STREAM: Towards READ-based In-Memory Computing for Streaming Based Processing for Data-Intensive Applications

Muhammad Rashedul Haq Rashed*, Sven Thijssen†, Sumit Kumar Jha‡, Fan Yao*, and Rickard Ewetz*
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, USA
†Department of Computer Science, University of Central Florida, Orlando, USA
‡Computer Science Department, University of Texas at San Antonio, San Antonio, USA
{rashed09, sven.thijssen}@knights.ucf.edu, sumit.jha@utsa.edu, {fan.yao, rickard.ewetz}@ucf.edu

***Abstract*—With the rise of data intensive applications, traditional computing paradigms have hit the *memory-wall*. In-memory computing using emerging non-volatile memory (NVM) technology is a promising solution strategy to overcome the limitations of the von-Neumann architecture. In-memory computing using NVM devices has been explored in both analog and digital domains. Analog in-memory computing can perform matrix-vector multiplication (MVM) in an extremely energy-efficient manner. However, analog in-memory computing is prone to errors and resulting precision is therefore low. On the contrary, digital in-memory computing is a viable option for accelerating scientific computations that require deterministic precision. In recent years, several digital in-memory computing styles have been proposed. Unfortunately, state-of-the-art digital in-memory computing styles rely on repeated WRITE operations which involves switching of NVM devices. WRITE operations in NVM cells are expensive in terms of energy, latency and device endurance. In this paper, we propose a READ-based in-memory computing framework called STREAM. The framework performs streaming-based data processing for data-intensive applications. The STREAM framework consists of a synthesis tool that decomposes an arbitrary Boolean function into in-memory compute kernels. Two synthesis approaches are proposed to generate READ-based in-memory compute kernels using data structures from logic synthesis. A hardware/software co-design technique is developed to minimize the inter-crossbar data communication. The STREAM framework is evaluated using circuits from ISCAS85 benchmark suite, and Suite-Sparse applications to scientific computing. Compared with state-of-the-art in-memory computing framework, the proposed framework improves latency and energy performance with up to $200X$ and $20X$, respectively.**

I. Introduction

With the advent of internet-of-things [1], modern computing paradigms are burdened with an unprecedented amount of digital data [2–4]. Data intensive applications such as computer-vision [5] and system simulation [6] are rendering high performance computing (HPC) systems ill-equipped. This is due to that the traditional von Neumann based HPC systems are limited by the "memory wall" due to the separation of the memory and processing units [7, 8]. With the objective of enabling scaleable big data processing, emerging computing paradigms such as optical computing [9], quantum computing [10], and near/in-memory computing [11, 12] are receiving increasing attention.

In-memory computing using the emerging non-volatile memory (NVM) devices [13–15] promises to break the memory wall bottleneck of von Neumann architecture. Analog computation using the multiply-and-accumulate feature of in-memory computing platforms has shown superior energy and latency performance compared to the traditional CMOS based computing platforms [16–20]. However, the precision of analog in-memory computing is limited and therefore it is not suitable for high-precision demanding scientific computations [21, 22].

To overcome the computational accuracy issue, digital in-memory computing based on in-memory Boolean logic operations has been proposed. Several digital in-memory logic styles such as IMPLY [23], MAGIC [24], FLOW [25], Bit-wise-in-bulk [26], PATH [27], and OR-plane [28] logic have been developed over the years. While all of these logic styles offer deterministic precision, they have their distinctive operating principles and they perform differently in terms of area, energy and latency. State-of-the-art digital in-memory computing paradigms are based on evaluating Boolean functions using repeated WRITE operations [29–34]. Unfortunately, WRITE operations in NVM cells are both power-hungry and slow [35].

In this paper, we propose a framework that performs READ-based in-memory computing for streaming-based processing for data-intensive applications, which is called STREAM. The framework adopts the OR-plane logic style to evaluate high fan-in OR/NOR gates using efficient READ operations. The main difficulty of utilizing OR-plane logic comes from that inter-crossbar data transfer is required to evaluate complex functions. This leads to substantial performance and hardware overheads especially when processing big data.

To address these issues, the STREAM framework provides a software/hardware-centered solution to minimize overheads[1]. With the objective of minimizing the data transfer costs, we first design processing elements (PEs) that have multiple series-connected crossbars using hardwired connections. Next, a synthesis tool is developed that can map arbitrary

This work was in part supported by NSF awards CNS-1908471, CNS-2008339, CCF-1822976, CCF-2113307, DARPA cooperative agreement #HR00112020002 and ONR grant #N000142112332.

[1]A preliminary version of the framework has been published in [36].

Boolean functions to the PEs. We propose two synthesis approaches called STEAM-O and STREAM-M, which are based on data structures for traditional logic synthesis. The STEAM-O approach is based on OR-Inverter Graphs (OIG) that can straightforward be mapped to a single crossbar. The STREAM-M approach is based on multi-level logic (MLL) that can directly be mapped to two crossbars connected in series. Furthermore, we develop an algorithm to decompose complex Boolean functions into parts that fit into the PEs while minimizing the costly inter-PE communication.

We evaluate the STREAM framework using circuits from the ISCAS85 benchmark suite, and Suit-Sparse applications to scientific computing. We compare the results with the state-of-the-art WRITE-based in memory computing paradigms.

The main contributions of the STREAM framework are, as follows:

- A novel staircase architecture for READ-based in-memory computing using OR-plane logic.
- A synthesis tool for mapping Boolean logic into in-memory compute kernels using OIGs and MLL data structures from logic synthesis.
- A (i) spatial partitioning technique and (ii) a bit-wise partitioning technique for decomposing complex computations into parts that each fit inside a PE.
- The STREAM framework is evaluated on the ISCAS85 benchmark suite [37], and the Suit-Sparse matrix collection [38]. A comparison with CMOS ASIC-like systems is also performed.
- Compared with the state-of-the art WRITE-based in-memory computing paradigms, the proposed framework improves average power and latency with up to $20X$ and $200X$, respectively.

The remainder of this paper is organized as follows: Preliminaries in Section II. An overview of the STREAM framework is given in Section III. The synthesis tool STREAM-O is detailed in Section IV. Section V describes the synthesis tool STREAM-M. Partitioning techniques for applications based on matrix-vector multiplication are proposed in Section VI. The STREAM architecture is detailed in Section VII. Experimental results are presented in Section VIII. Other related works are discussed in Section IX. The paper is concluded in Section X.

II. Preliminaries

In this section, we first review WRITE-based and READ-based digital in-memory computing. Next, we review two data structures for logic synthesis. Finally, we discuss the limitations of previous works and motivate the proposed framework.

A. Digital in-memory computing

In this section, we explain the working principle of digital in-memory computing. Specifically, we illustrate how digital in-memory computing can be used to evaluate Boolean logic operations. Digital in-memory computing can be divided into two major branches: WRITE-based in-memory computing and READ-based in-memory computing. We review MAGIC [24] and OR-plane logic [28] as representative logic styles for WRITE-based and READ-based in-memory computing, respectively.

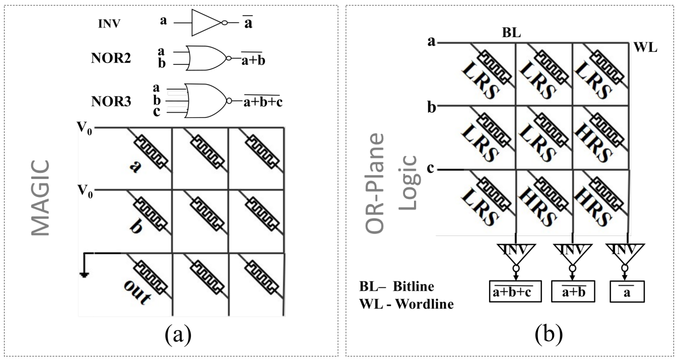


Fig. 1: Evaluation of Boolean gates using (a) WRITE-based MAGIC and (b) READ-based OR-plane logic.

WRITE-based in-memory computing: WRITE-based in-memory computing is based on evaluating Boolean gates by repeatedly switching non-volatile memory (NVM) devices. The devices are switched using special WRITE operations. Memristor devices are programmed either to high resisitve state (HRS) or low resisitive state (LRS) which correspond to logic “0” and logic “1” respectively. Figure 1(a) shows how Boolean NOR operation can be performed using MAGIC style in-memory computing [24]. In MAGIC, both the input and the output operands are stored inside the memory. A MAGIC operation is performed in two steps: an initialization step and an evaluation step. In the initialization step, the *out* memristor is programmed to LRS and the input variables *a* and *b* are programmed along the same bitline (or wordline) [24]. In the evaluation step, the NOR2 operation $out = \overline{a+b}$ is realized by applying controlled voltage V_0 and ground to the input and the output memristors respectively as shown in Figure 1(a). MAGIC enables parallel logic NOR operations by programming inputs and outputs to multiple bitlines (or wordlines).

READ-based in-memory computing: READ-based in-memory computing is based on evaluating Boolean logic operations using READ operations. Figure 1(b) illustrates how variable input NOR operations can be performed using the OR-plane logic [28]. In OR-plane logic, an arbitrary input OR gate can be realized by setting the memristors in a bitline either to the HRS or the LRS. Dedicated inverters in the peripheral enables arbitrary input NOR operations. Input operands *a*, *b*, and *c* are fed as binary voltages to the wordlines as shown in the Figure 1(b). A single READ operation is used to decode the outputs of the INV/NOR2/NOR3 functions parallelly from the bitlines, respectively. The evaluation involves no WRITE operations to the non-volatile memory devices.

B. Data structures for logic synthesis

Logic synthesis for traditional CMOS technology has been explored using data structures such as AIG [39], OIG [40], MLL [41], BDD and ROBDD [42, 43]. In this section, we review the OIG and MLL data structures due to their underlying similarity with in-memory computing kernels.

OR-Inverter Graphs: OR-Inverter Graphs (OIGs) is a class of directed acyclic graphs (DAG), $O = (V, E)$ where a node, $v \in V$, represents either an inverter or a variable input OR gate and an edge, $e \in E$, represents an interconnection between two nodes of the graph. OIGs are a complete data structure that can represent any Boolean function. The minimization of OIGs has been studied in [40]. The number of nodes and the depth of the OIGs is minimized by iteritively applying logic transformations based on merging and splitting nodes [40].

Multi-Level Logic: The multi-level logic (MLL), as the name suggests, has several levels of Boolean logic. MLL can be visualized as a graph, $M = (V, E)$ where a node, $v \in V$, represents a two-level logic and an edge, $e \in E$, represents an interconnection between two nodes. Here, the two-level logic is a representation of a Boolean function, f, as the disjunction (Boolean OR) of a set of products (Boolean AND) of its literals. This representation is also known as the sum of products (SOP). Each level of the MLL consists of several nodes, each node containing a SOP expression. The minimization of MLL has been studied in [44]. The minimization techniques include node factorization, collapsing of nodes, and removal of redundancies.

C. Limitations of previous work

In this section, we present a comparison of the working principles of different digital in-memory computing paradigms. All of the computing paradigms consist of a one-time initialization phase and an evaluation phase. In the initialization phase, the computing platforms are programmed to perform arbitrary Boolean function. In the evaluation phase, the Boolean functions are evaluated for different instances of the input variables. The initialization and the evaluation is performed either by the WRITE or the READ operations. Table I summarizes the use of WRITE and READ operations for different computing paradigms.

TABLE I: Comparison of WRITE/READ operations in the initialization and evaluation phase for different logic styles.

Logic style	Work in	Initialization phase	Evaluation phase
IMPLY	[23]	WRITE/READ	WRITE/READ
MAGIC	[24]	WRITE/READ	WRITE/READ
Flow-based computing	[25]	WRITE/READ	WRITE/READ
Bitwise-In-Bulk	[26]	WRITE/READ	WRITE/READ
OR-plane logic	(this work)	WRITE/READ	**READ**

It is evident from the table that the state-of-the-art digital in-memory computing paradigms depend on repeated WRITE operations in the evaluation phase. Unfortunately, WRITE operations to NVM devices are expensive in terms of speed and energy-cost. In contrast, the proposed STREAM framework is based on OR-plane logic [28] which only uses READ operations in the evaluation phase. Contrary to the WRITE operation, the READ operations are fast and very energy-efficient. For example, the latency of a WRITE and a READ operations are 50.88ns and 29.31ns respectively [35]. Also, the energy-cost for a WRITE and a READ operations are 3.91nJ and 1.08pJ respectively [35]. Therefore, the READ operations in NVM cells are roughly 2X faster and 4000X more energy effcient than the WRITE operations. One additional benefit of READ-based evaluation phase is that the endurance of NVM cells are improved. The lifespan of NVM devices are estimated in terms of the number of WRITE operations before device breakdown. Current technology can fabricate NVM devices with expected lifespan in the range of only 10^3 to 10^9 WRITE operations [45]. Therefore, compared to the WRITE-based in-memory computing paradigms, the READ-based in-memory computing paradigm is expected to have a longer device lifetime.

While OR-plane logic has advantageous properties, it requires multiple crossbars to be connected together in series to evaluate arbitrary Boolean functions. We call a set of series connected crossbars a *staircase structure*. In such structure, the primary inputs are fed to the first crossbar and the evaluation results are obtained from the last crossbar. The intermediate crossbars each receives inputs from its previous crossbar and feeds outputs to the following crossbar. The outlined approach becomes increasingly challenging for data-intensive applications since the data transfer between crossbars introduce substantial performance overheads when performed using reconfigurable interconnects (or busses).

On the other hand, performance can be improved if interconnects are hardwired together. However, it becomes more challenging to maximize utilization and handle hardware imposed constraints. Table II shows the trade-off between performance and ease-of-utilization for hardwired and reconfigurable connections. The objective of the STREAM framework is to combine software/hardware co-design to enable streaming-based processing. The goal is to establish a balance of overheads introduced by the utilization ease of reconfigurability and the efficiency of hardwiring.

TABLE II: Hardwired vs. reconfigurable connections.

	Performance	Ease of utilization
Hardwired	high	difficult
Reconfigurable	low	smooth

III. The STREAM Framework

In this section, we introduce the STREAM framework. The framework consists of an in-memory computing platform and a synthesis tool capable of mapping computation to the platform, which is shown in Figure 2. The platform consists of processing elements (PEs) connected together using high-speed interconnects. The PEs mainly consist of a staircase structure of connected crossbars. The details of the PEs are provided in Section VII. The input to the synthesis tool is a specification of a Boolean function. The synthesis tool maps the computation into in-memory compute kernels and binds the kernels to the in-memory computing platform. Next, streaming-based processing is performed by providing input data to the reconfigured platform.

In the STREAM framework, we break the synthesis problem into two parts, as follows:

- **Problem I:** The first subproblem consists of mapping an arbitrary Boolean function to a PE with relaxed hardware constraints. Here, it is assumed that the crossbars are of

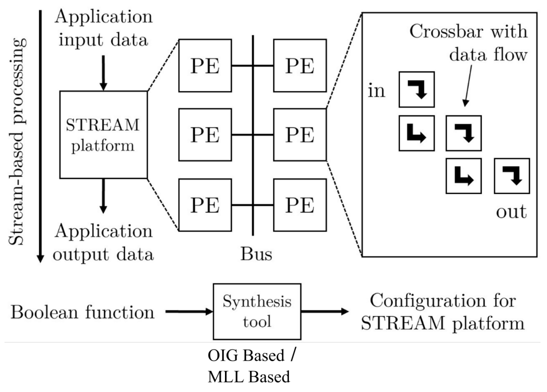


Fig. 2: Overview of the STREAM framework.

arbitrary dimension and there are an arbitrary number of crossbars connected in series.

- **Problem II:** The second subproblem consists of decomposing the Boolean function into multiple parts. The objective is to map each part to a PE using the solution to Problem I while satisfying the hardware constraints on the PEs, e.g., there is a fixed number of crossbars with fixed dimensions within each individual PE.

A synthesis solution on OIGs and MLL is proposed in Section IV and Section V, respectively. Both OIGs and MLL capture a Boolean function using a DAG of nodes. In an OIG, each node can be mapped to a single crossbar. On the other hand, each node within MLL can be mapped to two series-connected crossbars. We explore both approaches to empirically analyse the trade-off in terms of performance and hardware cost in Section VIII.

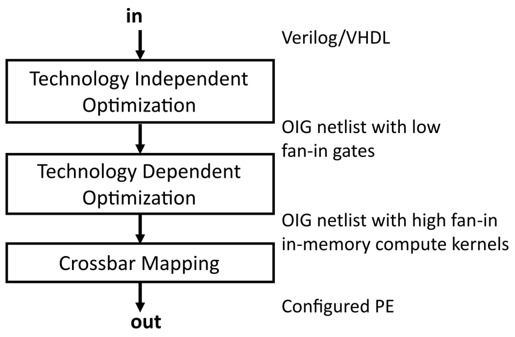
After solving the first subproblem, it is crucial to tailor the in-memory computation to the architectural resources. This leads to the second subproblem as discussed above. Section VI presents a synthesis solution to the second subproblem. For the choice of data-intensive applications, high precision matrix-vector multiplication within scientific simulation is considered.

IV. Synthesis Approach 1: STREAM-O

In this section, we provide an OIG-based synthesis solution for the first subproblem of the previous section. The objective is to map arbitrary Boolean functions to a staircase structure of crossbars. We call this synthesis approach STREAM-O. Figure 3 shows an overview of the flow of STREAM-O.

The synthesis tool takes a Boolean function f as the primary input. The final output of the synthesis tool consists of i) an allocation of the input variables of f to the first crossbar, ii) the resistance state of all memristor cells within the staircase structure, and iii) an assignment of the output variables of f to the last crossbars.

The overall synthesis process can be decomposed into three steps: a technology independent optimization step, a technology dependent optimization step, and a crossbar mapping step. In the technology independent optimization step, the input specification is synthesized into a netlist with low fan-in gates. In the technology dependent optimization step, the initial netlist is mapped into a netlist of high fan-in OR-INV in-memory compute kernels. In the crossbar mapping step, the in-memory compute kernels are sorted depth-wise and are bound to the crossbar staircase structure.



in
Verilog/VHDL
Technology Independent Optimization
OIG netlist with low fan-in gates
Technology Dependent Optimization
OIG netlist with high fan-in in-memory compute kernels
Crossbar Mapping
Configured PE
out

Fig. 3: Overview of the STREAM-O synthesis.

A. Technology Independent Optimization

The input to the technology independent optimization is a Boolean function specification in hardware description language (HDL), such as Verilog or VHDL. The input is passed into a standard logic synthesis tool, e.g., ABC [46]. ABC performs logic optimization based on And-Inverter Graphs (AIGs) [39].

We aim to take advantage of the efficient technology independent optimization within ABC when converting the Boolean function into in-memory compute kernels supported by OR-plane logic. Unfortunately, ABC was developed for CMOS technology with gates with low fan-in. However, OR-plane logic supports efficient evaluation of OR-gates with high fan-in. Note that every OR-gate is realized using a single bitline, regardless of the number of inputs. Therefore, we utilize ABC to synthesize a netlist consisting of INV and low fan-in OR-gates that is amenable for technology dependent optimization. The technology dependent optimization will merge many low fan-in gates into fewer high fan-in gate. We utilize a custom cell library consisting of INV and variable input OR gates to generate the OIG netlist. An important detail is that we set the cost of all the gates of the custom cell library within ABC to 1, which encourages the use of larger OR-gates.

B. Technology Dependent Optimization

The input to the technology dependent optimization step is the ABC generated netlist with low fan-in gates. This initial netlist consists of gates with at most 5 inputs since the target technology for the ABC tool is CMOS-based [46]. In contrast, in-memory computing platform based on OR-plane logic is capable of executing arbitrary-input OR/NOR gates using a single bitline. The technology dependent optimization step takes advantage of this by converting the initial netlist with low fan-in gates into a netlist with high fan-in gates that can be executed using OR-plane logic.

First, the input netlist is converted into a directed acyclic graph (DAG) $G=(V,E)$, where nodes V and edges E correspond to gates and wire connections, respectively. The graph is called a subject graph. Next, the subject graph is covered with in-memory compute kernels by performing technology dependent optimization. Lastly, a DAG representation of the netlist with high fan-in gates is extracted from the cover.

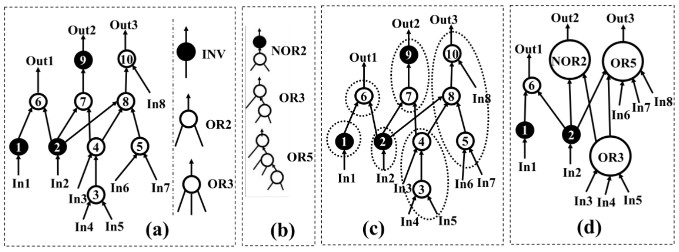


Fig. 4: Technology dependent optimization within STREAM-O. (a) Initial netlist with gate encoding, (b) library of in-memory compute kernels, (c) cover of subject graph, (d) optimized netlist.

The workflow of technology dependent optimization is illustrated with an example in Figure 4. Figure 4(a) shows a subject graph with 10 nodes. The encoding of gates within the graph is shown on the right of Figure 4(a). Figure 4(b) shows a subset of library of in-memory compute kernels that can be executed using OR-plane logic. In entirety, the cover library consists of both the rudimentary gates and high fan-in OR/NOR gates. A cover of the subject graph using the gates from the library in Figure 4(b) is shown in Figure 4(c). The cover is obtained using a tree-covering algorithm. The algorithm first decomposes the subject graph into multiple trees by breaking edges that have multiple fan-outs (e.g., gate 2 and 4 in Figure 4(a)). Next, each tree is covered with gates from the in-memory compute kernel library using the dynamic programming formulation in DAGON [47]. Figure 4(d) shows the resulting netlist after covering the complete subject graph. It can be observed that the initial netlist with 10 gates has been reduced to an in-memory compute kernel netlist with only 6 gates.

C. Crossbar Mapping

The input to the crossbar mapping step is the updated netlist obtained from the technology dependent optimization step. In this step, the netlist is bound to the staircase structured crossbars within a PE. Each crossbar within the PE can execute arbitrary in-memory compute kernel. However, it must be ensured that kernels that are adjacent in the netlist must be placed in adjacent crossbars. We solve this connection challenge by inserting dummy nodes to communicate signals to desired crossbars within the staircase.

We illustrate the proposed crossbar mapping algorithm with an example in Figure 5. Figure 5(a) shows the DAG representation of a netlist with 3 primary inputs and 4 nodes.

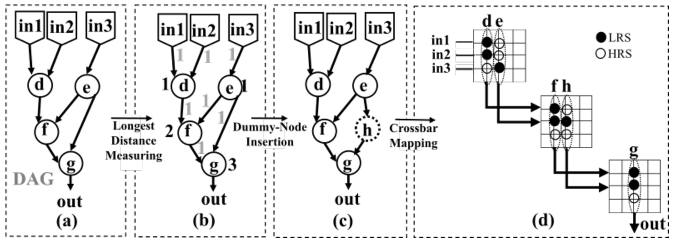


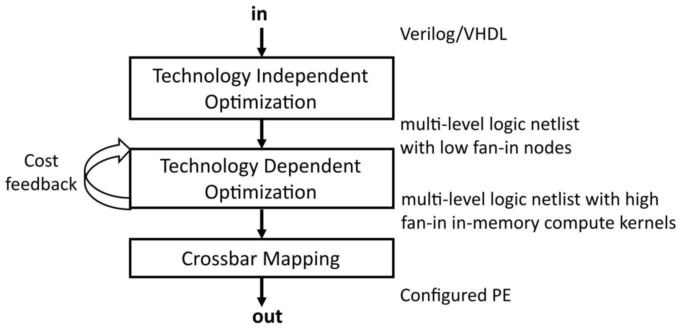
Fig. 5: Flow for binding in-memory kernels to crossbars. (a) Input netlist in DAG format, (b) longest distance to each node, (c) dummy node insertion and, (d) crossbar mapping.

Each node within the DAG represents an in-memory compute kernel that can be evaluated in a crossbar.

First step of the algorithm is to determine the longest path to each node in the graph, which is shown in Figure 5(b). This is achieve by performing a topological sort of DAG nodes which is followed by an in-order traversal. We assign a value of 1 to each edge in the DAG. The distance to a node is then defined as the sum of the edges along the longest path from the input layer to the respective node. Let each crossbar in the staircase structure be labeled layer 1 to layer N. The longest path to a node corresponds to the crossbar that the node will be assigned. The outlined method ensures that all connections go from crossbars with lower layers to higher layers. To eliminate connections that skip layers, dummy nodes realized by buffers are inserted into the netlist. The insertion of a dummy node between 'e' and 'g' is shown in Figure 5(c). The height and width of the crossbar layer l is equal to the number of nodes in layer (l) and $(l+1)$, respectively. Finally, it is straightforward to assign the kernels to the crossbars in the staircase structure, which is shown in Figure 5(d).

V. Synthesis Approach 2: STREAM-M

In this section, we provide a MLL-based synthesis solution for mapping a Boolean function to a staircase structure of crossbars. We call this synthesis approach STREAM-M and an overview of the synthesis flow is shown in Figure 6.



in
Verilog/VHDL
Technology Independent Optimization
multi-level logic netlist with low fan-in nodes
Cost feedback
Technology Dependent Optimization
multi-level logic netlist with high fan-in in-memory compute kernels
Crossbar Mapping
Configured PE
out

Fig. 6: Overview of the STREAM-M synthesis.

The input to the STREAM-M is the specification of a Boolean function. The output is a set of configured PEs to execute the specified Boolean function. STREAM-M consists of a technology independent optimization step, a technology dependent optimization step, and a crossbar mapping step. In the technology independent optimization step, the input specification is mapped into an initial MLL netlist. This initial netlist consists of low fan-in nodes. In general, MLL nodes with low fan-in result in OR gates with low fan-in. In the technology dependent optimization phase, low fan-in nodes are merged into high fan-in nodes, which reduces the hardware requirements. In the crossbar mapping phase, the in-memory compute kernels are bound to the crossbar staircase structure.

A. Technology Independent Optimization

In the technology independent optimization step, an initial multi-level netlist is created based on the input specifications of the Boolean function. The size of the netlist is next reduced

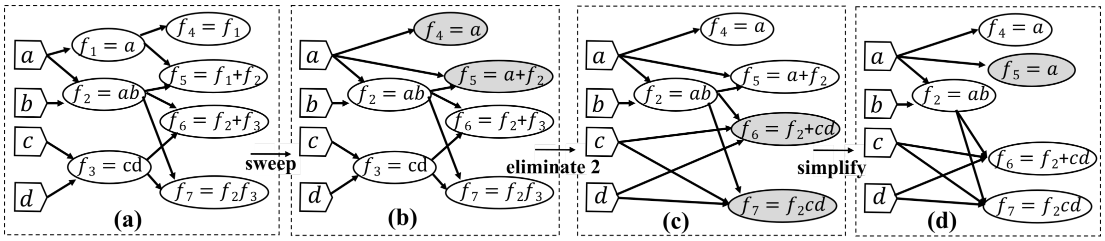


Fig. 7: An example of MLL synthesis step of STREAM-M. (a) Initial netlist, (b) netlist clean-up using *sweep* command, (c) selective node collapse using *eliminate* command and (c) SOP minimization using *simplify* command. Updated nodes in each step are marked in gray.

by iteratively applying transformations using the SIS tool [48]. Three key transformations are, as follows:

1) **sweep:** the *sweep* command is a clean-up operation that removes all the constant nodes (0 or 1) and the nodes with single fan-in. For instance, a subset of a MLL netlist is presented in Figure 7(a). The netlist consists of four primary inputs ($a-d$) and seven nodes ($f_1 - f_7$). Each of the node consists of a two-level logic. The node f_1 has a single fan-in. Therefore, when the *sweep* command is invoked on the netlist, the f_1 node is cleaned and its input a is collapsed into its fan-outs f_4 and f_5 as shown in Figure 7(b).
2) **eliminate:** the *eliminate i* commands removes all nodes with a fan-out lower or equal to *i*. The elimination is performed by collapsing the selected nodes into their fan-outs. For example, when *eliminate* 2 command is invoked on the netlist of Figure 7(b), node f_3 which has a value of 2 is collapsed into its fan-outs: f_6 and f_7. The resultant netlist is shown in Figure 7(c). Using higher order or eliminations, it is possible to merge low fan-in nodes to create high fan-in nodes.
3) **simplify:** the *simplify* command minimizes the SOP expression for each node in the netlist using a subset of the implicit don't cares. The command essentially invokes the two-level logic minimizer ESPRESSO [49] to minimize the SOP expressions. The SOP minimization aims to find a set of product terms that covers the function f and also minimizes $\sum_{(p\in P)} Cost(p)$ where P is the SOP and any p_i is a product term of the SOP [50]. For example, node f_5 in Figure 7(c) can be minimized as follows: $f_5 = a + f_2 = a + ab = a(1+b) = a$. The simplified f_5 node is shown in Figure 7(d).

The resulting MLL netlist is next forwarded to the technology dependent optimization step.

B. Technology Dependent Optimization

The objective of the technology dependent optimization step is to modify the netlist to ease the utilization of high fan-in OR-gates, which can cost-effectively be realized using OR-plane logic. The netlist from the technology independent optimization step has many nodes with low fan-in because the synthesis scripts were developed for CMOS technology.

An overview of the proposed technology dependent optimization is shown in Algorithm 1. In principle, it is advantageous to collapse MLL nodes with low fan-out into larger nodes. Larger MLL nodes are more likely to utilize high fan-in OR-gates when realized using OR-plane logic. However, the collapsing of MLL nodes reduces the degree of logic sharing, which naturally introduces hardware overheads. The proposed algorithm explores the trade-off between logic sharing and effective utilization of high fan-in OR-gates. The trade-off is explored by replacing the "*eliminate* 0" command with "*eliminate i*" in the default MLL synthesis script [48]. First, an initial synthesis is performed with i equal to 0 and the hardware cost C_i is estimated. The details of the hardware cost estimation using the function $HardwareCost()$ is provided below. Next, i is incremented to $(i+1)$ and the synthesis process and hardware cost estimation is repeated. The process is continued if the estimated hardware cost is reduced compared with in the previous iteration. Otherwise, the algorithm is terminated and the netlist with the minimum estimated hardware cost is returned.

Algorithm 1: Technology Dependent Optimization

```
Inputs: Initial netlist $n$;
Output: Optimized netlist $n_{opt}$;
main {
i ← 0; $n_{i-1} \leftarrow \phi$;
$C_{i-1} \leftarrow \infty$;
$n_i$ ← Perform eliminate i $n$
$C_i$ ← HardwareCost($n_i$);
while $C_i \leq C_{i-1}$ do
    i ← i+1; $C_{i-1} \leftarrow C_i$; $n_{i-1} \leftarrow n_i$;
    $n_i$ ← Perform eliminate i on n;
    $C_i$ ← HardwareCost($n_i$);
end
$n_{opt} \leftarrow n_{i-1}$; return $n_{opt}$;
}
```

The estimated hardware cost C_i of a netlist n_i is computed using a function $HardwareCost()$. The hardware cost estimation is performed by estimating the total crossbar area needed realize the netlist. The area is estimated by organizing the MLL netlist into levels and mapping each level into two series connected crossbars. The level of a node i is equal to the maximum number of nodes that are required to be traversed to reach node i from a primary input. Level l

consists of all nodes assigned to level l. For each level l, the dimensions of the two series connected crossbars are $X_l \times Y_l$ and $Y_l \times Z_l$, respectively. The dimensions X_l, Y_l, and Z_l are computed, as follows:

$$X_l = 2 \times \sum_{j=1}^{N_{l-1}} \#\textit{fan-out}(node_{(l-1)j}) \tag{1}$$

$$Y_l = \sum_{j=1}^{N_l} \#product\_terms(node_{lj}) \tag{2}$$

$$Z_l = \sum_{j=1}^{N_l} \#sum\_terms(node_{lj}) \tag{3}$$

Here, N_{l-1} and N_l are the total number of nodes in level $l-1$ and l of the MLL respectively. $node_{lj}$ represents the j^{th} node in level l. Next, the hardware cost C_i is now calculated as follows:

$$C_i = \sum_{l=1}^{L} ceil(\tfrac{X_l \times Y_l}{r \times c}) + ceil(\tfrac{Y_l \times Z_l}{r \times c}) \tag{4}$$

where $r \times c$ is the architecture constrained dimension of a crossbar and L is the maximum number of levels in the MLL netlist.

At the end of the technology dependent optimization, the technology optimized netlist is forwarded to the crossbar mapping step.

C. Crossbar mapping

In this section, we bind the MLL netlist from the previous step to the crossbar hardware. We first propose a method to map a MLL node to two series-connected crossbars. Next, we explain how to map the DAG of a MLL netlist to staircase structure.

We first note that each node in a MLL netlist represents a two-level logic or a SOP expression. Any SOP expression can be mapped to OR-plane logic using two series-connected crossbars. For instance, consider the following two Boolean functions,

$$f_1 = x_0 + x_1\overline{x_2}$$

$$f_2 = \overline{x_0} + x_0\overline{x_1}x_2$$

where, x_0, x_1, x_2 and their complemented forms are the literals of f_1 and f_2. Using De Morgan's laws [51], f_1 and f_2 can be re-written as,

$$f_1 = x_0 + (\overline{\overline{x_1} + x_2})$$

$$f_2 = \overline{x_0} + (\overline{\overline{x_0} + x_1 + \overline{x_2}})$$

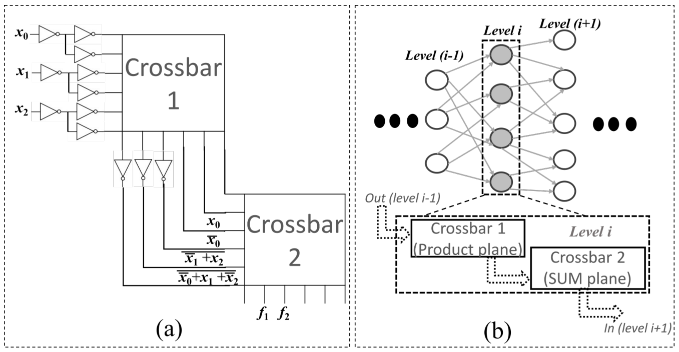


Fig. 8: (a) Two-level logic mapping and (b) MLL mapping to OR-plane logic.

The current expressions of f_1 and f_2 can be mapped to a OR-plane logic based framework using two series-connected crossbars as shown in Figure 8(a). In the figure, the first crossbar evaluates the product terms of the SOP of each function. The second crossbar evaluates the sum of the product terms of each function. The function literals are received as inputs in the first crossbar. The functions are then realized as the output of the second crossbar.

Now, we introduce a method for mapping a MLL netlist to a staircase structure. First, the DAG of a MLL netlist is organized into levels. This is achieved by assigning each node of the DAG a level number based on the maximum number of nodes that have to be traversed to reach that node. Next, similarly to the STREAM-O synthesis, dummy nodes are inserted to align all the inputs of a level to its immediate previous level. Note that while a dummy node in STREAM-O synthesis represents a single buffer, a dummy node in STREAM-M represents two series-connected buffers. After the dummy node insertion, the DAG is now ready for hardware mapping. A simplistic representation of the DAG of a MLL netlist is presented on the top of Figure 8(b). The hardware mapping of a level i of MLL into staircase structure is illustrated at the bottom of Figure 8(b). Each level of the MLL consists of several nodes. Each of the nodes can be mapped into two series-connected hardware as shown in Figure 8(a). Therefore, all of the nodes of level l can be mapped into OR-plane logic using two series-connected crossbars. The first crossbar for level l receives input signals from all the nodes of level $l-1$ and evaluates the product terms of all the nodes in level l. The second crossbar sums the products of all the nodes of level l and propagates signals to the logic level $l+1$.

VI. STREAM Framework for MVM Applications

In this section, we leverage the STREAM framework to accelerate data-intensive applications that are dominated by high precision matrix-vector multiplication (MVM). This requires the computation to be broken into parts such that each part is mapped to a PE with specified hardware resources. This is Problem II outlined in Section III.

The motivation for breaking the computation into parts is that the hardware requirements would otherwise be unacceptably high. While it is desired to synthesize a complete MVM operation into a single netlist, our experiments show that for traditional synthesis tools the netlist tends to explode in size for high-precision MVM operations. For example, mapping a 128x128 matrix with 32 bit precision to a PE requires a staircase structure with 74.5 million crossbars. The crossbar with the largest required dimension would have 40,500 wordlines and 40,000 bitlines. In STREAM, we propose to reduce the hardware requirements using spatial and bit-wise partitioning.

A. Spatial Partitioning

In this section we propose to partition the matrix vector multiplication using blocks with dynamic size, which is illustrated in Figure 9.

Many matrices within scientific computing applications are sparse, one of which is shown in Figure 9(a). The workload of a matrix block is largely dependent on the number of non-zero matrix elements. Therefore, it is easy to understand that partitioning using a fixed block size, which is shown in Figure 9(b), results in that some PEs are heavily under-utilized. Instead, we propose to utilize a dynamic partitioning scheme that uses blocks of dynamic size, which is shown in Figure 9(c).

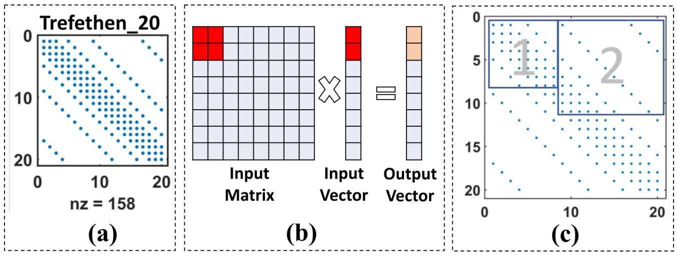


Fig. 9: (a) Sparse matrix of Trefethen-20 benchmark, (b) partitioning with fixed block size, and (c) partitioning with dynamic block size.

The blocking problem has been explored in previous works [17, 52]. However, there is no efficient performance-optimal blocking algorithm available. We present a dynamic blocking algorithm in Algorithm 2. In our implementation, we dynamically expand the block size column-wise and row-wise until a threshold of non-zero elements have been covered.

The inputs to the blocking algorithm is the system matrix M and a block density threshold d. The density threshold parameter d is dictated by the PE architecture, i.e., how many non-zero computations the PE resources can accommodate. To find a potential block, the algorithm traverses through the system. This traversal is very efficient for a sparse matrix. Traditional data structures store a sparse matrix as a set of three vectors: a row vector for row indices of non-zero elements, a column vector for the column indices of non-zero elements and a values vector for the corresponding non-zero values. We define this vectors as R, C and Z respectively. We use these vectors to navigate the original matrix only in its non-zero operand locations. When the algorithm selects a non-zero element of index i, it explores an initial block space starting from the location of i. For simplicity, we select a square block with dimension $k \times k$. If the selected block space meets the density threshold requirement d, the algorithm extracts the block from the original matrix and moves onto the next remaining non-zero operand. If the density threshold is not met, the algorithm expands the block dimension column-wise and row-wise to $(k+1) \times (k+1)$ and continues the process. At the end of the block extraction, the extracted blocks are forwarded to the logic synthesis step to generate in-memory computing kernel.

Algorithm 2: Spatial Partitioning of Sparse Matrix

```
Inputs: Matrix, $M$; block density threshold, $d$
Output: Spatially partitioned matrix blocks
$[R, C, Z] \leftarrow$ find(M); \\$R$, $C$ and $Z$ are vectors of row
  index, column index and value of non-zero elements
$n_z \leftarrow$ length(V); \\number of non-zeros
while $i \neq n_z$ do
    \\Start block extraction
    select a matrix block starting from $i^{th}$ nonzero;
    calculate the density of the selected block;
    if (block density$\geq$ threshold) then
        extract current block from M;
        point i to the next non-extracted non-zero;
    else
        increment block size;
    end
end
```

B. Bit-wise Partitioning

In this section, we propose to utilize bit-slicing to partition the computation across multiple time steps.

The concept of bit-slicing for a 32-bit fixed point multiplication is shown in Figure 10. We aim to decompose the 32-bit element-wise multiplication into a series of multiplications with smaller bit-widths, as shown in Figure 10-(i). The key idea is to bit-slice the input vector with unknown operands, as shown in Figure 10-(ii). The figure shows the bit-slicing of the unknown operand at bit-width of 16 bits. In practice, the bit-slicing-width will be driven by architectural constraints. After bit-slicing, the overall multiplication is performed in a series of multiplications and shift&add operations, as demonstrated in Figure 10-(iii). All in all, the bit-slicing introduces a trade-off between time steps (latency) and hardware utilization.

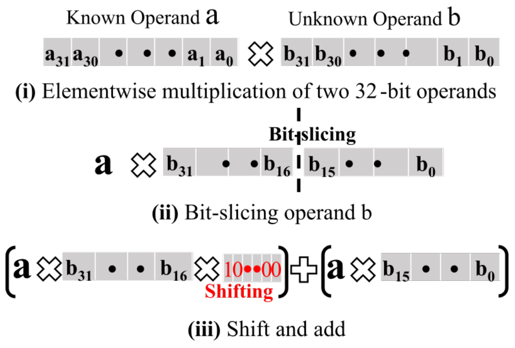

Known Operand a
Unknown Operand b
$a_{31} a_{30}$ • • • a_1 a_0 $b_{31} b_{30}$ • • • b_1 b_0
(i) Elementwise multiplication of two 32-bit operands
Bit slicing
a b_{31} • • b_{16} b_{15} • • b_0
(ii) Bit-slicing operand b
a b_{31} • • b_{16} 10••00 a b_{15} • • b_0
Shifting
(iii) Shift and add

Fig. 10: Bit-slicing element-wise multiplication

VII. PE Architecture

In this section, we discuss the overall architecture of the STREAM PEs. First, we discuss the components of the architecture and later we discuss the parallelism and pipelining for performing MVM operations using the PEs.

Overview: The overview of the STREAM PE architecture is demonstrated in Figure 11(a). The PE consists of a collection of staircase-structured crossbar units. The detailed architecture of a staircase unit is demonstrated in Figure 11(b). The crossbars in the staircase are of dimension 128×128. These crossbars are reconfigurable, meaning they can be programmed to perform arbitrary Boolean functions. The crossbar peripherals are equipped with input register (IR) units and an output

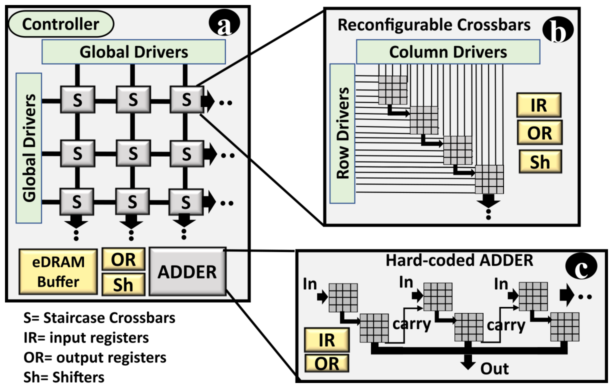


Fig. 11: STREAM PE architecture.

register (IR) units to communicate inputs and output operands respectively. Shifters (Sh) are used in case of bit-slicing of computation as discussed in Section VI-B.

Each PE is equipped with a general purpose ADDER to accumulate the outputs of element-wise multiplication of MVMs. The ADDER is also realized using a staircase structure with the exception that some outputs are accumulated from every second crossbar of the staircase. The detailed architecture of the ADDER is shown in Figure 11(c). For the addition of two $N-$bit numbers, the first pair of crossbars in the staircase perform addition of the X LSBs of the two numbers and the carry-in bit where $X << N$. The next pair of crossbars performs the addition of the $X+1$ to $2X$ bits of the two numbers and the carry-out bit of the previous adder. Similarly, the i^{th} pair of crossbars compute the sum of the $(i-1)X+1$ to iX bits of the two numbers and the carry-out of the $(i-1)^{th}$ pair of crossbars.

The PE also contains a central eDRAM buffer and output register (OR) unit to store the input, the output and the intermediate operands of the PE unit. During the one-time initialization stage, the staircase units are programmed in a pipelined fashion. This helps to keep the eDRAM buffer overhead low. The overall STREAM architecture consists of several tiles of PEs. If the resource requirement of a target problem exceeds a PE resource, the problem space is partitioned and assigned into multiple PE tiles. The PE components utilize both parallelism and pipelining to maximize performance.

Parallelism: The matrix-vector multiplication is highly parallelizable. Each matrix-row $\times$ input-vector operation is independent of each other. As discussed in Section VI, we adopt a partitioned MVM approach in the STREAM framework. All the partitioned segments of the matrix can be evaluated in separate staircase crossbars in parallel. This high order of parallelism speeds up the computation in manifold. The time complexity of the complete MVM is reduced to the time complexity of the segmented MVM.

Pipelining: It is noteworthy that the ADDER in the PE is general purpose. This means that all the element-wise multiplication results can share the adder. This sharing of adder significantly reduces the hardware requirement. The outputs of each element-wise multiplication is therefore pipelined to the shared adder for accumulation of results to generate a row-vector$\times$column-vector multiplication of MVM.

TABLE III: Area-Power Cost of STREAM Components

Component	**Parameter**	**Specs**	**Area**	**Power**
Crossbar	dimension	128×128	25 μm^2	0.3 mW
Controller	# unit	1	6.27 μm^2	0.01 mW
Shifter	# unit	1	60 μm^2	0.05 mW
IR	size	1 B	1.05 μm^2	0.0006 mW
OR	size	1 B	2.93 μm^2	0.001 mW
local bus	#wires	128	0.03 mm^2	2.33 mW
Staircase Unit	# crossbars	10	**0.0003 mm^2**	**3.06 mW**
	#IR/#OR	1/5		
	Shifter	1		
ADDER	# size	9 B	**0.0012 mm^2**	**14.42 mW**
	# crossbars	48		
	#IR/#OR	18/9		
eDRAM Buffer	size	20 KB	0.026 mm^2	6.47 mW
PE	# ADDER	1	**0.0657 mm^2**	**412.63 mW**
	# Staircase Unit	128		
	#OR/ Shifter	9/1		
Bus	bandwidth	128-bits	15.7 mm^2	13 mW

VIII. Experimental Evaluation

In this section, we experimentally evaluate the performance of the STREAM framework based on STREAM-O and STREAM-M synthesis approaches. The experiments are executed on a machine with an Intel Core i9 processor and NVIDIA GeForce RTX 2070S GPU. A SPICE-level simulation is performed with the fitting characteristics of the VTEAM model [53]. The crossbar dimensions are selected to be 128×128, and the resistance R_{LRS} and R_{HRS} of the memristors are selected as $10k\Omega$ and $10M\Omega$ respectively. We use a read and write latency of 29.31ns and 50.88ns respectively, as reported in [35]. In Table III, the remaining parameters for the different architectural components of the STREAM PEs are provided. For each component, we provide the specifications, area, and power consumption. The parameters have been obtained from [53–55]. To perform a comparison with the CMOS ASIC-like systems, we utilize the Synopsys Design Compiler (DC) tool with *gscl*-45nm technology library [56]. We use the CACTI 7 tool [57] to simulate the data transfer cost for 1GB DDR3 memory on 45nm technology node.

For the experimental evaluation, we utilize the benchmarks from the ISCAS85 benchmark suite [37], and matrices from the Suit-Sparse matrix collection of [38]. The overview of the function type and properties (number of inputs and outputs) of the ISCAS85 benchmark suit are listed in Table IV. The ISCAS85 benchmark suit consists of ten combinational network circuits. An overview of the matrices selected from the Suit-Sparse matrix collection is provided in Table V. We select eleven sparse matrices with different sparsity patterns.

TABLE IV: Overview of ten ISCAS85 benchmarks.

Benchmark	Function	Inputs	Outputs
c432	Priority Decoder	36	7
c499	ECAT	41	32
c880	ALU and control	60	126
c1355	ECAT	41	32
c1908	ECAT	33	25
c2670	ALU and control	233	140
c3540	ALU and control	50	22
c5315	ALU and selector	178	123
c6288	16-bit multiplier	32	32
c7552	ALU and control	207	108

TABLE V: Overview of eleven matrices of the SuitSparse Matrix Collection in terms of application type, matrix dimensions, and number of non-zero elements.

Applications	Systems	Matrix Dimensions	#Non-zeros
Trefethen-20	Combinatorial	20×20	158
mesh3em5	Structural	289×289	1377
Trefethen-150	Combinatorial	150×150	2040
Trefethen-200b	Combinatorial	199×199	2873
Trefethen-200	Combinatorial	200×200	2890
bcsstk02	Structural	66×66	4356
Trefethen-300	Combinatorial	300×300	4678
Chem97ZtZ	Statistical/Mathematical	2541×2541	7361
Trefethen-500	Combinatorial	500×500	8478
Journals	Undirected Weighted Graph	124×124	12068
Trefethen-700	Combinatorial	700×700	12654

The matrices are from various domains of physical systems such as statistics, mathematics and structural engineering.

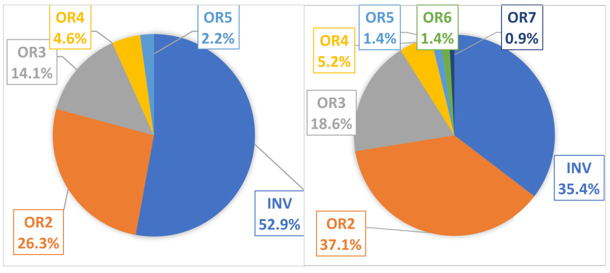
In Section VIII-A, we evaluate the effectiveness of the proposed synthesis flows. We evaluate the architectural design in Section VIII-B. In Section VIII-C, we compare the performance of the two synthesis approaches with the state-of-the-art computing paradigms.

A. *Evaluation of Synthesis Approaches*

In this section, we evaluate the effectiveness of the STREAM-O and STREAM-M synthesis flows. For the evaluation, we use benchmarks from the ISCAS85 benchmarks suite [37].

1) STREAM-O: In this section, we evaluate the effectiveness of the SREAM-O synthesis flow. In Figure 12, we show the distribution of OR-INV gates before and after technology dependent optimization on the $c880$ circuit. We observe that the maximum fan-in within the initial netlist is five. On the other hand, the maximum fan-in is seven after technology dependent optimization. It can also be observed that the percentage of OR-gates with higher fan-in has been increased by the optimization. The OR-gates with high fan-in are created by merging OR-gates with smaller fan-in.

The total number of gates after technology independent optimization, after technology dependent optimization and, after crossbar mapping on the ISCAS85 circuits is shown in Figure 13. The figure shows that the technology dependent optimization reduces the total number of gates with 30% on average. However, in the crossbar mapping step, the total number of gates are increased by $2.5X$. This is due to that dummy-nodes are inserted to connect OR-gates that are not placed in adjacent crossbars. Nevertheless, the dummy nodes allows the crossbar inter-connections to be hardwired avoiding the use of expensive reconfigurable interconnects.



(a) (b)

Fig. 12: Gate distribution of the netlist of $c880$ benchmark (a) before and, (b) after technology dependent optimization step of STREAM-O.

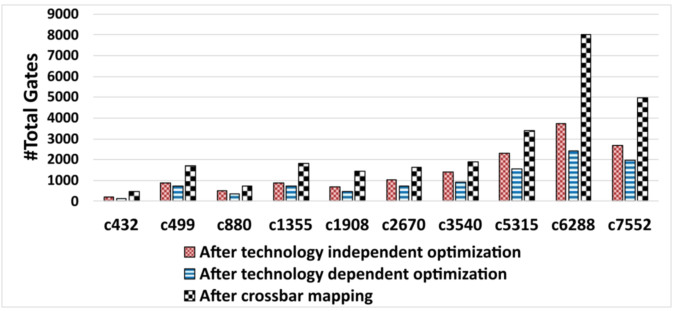
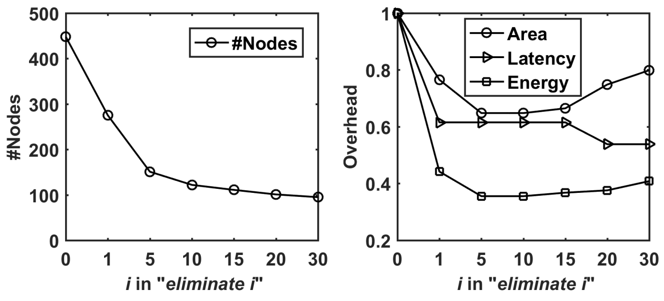


Fig. 13: Comparison of total number of gates in ISCAS85 benchmark circuits after technology independent optimization, technology dependent optimization and crossbar mapping steps of STREAM-O.

2) STREAM-M: In this section, we evaluate the technology dependent optimization within the STREAM-M synthesis flow. The total number of nodes on the $c1355$ circuit with respect to the "*elimination i*" operation is shown in Figure 14(a). As expected, the figure shows that the total number of nodes is reduced when more nodes are collapsed. For example, it can be observed that the eliminate 30 operation reduces the number of nodes from 448 to 95. In Figure 14(b), we show the area, latency, and energy with respect to the eliminate operation. When i is increased, it can be observed that the area is decreased until a turning-point from where it starts to increase again. This stems from the trade-off between logic share and used of OR-gates with high fan-in. We select to minimize the area in Algorithm 1 because it results in both small energy and area. The delay is improved with respect to if no optimization was performed. However, it could be improved further if it was set as the primary objective.

The total number of gates after technology independent optimization, after technology dependent optimization, and after crossbar mapping on the ISCAS85 circuits is shown in Figure 15. The figure shows that the technology dependent optimization reduces the total number of gates with 57% on average. The crossbar mapping increases the total number of



(a) (b)

Fig. 14: Analysis of elimination on c1355 circuit. (a) Number of multi-level nodes vs. value of i in *eliminate i*. (b) Area-Latency-Energy overhead vs. value of i in *eliminate i*.

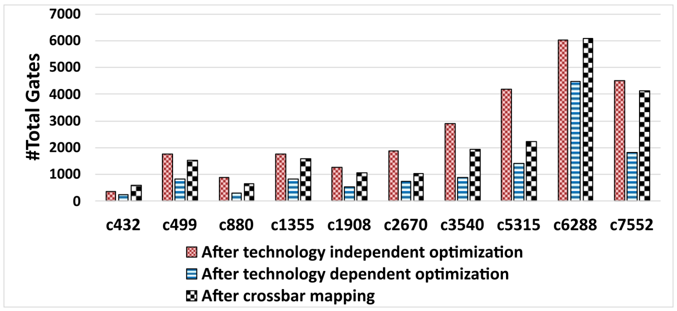


Fig. 15: Comparison of total number of gates in ISCAS85 benchmark circuits after technology independent optimization, technology dependent optimization and crossbar mapping steps of STREAM-M.

gates with 80% by inserting dummy nodes. In similar to within STREAM-O, the dummy nodes allow the crossbars within the staircase structure to be hardwired, i.e., avoiding the use of costly reconfigurable interconnects.

B. Evaluation of Architectural Design

In this section we evaluate the proposed architectural design. In particular, we calculate the maximum resolution of an adder that can be fit into a pair of series connected crossbars of architecture specified dimension.

Figure 16(a) shows the normalized area overhead for per bit addition using adders with different bit resolutions. The figure shows that a 3-bit adder yields minimum area per bit addition. This is due to that the proposed architecture specifies a crossbar dimension constraint of 128×128. And a 3-bit adder is the largest full adder that can be realized using a pair of 128×128 crossbars. The required maximum crossbar dimension for adders of different bit resolutions is shown in Figure 16(b). The figure shows that the maximum crossbar dimension scale up rapidly for adders with higher bit resolutions.

C. Comparison with State-of-the-art

In this section, we first compare STREAM with state-of-the-art in-memory computing paradigms. In Section VIII-C1, we compare the performance on the ISCAS benchmarks. In Section VIII-C2, we evaluate the STREAM framework with data-intensive MVM operations on the applications from the Suit-Sparse matrix collection. In Section VIII-C3, we present a comparison with CMOS ASIC-like systems.

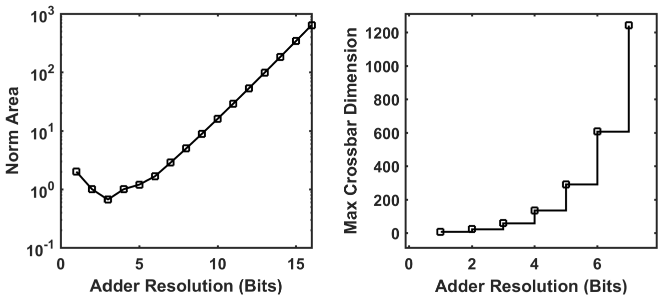


Fig. 16: Evaluation of architecture constraint adder resolution. (a) Normalized area overhead for per bit computation using adders of different bit resolution and, (b) maximum crossbar dimensions for adders of different bit resolution

1) Evaluation on ISCAS85 Benchmarks: In this section, we compare the state-of-the-art WRITE-based in-memory computing scheme CONTRA [29] with STREAM-O and STREAM-M. We evaluate the area, latency, and energy consumption on the ISCAS85 benchmarks listed in Table IV.

The comparative result of the evaluation on the ISCAS85 benchmarks are listed in Table VI. The results show that CONTRA uses $5.3X$ and $5.0X$ less area than STREAM-O and STREAM-M, respectively. This is because CONTRA is a WRITE-based in-memory computing paradigm that allows non-volatile memory cells to be reused to minimize area. On the other hand, the STREAM frameworks are based on READ-based in-memory computing and are expected to improve delay and energy. Compared with CONTRA, STREAM-O improves the average latency and energy with $143.3X$ and $1.6X$, respectively. This is a result of that no WRITE operations are performed within READ-based in-memory computing. The READ-based computing scheme also eliminates many expensive copy operations that are used within CONTRA [29]. Compared with STREAM-O, STREAM-M improves latency and energy with $1.4X$ and $1.6X$, respectively. The improvements stem from that the MLL data structure empirically results more compact hardware representations than using OIGs. This is not surprising considering the total gate count in Figure 13 and Figure 15. Given these results, we conclude that it is better to use MLL than OIGs when mapping Boolean logic to in-memory compute kernels for OR-plane logic.

2) Evaluation with SuitSparse Matrix Collection: In this section, we evaluate the performance of the STREAM framework for matrix-vector-multiplication (MVM) operation. MVM is the most expensive computational kernel in many scientific applications. In particular, MVM is the dominating computation in platform for solving systems of linear equations. The state-of-the-art solution for solving systems of sparse matrices is based on iterative Kyrlov subspace methods such as conjugate gradient (CG) [58] and GMRES [59]. These methods are based on iteratively refining an initial solution through error corrections. In each iteration of the iterative refinement algorithm, an expensive MVM operation is performed. An overview of 11 positive-definite [60] matrices are shown in Table V. We use a 32-*bit* representation for both the matrix operands and input vector operands to generate the HDL description. For bit-wise partitioning of unknown multiplication operands, we select a bit-slicing width of 4-*bits*. For a given system, we consider the matrix operands as constants and the input/output vectors are considered as variables. For new systems, the PEs are reconfigured for new matrix values.

We compare the performance of the STREAM framework with SIMPLER [30], which the state-of-the-art in-memory computing paradigm for MVM. We chose SIMPLER for this evaluation because SIMPLER outperforms CONTRA for arithmetic operations. SIMPLER is based on MAGIC and it is ideal for applications with high order of parallel computation. SIMPLER can perform each matrix-row$\times$input-vector operation in parallel using a row-mapping fashion, which greatly

TABLE VI: Comparison of area, latency, and energy consumption for CMOS technology, CONTRA, STREAM-O and STREAM-M on ten benchmarks of the ISCAS85 benchmarks suite.

	CONTRA [29]			STREAM-O			STREAM-M		
Benchmark	Area (μm^2)	Latency (μs)	Energy (nJ)	Area (μm^2)	Latency (μs)	Energy (nJ)	Area (μm^2)	Latency (μs)	Energy (nJ)
c432	447.25	39.18	56.81	1418.50	0.64	41.82	1141.00	0.47	21.60
c499	447.25	68.33	99.08	1742.20	0.73	63.57	1649.80	0.41	33.03
c880	447.25	64.26	93.18	1742.20	0.85	73.75	1048.50	0.41	16.34
c1355	447.25	68.38	99.15	1511.00	0.59	41.68	1649.80	0.53	42.47
c1908	447.25	74.74	108.37	1649.80	0.79	63.71	1326.00	0.53	30.91
c2670	447.25	104.81	151.97	1696.00	0.62	51.48	1834.80	0.35	32.72
c3540	447.25	181.89	263.74	2574.80	1.35	192.94	2066.00	0.70	76.44
c5315	447.25	245.80	356.41	3314.80	0.97	186.85	2713.50	0.47	71.51
c6288	447.25	401.00	581.45	8494.80	3.31	1800.88	6968.50	3.52	1549.15
c7552	447.25	356.00	516.20	4979.80	1.49	457.20	5904.80	1.00	367.19
Norm. Avg.	**0.190**	1.000	1.000	1.000	0.007	0.640	0.950	**0.005**	**0.400**
Avg. Improvement over CONTRA [29]	**1.0X**	1.0X	1.0X	0.2X	143.3X	1.6X	0.2X	**200.0X**	**2.5X**

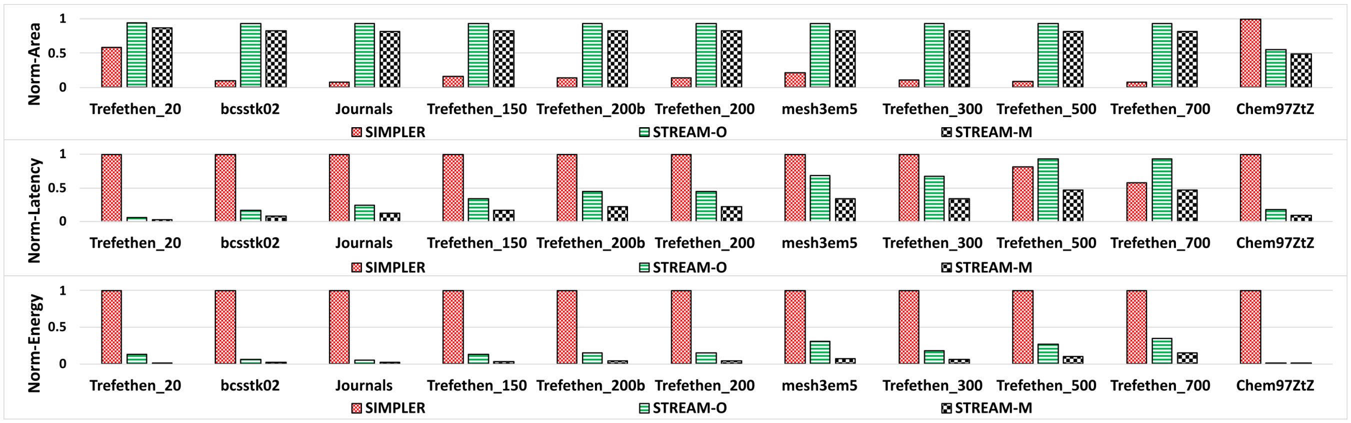


Fig. 17: Comparison of area, latency, and energy for SIMPLER, STREAM-O and STREAM-M synthesis based STREAM frameworks on eleven benchmarks of the SuitSparse Matrix Collection.

improves the performance of the accelerator.

In Figure 17, we compare the performance of STREAM framework with SIMPLER in terms of area, latency and energy consumption. The experimental result shows that the STREAM-O and STREAM-M based frameworks require on average $3.7X$ and $3.3X$ more area usage respectively than SIMPLER. However, STREAM-O achieves $6.2X$ energy efficiency, and improves the latency by $2X$ compared with the latency of SIMPLER. Compared with STREAM-O, STREAM-M improves energy and latency with $3.2X$ and $2.1X$, respectively.

Note that the improvements achieved for MVM applications are relatively less than that for the ISCAS85 benchmarks. This is due to that, MVM is a computationally expensive task. Therefore, it requires multiple PEs to cover the whole computation. This computational partitioning in multiple PEs incurs latency and energy overhead due to the inter-PE data transfer. Nevertheless, the experimental results clearly demonstrate that the proposed STREAM framework is advantageous overall to the state-of-the-art WRITE-based in-memory computing paradigms.

3) Comparison with CMOS ASIC: In this section, we compare the performance of the STREAM framework with CMOS ASIC-like systems.

Our experiments show that the STREAM framework has distinct advantages while evaluating arbitrary Boolean logic and MVM operations. For Boolean logic operations, the major improvement of the proposed system comes from the area reduction that comes with the reconfigurability of the architecture. For instance, Table VII lists the combinational area requirements to develop the first 8 circuits from the ISCAS-85 benchmark suit. For CMOS ASIC chips, the total area requirement is 16318 μm^2. On the other hand, these circuits can be sequentially reconfigured in the STREAM architecture with a *worst-case* area of 1250 μm^2 which reduces the area requirement by 13X.

For MVM applications, the improvement of the STREAM framework comes from the reduction in cross-architecture data transfer cost. To perform MVM within CMOS ASIC, both the matrix operands and input/output operands need to be

TABLE VII: Combinational area requirements for 8 circuits from the ISCAS-85 benchmark suit.

Benchmark	CMOS ASIC Area (μm^2)	STREAM Worst Case Area (μm^2)
c432	575.4	
c499	726.01	
c880	1059.7	
c1355	1729.84	1250
c1908	1419.2	
c2670	2212.28	
c3540	3348.9	
c5315	5246.3	
Total	**16318**	**1250**
Normalized	**13X**	**1X**

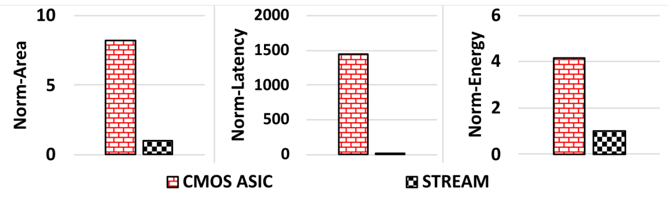


Fig. 18: Comparison of average area, latency, and energy for CMOS ASIC and STREAM frameworks on benchmarks of the SuitSparse Matrix Collection in Table V.

repeatedly accessed using expensive data transfer. Conversely, in the STREAM framework the matrix values are programmed into the memory. Therefore, the data transfer complexity is reduced from $O(N^2)$ to $O(N)$. We evaluate the dynamic memory access latency/energy cost using the CACTI 7 tool and the processor area/latency/energy cost using the Synopsis Design Compiler tool. Our experimental evaluation show that for the sparse matrices in Table V, the STREAM framework reduces the average total area by 8.21X, latency by 1449X, and energy consumption by 4.14X compared to the CMOS ASICs which is shown in Figure 18. Note that such significant speed-up is typical for in-memory computing paradigms [30, 55].

IX. Other Related Works

In this section we discuss prominent in-memory computing paradigms for accelerating data-intensive applications. We perform the review by dividing the different paradigms and systems based on if they target (i) approximate computing applications, (ii) structured high precision workloads, and (iii) irregular high precision workloads. Different paradigms are advantageous for different types of applications and workloads.

Approximate Computing Applications: High density and energy-efficient analog matrix-vector multiplication is advantageous for approximate computing applications such as neural networks, image processing, and search. Prominent studies include Pipelayer [35], ISAAC [54], IMP [61] , Prometheus [62], and AccuRED [63]. The limitation of these paradigms is that they cannot perform Boolean logic or deliver the high precision required by scientific computing applications considered in this paper.

Structured High Precision Workloads: Vector-based digital in-memory computing such as MAGIC [24], IMPLY [23], bit-wise-in-bulk [26] and DRAM-based in-memory computing paradigms can be used to accelerate applications based on high precision dense matrix vector multiplication. These paradigms execute parallel vector-based instructions very efficiently. Prominent studies include SIMPLY [31], FloatPIM [55], Ambit [33], and Compute Cache [64]. However, these paradigms cannot handle Boolean logic or irregular computational patterns such as sparse matrix-vector multiplication.

Irregular High Precsision Workloads: Irregular in-memory approaches such as OR-plane logic and some MAGIC and IMPLY logic are advantageous for unstructured applications such as combinational logic and sparse MVM. Important contributions and studies include CONTRA [29], SIMPLER [30], and STREAM [36]. For combinational logic circuits, STREAM outperforms CONTRA due to the use of READ-based in-memory computing instead of WRITE-based in-memory computing. For sparse matrix-vector multiplication, the STREAM framework outperforms SIMPLER due to the ability of considering the sparsity of the matrix.

X. Summary and Future Work

In this paper, we proposed the STREAM framework for executing Boolean logic using nanoscale crossbars. The framework consists of a synthesis tool that decomposes high-level application into READ-based in-memory compute kernels. We develop two synthesis approaches based on contemporary data structures from logic synthesis. We also design an in-memory computing architecture and techniques to decompose the desired computation with respect to the architectural constraints. The experimental evaluation illustrates the effectiveness of READ-based computing compared with WRITE-based in-memory computing paradigms. In the future, we plan to use STREAM to accelerate genome sequencing applications. We also plan to augment STREAM with analog in-memory accelerators.

References

[1] K. Rose, S. Eldridge, and L. Chapin, "The internet of things: An overview," *The internet society (ISOC)*, vol. 80, pp. 1–50, 2015.
[2] R. Taylor, D. Baron, and D. Schmidt, "The world in 2025-predictions for the next ten years," in *IMPACT*, pp. 192–195, IEEE, 2015.
[3] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future*, vol. 2007, no. 2012, pp. 1–16, 2012.
[4] D. Quick and K.-K. R. Choo, "Impacts of increasing volume of digital forensic data: A survey and future research challenges," *Digital Investigation*, vol. 11, no. 4, pp. 273–294, 2014.
[5] S. Chai, "Mobile challenges for embedded computer vision," in *Embedded Computer Vision*, pp. 219–235, Springer, 2009.
[6] F. Pop, "High performance numerical computing for high energy physics: A new challenge for big data science," *Advances in High Energy Physics*, vol. 2014, pp. 1–13, 02 2014.
[7] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *CACM*, vol. 21, no. 8, pp. 613–641, 1978.
[8] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH*, vol. 23, no. 1, pp. 20–24, 1995.
[9] A. A. Sawchuk and T. C. Strand, "Digital optical computing," *Proceedings of the IEEE*, vol. 72, no. 7, pp. 758–779, 1984.
[10] A. Steane, "Quantum computing," *Reports on Progress in Physics*, vol. 61, no. 2, p. 117, 1998.
[11] G. Singh *et al.*, "A review of near-memory computing architectures: Opportunities and challenges," in *2018 21st DSD*, pp. 608–617, IEEE, 2018.
[12] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature electronics*, vol. 1, no. 6, pp. 333–343, 2018.
[13] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, no. 5, pp. 507–519, 1971.
[14] G. W. Burr *et al.*, "Phase change memory technology," *JVSTB*, vol. 28, no. 2, pp. 223–262, 2010.
[15] Y. Huai *et al.*, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.
[16] C. Li *et al.*, "Analogue signal and image processing with large memristor crossbars," *Nature Electronics*, vol. 1, no. 1, p. 52, 2018.
[17] B. Feinberg *et al.*, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th ISCA*, pp. 367–382, IEEE, 2018.
[18] B. Zhang *et al.*, "Computational restructuring: Rethinking image compression using resistive crossbar arrays," *IEEE TCAD*, vol. 40, no. 5, pp. 836–849, 2020.
[19] B. Zhang *et al.*, "Towards resilient deployment of in-memory neural networks with high throughput," in *2021 58th ACM/IEEE DAC*, pp. 1081–1086, IEEE, 2021.
[20] N. Uysal, B. Zhang, S. K. Jha, and R. Ewetz, "Xmap: Programming memristor crossbars for analog matrix–vector multiplication: Toward high precision using representable matrices," *IEEE TCAD*, vol. 41, no. 6, pp. 1827–1841, 2021.
[21] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st HPCA*, pp. 476–488, IEEE, 2015.
[22] M. Le Gallo *et al.*, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, no. 4, pp. 246–253, 2018.

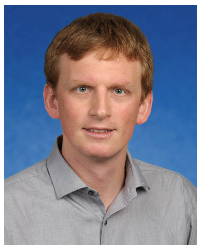
[23] J. Borghetti *et al.*, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
[24] S. Kvatinsky *et al.*, "Magic—memristor-aided logic," *TCAS-II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.
[25] S. K. Jha *et al.*, "Computation of boolean formulas using sneak paths in crossbar computing," 2016. US Patent 9,319,047.
[26] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, pp. 1–6, IEEE, 2016.
[27] S. Thijssen, S. K. Jha, and R. Ewetz, "Path: Evaluation of boolean logic using path-based in-memory computing," in *59th DAC*, 2022.
[28] A. Dehon, "Nanowire-based programmable architectures," *JETC*, vol. 1, no. 2, pp. 109–162, 2005.
[29] D. Bhattacharjee *et al.*, "Contra: area-constrained technology mapping framework for memristive memory processing unit," in *ICCAD*, pp. 1–9, 2020.
[30] R. Ben-Hur *et al.*, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *TCAD*, vol. 39, no. 10, pp. 2434–2447, 2019.
[31] T. Zanotti, F. M. Puglisi, and P. Pavan, "Smart logic-in-memory architecture for low-power non-von neumann computing," *IEEE Journal of the Electron Devices Society*, vol. 8, pp. 757–764, 2020.
[32] S. Thijssen *et al.*, "Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter," in *2021 DATE*, pp. 232–237, IEEE, 2021.
[33] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM MICRO*, pp. 273–287, IEEE, 2017.
[34] M. H. I. Chowdhuryy *et al.*, "Ladder: Architecting content and location-aware writes for crossbar resistive memories," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 117–130, 2021.
[35] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, pp. 541–552, IEEE, 2017.
[36] M. R. H. Rashed, S. Thijssen, S. K. Jha, F. Yao, and R. Ewetz, "Stream: Towards read-based in-memory computing for streaming based data processing," in *2022 27th ASP-DAC*, pp. 690–695, IEEE, 2022.
[37] M. C. Hansen *et al.*, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.
[38] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, vol. 38, no. 1, pp. 1–25, 2011.
[39] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE TCAD*, vol. 21, no. 12, pp. 1377–1394, 2002.
[40] A. Deb, R. Wille, and R. Drechsler, "Or-inverter graphs for the synthesis of optical circuits," in *2017 IEEE 47th ISMVL*, pp. 278–283, IEEE, 2017.
[41] R. K. Brayton *et al.*, "Mis: A multiple-level logic optimization system," *IEEE TCAD*, vol. 6, no. 6, pp. 1062–1081, 1987.
[42] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.
[43] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *27th ACM/IEEE DAC*, pp. 40–45, IEEE, 1990.
[44] K. A. Bartlett *et al.*, "Multi-level logic minimization using implicit don't cares," *IEEE TCAD*, vol. 7, no. 6, pp. 723–740, 1988.
[45] H. Wu *et al.*, "Resistive random access memory for future information processing system," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1770–1789, 2017.
[46] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification." "http://www.eecs.berkeley.edu/alanmi/abc".
[47] K. Keutzer, "Dagon: Technology binding and local optimization by dag matching," in *DAC*, pp. 341–347, 1987.
[48] E. M. Sentovich *et al.*, "Sis: A system for sequential circuit synthesis," 1992.
[49] R. K. Brayton *et al.*, *Logic minimization algorithms for VLSI synthesis*, vol. 2. Springer Science & Business Media, 1984.
[50] O. Coudert, "Doing two-level logic minimization 100 times faster.," in *SODA*, pp. 112–121, 1995.
[51] P. J. Hurley, *A concise introduction to logic*. Cengage Learning, 2014.
[52] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*. University of California, Berkeley, 2003.
[53] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *TCAS-II: Express Briefss*, vol. 62, no. 8, pp. 786–790, 2015.
[54] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *SIGARCH*, vol. 44, no. 3, pp. 14–26, 2016.
[55] M. Imani *et al.*, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *ISCA*, pp. 802–815, IEEE, 2019.
[56] J. E. Stine *et al.*, "Freepdk: An open-source variation-aware design kit," in *2007 IEEE MSE*, pp. 173–174, IEEE, 2007.
[57] R. Balasubramonian *et al.*, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, vol. 14, no. 2, pp. 1–25, 2017.
[58] M. R. Hestenes, E. Stiefel, *et al.*, *Methods of conjugate gradients for solving linear systems*, vol. 49. NBS Washington, DC, 1952.
[59] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.
[60] R. Bhatia, *Positive definite matrices*. Princeton university press, 2009.
[61] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 1–14, 2018.
[62] Y. Xiao, S. Nazarian, and P. Bogdan, "Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy," in *2018 DATE*, pp. 1387–1392, IEEE, 2018.
[63] B. K. Joardar *et al.*, "Accured: High accuracy training of cnns on reram/gpu heterogeneous 3-d architecture," *IEEE TCAD*, vol. 40, no. 5, pp. 971–984, 2020.
[64] S. Aga *et al.*, "Compute caches," in *2017 IEEE HPCA*, pp. 481–492, IEEE, 2017.

Muhammad Rashedul Haq Rashed is a Ph.D. candidate in Computer Engineering at the University of Central Florida (UCF). Rashed received his bachelor's degree in Electrical and Electronics Engineering from Bangladesh University of Engineering and Technology (BUET) in 2015. His research interests include EDA for emerging computing paradigms, computer-aided design, and artificial intelligence. He has received a best paper nomination at ICCAD 2022.

Sven Thijssen is a Ph.D. student in Computer Science at the University of Central Florida (UCF). Sven received his bachelor's degree in Informatics from KU Leuven, Belgium, in 2018, and his master's degree in Computer Science from UCF in 2021. His research interests are in-memory computing and beyond von Neumann computing. In 2020 he has received the ORCGS Doctoral Fellowship from UCF and in 2021 he has received a best paper nomination at DATE.

Sumit K. Jha is Professor of Computer Science at the University of Texas San Antonio. Dr. Jha received his Ph.D. in Computer Science from Carnegie Mellon University. He completed B.Tech (Honors) in Computer Science and Engineering from the Indian Institute of Technology Kharagpur. Dr. Jha has worked on R&D problems at Microsoft Research India, GM, INRIA and the Air Force Research Lab Information Directorate. Dr. Jha was awarded the prestigious Air Force Young Investigator Award and his research has led to four Best Paper awards.

Fan Yao (Member, IEEE) received the PhD degree in computer engineering from George Washington University, Washington, DC, in 2018. He is currently an assistant professor with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida. His research interests include computer architecture, hardware security, security of emerging applications and computing infrastructure, energy efficient computing.



Rickard Ewetz received the M.S. degree, in Applied Physics and Electrical Engineering, from Linkopings Universitet in 2011. He received the Ph.D. degree in Electrical and Computer Engineering from Purdue University in 2016. Currently, he is an associate professor in the ECE Department at the University of Central Florida. His research interests include physical design and computer-aided design for in-memory computing using emerging technologies. He has best paper nominations from ASP-DAC 2019, DATE 2021, and ICCAD 2022.