STAR: Synthesis of Stateful Logic in RRAM Targeting High ARea Utilization

Feng Wang[1,*], Guojie Luo[1,#], Guangyu Sun[1], Jiaxi Zhang[1], Jinfeng Kang[2],
Yuhao Wang[3], Dimin Niu[3], Hongzhong Zheng[3]
[1]Center for Energy-efficient Computing and Applications, Peking University, Beijing, China
[2]Institute of Microelectronics, Peking University, Beijing, China
[3]Pingtouge, Alibaba Group
Email: {*yzwangfeng, #gluo}@pku.edu.cn

***Abstract*—Processing-in-memory (PIM) exploits massive parallelism with high energy efficiency and becomes a promising solution to the von Neumann bottleneck. Recently, the emerging metal-oxide Resistive Random Access Memory (RRAM) shows its potential to construct a PIM architecture, because several stateful logic operations, e.g., IMP and NOR, can be executed in an RRAM crossbar in parallel. Previous synthesis flows focus on improving latency with stateful logic operations, but they ignore that the memory should be used primarily for storage. i.e., most of the area in the crossbar is used for computation but not storage. In this situation, storage and computation still have to be separated into different crossbars, which leads to considerable data transfer overhead and limited parallelism.**

In this work, we define the ratio of storage in a crossbar as area utilization. We aim to improve the area utilization without throughput loss by proposing STAR, a novel synthesis flow for the stateful logic. We present two optimization strategies to reduce the computation area in STAR. First, we reduce the area for redundant inputs. For the shared constants among different rows (or columns), we encode them as immediate values into the control signals without writing them into the crossbar at run time. For the other inputs, we only store one copy of them in the crossbar. Second, we reduce the area for intermediate variables by reusing invalid cells. And we design a scheduling algorithm to find a computation sequence with the minimal variable erasing cycles. Invalid primary inputs can also be erased in this algorithm. Furthermore, we present a case study of the image convolution to demonstrate the effectiveness of STAR. Experimental evaluation shows that STAR achieves 33.03% more area utilization and a 1.43x throughput compared to SIMPLER, the state-of-the-art stateful logic synthesis flow. Our image convolution implementation also provides 78.36% more area utilization and a 1.48x throughput compared with IMAGING, the state-of-the-art stateful logic based image processing accelerator.

***Index Terms*—synthesis, memory, scheduling, performance optimization**

I. Introduction

Huge energy-hungry data transfer between processors and memory has been the limitation of computation speed and energy efficiency, i.e., the von Neumann bottleneck [1]. To alleviate the memory wall, researchers have explored to process data within the memory, which is an attractive solution.

This work is partly supported by Beijing Municipal Science and Technology Program under Grant No. Z201100004220007, Key Area R&D Program of Guangdong Province with grant No. 2018B030338001, Beijing Academy of Artificial Intelligence (BAAI), and Alibaba Innovative Research (AIR) Program.

The metal-oxide Resistive Random Access Memory (RRAM) is one type of non-volatile memories (NVM) [2]. It has emerged as one of the most promising technologies to implement a processing-in-memory (PIM) architecture for two reasons. First, it is a non-volatile, ultra-compact memory with low leakage power and excellent scalability. Industrial demonstrations have been presented [3] to showcase the viability of large memory crossbars (GB level in total with 512×512 bits per crossbar). Second, digital RRAM has been demonstrated to perform stateful logic operations [4] beyond storage. It is possible to combine computation and storage in the same RRAM crossbar due to its flexibility.

In a digital RRAM crossbar, each RRAM cell can store one-bit information because it has two different resistance states, the low resistance state (LRS) and the high resistance state (HRS). These two states can be switched by applying certain voltage patterns. For RRAM cells that are connected in series, they can change the states of others under specific voltage patterns. This important feature has been leveraged for computation, and several stateful logic operations have been conducted in recent years, including IMP [4], NOR [5], NAND [6], and OR [7].

Besides the PIM benefit, RRAM also provides massive Single Instruction Multiple Lines (SIML)-fashion parallelism [8]. In an RRAM crossbar, we can implement multiple stateful logic operations along different columns (or rows) in parallel by applying the same voltage pattern [9], if the input and the output RRAM cells are aligned along rows (or columns). The degree of parallelism can reach the size of the crossbar and scale with the data size due to the PIM capability of RRAM. On contrary, the degree of parallelism in the conventional von Neumann architecture is limited by the amount of computing resources, e.g., Arithmetic Logical Units (ALUs). Despite their equivalence in computation capability, we can achieve lower time complexity in RRAM if fully exploiting its parallelism.

To utilize the PIM feature and the SIML-fashion parallelism of RRAM, storage and computation have to be combined in the same RRAM crossbars. Otherwise, data need to be transferred from storage crossbars to computation crossbars at run time, which still introduces a considerable overhead [20]. Also, the parallelism is dominated by the number of computation crossbars and cannot scale with the data size. Furthermore, to meet this requirement and still suit for storage, storage have to take up the major part in the crossbar.

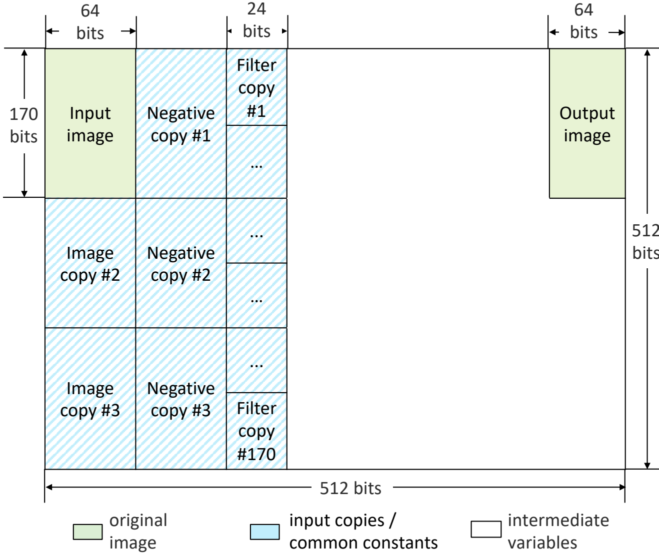


Fig. 1: Data layout during the computation procedure of image convolution in IMAGING. For a 3×3 filter and a 512×512 RRAM crossbar, the maximum input image size is 170×8 with 8 bits per pixel.

TABLE I: Ultra-low area utilization of previous works. We list the result of the state-of-the-art work SIMPLER for each benchmark.

Benchmark	Work	Area utilization
ISCAS'85 [10]	Logic [9]	3.69%
	Scalable [11]	2.52%
	Look-ahead [12]	2.64%
	Staircase [13]	2.01%
	SAID [14]	8.25%
	SIMPLER [15]	63.21%
LgSynth'91 [16]	SIMPLE [17]	8.02%
	SIMPLER	48.49%
IWLS'93 [18]	SIMPLER	41.93%
Image convolution	IMAGING [19]	8.10%

In this work, we define:

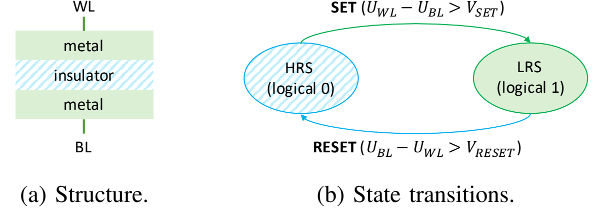
$$\textit{area utilization} \overset{\text{def}}{=\!=} \frac{\textit{storage area}}{\textit{crossbar area}}, \tag{1}$$

which represents the ratio of storage in the crossbar. Specifically, for a logic function implemented in the RRAM crossbar, the storage consists of inputs and outputs, and its area utilization becomes

$$\textit{area utilization} = \frac{|\textit{inputs}| + |\textit{outputs}|}{\textit{bounding box area}}. \tag{2}$$

The synthesis flow for the stateful logic should have high area utilization and reduce the area for redundant inputs, intermediate variables, unused cells as much as possible.

For example, Fig. 1 shows the data layout during the computation procedure of image convolution in IMAGING [19]. Despite of high parallelism, the area utilization of image convolution is 8.30%. Several copies of the input image and the filter occupy 25.53% of the crossbar, and the intermediate variables occupy 66.17%. To combine storage and computation in the same crossbar, 25.53%+66.17%=91.70% of the crossbar needs to be reserved for computation. Assuming that we have a 1 GB RRAM, we can only store 83 MB images in it. It cannot play the role of storage well.



(a) Structure. (b) State transitions.

Fig. 2: Schematic of an RRAM cell.

As shown in Table I, most of the previous synthesis flows [9], [17], [11], [12], [13], [14] optimize the latency of a given logic function at a cost of less than 10% area utilization. Large amount of extra inputs and intermediate variables are written into the crossbar during computation, and more than half of the RRAM cells are unused. SIMPLER [15] optimizes the throughput instead of the latency by reusing invalid cells and computing multiple instances in parallel. However, its reusing algorithm cannot get the optimal computation sequence. As a result, the area utilization still stays around 50%.

For a given crossbar size and a synthesis flow, considering that the throughput (per crossbar) is proportional to the area for computation, it is unacceptable to improve the area utilization directly in previous works, which will lead to a significant throughput loss. To address this problem, we propose STAR, a novel synthesis flow for the stateful logic in RRAM, in this work. The key point is to **improve the area utilization without throughput loss**. The main contributions of this work are listed as follows:

- We propose STAR, a synthesis flow for stateful logic in RRAM, which improves area utilization by reducing the area for primary inputs and intermediate variables without throughput loss.
- We present a case study of the image convolution. We first design a dense data placement scheme and then perform highly parallel computation using the limited area under the guide of STAR.
- We experimentally evaluate STAR and the image convolution implementation with the state-of-the-art works on the stateful logic to show our advantages in area utilization and throughput.

The rest of this paper is organized as follows. Section II summarizes the mechanism of the stateful logic. Section III proposes the synthesis flow in detail. In Section IV, we present a case study of the image convolution. The proposed synthesis flow STAR and the image convolution implementation are evaluated with experimental results in Section V. Finally, Section VI introduces some related works, and Section VII concludes the paper.

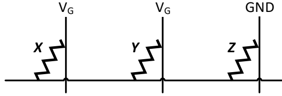


Fig. 3: A NOR operation $Z = \mathrm{NOR}(X, Y)$. RRAM Z is initialized to LRS. V_G satisfies $V_G > 2 \cdot V_{\mathrm{RESET}}$. Z will be reset to HRS if X or Y stays at LRS.

R_{11}	R_{12}	...	...	R_{1m}
R_{21}	R_{22}	...	...	R_{2m}
...	...	...	...	...
...	...	...	...	...
R_{m1}	R_{m2}	...	...	R_{mm}

Fig. 4: Parallel NOR operations in a crossbar. Each cell in the figure represents an RRAM cell. We can execute WL operations $R_{im} = \mathrm{NOR}(R_{i1}, R_{i2})$ or BL operations $R_{mi} = \mathrm{NOR}(R_{1i}, R_{2i})$ for $1 \le i \le m$ in parallel.

II. Background

A. Digital RRAM Cell

RRAM [21] is made of a dielectric material that is fabricated between two metal electrodes, as shown in Fig. 2a. The two terminals connect to the word line (WL) and the bit line (BL), respectively. The logical state of a digital RRAM cell is represented by its resistance, where the high resistance state (HRS) is logical zero and the low resistance state (LRS) is logical one. The two states can be switched mutually at certain conditions, as summarized in a state machine in Fig. 2b. When applying a positive voltage which is larger than V_{SET}, RRAM cells can be switched from HRS to LRS. When applying a negative voltage with a magnitude larger than the erase voltage V_{RESET}, RRAM cells can be switched from LRS to HRS. These two voltage patterns implement the logic operations SET $Y = 1$ and RESET $Y = 0$, respectively.

B. Memristor-Aided loGIC (MAGIC)

Stateful logic is one of the techniques for RRAM based in-memory logic, where both the inputs and outputs of the logic gates are the states of the RRAM. The applied voltages across the input cells write the result to the output cell based on the values stored in the input cells initially. Fig. 3 shows the schematic of Memristor-Aided loGIC (MAGIC) [5], a widely-used stateful logic family. In this example, Z is initialized to LRS. When we apply a voltage pulse of V_G ($V_G > 2V_{\mathrm{RESET}}$), V_G, and GND on BLs of cells X, Y, and Z, respectively and connect their WLs, we perform a NOR operation $Z = \mathrm{NOR}(X, Y)$. We can also perform the NOR operation with more than two inputs similarly.

Here we give a detailed examination of this two-input NOR operation. Two input cells X and Y are connected in parallel. When one of the inputs stays at LRS, the total resistance of the inputs is smaller than LRS. As a result, the negative voltage on Z is greater than $V_G/2 > V_{\mathrm{RESET}}$ and is large enough to reset it into HRS. Otherwise, the voltage on Z is close to zero, and Z remains unchanged. Z's value becomes 0 only if at least one of the inputs is 1, which is consistent with the NOR logic.

TABLE II: Stateful logic families.

Work	Stateful logic operations
[4]	IMP
[5]	NOR, NOT
[6]	NAND, NIMP
[7]	NOR, NAND, Min, OR
[22]	NOR, NOT, NAND, NIMP, XOR

Fig. 4 shows the schematic of NOR performed over rows and columns within a symmetric RRAM crossbar. The m horizontal wires are WLs and the m vertical ones are BLs. Each junction of a WL and a BL has an RRAM cell. Parallel execution of operations requires alignment of their inputs and outputs. Thus, we can apply a logic operation to WLs (also referred as WL operations) or BLs (also referred as BL operations) simultaneously using the same voltage pattern. The operation takes the period of a single voltage pulse, regardless of the number of parallel WLs or BLs [9].

C. Stateful Logic Families

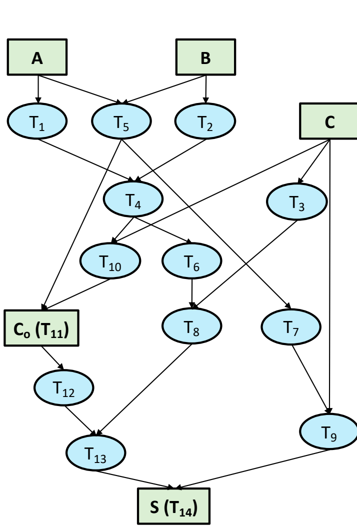
Previous works have demonstrated some other stateful logic families, which are summarized in Table II. These works are a little different in their implementation details. For example, Huang *et al.* define HRS and LRS as logical 1 and 0, respectively [6]. Xu *et al.* combine the unipolar and bipolar devices in the same crossbar [22]. Despite the differences, all of them support SIML-fashion parallelism in a symmetric RRAM crossbar. Also, these stateful logic families are functionally complete, and thus, any logic functions can be implemented in a finite number of RRAM cells using finite voltage pulses. We have proved that these differences have no effect on the order of the time complexity [8]. Without loss of generality, we take two-input MAGIC NOR operations as an example in the rest of the paper.

III. STAR: Synthesis Flow

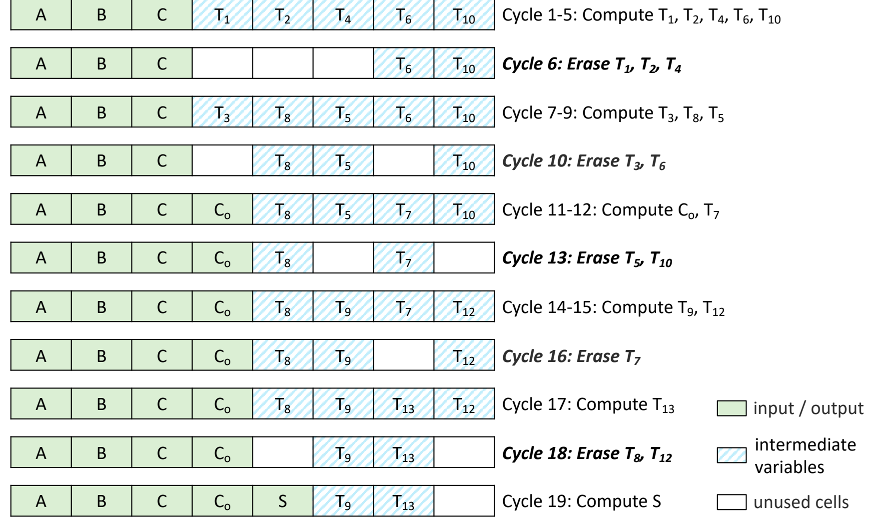
We propose STAR, a synthesis flow with high area utilization, in this section, as shown in Fig. 6. STAR contains seven steps labelled by *A* to *G*. The first six steps *A* to *F* target low latency of a serial function, i.e., all of the stateful logic operations are executed in series in an RRAM line, and we try to reduce the number of the stateful logic and the variable erasing operations at these steps. The last step *G* extends the one-line synthesis result to multiple lines. We integrate the optimization strategies on area utilization into STAR (Opt 1: Step *B*, Opt 2: Step *C* to *E*). We introduce STAR step by step in the corresponding subsections using the full adder in Fig. 5 as an example and focus on the optimization strategies.

A. Synthesize for Least Primitive Operations

For a logic function given in the Verilog format, we convert it into an operation graph using the ABC synthesis tool [23] with a customized logic library. The operation graph can be regarded as a Directed Acyclic Graph (DAG) $G = \langle V, E \rangle$, in which V is the set of variables, and E represents the stateful logic operations. If $Y = F(A_1, A_2, ..., A_n)$ (F is an operation in the library) exists in the synthesis result, there is



(a) The full ader NOR-Inverter Graph.



(b) Our implementation with the highest area utilization (the least cells).

Fig. 5: Our RRAM implementation of the one-bit full adder. For one full adder instance, our implementation takes 19 cycles with 8 cells. We insert 5 erasing operations during the computation procedure. We can also compute multiple instances within the same latency. Our area utilization is (3+2)/8=62.5%.

an edge from A_1, A_2, ..., A_n to Y in G, respectively. Although different stateful logic families may affect the area utilization and latency of the function, we can deal with them using a uniform synthesis flow.

Example. If we use the MAGIC stateful logic family, the customized library consists of two types of gates, the two-input NOR gate (NOR2X1) and the NOT gate. The operation graph becomes NOR-Inverter Graph. The one-bit full adder consists of 14 NOR (NOT) gates.

B. Opt 1: Reduce the Area for Redundant Inputs

We use three methods to reduce the area for primary inputs. First, we utilize the inter-instance parallelism and do not copy inputs. That is to say, we compute one instance, i.e., one full adder, in one RRAM line and compute multiple instances using the SIML-fashion parallelism instead of parallelizing logic operations within one instance. All of the cells in the bounding box participate in the computation, which provides high area utilization. It is worth noting that we do not copy input instances, and the parallelism only comes from the instance number. On contrary, although some previous works, e.g., IMAGING [19], also designs stateful logic algorithms using this natural parallel manner, they improve parallelism by copying one instance to multiple lines, which still introduces area and initialization overhead.

Example. If we compute multiple adder instances in parallel, it still takes 19 cycles. The area utilization remains unchanged.

Second, we encode the shared constants among all of the instances into the control signals once to save the cells storing them and the operations writing multiple copies into the crossbar. In detail, in the operation graph, we simplify the operations that directly contain a shared constant p or their negations by deleting unitary operations and converting n-input operations to $(n{-}1)$-input operations. If we use the MAGIC stateful logic family in STAR, we delete $Y = \mathrm{NOT}(p)$ and convert $Y = \mathrm{NOR}(A, p)$, $Y = \mathrm{NOR}(A, \mathrm{NOT}(p))$ to $Y = 0$ or $Y = \mathrm{NOT}(A)$ according to the value of p.

Example. If we already know $C = 0$ or $C = 1$ for all adder instances, we do not need to write C and its negation into the crossbar. In the full adder NOR-Inverter Graph, we delete C, its negation T_3, and four outgoing edges. The control signals generating T_{10} and T_8 depend on the value of C.

Third, if the inputs will not be used in the future, we treat them as the intermediate variables and erase them during the computation procedure. We set $coverInput = true$ in the following scheduling algorithm (Algorithm 1) of this flow.

Example. If $coverInput = true$, we can implement the full adder with only 6 cells and increase the area utilization to $(3 + 2)/6 = 83.33\%$..

C. Opt 2a: Insert the Variable Erasing Operations

Step C to E constitute Opt 2. We reuse RRAM cells by erasing invalid variables. If the internal inputs will no longer be used, we also erase them during the computation. In this strategy, the area utilization of STAR is affected by the variable computation sequence, the cell number, and the variable erasing rule. The former two depend on the variable erasing rule, so we first introduce the rule.

For a given computation sequence and the cell number, considering that multiple RRAM cells can be set at the same time [24], we follow a *lazy erasing rule* in STAR, i.e., we erase all invalid variables by parallel SET operations when the intermediate variables use up the RRAM cells. In fact, the number of cells set together has an upper bound SET_{max} [1], which varies with the RRAM physical parameters. We ideally assume that SET_{max} equals to the columns in the crossbar in this section and demonstrate that a finite SET_{max} has a small impact on the number of erasing operations in Section V.

[1] SET_{max} has the same meaning as $LimitInitCells$ in SIMPLER.

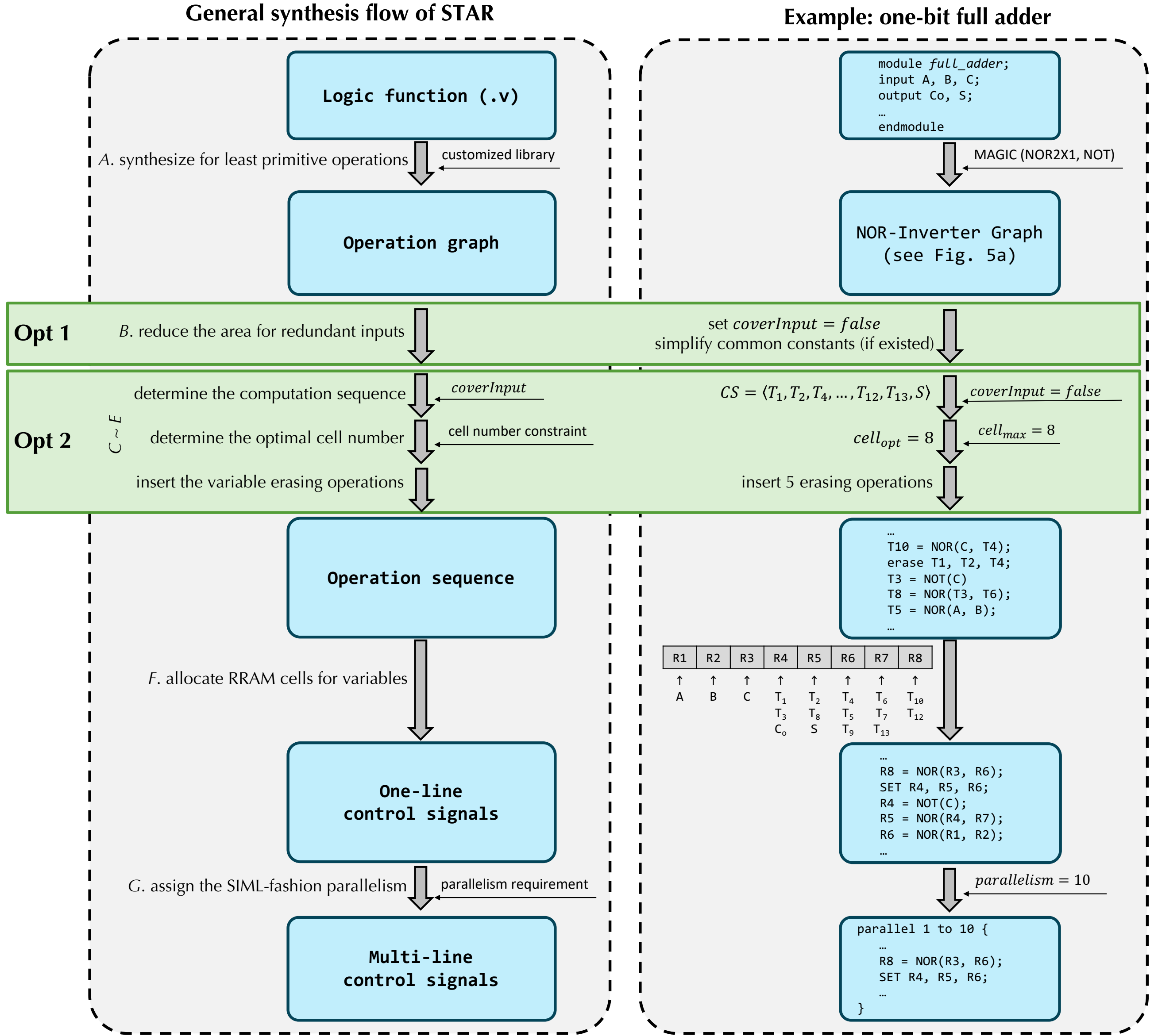


Fig. 6: Our high area utilization synthesis flow STAR with two optimization strategies. Optimization 1: reduce the area for redundant inputs. Optimization 2: reuse the RRAM cells for intermediate variables. The left part is the general synthesis flow of STAR, and the right part takes the one-bit full adder in Fig. 5 as an example. For simplicity, we assume that the NOR-Inverter Graph in Fig. 5a is the synthesis result of ABC. (In fact, the fastest series full adder only takes 9 cycles.)

Example. All of the cells are exhausted after computing T_{10}, so we erase T_1, T_2, T_4 in the next cycle in parallel as they will not participate in the following computation.

D. Opt 2b: Determine the Computation Sequence

Inserting erasing operations will increase the total latency despite of improving area utilization. To alleviate the negative effect, we want to find a computation sequence that can minimize the erasing operations. We formulate the erasing operation minimization problem as follows.

Erasing operation minimization problem. Given an operation graph $G = \langle V, E \rangle$ and $cell_{opt}$ cells for the logic function, find a computation sequence CS, which can minimize the variable erasing operations under the lazy erase rule.

We prove that the erase operation minimization problem is NP-hard by showing that the register allocation problem, a known NP-hard problem [25], is a special case of it. Register allocation is a problem of assigning a large number of target variables onto a small number of registers. On the one hand, if we consider registers as RRAM cells, the register allocation problem is equivalent to solving the minimal cell number. On the other hand, the erasing operation minimization problem can decide whether the register allocation at a given cell number is feasible. We can solve the minimal cell number by binary search of the cell number and this problem, so the erasing operation minimization problem is also NP-hard. As a result, we propose a heuristic scheduling algorithm to find a near optimal solution, as shown in Algorithm 1.

Note that no more than $cell_{opt}$ variables can be stored in

Algorithm 1 Scheduling algorithm for the erasing operation minimization problem

Input: An operation graph $G = \langle V, E \rangle$, $coverInput$
Output: The computation sequence CS with the minimal variable erasing operations
1: $CS \leftarrow \varnothing$
2: **if** $coverInput == false$ **then**
3: $\quad valid \leftarrow \varnothing$
4: **else**
5: $\quad valid = G.input$
6: **end if**
7: **for** $i = 1$ to $|V| - |G.input|$ **do**
8: $\quad$ **for** $v \in (V - G.input - CS) \cup valid$ **do**
9: $\quad\quad$ Update $erasePre(v, CS)$
10: $\quad$ **end for**
11: $\quad$ **if** $valid == \varnothing$ **then**
12: $\quad\quad candidate \leftarrow V$
13: $\quad$ **else**
14: $\quad\quad$ // select $toErase$
15: $\quad\quad toErase = \arg\min_{v \in valid}\{erasePre(v, CS)\}$
16: $\quad\quad candidate \leftarrow erasePre(toErase)$
17: $\quad$ **end if**
18: $\quad$ // select $toCompute$
19: $\quad candidate - = \{x | pre(x)\ is\ not\ computed\}$
20: $\quad toCompute = \arg\min_{v \in candidate}\{erasePre(v, CS)\}$
21: $\quad$ Add $toCompute$ into $CS, valid$
22: $\quad$ Update $valid$
23: **end for**

RRAM when the computation is finished. As a result, at least $|V| - cell_{opt}$ variables have to be erased, and the number of the erasing operations can be represented as follows:

$$\begin{aligned} erasing\ operations &= \frac{total\ erased\ variables}{average\ erased\ variables\ once} \\ &\geq \frac{|V| - cell_{opt}}{average\ erased\ variables\ once}. \end{aligned} \tag{3}$$

We can minimize the variable erasing operations by maximizing the erased variables once.

$erasePre(v, CS)$ is the core concept in the algorithm. v is a vertex in V. At a certain moment during the algorithm execution, CS is the partial computation sequence. $|erasePre(v, CS)|$ represents the minimal variables to compute, i.e., the minimal cells to use, before v can be erased in the next erasing operation.

A vertex can be erased only if it will not be used in the future. For example, after the 4th cycle in our full adder implementation, $CS = \{T_1, T_2, T_4, T_6\}$. To erase T_4, its successor T_{10} has to be computed, i.e., $erasePre(T_4, CS) = \{T_{10}\}$. Similarly, to erase T_6, its successor T_8 has to be computed. T_3, the other precursor of T_8 also needs to be computed first. As a result, $erasePre(T_6, CS) = \{T_8, T_3\}$. Since there is only one cell left, we choose to compute $erasePre(T_4)$ in the 5th cycle and erase 3 variables in the 6th cycle. Otherwise, if we compute $erasePre(T_6, CS)$ first, we can only erase T_1, T_2 in the 6th cycle.

From the above example we can see that, *for a given cell number, to erase as many variables as possible once, we should erase the variables in the crossbar with the minimal* $|erasePre|$ *by computing their* $erasePre$ *sets first.*

Here gives a formal definition for $erasePre$. For a vertex $v \in V$, $pre(v)$ is the precursors of v, and $suc(v)$ is the successors of v. At a certain moment, $erasePre(v, CS)$ can be defined recursively:

- $suc(v) \subset erasePre(v, CS)$.
- If $x \in erasePre(v, CS)$, $y \in pre(x)$ and $y \notin CS$, $y \in erasePre(v, CS)$.

With the computation of the function, $|CS|$ becomes larger and larger, while $erasePre$ sets become smaller and smaller. Finally, the variables with large $|erasePre|$ at the beginning can also be computed and erased using the limited cells.

We also define $valid$ to represent the variables that cannot be erased now:

$$valid = \{x | x \in CS \wedge erasePre(x, CS) \neq \varnothing\}. \tag{4}$$

All of the intermediate variables in $valid$ have to be stored in the crossbar at this moment because their $erasePre$ sets depend on them. If $coverInput = true$, we also add the input variables into $valid$..

Back to Algorithm 1, Line 2 to Line 6 initialize $valid$. Each iteration from Line 7 to Line 23 selects $toCompute$ and adds it into CS. In detail, Line 13 to Line 17 select the variable in $valid$ with the minimal $|erasePre|$ as $toErase$, which means we want to erase it in the next erasing cycle. Then, Line 18 to Line 20 select the variable in $candidate(erasePre(toErase))$ as $toCompute$. If there are several variables in $candidate$ that can be computed now, we select the variable with the minimal $|erasePre|$ such that it can be erased as soon as possible.

The algorithm iterates for less than $|V|$ times. In each iteration, for a given v, if $toCompute$ in the previous cycle belongs to $erasePre(v, CS)$, we delete it from the set. Otherwise, $erasePre(v, CS)$ remains unchanged. As a result, it takes $O(|V|)$ cycles to update all of the vertices. Both $|valid|$ and $|candidate|$ is smaller than $|V|$, so it takes $O(|V|)$ cycles to select $toErase$ and $toCompute$. The total time complexity of the scheduling algorithm is $O(|V|^2)$. As an example, STAR maps a graph with over 12K vertices using 10.5 seconds on a Core i5 with 16-GB RAM.

Example 1. In the 1st cycle, $valid = \varnothing$. We select T_1 as $toCompute$ because it is the variable in V with the minimal $|erasePre|$ ($|\{T_2, T_4\}| = 2$).

Example 2. In the 4th cycle, $valid = \{T_4, T_6\}$. $erasePre(T_4)$ is smaller, so we select T_4 as $toErase$. We select T_{10} in $erasePre(T_4)$ as $toCompute$.

E. Opt 2c: Determine the Optimal Cell Number

We can get the minimal cell number $cell_{min}$ during the execution of the scheduling algorithm:

$$cell_{min} = \begin{cases} |G.input| + |valid|_{max} + 1 & coverInput = false \\ |valid|_{max} + 1 & coverInput = true \end{cases} \tag{5}$$

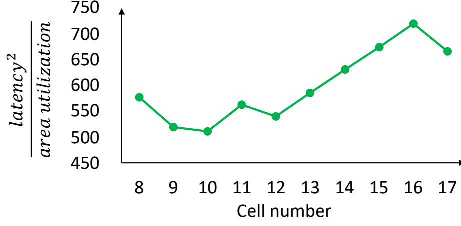


Fig. 7: The optimal cell number of the full adder ($\alpha = 2$).

```
int image[N + 2][N + 2], filter[s][s];
for (int x = 0; x < N; ++x)
    for (int y = 0; y < N; ++y){
        int sum = 0;
        for (int i = 0; i < s; ++i)
            for (int j = 0; j < s; ++j)
                sum += image[x+i][y+j] * filter[i][j];
        image[x + s/2][y + s/2] = sum;
    }
```

Fig. 8: Image convolution.

Besides $|G.input|$ cells for the inputs, $|valid|$ cells for the intermediate and output variables that cannot be erased now, another cell is reserved for the variable to be compute in the next cycle.

Only using $cell_{min}$ cells will introduce several erasing operations and increases the total latency. Still taking the full adder as an example, the implementation without reusing only needs 14 cycles in spite of 17 cells. As a result, we make a trade-off between the area utilization and the latency by selecting the optimal $cell_{opt}$ for a function:

$$\begin{aligned} cell_{opt} &= \arg\min_{cell_{opt}} \frac{latency^{\alpha}}{area\ utilization} \\ &= \arg\min_{cell_{opt}} \frac{(|CS| + erasing\ operations)^{\alpha}}{\frac{|G.input|+|G.output|}{cell_{opt}}} \end{aligned} \tag{6}$$

α is a parameter defined by the programmer. If $\alpha \to +\infty$, the algorithm prefers to minimize latency without reusing cells. If $\alpha \to 0$, the algorithm prefers to maximize area utilization by reusing as many cells as possible. If the total cell number $cell_{max}$ stays between $cell_{min}$ and $cell_{opt}$, we let $cell_{opt} = cell_{max}$.

The computation sequence CS in our algorithm is cell number irrelevant. We can compute the total erasing operations with a given cell number under the lazy erasing rule. We can compute the number of the erasing operations for a given cell number by simulating the computation procedure using $O(|V|)$ cycles, so the total time complexity of determining $cell_{opt}$ is $O(|V| \cdot (cell_{max} - cell_{min}))$.

Example. As shown in Fig. 7, the optimal cell number of the full adder is $cell_{opt} = 10$ when we set $\alpha = 2$. If $cell_{max} = 8$, we select $cell_{opt} = 8$.

F. Allocate RRAM Cells for Variables

We allocate RRAM cells for variables and convert the operation sequence to the control signals, which consist of the input and output RRAM cells and the operation types. If the inputs cannot be covered, we do not reuse the cells storing them. The cells storing the output variables can only be reused before the output variables are computed. Two variables can share the same cell only if one is erased before the other is computed.

Example. T_8 is computed after T_1 is erased so they can share the cell R3. R_4 is not reused after the output variable C_o is computed.

G. Assign the SIML-fashion Parallelism

We extend the one-line control signals to multiple lines using the inter-instance parallel manner. According to the parallelism requirement, we apply the control signals to these lines at run time in parallel.

IV. Case Study: Image Convolution

We present a case study of the image convolution to demonstrate the effectiveness of STAR in this section. Convolution is a widely-used operator in image processing. Figure 8 shows an example. An $s \times s$ filter slides over an $N \times N$ image and multiplies with the corresponding pixel values of the image.

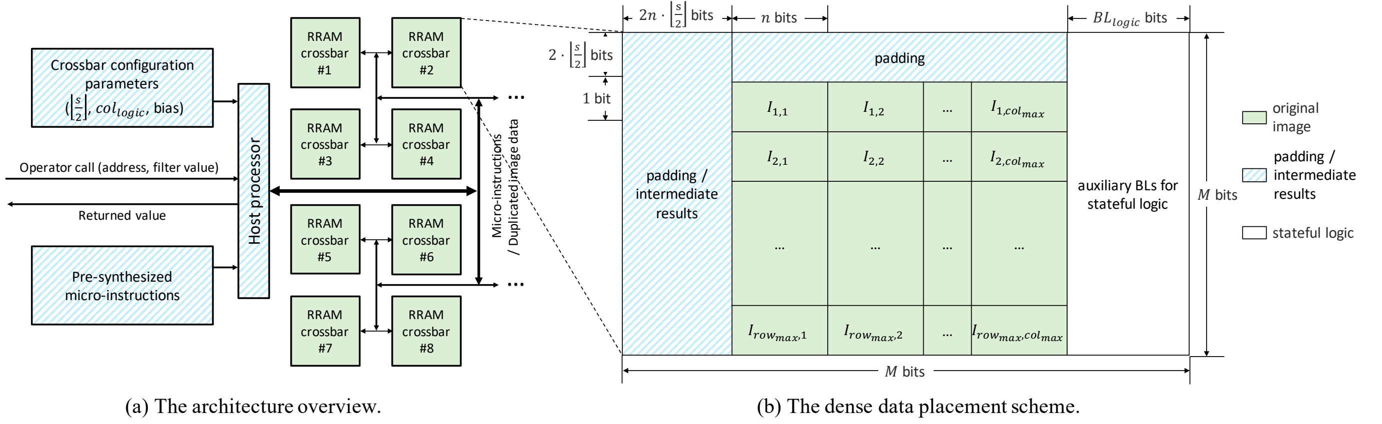
Similarly to the previous works [19], [26] that implement convolution using the stateful logic, we only consider optimizing the classical four-loop image convolution. We focus on the computation in one crossbar by proposing a dense data placement scheme and a highly parallel computation procedure under the guide of STAR.

Compared to IMAGING, for Opt 1, first, we store only one copy of the input image in the crossbar without parallelism loss. Second, we encode the filters, the shared constants of all pixels, in the control signals. Third, we mark $coverInput$ of additions as $true$ because addends will not be used. For Opt 2, we significantly reduce the area for stateful logic by reusing. The rest of this section gives the details of our implementation.

A. Dense Data Placement Scheme

Figure 9a depicts the architecture for image processing, in which each RRAM crossbar plays two roles at the same time, as both a storage unit and a computation unit. Images are stored and processed in the same crossbars with little data transfer. The area and offset of the computation region in one crossbar are determined by the global crossbar configuration parameters. Multiple RRAM crossbars are connected using the H-tree topology. Data can be transferred between adjacent arrays via high speed links. We store each image in several contiguous crossbars, and thus, different images can be processed in parallel without I/O conflicts. It is able to store a large number of images in a single chip due to the high density of RRAM, and the total parallelism is considerable.

The image processing functions are pre-synthesized to one-line control signals. Once the host processor receives an operator call that contains the image address and the filter value, it first writes the padding data in each crossbar. For big images stored in multiple crossbars, convolution of pixels on the border of sub images requires padding pixels in the adjacent crossbars. Then, it fills in the control signals with the filter values and assigns parallelism according to the image size. Finally, it dispatches the instantiated control signals to the corresponding crossbar(s).



(a) The architecture overview.

(b) The dense data placement scheme.

Fig. 9: (a) The architecture overview. The architecture consists of a host processor and several RRAM crossbars that are connected using the H-tree topology. (b) The dense data placement scheme. The whole RRAM crossbar is divided into three parts, which are assigned different colors. Symbols used in this figure are summarized in Table III.

TABLE III: Symbol explanation and their typical values.

Symbol	Explanation	Typical value
n	pixel width	8
s	filter size	3
M	crossbar row/column	512
BL_{logic}	# of BLs for stateful logic	48
row_{max}	the maximum image height	510
col_{max}	the maximum image width	56

We design a dense data placement scheme to ensure high area utilization with high SIML-fashion parallelism in an RRAM crossbar, which is shown in Figure 9b. The crossbar is divided into three parts. The leftmost $2n \cdot \lfloor \frac{s}{2} \rfloor$ BLs and the $2 \cdot \lfloor \frac{s}{2} \rfloor$ WLs in the top (the blue region) are left for padding data. Some intermediate results can also be stored here when padding data are invalid. The rightmost BL_{logic} BLs (the white region) are left for performing the stateful logic operations. We do not reserve a region at the bottom of the crossbar for the stateful logic operations because our computation of each pixel is directly right to the it. We do not reserve a region for the filter in the crossbar because they are shared constants of all image pixels in the operator call. We encode it into control signals before computation according to the second method of Opt 1.

The original image is aligned in the center of the crossbar (the green region). Each image row occupies an RRAM WL. The maximal height and width of a gray-scale image in one crossbar is:

$$row_{max} = M - 2 \cdot \lfloor \frac{s}{2} \rfloor, \tag{7}$$

$$col_{max} = \lfloor \frac{M - BL_{logic}}{n} \rfloor - 2 \cdot \lfloor \frac{s}{2} \rfloor. \tag{8}$$

row_{max} have to be divided by 3 for an RGB image. If the image size exceeds the limitation, we split it into multiple crossbars. We only copy the padding data at run time but not the whole image. The area utilization of our data placement scheme is:

$$\begin{aligned} area\ utilization &= \frac{n \cdot row_{max} \cdot col_{max}}{M^2} \\ &\approx 1 - \frac{(2n+2) \cdot \lfloor \frac{s}{2} \rfloor + BL_{logic}}{M}. \end{aligned} \tag{9}$$

Considering the typical values in Table III, the typical area utilization is about 88.16%. This area utilization is high enough to fit the PIM concept. Still assuming that we have a 1 GB RRAM, we can store 881 MB images in it.

B. Highly Parallel Computation Procedure

We propose an inter-instance parallel implementation for image convolution. We compute image Column $\#1$ to Column $\#col_{max}$ in turn, so different columns can share the stateful logic region. The computation of one column involves $\lfloor \frac{s}{2} \rfloor$ columns on its left. Therefore, if the results should be written to the original position, we will store the result of a column in the leftmost region of the crossbar temporarily until the original data become invalid.

We divide image Column $\#c$ into s groups, $\{I_{1,c}, I_{1+s,c}, I_{1+2s,c}, ...\}$, $\{I_{2,c}, I_{2+s,c}, I_{2+2s,c}, ...\}$, ..., and $\{I_{s,c}, I_{2s,c}, I_{3s,c}, ...\}$. Each group contains about $\frac{row_{max}}{s}$ pixels. The computation of the pixels in the same group has no data overlap, so we can compute them in parallel. Figure 10 shows our computation of one group, which consists of five steps:

1) Multiply pixels in one BL with the corresponding filters in parallel (①).
2) Add the partial results in the same WL in parallel (②). Both n-bit multiplication and addition are synthesized using STAR. If there exist pixels that have not been computed yet, go to Step(1).
3) Negate half of the partial results to the right using logic NOT operations in parallel (③⑦).
4) Negate these data up one by one (④⑤).
5) Add all WLs in parallel (⑥). If there still exist partial results, go to Step (3).

Step (1) to (2) multiply pixels and add the products in turn. The products generated in Step (1) can be erased after Step (2) to improve the area utilization. Step (3) to (5) form an adder tree and iterate for $\lceil \log s \rceil$ times. We add two partial results in a WL once but not all to save auxiliary RRAM cells and improve the area utilization. Assuming that computing an

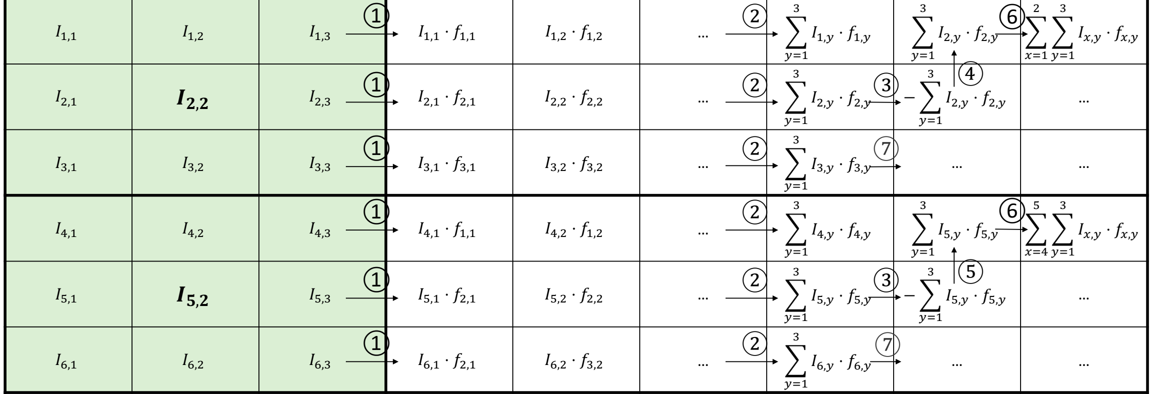


Fig. 10: One group computation. This figure takes $s = 3$ as an example. $I_{2,2}$ and $I_{5,2}$ are being computed. Some intermediate results of the stateful logic operations are not shown. Although the auxiliary area for stateful logic is much larger than the one for one group, different groups (or columns) can reuse the same region for stateful logic to keep a high area utilization.

addition in series takes $cycle_+$ cycles, the total cycles from Step (2) to (4) are:

$$cycle_{inter} \approx \sum_{i=1}^{\lceil \log s \rceil} (n + \lfloor \frac{row_{max}}{s} \rfloor \cdot \lfloor \frac{s}{2^i} \rfloor + cycle_+). \quad (10)$$

The i-th iteration adds $\lfloor \frac{s}{2^i} \rfloor$ partial sums of each target pixel. The degree of parallelism reaches $O(M)$ at most of the steps.

We select the global BL_{logic} for a crossbar:

$$BL_{logic} = \max\{cell_{\times,opt}, cell_{+,opt}\} + n. \quad (11)$$

We can get $cell_{\times,opt}$ and $cell_{+,opt}$ according to Equation 6 in Opt 2 of STAR. We have to compute at least two multiplications in a WL and then add them, so we reserve n cells for the product computed first. The input of the multiplication is the original image, so the $coverInput$ option is *false*. The input of the addition is the partial sums that will not be used in the future, so the $coverInput$ option is *true*. BL_{logic} is positively correlated to n, s, so we can get a higher area utilization under our dense data placement scheme with a smaller n, a smaller s, and a bigger M.

Our computation procedure achieves a high area utilization by fully reusing cells. We not only reuse cells during all n-bit additions and multiplications but also reuse the whole stateful logic region among different groups and columns.

V. Experimental Evaluation

We compare our work with previous works on stateful logic in this section. We first evaluate the general-purpose synthesis flow and focus on the scheduling algorithm. Then, we evaluate our implementation of image convolution. We only optimize the shared constants in the image convolution but not in the general-purpose comparison. We normalize the physical parameters, e.g., technology node, crossbar size, operation latency, in the comparison. The latency can be represented by the number of cycles, and the area can be represented by the number of RRAM cells.

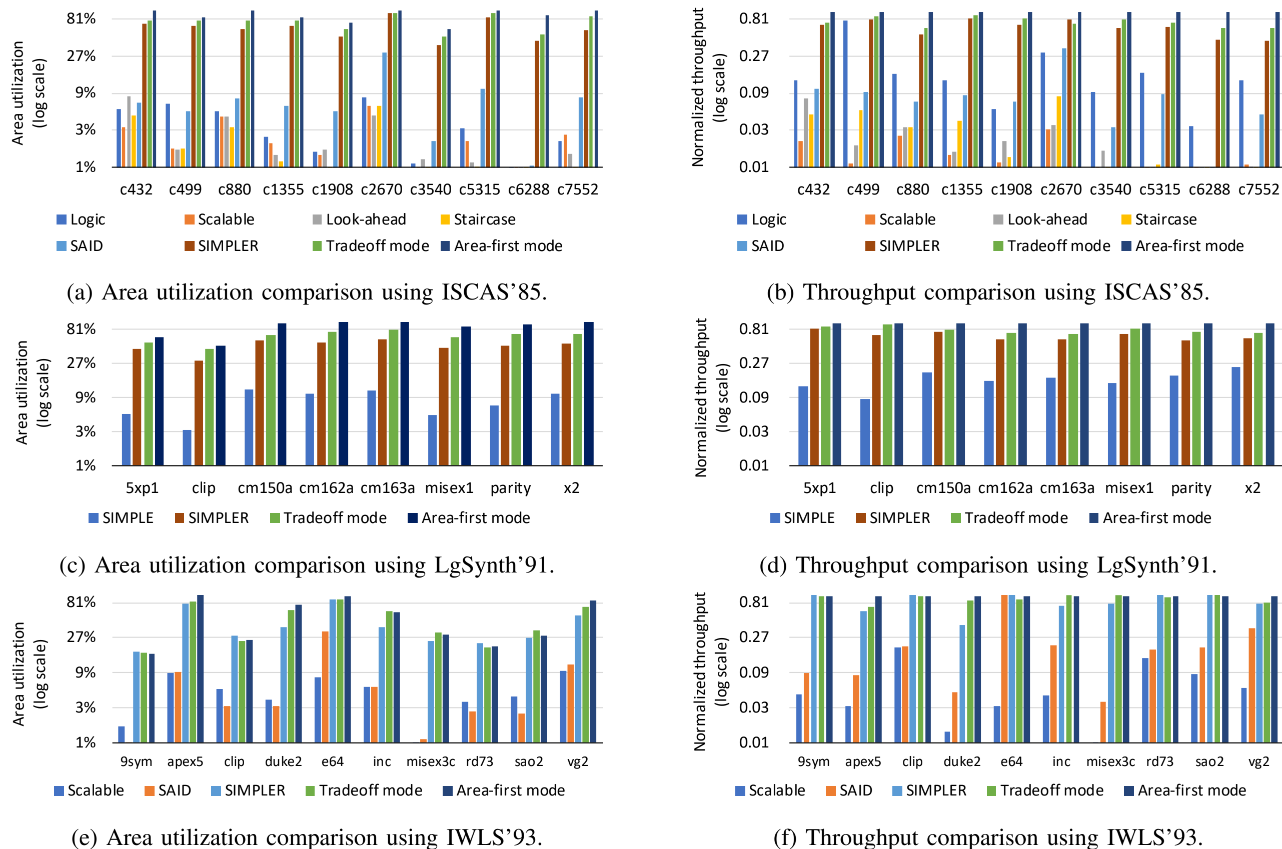
A. Synthesis Flow Evaluation

In Fig. 11, we compare STAR with seven other synthesis flows listed in TABLE I previously, in the area utilization and the throughput using three benchmark suites, ISCAS'85 [10], LgSynth'91 [16], and IWLS'93 [18]. Each previous synthesis flow uses one or more suites in the original publication. We support the four-input NOR operation additionally in the IWLS'93 benchmark, similarly to the previous works. We evaluate two modes of STAR: '*tradeoff mode*' makes a tradeoff between area utilization and throughput by using $cell_{opt}$ cells ($\alpha = 1$) and setting $coverInput = false$; '*Area-first mode*' uses $cell_{min}$ cells and can cover the input, so it has the highest area utilization. TABLE IV lists the average area utilization of STAR on these benchmark suites.

Fig. 11a, 11c, 11e evaluates the area utilization. Our '*area-first mode*' achieves 102.10%, 85.77%, 76.85%, 103.15%, 103.78%, 72.91%, 33.03% more area utilization, compared to Logic [9], SIMPLE [17], Scalable [11], Look-ahead [12], Staircase [13], SAID [14], SIMPLER [15], respectively. For previous synthesis flows, the area utilization of most cases is less than 50%. The intermediate variables and unused cells occupy most of the area. On contrary, the area utilization of '*area-first mode*' is usually larger than 80% and even exceeds 100% in nine cases because the input and output variables share some cells. The '*tradeoff mode*' also achieves a significant area utilization improvement.

STAR has less advantage in a few cases, e.g., 9sym and clip in IWLS'93, because of the complex operation graph. These cases have very few input and output variables and many intermediate variables. Most variables have high fan-in and fan-out, so their $erasePre$ sets are relatively large. We have to store lots of intermediate variables in the RRAM crossbar during execution.

As shown in Fig. 11b, 11d, 11f, our '*area-first mode*' achieves 9.83×, 6.65×, 69.33×, 82.46×, 318.42×, 17.42×, 1.43× throughput (per crossbar) compared to Logic [9],



(a) Area utilization comparison using ISCAS'85.

(b) Throughput comparison using ISCAS'85.

(c) Area utilization comparison using LgSynth'91.

(d) Throughput comparison using LgSynth'91.

(e) Area utilization comparison using IWLS'93.

(f) Throughput comparison using IWLS'93.

Fig. 11: Synthesis flow evaluation using three benchmark suites. In (a)(c)(e), partial input and output variables may share the same cells, so the area utilization of the '*area-first mode*' can exceed 100%. In (b)(d)(f), the throughput of the '*area-first mode*' is normalized to 1. We ideally assume that previous flows can compute all of the instances in parallel. STAR achieves significant area utilization and throughput improvement compared to previous flows.

TABLE IV: High area utilization of STAR ('*Area-first mode*').

Benchmark	Work	Area utilization
ISCAS'85	STAR	105.79%
LgSynth'91		93.80%
IWLS'93		55.59%

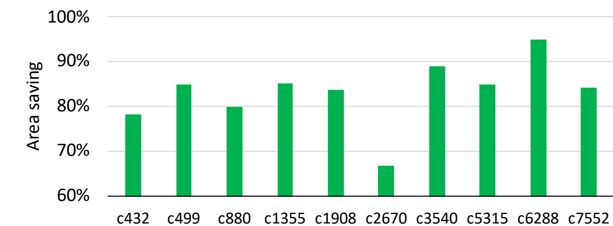
SIMPLE [17], Scalable [11], Look-ahead [12], Staircase [13], SAID [14], SIMPLER [15], respectively. In fact, the actual throughput gain can be much higher for two reasons. First, we fix the total area of each synthesis flow in this comparison, so the computation area of STAR is much less than that in other flows because of higher area utilization. Assuming that our area utilization is α and the area utilization of a previous work is β ($\alpha > \beta$), the throughput gain have to be multiplied by $\frac{1-\beta}{1-\alpha}$ if we fix the computation area of each synthesis flow. Second, the previous flows except SIMPLER cannot compute all of the instances in parallel, and the latency increases with the instance number. The throughput gain have to be multiplied for multiple instances. We just estimate an ideal situation for them in the comparison as SIMPLER.

B. Scheduling Algorithm Evaluation

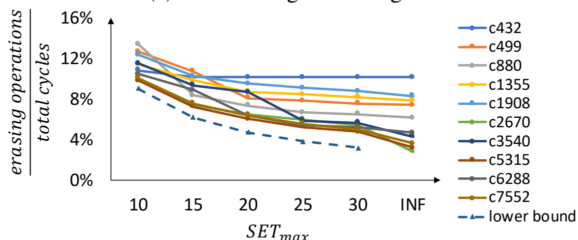
We evaluate our scheduling algorithm using the ISCAS'85 benchmark, as shown in Fig. 12. We use the configuration of the '*tradeoff mode*' in this figure. Fig. 12a evaluates the area saving of the algorithm compared with the one without reusing. Our algorithm saves 83.20% area compared to the original ABC synthesis result without reusing.

Fig. 12b evaluates the effect of different SET_{max}s on the number of erasing operations. We obtain three conclusions from this figure. First, the extra erasing operations inserted by the scheduling algorithm has a small overhead. The average erasing operation ratio is only 11.22% when $SET_{max} = 10$. It becomes 5.41% if we do not restrict SET_{max}. Second, the number of erasing operations increases slowly when decreasing SET_{max}. A smaller SET_{max} has less negative effect on the total latency. Third, our algorithm does well for any given SET_{max}. The minimal erasing operation ratio is $\frac{1}{1+SET_{max}}$ (the dotted line in the figure) if we always erase SET_{max} variables once after they are computed. The average erasing operation ratio of our algorithm is empirically less than twice of the lower bound.

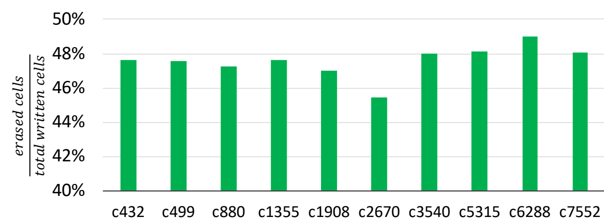
Fig. 12c evaluates the erased cell ratio on one cell, which stays between 40% to 50%. Since the energy consumption is proportional to the written cells, we can infer that the erasing operations consumes 40%-50% energy in the whole computation procedure. A cell is written by either an erasing operation or a MAGIC operation. That is to say, the number of erased cells is a little smaller than the number of MAGIC operations. In fact, a cell can be reused only after it is erased.



(a) Area saving of the algorithm.



(b) Effect of different SET_{max}s on the erasing operation ratio.



(c) Erased cell ratio. If a cell is erased or written multiple times, we record it multiple times.

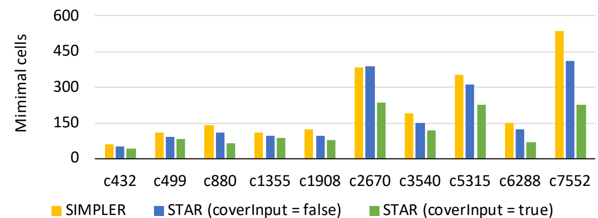
Fig. 12: Scheduling algorithm evaluation.

For all synthesis flows (with or without the reusing strategy), they need to erase (or initialize) the cells for intermediate variables before computing the new instance. If considering the cost of the initialization procedure, these flows erase a similar number of cells as STAR and consume similar energy on erasing operations. They only differ in when and how to erase these cells, so they have different latency and throughput.

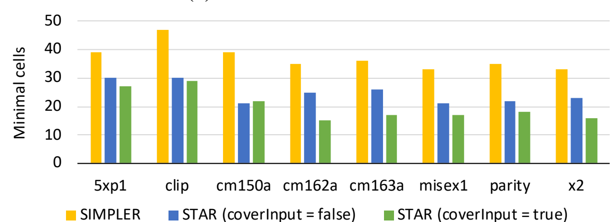
We also compare our scheduling algorithm with SIMPLER. Fig. 13 compares the minimal RRAM cells achieved by two scheduling algorithms. When setting $coverInput = false$, our algorithm saves 16.06%, 33.19%, 9.24% cells on the ISCAS'85, LgSynth'91, EPFL benchmarks, respectively. Our algorithm needs fewer cells on most of the cases. For the other cases where SIMPLER has got the near-optimal result, our algorithm can get the similar result. When setting $coverInput = true$, our algorithm saves 38.85%, 46.39%, 37.17% cells on the ISCAS'85, LgSynth'91, EPFL benchmarks, respectively. The area saving from erasing the inputs depends on the netlist structure and varies among different cases.

Our algorithm also saves most of the erasing operations and helps improve the throughput. For example, according to Fig. 14, our algorithm also saves 77.40% of the erasing operations when the number of the RRAM cells is fixed to the minimal cells of SIMPLER. Because the erasing operations only occupy a small part in the total computation procedure (see Fig. 12b), the throughput improvement is not obvious (see Fig. 11).

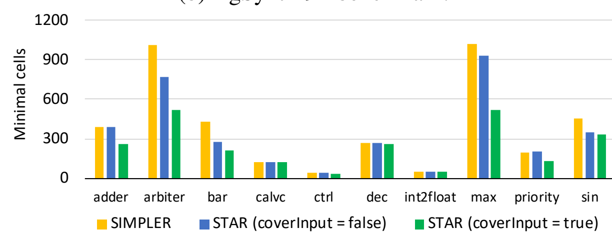
Our algorithm needs fewer cells and cycles because of our better heuristic function. SIMPLER estimates the minimal number of cells required for the execution of one vertex in the computation graph by the Strahler number, which is intended for trees only. The estimation is static and may be incorrect if a vertex has multiple successors. On contrary, the $erasePre$ set in our algorithm can exactly represent the minimal variables to compute before a vertex can be erased and dynamically adjust during the algorithm execution.



(a) ISCAS'85 benchmark.



(b) LgSynth'91 benchmark.



(c) EPFL benchmark.

Fig. 13: Minimal cells of two scheduling algorithms.

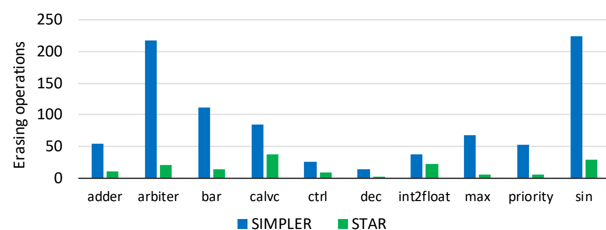
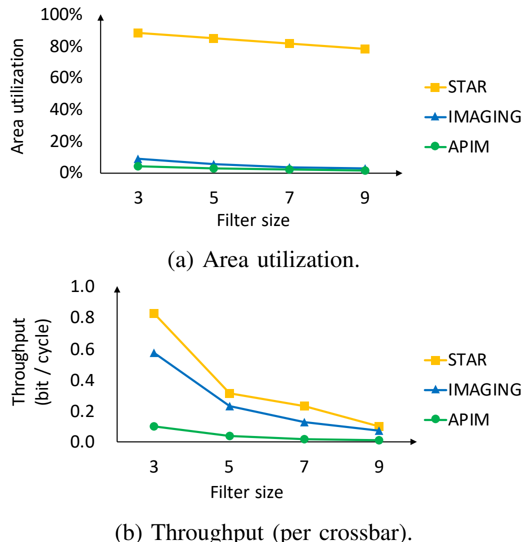


Fig. 14: Erasing operations of two scheduling algorithms using the minimal cells of SIMPLER on the EPFL benchmark.

C. Case Study Evaluation: Image Convolution

We compare our work with two MAGIC-based image processing accelerator APIM [26] and IMAGING [19]. Both APIM and IMAGING manually design the addition and multiplication. We get the results of IMAGING by using limited precision multiplication proposed in its work.

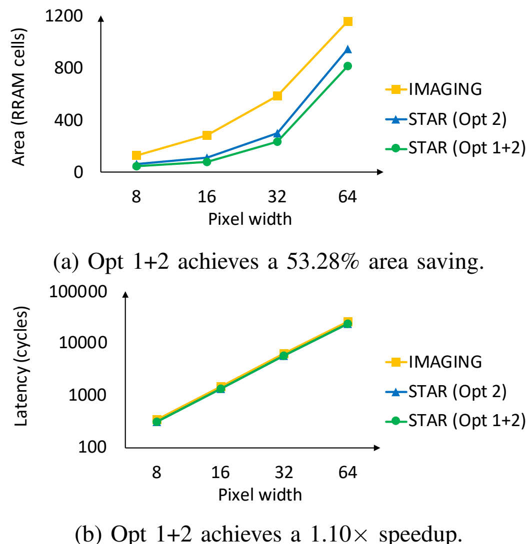
As shown in Figure 15a, STAR achieves 78.36% and 80.97% more area utilization on average over IMAGING and APIM, respectively. STAR keeps a greater than 75% area utilization in all cases while the other two works are smaller



(a) Area utilization.

(b) Throughput (per crossbar).

Fig. 15: Convolution comparison with different filter sizes. The crossbar size is 512×512.



(a) Opt 1+2 achieves a 53.28% area saving.

(b) Opt 1+2 achieves a 1.10× speedup.

Fig. 16: Limited precision multiplication comparison. Opt 1: encode the filter into control signals. Opt 2: reuse RRAM cells.

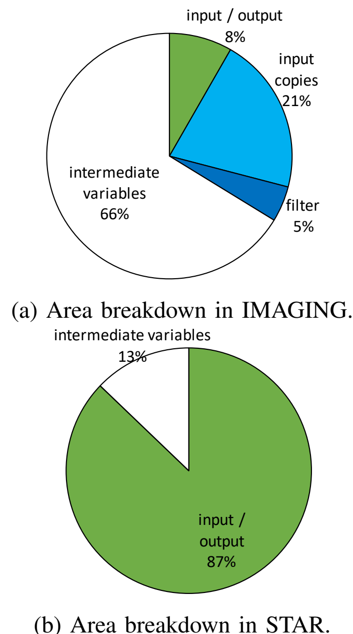
than 10%. As shown in Figure 15b, STAR achieves a 1.48× and 8.53× higher throughput per crossbar than IMAGING and APIM, respectively. The area utilization and throughput of three works decreases with the increase of the filter size as the computation becomes more and more complex.

We make a detailed comparison between our implementation and the state-of-the-art work IMAGING. Figure 16 compares the implementation of the most complex function in convolution, limited-precision multiplication, in which the input and output have the same pixel width. According to Figure 16a, our implementation saves 56.17% area. Two optimization strategies contribute 11.29% and 44.88%, respectively. According to Figure 16b, our implementation achieves an up to 1.10× speedup besides significant area saving.

Table V and Fig. 17 gives a detailed analysis for the filter size $s = 3$. The area utilization improvement comes from two parts. First, our row_{max} is 3× over IMAGING, as IMAGING

TABLE V: Detailed analysis when the filter size $s = 3$. The crossbar size is 512×512.

Work		**IMAGING**	**STAR**
Image size	row_{max}	170	510
	col_{max}	8	56
Area utilization		8.10%	88.16%
Data transfer / pixel (bit)	duplication	24	0
	filter	4.24	0
	padding	1.06	0.31
Execution cycles / pixel		11.03	9.63
Throughput (bit / cycle)		0.57	0.83



(a) Area breakdown in IMAGING.

(b) Area breakdown in STAR.

Fig. 17: Area breakdown when the filter size $s = 3$. The crossbar size is 512×512.

duplicates the (negative) image for $2s = 6$ times to compute s adjacent pixels in parallel. Second, our col_{max} is 3.3× over IMAGING, as IMAGING stores row_{max} filters, col_{max} partial sums, and many other intermediate variables simultaneously in a WL. The area utilization gap expands gradually with the increase of the filter size.

The throughput improvement also comes from two parts. First, our data placement scheme saves most of the data transfer. Only a few padding data will be written into the crossbar during initialization. Here assumes that the computation crossbars and the storage crossbars are adjacent and the data transfer speed is one image row / cycle. The initialization overhead will increase by about an order of magnitude or more if the original image is farther away [27], i.e., outside the bank or the chip. Second, our MAGIC execution achieves a 1.15× speedup over IMAGING on the same input image. Both works exploit the inter-instance parallel manner, so the total speedup in MAGIC execution mainly comes from that in series.

VI. Related Work

Recent works implement several stateful logic operations in the RRAM crossbar, e.g., IMP [4], NOR [5], and corresponding synthesis flows propose to fully utilize the massive

parallelism of the stateful logic. SIMPLE [17] minimizes the latency by solving an Integer Linear Programming (ILP) problem but has exponential time complexity. The following works design some heuristic algorithms to find an approximate solution. For example, Staircase [13] performs WL and BL operations alternately using a staircase structure to reduce extra data copy. SAID [14] represents the logic function by Look-Up Tables (LUTs) and then map the LUTs onto the crossbar to maximize LUT-level parallelism. However, these heuristic synthesis flows neglect the use of RRAM for data storage and have ultra-low area utilization. SIMPLER [15] tries to improve area utilization by reusing RRAM cells, but its scheduling algorithm cannot get the optimal result.

Some works map the function onto the RRAM crossbar manually. For example, APIM [26] designs a carry save tree structure for fast addition and multiplication. IMAGING [28], [19] proposes four algorithms for efficient execution of Fixed Point (FiP) multiplication using MAGIC gates under different constraints and then accelerate three common image processing applications. Although these designs achieve high throughput, their mapping methods are only optimized for a particular application and cannot apply to other applications.

Some studies perform computation on RRAM without the non-volatile stateful logic. For example, Pinatubo [20] and PIMA [29] implements bulk bitwise operations by redesigning the read circuitry. Besides digital fashion, RRAM also supports matrix-vector multiplication in the analog fashion, which has been exploited to accelerate convolutional neural networks (CNN) [30], [31], [32] and binary neural networks (BNN) [33]. Despite low latency, it lacks in accuracy and suffers from high variation, which restricts its scope of application. Moreover, power consumption from additional AD-conversion and I/Os cannot be ignored [34].

VII. Conclusion

In this work, we propose STAR, a synthesis flow for stateful logic in RRAM, to improve the area utilization without throughput loss. In STAR, we first reduce the area for primary inputs by not copying inputs, encoding the shared constants into the control signals, and erasing invalid inputs during the computation procedure. Then, we propose a scheduling algorithm to get the computation sequence with minimal erasing operations. Experimental evaluation shows that we can achieve ultra-high area utilization and improve the throughput in various benchmarks.

References

[1] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[2] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[3] T.-y. Liu, T. H. Yan, R. Scheuerlein, Y. Chen, J. K. Lee, G. Balakrishnan, G. Yee, H. Zhang, A. Yap, J. Ouyang *et al.*, "A 130.7-mm^2 2-Layer 32-Gb ReRAM Memory Device in 24-nm Technology," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 1, pp. 140–153, 2014.

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

[5] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC–Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[6] P. Huang, J. Kang, Y. Zhao, S. Chen, R. Han, Z. Zhou, Z. Chen, W. Ma, M. Li, L. Liu *et al.*, "Reconfigurable nonvolatile logic operations in resistance switching crossbar array for large-scale circuits," *Advanced Materials*, vol. 28, no. 44, pp. 9758–9764, 2016.

[7] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[8] F. Wang, G. Luo, G. Sun, J. Zhang, P. Huang, and J. Kang, "Parallel Stateful Logic in RRAM: Theoretical Analysis and Arithmetic Design," in *Application-specific Systems, Architectures and Processors*. IEEE, 2019.

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

[10] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *Int'l Symp. on Circuits and Systems (ISCAS)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

[11] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta, "A scalable in-memory logic synthesis approach using memristor crossbar," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 2, pp. 355–366, 2018.

[12] D. N. Yadav, P. L. Thangkhiew, and K. Datta, "Look-ahead mapping of boolean functions in memristive crossbar array," *Integration*, vol. 64, pp. 152–162, 2019.

[13] A. Zulehner, K. Datta, I. Sengupta, and R. Wille, "A staircase structure for scalable and efficient synthesis of memristor-aided logic," in *Asia and South Pacific Design Automation Conf. (ASP-DAC)*. ACM, 2019, pp. 237–242.

[14] V. Tenace, R. G. Rizzo, D. Bhattacharjee, A. Chattopadhyay, and A. Calimera, "Said: A supergate-aided logic synthesis flow for memristive crossbars," in *Design, Automation, and Test in Europe (DATE)*. IEEE, 2019, pp. 372–377.

[15] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2019.

[16] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.

[17] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky, "SIMPLE MAGIC: Synthesis and In-memory MaPping of Logic Execution for Memristor-Aided loGIC," in *Int'l Conf. on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 225–232.

[18] K. McElvain, "Iwls'93 benchmark set: Version 4.0," in *Distributed as part of the MCNC International Workshop on Logic Synthesis' 93 benchmark distribution*, 1993, pp. 1–6.

[19] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, "IMAGING-In-Memory AlGorithms for Image processiNG," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 12, pp. 4258–4271, 2018.

[20] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conf. (DAC)*, 2016, pp. 1–6.

[21] S.-S. Sheu, M.-F. Chang, K.-F. Lin, C.-W. Wu, Y.-S. Chen, P.-F. Chiu, C.-C. Kuo, Y.-S. Yang, P.-C. Chiang, W.-P. Lin *et al.*, "A 4Mb embedded SLC resistive-RAM macro with 7.2 ns read-write random-access time and 160ns MLC-access capability," in *Int'l Solid-State Circuits Conf. (ISSCC)*, 2011.

[22] L. Xu, L. Bao, T. Zhang, K. Yang, Y. Cai, Y. Yang, and R. Huang, "Nonvolatile memristor as a new platform for non-von Neumann computing," in *Int'l Conf. on Solid-State and Integrated Circuit Technology (ICSICT)*, 2018.

[23] B. L. SYNTHESIS, "Abc: a system for sequential synthesis and verification, release 70930," 2007.

[24] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 476–488.

[25] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981.

[26] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Design Automation Conf. (DAC)*. ACM, 2017, p. 6.
[27] N. Talati, A. H. Ali, R. B. Hur, N. Wald, R. Ronen, P.-E. Gaillardon, and S. Kvatinsky, "Practical challenges in delivering the promises of real processing-in-memory machines," in *Design, Automation, and Test in Europe (DATE)*. IEEE, 2018, pp. 1628–1633.
[28] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," in *Int'l Symp. on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5.
[29] S. Angizi, Z. He, and D. Fan, "Pima-logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation," in *Design Automation Conf. (DAC)*. ACM, 2018, p. 162.
[30] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, 2016.
[31] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Int'l Symp. on Computer Architecture (ISCA)*. IEEE Press, 2016, pp. 14–26.
[32] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 541–552.
[33] X. Sun, S. Yin, X. Peng, R. Liu, J.-s. Seo, and S. Yu, "Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks," in *Design, Automation, and Test in Europe (DATE)*. IEEE, 2018, pp. 1423–1428.
[34] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu, and H. Jiang, "A spiking neuromorphic design with resistive crossbar," in *Design Automation Conf. (DAC)*, 2015, pp. 1–6.

Feng Wang Feng Wang is a PhD student in the Department of Computer Science at Peking University. His research interests include NVM and PIM. Wang has a BS in computer science from Peking University. Contact him at yzwangfeng@pku.edu.cn.

Guojie Luo Guojie Luo is an associate professor in the Department of Computer Science at Peking University. His current research interests include electronic design automation, heterogeneous computing with FPGAs and emerging devices, and medical imaging analytics. Luo received a PhD in Computer Science from the University of California, Los Angeles. Contact him at gluo@pku.edu.cn.

Guangyu Sun Guangyu Sun is an associate professor in the Department of Computer Science at Peking University. His current research interests include energy-efficient memory architectures, storage system optimization for new devices and acceleration systems for deep learning applications. Sun received a PhD in Computer Science from The Pennsylvania State University. Contact him at gsun@pku.edu.cn.

Jiaxi Zhang Jiaxi Zhang is a PhD student in the Department of Computer Science at Peking University. His research interests include EDA and FPGA acceleration. Zhang has a BS in microelectronics from Peking University. Contact him at zhangjiaxi@pku.edu.cn.

Jinfeng Kang Jinfeng Kang is now a Full Professor of Electronics Engineering Computer Science School in Peking University. His research interest is to explore novel device concepts, structures, materials, circuits, and the system architectures for the applications of future computing and data storage systems. His accomplishments focus on the RRAM technology and applications. Contact him at kangjf@pku.edu.cn.

Yuhao Wang Yuhao Wang is a Ph.D. from Nanyang Technological University, Singapore. He is now with Alibaba Damo Academy as a research scientist. His research interests mainly lie in the intersection of emerging non-volatile memory, processing in memory architecture and hardware/algorithm co-designs. He has authored or co-authored two monographs, 16 scientific publications and holds 2 US patents. Contact him at yuhao.w@alibaba-inc.com.

Dimin Niu Dimin Niu is a research scientist in the Computing Technology Lab of Alibaba DAMO Academy. Prior to joining Alibaba, he was a staff memory architecture in Memory Solutions Lab at Samsung Semiconductor Inc. His current research interests include computer architecture, memory architecture, storage system, process-in-memory, and domain specific architectures. Niu received a PhD in Computer Science from the Pennsylvania State University. Contact him at dimin.niu@alibaba-inc.com.

Hongzhong Zheng Hongzhong Zheng is a leading research scientist in field of new memory and storage system, processing in memory technology and architecture in the Computing Technology Lab of Alibaba DAMO Academy. Prior to joining Alibaba, he was a director in Memory Solutions Lab at Samsung Semiconductor Inc. His current research interests include computer architecture, memory architecture, storage system, process-in-memory, and domain specific architectures. Mr. Zheng received a PhD in Computer Engineering from the University of Illinois at Chicago. Contact him at hongzhong.zheng@alibaba-inc.com.