PFP: Parallel Floating-Point Vector Multiplication Acceleration in MAGIC ReRAM

Wenqing Wang, Ziming Chen, Quan Deng[✉], Liang Fang
National University of Defense Technology
{wangwenqing, dengquan12, chenziming18, lfang}@nudt.edu.cn

***Abstract*—Emerging applications, e.g., machine learning, large language models (LLMs), and graphic processing, are rapidly developing and are both compute-intensive and memory-intensive. Computing in Memory (CIM) is a promising architecture that accelerates these applications by eliminating the data movement between memory and processing units. Memristor-aided logic (MAGIC) CIM achieves massive parallelism, flexible computing, and non-volatility. However, MAGIC ReRAM performs floating-point (FP) vector multiplication sequentially, which wastes parallel computing resources and is limited by the array size.**

To solve this issue, we propose a parallel floating-point vector multiplication accelerator in MAGIC ReRAM. We exploit three levels of parallelism during the calculation of FP vector multiplication, referred to as PFP. First, we leverage the parallelism of MAGIC ReRAM. Second, we bring forward the final exponent to make the exponent calculations parallel. Third, we decouple the calculation of exponent, mantissa, and sign, which allows parallel calculation across accumulation. The experimental results show that PFP achieves a performance speedup of 2.51× and 15% energy savings compared to AritPIM when performing FP32 vector multiplication with a vector length of 512.

***Index Terms*—CIM, Floating-point vector multiplication, Emerging application, Parallel computing, ReRAM array**

I. Introduction

Emerging applications, e.g., machine learning, large language models, and graphic processing, have been widely adopted [1]–[4]. However, these applications generate significant amounts of data movement between the computer and memory unit in traditional hardware architecture, which consumes considerable time and energy [5], [6]. Computing in memory (CIM) architecture is proposed to eliminate unnecessary data movement [7], [8]. Researchers exploit various kinds of memory cells for CIM, such as SRAM [9], [10], DRAM [11], [12], and ReRAM [13]–[16]. ReRAM is a promising candidate that can support arithmetic calculation in either analog or digital formats. Although the performance of analog ReRAM-based CIM is promising, the peripheral circuits, i.e., ADC and DAC, are area-consuming. Furthermore, it faces accuracy and reliability issues [1], [17]–[19]. The digital ReRAM-based CIM supports arithmetic calculation based on the logic operations [8], [20], [21] without ADC/DAC and accuracy loss.

The MAGIC ReRAM [22] distinguishes itself from other ReRAM digital CIM because it supports complete logic and enables a full implementation within a standard ReRAM array. Talati et al. [23] introduced a bit-parallel full adder implementation in MAGIC ReRAM. Imani et al. [24] presented an implementation of integer addition and multiplication in the ReRAM array. To further improve the efficiency of fixed-point operations, Ali et al. [25] proposed parallel computing across rows in MAGIC ReRAM. FloatPIM [18] was the first to propose an implementation to support floating-point (FP) operations. RIME [26] optimized the floating-point multiplication by employing parallel full adders (FAs) within a row. AritPIM [20] supports mature arithmetic operations of both fixed-point and floating-point computations in a bit-parallel manner. To increase the parallelism in MAGIC ReRAM, the following design principles have been developed: i) synthesis and mapping in a row (or column) and parallelizing rows (or columns) MAGIC ReRAM array [27]; ii) utilizing transistors to partition array rows (or columns) into segments to enhance parallelism [20], [26], [28], [29].

FP vector multiplication is a fundamental operation for numerous emerging applications. However, the sequential calculation of FP vector multiplication throttles its performance in MAGIC ReRAM. There are several challenges to achieve parallel FP vector multiplication acceleration. Firstly, there are two kinds of parallelism in the MAGIC ReRAM array, which are row-parallel and column-parallel. There lacks of available data mapping methods and workflow to make full use of both row-parallel and column-parallel. Secondly, the calculation of floating-point accumulation breaks the parallelism of vector multiplication. Floating-point accumulation first compares the exponents, and then adds the aligned mantissa. The shift operation of the aligned mantissa depends on the exponent deviation, which throttles the parallel processing of FP vector multiplication. Thirdly, the potential for unrelated data parallelism within the floating-point vector multiplication algorithm has not been fully exploited. To solve these issues, we propose a parallel floating-point vector multiplication accelerator in MAGIC ReRAM, which exploits three levels of parallelism during the calculation of FP vector multiplication. The main contributions of this paper can be summarized as follows:

- We propose PFP-V to make full use of hardware parallelism. It performs floating-point multiplication in parallel and employs tree-based parallel addition in the accumulation process within the ReRAM array.
- We propose PFP-E based on PFP-V, which decouples the serialized exponent comparison across vector multiplication. We propose an efficient algorithm to accelerate the maximum exponent calculation within the ReRAM array.
- We propose PFP-D based on PFP-E. It further accelerates floating-point vector multiplication by parallelizing the

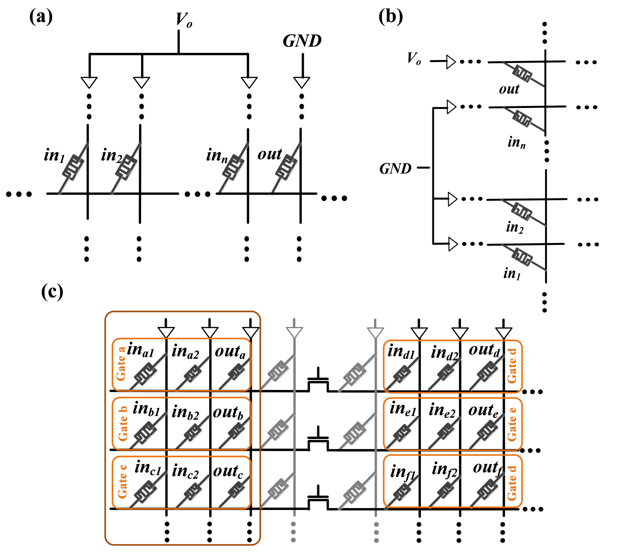

Fig. 1. (a) n-input NOR within a single crossbar row. (b) n-input NOR within a single crossbar column. (c) Partitions increase parallelism, e.g., performing all of the highlighted MAGIC gates in a single clock cycle.

calculations of the independent parts of the floating-point data, i.e., sign, exponent, and mantissa.

- Our experiment shows that the PFP-E achieves 2× speedup and 13% energy savings on average compared to PFP-V. The PFP-D further improves performance by 11%. Compared with the baseline, PFP-D improves performance by 2.51× and reduces energy consumption by 15% when performing FP32 vector multiplication with a vector length of 512.

II. Background

In this section, we briefly introduce the MAGIC NOR operation, data movement in the ReRAM array, and the floating-point vector multiplication algorithm.

A. MAGIC NOR

MAGIC NOR is both logic-complete and non-destructive, enabling the implementation of complex functions. Furthermore, MAGIC NOR can be executed within a ReRAM array, performing operations over both rows and columns [22], [23]. This allows data to be shifted anywhere within the ReRAM array by NOR operations, without the need for extra circuitry.

As shown in Fig. 1(a), when the MAGIC NOR operation is applied to the row, the voltage V_o is applied to input ReRAMs (in_1, in_2, ... in_n), and the output ReRAM (out) is grounded. The initial state of the out is '1'. After the operation, the out becomes the NOR($in_1, in_2, ... in_n$), and $in_1, in_2, ... in_n$ remain unchanged. When executing MAGIC NOR over a column, as shown in Fig. 1(b), the input ReRAMs are connected to the ground, and voltage V_o is applied to the output ReRAM. NOT is a special NOR because the number of input ReRAMs is only one.

MAGIC NOR ReRAM array also provides massive parallelism by synthesizing and mapping logic executed in a single row (or column) and parallel rows (or columns) in the array. To enhance parallelism in MAGIC ReRAM arrays, transistors divide rows (or columns) into partitions (p) for parallel partition computation over rows (or columns) [28], as shown in Fig. 1(c).

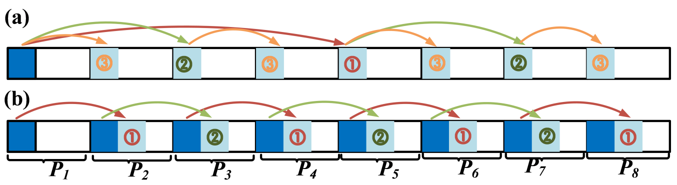

Fig. 2. (a) Broadcasting technique. (b) Shifting technique.

B. Data Movement Techniques in ReRAM Array

Two data movement techniques, broadcasting and shifting, are adopted in partitioned ReRAM arrays. The broadcasting technique copies a single bit from one partition to all k partitions, which are divided by transistors within a ReRAM array. As shown in Fig. 2(a), first, the bit from partition p_1 is copied to partition $p_{k/2+1}$. Then, the transistor between $p_{k/2}$ and $p_{k/2+1}$ is set to non-conducting, and the process is repeated recursively in parallel for partitions p_1 to $p_{k/2}$ and $p_{k/2+1}$ to p_k, as steps 2 and 3.

The shifting technique involves bits being shifted across k partitions. As shown in Fig. 2(b), each partition initially holds a bit, which is then shifted between partitions. This technique involves two steps: first, copying bits from all odd partitions to even partitions; second, copying bits from even partitions to odd partitions. Note that the copy logic is achieved using two NOT gates.

C. Floating-Point Vector Multiplication

A floating-point number N consists of a sign bit s (1 bit), an exponent e (N_e bits), and a mantissa m (N_m bits). It can be expressed as:

$$N = (-1)^{\mathrm{s}} \cdot 2^{\mathrm{e-b}} \cdot (1.\mathrm{m}) \tag{1}$$

b is a constant value (e.g., 127 for 32-bit) and the "1" is the hidden bit. Floating-point vector multiplication can be broken down into floating-point multiplications and additions:

$$\begin{aligned} A \cdot B &= \sum_{i=1}^{n} a_i b_i \\ &= \sum_{i=1}^{n} (-1)^{s_{ai}+s_{bi}} \cdot 2^{e_{ai}+e_{bi}-2b} \cdot (1.m_{ai}) \cdot (1.m_{bi}) \end{aligned} \tag{2}$$

Floating-point multiplication involves: (1) XORing of the sign bits, (2) adding the exponents, and (3) multiplying the mantissa, considering right shift and exponent increment.

Floating-point addition is more complex as it requires aligning the exponents first. The steps are: (1) comparing exponents, (2) shifting the mantissa of the smaller exponent to match the larger exponent, (3) adding the mantissa considering their signs, and (4) normalizing the result.

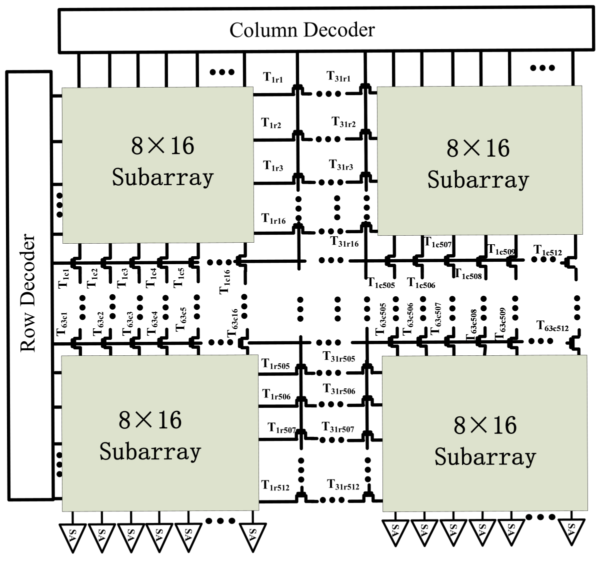

Fig. 3. Architecture of the ReRAM array.

III. The Proposed Design

We optimize floating-point vector multiplication parallelism in MAGIC ReRAM at three levels: PFP-V, PFP-E, and PFP-D. The PFP-V supports FP vector multiplication within a compact ReRAM array. It reduces the latency of FP vector multiplication from the sequential calculation $\mathrm{n} \cdot \mathrm{T}_{mul} + (\mathrm{n}-1)\cdot\mathrm{T}_{add}$ to $\mathrm{T}_{mul} + \log_2(\mathrm{n})\cdot\mathrm{T}_{add}$, with nearly negligible transformer cycles. The PFP-E further reduces the number of exponent comparisons, mantissa shifts, and left-shift normalizations in FP vector multiplication from $\log_2(n)$ times to just once with an additional computation of E_{max}. The PFP-E accelerates FP vector multiplication by parallelizing the unrelated computing processes of the sign, exponent, and mantissa parts.

A. Array Structure

We adopt the ReRAM array, as shown in Fig. 3, to perform floating-point (i.e., FP32, FP16, and BF16) vector multiplication. The array size is 512×512. Each row is divided into 32 partitions by row transistors T_{1r1}, T_{1r2}, ..., T_{31r512}. Similarly, each column is divided into 64 partitions by column transistors T_{1c1}, T_{1c2}, ..., T_{63c512}.

This array has two parallel computing patterns: the row-partitioned parallel computation (RPPC) and the column-partitioned parallel computation (CPPC). In the RPPC, all column transistors are turned on while row transistors are selectively turned off to create various combinations of row partitions. For instance, to enable the row parallel computing pattern with every 64 columns as a parallel partition, turn on all T_c and turn off T_{2r1}, T_{4r1}, ..., T_{30r512}. In CPPC, all row transistors are turned on while column transistors are selectively turned off.

B. PFP-V Design

Algorithm 1 PFP-V

```
Require: $A(a_1, a_2 \ldots, a_m)$, $B(b_1, b_2 \ldots, b_m)$, each element is a n-bit FP number.
    K is an upward integer of $\log_2(m)$.
Ensure: The result of $A * B$
1: for i = m to 1 do
       $c_i = a_i * b_i$ {In parallel}
2: end for
3: for i = 1 to K do
4:     for k= 0, 1... until $k * 2^i + 2^{i-1} + 1 \leqslant m$ do
5:         $j = k * 2^i + 1$
           $c_j = c_j + c_{j+2^{i-1}}$ {In parallel}
6:     end for
7: end for
```

To support and accelerate FP vector multiplication within the size-limited ReRAM arrays constrained by sneak path leakage, wire resistance, and capacitance, a compact design is necessary. The PFP-V design supports floating-point vector multiplication within the compact ReRAM array. It multiplies floating-point numbers using RPPC, then employs an adder tree structure to parallelly accumulate partial results, as illustrated in Algorithm 1.

The workflow of a bit partition for PFP-V in the array is illustrated in Fig. 4. The number of vector elements is set to 4 for simplification. First, place vectors $A(a_1, a_2, a_3, a_4)$ and $B(b_1, b_2, b_3, b_4)$ row by row in the array. Each element has n bits, and each bit is placed in a partition. Second, perform RPPC of $a_1 * b_1, a_2 * b_2, a_3 * b_3, a_4 * b_4$, resulting in c_1, c_2, c_3, c_4. Third, clear intermediate data and transform c_1, c_2, c_3, c_4, grouping every two in a row. Fourth, add c_1, c_2, c_3, c_4 in the RPPC by the bit-parallel floating-point addition algorithm [20]. Fifth, iterate over processes 3 and 4, building a parallel computational tree structure to expedite the final result.

Another question is how to handle excessive or small sizes of FP vector multiplications in this architecture. For excessive sizes of FP vector multiplications, the task can be divided into manageable sizes. For small sizes of FP vector multiplications, they can be computed in parallel within the architecture.

The latency of the PFP-V design compared to sequential calculation FP vector multiplication in a row changes from $\mathrm{n} \cdot \mathrm{T}_{mul} + (\mathrm{n}-1) \cdot \mathrm{T}_{add}$ to $\mathrm{T}_{mul} + \log_2(\mathrm{n}) \cdot \mathrm{T}_{add}$, with nearly negligible transformer cycles.

C. PFP-E Design

The accumulating process throttles FP vector multiplication performance, particularly for long vectors. Although the parallel addition tree structure is adopted in the PFP-D design accumulating process, it still involves $\log_2(n)$ times parallel FP addition. Each time FP addition involves comparing exponents, shifting mantissa, and performing left-shifted normalization. These operations contribute significantly to latency. For example, for a 512-element vector, the accumulation process needs 9 times parallel floating-point additions. In state-of-the-art parallel bit floating-point addition [20], these operations account for almost 67% of the latency of FP addition. That

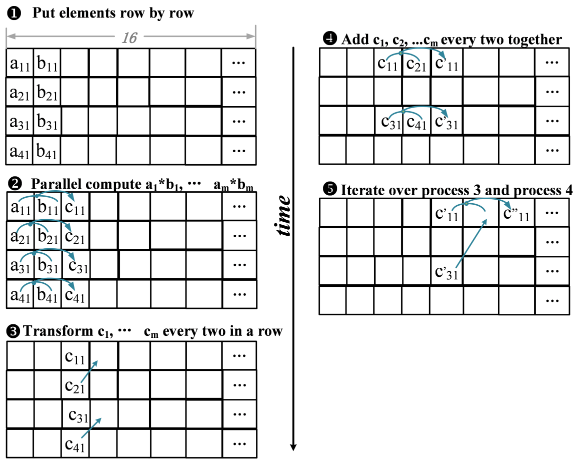
❶ Put elements row by row
16
❷ Parallel compute a1*b1, ··· am*bm
❸ Transform c1, ··· cm every two in a row
❹ Add c1, c2, ...cm every two together
❺ Iterate over process 3 and process 4
time

Fig. 4. The workflow of PFP-V design within a bit partition in this ReRAM array structure. These blocks represent ReRAMs. a_{xy} represents the y-th bit of a_x, similar with b_{xy} and c_{xy}.

greatly impacts the overall latency of floating-point vector multiplication.

To address this, we propose the PFP-E design, which builds on the PFP-V design by reducing the number of exponent part comparisons, mantissa shifts, and left-shifted normalization by utilizing the maximum exponent (E_{max}) during accumulation.

Algorithm 2 PFP-E

```
Require: $E_{\max}$, m floating point numbers $R_1$,$R_2$,.. $R_m$
Ensure: The result of $R_1 + R_2 + ... + R_m$
 1: Broadcast $E_{\max}$ to each row
 2: for i = m to 1  do
 3:     $D_i = E_{\max} - E_i$ {In parallel}
 4: end for
 5: for i = m to 1  do
 6:     Shifting $M_i$ according to $D_i$ {In parallel}
 7: end for
 8: repeat
 9:     Transfer every two mantissas in a row in parallel.
        Add mantissas considering their signs in parallel.
        C = OR (all carry bits).
        $\mathrm{M} = mux_C(\mathrm{M} << 1, \mathrm{M})$
        $E_{\max}$ = $E_{\max}$+ C
10: until  All mantissas are added
11: Perform left-shift normalization.
```

1) The Procedure for PFP-E Design: First, the PFP-E design calculates the maximum exponent (E$_{max}$). Then, it computes the differences between E$_{max}$ and other exponents. All mantissa are shifted according to these differences only once during accumulation. If the mantissa addition introduces a carry bit, the new E$_{max}$ is used to replace the original E$_{max}$, and all mantissa in different rows are uniformly shifted by 1 bit. Left-shifting normalization is only required in the final addition of the accumulation because the exponents of the added numbers must be equal during accumulation.

Algorithm 2 details the computation steps. R represents elements of FP accumulating computation, M represents the mantissa of an FP number, and E represents the exponent of an FP number. $M = mux_C(M << 1, M)$ means if $C = 1$, M is right-shifted a bit and if $C = 0$, M does not change. The workflow and mapping of data to the ReRAM array of the PFP-E design are shown in Fig. 5.

During the accumulation process, mantissa parallel shifting according to the difference, as well as the handling of the hidden bit and the sign bit, follow the same principles as in bit-parallel floating-point addition [20]. Intermediate values do not take part in further computation and are cleared to free up space for further calculation.

The PFP-E design introduces an additional computation of E$_{max}$, which reduces the number of exponent comparisons, mantissa shifts, and left-shift normalizations from $\log_2(n)$ times to just once. This significantly decreases the latency of floating-point vector multiplication in the ReRAM array.

Algorithm 3 Find the maximum

```
Require: $X_1, X_2, X_3...X_m$, each number has n bits.
    $X_{fi}$ represent the i-th bit of $X_f$.
Ensure: The maximum of $X_1, X_2, X_3...X_m$
 1: for i = 1 to n  do
 2:   if $\exists\ X_{fi} = 1, f \in (1, m)$ then
 3:       $\forall\ X_c = 0, c \in (1, m)\ and\ c \neq f$ {In parallel}
 4:   end if
 5: end for
```

2) E_{max} Calculation: The core of the PFP-E design is the computation of E$_{max}$. Traditionally, identifying the maximum in a set of numbers involves comparing them one by one. Here, we introduce a novel algorithm that leverages the ReRAM array's parallelism and logic functions to efficiently identify the maximum number, as detailed in Algorithm 3. If any numbers have a 1 in the highest bit, eliminate those with a 0 in that bit and compare the next bit among the remaining numbers. If all numbers have a 0 in the highest bit, move to the next bit. Continue this process from the highest to the lowest bit until only the maximum number remains. This algorithm is 3× faster than the state-of-the-art conventional approach [20] with a tree-based parallelism structure when identifying the maximum exponent of 512 FP32 numbers.

Fig. 6 illustrates the maximum identifying process in a bit partition of the ReRAM array, where numbers are stored row by row with each bit placed in a partition. The steps are as follows: (1) Compute the NOR of all same-position bits of these numbers, and obtain the value K$_f$, where f represents the bit position; (2) For each bit X$_{qc}$ (representing number X$_q$ in position c bit), operate $\mathrm{K}_f\mathrm{X}_{qc} + \mathrm{NOT}(\mathrm{K}_f)\mathrm{X}_{qf}\mathrm{X}_{qc}$. If K$_f$ is 1 (all numbers in the f position are 0), each bit of X$_q$ remains unchanged. If K$_f$ is 0 (at least one number in the f position is 1): If X$_{qf}$ is 1 (indicating number X$_q$ in position f bit is 1), the value of the number X$_q$ remains unchanged; If X$_{qf}$ is 0 (indicating number X$_q$ in position f bit is 0), then the number X$_q$ will change to 0. Repeat the computation of f from the highest position to the lowest position, these numbers are all 0 except the largest one remains. (3) OR all bits at the same

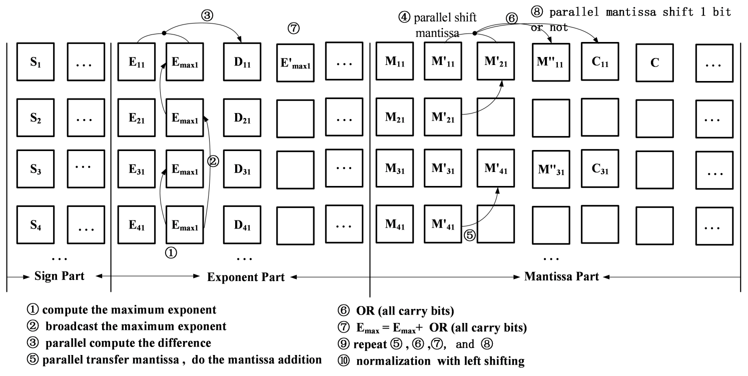
③
⑦
④ parallel shift mantissa
⑥
⑧ parallel mantissa shift 1 bit or not
Sign Part
Exponent Part
Mantissa Part
① compute the maximum exponent
② broadcast the maximum exponent
③ parallel compute the difference
⑤ parallel transfer mantissa , do the mantissa addition
⑥ OR (all carry bits)
⑦ Emax = Emax+ OR (all carry bits)
⑨ repeat ⑤ , ⑥ ,⑦, and ⑧
⑩ normalization with left shifting

Fig. 5. The workflow of PFP-E design.

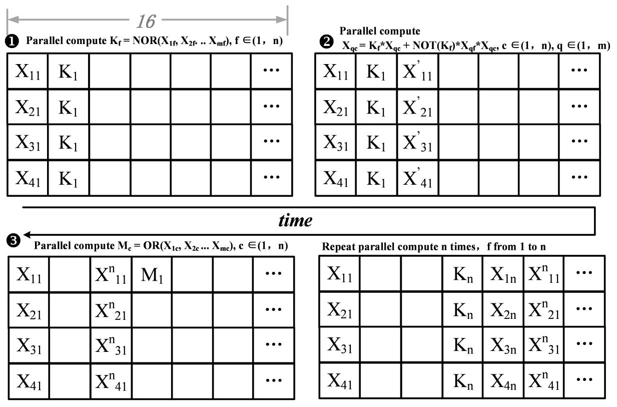
16
❶ Parallel compute Kf = NOR(X1f, X2f, .. Xmf), f ∈(1, n)
❷ Parallel compute Xqc = Kf*Xqc + NOT(Kf)*Xqf*Xqc, c ∈(1, n), q ∈(1, m)
time
❸ Parallel compute Mc = OR(X1c, X2c ... Xmc), c ∈(1, n)
Repeat parallel compute n times, f from 1 to n

Fig. 6. The maximum number identifying process in a bit partition of this ReRAM array. These blocks represent ReRAMs.

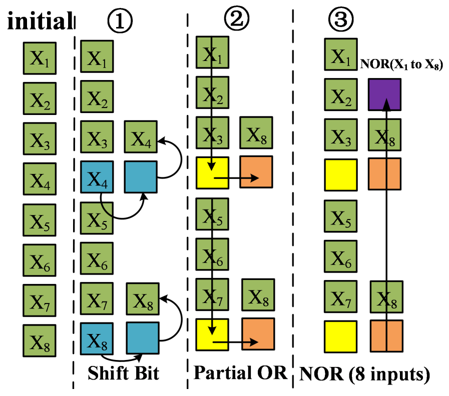

Fig. 7. The NOR of whole column bits operation within the ReRAM array.

position across these numbers, and the resulting value indicates the largest number. The original X_{qc} is still retained without overwriting for the subsequent calculation.

The maximum identifying process in the array involves NOR of whole column bits operation. Ideally, the inputs of the NOR operation could be hundreds, but considering practical circuit constraints, we implement inputs of the NOR operation $<= 4$ to compute NOR of whole column bits. This process is shown in Fig. 7. The initial state is each bit within each partition is in the same column. First, shift a bit to another column by twice NOT operation to create space to do NOR operation within the column. Second, move the result of partial NOR to another column by NOT operation and the result is a partial OR. Third, NOR the value of partial OR with the shifted bits, and get the result of 8 inputs NOR. Since the array is column-partitioned, we can parallelize these operations in column partitions. Afterward, using the partial result parallel to succeed NOR operation, we obtain the NOR all bits at position f. All positions of NOR whole column bits operations can be computed in parallel. The operation of OR whole column bits is the NOT of the NOR whole column bits operation. In addition, due to this method destroying column data, we need to save it to another column first.

D. PFP-D Design

To speed up floating-point vector multiplication, it is crucial to identify and leverage data-independent parallelism within the algorithm. We further accelerate the PFP-E by parallelizing the unrelated computing processes of the sign part, the exponent part, and the mantissa part.

Considering the previous floating-point addition during the vector multiplication process, once the mantissa addition is complete, handle the mantissa operation by determining whether to right-shift one bit. Next, handle the exponent by deciding whether to increment it. Finally, determine the sign bit to obtain the final result. However, when obtaining the result of the mantissa addition, all the necessary computational information for the three parts is available. Since these three parts are independent, they can be computed in parallel. This design can further enhance the computational throughput of floating-point vector multiplication.

IV. Evaluation

To evaluate the performance of our proposed floating-point vector multiplication designs, we utilize a cycle-accurate Arit-PIM simulator [20]. The parameters of ReRAM in the simulator are based on RACER [30], as shown in Tab.I.

The area estimation of the array is 3495 μm^2 by the NVSim [31] in the 22 nm process. Thanks to back-end-of-line (BEOL)

TABLE I
ReRAM Parameters

switch latency (ns)	1
ratio of on-off resistances	1: 500
switch transition (pJ)	0.0128

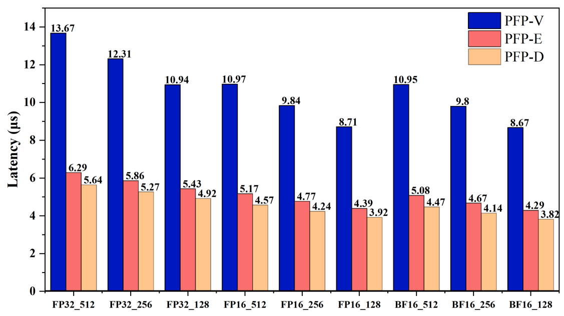

Fig. 8. The latency of PFP-V, PFP-E, and PFP-D design.

integration, the ReRAM can be deposited above transistor layers, improving area efficiency.

We compare our work with FloatPIM [18] and AritPIM [20] for the FP32 vector multiplication with vector lengths of 512, as shown in Tab.II. For consistency, we set the switching time for all ReRAMs to 1 ns and arrange the elements of the FP vector multiplication row by row, perform parallel FP multiplication in each row, shift every two partial results in a row, and perform parallel addition. The PFP-D design improves performance by 2.51 × and 15% energy savings compared to AritPIM when doing FP32 vector multiplication with a vector length of 512. (FloatPIM is excluded from the energy comparison due to its extra energy consumption for the search process.)

TABLE II
Comparison with Other Works

	FloatPIM	AritPIM	PFP-V	PFP-E	PFP-D
Latency(μs)	14.3875	14.158	13.67	6.29	5.64
Energy (μJ)	-	14.37	14.37	12.24	12.25

Fig. 8 shows the latency of PFP-V, PFP-E, and PFP-D designs of FP32, FP16, and BF16 vector multiplication with vector lengths of 512, 256, and 128. The PFP-E design achieves an average 2× speedup over the PFP-V, and PFP-D has an average 11% speedup over PFP-E. The PFP-E design's speedup is due to (i) efficient E$_{max}$ method; (ii) reduced exponent comparisons, the mantissa shifts, and the left-shifted process during multiply-accumulation; The PFP-D design improves speed by parallelizing data-unrelated computations in exponent, sign and mantissa during multiply–accumulation.

Fig. 9 presents the energy of PFP-V, PFP-E, and PFP-D designs for FP32, FP16, and BF16 vector multiplication with vector lengths of 512, 256, and 128. The PFP-E design consumes, on average, 87% of the energy consumption of PFP-V. The PFP-E design's energy efficiency is achieved through algorithm optimization and reducing computational load. The

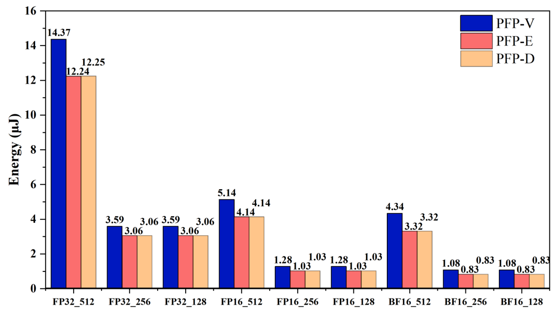

Fig. 9. The energy of PFP-V, PFP-E, and PFP-D design.

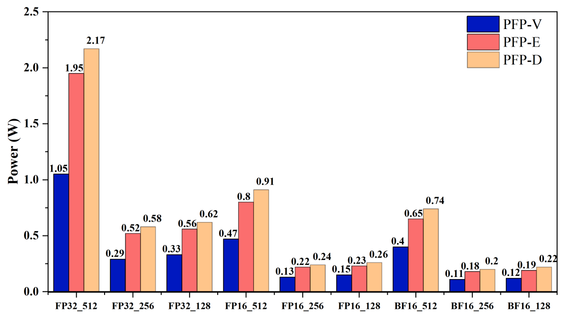

Fig. 10. The power of PFP-V, PFP-E, and PFP-D design.

energy consumption of the PFP-E and PFP-D is nearly identical, as PFP-D only enhances parallelism without reducing computational load.

Fig. 10 displays the power consumption of PFP-V, PFP-E, and PFP-D designs of FP32, FP16, and BF16 vector multiplication with vector lengths of 512, 256, and 128. PFP-V has the lowest power consumption, followed by PFP-E, with PFP-D having the highest. The PFP-E design's energy is higher than PFP-V because PFP-E's speed is almost 2× that of PFP-V while consuming 87% of the energy. PFP-D consumes the same energy in less time, resulting in higher power consumption.

V. Conclusion

Floating-point vector multiplication is essential for many emerging applications. This paper presents the design and implementation of a parallel floating-point vector multiplication accelerator using MAGIC ReRAM, named PFP. PFP-V leverages hardware parallelism for floating-point multiplication and employs an adder tree structure to accumulate partial results. PFP-E optimizes accumulation by reducing calculations for exponent comparison, the mantissa shifts, and the left-shift process. PFP-D further accelerates computation by parallelizing the independent parts of the floating-point data (sign, exponent, mantissa). Experimental results show significant improvement in latency and partial improvement in energy compared to other state-of-the-art floating-point multiplication methods.

Acknowledgment

This work was supported by NSFC-62202481, NSFC-62172155, U22A2027, 2024RC3116, ZK22-05 and 22-TDRCJH-02-006.

References

[1] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *Fiber*, vol. 56, no. 4, pp. 3–7, 2015.

[2] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *ECCV*, 2014.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. a. Lanctot, "Mastering the game of go with deep neural networks and tree search," *Nature*.

[4] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations Trends in Signal Processing*, vol. 7, no. 3, pp. 197–387, 2014.

[5] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, and P. Ranganathan, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Architectural Support for Programming Languages and Operating Systems*, 2018.

[6] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *IEEE International Symposium on Workload Characterization*, 2013.

[7] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[8] V. Seshadri and O. Mutlu, "Simple operations in memory to reduce data movement," *Advances in Computers*, vol. 106, pp. 107–166, 2017.

[9] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[10] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," 2017.

[11] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *International Symposium on Microarchitecture*, 2019.

[12] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. G. Luna, and O. Mutlu, "Simdram: An end-to-end framework for bit-serial simd computing in dram," 2021.

[13] A. Ankit, K. Roy, D. S. Milojicic, I. E. Hajj, and J. P. Strachan, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," 2019.

[14] T. Chou, W. Tang, J. Botimer, and Z. Zhang, "Cascade: Connecting rrams to extend analog dataflow in an end-to-end in-memory processing paradigm," in *the 52nd Annual IEEE/ACM International Symposium*, 2019.

[15] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, and K. Bertels, "Memristor for computing: Myth or reality?" in *2017 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[16] S. Hamdioui, L. Xie, M. Taouil, K. Bertels, and H. A. D. Nguyen, "Memristor based computation-in-memory architecture for data-intensive applications," *IEEE*, 2015.

[17] M. Courbariaux, Y. Bengio, and J. P. David, "Training deep neural networks with low precision multiplications," *Computer Science*, 2014.

[18] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: in-memory acceleration of deep neural network training with high precision," in *the 46th International Symposium*, 2019.

[19] F. Tu, Y. Wang, Z. Wu, L. Liang, Y. Ding, B. Kim, L. Liu, S. Wei, Y. Xie, and S. Yin, "Redcim: Reconfigurable digital computing- in - memory processor with unified fp/int pipeline for cloud ai acceleration," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 243–255, 2023.

[20] O. Leitersdorf, D. Leitersdorf, J. Gal, M. Dahan, R. Ronen, and S. Kvatinsky, "Aritpim: High-throughput in-memory arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, 2023.

[21] H. A. Ameer, B. H. Rotem, W. Nimrod, R. Ronny, and K. Shahar, "Imaging: In-memory algorithms for image processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, pp. 4258–4271, 2018.

[22] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[23] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[24] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[25] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, "Efficient algorithms for in-memory fixed point multiplication using magic," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[26] Z. Lu, M. T. Arafin, and G. Qu, "Rime: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 120–125.

[27] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.

[28] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[29] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Multpim: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1647–1651, 2022.

[30] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "Racer: Bit-pipelined processing using resistive memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 100–116.

[31] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.