Sorting in Memristive Memory

MOHSEN RIAHI ALAM, University of Louisiana at Lafayette, USA
M. HASSAN NAJAFI, University of Louisiana at Lafayette, USA
NIMA TAHERINEJAD, TU Wien, Austria

Sorting data is needed in many application domains. Traditionally, the data is read from memory and sent to a general-purpose processor or application-specific hardware for sorting. The sorted data is then written back to the memory. Reading/writing data from/to memory and transferring data between memory and processing unit incur significant latency and energy overhead. In this work, we develop the first architectures for in-memory sorting of data to the best of our knowledge. We propose two architectures. The first architecture is applicable to the conventional format of representing data, i.e., weighted binary radix. The second architecture is proposed for developing unary processing systems, where data is encoded as uniform unary bit-streams. As we present, each of the two architectures has different advantages and disadvantages, making one or the other more suitable for a specific application. However, the common property of both is a significant reduction in the processing time compared to prior sorting designs. Our evaluations show on average 37× and 138× energy reduction for binary and unary designs, respectively, compared to conventional CMOS off-memory sorting systems in a 45nm technology. We designed a 3×3 and a 5×5 Median filter using the proposed sorting solutions, which we used for processing 64×64 pixel images. Our results show a reduction of 14× and 634× in energy and latency, respectively, with the proposed binary, and 5.6× and $152{\times}10^3$ in energy and latency with the proposed unary approach compared to those of the off-memory binary and unary designs for the 3×3 Median filtering system.

Additional Key Words and Phrases: In-memory computation, sorting networks, unary processing, stochastic computing, memristor, median filtering, ReRAM.

1 INTRODUCTION

Sorting is a fundamental operation in computer science, used in databases [23, 24], scientific computing [18], scheduling [58], artificial intelligence and robotics [11], image [35], video [14], and signal processing [46]. The latency and energy consumptions of the sorting algorithm directly affect the efficiency of these systems. A sizeable body of research has focused on harnessing the computational power of many-core Central Processing Unit (CPU)- and Graphics Processing Unit (GPU)-based systems for efficient sorting [12, 13, 56]. For high-performance applications, sorting is implemented in hardware using either Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) [15, 31, 44]. The parallel nature of hardware-based solutions allows them to outperform software-based solutions executed on CPUs/GPUs.

The usual approach for hardware-based sorting is to wire up a network of Compare-and-Swap (CAS) units in a configuration called a Batcher (or bitonic) network [8]. Batcher networks provide low-latency solutions for hardware-based sorting [2, 20]. Each CAS block compares two input values and, if required, swaps the values

This work was supported in part by the Louisiana Board of Regents Support Fund no. LEQSF(2020-23)-RDA- 26, National Science Foundation grant #2019511, and generous gift from Cisco.
Authors' addresses: Mohsen Riahi Alam, University of Louisiana at Lafayette, Louisiana, USA; M. Hassan Najafi, University of Louisiana at Lafayette, Louisiana, USA; Nima TaheriNejad, TU Wien, Vienna, Austria.

1550-4832/2022/2-ART $15.00
https://doi.org/10.1145/3517181

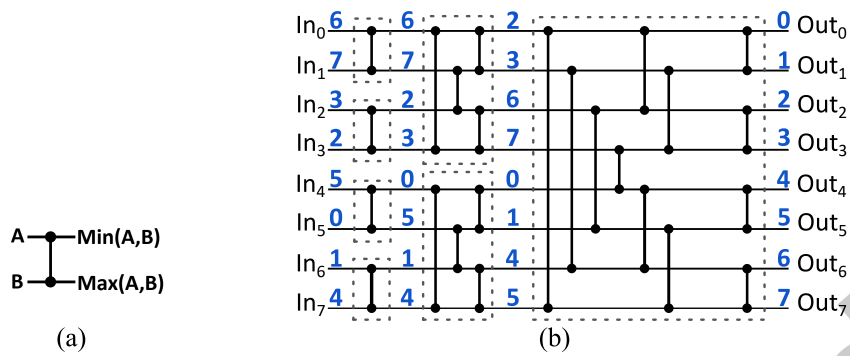

Fig. 1. (a) Schematic symbols of a CAS block (b) CAS network for an 8-input bitonic sorting.

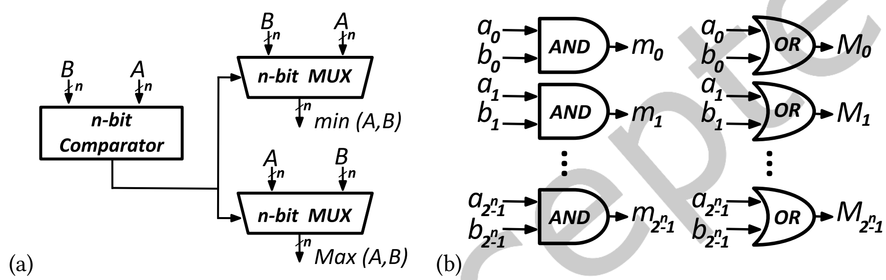

Fig. 2. Logic design of a CAS block: (a) conventional binary design processing data (b) parallel unary design processing unary bit-streams.

at the output. Fig. 1(a) shows the schematic symbol of a CAS block. Fig. 1(b) shows the CAS network for an 8-input bitonic sorting network, made up of 24 CAS blocks. Batcher sorting networks are fundamentally different from software algorithms for sorting (such as the quicksort, merge sort, and the bubble sort), since the order of comparisons is fixed in advance. That is, in contrast to software algorithms, the order is data-dependent [40]. The implementation cost of a batcher network is a direct function of the number of CAS blocks and the cost of each block. A CAS block is conventionally designed based on the weighted binary radix representation of data. The CAS design consists of an n-bit comparator and two n-bit multiplexers, where n is the data-width of the input data[1]. Fig. 2(a) shows the conventional design of a CAS unit. In the conventional binary design, increasing the data-width increases the complexity of the design.

All these prior sorting designs were developed based on the Von-Neumann architecture, separating the memory unit where the data is stored and the processing unit. where the data is processed (i.e., sorted). A significant portion of the total processing time and the total energy consumption is wasted on 1) reading the data from memory, 2) transferring the data between memory and processing unit, and 3) writing the result back into the memory [7] [30] [53] [32] [63]. In-Memory Computation (IMC) or Processing in Memory (PIM) is an emerging computational approach that offers the ability to both store and process data within memory cells [5, 26, 27, 52, 60, 62, 66, 69]. This technique eliminates the high overhead of transferring data between memory and processing unit, improving the performance and reducing the energy consumption by processing data in memory. Memristive storage is a Non-Volatile Memory (NVM) with high storage density and IMC capability. This

[1]We note that later on, we take the precision of binary to unary conversion (for unary sorting solution) to be also equal to n, the data-width.

emerging technology is one of the most promising candidates for the next generation of storage systems. The IMC capability of NVM devices allows accelerating sorting by avoiding the overheads of transferring the data between memory and processing unit. New sorting approaches based on NVM technology are on the table to increase the efficiency of the hardware-based sorters. Some previous studies worked on optimizing sorting algorithms for NVM and presented NVM-friendly sorting algorithms [9] [17]. Prasad et al. [49] proposed RIME which provides an API library for sorting algorithms using a bit-level search operation within the memory. They use some additional CMOS circuitry including a sensing circuit to compute the minimum and maximum values by performing XOR operation on the memory peripheral circuits. Some previous studies focused on near storage/memory computing techniques for providing efficient sorting algorithms. Li et al. [36] proposed IMC-Sort, an in-memory parallel sorting architecture using the hybrid memory cube. Their architecture incorporates a custom parallel sorting unit to accelerate the sort workloads in DRAM based on 3D stacking technology. Pugsey et. al. [50] suggested 3D-stacked near-data processing for sorting data in DRAM. Processing units and memory are integrated with 3D stacking technology using through-silicon vias. Salamat et al. [54] proposed a near-storage accelerator for databases sort based on the bitonic sort. Their accelerator utilizes an NVMe flash drive with an onboard FPGA chip. The authors in [51] propose FANS; an FPGA accelerated near-storage sorting system. Their system is able to sort hundreds of gigabytes of data on a single Samsung SmartSSD. The authors in [55] introduced Bonsai, an adaptive FPGA-based near-memory sorting solution. Their design considers the off-chip memory bandwidth and on-chip resources to optimize sorting time. Casper and Olukotun [50] presented three hardware accelerator designs to perform near-memory database operations including sort. Evaluation results by implementing their designs on FPGA showed close to ideal utilization of available memory bandwidth. These prior works sort the data near-memory or in-memory within peripheral circuitry. None of them perform sorting in memory within the memory array. For a detailed classification of different near- and in-memory computing methods, the readers are referred to [42].

In this paper, we take advantage of IMC to implement sorting units on memristive memory arrays. To the best of our knowledge, this work introduces the first *in-array* architectures for high-performance and energy-efficient sorting of data completely in memory (CIM-A using the [42] terminology). Our work is different from the aforementioned prior works in the sense that all computation results are produced within the memory array. We then go further to show how we can benefit from the concept of Unary Computing [40, 48] to improve the sorting hardware further for particular applications. We propose two different architectures. The first architecture, "Binary Sorting," is based on the conventional weighted binary representation and is applicable to conventional systems that store the data in memory in the binary format. The second architecture, "Unary Sorting," is based on the non-weighted unary representation. For each of these designs, we first discuss the basic operation of sorting two n-bit data (i.e., a CAS block). We then elaborate on the design of complete sorting networks, which are made up of the proposed in-memory CAS units. We showcase the role and importance of the achieved gains in the context of a median filter used in image processing applications. Our experiments demonstrate a reduction of 14× and 634× in energy and latency for the proposed binary, and 5.6× and 152×10^3 in energy and latency for the proposed unary approach compared to those of the off-memory binary and unary designs when implementing a 3×3 Median filtering system.

The rest of this paper is structured as follows. Section 2 provides a brief overview of memristive IMC and the unary processing technique used in this work. Section 3 and Section 4 present the proposed in-memory Binary and Unary Sorting designs. Section 5 compares the performance of the proposed designs with the conventional off-memory CMOS-based designs and applies the proposed architectures to an important application of sorting, i.e., median filtering. Finally, conclusions are drawn in Section 7.

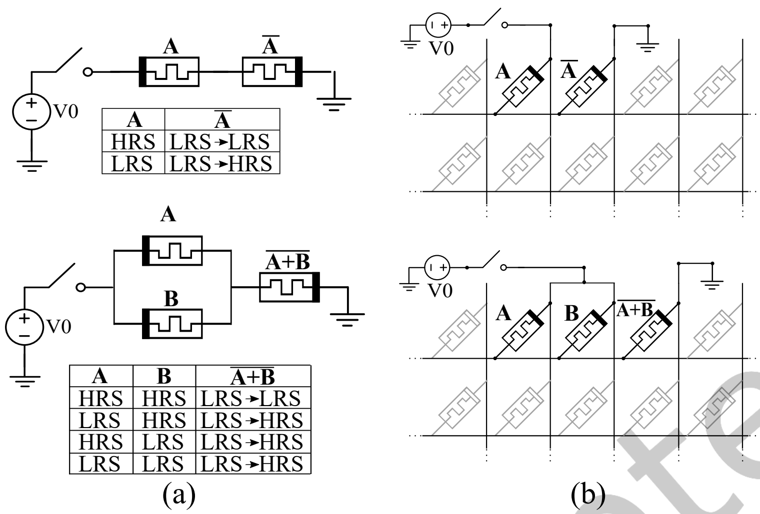

Fig. 3. (a) NOT and NOR logical operations in MAGIC and their truth tables. Low Resistance State (LRS) and High Resistance State (HRS) represent logical âĂŸ1âĂŹ and logical âĂŸ0âĂŹ, respectively. (b) Crossbar implementation of NOT and NOR logical operations.

2 BACKGROUND

2.1 Memristive IMC

One of the promising technologies for IMC is memristive technology. Among various memristive-based IMC methods, stateful logic such as Material Implication (IMPLY) [10], Memristor-Aided Logic (MAGIC) [33], FELIX [25], and Single-cycle In-Memristor XOR (SIXOR) [59] are of the most efficient solutions. In stateful logic, the input and output are both presented as the state of input and output memristors. Hence, no access to the world outside the array (e.g., read or write) is necessary for stateful logic operations. In this work, we use MAGIC NOR operation, which can be used to implement any Boolean logic. MAGIC considers two states of memristors: LRS as logical âĂŸ1âĂŹ and HRS as logical âĂŸ0âĂŹ. Fig. 3(a) shows how NOR and NOT logical operations can be implemented in MAGIC [33], where the memristors connected to the ground are output memristors [33]. Before starting the execution of an operation, the output memristors are first initialized to LRS. By applying a specific voltage (V_0) to the negative terminal of the input memristors, the output memristors may experience a state change from LRS to HRS, depending on the states of the inputs [33]. The truth tables embedded in Fig. 3(a) show all possible cases of the input memristors' states and switching of the output memristors. Fig. 3(b) shows how MAGIC NOT and NOR can be realized in a crossbar memory. These operations can be natively executed within memory with a high degree of parallelism. Thus, parallel architectures such as sorting networks can benefit greatly from such IMC logic operations.

2.2 In memory Comparator

Comparison is an essential operation in implementing sorting functions. Comparing memory content has been always challenging in computing systems. Content-addressable memory (CAM) [45] uses a dedicated equality comparator circuit to return the location of the matching data. CAMs help searching architectures and can be applied to packet forwarding in network routers. Authors in [49] propose a method for finding the minimum and maximum values within a set of numbers in memory. They employ bitwise column search to design a bit-serial algorithm for finding the minimum and maximum value. The sorting mechanisms proposed in [36] and [50] are based on a bitonic sorting network and include some comparison units. The comparison units in these works are implemented at the logic layer, which is integrated with 3D stacking DRAM memory banks using through-silicon vias technology. The authors in [1] and [29] further propose two in-memory equality comparators for SRAM memories.

Authors in [16] developed a multivalued 1T1R memristor method for in-memory computing. They exploit the multivalued resistance for performing a 1-bit in-memory comparison and then expand the design to a 4-bit magnitude comparator. Angizi et al propose an in-memory magnitude comparator in [6]. Their design uses in-memory XOR operations to perform a bit-wise comparison between corresponding bits of two data beginning from the most significant bit towards the least significant bit. However, the comparison process involves reading the output of the XOR operations and the data from memory by the control unit. Therefore, its latency (i.e., number of processing cycles) is non-deterministic and depends on the data being compared. In Section 3.1, we propose an in-memory magnitude comparator with deterministic latency and no memory read operations (a stateful comparator). Our in-memory comparator does not also need multivalued memristors.

2.3 Unary Sorting

Unary (or burst) processing [47, 48] is an alternative computing paradigm to conventional binary offering simple and noise-tolerant solutions for complex arithmetic functions [19, 28, 37–40, 57, 64, 65]. The paradigm borrows the concept of averaging from stochastic computing [4, 21], but is deterministic and accurate. In unary processing, unlike weighted binary radix, all digits are weighted equally. Numbers are encoded uniformly by a sequence of one value (e.g., 1) followed by a sequence of the other value (e.g., 0) in a stream of 1's and 0's– called a *unary bit-stream*. The value of a unary bit-stream is determined by the frequency of the appearance of 1âĂŹs in the bit-stream. For example, 11000000 is a unary bit-stream representing 2/8 or 1/4.

Unary computing was first used in [40, 41] for the simple and low-cost implementation of sorting network circuits. Zhang et al. [68] developed an SC-based neural network accelerator by employing a bit-stream-based bitonic sorting network for simultaneously implementing the accumulation and activation functions. With unary bit-streams and also when using correlated stochastic bit-streams [3], minimum and maximum functions (the main operations in a CAS block) can be implemented using simple standard AND and OR gates. In a serial manner, one AND and one OR gate implements a CAS block by processing one bit of the two bit-streams at each cycle. Hence, a total of 2^n processing cycles is needed to process two 2^n-bit bit-streams (equivalent to two n-bit binary data since we chose the precision of binary to unary conversion to be equal to the data-width, that is, equal to n). More than 90% saving in the hardware cost is reported for a 256-input serial unary sorting circuit at the cost of processing time [40]. Alternatively, the bit-streams can be processed in one cycle by replicating the logic gates and performing the logical operations in parallel. Fig 2(b) shows the parallel unary design of a CAS block. 2^n pairs of AND and OR gates sort two 2^n-bit bit-streams.

3 PROPOSED IN-MEMORY BINARY SORTING

In this section, we present our proposed method for in-memory sorting of binary radix data. First, we discuss the implementation of a basic sorting unit and then generalize the architecture to complete sort systems.

3.1 Basic Binary Sorting Unit

A basic binary sorting unit (CAS unit) requires one comparator and two multiplexers. Implementing an n-bit comparator by using basic logic gates requires $(11n - 2)$ NOR and $(7n - 2)$ NOT logic gates. Figs. 4(a) and 5(a) show the generic logic and the NOR-based logic design of a 4-bit binary comparator. Fig. 5(b) shows our proposed

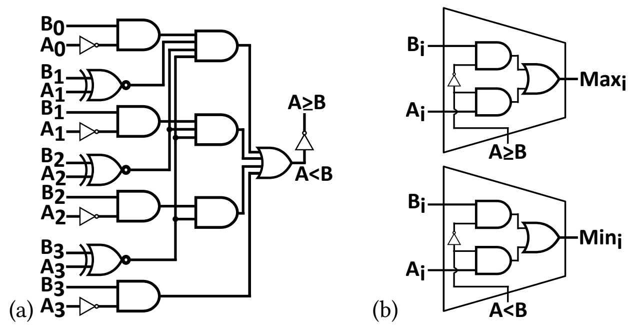

Fig. 4. Generic logic design of (a) a 4-bit binary magnitude comparator and (b) a multi-bit binary 2-to-1 multiplexer for Max/Min selection.

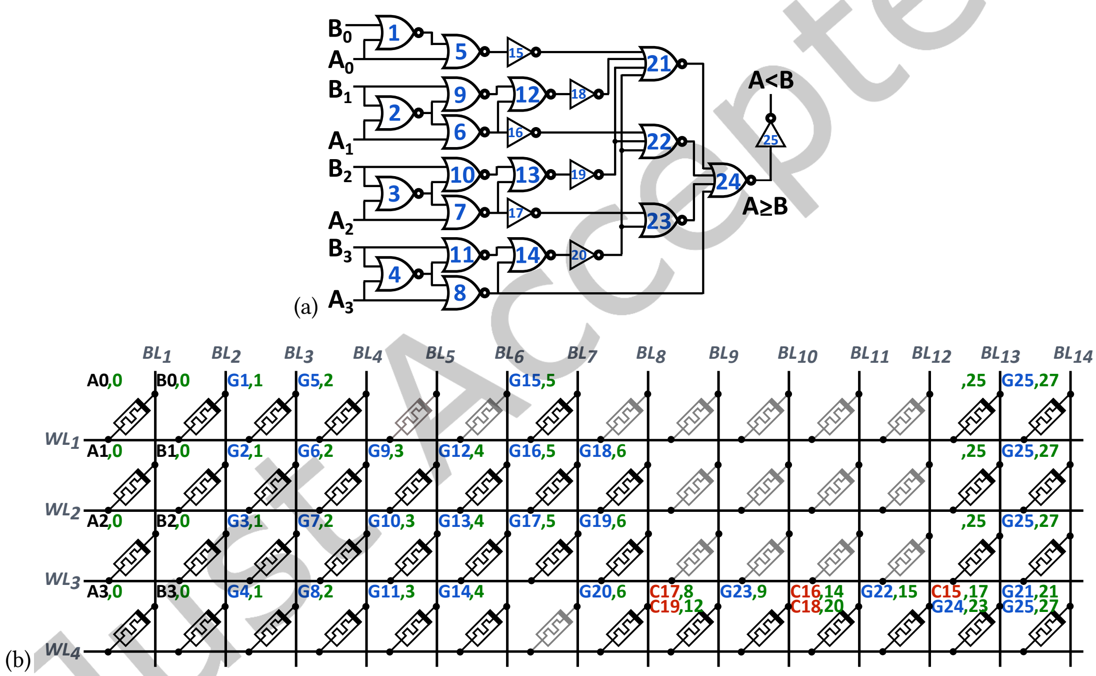

Fig. 5. (a) NOR-based logic design of a 4-bit binary comparator. (b) MAGIC-based 4-bit binary in-memory comparator. G_i memristor holds the output of the i-th gate. C_i memristor copies the state of G_i memristor. The second number shown on each memristor (e.g., 2 in G5,2) determines the processing cycle in which the memristor operates. (WL = Word Line, BL = Bit Line)

in-memory implementation using MAGIC. As shown, implementing this comparator using MAGIC NOR and NOT operations requires a crossbar with 4×14 memory cells. The input data (i.e., A and B) in binary format is stored in two different columns (BL_1 and BL_2), each column containing n memristors, where n is the size of the data being compared (in this example, $n = 4$). The computation includes NOR, NOT, and copy operations. Each $G_{i,j}$

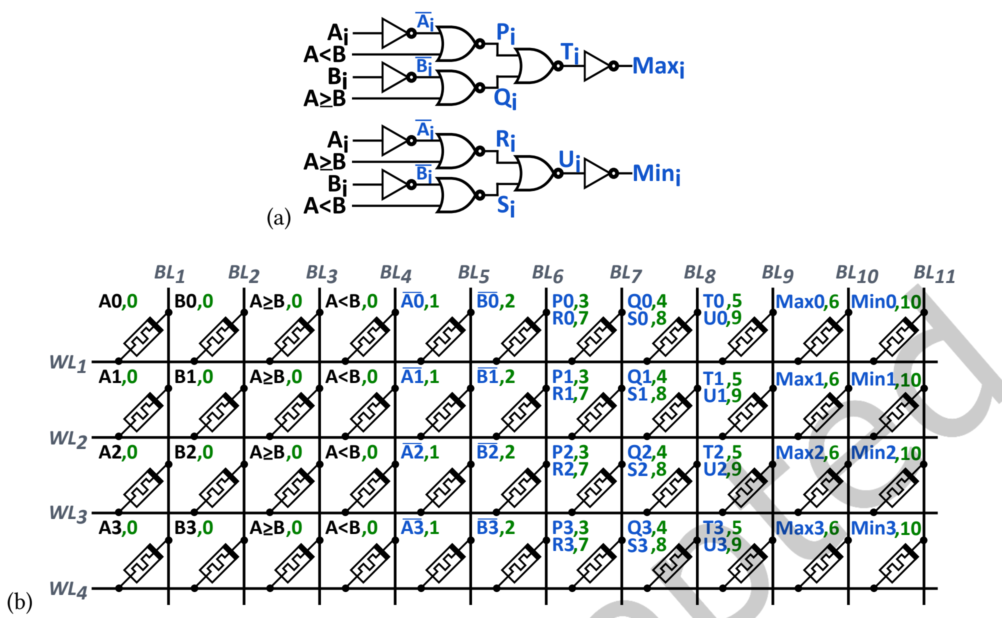

Fig. 6. (a) NOR-based logic design of a multi-bit binary 2-to-1 multiplexer and (b) in-memory MAGIC-based 4-bit binary multiplexer for Max/Min selection. The second number shown on each memristor (e.g., 3 in P0,3) determines the processing cycle in which the memristor operates. (WL = Word Line, BL = Bit Line)

memristor in Fig. 5(b) shows participation in a logical gate (operation) i in Fig. 5(a). $C_{i,j}$s on the other hand, show the copy operation i, in which the state of G_i memristor is duplicated. The index j marks the cycle number in which an operation is performed. In some cases, a memristor participates in two operations. For example, the memristor at the right-bottom end of Fig. 5(b) (WL_4, BL_{14}) is once used at cycle 21 in gate (operation) 21 and once at cycle 27 in gate (operation) 25.

To execute these operations, in each clock cycle, the memristor controller applies the proper voltage to crossbar columns and rows to execute some NOR or NOT operations concurrently. All memristors with the same cycle number produce their output value at the same time. When possible, we reuse memristors to avoid increasing area, i.e., the number of used memristors. The memristors that are being re-used as an output must be initialized to LRS in an additional clock cycle before reusing. The comparison result (i.e., the output of gate 24) is ready at cycle 23 on (WL_4, BL_{13}). At this time, some copies from the comparison result and its complement must be made. These will be used as the select inputs of the maximum and minimum multiplexers shown in Fig. 4(b) and Fig. 6. To this end, first, we make three copies of the output of gate 24 on three memristors in the same column (BL_{13}) and then invert these memristors on another column (BL_{14}) to make the required complements. This leads to a total processing time of 27 cycles plus one initialization cycle.

After the comparison process, we need values of only four columns (BL_1 and BL_2 for the two input data, and BL_{13} and BL_{14} for the two comparison results) to implement the multiplexer part of the sorting unit. Hence, we could reuse the rest of the memristors. Fig. 4(b) and Fig. 6 show the generic logic and the NOR-based logic circuit for a multi-bit 2-to-1 multiplexer. Fig. 6(b) shows our MAGIC-based in-memory design for the two 4-bit multiplexers the sorting circuit requires to select the maximum and minimum data. In implementing the multiplexers, we re-use the memory cells of the comparison step. To this end, we initialize the columns used in the comparison

Table 1. The Required Resources and Number of Processing Cycles for the Proposed Basic Binary Sorting Unit

Data-Width	Required Dimension	# of Initialized Memristors	# of Reused Memristors	# of Logical Operation Cycles	# of Copies [Copy Cycles]	Reused init Cycles	Total # of Cycles	Energy (pJ)
4	4×14	40	44	21	6 (12)	6	39	199.4
8	8×22	88	88	25	14 (28)	10	63	417
16	16×38	184	176	33	30 (60)	18	111	845
32	32×70	376	352	49	62 (124)	34	207	1728
n	$n \times (8+2n\text{-}2)$	$12n - 8$	$11n$	$18+(n\text{-}1)$	$2n\text{-}2$ [$2(2n\text{-}2)$]	$n + 2$	$6n + 15$	-

step (BL_3 to BL_{12}) to LRS in one clock cycle. The input data is inverted in two clock cycles, cycles 1 and 2 (on BL_4 and BL_5) shown in Fig. 6(b). The first multiplexer produces the maximum value on BL_{10} in cycles 3 to 6. The minimum value is produced on BL_{11} by the second multiplexer through cycles 7 to 10. Since three columns used by the first multiplexer (i.e., P, Q, T) are being re-used by the second multiplexer, an additional cycle is considered for the initialization of these columns before execution of the second multiplex operation. The execution of the multiplexers, therefore, takes two initialization and 10 operation cycles. Hence, execution of the proposed in-memory basic binary sorting takes a total of 39 processing cycles plus one initialization cycle.

We extend the proposed design from sorting of 4-bit data to higher data-widths, namely 8-, 16-, 32-, and in general n-bit data. We verified the correct functionality of the proposed design by high-level simulation and measured the energy and delay numbers by circuit-level simulation using Cadence Virtuoso. Table 1 reports the required resources, the number of cycles, and energy consumption. Further details on circuit-level simulations and the parameter values used in estimating the energy numbers will be discussed in Section 5.1. We see that the area, the latency, and the energy consumption of the proposed basic binary sorting design increase linearly by increasing the data-width.

3.2 Complete Binary Sort System

A complete sort network is made of basic sorting units (i.e., CAS blocks). In bitonic sorting, the network recursively merges two sets of size $N/2$ to make a sorted set of size N [22]. Fig. 1 shows the CAS network for an 8-input bitonic sorting. As it can be seen, the network is made of 24 CAS units. In general, an N-input bitonic sorting network requires

$$U_{CAS} = N \times log_2(N) \times (log_2(N) + 1)/4 \tag{1}$$

CAS units. These CAS units can be split into

$$S = log_2(N) \times (log_2(N) + 1)/2 \tag{2}$$

steps (also known as *stages*), each with $N/2$ CAS units that can operate in parallel [20].

Gupta et al [25] propose a memory partitioning method to improve the in-memory parallelism. In a similar fashion, we split the memory into multiple partitions to enable parallel execution of different CAS operations in each bitonic CAS stage. Fig. 7 shows how we implement an 8-input bitonic sorting network in memory. The memory is split into four partitions, namely partitions A, B, C, and D (each marked on a black vertical line in the bitonic network representation). The number of partitions is decided based on the number of CAS units that can run in parallel (i.e., $N/2$.). Each partition includes two out of the eight unsorted input data. The sorting process is split into six steps equal to the number of CAS groups (stages). In the first step, the two inputs in each partition are sorted using the basic sorting operation proposed in Section II-A. In the second step, each maximum number (i.e., the larger number between the two in the partition) found by the sorting operations of the first step is copied to another partition where it is needed. The bitonic network determines the destination partition. For instance, the maximum found by executing the sorting operation in partition A (i.e., the input with a value of 7 in the

Table 2. Number of Processing Cycles, Size of Crossbar Memory, and Energy Consumption (nJ) to Implement Different Bitonic Sorting Networks (DW = Data-Width, BL = Bit-Stream Length)

Network	**Binary Sorting** DW = 4			DW = 8			DW = 16			DW = 32		
Size	Cycles	Size	Energy	Cycles	Size	Energy	Cycles	Size	Energy	Cycles	Size	Energy
4	128	4×28	1.2	200	8×44	2.5	344	16×76	5.1	632	32×140	10
8	280	4×56	4.7	424	8×88	10	712	16×152	20	1288	32×280	41
16	544	4×112	15	784	8×176	33	1264	16×304	68	2224	32×560	138
32	1048	4×224	47	1408	8×352	100	2128	16×608	205	3568	32×1120	415
Network	**Unary Sorting** BL = 16			BL = 64 (DW = 6)			BL = 256 (DW = 8)			BL = 1024 (DW = 10)		
Size	Cycles	Size	Energy	Cycles	Size	Energy	Cycles	Size	Energy	Cycles	Size	Energy
4	26	16×10	1.37	26	64×10	5.4	26	256×10	21.88	26	1024×10	87
8	76	16×20	5.4	76	64×20	21	76	256×20	87	76	1024×20	350
16	194	16×40	18	194	64×40	72	194	256×40	291	194	1024×40	1168
32	538	16×80	54	538	64×80	218	538	256×80	875	538	1024×80	3503
64	1406	16×160	153	1406	64×160	613	1406	256×160	2452	1406	1024×160	9809
128	3624	16×320	408	3624	64×320	1635	3624	256×320	6540	3624	1024×320	26159
256	9176	16×640	1051	9176	64×640	4204	9176	256×640	16817	9176	1024×640	67268

example of Fig. 7) will be copied into partition B to be compared with the minimum number between the two initial data in partition B of the first step. Similarly, in each one of the next steps (i.e., steps 3 to 6), one output data from each partition is copied to another partition, and a sorting operation is executed.

In each step, the sortings in different partitions are executed in parallel. After six steps and the execution of a total of 24 (=4 × 6) basic sorting operations, the sorted data is ready in the memory. Each basic sorting operation is implemented based on the in-memory basic binary sorting proposed in Section 3.1. Table 2 shows the total number of processing cycles, the required size of crossbar memory, and the energy consumption of different sizes of in-memory bitonic networks. The total number of processing cycles, PC_t, is calculated using

$$PC_t = S \times (1 + PC_b) + CP, \tag{3}$$

where PC_b is the number of processing cycles necessary to execute a basic sorting operation, CP is the number of copy operations, and S the number of sorting steps. The required size of crossbar memory (M_t) is found by

$$M_t = n \times \frac{N}{2} \times M_b, \tag{4}$$

where M_b is the size of the crossbar memory required for one basic sorting unit[2].

4 PROPOSED IN-MEMORY UNARY SORTING

In this section, we propose a novel method for sorting unary data in memory to avoid the overheads of off-memory processing in the unary systems. We first discuss the basic operation of sorting two unary bit-streams in memory and then elaborate on the design of a complete unary sorting network.

4.1 Basic Unary Sorting Unit

The maximum and minimum functions are the essential operations in a basic sorting unit. Performing bit-wise logical `AND` on two unary bit-streams with the same length gives the minimum of the two bit-stream. Bit-wise

[2]We remember that n is the data-width and N is the network size or the total number of items to be sorted.

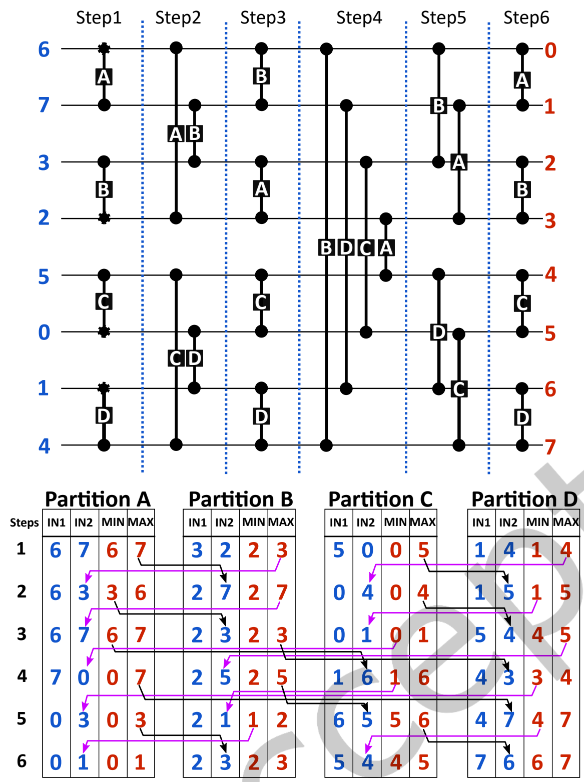

Fig. 7. High-level flow of 8-input Bitonic Sorting in Memory.

logical OR, on the other hand, gives the maximum of the two unary bit-streams with the same length. Fig. 8 shows an example of the maximum and the minimum operation on two unary bit-streams. The example presents these operations in a serial manner by processing one bit of the input bit-streams at each cycle. While the serial approach is extremely simple to implement with only one pair of AND and OR gates, it incurs a long latency proportional to the length of the bit-streams. In this work, we choose the precision of binary to unary conversion equal to the data-width, n. This means an n-bit data in the binary domain corresponds to a 2^n-bit bit-stream in the unary domain. This implies a latency of 2^n cycles with a serial unit. Parallel sorting of two n-bit precision data represented using two 2^n-bit bit-streams requires performing 2^n logical AND operations (to produce the minimum bit-stream), and 2^n logical OR operations (to produce the maximum bit-stream) in parallel as shown in Fig 2(b). The suitability of the memristive crossbar for running parallel logical operations in memory makes it a perfect place for low-latency parallel sorting of unary bit-streams.

Fig. 9 shows our proposed design for MAGIC-based in-memory execution of minimum and maximum operations on two unary bit-streams. As shown in Fig. 9, implementing this sorting unit using MAGIC NOR and NOT operations requires a memristor crossbar proportional to the length of the bit-streams. The unsorted unary data (i.e., A and B bit-streams) are stored in two different columns (BL_1 and BL_2). Both inputs have the same length of 2^n. As shown in Fig. 9(a), the AND operation (minimum function) is realized by first inverting the bit-streams through MAGIC NOT and then performing bit-wise MAGIC NOR on the inverted bit-streams. This effectively implements

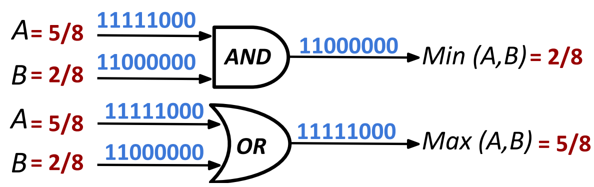

Fig. 8. Example of performing maximum and minimum operations on unary bit-streams.

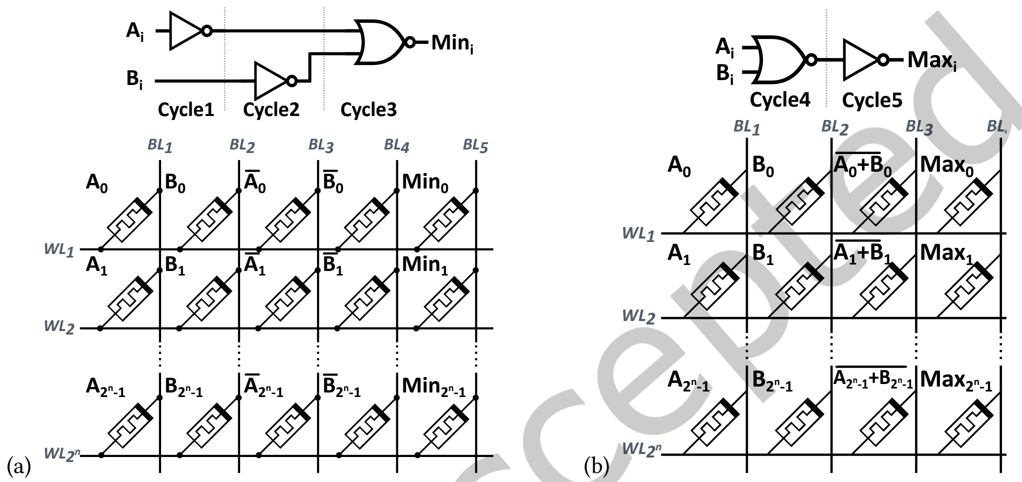

Fig. 9. Proposed in-memory unary sorting (a) in-memory minimum operation and (b) in-memory maximum operation on two unary bit-streams.(WL = Word Line, BL = Bit Line)

Table 3. The Required Resources, Number of Processing Cycles, and Energy Consumption of the Proposed Basic Unary Sorting

BitStream Length	Required Dim.	# of Initialized Memristors	# of Reused Memristors	# of NOR Operations	# of NOT Operations	Initial Cycles	Operation Cycles	Energy (pJ)
2^4	16×5	64	32	32	48	1	5	227
2^6	64×5	256	128	128	192	1	5	910
2^8	256×5	1024	512	512	768	1	5	3640
2^{10}	1024×5	4096	2048	2048	3072	1	5	14558

the AND operation as $A \wedge B = \overline{\overline{A} \vee \overline{B}}$. The first and the second bit-stream are inverted on BL_3 and BL_4 in the first and the second cycle, respectively. The NOR operation is executed in the third cycle on BL_5. As shown in Fig. 9(b), the OR operation (maximum function) is achieved by first performing MAGIC NOR on the input bit-streams and then MAGIC NOT on the outputs of the NOR operations. Hence, the execution of the OR operation takes two cycles.

The columns that we use during the execution of the AND operation to store the inverted version of the bit-streams (e.g., the third and fourth columns in Fig. 9(a)) are re-used in the execution of the OR operation to avoid using additional memristors. In contrast to the proposed in-memory binary sorting of Section 3.1, which has a variable latency dependent on the width of the input data, *the processing latency of the proposed unary* *sorting is fixed* at five cycles and does not change with the data-width. Table 3 shows the required resources, number of cycles, and energy consumption of the proposed basic sorting unit for different bit-stream lengths.

The number of memristors is directly proportional to the length of the bit-streams. In a fully parallel design approach, the size of the memory, particularly the number of rows, defines an upper-limit on the maximum data-width for the to-be-sorted unary data. In such a system, bit-streams with a length longer than the number of rows can be supported by splitting each bit-stream into multiple shorter sub-bit-streams, storing each sub-bit-stream in a different column, and executing the CAS operations in parallel. The sub-results will be finally merged to produce the complete minimum and maximum bit-streams. This design approach sorts the data with reduced latency as the primary objective. A different approach for sorting long bit-streams is to perform CAS operations on the sub-bit-streams in a serial manner by re-using the CAS unit(s). The above approach reduces the area (number of used memristors) at the cost of additional latency. In this case, after sorting each pair of sub-bit-streams, the result is saved, and a new pair of sub-bit-stream is loaded for sorting. Assuming that each input bit-stream is split into N sub-bit-streams, the number of processing cycles to sort each pair of input data increases by a factor of N. Some additional processing cycles are also needed for saving each sub-output and copying each pair of sub-input. Combining the parallel and the serial approach is also possible for further trade-offs between area and delay. These approaches increase the range of supported data-widths but incur a more complicated implementation and partition management.

4.2 Complete Unary Sort System

Implementing a bitonic sorting network in the unary domain follows the same approach as presented in Section 3.2 for binary implementation of sorting networks. The number of sorting steps and the required number of basic sorting operations are exactly the same as those of the binary sorting network design. The essential difference, however, is that in the unary sorting system, the data is in the unary format. Therefore, the basic 2-input sorting operation should be implemented based on the unary sorting unit proposed in Section 4.1. Table 2 shows the number of processing cycles and the required size of memory for implementing unary bitonic networks of different sizes. We report the latency, area, and energy of these networks as well.

5 COMPARISON AND APPLICATION

5.1 Circuit-Level Simulations

We implemented a 16×16 crossbar and necessary control signals in Cadence Virtuoso for circuit-level evaluation of the proposed designs. For memristor simulations, we used the Voltage Controlled ThrEshold Adaptive Memristor (VTEAM) model [34]. The Parameters used for the VTEAM model can be seen in Table 4. We evaluated the designs in an analog mixed-signal environment by using the Spectre simulation platform with $0.1ns$ transient step. For MAGIC operations, we applied V_{SET}=$2.08V$ with $1ns$ pulse-width to initialize the output memristors to LRS. For the simplicity of controller design, we consider the clock cycle period of $1.25ns$ and V_0 pulse-width of $1ns$ for all operations. V_0 voltage for NOT, 2-input NOR, 3-input NOR, and 4-input NOR is $1.1V$, $950mV$, $1.05V$, and $1.15V$, respectively. We perform the copy operations by using two consecutive NOT operations.

To estimate the total energy of in-memory computations, we first find the energy consumption of each operation. The energy number measured for each operation depends on the states of input memristors (i.e., LRS, HRS). We consider all possible cases when measuring the energy of each operation. For example, the 3-input NOR has eight possible combinations of input states. We consider the average energy of these eight cases as the energy of 3-input NOR. The average measured energy of different operations is reported in Table 5. Note that higher energy consumption for NOT operation compared to 2-input NOR is due to using a higher V_0 voltage for NOT. The reported energy for the proposed in-memory sorting designs is the sum of the energy consumed by all operations.

Table 4. Memristor Parameter Values from [33] for the VTEAM Model [34].

Parameter	Value	Parameter	Value
R_{on}	1 kΩ	x_{off}	3 nm
R_{off}	300 kΩ	k_{on}	-216.2 m/sec
VT_{on}	-1.5 V	k_{off}	0.091 m/sec
VT_{off}	300 mV	α_{on}	4
x_{on}	0 nm	α_{off}	4

Table 5. The Average Measured Energy Consumption of Each Operation Based on VTEAM Model.

Operation	Average Energy
memristor initialization	2350 fJ
memristor copy	40.08 fJ
NOT	20.04 fJ
2-input NOR	9.01 fJ
3-input NOR	37.24 fJ
4-input NOR	54.51 fJ

5.2 Comparison of In- and Off-Memory

We compare the latency and energy consumption of the proposed in-memory binary and unary sorting designs with the conventional off-memory CMOS-based designs for the case of implementing bitonic networks with a data-width of eight. For a fair comparison, we assume that the to-be-sorted data are already stored in memristive memory when the sorting process begins and hence do not consider the delay for initial storage. We do not consider this latency because it is the same for both cases of the proposed in-memory and the off-memory counterparts. For the case of off-memory binary designs, we assume 8-bit precision data are read from and written to a memristive memory. For the case of off-memory unary design, we evaluate two approaches: 1) unary data (i.e., 256-bit bit-streams) are read from and written to memory, and 2) 8-bit binary data are read from and written to memory. For the second approach, the conversion overhead (i.e., binary to/from unary bit-stream) is also considered. This conversion is performed off-memory using combinational CMOS logic [40]. The conventional CMOS-based off-memory sorting systems read the raw data from memory, sort the data with CMOS logic, and write the sorted data into memory. These read and write operations take the largest portion of the latency and energy consumption. We use the per-bit read and write latency and per-bit energy consumption reported in [67] to calculate the total latency and energy of reading from and writing into the memristive memory. For the proposed in-memory designs, the entire processing step is performed in memory, and so there is no read and write operations from and to the memory. For the off-memory cases, we do not incorporate the transferring overhead between the memory and the processing unit as it depends on the interconnects used. We implemented the off-memory processing units using Verilog HDL and synthesized them using the Synopsys Design Compiler v2018.06-SP2 with the 45nm NCSU-FreePDK gate library.

Table 6 shows the summary of performance results. As reported, the proposed in-memory designs provide a significant latency and energy reduction, compared to the conventional off-memory designs. That is, on average 14× and 37×, respectively, for the binary sorting. For the unary design, the average latency and energy reductions are 1200× and 138×, respectively. For the unary systems with the data stored in memory in a binary format, the proposed in-memory design can reduce the latency and energy by a factor of up to 65× and 9.7×, respectively. For a realistic and more accurate energy consumption comparison, however, the overhead of transferring data on the interconnect between the memory and the processing unit must be added for the off-memory cases. We note

Table 6. Energy Consumption (nJ) and Latency (μs) of the Implemented In-Memory and Off-Memory Bitonic Sorting Designs with Data-Width=8 (E: Energy, L: Latency)

Network Size	8		16		32		64		128		256	
Design Method	E	L	E	L	E	L	E	L	E	L	E	L
Off-Memory Binary Sorting (+ Binary R/W)	850	6.5	1701	13	3403	26	6806	52	13613	104	27227	209
Proposed In-Memory Binary Sorting	10	0.55	33	1.02	100	1.8	281	3.4	794	6.8	1927	14
Off-Memory Unary Sorting (+ Binary R/W)	851	6.5	1703	13	3406	26	6811	52	13622	104	27244	209
Off-Memory Unary Sorting (+ Unary R/W)	27226	210	54452	419	108904	839	217809	1679	435618	3358	871236	6717
Proposed In-Memory Unary Sorting	87	0.10	291	0.25	875	0.7	2452	1.8	6,540	4.7	16,817	12

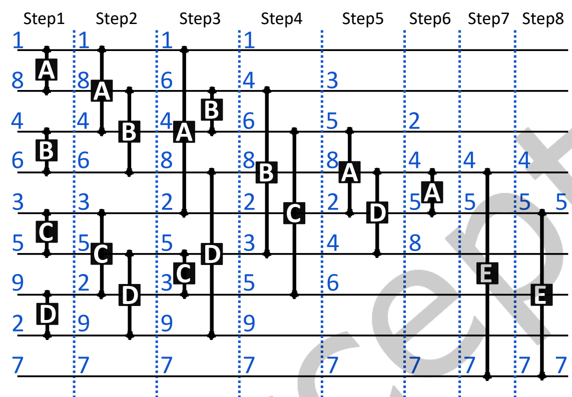

Fig. 10. Processing Steps and Memory Partitioning of the 3×3 Median Filter Design.

that these numbers are highly dependent on the architecture of the overall system and the interconnects used. Therefore, different system architectures may substantially change these numbers; however, they do not change the fact that our proposed method is more advantageous. In fact, they only change the extent of this improvement (and further increase it) since no data transfer happens in the in-memory sorting solution. Hence, by eliminating them, we present the minimum improvement obtained by our method and leave the further improvement to the final implementation details of designers.

5.3 Application to Median Filtering

Median filtering has been widely used in different applications, from image and video to speech and signal processing. In these applications, digital data is often affected by noise. A median filter —which replaces each input data with the median of all the data in a local neighborhood (e.g., a 3×3 local window)— is used to filter out impulse noises and smoothen the data [43]. A variety of methods for the implementation of Median filters have been proposed. Sorting network-based architectures made of CAS blocks are one of the most common approaches [40]. The incoming data is sorted as it passes the network. The middle element of the sorted data is the median. We developed in-memory architectures for a 3×3 and a 5×5 median filtering based on our proposed in-memory binary and unary sorting designs.

Fig. 10 and Fig. 11 depict a high-level flow of memory partitioning for our in-memory 3×3 and 5×5 Median filter design. Similar to our approach in implementing the complete sort system, the memory is split into multiple partitions. For the 3×3 design, partitions are A, B, C, D, and E (five partitions), and for the 5×5 design, they are

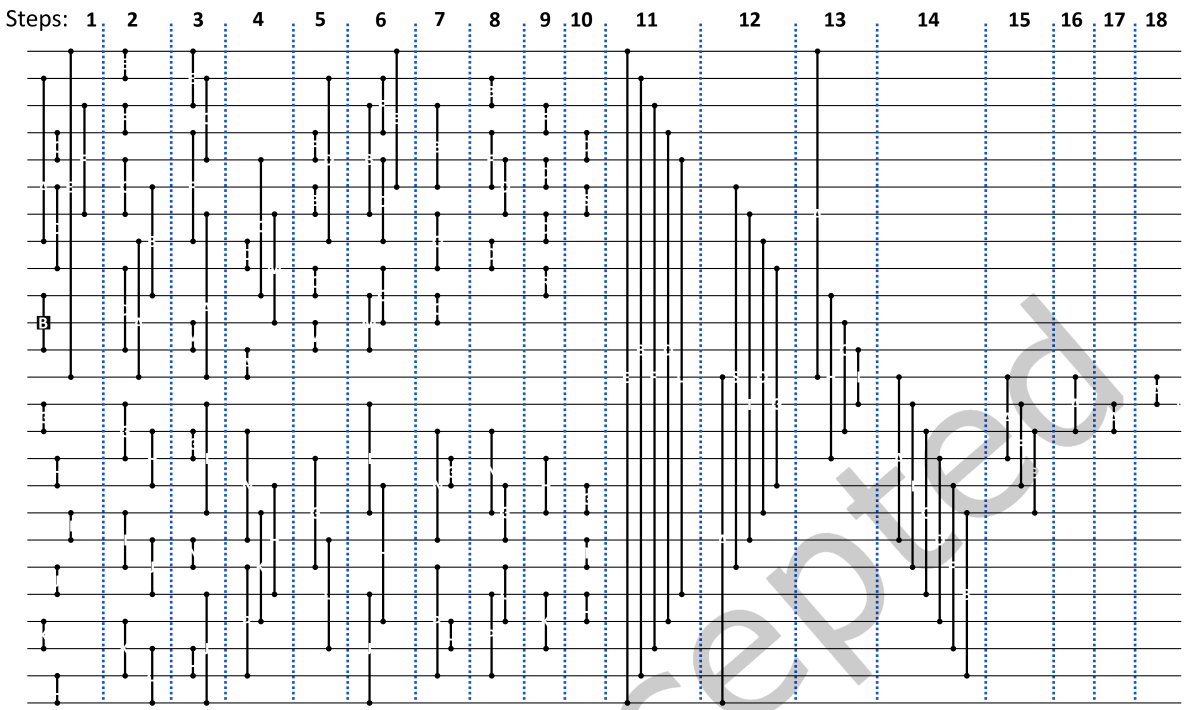

Fig. 11. Processing Steps and Memory Partitioning of the 5×5 Median Filter Design.

A to T (20 partitions). Each sorting unit sorts the data in one particular partition. Some partitions are initialized with the input data in the first step and the others are initialized and used in the following steps. The process is split into eight steps for the 3×3, and to 18 steps for the 5×5 design, each step executing some basic sorting operations in parallel. After each step, to prepare data for the next step, some data must be transferred between partitions similar to what we did in sorting. These data transfers are done by using copy operations. Compared to a complete sorting network, fewer sorting units are required as only the median value is targeted.

We evaluated the implementation of these basic sorting operations using both the proposed binary and unary bit-stream-based in-memory architectures. Table 7 reports the latency, the number of required memristors, and the energy consumption of the developed designs for (i) a single 3×3 Median filter and a single 5×5 Median filter and (ii) a 3×3 and a 5×5 Median filter image processing system that process images of 64×64 size. The corresponding latency and energy consumption of the off-memory CMOS-based binary and unary designs are also reported in Table 7. As it can be seen, the proposed in-memory binary and unary designs reduce the energy by a factor of 14× and 5.6×, respectively, for the 3×3-based image processing system, and 3.1× and 12× for the 5×5-based image processing system, compared to their corresponding off-memory designs. The latency of the binary and unary design is also reduced by a factor of 634× and 152×10^3 × with the 3×3 window, and by a factor of 110× and 19.2×10^3 × with the 5×5 window, for the 64×64 image processing system.

Note that we did not incorporate the overhead latency and the energy of transferring data on the bus or other interconnects for the off-memory cases, which is a large portion of energy of consumption in transferring data between memory and processing unit [30]. By considering this overhead, our approach would have a significantly larger advantage over others in a complete system.

Table 7. The Required Resources (M_t), Latency (L), and Energy Consumption (E) of the Implemented Median Filter Designs

	Median Filter 3 × 3						
Design	PC_t	M_t	E (μ J)	L (μ s)	Design	E (μ J)	L (μ s)
Proposed Binary	544	8 × 110	0.0085	0.68	Off-Memory Binary	0.121	0.94
Proposed Unary	72	256 × 25	0.069	0.09	Off-Memory Unary	3.882	30.17
	64 × 64 Image Processor						
Proposed Binary	4896	208 × 1980	35	6.1	Off-Memory Binary	490	3870
Proposed Unary	684	2048 × 1425	283	0.81	Off-Memory Unary	1590	123578
	Median Filter 5 × 5						
Design	PC_t	M_t	E (μ J)	L (μ s)	Design	E (μ J)	L (μ s)
Proposed Binary	1416	8 × 440	0.049	1.77	Off-Memory Binary	0.151	1.19
Proposed Unary	259	256 × 100	0.401	0.324	Off-Memory Unary	4841	38.02
	64 × 64 Image Processor						
Proposed Binary	35400	328 × 1760	200	44.25	Off-Memory Binary	620	4875
Proposed Unary	6475	2048 × 2000	1643	8.09	Off-Memory Unary	19829	155739

6 DISCUSSION

The high latency and energy overhead of reading from and writing to memory, and transferring data between the processing unit and memory, take up a significant amount of resources in sorting data in the conventional systems. IMC is a promising solution to mitigate these overheads. IMC is particularly beneficial for 1) applications with large data or a large number of memory accesses and 2) applications with extensive parallelism that can independently run a large number of operations in parallel. Sorting is one of the applications that has both properties. As we showed, implementing sorting in memory can save significant time and energy by avoiding the overheads of memory access and off-chip data transfer. This is particularly important for the unary systems, where data are stored in memory in the form of long bit-streams. Reading and writing long bit-streams from and to memory make off-memory unary sorting highly inefficient. However, one should note that the size of the memory array puts an upper limit on the size of the sorting network and the data-width. For example, given a (memristive) memory array of 1024×1024, the proposed binary sorting approach supports the complete sorting of 64 8-bit and 128 4-bit input data. For the proposed unary approach, an array of that size (1Mb or 128kB) supports the complete sorting of 256 1024-bit unary bit-streams. For the larger bit-streams or data-width, or larger number of data to be sorted, we would need to partition the data into different arrays. That means a more complex control and partition management mechanism, which reduces the benefits of fully in-array sorting.

Memristive technology is an emerging technology still in evolution, with many competing implementation methods in the process of maturation [61]. Properties such as the delay, power, and energy consumption are heavily dependent on the used technology and change considerably from one to another. In this article, the experimental results are provided by simulation tools using the VTEAM model. Using different memristive technologies and models, LRS and HRS values, as well as programming or reading pulses with different amplitude or width, affect the actual delay and energy numbers reported here. The actual memristive implementation is a showcase of the feasibility and meaningfulness of such an in-memory sorting design. That is, there exists an in-memory implementation (namely using the memristive technology we have used here) to be significantly beneficial. Therefore, we consider the properties and comparison of other ways of implementing our proposed architecture (using CMOS IMC, other memristor technologies, or other emerging memory technologies) as an exciting future work but outside the current article's scope. Nonetheless, we would like to point out that we have provided the number of memristors (memory cells) and the number of operation cycles that are technology-independent. Therefore, others can independently evaluate and compare their own implementations with ours in a technology-agnostic fashion and using the number of memory cells and the number of cycles their implementation needs (regardless of the LRS and HRS values or pulse amplitude and width).

7 CONCLUSION

Thus far, sorting solutions were based on the conventional approach of processing off-memory, incurring a high overhead of reading/writing from/to memory and transferring between the memory and the processing unit. In this paper, for the first time -to the best of our knowledge- we developed two methods for in-array sorting of data: a binary and a unary sorting design. We compared the area, latency, and energy consumption of the basic and the complete sorting systems for different data-widths and network sizes. The latency and energy are significantly reduced compared to prior off-memory CMOS-based sorting designs. Further, we developed in-memory binary and unary designs for an important sorting application, median filtering. In future works, we plan to extend the proposed architectures to other applications of sorting, for instance, efficient in-memory implementation of weighted and adaptive median filters. We will also explore applications of in-memory sorting in communications and coding domains.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 481–492. https://doi.org/10.1109/HPCA.2017.21

[2] Sherenaz W. Al-Haj Baddar and Basel A. Mahafzah. 2014. Bitonic Sort on a Chained-cubic Tree Interconnection Network. *JPDC* 74, 1 (2014), 1744–1761. https://doi.org/10.1016/j.jpdc.2013.09.008

[3] Armin Alaghi and John P. Hayes. 2013. Exploiting correlation in stochastic circuit design. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 39–46. https://doi.org/10.1109/ICCD.2013.6657023

[4] Armin Alaghi, Weikang Qian, and John P. Hayes. 2018. The Promise and Challenge of Stochastic Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (2018), 1515–1531. https://doi.org/10.1109/TCAD.2017.2778107

[5] Mohsen Riahi Alam, M. Hassan Najafi, and Nima Taheri Nejad. 2021. Exact Stochastic Computing Multiplication in Memristive Memory. *IEEE Design Test* (2021), 1–1. https://doi.org/10.1109/MDAT.2021.3051296

[6] Shaahin Angizi, Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. 2018. CMP-PIM: An Energy-Efficient Comparator-Based Processing-in-Memory Neural Network Accelerator. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 105, 6 pages. https://doi.org/10.1145/3195970.3196009

[7] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014), 36–42. https://doi.org/10.1109/MM.2014.55

[8] Ken E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '68 (Spring))*. Association for Computing Machinery, New York, NY, USA, 307âĂŞ314. https://doi.org/10.1145/1468075.1468121

[9] Mohammed Bey Ahmed Khernache, Arezki Laga, and Jalil Boukhobza. 2018. MONTRES-NVM: An External Sorting Algorithm for Hybrid Memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 49–54. https://doi.org/10.1109/NVMSA.2018.00013

[10] Julien Borghetti, Gregory S. Snider, Philip J. Kuekes, J. Joshua Yang, Duncan R. Stewart, and Richard Stanley Williams. 2010. âĂŸMemristiveâĂŹ switches enable âĂŸstatefulâĂŹ logic operations via material implication. *Nature* 464 (April 2010), 873–876. https://doi.org/10.1038/nature08940

[11] Vladimir Brajovic and Takeo Kanade. 1999. A VLSI sorting image sensor: global massively parallel intensity-to-time processing for low-latency adaptive vision. *IEEE Transactions on Robotics and Automation* 15, 1 (1999), 67–75. https://doi.org/10.1109/70.744603

[12] Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. 2012. Sorting on GPUs for large scale datasets: A thorough comparison. *Information Processing & Management* 48, 5 (2012), 903 – 917. https://doi.org/10.1016/j.ipm.2010.11.010 Large-Scale and Distributed Systems for Information Retrieval.

[13] Daniel Cederman and Philippas Tsigas. 2010. GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. *ACM JEA* 14, Article 4 (2010), .84 pages. https://doi.org/10.1145/1498698.1564500

[14] Chaitali Chakrabarti and Li-Yu Wang. 1994. Novel sorting network-based architectures for rank order filters. *IEEE TVLSI* 2, 4 (Dec 1994), 502–507. https://doi.org/10.1109/92.335027

[15] Ren Chen and Viktor K. Prasanna. 2017. Computer Generation of High Throughput and Memory Efficient Sorting Designs on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 28, 11 (2017), 3100–3113. https://doi.org/10.1109/TPDS.2017.2705128

[16] Long Cheng, Hao-Xuan Zheng, Yi Li, Ting-Chang Chang, Simon M. Sze, and Xiangshui Miao. 2020. In-Memory Digital Comparator Based on a Single Multivalued One-Transistor-One-Resistor Memristor. *IEEE Transactions on Electron Devices* 67, 3 (2020), 1293–1296. https://doi.org/10.1109/TED.2020.2967401

[17] Zhaole Chu, Yongping Luo, Peiquan Jin, and Shouhong Wan. 2021. NVMSorting: Efficient Sorting on Non-Volatile Memory. In *The 33rd International Conference on Software Engineering & Knowledge Engineering (SEKE 2021)*.

[18] Alberto Colavita, Enzo Mumolo, and Gabriele Capello. 1997. A novel sorting algorithm and its application to a gamma-ray telescope asynchronous data acquisition system. *Nuclear Inst. and Methods in Physics Research Sec. A* 394, 3 (1997), 374 – 380. https://doi.org/10.1016/S0168-9002(97)00567-6

[19] S. Rasoul Faraji and Kia Bazargan. 2020. Hybrid Binary-Unary Hardware Accelerator. *IEEE Trans. Comput.* 69, 9 (2020), 1308–1319. https://doi.org/10.1109/TC.2020.2971596

[20] Amin Farmahini-Farahani, Henry J. Duwe III, Michael J. Schulte, and Katherine Compton. 2013. Modular Design of High-Throughput, Low-Latency Sorting Units. *IEEE Trans. Comput.* 62, 7 (2013), 1389–1402. https://doi.org/10.1109/TC.2012.108

[21] Brian R. Gaines. 1969. Stochastic Computing Systems. In *Advances in Information Systems Science*. Springer US, 37–172. https://doi.org/10.1007/978-1-4899-5841-9_2

[22] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. 2007. CellSort: High Performance Sorting on the Cell Processor. In *VLDB* (Vienna, Austria). 1286–1297.

[23] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD* (Chicago, IL, USA). 325–336. https://doi.org/10.1145/1142473.1142511

[24] Goetz Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3, Article 10 (Sept. 2006). https://doi.org/10.1145/1132960.1132964

[25] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. FELIX: Fast and Energy-Efficient Logic in Memory. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7. https://doi.org/10.1145/3240765.3240811

[26] Saransh Gupta, Mohsen Imani, Joonseop Sim, Andrew Huang, Fan Wu, M. Hassan Najafi, and Tajana Rosing. 2020. SCRIMP: A General Stochastic Computing Architecture using ReRAM in-Memory Processing. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1598–1601. https://doi.org/10.23919/DATE48585.2020.9116338

[27] Said Hamdioui, Shahar Kvatinsky, Gert Cauwenberghs, Lei Xie, Nimrod Wald, Siddharth Joshi, Hesham Mostafa Elsayed, Henk Corporaal, and Koen Bertels. 2017. Memristor for Computing: Myth or Reality? *(DATE âĂŹ17)*.

[28] Amir Hossein Jalilvand, M. Hassan Najafi, and Mahdi Fazeli. 2020. Fuzzy-logic using Unary Bit-Stream Processing. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5.

[29] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits* 51, 4 (2016), 1009–1021. https://doi.org/10.1109/JSSC.2016.2515510

[30] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (2011), 7–17. https://doi.org/10.1109/MM.2011.89

[31] Dirk Koch and Jim Torresen. 2011. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting. In *FPGA*. https://doi.org/10.1145/1950413.1950427

[32] Gunjae Koo, Kiran Kumar Matam, Te I., H.V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 219–231.

[33] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. 2014. MAGICâĂŤMemristor-Aided Logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899. https://doi.org/10.1109/TCSII.2014.2357292

[34] Shahar Kvatinsky, Misbah Ramadan, Eby G. Friedman, and Avinoam Kolodny. 2015. VTEAM: A General Model for Voltage-Controlled Memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 8 (2015), 786–790. https://doi.org/10.1109/TCSII.2015.2433536

[35] Peng Li, D.J. Lilja, Weikang Qian, K. Bazargan, and M.D. Riedel. 2014. Computation on Stochastic Bit Streams Digital Image Processing Case Studies. *IEEE TVLSI* 22, 3 (2014), 449–462. https://doi.org/10.1109/TVLSI.2013.2247429

[36] Zheyu Li, Nagadastagiri Challapalle, Akshay Krishna Ramanathan, and Vijaykrishnan Narayanan. 2020. IMC-Sort: In-Memory Parallel Sorting Architecture Using Hybrid Memory Cube. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (Virtual Event, China) *(GLSVLSI '20)*. Association for Computing Machinery, New York, NY, USA, 45âĂŞ50. https://doi.org/10.1145/3386263.3407581

[37] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. 2014. Race Logic: A hardware acceleration for dynamic programming algorithms. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 517–528. https://doi.org/10.1109/ISCA.2014.6853226

[38] Soheil Mohajer, Zhiheng Wang, and Kia Bazargan. 2018. Routing Magic: Performing Computations Using Routing Networks and Voting Logic on Unary Encoded Data. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 77âĂŞ86. https://doi.org/10.1145/3174243.3174267

[39] M. Hassan Najafi, S. Rasoul Faraji, Kia Bazargan, and David Lilja. 2020. Energy-Efficient Pulse-Based Convolution for Near-Sensor Processing. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS45731.2020.9181248

[40] M. Hassan Najafi, David Lilja, Marc D. Riedel, and Kia Bazargan. 2018. Low-Cost Sorting Network Circuits Using Unary Processing. *IEEE Trans. on VLSI Systems* 26, 8 (Aug 2018), 1471–1480. https://doi.org/10.1109/TVLSI.2018.2822300

[41] M. Hassan Najafi, David J. Lilja, Marc Riedel, and Kia Bazargan. 2017. Power and Area Efficient Sorting Networks Using Unary Processing. In *2017 IEEE International Conference on Computer Design (ICCD)*. 125–128. https://doi.org/10.1109/ICCD.2017.27

[42] Hoang Anh Du Nguyen, Jintao Yu, Muath Abu Lebdeh, Mottaqiallah Taouil, Said Hamdioui, and Francky Catthoor. 2020. A Classification of Memory-Centric Computing. *J. Emerg. Technol. Comput. Syst.* 16, 2, Article 13 (Jan. 2020), 26 pages. https://doi.org/10.1145/3365837

[43] Eesa Nikahd, Payman Behnam, and Reza Sameni. 2016. High-Speed Hardware Implementation of Fixed and Runtime Variable Window Length 1-D Median Filters. *IEEE Tran. on Circ. & Sys. II: Express Briefs* (2016). https://doi.org/10.1109/TCSII.2015.2504945

[44] Stephan Olarlu, M. Cristina Pinotti, and Si Qing Zheng. 2000. An optimal hardware-algorithm for sorting using a fixed-size parallel sorting device. *IEEE Trans. Comput.* 49, 12 (2000), 1310–1324. https://doi.org/10.1109/12.895849

[45] Kostas Pagiamtzis and Ali Sheikholeslami. 2006. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits* 41, 3 (2006), 712–727. https://doi.org/10.1109/JSSC.2005.864128

[46] David K. Pok, Chien-In Chen, John J. Schamus, Christine T. Montgomery, and James B. Y. Tsui. 1997. Chip design for monobit receiver. *IEEE Trans. on Microwave Theory and Techniques* 45, 12 (Dec 1997), 2283–2295. https://doi.org/10.1109/22.643832

[47] Wolfgang J. Poppelbaum. 1978. Burst Processing: A Deterministic Counterpart to Stochastic Computing. In *Proceedings of the 1st Intern. Symp. on Stochastic Computing and its Apps.*

[48] Wolfgang J. Poppelbaum, A. Dollas, J.B. Glickman, and C. O'Toole. 1987. Unary Processing. In *Advances in Computers*. Vol. 26. Elsevier, 47 – 92. https://doi.org/10.1016/S0065-2458(08)60005-4

[49] Ananth Krishna Prasad, Morteza Rezaalipour, Masoud Dehyadegari, and Mahdi Nazm Bojnordi. 2021. Memristive Data Ranking. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 440–452. https://doi.org/10.1109/HPCA51647.2021.00045

[50] Seth H. Pugsley, Arjun Deb, Rajeev Balasubramonian, and Feifei Li. 2015. Fixed-function hardware sorting accelerators for near data MapReduce execution. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. 439–442. https://doi.org/10.1109/ICCD.2015.7357143

[51] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-Accelerated Near-Storage Sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 106–114. https://doi.org/10.1109/FCCM51124.2021.00020

[52] David Radakovits and Nima TaheriNejad. 2019. Implementation and Characterization of a Memristive Memory System. In *2019 IEEE 32nd Canadian Conference on Electrical and Computer Engineering (CCECE)*. 1–5.

[53] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. Analyzing and Modeling In-Storage Computing Workloads On EISC âĂŤ An FPGA-Based System-Level Emulation Platform. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942135

[54] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 262–272.

[55] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-Performance Adaptive Merge Tree Sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 282–294. https://doi.org/10.1109/ISCA45697.2020.00033

[56] Nadathur Satish, Mark Harris, and Michael Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–10. https://doi.org/10.1109/IPDPS.2009.5161005

[57] James E. Smith. 2018. Space-time Algebra: A Model for Neocortical Computation. In *ISCA'18* (Los Angeles, California). 289–300. https://doi.org/10.1109/ISCA.2018.00033

[58] Donpaul C. Stephens, Jon C. R. Bennett, and Hui Zhang. 1999. Implementing scheduling algorithms in high-speed networks. *IEEE JSAC* 17, 6 (Jun 1999), 1145–1158. https://doi.org/10.1109/49.772449

[59] Nima TaheriNejad. 2021. SIXOR: Single-cycle In-memristor XOR. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 29, 5 (2021), 925–935. https://doi.org/10.1109/TVLSI.2021.3062293

[60] Nima Taherinejad, P. D. Sai Manoj, and Axel Jantsch. 2015. Memristors' Potential for Multi-bit Storage and Pattern Learning. In *2015 IEEE European Modelling Symposium (EMS)*. 450–455. https://doi.org/10.1109/EMS.2015.73

[61] N. TaheriNejad and D. Radakovits. 2019. From Behavioral Design of Memristive Circuits and Systems to Physical Implementations. *IEEE Circuit and Systems (CAS) Magazine* (2019), 1–11.

[62] Nima Taherinejad, Manoj P.D. Sai, Michael Rathmair, and Axel Jantsch. 2016. Fully digital write-in scheme for multi-bit memristive storage. In *2016 13th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*. 1–6. https://doi.org/10.1109/ICEEE.2016.7751193

[63] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2015. Gullfoss : Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources.

[64] Georgios Tzimpragos, Dilip Vasudevan, Nestan Tsiskaridze, George Michelogiannakis, Advait Madhavan, Jennifer Volk, John Shalf, and Timothy Sherwood. 2020. A Computational Temporal Logic for Superconducting Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 435âĂŞ448. https://doi.org/10.1145/3373376.3378517

[65] Di Wu, Jingjie Li, Ruokai Yin, Hsuan Hsiao, Younghyun Kim, and Joshua San Miguel. 2020. UGEMM: Unary Computing Architecture for GEMM Applications. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 377–390. https://doi.org/10.1109/ISCA45697.2020.00040

[66] Lei Xie, Hoang Anh Du Nguyen, Jintao Yu, Ali Kaichouhi, Mottaqiallah Taouil, Mohammad AlFailakawi, and Said Hamdioui. 2017. Scouting logic: A novel memristor-based logic design for resistive computing. In *ISVLSI'17*. 176–181.

[67] Cong Xu, Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. 2011. Design implications of memristor-based RRAM cross-point structures. In *2011 Design, Automation Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763125

[68] Yawen Zhang, Sheng Lin, Runsheng Wang, Yanzhi Wang, Yuan Wang, Weikang Qian, and Ru Huang. 2020. When Sorting Network Meets Parallel Bitstreams: A Fault-Tolerant Parallel Ternary Neural Network Accelerator based on Stochastic Computing. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1287–1290. https://doi.org/10.23919/DATE48585.2020.9116390

[69] Mohammed A. Zidan, John Paul Strachan, and Wei D. Lu. 2018. The future of electronics based on memristive systems. *Nature electronics* 1 (2018), 22–29.