

Parallel Stateful Logic in RRAM: Theoretical Analysis and Arithmetic Design

(Invited Paper)

Feng Wang^{1,*}, Guojie Luo^{1,#}, Guangyu Sun¹, Jiayi Zhang¹, Peng Huang², Jinfeng Kang²

¹Center for Energy-efficient Computing and Applications, Peking University, Beijing, China

²Institute of Microelectronics, Peking University, Beijing, China

Email: {*yzwangfeng, #gluo}@pku.edu.cn

Abstract—Processing-in-memory (PIM) provides massive parallelism with high energy efficiency and becomes a promising solution to the “memory wall” problem. Recently, the emerging metal-oxide resistive random access memory (RRAM) has shown its potential to design a PIM architecture. Several stateful logic operations, *e.g.*, NOR and NAND, can be executed in parallel in an RRAM crossbar. Although previous works have designed some algorithms using the stateful logic, it is still under exploration how to fully exploit its potential high parallelism and design an asymptotically fast algorithm for a given function.

In this work, we theoretically analyze the parallelism in an RRAM crossbar and design several asymptotically optimal arithmetic algorithms. In detail, we first propose the Single Instruction Multiple Lines (SIML) model to unify the stateful logic families and prove three lower bounds on the time complexity of a parallel RRAM algorithm. Then, we design three algorithms for integer addition functions with the stateful logic, guided by the lower bound analysis. All of them reach the time complexity lower bound. Finally, we make two extensions of the integer addition algorithms, supporting multiplication functions by decomposing them to additions and supporting the flex-point data type by proposing an exponent and mantissa update flow. Experimental evaluation shows that our integer algorithms achieves a speedup up to 13.79x over the previous RRAM algorithms. Our flex-point implementation achieves a 26.60x speedup and saves 73.68% energy compared to an ARM.

I. INTRODUCTION

In the conventional CMOS-based von Neumann architecture, both programs and data are held in memory. The processor and memory are separate, and data moves between the two. In this type of architecture, huge energy-hungry data transfer between memory and processing units has been the limitation of computation speed and energy efficiency, *i.e.*, the von Neumann bottleneck [1]. Aiming at breaking this bottleneck, processing-in-memory (PIM) architectures and related devices are receiving widespread research interest.

The emerging metal-oxide resistive random access memory (RRAM) is one type of non-volatile memories (NVM) [2]. It has been shown a strong candidate to implement a PIM architecture for two reasons. First, it can be used for low-cost storage due to its high density, excellent scalability, and low power. Industrial demonstrations have been presented [3] to showcase the viability of large memory crossbars. Second, RRAM has the capability to perform stateful logic opera-

tions [4] beyond storage. It is possible to combine computation and storage in the same RRAM crossbar due to its flexibility.

In a single-level-cell (SLC) 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. If several RRAM cells are connected in series, their states can be affected by others in certain conditions. 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].

If the input and the output RRAM cells are aligned along row (or column) positions, we can implement multiple stateful logic operations along different columns (or rows) simultaneously by applying the same voltage pattern [8]. 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.

Due to the promising possibility of massive parallelism, previous works have designed several arithmetic functions, *e.g.*, addition [9] and multiplication [10], [11], using stateful logic in RRAM. However, these works lack in two aspects. First, they do not theoretically analyze the parallel computation capability of the RRAM crossbar and only optimize the operations ad hocly. Thus, most of their designs cannot get the optimal time complexity. Second, most of these works only target the integer or fixed-point data type, which does not support the real number functions well and limits the scope of their applicability.

In this work, we address these two problems with the following contributions:

- We propose a uniform SIML model for stateful logic families in RRAM. Based on this model, we prove three lower bounds on the time complexity of a parallel RRAM algorithm. The time complexity is determined by both the data layout and the logic operations.

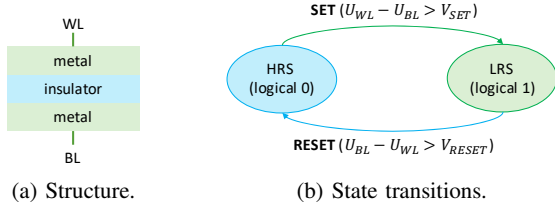


Fig. 1: Schematic of an RRAM cell.

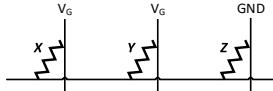


Fig. 2: A NOR operation $Z = \text{NOR}(X, Y)$. RRAM Z is initialized to LRS. V_G satisfies $V_G > 2V_{\text{RESET}}$. Z will be reset to HRS if X or Y stays at LRS.

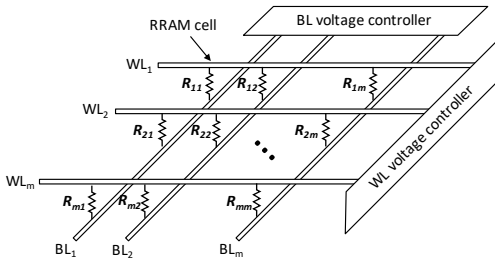


Fig. 3: Parallel NOR operations in a crossbar. We can execute WL operations $R_{im} = \text{NOR}(R_{i1}, R_{i2})$ or BL operations $R_{mi} = \text{NOR}(R_{1i}, R_{2i})$ for $1 \leq i \leq m$ in parallel.

- We design three parallel RRAM adders: a ripple carry adder, a carry select adder, and a carry save adder, for the integer addition functions guided by the lower bound analysis. We prove that these algorithms reach the time complexity lower bound.
- We make two extensions of the integer addition algorithms. The first is supporting multiplication functions by decomposing them to several additions. The second is supporting the flex-point data type by proposing a flow to update the exponent and the mantissa part simultaneously.
- We experimentally evaluate our integer addition and multiplication algorithms and show a speedup up to $13.79\times$ over previous RRAM algorithms. Our flex-point implementation achieves a $26.60\times$ speedup and saves 73.68% energy compared to an ARM processor.

II. BACKGROUND

A. RRAM Resistance for Representing Logic Values

An RRAM cell [12] has a simple metal-insulator-metal sandwich structure with two terminals connecting to the word line (WL) and the bit line (BL), respectively, as shown in Fig. 1a. Its internal resistance has two states, the low resistance state (LRS) and the high resistance state (HRS), which can be switched mutually at certain conditions, as summarized in a state machine in Fig. 1b. When applying a positive voltage which is larger than V_{SET} , RRAM cells can be switched

TABLE I: Stateful logic families.

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

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. Usually, we define HRS as logical 0 and LRS as logical 1. Under this definition, SET and RESET implement the logic operations $Y=1$ and $Y=0$, respectively.

B. RRAM State Switching as Primitive Logic Operations

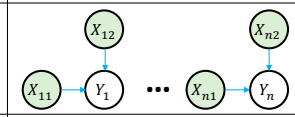
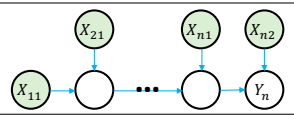
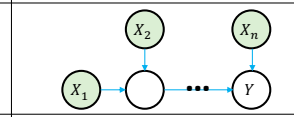
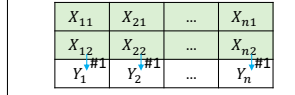
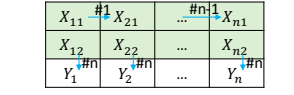
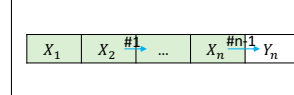
Stateful logic, where both the inputs and outputs of a logic gate are the RRAM resistive states, is one of the processing-in-RRAM techniques. Fig. 2 shows the schematic of Memristor-Aided loGIC (MAGIC) [5], a widely-used stateful logic family. In this example, we apply a voltage pulse of V_G , V_G , and GND on one end of cells X , Y , and Z , respectively and connect their other ends. When we initialize Z to LRS and set $V_G > 2V_{\text{RESET}}$, we perform a NOR operation $Z = \text{NOR}(X, Y)$.

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 $V_G/2 > V_{\text{RESET}}$ 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.

Fig. 3 shows the schematic of NOR performed over rows and columns within a symmetric RRAM crossbar. The m wires at the top are WLs and the m ones at the bottom 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 rows or columns [8].

C. RRAM-based Stateful Logic Families

Previous works have demonstrated some other stateful logic families, which are summarized in Table I. 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 [13]. Despite the differences, all support parallel execution across multiple WLs or BLs in a symmetric RRAM crossbar if the inputs and outputs are aligned. 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.

	(a) Theorem 1	(b) Theorem 2	(c) Corollary 1	(d) Theorem 3
Condition	bitwise functions	most arithmetic functions	square layout	a given algorithm
Parallelism upper bound	$\max(w, h)$	$O\left(\frac{T}{w+h}\right)$	$O\left(\frac{T}{\sqrt{wh}}\right)$	$O\left(\frac{T}{len_{max}}\right)$
Time complexity lower bound	ordinary bound: $O\left(\frac{T}{\max(w,h)}\right)$	shape bound: $O(w+h)$	function bound: $O(\sqrt{wh})$	algorithm bound: len_{max}
Example function	$Y_i = X_{i1} \text{ NOR } X_{i2} (i = 1, 2, \dots, n)$	$Y_i = \text{NOR}_{k=1}^i X_{ki} \text{ NOR } X_{i2} (i = 1, 2, \dots, n)$	—	$Y = \text{NOR}_{k=1}^n X_{k1}$
Example algorithm (netlist)			—	
Example layout in RRAM			—	

$O(T)$: total cycles in the series implementation, w : width of layout (# of BLs), h : height of layout (# of WLs), len_{max} : length of the critical path.

Fig. 4: A diagram for four types of the time complexity lower bound of a parallel RRAM algorithm in the SIML computation model. The three example functions and their algorithms (netlists) reach the corresponding lower bound.

III. THEORETICAL ANALYSIS

This section first proposes a model to unify different stateful logic families and then proves three lower bounds of the time complexity for this model.

A. SIML Computation Model

We propose a uniform Single Instruction Multiple Lines (SIML) model for all of the stateful logic families summarized in TABLE I. There are mainly four assumptions in this model:

- The latency of a single stateful logic operation in TABLE I is identical. And it is independent of the input number and the distances among input and output lines.
- The input number of a single operation cannot exceed a constant $input_{max}$ due to some physical constraints. $input_{max}$ is usually less than 10 in the literature.
- In an RRAM crossbar, the latency of WL and BL operations is identical. And it is independent of the degree of parallelism and the distances among active lines.
- In an RRAM crossbar, the degree of parallelism for any operation can reach the crossbar size, and the crossbar size scales with the problem size.

The former two assumptions are for a single stateful logic operation, and the latter two are related to the parallel operations. The time complexity in this model is only dominated by the function and the algorithm design but not the crossbar size. We will discuss the relationship between the crossbar size and the time complexity in the experimental section. This computation model runs in a SIML fashion, a two-dimensional Single Instruction Multiple Data (SIMD) execution. Compared to the CMOS-based von Neumann architecture, the degree of parallelism in this model can scale with the data size. Moreover, the parallelism improvement costs only energy but not hardware.

All of the stateful logic families in TABLE I satisfy the four assumptions. They only have two differences in this model, both of which have no effect on the order of the time complexity. First, they propose different operations executed

in one voltage pulse. Nevertheless, we can implement a logic family with another logic family in finite pulses due to its completeness. For example, we can implement XOR, the most complex operation proposed by Xu *et al.* [13], using MAGIC [5] in six voltage pulses, and thus, Xu *et al.*'s work is at most six times faster than MAGIC. Second, they have different $input_{max}$'s. It is not hard to prove that an $input_{max}$ -input operation can be replaced by no more than $input_{max} - 1$ two-input operations. Since $input_{max}$ is a constant in our model, a larger $input_{max}$ cannot affect the time complexity order, either. Without loss of generality, we design arithmetic algorithms using two-input MAGIC operations in this work.

B. Lower Bounds of the Time Complexity

We prove three lower bounds concerning the time complexity of the parallel RRAM algorithms in this SIML computation model, as summarized in Fig. 4. For a given logic function, we assume that the serial RRAM implementation takes $O(T)$ voltage pulses. We can design a parallel RRAM algorithm for this function, and the computation in the RRAM crossbar occupies a w BLs \times h WLs rectangular layout.

Theorem 1. Ordinary lower bound. *The time complexity lower bound of the parallel RRAM algorithm is $O\left(\frac{T}{\max(w,h)}\right)$.*

Proof. In a single voltage pulse, at most $\max(w, h)$ operations can be executed in parallel, with w BL operations or h WL operations. When the maximal parallel operations are executed in every step, the time complexity is $O\left(\frac{T}{\max(w,h)}\right)$. \square

Theorem 1 gives the ordinary time complexity lower bound. Reaching this lower bound requires parallel operations along a single direction and negligible data copy overhead; thus, only a bitwise function with aligned data placement can be a tight instance.

Theorem 2. Shape lower bound. *If the inputs are stored in an $O(w) \times O(h)$ rectangular area, and the outputs are stored in an $o(w) \times o(h)$ rectangular area, the time complexity lower bound of a parallel RRAM algorithm is $O(w+h)$, assuming*

that there are no useless inputs, i.e., every input affects at least one output.

Proof. Input data in at least $O(w) - o(w) = O(w)$ BLs need to affect an output in the different BL, which can be realized only through a WL operation. Note that a WL operation contains constant BLs, there are at least $O(w)$ voltage pulses for parallel WL operations. BL operations can be analyzed in the same way. The total time complexity is at least $O(w+h)$. \square

In particular, Theorem 2 proves a stronger conclusion than Theorem 1 when $T = O(wh)$ and the *ordinary lower bound* is $O(\min(w, h))$. We can infer that the time complexity lower bound still holds if there are more outputs. For example, in Fig. 4b, computing Y_n requires the inputs X_{11} to X_{n1} , which occupy an $n \times 1$ area, so the time complexity is $O(n)$. Computing all Y_i 's still takes at least $O(n)$ voltage pulses. Most arithmetic functions can satisfy the conditions in this theorem because the most significant bit of the output is usually related to all of the input bits.

The input and output data size for the given function is a constant, i.e., wh is given. As a result, it is better to constraint the computation in a square area. We have the following corollary on a shape irrelevant lower bound:

Corollary 1. Function lower bound. *Under the conditions specified in Theorem 2, $O(w+h)$ is further lower bounded by $O(\sqrt{wh})$ for any input shape. A tight instance for the lower bound $O(\sqrt{wh})$ can be attained only if $w = \Theta(h)$.*

These two theorems prove the time complexity lower bound dominated by the size and the shape of the layout. Besides, different parallel RRAM algorithms can also lead to different time complexity, and we have the following lower bound.

Theorem 3. Algorithm lower bound. *The algorithm corresponds to a netlist, which can be regarded as a directed acyclic graph $G = \langle V, E \rangle$, in which V is the set of variables, and E represents the MAGIC operations. If $Y = \text{NOR}(A, B)$ exists in the algorithm, there is an edge from A, B to Y in G , respectively. If the length of the critical path, i.e., the longest path from an input to an output, in G is len_{\max} , the time complexity lower bound is $O(\text{len}_{\max})$.*

Proof. Computing the output in the critical path requires len_{\max} operations, which have to be executed in series. As a result, the time complexity lower bound is $O(\text{len}_{\max})$. \square

Fig. 4d shows an extreme case. The critical path is from the input x_1 to the output Y , and the whole computation procedure is inherently sequential. In fact, we can get a lower time complexity by reducing the inputs with a NOR tree. This theorem tells us that if we want to reach the lower bound proved in Theorem 1 and 2, we need to select an algorithm with an equal or lower time complexity.

IV. INTEGER ADDITION

According to the lower bound analysis in the last section, to design an efficient parallel RRAM algorithm, we first determine the data layout and then select a suitable algorithm.

This section proposes three parallel algorithms for integer addition, as shown in Fig. 5. The ‘‘ripple carry adder’’ and

TABLE II: One-bit full adder. $A + B + C_i = (C_o, S)$.

Pulse	Logic operation
1	$T_1 = \text{NOR}(A, B)$
2	$T_2 = \text{NOR}(A, T_1)$
3	$T_3 = \text{NOR}(B, T_1)$
4	$T_4 = \text{NOR}(T_2, T_3)$
5	$T_5 = \text{NOR}(T_4, C_i)$
6	$C_o = \text{NOR}(T_1, T_5)$
7	$T_6 = \text{NOR}(T_4, T_5)$
8	$T_7 = \text{NOR}(T_5, C_i)$
9	$S = \text{NOR}(T_6, T_7)$

‘‘carry select adder’’ are two parallel RRAM algorithms for the addition of two n -bit integers. And the ‘‘carry save adder’’ is a parallel RRAM algorithm for the addition of M n -bit integers. We prove that these algorithms attain some of the lower bounds.

A. Ripple Carry Adder

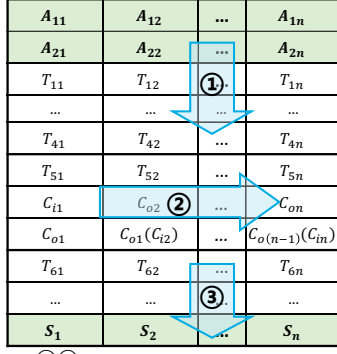
Fig. 5a shows the ripple carry adder, which is extended from the one-bit full adder, for integers stored in two WLs. As shown in Table II, the computation procedure of a one-bit full adder takes 9 voltage pulses and 7 temporary RRAM cells (T_1 to T_7). The 5th and 6th steps generate carries and they need to be executed in series, while the other steps can be executed in parallel among different bits. Thus, it takes about $3n+7$ voltage pulses to add two n -bit integers, in which the coefficient 3 is from the 5th, the 6th, and the carry propagation pulses.

Our ripple carry adder algorithm reaches the *shape lower bound* and the *algorithm lower bound*. On the one hand, if the input integers are stored in an $O(n) \times O(1)$ layout, we can infer from Theorem 2 that the *shape lower bound* is $O(n)$. On the other hand, the critical path in the ripple carry adder is from A_{11}, A_{21} to S_n , so the *algorithm lower bound* is no less than its length $O(n)$.

B. Carry Select Adder

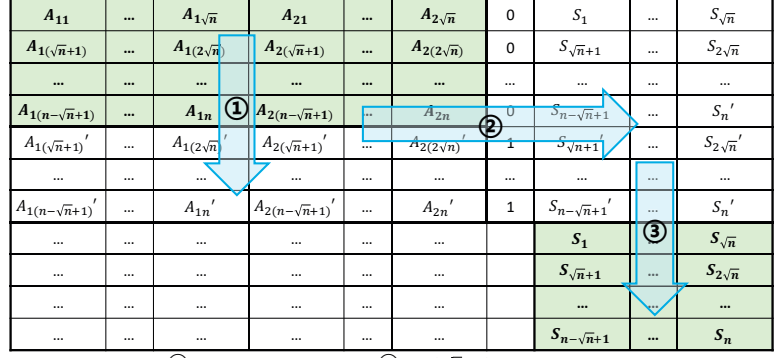
To further reduce the time complexity, considering that the input and output data in the addition occupy $O(n)$ cells, i.e., $wh \geq O(n)$, we set $w = \Theta(h) = \Theta(\sqrt{n})$ to reach the *function lower bound* $O(\sqrt{n})$. Meanwhile, according to Theorem 3, the $O(n)$ ripple carry adder is not qualified, so we use the faster carry select adder. The key point of the carry select adder is to divide the integer into several segments and add the more-significant segments by trying both possible carry-ins.

Fig. 5b shows our carry select adder. For simplicity's sake, we assume that n is a square number, and each input integer occupies a $\sqrt{n} \times \sqrt{n}$ area; otherwise, we store the input in a rectangular area close to a square. The computation procedure consists of three stages. First, we copy the whole input except the first WL, which takes one voltage pulse per WL and $\sqrt{n}-1$ pulses in total. Second, we add each WL in parallel. We assign the carry-in 0 for the top \sqrt{n} WLs and 1 for the bottom $\sqrt{n}-1$ WLs. This stage performs $2\sqrt{n}-1$ \sqrt{n} -bit addition in parallel and takes $O(\sqrt{n})$ pulses. Third, we select the correct output of the more-significant segments according to the carry-out of



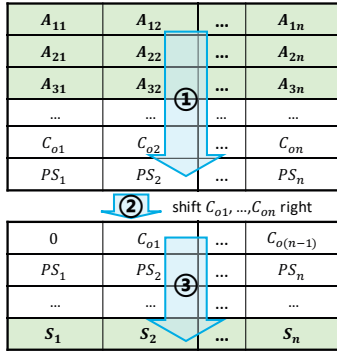
① ③ add n BLs in parallel
② generate and propagate carries in series

(a) Ripple carry adder $A_1 + A_2 = S$.



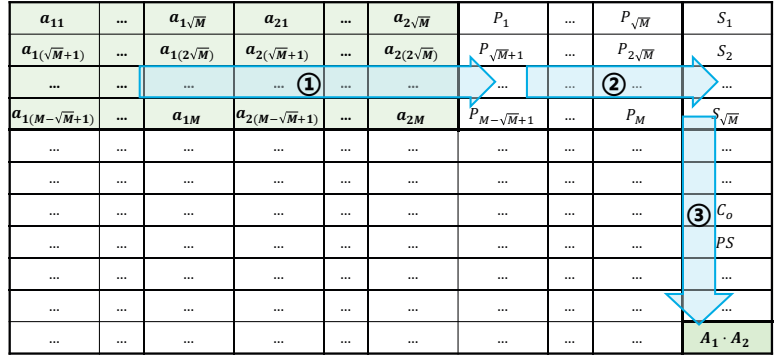
① copy the input ② add $2\sqrt{n} - 1$ WLs in parallel
③ select the correct result

(b) Carry select adder $A_1 + A_2 = S$.



① add n BLs in parallel
② shift C_{01}, \dots, C_{0n} right
③ add the last two addends C_o, PS

(c) Carry save adder $A_1 + A_2 + A_3 = S$.



① element-wise multiplication ② accumulate \sqrt{M} WLs in parallel
③ accumulate S_k s using the carry save adder

(d) Dot product $A_1 \cdot A_2$. $A_1 = (a_{11}, \dots, a_{1M})$.

Fig. 5: Four parallel RRAM algorithms for the integer arithmetic functions. The bit width is n . Each cell represents a single RRAM cell in (a)-(c) and n cells in (d). The input and output data are stored in the green area, and the computation procedure is labelled by blue arrows. All of the parallelism reaches the time complexity lower bound, as summarized in TABLE III.

TABLE III: Time complexity and their corresponding lower bound type of the integer arithmetic algorithms.

Function	Algorithm	Layout	Time complexity	Lower bound type
two n -bit integer addition	ripple carry adder	$O(n) \times O(1)$	$O(n)$	shape / algorithm bound
two n -bit integer addition	carry select adder	$O(\sqrt{n}) \times O(\sqrt{n})$	$O(\sqrt{n})$	function / algorithm bound
M integer addition	carry save adder	$O(\sqrt{M}) \times O(\sqrt{M})$	$O(\sqrt{M})$	function bound
M -dimensional dot product	–	$O(\sqrt{M}) \times O(\sqrt{M})$	$O(\sqrt{M})$	function bound

the less-significant segments, which takes $O(1)$ pulses per WL and $O(\sqrt{n})$ pulses in total. The time complexity of the carry select adder reaches the *function lower bound* $O(\sqrt{n})$.

C. Carry Save Adder

We also propose an algorithm for the addition of multiple integers. Considering that the time complexity is mainly dominated by the integer number M , we assume that the bit width n is a constant. For simplicity, we also assume that M is a square number.

According to Corollary 1, we store the integers in an $n\sqrt{M} \times \sqrt{M}$ square area, with \sqrt{M} integers in each WL, to reach the *function lower bound*. We first accumulate each WL in parallel, and the time complexity is $O(\sqrt{M})$. Then, we accumulate the sums in the same BLs. The carry generation

and propagation steps are the bottleneck in the ripple carry adder, so we exploit another adder similar to a carry save adder, which consists of three steps:

- 1) Add n BLs (or bits) in parallel. The carries are stored in original BLs temporarily.
- 2) Shift carries to the right for a certain length. If there are more than two addends, go to Step 1).
- 3) Add the last two addends using a ripple carry adder.

Step 2) aligns the carries so they can be added again. Fig. 5c gives an example of three addends. First, n BLs are added in parallel to generate carries (C_{ok}) and partial sums (PS_k). Then the carry WL is shifted right for one bit and the two-integer ripple carry addition is performed on partial sums and shifted carries. All operations in the procedure can be highly parallel except the shift, which takes $O(n)$ pulses for an integer.

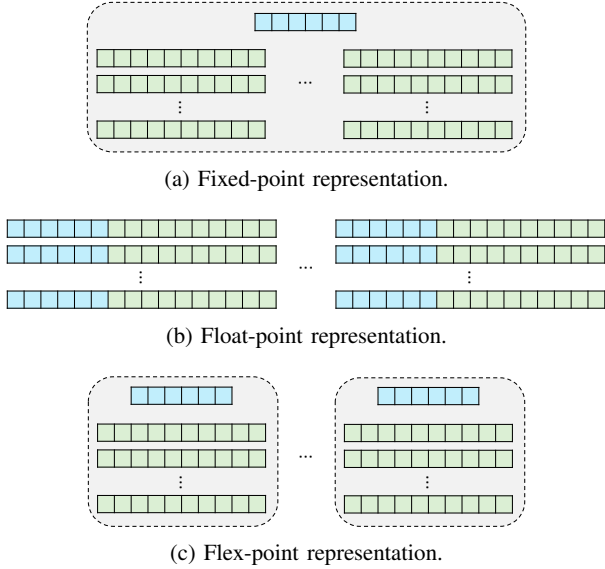


Fig. 6: Bit representations of three real data types. Green and blue cells signify the mantissa and exponent bits, respectively.

After one iteration, \sqrt{M} addends are reduced to $\log \sqrt{M}$ addends with $\log \sqrt{M}$ right shift operations. To accumulate all addends, the number of shift operations is $\log \sqrt{M} + \log \log \sqrt{M} + \dots = O(\log \sqrt{M})$, and it takes $O(n \log \sqrt{M})$ pulses. As a result, the time complexity of the carry save adder is $O(\sqrt{M} + n \log \sqrt{M}) = O(\sqrt{M})$. The total complexity of M -integer addition reaches its *function lower bound* $O(\sqrt{M})$ proved by Corollary 1.

V. EXTENSION

A. Multiplication Extension

Due to the high parallelism of the integer addition, we can implement some multiplication functions, *e.g.*, dot product, vector-matrix multiplication, matrix-matrix multiplication, and Hadamard product, by decomposing them to series of additions. To achieve a low time complexity, the key idea is to store the input in a squared area and utilize a fastest adder, *e.g.*, the carry select adder or the carry save adder.

Fig. 5d gives an example of the M -dimension dot product. Both input vectors occupy an $n\sqrt{M} \times \sqrt{M}$ square area. We first perform the element-wise multiplication in parallel. Two n -bit integer multiplication can be decomposed to n n -bit integer additions and takes $O(n^2)$ voltage pulses in series. It takes $O(n^2\sqrt{M})$ pulses to multiply all elements. Then, we accumulate the M products P_1 to P_M as described in the previous section, *i.e.*, accumulate all WLS to S_1 to $S_{\sqrt{M}}$, and accumulate S_k 's using the carry save adder, which takes $O(\sqrt{M})$ pulses. We still consider n a constant, and the time complexity of the whole algorithm is $O(\sqrt{M})$, which also reaches the *function lower bound* proved by Corollary 1.

B. Flex-point Extension

The real number is employed more widely than the integer in actual applications, so this section extends the proposed

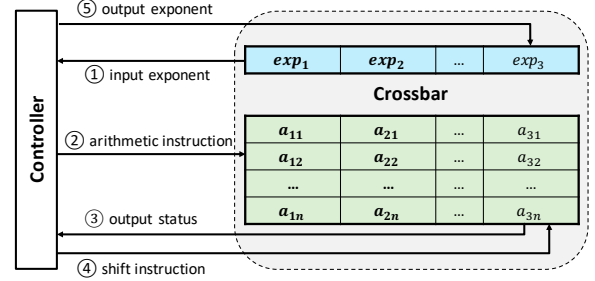


Fig. 7: Implementation of a flex-point function across vectors.

integer functions to adapt the real number applications.

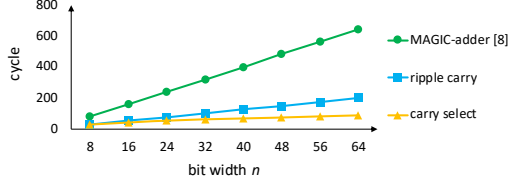
Fig. 6 illustrates three common real data types. The fixed-point type (Fig. 6a) shares a common exponent among all of the numbers, which leads to a significant precision loss when the data vary in a wide range. The float-point type (Fig. 6b) assigns an exponent for each number to improve the precision, but it occupies much more area and has higher computation complexity. Different from them, the flex-point [14] (Fig. 6c) is a data type that combines their advantages. By using a common exponent for integer values in a vector, flex-point reduces computational and memory requirements simultaneously. As a result, we select the flex-point data type to represent the real number in the RRAM crossbar.

To fully exploit the two-dimensional parallelism in the SIML model, we set the vector length of the flex-point type to the crossbar size. That is to say, numbers in the same BLs share the same exponent, while numbers in the same WL can have different exponents. As shown in Fig. 7, $\{a_{ij} | j \in [1, n]\}$ are mantissas and share the exponent exp_i . The exponents occupy the first WL in the crossbar. We only update an exponent when its corresponding vector is created or modified by the other vectors.

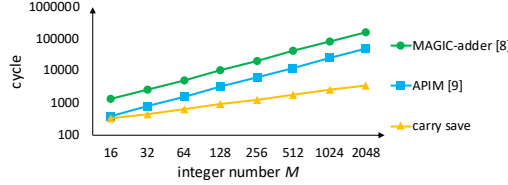
Numbers in the same BLs can be regarded as the fixed-point type. The fixed-point addition is the same as the integer one, but the fixed-point multiplication is a little different. In order to preserve a faithful representation, numbers with a shared exponent must have a sufficiently narrow dynamic range such that mantissa bits alone can encode variability. However, if the exponent is not equal to 1, the product needs a different exponent from the inputs. Therefore, we do not allow multiplication in the same vector.

Fig. 7 shows the implementation of a flex-point function across numbers in multiple vectors. The controller outside the crossbar not only dispatches instructions (voltage patterns) to the crossbar as in the integer function implementation but also participates in the computation by updating the output exponent. In detail, the implementation consists of five steps:

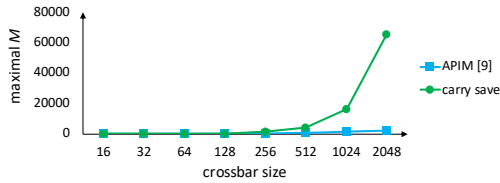
- 1) Read the input exponents to the controller. The controller examines whether the output will overflow.
- 2) Send arithmetic instructions to the mantissas in the crossbar and compute the output mantissa. This step preserves full precision of the output.
- 3) Check the output status to find the most significant



(a) Two n -bit integer addition.



(b) M 8-bit integer addition.



(c) The maximal data size M in one crossbar.

Fig. 8: Comparison of two types of the integer addition.

bit that differentiate the mantissas. This check can be parallelized among different BLs using a bitwise OR and AND operation [15] and takes $O(n)$ pulses.

- 4) Send the left shift instructions to the output mantissa. The bits more significant than the bit found in Step 3) are useless and have to be erased. Shift operations can be parallelized among multiple WLs and take $O(n)$ pulses.
- 5) Compute the output exponent according to the function type, the input exponents, and the left shift length.

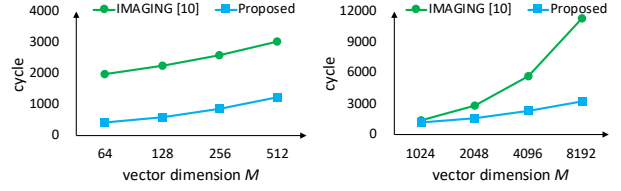
Step 3) to 5) dynamically adjusts the output exponent to minimize overflows of the output mantissas and maximize the available dynamic range. The CMOS-based controller is faster than the RRAM, so its latency overhead can be ignored. The total time complexity is dominated by Step 2), the mantissa computation.

VI. EXPERIMENTAL EVALUATION

A. Addition Algorithm Evaluation

We compare our two n -bit integer addition algorithms with the MAGIC-adder [8] using one RRAM crossbar, as shown in Fig. 8. Our ripple carry adder achieves a $3.09\times$ speedup on average. Both this adder and the previous work have the same time complexity $O(n)$, and the speedup mainly comes from our parallel computation among different bits in Step ① and Step ③ (see Fig. 5a).

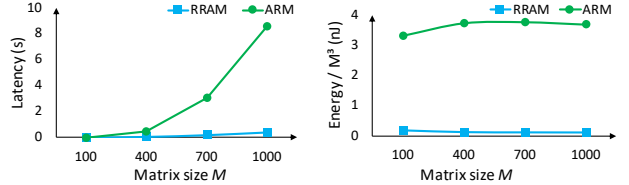
Our carry select adder achieves a $5.05\times$ speedup on average and is the fastest adder. The ripple carry adder has advantages when n is sufficiently small, *i.e.*, no more than 8, because the copy and selection overhead in the carry select adder cannot be ignored. However, the speedup of the carry select adder



(a) Dot product.

(b) Hadamard product.

Fig. 9: Comparison of two integer multiplication kernels.



(a) Latency.

(b) Energy per bit.

Fig. 10: A comparison between RRAM and CPU on square matrix multiplication using the flex-point data type.

continues to improve with the integer bit width and reaches $7.3\times$ when $n = 64$ due to its lower time complexity $O(\sqrt{n})$.

Fig. 8b compares our carry save adder with two previous MAGIC-based algorithms in the M integer addition. We assume that the crossbar size is infinite so the latency is only dominated by the algorithm itself. Our work achieves a $13.79\times$ and a $4.15\times$ speedup compared to MAGIC-adder [8] and APIM [9]. Both of the two previous works have the $O(M)$ time complexity, which is higher than our $O(\sqrt{M})$ complexity. As a result, the speedup of our carry save adder also improves with the data size.

Fig. 8c takes the crossbar size into consideration. APIM places the input data in the same WL or BL, so the parallelism reaches the upper bound, *i.e.*, the crossbar size, when the data size reaches the crossbar size. Different from that, our carry save adder places the input in a square area, and the maximal M in one crossbar is much higher than that in APIM. As a result, our work has higher area efficiency and throughput for a given crossbar size.

B. Multiplication Algorithm Evaluation

We evaluate our multiplication implementation using two image processing kernels, the dot product and the Hadamard product. We compare our algorithms with the state-of-the-art MAGIC-based image processing accelerator IMAGING [11], as shown in Fig. 9. Our work achieve a $3.36\times$ and a $2.07\times$ speedup on two benchmarks, respectively. IMAGING implements these two benchmarks both using a $O(M)$ algorithm, while we can design $O(\sqrt{M})$ algorithms. Similar to our analysis on M integer addition, we can place much more data in a crossbar and have higher area efficiency and throughput due to the square placement scheme.

C. Flex-point Support Evaluation

We evaluate our flex-point support by comparing our work with an ARM-v8 architecture that runs at 2 GHz with two 32

TABLE IV: RRAM configuration parameters.

Parameter	Value
Technology node	65 nm
Crossbar size	512×512
Frequency (#pulses/s)	769 MHz [5]
Energy per MAGIC operation	34 fJ [8]

KB L1 caches and a 2 MB L2 cache using the square matrix multiplication as a benchmark. The performance of this ARM is simulated by Gem5 [16]. We evaluate the performance of RRAM according to the parameters listed in Table IV. The power of the controller is obtained by synthesizing with Open Cell Library [17] using Synopsys Design Compiler. We write a simulator to estimate the overall performance from these data.

According to Fig. 10a, our work achieves a 26.60× speedup on average. The algorithm running in ARM is in serial, and the time complexity is $O(M^3)$. However, the time complexity of the parallel algorithm in RRAM can reach $O(M^2)$. In fact, it is time-consuming for RRAM to implement simple functions. An ARM can execute an integer-level operation per cycle while RRAM can only execute a bit-level operation. We make full use of the parallelism in RRAM to make up the disadvantage.

Fig. 10b illustrates the high energy efficiency of RRAM. Our work saves 73.68% energy compared to the ARM. The Y-axis measures the unit energy consumption to some degree, since matrix multiplication is an $O(M^3)$ problem. It is nearly a constant for a given hardware. For RRAM, most of the computation is concentrated in the crossbar, and the energy consumed by the controller only accounts for less than 10%.

VII. RELATED WORK

There are lots of efforts on RRAM-based stateful logic. IMP [4] is the earliest one. Several Boolean logic operations, including NOR [5], NAND [6], OR [7], and XOR [13], are also implemented after then. Several studies [18], [8], [9], [10] optimize the addition and multiplication manually. They implement some more complex applications in image processing [11] based on these arithmetic functions. However, they do not fully utilize the parallel potential of RRAM and thus do not achieve the optimal result.

Some studies perform computation on RRAM without stateful logic. For example, Pinatubo [15] implements bulk bitwise operations by redesigning the read circuitry. Besides digital fashion, some works [19], [20] accelerate neural networks acceleration in the analog fashion. Despite low latency, it lacks in accuracy and suffers from high variation, which restricts its scope of applications. Moreover, power consumption from additional AD-conversion and I/Os cannot be ignored [21].

VIII. CONCLUSION

In this work, we propose the SIML computation model for stateful logic in RRAM and prove three lower bounds on the time complexity. Under the guide of such analysis, we design three integer addition algorithms and prove that all of them

reach the lower bound. Moreover, We make two extensions of our algorithms to support multiplication and flex-point functions and broaden the application scope. Experimental evaluations demonstrate that our approach has advantages in both latency and energy efficiency.

REFERENCES

- [1] V. Kumar, R. Sharma *et al.*, “Airgap Interconnects: Modeling, Optimization, and Benchmarking for Backplane, PCB, and Interposer Applications,” *IEEE Trans. on Components Packaging & Manufacturing Technology*, vol. 4, no. 8, pp. 1335–1346, 2014.
- [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 *et al.*, “A 130.7-mm² 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 *et al.*, “‘Memristive’ switches enable ‘stateful’ logic operations via material implication,” *Nature*, vol. 464, no. 7290, p. 873, 2010.
- [5] S. Kvatinsky, D. Belousov *et al.*, “MAGIC–Memristor-aided logic,” *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, 2014.
- [6] P. Huang, J. Kang *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] N. Talati, S. Gupta *et al.*, “Logic design within memristive memories using memristor-aided loGIC (MAGIC),” *IEEE Trans. Nanotechnol.*, vol. 15, no. 4, pp. 635–650, 2016.
- [9] M. Imani, S. Gupta, and T. Rosing, “Ultra-Efficient Processing In-Memory for Data Intensive Applications,” in *Design Automation Conf. (DAC)*, 2017.
- [10] A. Haj-Ali, R. Ben-Hur *et al.*, “Efficient Algorithms for In-memory Fixed Point Multiplication Using MAGIC,” in *Int’l Symp. on Circuits and Systems (ISCAS)*, 2018.
- [11] —, “IMAGING-In-Memory AlGorithms for Image processing,” *IEEE Trans. Circuits Syst. II, Exp. Briefs*, no. 99, pp. 1–14, 2018.
- [12] S.-S. Sheu, M.-F. Chang *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.
- [13] L. Xu, L. Bao *et al.*, “Nonvolatile memristor as a new platform for non-von Neumann computing,” in *Int’l Conf. on Solid-State and Integrated Circuit Technology (ICSICT)*, 2018.
- [14] U. Köster, T. Webb *et al.*, “Flexpoint: An adaptive numerical format for efficient training of deep neural networks,” in *Advances in neural information processing systems*, 2017.
- [15] S. Li, C. Xu *et al.*, “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *Design Automation Conf. (DAC)*, 2016.
- [16] N. Binkert, B. Beckmann *et al.*, “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
- [17] J. E. Stine, I. Castellanos *et al.*, “FreePDK: An open-source variation-aware design kit,” in *Int’l Conf. on Microelectronic Systems Education (MSE)*, 2007.
- [18] H. Li, B. Gao *et al.*, “A learnable parallel processing architecture towards unity of memory and computing,” *Scientific reports*, vol. 5, p. 13330, 2015.
- [19] P. Chi, S. Li *et al.*, “Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory,” in *Int’l Symp. on Computer Architecture (ISCA)*, 2016.
- [20] X. Sun, S. Yin *et al.*, “XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks,” in *Design, Automation, and Test in Europe (DATE)*, 2018.
- [21] C. Liu, B. Yan *et al.*, “A spiking neuromorphic design with resistive crossbar,” in *Design Automation Conf. (DAC)*, 2015.