGIC framework is to accelerate applications that are dominated by sparse matrix-vector multiplications. The framework is based on digital in-memory computing. It comprises three integral steps: (a) synthesis of in-memory compute kernels, (b) execution sequence optimization, and (c) data layout re-organization. In the synthesis step, the target arithmetic operation is synthesized into a netlist of in-memory operations. During the synthesis, the aim

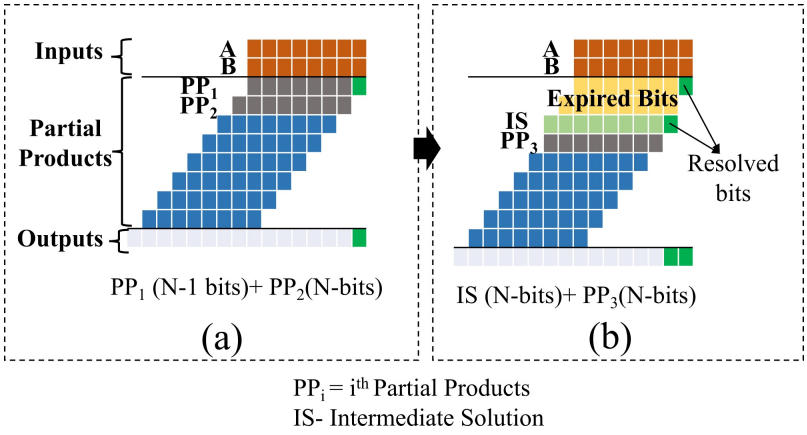


Fig. 3. Decomposition of fixed-point multiplication into addition of partial products.

is to minimize the size of the netlist. This step is detailed in Section 4. In the execution sequence optimization step, an execution sequence for the in-memory operations within the netlist is generated. The goal of this step is to minimize the number of intermediate memory cells while performing the in-memory operations. This step is described in Section 5. The synthesis step and the sequence optimization step combined compose a library of execution sequences for different in-memory arithmetic operations. This library is utilized to accelerate sparse matrix-vector multiplication operations. In the data layout re-organization step, the data layout within large sparse matrices is reorganized into dense blocks of matrix elements. The goal of this step is to improve the hardware utilization for in-memory sparse MVM operations. This step is explained in the Section 6.

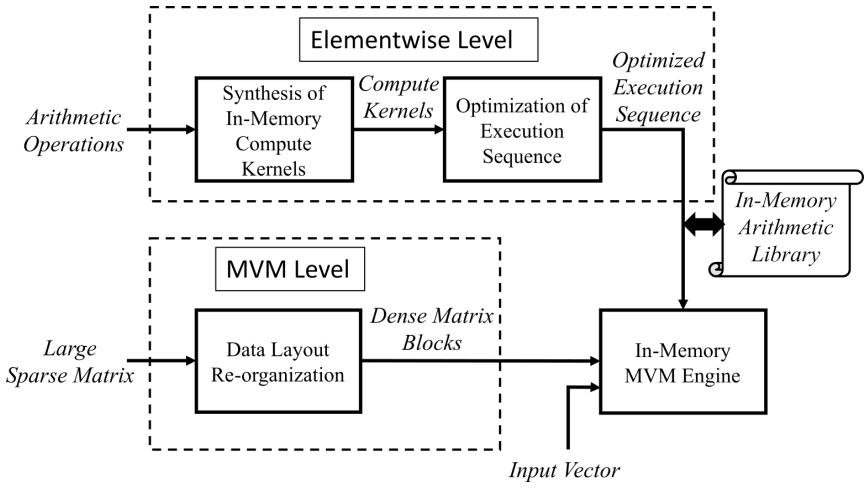


Fig. 4. Overview of the LOGIC framework

4 Synthesis of In-Memory Compute Kernels

In this section, we describe how rudimentary arithmetic operations (e.g., addition, multiplication) are synthesized into in-memory NOR-INV netlists. The overview of synthesis flow is illustrated in Figure 5.

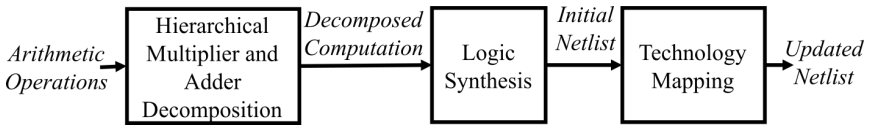


Fig. 5. Synthesis of in-memory compute kernels.

The synthesis flow comprises three key steps: (1) hierarchical multiplier and adder decomposition, (2) conventional logic synthesis, and (3) technology mapping. Step 1 involves a hierarchical decomposition of element-wise multiplication. This step breaks down the multiplication operation into partial product computations and adder operations. In step 2, the decomposed computations are synthesized into an initial NOR-INV netlist. This is achieved using a conventional logic synthesis tool such as ABC [40] and a custom NOR-INV cell library. We exclude the specifics of this step in this manuscript as we directly use the ABC tool to perform this step. In step 3, the ABC-generated netlist is optimized by converting low fan-in gates into high fan-in gates.

4.1 Hierarchical Multiplier and Adder Decomposition

In this section, we adopt a hierarchical approach to break down arithmetic operations into smaller components. The motivation for this decomposition is that the netlists of arithmetic operations tend to scale exponentially with higher bit-widths of operands. A hierarchical decomposition limits this exponential growth in netlist size.

In the hierarchical decomposition approach, an n-bit multiplication operation is decomposed into sequential additions of partial products. The decomposition converts the n-bit multiplication into a series of n-bit partial product additions as illustrated in Figure 3. Next, the n-bit addition operation is further decomposed into a chain of m-bit additions. This decomposition concept is illustrated with an example in Figure 6. The value of m is experimentally determined in Section 7.2. The goal is to select a value of m that minimizes the NOR-INV netlist size of the overall computation.

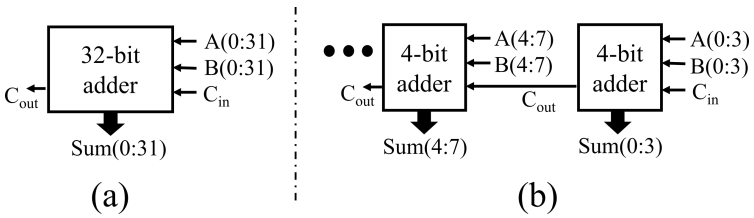


Fig. 6. Hierarchical decomposition of adders. (a) a 32-bit full adder and, (b) an equivalent adder chain of 4-bit full adders.

4.2 Technology Mapping

In this section, we elaborate on the technology mapping step. The goal of this step is to reduce latency by minimizing the number of in-memory operations in the initial netlist. This is achieved by merging the low fan-in gates to generate gates with high fan-in. This process is illustrated using an example in Figure 7.

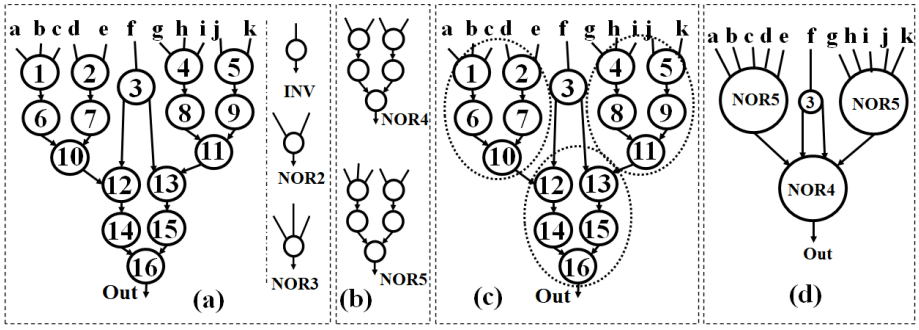


Fig. 7. Technology mapping: (a) initial netlist with gate encoding, (b) library of high fan-in gates, (c) cover of subject graph, and (d) updated netlist.

In the technology mapping step, the NOR-INV netlist is first converted into a directed acyclic graph (DAG), $G = (V, E)$. The DAG representation of an example netlist is shown on the left of Figure 7(a). The corresponding gate encoding of the DAG nodes is provided on the right of Figure 7(a). The gates in the initial netlist have low fan-ins of $1-3$. We aim to merge these low fan-in gates into higher fan-in gates. A representative cell library of higher fan-in gates is shown in Figure 7(b). The cell library is utilized to cover the DAG in Figure 7(c). We use the DAGON [30] algorithm to solve this DAG covering problem. The updated netlist with higher fan-in gates is shown in Figure 7(d). In this example, the technology mapping step reduces the total number of nodes from 16 in the initial netlist to only 4 nodes in the updated netlist. This reduction in nodes leads to a decrease in total in-memory operation cycles, resulting in a reduction in latency. The latency improvement achieved from the technology mapping step is experimentally evaluated in Section 7.2.

5 Execution Sequence Optimization

In this section, we discuss the execution sequence optimization step. We first formulate the optimization problem. Next, we present an optimal solution based on enumeration. Lastly, we provide two practical approaches to solve the sequence ordering problem.

5.1 Problem Formulation

The input is a netlist represented using a DAG, $G = (V, E)$, where each node in V corresponds to a in-memory operations and each edge in E corresponds to a predecessor constraint. 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:

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

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 $|W_i|$.

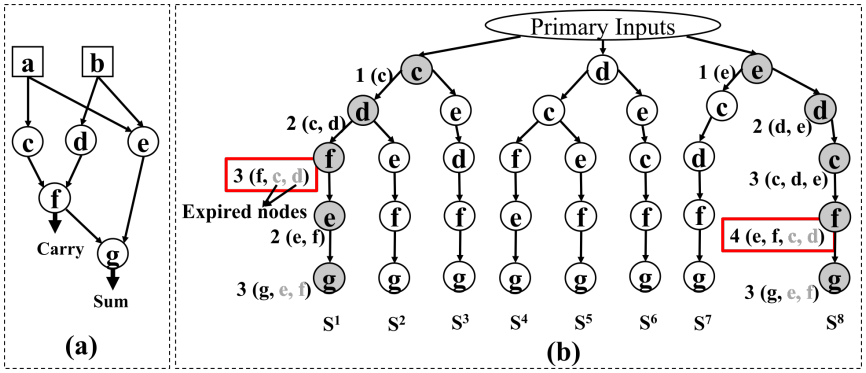


Fig. 8. (a) DAG representation of a half adder and, (b) enumeration of all the feasible execution sequences.

5.2 Enumeration of Execution Sequences

In this step, we outline a solution to the execution sequencing problem based on enumeration. Figure 8(a) shows the DAG representation of a half adder. Here, the gates of the netlist are represented by (c)–(g) nodes and, the primary inputs are represented by $\boxed{a}$ and $\boxed{b}$. Figure 8(b) shows the enumeration of all the topological orders of the DAG nodes. $S^1 - S^8$ represents the eight possible execution sequences of the nodes. We highlight the $cost(S_i)$ from Eq (1) for the left-most and the right-most sequences in the figure. In the figure, the expired nodes represent the nodes that are no longer required for any future node processing (i.e., they are not inputs to any future node). The memories occupied by the expired nodes can be released and reused for future processing. The figure shows, 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 aims to minimize $cost(S_i)$ while selecting a sequence. Based on the example, it is clear that the sequences with a lower cost process nodes in such an order that the intermediate memory can be released early.

It is easy to understand that the enumeration of all possible sequences can lead to the selection of an optimum execution sequence with minimum cost. However, this enumeration is computationally impractical for large DAGs where many nodes are involved. The computational complexity of execution sequence enumeration is experimentally evaluated in 7.3.

For adders with higher bit-width resolutions, we develop two practical algorithms to generate a relatively small number of sequences. The first algorithm is based on a node look-ahead covering of the system graph G and, the second algorithm is based on a cone look-ahead covering of G.

5.3 Node Look-Ahead Algorithm

In this section, we propose a node look-ahead algorithm for generating in-memory execution sequences. The algorithm is based on selecting to process a set of nodes that minimizes the intermediate memory cost the most. This cost minimization is achieved by immediately releasing the memory that is no longer required for future computation. We define a term *effective inclusion cost (EIC)* to drive the selection of nodes. Lets consider a set of nodes s that can be processed in order. The EIC $h(s)$ of the set is calculated as, $h(s)$ = *(memory cost of processing set s)−(memory released after processing s)*. Our proposed look-ahead algorithm iteritively selects to process the set s of $\leq k$ nodes that have the smallest EIC. The algorithm is presented in Algorithm 1.

Algorithm 1: Node Look-Ahead Algorithm for Execution Sequence Generation

Inputs: DAG, $G = (V, E)$, node set size threshold, k
Output: Execution sequence, $\mathcal{S}$;
main {
$\mathcal{S} \leftarrow \phi$; // sequence initialization
$N \leftarrow$ total nodes in G; // pruning primary inputs
while $size(\mathcal{S}) \neq N$ **do**
  // nodes that meet predecessor constraint
  $\mathcal{C} \leftarrow find\_candidate\_nodes(G, \mathcal{S})$;
  // enumerating all candidate node sets of set size $\leq k$
  $\mathcal{T} = \{s_1, s_2 \cdots s_{|\mathcal{T}|}\} \leftarrow construct\_sets(G, \mathcal{C}, k)$;
  // find minimum EIC of all sets
  $c_{min} \leftarrow compute\_min\_EIC(\mathcal{T})$;
  // pruning all set without minimum EIC
  $\mathcal{T}' = \{s'_1, s'_2 \cdots s'_{|\mathcal{T}'|}\} \leftarrow prune\_sets\_cost(\mathcal{T}, c_{min})$
  $s' \leftarrow rand(\mathcal{T}')$; // random selection
  $\mathcal{S} \leftarrow \mathcal{S} \cup s'$; // ordered execution set
end
return $\mathcal{S}$;
}

The input to the algorithm is the DAG representation G of the target netlist and a set size threshold k which defines the maximum number of nodes in a set. The output of the algorithm is an execution sequence $\mathcal{S}$. First, a pool of all candidate nodes $\mathcal{C}$ is selected. We define a *candidate node* as a node that meets predecessor constraint, i.e., all inputs to the node have already been processed. Next, the *construct_sets* function generates all possible set permutations $\{s_1, s_2 \cdots s_{|\mathcal{T}|}\}$ of candidate nodes where the set size is $\leq k$. Note that once a single node has been added to a set, we check if there are other nodes that can temporarily be added to the candidate node set. To generate an execution sequence that has the best chance of incurring minimum cost, the algorithm makes informed decision before including a set of nodes into the sequence. The algorithm first evaluates and records the EIC for each of the candidate sets of nodes. The minimum EIC is also stored in a parameter c_{min}. Next, all sets with an EIC higher than c_{min} are pruned out. If there are multiple candidate sets with the minimum cost c_{min}, the algorithm selects one of these sets at random. After each expansion of the sequence, the DAG and the candidate node pool are dynamically updated. The algorithm concludes when all the nodes within the DAG are covered.

Since the algorithm contains a non-deterministic component (the random selection to break EIC ties), the algorithm iteratively explores M different sequences and selects the sequence $\mathcal{S}$ that yields the minimum cost in Eq (1). The synthesis tool memoizes the $cost(S_i)$ of the generated sequences of length l, which is called $cost[l]$. Whenever the cost of a future sequence of length l is higher than $cost[l]$, that sequence is terminated.

We illustrate the algorithm with an example in Figure 9. The figure shows the generation of an execution sequence for the DAG of a half adder using a set size threshold $k = 2$. On the right of Figure 9(i), the target DAG of the half adder is shown. Note that the nodes (c), (d), and (e) already meet the predecessor constraint as inputs to these nodes are exclusively the primary inputs $\boxed{a}$ and $\boxed{b}$. Figure 9(ii) shows the first expansion of the sequence. The figure shows that there are nine feasible node sets that both meet the predecessor constraint and satisfy the set size threshold of $k = 2$. The EIC ($h(s)$) for each of the nine sets is also shown in the figure. Here, the three node sets (each with one node) yield the minimum EIC of $h(s)=1$. Therefore, a node set with minimum EIC (e.g., set (c)) is chosen at random. The resultant

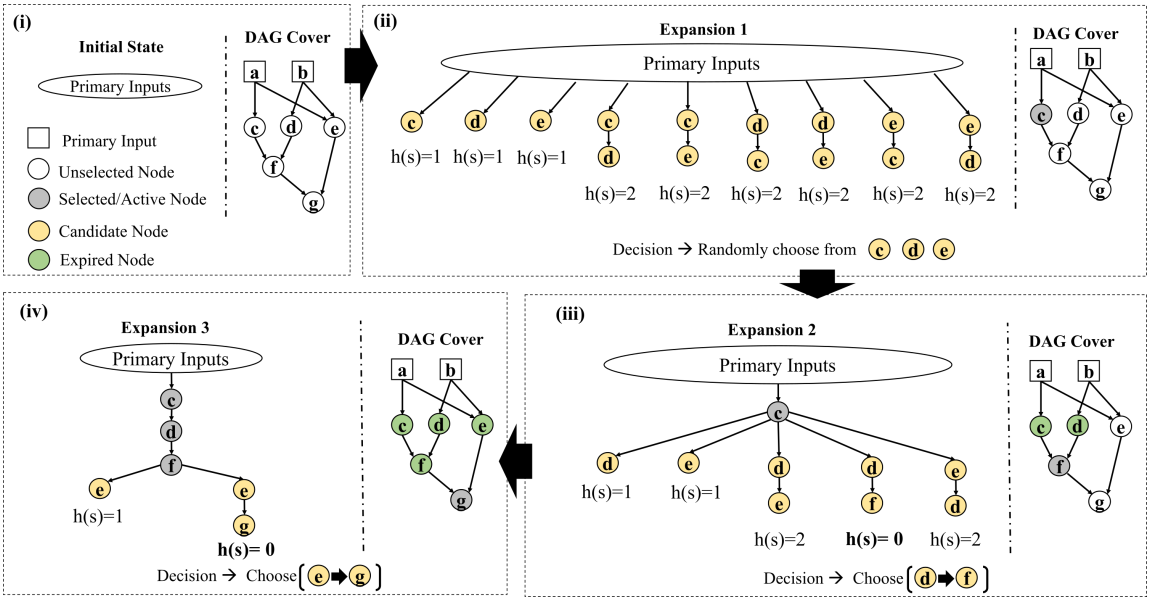


Fig. 9. Node look-ahead algorithm for execution sequence ordering. The example shows a DAG-covering workflow for a node set size threshold of $k = 2$.

DAG cover is shown in the right of Figure 9(ii). The second expansion of the sequence is shown in Figure 9(iii). In this expansion, the node set (d)→(f) yields the minimum EIC of *h(s)=0*. Therefore, the algorithm includes the node set (d)→(f) in the sequence. In the third expansion in Figure 9(iv), the node set (e)→(g) yields the minimum EIC of *h(s)=0*. The algorithm, therefore, makes the informed decision of including the node set (e)→(g) to the sequence. The algorithm concludes when all the nodes of the DAG are covered.

5.4 Cone Look-Ahead Algorithm

In this section, we present a cone look-ahead algorithm for generating in-memory execution sequences. Similar to the node look-ahead algorithm, the goal is to process nodes in an order that minimizes intermediate memory cost by immediately releasing expired memory contents. However, contrary to the node-look ahead algorithm, the cone look-ahead algorithm aims to explore sets of nodes where the nodes in a set are connected by edges in G. In particular, we define a cone (or a set of nodes) to be the nodes in the recursive fan-in of a node $n \in V$ in G. The advantage of restricting the nodes to be connected is that the number of possible sets of size k is much smaller, which allows sets of significantly larger size to be considered. In the experimental evaluation, we observe that the node look-ahead algorithm is limited to sets of size 2 to 4 due to the runtime complexity. (The cone look-ahead algorithm can handle sets of arbitrary sizes.) At the same time, the quality of the solutions are not degraded, this stems from that it is connected nodes that provide ample opportunities for intermediate memory to be released.

The cone look-ahead algorithm is presented in Algorithm 2. The algorithm receives a DAG G and a cone size threshold parameter k as inputs. k defines the maximum number of nodes that a cone can cover. The output is an execution sequence S. The algorithm iteratively selects the cone of size k with the smallest effective inclusion cost (EIC), as defined in Section 5.3.

Algorithm 2: Cone Look-Ahead Algorithm for Execution Sequence Generation

Inputs: DAG, $G = (V, E)$, cone size threshold, k
Output: Execution sequence, $\mathcal{S}$;
main {
$\mathcal{S} \leftarrow \phi$; // sequence initialization
$N \leftarrow$ total nodes in G; // pruning primary inputs
while $size(\mathcal{S}) \neq N$ **do**
  // extracting all possible cones in G
  $\mathcal{T} = \{c_1, c_2 \cdots c_{|\mathcal{T}|}\} \leftarrow cone\_extract(G, \mathcal{S})$;
  // pruning of cones with more than k nodes
  $\mathcal{T}' = \{c'_1, c'_2 \cdots c'_{|\mathcal{T}'|}\} \leftarrow prune\_cone\_size(\mathcal{T}, k)$;
  // calculating min EIC for all cones in $\mathcal{T}'$
  $c_{min} \leftarrow calculate\_EIC(\mathcal{T}')$;
  // pruning of cones with a cost higher than c_{min}
  $\mathcal{T}'' = \{c''_1, c''_2 \cdots c''_{|\mathcal{T}''|}\} \leftarrow prune\_cone\_cost(\mathcal{T}', c_{min})$
  $c'' \leftarrow rand(\mathcal{T}'')$; // random selection
  $\mathcal{S} \leftarrow \mathcal{S} \cup c''$; // ordered execution set
end
return $\mathcal{S}$;
}

First, the algorithm utilizes the *cone_extract* function to enumerate all possible cones $\{c_1, c_2 \cdots c_{|\mathcal{T}|}\}$. Next, all cones with size $> k$ are removed using the *prune_cone_size* function. $\{c'_1, c'_2 \cdots c'_{|\mathcal{T}'|}\}$ contains the candidate cones to be included in the execution sequence. Next, the EIC $h(c)$ is computed for all the candidate cones and the minimum cone cost c_{min} is recorded. Next, all cones that do not have the minimum cost are pruned using the *prune_cone_cost* function. Similar to the Algorithm 1, the algorithm next makes a random selection from the cones with minimum EIC. After a cone is selected, the DAG is dynamically updated to consist only the non-covered nodes $N \setminus \mathcal{S}$. The sequence generation concludes when $N \setminus \mathcal{S} = \phi$.

The algorithm explores M different execution sequences and selects the sequence $\mathcal{S}$ that yields the minimum cost in Eq (1). A similar memorization and pruning technique as in Algorithm 1 is implemented to eliminate the impact of the non-deterministic behavior of the random selection of equal cost cones.

We illustrate the workflow of Algorithm 2 with an example in Figure 10. Figure 10(i) shows the target DAG of a half adder. In Figure 10(ii) we explore all the cones of the nodes of the DAG. The DAG consists of five cones originated from the nodes (c) − (g). The cones are extracted in Figure 10(iii). The values of k for the extracted cones range between $1-5$. In this example, we impose a threshold value of $k = 3$. Based on the threshold, the cones are next pruned for $k > 3$ and the EIC of the candidate cones are evaluated in Figure 10(iv). The figure shows that the cone (c) → (f) ← (d) yields a minimum EIC of -1. The algorithm therefore selects the cone for processing as shown in the Figure 10(v). The DAG is next dynamically updated in Figure 10(vi)−(vii). The cones from the updated graph are explored and extracted in the Figure 10(viii)−(ix). After pruning for $k > 3$, EIC of the candidate cones are calculated in the Figure 10(x). The figure shows that the cone (f) → (g) ← (e) yields a minimum EIC of -3. The algorithm processes the cone in the Figure 10(xi) and concludes the node covering.

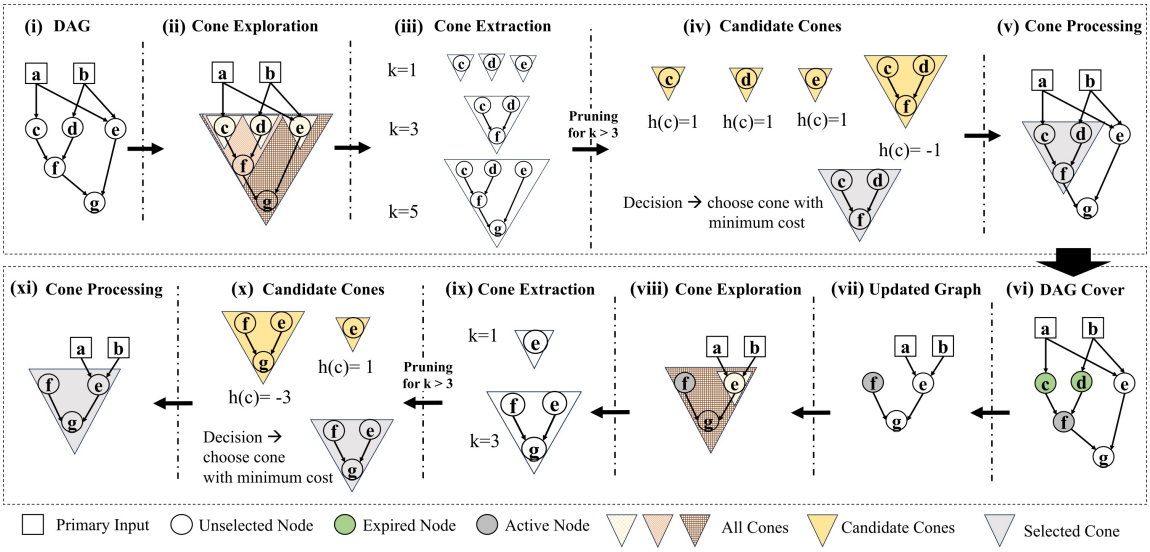


Fig. 10. Cone look-ahead algorithm for execution sequence ordering. The example shows a cone-covering workflow for a cone-size threshold of $k = 3$.

6 Data Layout Re-Organization

In this section, we present a novel algorithm to enhance the hardware utilization of in-memory sparse matrix-vector-multiplication (MVM) operations. The primary objective of this algorithm is to improve the hardware utilization and reduce the expensive inter-crossbar data transfers when performing in-memory MVM operations using sparse matrices. The key idea of the algorithm to reorganize the data layout of unstructured sparse matrices before decomposing them into the in-memory hardware.

The current state-of-the-art approach for decomposing MVM into crossbars involves organizing the matrix operands in a row-wise manner within the crossbar and then performing row-parallel arithmetic operations [3, 24]. This row-wise arrangement ensures that all the arithmetic operands related to a specific output vector element are situated within the

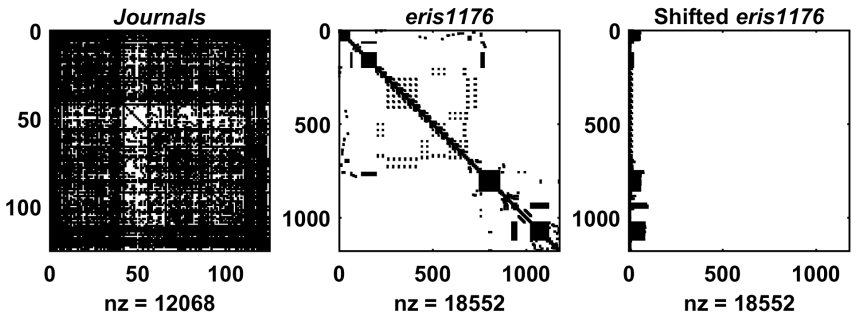


Fig. 11. Matrices from the SuitSparse matrix collection [10]. (a) Dense matrix *Journals*, (b) sparse matrix *eris*1176 and, (c) row-wise shifted *eris*1176 matrix. A black dot in matrix represents a non-zero data, and the white parts represent zero values.

same row of the crossbar, eliminating the need for inter-row data copying. This method works efficiently with dense matrices such as the *Journals* matrix in Figure 11(a).

Unfortunately, the majority of matrices found in physical systems tend to be sparse. Figure 11(b) illustrates an example of a sparse matrix *eris1176*. We observe that a naive row-wise alignment of data for sparse matrices results in hardware under-utilization due to that a significant hardware resource is wasted on arithmetic operations on operands with a zero value. To maximize hardware efficiency, a denser data layout is required. For instance, Figure 11(c) demonstrates a row-wise shifting of the operands within the *eris1176* matrix, ensuring a more compact assignment of matrix operands within crossbars. This denser arrangement translates to reduced hardware requirements and fewer inter-crossbar data transfers.

However, an interesting new challenge related to the routing of corresponding input vector operands arises from this data re-organization technique. In a regular dense matrix, the routing resources can be shared among all the crossbar rows. However, with a potentially extreme shifting shown in the Figure 11(c), each row of the shifted matrix is multiplied by a unique vector $v \subset \mathcal{V}$ where $\mathcal{V}$ is the original input vector. This results in that the routing of input vector to each crossbar row is required to be pipelined which voids the massive parallelism opportunity of row-parallel computing. Additionally, the enumeration complexity of all subsets of the input vectors is $O(R)$ where R is the number of matrix rows.

To address this challenge of input vector routability, we propose a novel data layout re-organization algorithm. We illustrate this concept with an example matrix segment shown in Figure 12(i). The primary objective of this algorithm is to reorganize the matrix operands in a manner that allows all rows within a crossbar to share the same set of input vector routing resources. This is achieved through a series of operations:

- **Padding:** this operation performs padding on matrix columns containing at least one non-zero element (Figure 12(ii)).
- **Shifting:** this operation horizontally shifts the matrix columns with non-zero elements (Figure 12(iii)).
- **Cleaning:** this operation removes the padded elements from the shifted matrix (Figure 12(iv)).

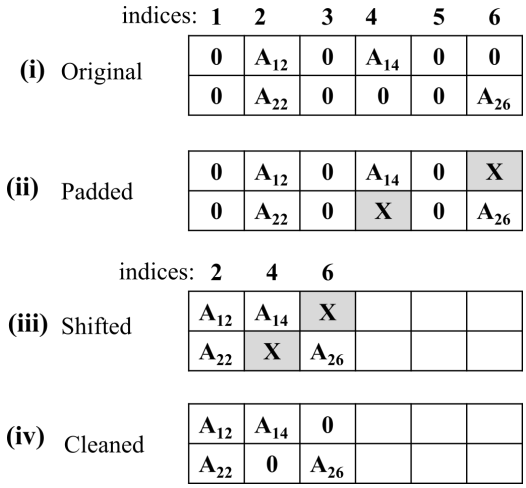


Fig. 12. Sparse matrix data layout re-organization

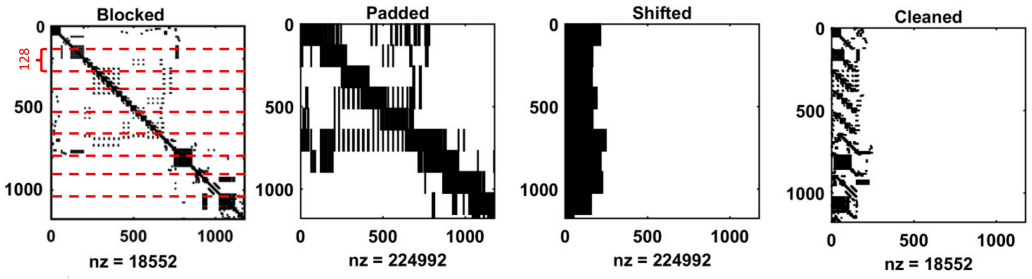


Fig. 13. Data layout re-organization of *eris*1176 matrix.

Note that the set of shifted column indices for both matrix rows is {2, 4, 6}, meaning that both shifted rows are multiplied by the 2^{nd}, 4^{th} and 6^{th} elements of the input vector. For a crossbar dimension of $P \times Q$, this algorithm guarantees that all P rows of the crossbar share the same subset of the input vector. This also reduces the complexity of enumerating the total subsets of input vectors from $O(R)$ to $O(R/r)$, where r is the row dimension of the partitioned matrix blocks.

Figure 13 presents the data layout re-organization workflow for the sparse matrix *eris1176*. Figure 13 (a) illustrates a blocked representation of the original matrix. This blocking step is required when the matrix row dimension exceeds the row dimension of crossbars. This decomposition of the matrix ensures that each of the smaller blocks can be parallelly processed in different sets of crossbars. In our architectural setup, we utilize crossbars with dimensions of 128×128. In Figure 13 (a), each block consists of 128 rows, matching the number of rows in the crossbar. Each block is multiplied by a unique subset of input vector elements, which are parallelly routed to all rows within the corresponding crossbars. The *padding* operation is shown in Figure 13 (b). This operation increases the number of non-zero elements from the original 18,552 to 224,992. Next, the *shifting* and *cleaning* operations are displayed in Figure 13 (c) and (d), respectively. After cleaning, the number of non-zero elements equals the original matrix's count.

7 Experimental Evaluation

In this section, we experimentally evaluate the performance of the LOGIC framework. For the experiments, we use an Intel Core i9 3.60 GHz octa-core processor with 64 GB RAM. We develop the synthesis tool using a blend of C++ and MATLAB codes. We also use the ABC tool [40] to convert our target computation into an initial Boolean netlist.

We first present the architecture in Section 7.1. Next, we evaluate the effectiveness of the synthesis steps of the LOGIC framework in Section 7.2. Next, we evaluate the performance of the proposed sequence optimization algorithms in Section 7.3. Subsequently, we perform a comparative performance evaluation of the proposed framework with the state-of-the-art in Section 7.4. We first evaluate the framework for arithmetic operations in Section 7.4.1. Next, we present a comparative evaluation of different frameworks for scientific computing applications in Section 7.4.2. The overview of the selected benchmarks from the Suite Sparse Matrix collection [10] are listed in Table 2.

7.1 Architecture

In this section, we present the architecture of the LOGIC framework. Figure 14(a) provides an overview of the architecture. The architecture comprises multiple accelerator tiles, each embedded with an array of processing elements (PE). Input and output registers (IR/OR) facilitate the transfer of input and output operands, respectively. An eDRAM buffer serves to

Table 2. Scientific computing applications from the SuiteSparse Matrix collection [10].

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

store intermediate results and routing indices of MVM operands. A high-speed bus facilitates communication among PEs, storage units, and the I/O interface. The components of a PE are shown in Figure 14(b). A PE is composed of several memristor crossbars arranged in a row-parallel configuration. Each crossbar has a dimension of 128×128. Routing blocks (RB) and drivers are employed to configure the crossbars. Row-parallel-copying (RPC) circuits and sense amplifiers (SA) enable inter-crossbar data transfer [24]. Figure 14(c) illustrates the cross-section of a routing block, where each intersection of nanowire within the RB incorporates a memristor that functions as a routing switch.

This architecture is particularly attractive as it offers a high degree of parallelism for MVM-dominated applications. For example, all decomposed blocks of an MVM operation can be executed in parallel, with each block assigned to a dedicated set of row-parallel crossbars.

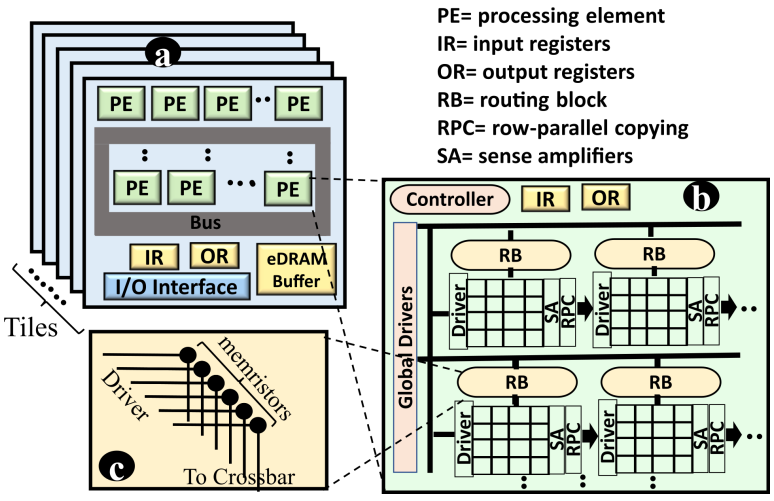


Fig. 14. (a) Overview of the architecture, (b) components of a processing element and, (c) architecture of the routing block

Table 3. Area-Power Cost of Architectural Components

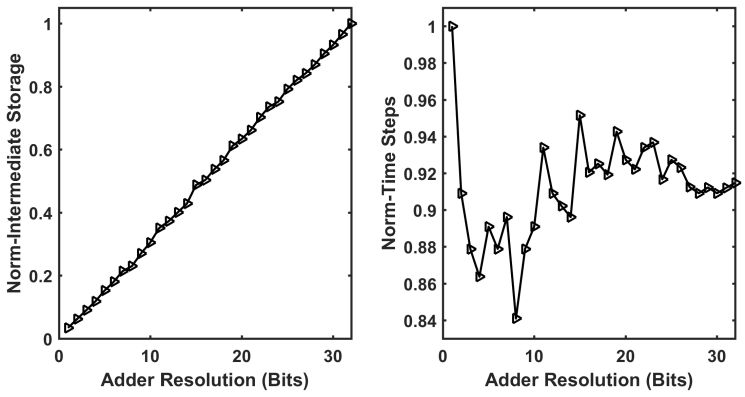
Component	Parameter	Specs	Area	Power
Crossbar	size	128×128	25 μm^2	0.30 mW
Sense Amp.	# unit	128	7.14 μm^2	0.29 mW
Controller	# unit	1	400 μm^2	0.65 mW
RB	# unit	1	12 μm^2	0.01 mW
eDRAM Buffer	size	128 KB	0.17 mm^2	41.40 mW
IR	size	16 B	16.80 μm^2	0.01 mW
OR	size	16 B	46.88 μm^2	0.02 mW
Bus	bandwidth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

Table 3 summarizes the area-power costs of different architectural components. The per-unit costs are appropriately adapted from previous works [24, 25, 32, 52]. The cross-architecture data transfer costs are adapted from [56].

7.2 Evaluation of Synthesis Steps

In this section, we experiment with the steps of the synthesis flow within the LOGIC framework. The experiments are designed to determine optimum parameters for the LOGIC framework and to evaluate the effectiveness of different synthesis steps.

Hierarchical Adder Decomposition: The first step of the synthesis of in-memory compute kernel is the hierarchical decomposition of element-wise multiplications into partial product addition operations. The optimal adder bit-width m for this decomposition is experimentally determined using the ABC tool [40]. ABC generates the netlists of adders with different bit-widths. A custom cell library is used such that the generated netlists consist only NOR gates of variable fan-ins. In Figure 15, we present a trend of memory cost and time-steps for adders with variable bit-widths. For comparison, the results are normalized with respect to per-bit addition. The result shows that the intermediate storage cost



(a) (b)

Fig. 15. Intermediate storage and time steps with respect to adder bit-width.

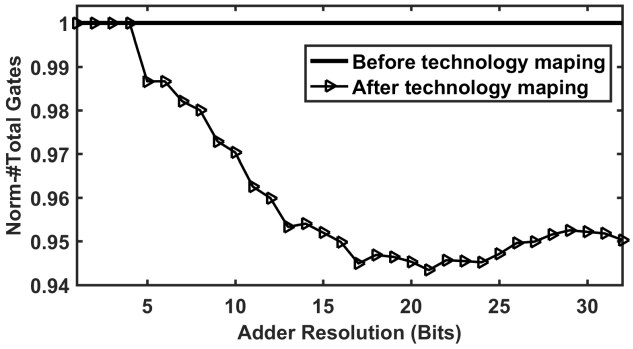


Fig. 16. Number of total gates before and after the technology mapping.

linearly grows for higher bit-width adders. However, the time-step trend does not follow a linear trajectory. The trend shows a minima point for the adder with a bit-width of 8. It is noteworthy that the ABC tool is based on heuristics. This limitation of ABC contributes to the noise in the time-step trend. Based on this experimental observation, we choose an adder decomposition of 8 bits to reduce the energy and latency costs.

Technology Mapping: In the technology mapping step, low-fan in gates are merged to create higher fan-in gates. Therefore, the total number of gates are reduced. Figure 16 shows a comparison of total gates before and after technology mapping for adders with different bit resolution. The result shows, on average the technology mapping reduces the number of total gates by 4%. This reduction in gates translates into reduction of in-memory operations and in turns a reduction in overall latency.

7.3 Evaluation of the Sequence Optimization Algorithms

In this section we compare the performance of the proposed algorithms. We first perform sensitivity analysis for each of the algorithms. Next, we compare the effectiveness and the complexity of the proposed algorithms

7.3.1 Sensitivity Analysis of the Enumeration-based Algorithm. The enumeration-based algorithm can ensure an optimal solution for the execution sequence optimization problem. Unfortunately, an exhaustive enumeration of all possible execution sequences will result in an exponential increase in the runtime for adders with higher bit-widths. Table 4 shows the scalability issue of the enumeration-based algorithm for adders with different bit-widths. The table shows that the number of execution sequences increases exponentially with the number of nodes in the DAG. For instance, a 2-bit adder with a mere 16 nodes generates more than 3 million feasible execution sequences. The enumeration was unsuccessful

Table 4. Scalability Issue of Enumeration-based Algorithm.

Adder Description	**Nodes (# NOR/INV) in DAG**	**# Total Topological Sequences**	**Runtime (minutes)**
half adder	5	8	2.1×10^{-5}
full adder	11	856	4.5×10^{-3}
2-bit adder	16	**3170496**	40.8
2-bit full adder	19	–	–

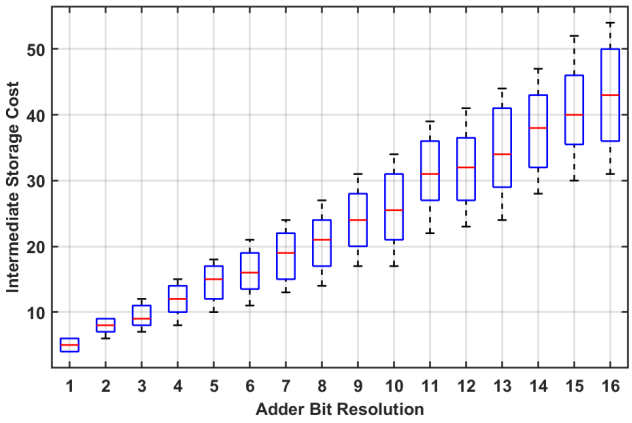


Fig. 17. Distribution of intermediate storage cost for 100 different iterations of the node look-ahead algorithm for adders with different bit resolutions. The distribution results from the random selection of node sets while breaking ties.

for adders with a higher number of bits such as a 2-bit full adder (2-bit adder + carry). These results emphasize that a more practical algorithm is required to generate the execution sequences for adders with higher bit-widths.

7.3.2 Sensitivity Analysis of the Node Look-Ahead Algorithm. We first perform a sensitivity analysis to show the effect of random node set selection (to break ties between candidate node sets) in the node look-ahead algorithm. The Figure 17 shows distribution of intermediate storage cost for 100 different iterations for adders with different bit resolutions. We consider a set size threshold of $k = 1$ for this analysis. As shown in the figure, the intermediate cost may vary in different iterations due to the random selection step in the algorithm. The algorithm runs for M (=100 here) iterations and selects the sequence that requires the minimum intermediate storage cost.

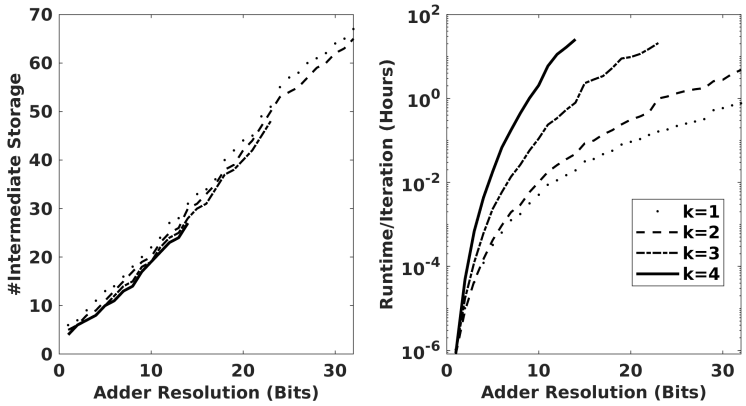


Fig. 18. Sensitivity analysis of the node look-ahead algorithm for variable values of k. (a) Number of intermediate storage for different adder bit-widths. (b) Runtime/Iteration for different adder bit-widths.

Next we illustrate the area-runtime performance of the node look-ahead algorithm for different values of k in Figure 18. Figure 18(a) shows the trend of the intermediate storage requirement for adders with different bit-widths. It can be observed from the figure that with higher values of k, the intermediate storage requirement gradually reduces. However, for higher values of k, the runtime of the algorithm becomes prohibitively large for adders with higher bit-widths. For instance, for $k = 4$ and adder bit-width of $14-$bits, each iteration of the node look-ahead algorithm incurs an average runtime of 25 hours. The average runtime/iteration for different values of k is shown in Figure 18(b). The exponential growth in runtime is caused by the exhaustive enumeration of all the sets of k nodes within the node look-ahead algorithm.

7.3.3 Sensitivity Analysis of the Cone Look-Ahead Algorithm. We first perform a sensitivity analysis to show the effect of random cone selection (to break ties) in the cone look-ahead algorithm. The Figure 19 shows distribution of intermediate storage cost for 100 different iterations. We consider a cone size threshold of $k = 3$ for this analysis. The figure shows, compared to the node look-ahead algorithm, the cone look-ahead algorithm is less effected by the random selection step in the algorithm. The algorithm runs for M (=100 here) iterations and selects the sequence that requires the minimum intermediate storage cost.

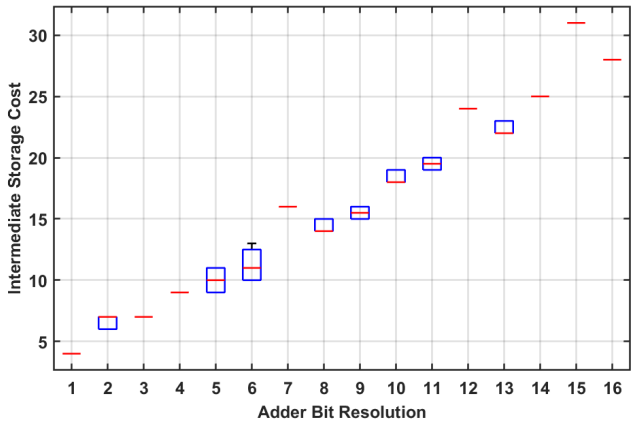
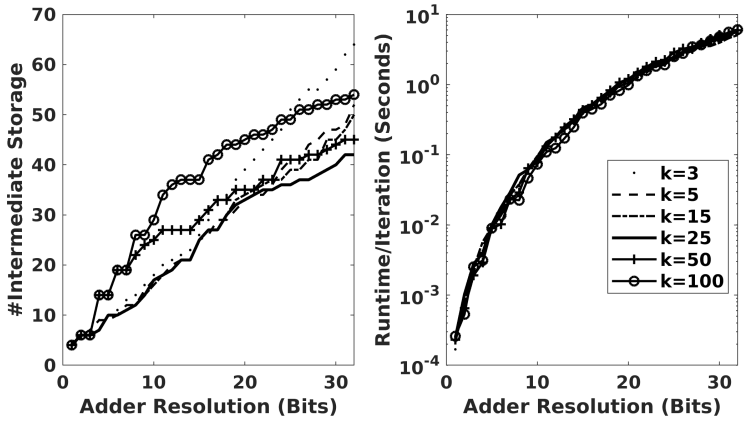


Fig. 19. Distribution of intermediate storage cost for 100 different iterations of the cone look-ahead algorithm for adders with different bit resolutions. The distribution results from the random selection of cones while breaking ties.

Figure 20 shows the area-runtime performance of the cone look-ahead algorithm for different values of k. The intermediate storage requirement for adders with different bit-widths is shown in Figure 20(a). The figure shows that the intermediate storage requirement significantly reduces for $k > 3$. However, the storage improvement tends to saturate for higher values of k (e.g., $k = 15, 25$). Interestingly, for even higher values of k (e.g., $k = 50, 100$), the results tend to deteriorate. This is due to that for higher values of k, the algorithm selects some large cones during sequence generations. While these cones may generate local minima (EICs), the overall results become worse.

The average runtime/iteration for different values of k is shown in Figure 20(b). It can be observed that the algorithm is very efficient even for adders with higher bit-widths. Each sequence exploration concludes within seconds for different values k.

7.3.4 Comparison of Different Algorithms. In this section, we compare the effectiveness of different algorithms while generating execution sequences. Table 5 shows the intermediate memory cost for different algorithms while performing



(a) (b)

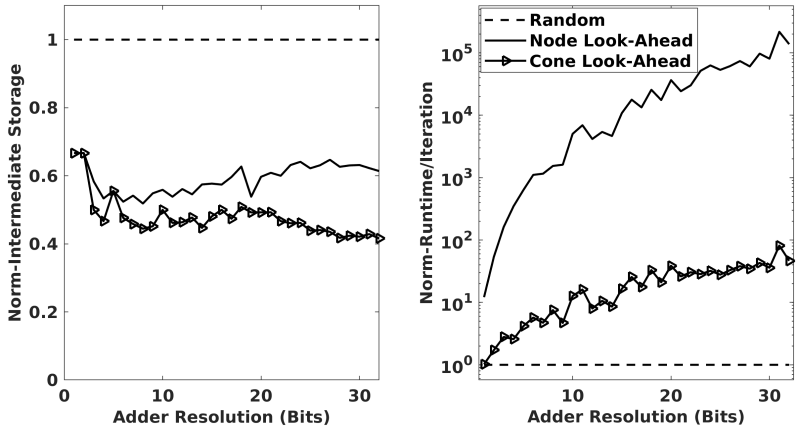
Fig. 20. Sensitivity analysis of cone look-ahead algorithm for variable values of k. (a) Number of intermediate storage for different adder bit-widths. (b) Runtime/Iteration for different adder bit-widths.

adder operations of different bit-widths. For the enumeration algorithm, we enumerate all feasible execution sequences for the adders as shown in Table 4. On the other hand, we run the node look-ahead and cone look-ahead algorithms for only 100 iterations to generate 100 execution sequences (and select minimum cost sequence). Note that these 100 iterations correspond to $M = 100$ in the node look-ahead and cone look-ahead algorithms. For this evaluation, we use $k = 1$ and $k = 3$ for the node look-ahead and cone look-ahead algorithms, respectively which are the minimum k values for either algorithm. As previously shown in Table 4, the enumeration algorithm fails to scale for adders with higher bit-widths. Therefore, a direct comparison among different algorithms could only be performed for adders with bit-widths up to 2-bit. Table 5 shows that both the node look-ahead and cone look-ahead algorithms incur the same memory cost as the optimal enumeration algorithm for smaller adders. While it is expected that the proposed look-ahead algorithms might not be able to achieve optimal results for adders with higher bit-widths, we show in the next section that the proposed algorithms are capable of generating an execution sequence within a practical runtime that substantially minimizes the intermediate memory cost.

7.3.5 Comparison of the Practical Algorithms: Node Look-Ahead vs. Cone Look-Ahead. In this section we compare the performance and complexity of the node look-ahead and the cone look-ahead algorithms. We first define a baseline *random* algorithm to compare the improvements achieved by the proposed algorithms. The random algorithm is based on generating execution by performing entirely random sampling of candidate nodes. Unlike the proposed

Table 5. Performance Comparison of Different Algorithms.

Adder Description	Intermediate Storage Cost for Different Algorithms		
	Enumeration	Node Look-Ahead	Cone Look-Ahead
half adder	3	3	3
full adder	4	4	4
2-bit adder	5	5	5



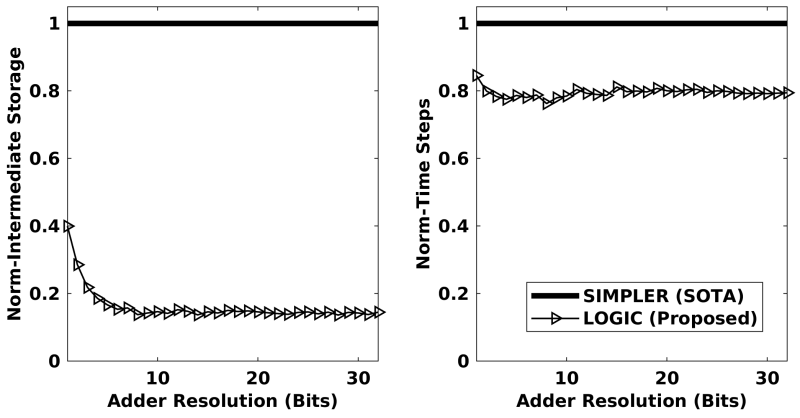
(a) (b)

Fig. 21. Comparison of the performance of different practical algorithms. (a) Normalized cost for intermediate storage for different adder bit-widths. (b) Normalized Runtime/Iteration for different adder bit-widths.

algorithms, the random algorithm does not make any informed decision while including a candidate node into the execution sequence. Instead, the algorithm simply assigns an equal EIC to all the candidate nodes. For the node look-ahead algorithm, we utilize the results for $k = 2$ as we were unable to generate the results for adders with higher bit-widths for $k > 2$. For the cone look-ahead algorithm, we utilize the results for $k = 25$.

Figure 21(a) shows the comparative intermediate storage requirement of different adders using different algorithms. The figure shows, compared with the random algorithm, the node look-ahead algorithm reduces the intermediate memory cost on average by 41%. However, the cone look-ahead algorithm achieves the best performance of the three algorithms. The algorithm reduces the intermediate memory cost by an additional 19% on average compared with the node look-ahead algorithm. This superior performance is the result of that the cones within the cone look-ahead algorithm are essentially trees of nodes. When a cone is processed, it is more likely that some of the child nodes of that tree gets expired and the corresponding memory can be released.

Figure 21(b) shows the average runtime/iteration for different algorithms. The figure shows that the random algorithm incurs the least runtime. This is expected as the random algorithm simply makes un-informed random choices while generating execution sequences. On the other hand, the node look-ahead algorithm incurs the worst runtime. This can be explained using the time complexity of the node look-ahead algorithm. The time complexity of the node look-ahead algorithm is $O(N \times V^k)$ where V is the total number of nodes in the DAG, k is the maximum size of the sets constructed from the candidate nodes, and, $N = (V \setminus$ primary input nodes). Due to the exponential dependence on k, the node look-ahead algorithm scales poorly. On the other hand, the time complexity of the cone look-ahead algorithm is $O(N \times (V + E))$ where V is the number of nodes, E is the number of edges in the DAG, and, $N = (V \setminus$ primary input nodes). It is evident that the cone look-ahead algorithm scales much better compared to the node look-ahead algorithm. This can be experimentally observed in the figure where the cone look-ahead reduces the runtime by 991$\times$ on average compared to the node look-ahead algorithm.



(a) (b)

Fig. 22. Area-latency overhead comparison of the proposed synthesis approach with the state-of-the-art in-memory computing approach SIMPLER [3].

From the experiments, we can conclude that the cone look-ahead algorithm provides superior performance and efficiency compared to the other algorithms. Therefore, in the following section we will incorporate the cone look-ahead algorithm into the LOGIC framework to compare with the state-of-the-art.

7.4 Comparison with the State-of-the-Art

In this section, we present a comparative performance evaluation of the proposed framework and the state-of-the-art computing framework.

7.4.1 Element-wise Arithmetic. Figure 22 shows a comparative area-latency evaluation of the proposed and the state-of-the-art SIMPLER [3] paradigms for in-memory arithmetic operations. The results show, that for variable bit-width fixed-point multiplications, the LOGIC framework reduces the area cost by 84% on average. Additionally, the latency is improved by 20% on average for the LOGIC framework. These improvements are accumulated over different steps in the synthesis process. For instance, the logic synthesis step within the LOGIC framework utilizes an automated synthesis flow using ABC. This advanced automated synthesis reduces the number of total gates compared to the manually synthesized netlists in the state-of-the-art. Additionally, the technology mapping step further reduces the number of total gates as shown in Section 7.2. This reduction in gates translates into reduction of in-memory operation and therefore a reduction in overall latency.

From the experimental results it is clear that compared with the manual decomposition-based approach, the proposed automated synthesis-based approach is substantially superior in performance.

7.4.2 Evaluation of Scientific Computing Application. In this section, we evaluate the performance of the LOGIC framework using sparse MVM dominated scientific computing applications.

Systems of linear equations are solved using generalized minimal residual method (GMRES) [50] and conjugate gradient method (CG) [9]. These method are centered on iteratively refining the system solution by performing an MVM operation in each iteration. The iterative MVM operations are the dominating computation in these processes. We aim to employ the LOGIC framework based in-memory computing platform to accelerate these costly MVM operations.

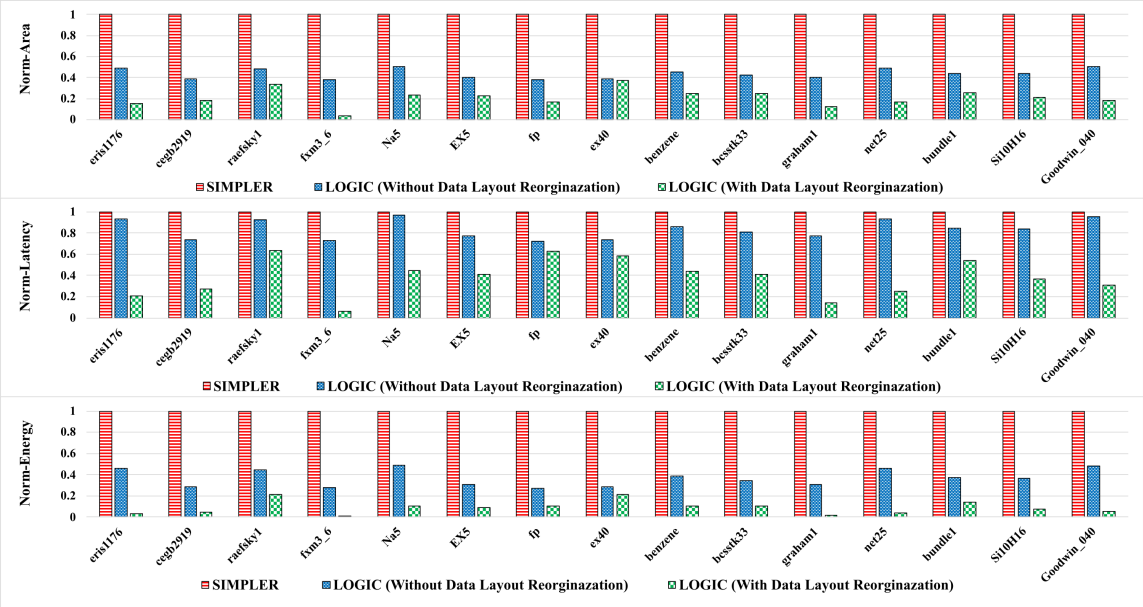


Fig. 23. Comparative overhead evaluation of the proposed and the state-of-the-art SIMPLER [3] frameworks for 15 scientific applications from SuiteSparse Matrix collection [10].

In Table 2, we select 15 benchmarks from the Suite Sparse Matrix collection [10]. The selected matrices are from different scientific domains and offer different sparsity patterns. We perform a comparative evaluation of area-latency-energy overhead with respect to the SIMPLER framework which is the state-of-the-art in-memory paradigm for MVM operations [3]. We evaluate LOGIC for both with and without data-layout re-organization. The comparative results are shown in Figure 23.

The results show that the LOGIC framework (without data layout reorganization) achieves area-latency-energy improvements in the order of 2.3×, 1.2×, and 2.7× respectively when compared with SIMPLER [3]. These improvements are the result of that in LOGIC, each arithmetic operation is performed within a more compact area and with fewer time steps. Additionally, the cross-architecture data communication is reduced as fewer crossbars are required. This directly translates into improvement in latency and overall energy consumption.

The results also show that LOGIC (with data layout organization) further improves area-latency-energy in the order of 2.1×, 2.2×, and 4.1×, respectively when compared with LOGIC (without data layout re-organization). These improvements are the results of a more efficient alignment of the matrix and input vector data within the architecture. The denser data layout further reduces the cross-architecture data communication cost which significantly improves the latency and energy efficiency.

From the results, it is evident that the proposed automated synthesis-based framework shows remarkable improvements over the manual template-based state-of-the-art framework [3]. These improvements are the result of the progress made in logic synthesis over the previous decades.

8 Conclusion

In this paper, we introduce the LOGIC framework which is designed for the efficient translation of high-level applications into digital in-memory compute kernels. Our framework incorporates automated techniques that reduces the number of in-memory computing operations which in turns reduces the required time-steps for execution. Additionally, we optimize the sequence in which these in-memory operations are executed to minimize the utilization of non-volatile memory for storing intermediate data. Furthermore, we propose a method for re-organizing data layouts to transform sparse MVM operations into dense MVM operations, thereby reducing inter-crossbar communication. When compared to manually designed template-based approaches, our approach demonstrates significant improvements in area, latency, and energy efficiency by factors of $4.8\times$, $2.6\times$, and $11\times$, respectively. In our future research, we intend to extend the capabilities of LOGIC to enable the automatic decomposition of entire multiplications without relying on a hierarchical approach. Additionally, we plan to explore the integration of LOGIC with other digital in-memory computing logic styles.

Acknowledgments

This work was in part supported by NSF awards #2319399, and #2404036.

References

[1] T Ali, P Polakowski, S Riedel, T Büttner, T Kämpfe, M Rudolph, B Pätzold, K Seidel, D Löhr, R Hoffmann, et al. 2018. High endurance ferroelectric hafnium oxide-based FeFET memory without retention penalty. *IEEE Transactions on Electron Devices* 65, 9 (2018), 3769–3774.

[2] Janna Anderson and Lee Rainie. 2022. The metaverse in 2040. *Pew Research Centre* 30 (2022).

[3] Rotem Ben-Hur, Ronny Ronen, Ameer Haj-Ali, Debjyoti Bhattacharjee, Adi Eliahu, Natan Peled, and Shahar Kvatinsky. 2019. 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* 39, 10 (2019), 2434–2447.

[4] Julien Borghetti, Gregory S Snider, Philip J Kuekes, J Joshua Yang, Duncan R Stewart, and R Stanley Williams. 2010. 'Memristive'switches enable 'stateful'logic operations via material implication. *Nature* 464, 7290 (2010), 873–876.

[5] Widodo Budiharto, Edy Irwansyah, Jarot Sembodo Suroso, and Alexander Agung Santoso Gunawan. 2020. Design of object tracking for military robot using PID controller and computer vision. *ICIC Express Letters* 14, 3 (2020), 289–294.

[6] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. 2010. Phase change memory technology. *Journal of Vacuum Science & Technology B* 28, 2 (2010), 223–262.

[7] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109* (2023).

[8] Long Cheng, Mei-Yun Zhang, Yi Li, Ya-Xiong Zhou, Zhuo-Rui Wang, Si-Yu Hu, Shi-Bing Long, Ming Liu, and Xiang-Shui Miao. 2017. Reprogrammable logic in memristive crossbar for in-memory computing. *Journal of Physics D: Applied Physics* 50, 50 (2017), 505102.

[9] James W Daniel. 1967. The conjugate gradient method for linear and nonlinear operator equations. *SIAM J. Numer. Anal.* 4, 1 (1967), 10–26.

[10] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *TOMS* 38, 1 (2011), 1–25.

[11] Herbert B Enderton. 2001. *A mathematical introduction to logic*. Elsevier.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*. 365–376.

[13] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. 2018. Enabling scientific computing on memristive accelerators. In *2018 ACM/IEEE 45th ISCA*. IEEE, 367–382.

[14] John Gantz and David Reinsel. 2012. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future* 2007, 2012 (2012), 1–16.

[15] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 100–113.

[16] Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, and Shahar Kvatinsky. 2018. Efficient algorithms for in-memory fixed point multiplication using magic. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[17] Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, Ronny Ronen, and Shahar Kvatinsky. 2018. Not in name alone: A memristive memory processing unit for real in-memory processing. *IEEE Micro* 38, 5 (2018), 13–21.

[18] Eman Hassan, Yasmin Halawani, Baker Mohammad, and Hani Saleh. 2021. Hyper-dimensional computing challenges and opportunities for AI applications. *IEEE Access* 10 (2021), 97651–97664.

[19] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).

[20] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R Stanley Williams. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *2016 53nd ACM/EDAC/IEEE DAC*. IEEE, 1–6.

[21] Yihong Hu, Nuo Xu, Chaochao Feng, Wei Tong, Kang Liu, and Liang Fang. 2024. LOSSS-Logic Synthesis based on Several Stateful logic gates for high time-efficient computing. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 805–811.

[22] Yiming Huai et al. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin* 18, 6 (2008), 33–40.

[23] Rotem Ben Hur, Nimrod Wald, Nishil Talati, and Shahar Kvatinsky. 2017. SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic. In *2017 IEEE/ACM ICCAD*. IEEE, 225–232.

[24] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA*. IEEE, 802–815.

[25] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. 2020. Dual: Acceleration of clustering algorithms using digital-based processing in-memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 356–371.

[26] Sumit Kumar Jha, Dilia E Rodriguez, Joseph E Van Nostrand, and Alvaro Velasquez. 2016. Computation of boolean formulas using sneak paths in crossbar computing. US Patent 9,319,047.

[27] Yuchen Jiang, Shen Yin, Kuan Li, Hao Luo, and Okyay Kaynak. 2021. Industrial applications of digital twins. *Philosophical Transactions of the Royal Society A* 379, 2207 (2021), 20200360.

[28] Maria G Juarez, Vicente J Botti, and Adriana S Giret. 2021. Digital twins: Review and challenges. *Journal of Computing and Information Science in Engineering* 21, 3 (2021), 030802.

[29] Vijay Kakani, Van Huan Nguyen, Basivi Praveen Kumar, Hakil Kim, and Visweswara Rao Pasupuleti. 2020. A critical review on computer vision and artificial intelligence in food industry. *Journal of Agriculture and Food Research* 2 (2020), 100033.

[30] Kurt Keutzer. 1987. DAGON: Technology binding and local optimization by DAG matching. In *DAC*. 341–347.

[31] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. 2014. MAGIC—Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[32] Shahar Kvatinsky, Misbah Ramadan, Eby G Friedman, and Avinoam Kolodny. 2015. VTEAM: A general model for voltage-controlled memristors. *TCAS-II: Express Briefss* 62, 8 (2015), 786–790.

[33] Manuel Le Gallo, Abu Sebastian, Roland Mathis, Matteo Manica, Heiner Giefers, Tomas Tuma, Costas Bekas, Alessandro Curioni, and Evangelos Eleftheriou. 2018. Mixed-precision in-memory computing. *Nature Electronics* 1, 4 (2018), 246–253.

[34] Seunggyu Lee, Wonjae Lee, and Youngsoo Shin. 2024. Integrated Netlist Synthesis and In-Memory Mapping for Memristor-Aided Logic. In *Proceedings of the Great Lakes Symposium on VLSI 2024*. 38–43.

[35] Orian Leitersdorf, Ronny Ronen, and Shahar Kvatinsky. 2021. MultPIM: Fast stateful multiplication for processing-in-memory. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 3 (2021), 1647–1651.

[36] Can Li, Miao Hu, Yunning Li, Hao Jiang, Ning Ge, Eric Montgomery, Jiaming Zhang, Wenhao Song, Noraica Dávila, Catherine E Graves, et al. 2018. Analogue signal and image processing with large memristor crossbars. *Nature Electronics* 1, 1 (2018), 52.

[37] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[38] Robert P Loce, Raja Bala, Mohan Trivedi, and John Wiley. 2017. *Computer vision and imaging in intelligent transportation systems*. Wiley Online Library.

[39] Arif Furkan Mendi, Tolga Erol, and Dilara Doğan. 2021. Digital twin in the military field. *IEEE Internet Computing* 26, 5 (2021), 33–40.

[40] Alan Mishchenko et al. [n. d.]. ABC: A system for sequential synthesis and verification. "http://www.eecs.berkeley.edu/alanmi/abc".

[41] Sparsh Mittal, Gaurav Verma, Brajesh Kaushik, and Farooq A Khanday. 2021. A survey of SRAM-based in-memory computing techniques and applications. *Journal of Systems Architecture* 119 (2021), 102276.

[42] Engineering National Academies of Sciences, Medicine, et al. 2019. Quantum computing: progress and prospects. (2019).

[43] Huansheng Ning, Hang Wang, Yujia Lin, Wenxi Wang, Sahraoui Dhelim, Fadi Farha, Jianguo Ding, and Mahmoud Daneshmand. 2023. A Survey on the Metaverse: The State-of-the-Art, Technologies, Applications, and Challenges. *IEEE Internet of Things Journal* (2023).

[44] Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2021. Hybrid analog-digital in-memory computing. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[45] Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2022. Logic Synthesis for Digital In-Memory Computing. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[46] Muhammad Rashedul Haq Rashed, Sven Thijssen, Sumit Kumar Jha, Fan Yao, and Rickard Ewetz. 2023. STREAM: Towards READ-based In-Memory Computing for Streaming Based Processing for Data-Intensive Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).

[47] Kaushik Roy, Indranil Chakraborty, Mustafa Ali, Aayush Ankit, and Amogh Agrawal. 2020. In-memory computing in emerging memory technologies for machine learning: An overview. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[48] Andrey Rudskoy, Igor Ilin, and Andrey Prokhorov. 2021. Digital twins in the intelligent transport systems. *Transportation Research Procedia* 54 (2021), 927–935.

[49] David Reinsel-John Gantz-John Rydning, John Reinsel, and John Gantz. 2018. The digitization of the world from edge to core. *Framingham: International Data Corporation* 16 (2018), 1–28.

[50] Youcef Saad and Martin H Schultz. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing* 7, 3 (1986), 856–869.

[51] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 273–287.

[52] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[53] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.

[54] Peter Svenmarck, Linus Luotsinen, Mattias Nilsson, and Johan Schubert. 2018. Possibilities and challenges for artificial intelligence in military applications. In *Proceedings of the NATO Big Data and Artificial Intelligence for Military Decision Making Specialists' Meeting*. 1–16.

[55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[56] Nishil Talati, Ameer Haj Ali, Rotem Ben Hur, Nimrod Wald, Ronny Ronen, Pierre-Emmanuel Gaillardon, and Shahar Kvatinsky. 2018. Practical challenges in delivering the promises of real processing-in-memory machines. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1628–1633.

[57] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology* 15, 4 (2016), 635–650.

[58] Sven Thijssen, Muhammad Rashedul Haq Rashed, Sumit Kumar Jha, and Rickard Ewetz. 2023. PATH: Evaluation of Boolean Logic Using Path-Based In-Memory Computing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).

[59] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine* (2023), 1–11.

[60] Jane Thomason. 2021. Big tech, big data and the new world of digital health. *Global Health Journal* 5, 4 (2021), 165–168.

[61] Joe Touch, Abdel-Hameed Badawy, and Volker J Sorger. 2017. Optical computing. , 503–505 pages.

[62] Elie Track, Nancy Forbes, and George Strawn. 2017. The end of Moore's Law. *Computing in Science & Engineering* 19, 2 (2017), 4–6.

[63] Naveen Verma, Hongyang Jia, Hossein Valavi, Yinqi Tang, Murat Ozatay, Lung-Yen Chen, Bonan Zhang, and Peter Deaville. 2019. In-memory computing: Advances and prospects. *IEEE Solid-State Circuits Magazine* 11, 3 (2019), 43–55.

[64] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, et al. 2018. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience* 2018 (2018).

[65] Ge Wang, Andreu Badal, Xun Jia, Jonathan S Maltz, Klaus Mueller, Kyle J Myers, Chuang Niu, Michael Vannier, Pingkun Yan, Zhou Yu, et al. 2022. Development of metaverse for intelligent healthcare. *Nature Machine Intelligence* 4, 11 (2022), 922–929.

[66] Zhuo-Rui Wang, Yi Li, Yu-Ting Su, Ya-Xiong Zhou, Long Cheng, Ting-Chang Chang, Kan-Hao Xue, Simon M Sze, and Xiang-Shui Miao. 2018. Efficient implementation of Boolean and full-adder functions with 1T1R RRAMs for beyond von Neumann in-memory computing. *IEEE Transactions on Electron Devices* 65, 10 (2018), 4659–4666.

[67] Maurice V Wilkes. 1995. The memory wall and the CMOS end-point. *SIGARCH* 23, 4 (1995), 4–6.

[68] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH* 23, 1 (1995), 20–24.

[69] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–488.

[70] Shimeng Yu. 2016. Resistive random access memory (RRAM). *Synthesis Lectures on Emerging Engineering Technologies* 2, 5 (2016), 1–79.

[71] Qilin Zheng, Xingchen Li, Zongwei Wang, Guangyu Sun, Yimao Cai, Ru Huang, Yiran Chen, and Hai Li. 2020. MobiLattice: A Depth-wise DCNN Accelerator with Hybrid Digital/Analog Nonvolatile Processing-In-Memory Block. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[72] Alwin Zulehner, Kamalika Datta, Indranil Sengupta, and Robert Wille. 2019. A staircase structure for scalable and efficient synthesis of memristor-aided logic. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 237–242.

Received 04 July 2024; revised 14 October 2024; accepted 11 December 2024