INTEGRATION, the VLSI journal 93 (2023) 102060

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

MAGIC-DHT: Fast in-memory computing for Discrete Hadamard Transform

Maliha Tasnim [a,*], Chinmay Raje [a], Shuyuan Yu [a], Elaheh Sadredini [b], Sheldon X.-D. Tan [a]

[a] Department of Electrical and Computer Engineering, University of California, Riverside, CA 92521, USA
[b] Department of Computer Science, University of California, Riverside, CA 92521, USA

ARTICLE INFO

Keywords:
Hadamard transform
Memristor
ReRAM
PIM
Parallel computing

ABSTRACT

Discrete Hadamard transform (DHT) is a signal processing tool that decomposes an arbitrary input vector into a superposition of Walsh functions. Due to its wide range of applications in processing big data, a fast and energy-efficient hardware design for DHT with high throughput capability is essential. Processing in memory (PIM) allows the in-place computation to reduce the data traffic, which is a major speed bottleneck in the existing computing. In this work, we propose an efficient hybrid parallel PIM-based computation for DHT. Our proposed method explores the recursive computation of DHT and is based on the memristor-aided logic (MAGIC) gates in which the arithmetic operations are carried out via simple logic NOR operation. We propose two in-memory computing methods for the DHT encoding process. At the arithmetic level, to improve efficiency, we propose to share the intermediate results between addition and subtraction in DHT in the first method called *MAGIC-DHT-1D* which provides an average speedup of 1.12× over the recently proposed DigitalPIM for 1D DHT. *Furthermore,MAGIC-DHT-1D* also outperforms SIMPLER in terms of energy and energy density in average. We also propose a second method, called *MAGIC-DHT-2D*, to share the carrier independent computation cycles among multi-bit parallel addition and subtraction. At the algorithm level, we also explore both row and column-based PIM NOR computing in the same crossbar to avoid the transposition operation required in the 2D DHT process. *MAGIC-DHT-2D* provides an average speedup of 4.84× and 7.25× over two state-of-the-art methods DigitalPIM and SIMPLER, respectively for each complete set of 2D DHT computing cycles. Our numerical results further show that our proposed optimized methods can lead up to 56.19× and 6.90× speed-up, as well as 57.84× and 5.96× higher throughput over NVIDIA RTX Titan GPU to compute 1D DHT and 2D DHT, respectively.

1. Introduction

The Discrete Hadamard Transform (DHT) is a special type of discrete Fourier transform that is widely used in signal processing. It decomposes an input vector into a superposition of Walsh functions, consisting of only −1 and 1 values. Due to its simplicity in computation, it is a suitable choice for many communication applications that require fast and efficient encoding of signal data. The Walsh–Hadamard transform-based Code Division Multiplexing (WHCDM) protocol has been proposed to reduce inter-code interference and improve the bit-error rate [1]. DHT also has various applications in image, video, and audio processing, such as image segmentation [2], motion detection [3], and video shot boundary detection [4]. A wide range of efficient signal processing algorithms such as dyadic convolution [5], adaptive filtering [6], and low-energy convolution neural networks [7,8] are connected to the Walsh–Hadamard Transform. With the increasing demand for encoding large data, such as audio, image, video, and communication channel signals, there is a need for fast and energy-efficient hardware design for DHT with high throughput capability. Over the years, tools, libraries, and mathematical software have been developed for both CPU [9,10] and GPU [11,12] to address the efficiency of Walsh–Hadamard transform.

With the increase of big data from smart devices and systems, processing in memory (PIM) has emerged as a promising computing platform to address the data movement issue by implementing computing logic within or near memory [13–15]. Traditional von-Neumann architecture requires large data movement between CPU and memory, and this can lead to the well-known memory-wall bottleneck. To overcome this challenge, recent effort tries to bring the processing unit to data in memory instead of moving the data to the processor. PIM is enabled by the emergence of new non-volatile memory technologies such as memristors, where the logic state of memories depends on the resistance of the devices, which is controlled by the currents flowing

* Corresponding author.
E-mail address: mtasn004@ucr.edu (M. Tasnim).

https://doi.org/10.1016/j.vlsi.2023.102060
Received 13 February 2023; Received in revised form 22 May 2023; Accepted 19 July 2023
Available online 24 July 2023

through them. The high density, low power consumption, fast switching, and CMOS compatibility make memristor a promising candidate for lower power main memory with processing capability [16].

In this work, we explore the use of MAGIC-based digital PIM for DHT. Unlike the fast Fourier transform, DHT can be computed without general multiplication as the transformation matrix only involves 1 and −1 and does not depend on special basis functions. This simple and recursive computation algorithm results in a low ratio of arithmetic operations to memory transfers when computing DHT over large data, making it more suitable for digital PIM computing for data-intensive applications. We propose a hybrid solution that exploits both arithmetic/logic level parallelism as done in previous works [17,18] and algorithm level parallelism. Our key contributions are as follows:

1. First, we propose the in-memory computing architecture and a hybrid computing scheme for DHT computing process, which is based on existing NOR-based MAGIC architecture [19]. At the arithmetic level, to improve efficiency, we propose to share the intermediate results between addition and subtraction in DHT in the first method, called *MAGIC-DHT-1D*, which provides an average speed-up of 1.12× over the recently proposed DigitalPIM for computing 1D DHT. Furthermore, *MAGIC-DHT-1D* also outperforms SIMPLER in terms of energy and energy density by 0.3% and 7% respectively.
2. To improve the efficiency of 2D DHT computing, we propose *MAGIC-DHT-2D* that computes the independent results with the same carrier among multi-bit addition and subtraction in parallel, and further reduces the cycles required for addition and subtraction operations in DHT. Furthermore, at the algorithm level, we explore both row and column-based MAGIC NOR computing to avoid the transposition operation required in the 2D DHT operation. This can lead to substantial communication savings compared to simple two 1D DHT operations in either row-only or column-only based method.

We compared the proposed MAGIC-DHT methods against two PIM frameworks, *DigitalPIM* [20] and *SIMPLER* [21] as well as the GPU implementation on NVIDIA RTX Titan GPU.

Our numerical results show that *MAGIC-DHT* can provide 1.12× and 15.75× average speed-up over *DigitalPIM* for computing 1D and 2D N-point DHT, respectively. The average speedup of *MAGIC-DHT* over *SIMPLER* is 0.97×, 0.41×, for 1D and 2D DHT, respectively. Although, *MAGIC-DHT* has higher latency than *SIMPLER*, on average, the hardware resource consumption in *MAGIC-DHT* improves significantly over *SIMPLER* for both 1D and 2D DHT computation. *MAGIC-DHT* can lead to 1.07× and 3.18× less energy density over *SIMPLER* for computing 1D and 2D DHT, respectively.

Furthermore, we show that the proposed PIM-based method *MAGIC-DHT-1D* can provide up to 56.19× speed-up and 57.84× higher throughput over NVIDIA RTX Titan GPU to compute 1D DHT. The proposed *MAGIC-DHT-2D* can provide up to 6.9× speedup and 5.96× higher throughput over NVIDIA RTX Titan GPU to compute 2D DHT.

This paper is organized as follows: Section 2 reviews the PIM concept and basic formula and operations in the Discrete Hadamard transform process. Section 3 presents the proposed digital in-memory Hadamard encoder architecture and the hardware implementation details. Numerical results and discussion are presented in Section 4. Finally, Section 5 concludes this paper.

2. Review of related works

2.1. Processing in memory

PIM has been applied to a number of applications ranging from imaging process to the neural network processing [22–27]. Existing PIM can be roughly classified into two classes: (1) analog based PIM for arithmetic operations in which analog voltage solutions are computed via currents flowing through PIM resistance networks [25–28], (2) Digital logic operations enabled in PIM, where the additions and multiplications are performed using basic logic operations like *NOR* in multiple clock cycles, such as MAGIC [19] and FELIX [16] frameworks. The analog PIM is very fast, but it still suffers the low accuracy issues and a large area footprints for required analog to digital converter (ADC) and digital to analog converter (DAC) interface modules [20,29]. On the other hand, digital based PIM is much more accurate but has larger latency as many clock cycles are required for computation, especially for the multiplications. But one can exploit the inherent parallelism in the application algorithms to speed up the computation process.

Memristor aided logic gate (MAGIC) is based on the NOR logic operation and can implement only one single-cycle Boolean logic (NOR) in memory. The rest of the logic operations such as XOR, NAND, OR etc. require multiple cycles with NOR gates in memory. FELIX [16] resolves this issue by exploiting the memristors with non-binary states (additional intermediate states between R_H and R_L) and multi-level logic voltage driver. Some logic synthesis and optimization methods were also proposed to map logic functions into the PIM processors based on the BDD [17] and mathematical programming [18]. However, those methods mainly focus on the logic gate level optimization to reduce the clock cycles for both logic NOR operations and data movement for specific crossbar PIM structures. In this work, we mainly focus on the algorithmic level latency optimization and data movement reduction. Moreover, our proposed method is actually orthogonal to those gate level PIM optimization techniques.

In a digital memristive crossbar array, the state of a memristor only changes when a current beyond a minimum threshold (I_{reset} or I_{set}) passes through the device from a particular direction as shown in Fig. 1(a). This property can be exploited to implement several Boolean logic such as NOR, NAND, XOR, etc. by connecting multiple memristors in series/parallel. As an example, Fig. 1(b) shows the 2-input NOR gate implementation in memory by connecting the input memristors in parallel with each other and the output memristor in series from the opposite polarity. The output memristor is pre-programmed to logic '1' (R_L) before a voltage V_d is applied to the input terminals. If at least one of the input memristors is storing logic '1' (R_L), a current greater than I_{reset} will pass through the output memristor, and the state of the output memristor will switch to logic '0' (R_H) as shown in Fig. 1(c). If both input memristors are at logic '0' (R_H), the current passing through the output memristor will not be enough to change the state as shown in Fig. 1(d). V_d should be large enough such that the state of output is switched only when at least one of the input memristors is at a low resistance state. However, V_d should not be larger than the minimum voltage required to switch any input memristors in the state R_H to state R_L as shown in (1) [30]:

$$2 \times |V_{T,off}| \leq V_d \leq |V_{T,on}| \times (1 + \frac{2R_L}{R_H}) \tag{1}$$

2.2. Review of discrete Hadamard transform

Mathematically, 1D N-point Discrete Hadamard Transform (N-DHT) $\mathbf{y}_N$ of the sequence $\mathbf{x}_N = [x_1\, x_2\, x_3\, \ldots\, x_N]^T$ can be defined as follows:

$$\mathbf{y}_N = \mathbf{H}_N \mathbf{x}_N \tag{2}$$

Here, $\mathbf{H}_N$ is an $N \times N$ Hadamard matrix and can be iteratively computed as

$$\mathbf{H}_N = \begin{bmatrix} \mathbf{H}_{N/2} & \mathbf{H}_{N/2} \\ \mathbf{H}_{N/2} & -\mathbf{H}_{N/2} \end{bmatrix} = \mathbf{H}_{N/2} \otimes \mathbf{H}_2;\ \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3}$$

where, '⊗' represents Kronecker operation. Based on (2) and (3), N point DHT of vector $\mathbf{x}_N$, can be computed from $K = log_2(N)$ number

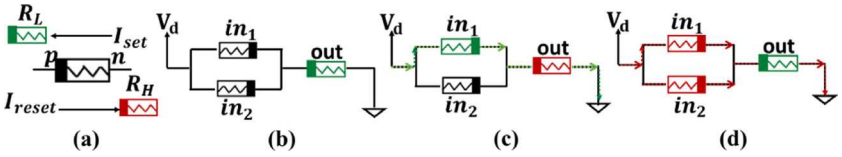

Fig. 1. (a) Resistance state change in memristor. Red: High Resistance, Green: Low Resistance, Black: Arbitrary State; (b) Connection of input and output memristor for 2-input NOR; (c)–(d) Two-input NOR operation in memristor for different input combination: $(1+x)' = 0$ and $(0+0)' = 1$.

of iterative computation of 2-point DHT over $\mathbf{x}_N^o$ and $\mathbf{x}_N^e$, which are defined as below:

$$\mathbf{x}_N^o = \begin{bmatrix} x_1 & x_3 & x_5 & \dots & x_{N-1} \end{bmatrix} \tag{4a}$$

$$\mathbf{x}_N^e = \begin{bmatrix} x_2 & x_4 & x_6 & \dots & x_N \end{bmatrix} \tag{4b}$$

$$\mathbf{H}_2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ x_1 - x_2 \end{bmatrix} \tag{4c}$$

where the $\mathbf{x}_N^o/\mathbf{x}_N^e$ takes the odd/even elements from $\mathbf{x}_N$. Note that for (4c), we can perform such 2-point DHT operation on a matrix block level. In the following, we illustrate the recursive calculation of discrete 8-point Hadamard transform $\mathbf{y}_8$ from 2-point DHT operation over input $\mathbf{x}_8$ in (5)

$$\mathbf{y}_2 = \mathbf{H}_2 \begin{bmatrix} \mathbf{x}_8^o \\ \mathbf{x}_8^e \end{bmatrix} = \begin{bmatrix} \mathbf{x}_8^o + \mathbf{x}_8^e \\ \mathbf{x}_8^o - \mathbf{x}_8^e \end{bmatrix}_{2\times 4} \tag{5a}$$

$$\mathbf{y}_4 = \begin{bmatrix} \mathbf{y}_2^o + \mathbf{y}_2^e \\ \mathbf{y}_2^o - \mathbf{y}_2^e \end{bmatrix}_{4\times 2} \tag{5b}$$

$$\mathbf{y}_8 = \begin{bmatrix} \mathbf{y}_4^o + \mathbf{y}_4^e \\ \mathbf{y}_4^o - \mathbf{y}_4^e \end{bmatrix}_{8\times 1} \tag{5c}$$

where,

$$\mathbf{x}_8{}^\circ = \begin{bmatrix} x_1 & x_3 & x_5 & x_7 \end{bmatrix}_{1\times 4}; \qquad \mathbf{x}_8^e = \begin{bmatrix} x_2 & x_4 & x_6 & x_8 \end{bmatrix}_{1\times 4}$$

$$\mathbf{y}_2{}^\circ = \begin{bmatrix} x_1 + x_2 & x_5 + x_6 \\ x_1 - x_2 & x_5 - x_6 \end{bmatrix}_{2\times 2}; \qquad \mathbf{y}_2^e = \begin{bmatrix} x_3 + x_4 & x_7 + x_8 \\ x_3 - x_4 & x_7 - x_8 \end{bmatrix}_{2\times 2}$$

$$\mathbf{y}_4{}^\circ = \begin{bmatrix} x_1 + x_2 + x_3 + x_4 \\ x_1 - x_2 + x_3 - x_4 \\ x_1 + x_2 - x_3 - x_4 \\ x_1 - x_2 - x_3 + x_4 \end{bmatrix}_{4\times 1}; \qquad \mathbf{y}_4^e = \begin{bmatrix} x_5 + x_6 + x_7 + x_8 \\ x_5 - x_6 + x_7 - x_8 \\ x_5 + x_6 - x_7 - x_8 \\ x_5 - x_6 - x_7 + x_8 \end{bmatrix}_{4\times 1} \tag{5d}$$

We observe that in every recursive step, we perform element-wise addition and subtraction of two matrices or vectors. As a result, we can exploit such combined addition and subtraction in our PIM computing as shown in the next section.

2D $M \times N$ discrete Hadamard transform of data matrix $\mathbf{X}_{M\times N}$ can be computed by passing $\mathbf{X}_{M\times N}$ through 1D Hadamard transformer twice. First, N point 1D Hadamard transform $\mathbf{y}_N$ of M row vectors are calculated based on recursive formula in (5). Then M-point DHT of N column vectors of resulting matrix $\mathbf{Y}_{M\times N}$ is calculated to get final $M \times N$ DHT encoded matrix $\mathbf{Z}_{M\times N}$ as shown in (6).

$$\mathbf{Z}_{M\times N} = \mathbf{H}_M \mathbf{Y}_{M\times N}^T = \mathbf{H}_M (\mathbf{H}_N \mathbf{X}_{M\times N}^T)^T \tag{6}$$

In total, $(N \times (log_2 M) \times (M/2) + M \times (log_2 N) \times (N/2))$ iterative 2-point DHT computation is required to encode data matrix $\mathbf{X}_{M\times N}$ with 2D discrete Hadamard transform.

Several conventional VLSI and recent FPGA-based architectures for discrete Hadamard transform have been proposed in numerous literature. A recursive sparse-matrix factorization for the Hadamard matrix in terms of the Kronecker products of 2 × 2 Hadamard and identity matrices of consecutively lower orders have been proposed in [31] that simplified fast Hadamard transform algorithm. A chip for a systolic array of the Hadamard transform was proposed in [32] which requires $2 \cdot (N-1)$ clock cycles for computation while its latency is N cycles. To improve the efficiency of computation, a fully-pipelined hardware design for DHT-calculator has been realized in [33] where four different pipelined modular designs for transform length $N = 4$ from kernel matrix of HT have been derived. Three different structures, namely the transposition-free structure, the folded structure and the pipeline structure have been proposed in [34] to implement 2D DHT encoder. Among these three proposed structures, the pipeline structure involves least area-delay product (ADP) and energy per sample (EPS) according to the ASIC synthesis result. To improve the marginal bit-saving and truncation error associated with the fixed-width structures of DHT based on the conventional approach, a decimation-in-frequency (DIF) generator for DHT has been proposed in [35]. Their proposed model comprising of pre-truncation and post-truncation phases, as well as a logic optimized addition–subtraction design customized for DHT structure, achieved higher bit-saving with relatively less truncation error for both 1D and 2D models.

Earlier architectures focused on optimizing the throughput of systolic array-based DHT computation. However, the optimization of throughput and efficiency is still limited by the maximum number of computational units available in the processor as well as the maximum transfer rate of data-bus from memory to processing unit. Therefore, a highly parallel computing platform such as PIM, which significantly reduces in the memory traffic, can provide a fast and efficient computation of both 1D and 2D DHT with significantly higher throughput.

We remark that in this work, we focus on the digital PIM based computing without using analog based PIM. Analog based PIM is an excellent candidate for large scale parallel computation without the necessity to tackle issues with memory traffic. Also, analog PIM has the scope of archiving massive acceleration by bringing down the latency of complex arithmetic operation such as addition/multiplication to practically single cycle. However, analog PIM suffers from low accuracy issues as they operates directly on the voltage or current levels. Also, large nonuniform analog resistance for undetermined states contributes to building up error. Furthermore, analog PIM, specially those based on current ReRAM technology has the disadvantage of requiring complex ADC/DAC block as well as multi level voltage converter in each crossbar peripheries. This results in increasing area and power consumption in crossbar peripherals. In fact, studies shows that ADC/DAC block in analog ReRAM tile occupies around 98% of total area and consumes around 89% of total power [29]. Furthermore, they have low scalability issue. Digital based in memory computation simplifies this peripheral architecture by removing the ADC/DAC block. Also computation in ReRAM crossbar based on digital logic states ($R_H = 0$ and $R_L = 1$) are less prone to signal corruption that usually results from noise such as leakage current in sneaky path, thermal noise, device parameter variation etc. In next section, we exploit these advantages of ReRAM crossbar in digital domain and propose a digital based PIM implementation of discrete Hadamard transformer.

3. Proposed digital in-memory Hadamard encoder architecture

In this section, we present the novel MAGIC-based processing in memory of digital data for optimized discrete Hadamard transform. First, we discuss a straightforward implementation of DHT in *DigitalPIM* which is a digital-based PIM platform capable of accelerating basic arithmetic operation as proposed in [20]. Next, we propose our optimized one dimensional DHT encoding algorithm, called *MAGIC-DHT-1D* and two dimensional DHT encoding called *MAGIC-DHT-2D*, which recursively computes N-point DHT from 2-point DHT in memory.

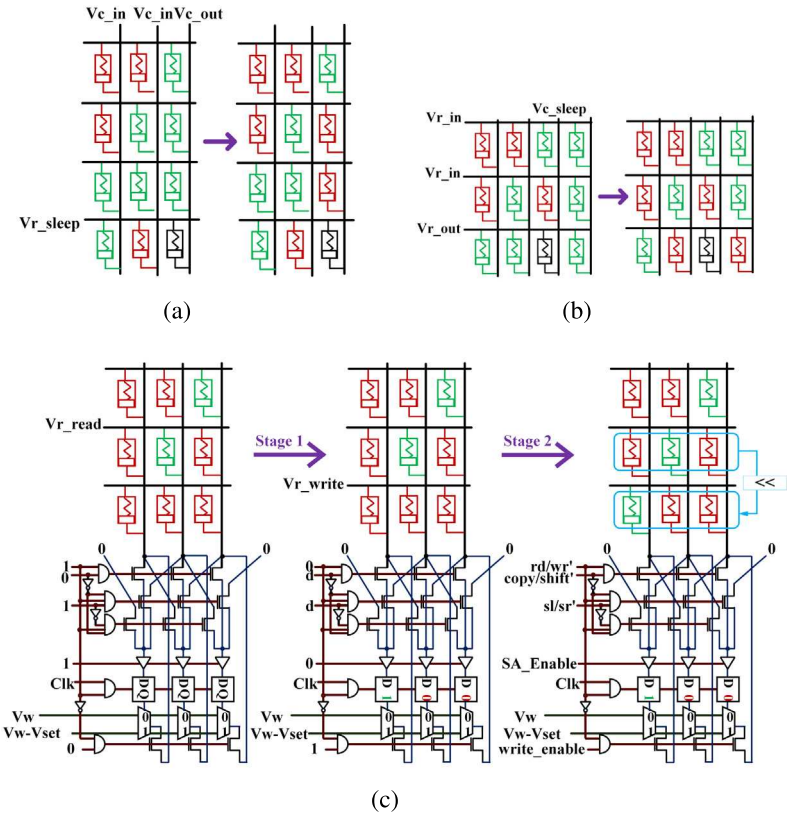

Fig. 2. (a) Column-wise NOR (parallel in rows). (b) Row-wise NOR (parallel in columns). (c) Left shift operation: Stage 1- Left shift 2nd row while reading into read–write buffer; Stage 2- write back to 3rd row from read–write buffer.

3.1. Logic and memory operation cycle

Before we present our algorithms, we first review the logic operations in the PIM. For logic operation in memory, all the memristors of output row/column are set to logic '1' at the first cycle. During column-wise logic operation, the column driver provides high voltage to input bit-lines (Vc_in) and low voltage (Vc_out) to output bit-line, as shown in Fig. 2(a). Thus, in each row of a crossbar, a column-wise NOR operation is performed in parallel within a single cycle. A voltage of Vr_sleep can be provided to a particular range of wordline so that the respective rows are excluded from the parallel computation cycle if required. Here, $Vc\_in - Vc\_out = V_d$ is selected in compliance with (1). Vr_sleep is chosen carefully so that none of the memristor cells in the excluded row changes their states. During row-wise logic operation, the row driver provides low voltage to input word-lines (Vr_in) and high voltage (Vr_out) to output wordline as shown in Fig. 2(b). Again, $Vr\_out - Vr\_in = V_d$ is selected in compliance with (1). Similar to column-wise logic operation, a carefully selected sleep voltage of Vc_sleep can be provided to any particular column to exclude it from the parallel logic operation.

During the read operation, the row buffer is activated and a read voltage is applied to the specified row. The read voltage (Vr_read) is selected carefully so that any memory cell with the state R_L does not switch to the state R_H. To write data into memory cells, all the memristors in the selected row are reset to a high resistance state R_H. In the next cycle, the wordline of the target row is connected to voltage Vr_write by the row driver, and the multiplexer output of $Vw = Vr\_write$ is selected by the read–write buffer and is applied to the bit-line if the buffer is storing logic '0' in that particular bit position. Otherwise, voltage $Vw + Vset$ is applied to the bit-line if the buffer stores '1'. Here, $Vset$ is the voltage required to set a memristor to logic state '1'. Thus, writing to memristor cells requires two cycles to complete. The read/write buffer can also be used as a shifter as depicted in Fig. 2(c).

Furthermore, one row of a crossbar can be left/right-shifted by one bit within two stages. As shown in Fig. 2(c), In the first stage, 010 from the second row is read first and then is left-shifted as 100 and stored in the read–write buffer. Then, the shifted data (100) from the buffer is written back to the target row (third row) in the next stage. As a result, the parallel shift of a row in a memristor crossbar requires three cycles to complete.

By default, the logic NOR operation is column-wise, which means that NOR is performed between columns, where multi-bit binary numbers of an input row-vector are stored in a single row with the typical row-wise storage format (see Fig. 2(a)). The two dimensional input matrix is stored as multiple row vectors, each stored in one row of the crossbar. In general, a $M \times N$ matrix of w-bit binary numbers is stored in $M \times Nw$ cells of a crossbar memory.

3.2. DigitalPIM method

Now let us take 1-bit addition operation as an example. Their Boolean operations in terms of NOR can be shown in (7) where x_i and y_i are two input bits, c_{i-1} is carry in and c_i is carry out, a_i is addition result, b_i is borrow out, s_i is subtraction result.

$$c_i = \overline{\overline{(x_i + y_i)} + \overline{(x_i + c_{i-1})} + \overline{(y_i + c_{i-1})}} \tag{7a}$$

$$a_i = \overline{\overline{(x_i' + y_i' + c_{i-1}')} + \overline{(x_i + y_i + c_{i-1})} + c_i} \tag{7b}$$

$$b_i = \overline{\overline{(x_i + y_i')} + \overline{(x_i + b_{i-1})} + \overline{(y_i' + b_{i-1})}} \tag{7c}$$

$$s_i = \overline{\overline{(x_i' + y_i + b_{i-1}')} + \overline{(x_i + y_i' + b_{i-1})} + b_i} \tag{7d}$$

Cycle →			2	3	4	5	6	7	8	9	10
d_1	d_2	d_3	m_1	m_2	m_3	m_4	m_5	m_6	m_7	r_1	r_2
x_i	y_i	c_{i-1}	$\overline{x_i + y_i}$	$\overline{m_1 + x_i}$	$\overline{m_1 + y_i}$	$\overline{m_2 + m_3}$	$\overline{m_4 + c_{i-1}}$	$\overline{m_5 + m_4}$	$\overline{m_5 + c_{i-1}}$	$\overline{m_6 + m_3}$	$\overline{m_1 + m_5}$
										a_i	c_i

(a)

Cycle →			2	3	4	5	6	7	8	9	10	11
d_1	d_2	d_3	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	r_3	r_4
x_i	y_i	b_{i-1}	$\overline{y_i}$	$\overline{x_i + y_i}$	$\overline{m_1 + x_i}$	$\overline{m_1 + \overline{y_i}}$	$\overline{m_2 + m_3}$	$\overline{m_4 + b_{i-1}}$	$\overline{m_5 + m_4}$	$\overline{m_5 + b_{i-1}}$	$\overline{m_6 + m_3}$	$\overline{m_1 + m_5}$
											s_i	b_i

(b)

Cycle →				2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
d_1	d_2	d_3	d_4	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}	m_{12}	m_{13}	r_1	r_2	r_3	r_4
x_i	y_i	c_{i-1}	b_{i-1}	$\overline{y_i}$	$\overline{x_i + y_i}$	$\overline{x_i + \overline{y_i}}$	$\overline{m_2 + x_i}$	$\overline{m_2 + y_i}$	$\overline{m_4 + m_5}$	$\overline{m_6}$	$\overline{t_i + c_{i-1}}$	$\overline{m_8 + t_i}$	$\overline{m_8 + c_{i-1}}$	$\overline{t_i + b_{i-1}}$	$\overline{m_{11} + \overline{t_i}}$	$\overline{m_{11} + b_{i-1}}$	$\overline{m_9 + m_{10}}$	$\overline{m_8 + m_2}$	$\overline{m_{12} + m_{13}}$	$\overline{m_{11} + m_3}$
									t_i	$\overline{t_i}$							a_i	c_i	s_i	b_i

(c)

Fig. 3. Computing in memory with sequential cycle of MAGIC-NOR: (a) 1 bit full addition, (b) 1 bit full subtraction, (c) 1 bit MAGIC-DHT-1D and shared intermediates between addition and subtraction.

DigitalPIM, as proposed in [20], can implement *addition* using *NOR* (single cycle) operation as shown in (7a) and (7b). As we can see, computing sum (a_i) and carry-out (c_i) of 3 inputs (x_i, y_i, c_{i-1}) require 9 *NOR* cycles with additional one cycle to preset output memristors. Fig. 3(a) shows the 9 NOR operations that start from second cycle in order to calculate one bit full addition in memory. The first cycle is used to set all the output memristors to R_L state before logic NOR operation can take place. The columns in pink indicate intermediate results and the columns in orange indicate the inputs and final results. This can further be extended to calculate binary borrow-out (b_i) and subtraction (s_i) using 2's complemented binary numbers as (7c) and (7d) which is shown in Fig. 3(b). Thus, calculating 2-point Hadamard transform in *DigitalPIM* involves addition and subtraction of two w-bit binary numbers that can be calculated with $19w$ MAGIC NOR cycles ($21w$ cycles in total). The total memory footprint to perform full addition and subtraction of w bit inputs is $15w$ cells using this method.

3.3. MAGIC-DHT-1D method

The 2-point DHT computation in memory requires separate addition and subtraction of two w-bit numbers where each bit requires 21 cycles (19 cycles for NOR operation and 2 cycles for re-initialization of output memristors) to compute addition (a_i), subtraction (s_i), carry-out (c_i) and borrow-out (b_i) in *DigitalPIM* architecture. We observe that some intermediate results generated in (7) during computation of addition and subtraction are the same. Therefore, they can be reused to reduce computational cycles. As a result, we propose a new algorithm that simultaneously computes the full addition and subtraction of each bit by reusing those intermediate results as in (8). The resulting computing sequence, called *MAGIC-DHT-1D*, is shown in Fig. 3(c) in which cycles 3 and $5-7$ are shared between addition and subtraction operation. As a result, only $17w$ column-wise NOR cycles are required to compute the full addition and subtraction of each bit in the proposed architecture, versus $19w$ column-wise NOR cycles required in *DigitalPIM*. Also,

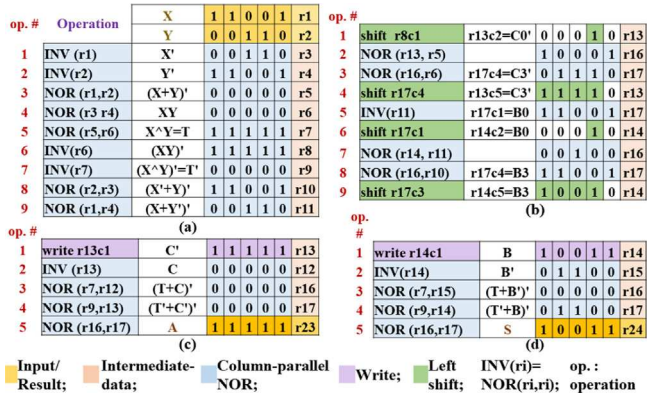

Fig. 4. Accelerated computing for multiple-bit addition and subtraction for inputs X and Y for $w = 5$. (a) 9 *NOR* operations independent of carry and borrow. (b) carry and borrow calculation with $(4w-1)$ *NOR* operations and $2w$ left shift operations (shown here: $C0, B0, C3, B3$.). (c-d) final (8 *NOR* + 2 write) cycles to compute $(X+Y)$ and $(X-Y)$ respectively. ri and ci denotes row number from $i = 1, \ldots, 24$ and column number from $i = 1, \ldots, w(bitwidth)$, respectively.

memory footprint (in terms of number of intermediate cells utilized) reduces from $15w$ cells in *DigitalPIM* to $13w$ cells in *MAGIC-DHT-1D* method. Note that, the number of cells for inputs (d_i) and outputs (r_i) is same in all methods. Therefore, they are not significant in comparative analyses of these methods in Fig. 3.

$$t_i = \overline{\overline{(x_i + y_i)} + x_i} + \overline{\overline{(x_i + y_i)} + y_i} \tag{8a}$$

$$c_i = \overline{\overline{(x_i + y_i)} + \overline{(t_i + c_{i-1})}} \tag{8b}$$

$$b_i = \overline{\overline{(x_i + y_i')} + \overline{(t_i' + b_{i-1})}} \tag{8c}$$

$$a_i = \overline{\overline{(t_i + c_{i-1})} + t_i} + \overline{\overline{(t_i + c_{i-1})} + c_{i-1}}; \tag{8d}$$

$$s_i = \overline{\overline{(t_i' + b_{i-1})} + t_i'} + \overline{\overline{(t_i' + b_{i-1})} + b_{i-1}} \tag{8e}$$

3.4. MAGIC-DHT-2D method

Now, we discuss the 2D DHT operation using the NOR-based computation in memory. In the *DigitalPIM* and *MAGIC-DHT-1D* method we have just proposed, we always perform the column-wise NOR operation sequentially to add/subtract multiple-bit numbers. One observation is that for arithmetic operation over multiple-bit data, there are some common intermediate results in add-sub operation. These intermediate results are independent of carry-out or borrow-out of previous bits and hence can be computed in parallel. As shown in Fig. 4, the first 9 *NOR* cycles for each bit of a binary number can be computed in parallel as they are independent of carry-out or borrow-out from the previous bit. As a result, we can use additional rows to compute and store those intermediate results as shown in Fig. 4 to enable parallel computing. After that, the carry-out and borrow-out of each bit are computed and shifted to the next bit position with $(4w-1)$ *NOR* cycles and $2w$ left shift cycles, where w is the bit-width of a binary number. Note that, in Fig. 4(b), we have shown the computation steps for $C0, C3, B0, B3$. Here, Ci and Bi are carry-out and borrow-out of bit-position i. The state of columns in row $r13, r14, r17$ changes during computation of $C1$ and $C2$. The computational steps for $C1, C2$ are same as that for $C3$ and the computational steps for $B1, B2$ are same as that for $B3$. Two write operations are required to reset column $c0$ in row $r13$ to '0' (carry in for LSB) and set column $b0$ in row $r14$ to '1' (borrow-in for LSB). Once all carry and borrow bits are ready, the final add-sub results can be calculated with 8 additional *NOR* cycles. As a result, we can further reduce the number of the required cycles as shown in Fig. 4.

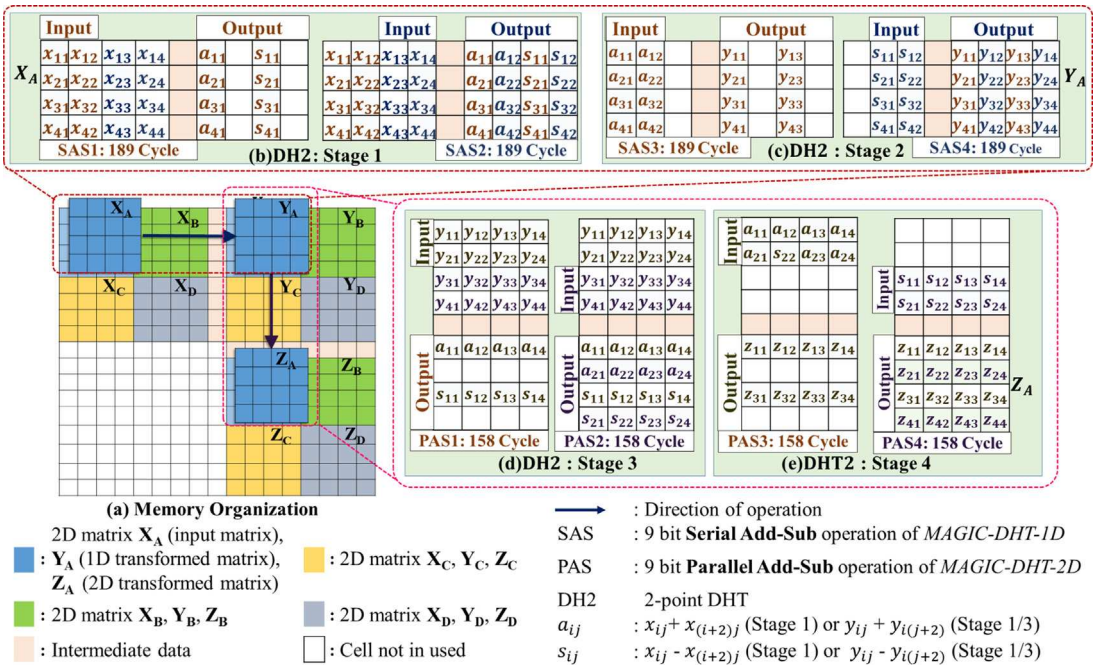

Fig. 5. Parallel computation of 4 × 4 point 2D-DHT in of 4 sets of data matrix X_A, X_B, X_C, X_D (each matrix is 4 × 4) with *MAGIC-DHT-2D* for $w = 9$. (a) Organization of data in memory; (b) Stage 1: 2-point DHT with *MAGIC-DHT-1D* to computes the sum a_{ij} and subtraction s_{ij} from x_{ij} in each row. (c) Stage2: 2-point DHT with *MAGIC-DHT-1D* over a_{ij} and s_{ij} to computes Y_A; (d) Stage 3: 2-point DHT with multi-bit add-sub over y_{ij} in each column to compute a_{ij} and s_{ij}; (e) Stage 4: 2-point DHT with column-wise multi-bit add-sub computes Z_A from a_{ij} and s_{ij}.

The second improvement is that now we can leverage the column-wise and row-based NOR computing to naturally avoid matrix transposition operation in the 2D DHT. Specifically, as shown in (6), the 2D $M \times N$-point Hadamard transform can be simply performed by computing two 1D DHT with bit-serial add-sub operation as in either *DigitalPIM* or *MAGIC-DHT-1D*. After computing the first 1D N-point DHT on M row vectors via column-based NOR operations, the matrix is transposed. Then, another 1D DHT is performed on all M-element vectors stored in N rows. However, as memristors have large write latency, the matrix transpose operation can be computationally expensive. A possible alternate method is to store the 2D data matrix by converting it into a 1D data vector with the row-major algorithm. However, this method requires a crossbar with a large column size as N or M increases and therefore, may not be achievable with practical memristor devices.

To mitigate this problem, we propose a combined row and column-based 1D DHT for $M \times N$-point 2D DHT as demonstrated in Fig. 5. In this figure, the input data are organized as four tiles X_A, X_B, X_C and X_D (each of them are 4 × 4 matrix). Each small square has nine memristor cells arranged horizontally and it stores an input number in 9-bit binary 2's complemented format. The upper block represents the $Y = \mathbf{H}_M X$ operation. In the first phase, Y_A and Y_C are computed in parallel via column-wise operation. This parallel computation consists of two stages of computing the addition and subtraction among X based on the DHT rule. These 2-point DHT computation stages (denoted as DH2) are broken down into four SAS (9-bit serial addition and subtraction) steps as in Fig. 5(b–c). Each stage takes $189 + 189 = 378$ cycles to finish as we have 9-bit data and each bit takes 21 cycles to finish in column-wise *MAGIC-DHT-1D*. Then Y_B and Y_D are computed in the next 378 cycles using the same method.

In the second phase, the bottom-right block represents the $Z = \mathbf{H}_M Y^T$ operation. In this case, row-wise NOR operations are performed, which also consists of two stages of multi-bit parallel addition and subtraction computing for Z_A and Z_B and another two stages for Z_C and Z_D. These 2-point DHT computation stages are broken down into four PAS (9-bit parallel addition and subtraction) steps as in Fig. 5(d–e). Each stage takes $158 + 158 = 316$ cycles to finish using multi-bit parallel computing as mentioned earlier. In total, four 4 × 4 2D data matrix can be encoded with 4 × 4 2D DHT in crossbar memory within $378 \times 2 + 316 \times 2 = 1388$ cycles. As a result, no matrix transpose is required for full 2D DHT computation in *MAGIC-DHT-2D*, which leads to the saving and speedup as shown in our numerical result sections.

3.5. Memory organization

Fig. 6 shows the memory organization for the proposed Hadamard encoder in *MAGIC-DHT* system. In this design, blocks of the crossbar memory array are arranged into T tiles. Each tile contains $B \times B$ blocks, where each block has one crossbar array of size $C \times C$ along with a set of pass transistors for shift/copy operations, sense amplifiers for reading, and finally, a set of read–write buffers with multiplexers for writing. In the proposed design, a tile with 16 × 16 blocks ($B = 16$) is selected, with each block having a crossbar array of size 1Mb ($C = 1024$). The read–write buffer is designed to read data from both the crossbar and global data bus, and is connected to a write multiplexer to provide appropriate voltage across the crossbar bitlines for writing data stored in the buffer to the selected row of the crossbar. Each wordline of a crossbar array is connected to an output signal line of a row driver. One end of each bit-line is connected to a column driver. The other end of each bit-lines is connected to a sense amplifier and read–write buffer through pass transistors for copy or shift operation.

The crossbar is controlled by a memory controller unit, which includes a control module, row and column drivers, decoders, and block selectors. The input to the memory controller consists of control signals and five address ports. The outputs of the control module is the selector signals for row and column driver-multiplexer and activation signal for sense amplifiers, buffers, and blocks. The control bus has a three-bit input-type port, named **operation**, that specifies whether memory operations (such as shift/copy/write) or logic operations (such as NOR/INV/set/reset) will be performed in the crossbar. The **row_col'** port specifies the direction of the logic operation, while the **sl_sr'** port specifies the direction of shifting during the shift operation. During shift operation, **sl_sr'** is set to one for left shifting and it is reset to zero for

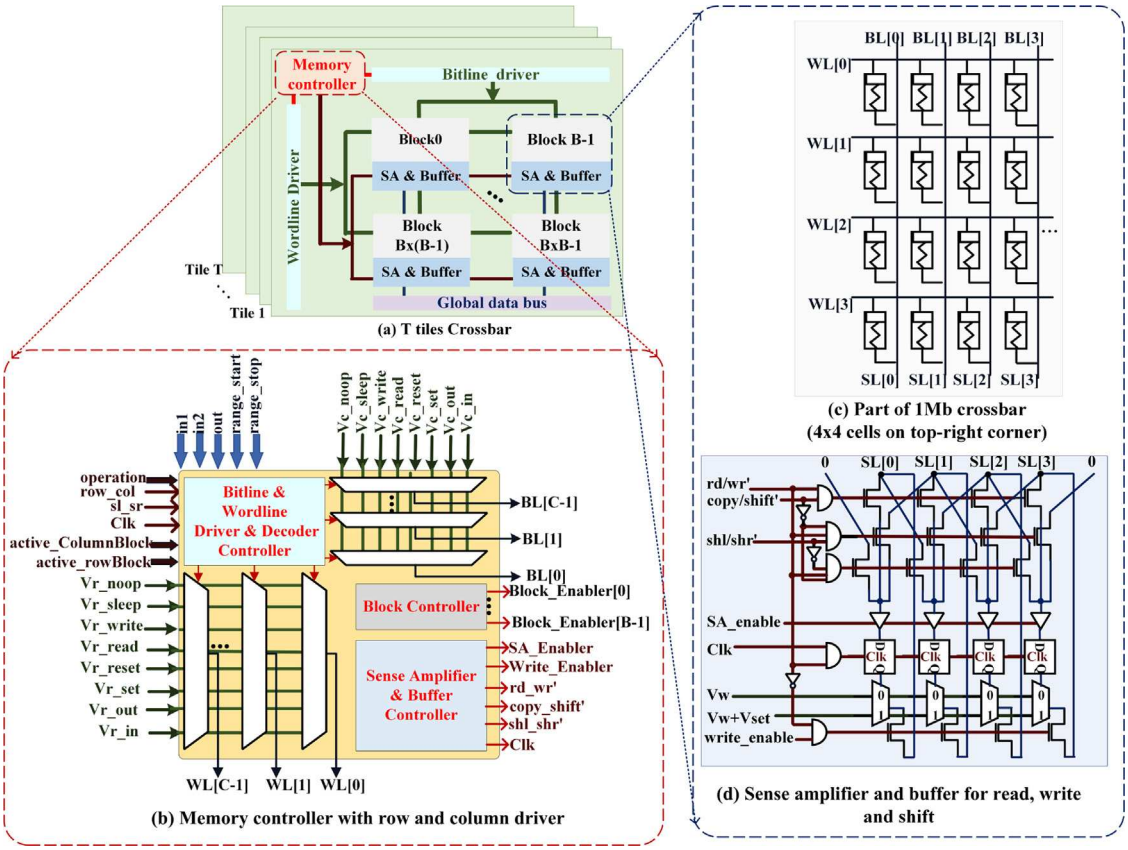

Fig. 6. Memory architecture for proposed in-memory Hadamard encoder: (a) T Tiles of crossbar memory where each tile contains $B \times B$ number of crossbars and each crossbar is 1024 × 1024 in size; (b) Organization of memory controller unit for wordline & bitline driver & decoder, sense amplifier, and buffer and block selector; (c) Top-right 4 × 4 cells in a 1024 crossbar; (d) Bottom peripherals circuits connected to signal line ($SL[i]$ in (c)) with shifter, sense amplifier, and read/write buffer.

right shifting the specified wordline before storing it into the read–write buffer. The **active_ColumnBlock** and **active_RowBlock** ports are B bits each and denote which blocks in a tile will be activated for parallel operation. For example, **active_ColumnBlock** = [0111 1000 0000 000] and **active_rowBlock** = [1110 0000 0000 000], then second, third, fourth and fifth blocks of first 3 rows will be activated for that particular operation. Other than these 12 blocks, the rest of the blocks in the tile will be deactivated.

The address bus consists of five ports, each $log_2 C$ bits long. During a logic operation, the **in1** port denotes the wordline/bitline address of the first (only) input of the NOR (INV) operation, while **in2** denotes the address of the second input of the Boolean NOR operation. During a memory operation, **in1** denotes the address of the row to be read to or written from the read–write buffer. The port named the **out** port denotes the address of the output wordline/bitline of the logic operation. The **range_start** and **range_stop** ports are used to activate only a selective part of the crossbar for parallel operation instead of activating the entire crossbar during two-dimensional DHT computation in memory. The memory-logic controller controls the bitlines by connecting them to either the column/bitline driver for logic operations or the sense amplifier-buffer for memory operations.

We remark that the proposed PIM DHT computation can compute DHT for large N due to its inherent recursive nature. With multiple crossbars, each crossbar can compute up to thirty 32 × 32 DHTs in parallel. The computed output matrix from different crossbar within a tile can then be properly transferred among/within these crossbars with proper read/write peripherals and data-bus interconnection among crossbar blocks. The latency of such memory transfer will be according to the global read–write and interconnection scheme among crossbars in memory tile. After proper transfer of data among multiple crossbar within a tile of memory, the computed 32 × 32-point DHT outputs can be passed through similar consecutive cycles of 2-point DHT, and complete the computation of DHT for large N. Thus our proposed architecture is suitable for recursive computation of DHT for any large value of N by utilizing proper data transfer among multi-crossbar.

4. Experimental results and discussions

4.1. Experimental setup

In this section, we present the experimental results. We have implemented the crossbar memory array in CADENCE Virtuoso. The memristor cells of the crossbar was designed with VTEAM model in Verilog-A which has relatively high accuracy (below 1.5% RMS error) and is computationally more efficient as compared with existing memristor models [36]. The parameters of the memristors are chosen to produce a switching time of 1.1 ns for a voltage pulse of 1.5 V for RESET and 2.5 V for SET. The memory peripherals circuits including shifter, sense amplifier, buffer and write-multiplexer was implemented using FinFET 15 nm technology. We have designed a memory tile of 16 × 16 crossbar array controlled by a single memory controller as in [29]. The *MAGIC-DHT* crossbar design parameters are given in Table 1. The memory controller was designed and synthesized with Synopsys Design Compiler.

We have further verified the functionality of the ReRAM crossbar as both a processing and memory element by simulating several consecutive read, write, Boolean NOR and Boolean NOT operation in Cadence Virtuoso ADE. After each such operations, the respective row was read back into the read-buffer in bottom peripheral circuit with proper voltage applied to word/bit lines and peripherals. We have also measured the current through and voltage across both active and inactive memristors to verify the impact of leakage current on state

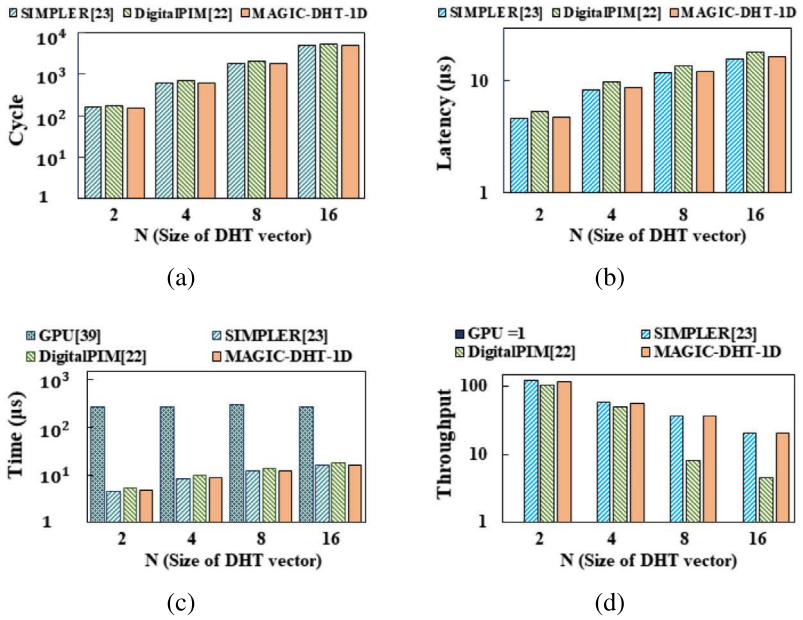

Fig. 7. Performance comparison for 1D DHT (a) # of cycle to compute single N-point DHT to encode $1.125N$ kilobytes of data in memory (b) Time to encode all input data stored 1Mb crossbar. (c) Time to encode USC-SIPI-misc dataset. (d) Throughput normalized to GPU.

Table 1
MAGIC-DHT design parameter.

Memristor design parameter		*MAGIC-DHT* design parameter	
k_{on}	-216.2 ms^{-1}	Crossbar block size	1024 × 1024 memristor
k_{off}	91 ms^{-1}	Tile size	16 × 16 Blocks
α_{on}	4	V_{set}	2.5 V
α_{on}	4	V_{reset}	1.5 V
x_{on}	0 nm	V_{shift}	3 V
x_{off}	3 nm	$V_{r/c,in}$	1.7 V
$V_{T,on}$	−1.5 V	$V_{r/c,out}$	0.1 V
$V_{T,off}$	0.3 V	$V_{c/r,sleep}$	0.3 V
R_{on}	1 kΩ	$V_{r,read}$	3 V
R_{off}	300 kΩ	$V_{r,write}$	3.5 V

of memory cells. We have observed a very small distortion in the state of inactive cells over several cycles. However, in digital ReRAM the binary states are further apart from each other ($R_{ON} \ll R_{OFF}$). Therefore, such small change of resistive state will be able switch digital state of the memristors only after a very long period of processing. The column-wise logic operation was performed with 25%, 50%, and 100% wordlines activated while the rest of the wordlines were given the voltage V_{sleep} and the average energy and latency of parallel logic operation in each case were calculated. Tables 2 and 3 show the execution time and energy consumption for each basic memory and logic operation in crossbar block for our simulation.

4.2. Implementation and comparison methods

We have designed a Python-based simulator that generates appropriate control word for memory controller in each cycle to calculate both $1D$ and $2D$ N-point DHT and calculates the state of memristor cells in crossbar given various configurations of control voltages from memory controller. The simulator also calculates the total number of logic and memory cycles, as well as energy and cell area, occupied in each crossbar memory block for parallel encoding of the stored data with N-point DHT. We have implemented both *MAGIC-DHT-1D* and *MAGIC-DHT-2D* as well as *DigitalPIM* [20] for comparison. Furthermore, we also compare the proposed method with the *SIMPLER MAGIC* or *SIMPLER* [21] method, which is the PIM optimizer for general logic functions based on mathematical programming. In *SIMPLER* method, we have first implemented the N-point 1D DHT and $N \times N$-point 2D DHT encoder in Verilog HDL and provided that to *SIMPLER* tool. The *SIMPLER* tool provides a synthesized netlist for crossbar memory optimized with both shared intermediate cycles with fused addition–subtraction similar to our proposed method, and memory mapping. However, algorithm for the memory mapping and reuse of intermediate cells are different in *SIMPLER* from our proposed method. In *SIMPLER* method, the cells to be reset and reused in a single cycle are not structured to be consecutive and hence, it requires more complex peripheral architecture and memory controller unit. Therefore, our numerical results show the difference in latency, area and energy consumption between our proposed method optimized for both 1D and 2D DHT and SIMPLER method. Furthermore, in *SIMPLER* method, the 2D DHT matrix is rolled out as row vector and stored in single row for DHT computation. This requires significantly large row-size for crossbar memory for $N > 4$ which is not practical for current memristor technology. We have simulated the theoretically correct latency, area and energy for a crossbar architecture for *SIMPLER* with the row size of minimum required cell usage for $N \times N$-point DHT and a column size of such that the total size of the crossbar remains one megabit.

4.3. Digital DHT implementation in GPU

We also compare the performance of our proposed architecture for in-memory Hadamard encoder with that of NVIDIA RTX Titan X GPU with 24 GB memory. For our GPU based implementation, we have modeled our kernels for calculating N-point DHT according to [37]. We have programmed a host code to arrange the dataset of multiple two dimensional images into an array of greyscale pixel values arranged in the row-major order. The array has the dimension of $1 \times I.P^2$, where I is the number of images in the dataset and P^2 is the number of pixels of each image. After that, the GPU kernel is launched that calculates one dimensional N-point DHT over the entire input dataset. Note that, the input data array is virtually divided into multiple subsets, and the number of subsets for Hadamard transform will vary with the value of N and the dimension of transformation employed. For example, an image of size $P \times P$ is divided into $\frac{P^2}{N}$ subsets for computing N-point 1D DHT. We have calculated the execution time of the kernel as the latency of computing 1D N-point DHT in GPU. The 2D $N \times N$ point DHT is computed in GPU by launching the kernel for 1D DHT computation

Table 2
Execution time for basic operation in crossbar.

Row-parallel NOR	Column-parallel NOR	Write	Read/Copy	Shift	Set	Reset
1.1 ns	1.1 ns	2.5 ns	1.1 ns	4 ns	1.1 ns	1.1 ns

Table 3
Energy consumption for basic operation in crossbar.

Active area	2-input NOR		INV		Write		Read		Reset		Set	
	Total (pJ)	Per cell area (fJ)	Total (pJ)	Per cell area (fJ)	Total (nJ)	Per cell area (fJ)	Total (nJ)	Per cell area (fJ)	Total (nJ)	Per cell area (fJ)	Total (nJ)	Per cell area (fJ)
25%	3.07	2.99	3.52	3.43	18.24	15 354.17	24	37 229	6.96	26.55	8.192	31.25
50%	7.68	7.5	10.24	10	17.92	14 797.67	26.24	37 583	14.44	27.35	15.97	30.5
100%	19.84	19.34	25.6	25	16	13 125	27.2	37 396	37.27	35.55	43	41

Table 4
Size of input data stored in each crossbar memory to be encoded with DHT (kilobytes).

	N	2	4	8	16
1D	SIMPLER	58.5	54	45	36
	DigitalPIM	56.25	54	54	36
	MAGIC-DHT-1D	56.25	54	54	36
2D	SIMPLER	54	36	31.64	32.34
	DigitalPIM	27.36	23.78	16.88	16.88
	MAGIC-DHT-2D	27.36	23.78	16.88	16.88

twice. First, a 1D N-point DHT is computed in GPU kernel over the entire input array that has been arranged in the row-major order. Then, a matrix transpose operation is performed in CPU by rearranging the 1D DHT transformed array of pixels in the column-major order. After that, another 1D N-point DHT is computed in GPU kernel over the newly arranged input array. In order to compare the latency of computing 2D DHT between GPU and PIM based architectures, only the execution time of two GPU kernels are calculated. Thus, the time for matrix transposition performed in CPU or any other data movement between GPU device and CPU host is ignored while we compute the latency of DHT encoding in GPU. The GPU kernels for computing DHT in the CUDA source code [37] is optimized for large value of N and the latency of computation largely depends on size of data vector/matrix. Since the size of dataset is same for all values of N in our experiment, the latency does not changes for value of $N < 10^{11}$. The throughput of computing both 1D (2D) N-point ($N \times N$-point) Hadamard transform is also calculated by dividing the kernel execution time by number of subsets of input data array over which the transformation has been performed.

4.4. Dataset comparison

To further evaluate our design we have encoded 22 greyscale images of different resolutions from USC-SIPI-misc dataset [38] in *MAGIC-DHT*, *SIMPLER* [21], *DigitalPIM* and NVIDIA Titan-X GPU. Each pixel is represented as an 8-bit signed integer ranging from −127 to 127. The image data is stored in the memristive array in $2's$ complemented format and extended to a total of 9 bits to accommodate for correct computation of carry-out and borrow-out in MSB. So the default bit-width is 9-bit in the following experiments.

4.5. Latency and throughput comparison

4.5.1. One-dimensional DHT comparison

Fig. 7(a) shows the number of cycle to compute single N-point one-dimensional Hadamard transform for $w = 9$-bit width data in *MAGIC-DHT-1D*, *SIMPLER* [21] and *DigitalPIM* [20]. As shown in the figure, both *MAGIC-DHT-1D* which is optimized with intermediate data sharing and *SIMPLER* which is optimized for minimum latency and maximum throughput, require fewer cycle than *DigitalPIM* to compute a single set of 1D N point DHT in memory for $N = 2, 4, 8$ or 16. In each crossbar the number of input data vectors stored in each row is $S_{row}(N) = \lfloor \frac{(1024-intermediate\ cells)}{N(2w+log_2^N)} \rfloor$. Therefore, $S_{row}(N)$ number of 1D N-point 1D DHT is performed sequentially in a single crossbar. The number of required intermediate cells increases with N and they are reused for each set of data-vector. The size of input data stored and encoded with 1D DHT for each value of N, and for each method is shown in Table 4. As before data stored in all 1024 rows is encoded in parallel in all three methods. Fig. 7(b) shows the time to encode $1.125 S_{row}(N) \times N$ kilobytes of data in a 1024×1024 crossbar. Latency for 1D DHT computation in a single memory block is low in *MAGIC-DHT-1D* which is on average 1.12× and 0.97× faster than *DigitalPIM* and *SIMPLER*. As *SIMPLER* applies similar optimized data and logic mapping in crossbar memory, the latency of computing a single 1D DHT vector in *MAGIC-DHT-1D* is very close to that in *SIMPLER*

Fig. 7(c) shows the time to encode the USC-SIPI-misc dataset of greyscale images in crossbar memory blocks. The images are stored across 256 blocks of memory in a single tile. We also include the time from our GPU implementation. As we can see, the proposed *MAGIC-DHT-1D* provides around 56.19×, 30.86×, 24.25× and 16.67× speed up over GPU for 2-point, 4-point, 8-point and 16-point 1D DHT operations respectively.

Fig. 7(d) shows the normalized throughput of in-memory Hadamard encoding in the proposed architecture compared to that in GPU. The throughput has been calculated for 256 blocks of 1024×1024 crossbar memory operating in parallel in a single tile with maximum hardware utilization. All methods for in-memory DHT computation provide significant improvement in terms of throughput compared to GPU. On average *MAGIC-DHT-1D* has 1.37×, 0.96× and 57.84× higher throughput over *DigitalPIM*, *SIMPLER* and GPU for 1D Hadamard encoding of data.

4.5.2. Two-dimensional DHT comparison

The number of cycles required to compute a single 2D $N \times N$-point DHT using three in-memory computation methods, namely *MAGIC-DHT-2D*, *SIMPLER*, and *DigitalPIM*, are shown in Fig. 8(a). *MAGIC-DHT-2D* has the lowest computation cycles for any value of N and provides a significant improvement over other processing-in-memory (PIM) methods. This speed-up is due to the multi-bit parallel arithmetic operations between rows and bit-serial arithmetic operations between columns performed in *MAGIC-DHT-2D* as explained in Section 3.4. In *DigitalPIM*, the $N \times N$-point DHT is computed through recursive computation of 2-point DHT, where bit-serial addition and subtraction is performed on each pair in both directions. On average, *MAGIC-DHT-2D* provides speed-up of 7.18× and 4.85× over *DigitalPIM* and *SIMPLER* respectively, to encode $1.125N^2$ bytes of data in memory.

The number of input data matrix stored in *MAGIC-DHT-2D* and *DigitalPIM* crossbar for 2D DHT encoding can be formulated as $\lfloor \frac{1024-metarow}{2N} \rfloor \times \lfloor \frac{1024-metacolumn}{N(2w+\log_2^N)} \rfloor$, where the number of required intermediate cells (also known as metacolumn/metarow/metacells) increases with the value of N. Thus, the amount of input data that can be stored

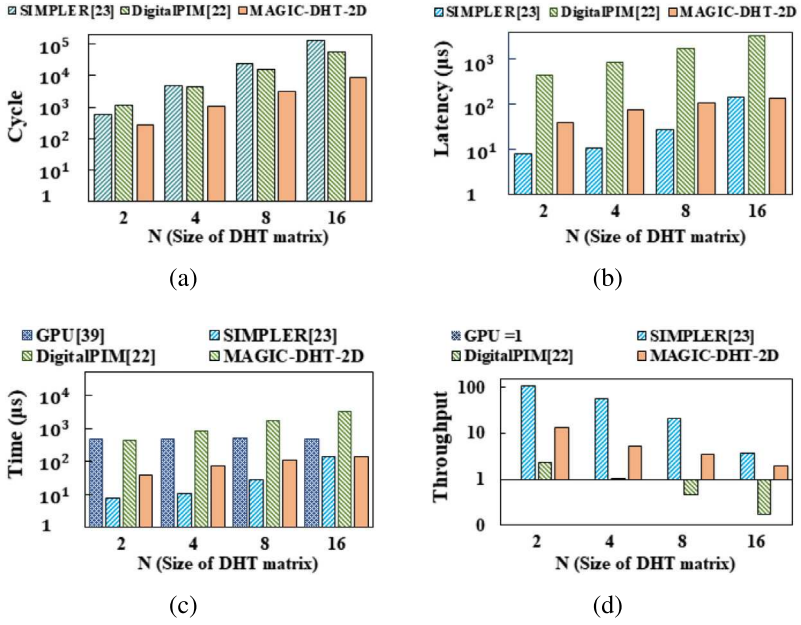

Fig. 8. Performance comparison for 2D DHT (a) # of cycle to encode single data matrix with $N \times N$-point 2D DHT. (b) Time to encode data in 1Mb memory block. (c) Time to encode USC-SIPI-misc image-set. (d)Throughput normalized to GPU.

and encoded in each crossbar block decreases in general as the value of N increases, as shown in Table 4.

We have simulated the 2D N-point DHT in *SIMPLER* and found that the cell area required for $N > 4$ is much higher than 1024, as the 2D data matrix is rolled out into a 1D vector with the row-major algorithm. The required minimum column size in *SIMPLER* is not physically possible to achieve with current memristor technology without facing severe distortion in signal. However, in order to compare the throughput of 2D DHT with our proposed architecture (which has been designed to optimally compute 2D DHT in 1024 × 1024 crossbar for values of N up to 32), we have assumed a theoretical design of crossbar for *SIMPLER* where the column size is equal to the minimum area required to compute N-point 2D DHT (calculated with the *SIMPLER* tool) and the row size is $\frac{(1024\times1024)}{(column\ size)}$.

The latency of computing the 2D $N \times N$-point DHT over all data stored in a one megabits of memory block is shown in Fig. 8(b). On average, *MAGIC-DHT-2D* has 15.75× and 0.41× lower latency compared to *DigitalPIM* and *SIMPLER* model, respectively, while computing 2D DHT in memory. The amount of data encoded in parallel is higher in *SIMPLER* as shown in Table 4. This leads to a relatively lower improvement in the throughput of *MAGIC-DHT-2D* over *SIMPLER*.

On average, *MAGIC-DHT-2D* provides around 5.96× higher throughput than *DigitalPIM*. Fig. 8(c) shows the time to encode the USC-SIPI-misc dataset with 2D N-point Hadamard transform both in memory and in GPU. Again, as expected, *MAGIC-DHT-2D* can encode the entire dataset 6.90× faster than GPU. However, as the value of N increases exponentially, the 2D DHT encoding time in all PIM-based architectures also increases accordingly. But the execution time of DHT encoder kernels in GPU almost remains constant for any value of $N < 2^{11}$. The slow change in the execution time of GPU kernel is due to the optimized design to handle large datasets. Hence, the GPU kernel execution time depends mainly on the size of the input dataset, rather than the value of N. Therefore, the PIM architectures become slower than GPU while encoding data with 2D DHT as the value of N increases. Although *MAGIC-DHT* is still faster than GPU, the speedup of DHT encoding in *MAGIC-DHT* over GPU reduces with the increased value of N

The throughput of encoding data with $N \times N$-point 2D DHT in *MAGIC-DHT-2D*, *DigitalPIM* and *SIMPLER* normalized to that in GPU are shown in Fig. 8(d). In general, the proposed *MAGIC-DHT-2D* can provide up to 5.96× improvement over GPU in terms of 2D DHT throughput.

In general, that the proposed *MAGIC-DHT* architecture has better performance in terms of cycle count, latency, encoding time, and throughput compared to the other methods. In particular, the latency of the *MAGIC-DHT* is faster than *DigitalPIM*, and it is close to *SIMPLER*. The encoding time of *MAGIC-DHT* is much faster than the GPU implementation for various DHT operations. The throughput of *MAGIC-DHT* is also mostly higher than the other methods, including the GPU.

4.6. Area and energy comparison

The performance of *MAGIC-DHT* compared to that of *SIMPLER* is further evaluated by computing energy density (energy per unit cell area), area delay product (where ADP is defined as # of cell × latency of each DHT in the crossbar), energy-delay product (where EDP is defined as energy× latency of each DHT in crossbar) in both methods. The optimization method employed during the implementation of the DHT encoder in the ReRAM crossbar is shown in Column 3 of Table 5. For each value of N, a 1D and 2D DHT encoder is designed with two optimization algorithms in both *SIMPLER* and *MAGIC-DHT*. The first algorithm minimizes the cell area occupied, while the second minimizes latency. Note that, the number of cell area can be reduced with expense of increased latency and vice versa in both method. To compare the performance of *MAGIC-DHT* and *SIMPLER*, the latency, area, energy, EDP, and ADP of *SIMPLER* are divided by those of *MAGIC-DHT-1D*. On average, *MAGIC-DHT* provides a speedup of 0.99× over *SIMPLER* with a maximum speedup of 1.01× for 1D-DHT. The mean cell area occupied by *SIMPLER* is 0.98× to that of *MAGIC-DHT-1D*, while the energy consumption of *SIMPLER* is 1.003× to that of *MAGIC-DHT-1D*. This results in a 1.07× higher energy density in *SIMPLER* over that in*MAGIC-DHT*.

Similarly, the latency, area, energy, EDP, and ADP of *SIMPLER* are divided by those of *MAGIC-DHT-2D* to compare their performance for 2D DHT. On average, *MAGIC-DHT* provides a speedup of 7.25× over *SIMPLER*, with a maximum speedup of 15.04× for 16-point 2D DHT. The mean cell area occupied by *SIMPLER* is 0.32× that of *MAGIC-DHT-2D*, but the energy density of *MAGIC-DHT* is 3.18× better than *SIMPLER*. Also note that, the entire cell area of *SIMPLER* is the minimum row size of crossbar required for 2D DHT calculation which is very high for DHT

Table 5
Comparison between *MAGIG-DHT* and *SIMPLER [21]* for computing single N-point DHT in crossbar.

	N	Optimization	Latency (Cycle)		Area (# of cell)		Energy		Improvement					
			SIMPLER [21]	MAGIC-DHT	SIMPLER [21]	MAGIC-DHT	SIMPLER [21]	MAGIC-DHT	Latency	Area	Energy	Energy density	ADP	EDP
1D DHT	2	Min area	161	160	59	66	0.88	0.90	1.01	0.89	0.98	1.09	0.90	0.98
		Min latency	148	151	100	166	0.91	0.90	0.98	0.60	1.00	1.67	0.59	0.98
	4	Min area	621	627	140	144	3.49	3.56	0.99	0.97	0.98	1.01	0.96	0.97
		Min latency	580	594	200	244	3.53	3.56	0.98	0.82	0.99	1.21	0.80	0.97
	8	Min area	1853	1868	315	272	10.73	10.63	0.99	1.16	1.01	0.87	1.15	1.00
		Min latency	1786	1771	350	368	11.17	10.63	1.01	0.95	1.05	1.10	0.96	1.06
	16	Min area	4941	5061	628	528	28.84	28.86	0.98	1.19	1.00	0.84	1.16	0.98
		Min latency	4784	4804	800	624	29.06	28.86	1.00	1.28	1.01	0.79	1.28	1.00
		Average improvement over SIMPLER [23]							**0.99**	**0.98**	**1.003**	**1.07**	**0.97**	**0.99**
2D DHT	2	Min area	621	280	140	620	3.78	3.51	2.22	0.23	1.08	4.77	0.50	2.39
		Min latency	580	271	200	820	3.78	3.55	2.14	0.24	1.07	4.38	0.52	2.28
	4	Min area	4941	1105	628	1960	30.11	27.91	4.47	0.32	1.08	3.37	1.43	4.82
		Min latency	4784	1069	800	2360	30.11	28.11	4.48	0.34	1.07	3.16	1.52	4.79
	8	Min area	24 543	3299	2279	6416	149.54	167.03	7.44	0.36	0.90	2.52	2.64	6.66
		Min latency	24 397	3192	2325	7216	149.54	168.60	7.64	0.32	0.89	2.75	2.46	6.78
	16	Min area	129 459	8873	9191	23 008	788.78	899.43	14.59	0.40	0.88	2.20	5.83	12.80
		Min latency	129 175	8588	9300	24 608	788.78	911.32	15.04	0.38	0.87	2.29	5.68	13.02
		Average improvement over SIMPLER [23]							**7.25**	**0.32**	**0.98**	**3.18**	**2.57**	**6.69**

beyond $N = 4$, and not practical to achieve without major performance degradation in the crossbar. On the other hand, the cell area reported by *MAGIC-DHT-2D* for each 2D-DHT is distributed over several rows and columns. For example, for 16-point 2D DHT, the maximum row and column size required in *MAGIC-DHT-2D* is 66 and 639, respectively. Furthermore, the average improvement of ADP and EDP provided by *MAGIC-DHT-2D* is around 2.57× and 6.69×, respectively. On average *MAGIC-DHT* significantly improves over *SIMPLER* in terms of hardware resource consumption for 2D DHT computation.

5. Conclusion

In this project, we have developed an architecture for the parallel in-memory computation of discrete Hadamard transform with our *MAGIC-DHT* method, which is based on the recently proposed MAGIC digital in memory computing framework. To optimize the one dimensional DHT computation, we have introduced *MAGIC-DHT-1D*, which exploits the shared intermediate results and provides 1.12× speed-up over *DigitalPIM* and 0.97× over *SIMPLER*. We have also proposed *MAGIC-DHT-2D*, which accelerates the two dimensional DHT calculation by sharing the intermediate results with the same carrier among multi-bit add-subtraction operation and row and column based NOR computing. This method can provide up to 15.75× speed-up over *DigitalPIM* method. We have compared the cell area utilization and energy consumption of our proposed methods with those of *SIMPLER*. Our numerical results show that the average hardware resource consumption in *MAGIC-DHT* improves significantly over *SIMPLER* for 2D DHT computation. The energy density in *MAGIC-DHT* is on average 1.07× and 3.18× less than *SIMPLER* while computing 1D and 2D DHT, respectively. Furthermore, *MAGIC-DHT-2D* can provide an average EDP improvement of 6.69× over *SIMPLER* for computing 2D DHT. Finally, we have compared the performance of our proposed methods with that of the NVIDIA RTX Titan GPU. Our proposed *MAGIC-DHT-1D* for 1D Hadamard transform can lead to 56.19× speed-up and 57.84× higher throughput over the GPU. For two dimensional DHT, our proposed *MAGIC-DHT-2D* outperforms the GPU with up to 6.90× speed-up and 5.96× higher throughput.

CRediT authorship contribution statement

Maliha Tasnim: Conceptualization, Methodology, Experimental setup, Writing – original draft. **Chinmay Raje:** Data curation, Investigation. **Shuyuan Yu:** Investigation. **Elaheh Sadredini:** Supervision. **Sheldon X.-D. Tan:** Supervision, Writing – abstract, Introduction, Review and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] T. Kojima, G. Muto, A low inter-code interference Walsh-Hadamard code division multiplexing for helicopter satellite communications, in: 2018 International Conference on Advanced Technologies for Communications, ATC, 2018, pp. 1–4, http://dx.doi.org/10.1109/ATC.2018.8587496.

[2] A. Vard, A. Monadjemi, K. Jamshidi, N. Movahhedinia, Fast texture energy based image segmentation using directional Walsh–Hadamard transform and parametric active contour models, Expert Syst. Appl. 38 (9) (2011) 11722–11729.

[3] X. Wang, X. Liang, J. Zheng, H. Zhou, Fast detection and segmentation of partial image blur based on discrete Walsh–Hadamard transform, Signal Process., Image Commun. 70 (2019) 47–56.

[4] L.P. GG, S. Domnic, Walsh–Hadamard transform kernel-based feature vector for shot boundary detection, IEEE Trans. Image Process. 23 (12) (2014) 5187–5197.

[5] G. Banegas, P.S. Barreto, E. Persichetti, P. Santini, Designing efficient dyadic operations for cryptographic applications, J. Math. Cryptol. 14 (1) (2020) 95–109.

[6] R. Boules, Adaptive filtering using the fast Walsh-Hadamard transformation, IEEE Trans. Electromagn. Compat. 31 (2) (1989) 125–128, http://dx.doi.org/10.1109/15.18779.

[7] H. Pan, D. Badawi, A.E. Cetin, Fast walsh-hadamard transform and smooth-thresholding based binary layers in deep neural networks, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 4650–4659.

[8] V. Mannam, Low-energy convolutional neural networks (CNNs) using Hadamard method, 2022, arXiv preprint arXiv:2209.09106.

[9] S. Picek, L. Batina, D. Jakobović, B. Ege, M. Golub, S-box, SET, match: a toolbox for S-box analysis, in: IFIP International Workshop on Information Security Theory and Practice, Springer, 2014, pp. 140–149.

[10] W. Stein, Sage mathematics software, 2007, http://www.sagemath.org/.

[11] D. Bikov, I. Bouyukliev, Parallel fast Walsh transform algorithm and its implementation with CUDA on GPUs, Cybern. Inf. Technol. 18 (5) (2018) 21–43.

[12] A.D. Copeland, N.B. Chang, S. Leung, GPU accelerated decoding of high performance error correcting codes, in: Proc. 14th Annual Workshop on HPEC, 2009, pp. 15–16.

[13] M. Imani, S. Gupta, 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, http://dx.doi.org/10.1145/3061639.3062337.

[14] M. Imani, S. Gupta, T. Rosing, GenPIM: Generalized processing in-memory to accelerate data intensive applications, in: 2018 Design, Automation Test in Europe Conference Exhibition, DATE, 2018, pp. 1155–1158, http://dx.doi.org/10.23919/DATE.2018.8342186.

[15] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, Y. Xie, Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories, in: 2016 53nd ACM/EDAC/IEEE Design Automation Conference, DAC, IEEE, 2016, pp. 1–6.

[16] S. Gupta, M. Imani, T. Rosing, Felix: Fast and energy-efficient logic in memory, in: 2018 IEEE/ACM International Conference on Computer-Aided Design, ICCAD, IEEE, 2018, pp. 1–7.
[17] S. Chakraborti, P.V. Chowdhary, K. Datta, I. Sengupta, BDD based synthesis of Boolean functions using memristors, in: 2014 9th International Design and Test Symposium, IDT, 2014, pp. 136–141, http://dx.doi.org/10.1109/IDT.2014.7038601.
[18] R.B. Hur, N. Wald, N. Talati, S. Kvatinsky, SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic, in: 2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD, IEEE, 2017, pp. 225–232.
[19] N. Talati, S. Gupta, P. Mane, S. Kvatinsky, Logic design within memristive memories using memristor-aided loGIC (MAGIC), IEEE Trans. Nanotechnol. 15 (4) (2016) 635–650.
[20] M. Imani, S. Gupta, Y. Kim, M. Zhou, T. Rosing, DigitalPIM: Digital-based processing in-memory for big data acceleration, in: Proceedings of the 2019 on Great Lakes Symposium on VLSI, GLSVLSI '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 429–434, http://dx.doi.org/10.1145/3299874.3319483.
[21] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, S. Kvatinsky, SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (10) (2019) 2434–2447.
[22] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, Y. Xie, iPIM: Programmable in-memory image processing accelerator using near-bank architecture, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA, 2020, pp. 804–817, http://dx.doi.org/10.1109/ISCA45697.2020.00071.
[23] J.-H. Kim, J. Lee, J. Lee, H.-J. Yoo, J.-Y. Kim, Z-PIM: An energy-efficient sparsity aware processing-in-memory architecture with fully-variable weight precision, in: 2020 IEEE Symposium on VLSI Circuits, 2020, pp. 1–2, http://dx.doi.org/10.1109/VLSICircuits18222.2020.9163015.
[24] C. Chu, Y. Wang, Y. Zhao, X. Ma, S. Ye, Y. Hong, X. Liang, Y. Han, L. Jiang, PIM-prune: Fine-grain DCNN pruning for crossbar-based process-in-memory architecture, in: 2020 57th ACM/IEEE Design Automation Conference, DAC, 2020, pp. 1–6, http://dx.doi.org/10.1109/DAC18072.2020.9218523.
[25] M.A. Hanif, A. Manglik, M. Shafique, Resistive crossbar-aware neural network design and optimization, IEEE Access 8 (2020) 229066–229085, http://dx.doi.org/10.1109/ACCESS.2020.3045071.
[26] H. Abunahla, Y. Halawani, A. Alazzam, B. Mohammad, NeuroMem: Analog graphene-based resistive memory for artificial neural networks, Sci. Rep. 10 (1) (2020) 1–11.
[27] B. Li, Y. Wang, Y. Chen, HitM: High-throughput ReRAM-based PIM for multi-modal neural networks, in: Proceedings of the 39th International Conference on Computer-Aided Design, ICCAD '20, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3400302.3415663.
[28] X. Peng, R. Liu, S. Yu, Optimizing weight mapping and data flow for convolutional neural networks on RRAM based processing-in-memory architecture, in: 2019 IEEE International Symposium on Circuits and Systems, ISCAS, 2019, pp. 1–5, http://dx.doi.org/10.1109/ISCAS.2019.8702715.
[29] M. Imani, S. Gupta, Y. Kim, T. Rosing, Floatpim: In-memory acceleration of deep neural network training with high precision, in: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture, ISCA, IEEE, 2019, pp. 802–815.
[30] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E.G. Friedman, A. Kolodny, U.C. Weiser, MAGIC—Memristor-aided logic, IEEE Trans. Circuits Syst. II 61 (11) (2014) 895–899.
[31] M. Lee, M. Kaveh, Fast Hadamard transform based on a simple matrix factorization, IEEE Trans. Acoust. Speech Signal Process. 34 (6) (1986) 1666–1667, http://dx.doi.org/10.1109/TASSP.1986.1164972.
[32] S.Y. Kung, VLSI Array Processors, Englewood Cliffs, 1988.
[33] P.K. Meher, J.C. Patra, Fully-pipelined efficient architectures for FPGA realization of discrete Hadamard transform, in: 2008 International Conference on Application-Specific Systems, Architectures and Processors, IEEE, 2008, pp. 43–48.
[34] B.K. Mohanty, P.K. Meher, S.K. Singhal, Efficient architectures for VLSI implementation of 2-D discrete Hadamard transform, in: 2012 IEEE International Symposium on Circuits and Systems, ISCAS, IEEE, 2012, pp. 1480–1483.
[35] B.K. Mohanty, Parallel VLSI architecture for approximate computation of discrete Hadamard transform, IEEE Trans. Circuits Syst. Video Technol. 30 (12) (2020) 4944–4952.
[36] S. Kvatinsky, M. Ramadan, E.G. Friedman, A. Kolodny, VTEAM: A general model for voltage-controlled memristors, IEEE Trans. Circuits Syst. II 62 (8) (2015) 786–790.
[37] NVIDIA CUDA SDK for Fast Walsh Hadamard Transformed. URL https://www.nvidia.com/content/cudazone/cuda_sdk/Linear_Algebra.html#fastWalshTransform.
[38] A.G. Weber, The USC-SIPI Image Database Version 6, USC-SIPI Report 432, 2018.