Memristive Data Ranking

Ananth Krishna Prasad*, Morteza Rezaalipour†, Masoud Dehyadegari† and Mahdi Nazm Bojnordi*
*University of Utah, †K.N. Toosi University of Technology
Email: *{ananth, bojnordi}@cs.utah.edu, †{amrezaalipour, dehyadegari}@kntu.ac.ir

***Abstract*—Sorting is a fundamental operation in many large-scale data processing applications. In big data computing, sorting imposes a massive requirement on the available memory bandwidth because of its natural demand for pairwise comparison. This high bandwidth requirement often leads to a significant degradation in performance and energy-efficiency. Processing-in-memory has been examined as an effective solution to the memory bandwidth problem for SIMD and data-parallel operations, which does not necessarily solve the bandwidth problem for pairwise comparison. This paper proposes a viable hardware/software mechanism for performing large-scale data ranking in memory with a bandwidth complexity of $O(1)$. Large-scale comparison that forms the core computation of sorting algorithms is reformulated in terms of novel bit-level operations within the physical memory arrays for in-situ ranking, thereby eliminating the need for any pairwise comparison outside the memory arrays. The proposed mechanism, called RIME, provides an API library granting the user application sufficient control over the fundamental operations for in-situ ranking, sorting, and merging. Our simulation results on a set of high-performance parallel sorting kernels indicate $12.4 - 50.7\times$ throughput gains for RIME. When used for ranking and sorting in a set of database applications, graph analytics, and network processing, RIME achieves more than 90% energy reduction and $2.3-43.6\times$ performance improvements.**

I. Introduction

The continued growth in IoT, mobile devices, and cloud-based services have led to the emergence of large datasets and big data workloads. Analyzing, querying, and filtering massive amounts of data in a structured manner becomes increasingly hard. In these cases, a large amount of data often require to be sorted, either because of dataset properties [1], or real-time requests from web users [2], or algorithm features [3]. Also, sorting data is often the key to enabling efficient searching algorithms [4]. Data clustering, an important kernel in data mining applications, depends heavily on sort and search operations [5]. Therefore, sorting an array of numbers is an active area of research and a vital operation in many application domains such as image processing, database processing, genome analysis, and text analysis [6].

Several sorting algorithms were invented in every decade to be well adapted to computer architecture and distributions of data, such as radixsort [7], mergesort [8], and quicksort [9]. Further research has been conducted to identify efficient sorting algorithms using hardware accelerators [10], multiple cores by exploiting SIMD instructions [11, 12], GPUs [13], and ASIC [14]. With the increase in the computational capability of processors and GPUs by enabling more cores and threads, the demand for memory bandwidth increases proportionally [15]. For large datasets of size magnitudes larger than the on-chip cache capacity, the demand for high memory bandwidth results in sorting performance being bottlenecked by the limited off-chip memory bandwidth. Large scale memory management [16] and in-memory databases [17] have been recently explored as a promising solution to the data movement and bandwidth challenges. The efficiency of these techniques primarily depends on minimizing data movement between the processor cores and off-chip memory using a hierarchy of memories, non-uniform access to memory, transactional memories, and non-volatile technologies. Nevertheless, such optimizations do not eliminate, but rather mitigate the extent of data accesses to perform sorting on the processing core.

The recent advent of emerging memory interfaces [18, 19] and cell technologies [20, 21] has enabled in-memory computation with large-scale data parallel operations, such as bitwise XOR. Prior work on processing in memory (PIM) has shown various applications ranging from combinatorial optimization [22, 23] and neural network computation [24–26] to graph analytics [27]. The existing PIM solutions mostly focus on accelerating matrix/vector operations inside memory arrays or utilizing high bandwidth interfaces for near data computation. Instead, RIME proposes an in-situ approach for memristive ranking-in-memory using a HW/SW co-design that minimizes the bandwidth requirements of sort algorithms. The main contributions of this paper are as follows. (1) Large-scale sorting workloads are characterized in terms of bandwidth and throughput requirements. The primary reason for poor performance of sorting at low bandwidths is identified. (2) A novel memory system architecture is designed to enable in-memory min/max computation using a large-scale massively parallel bitwise algorithm. (3) The necessary driver support and userspace API are provided to enable fine-grained control over the proposed system for efficient in-situ ranking and ordinary memory operations. (4) Detailed evaluations of the proposed architecture at the system and circuit levels are provided, which indicate significant performance improvements and energy savings over the existing systems.

II. Background and Motivations

A. Applications of Sorting

Sorting is a fundamental operation in database applications. For example, sorting is very common in query retrieval to prepare the query results in a particular order by using OrderBy clause. In addition, sorting may be necessary in several join operations such as sort-merge join algorithm. It serves in index creation, user-requested output sorting, ranking, duplicate removal, and grouping operations [28]. Numerous techniques have been proposed ro realize efficient sorting based on multi-core processors, GPUs, and SIMD architectures [4].

MapReduce is used to perform massive data sorting in distributed system. In particular, Shuffle is an important part of MapReduce that performs sorting and transferring outputs of the maps to reducers [29]. The execution time of algorithms such as Kruskal is dominated by sorting. Prim's string processing and Dijkstra's algorithms are based on the priority queue, which relies on sorting and ranking data in a queue. Many other applications such as numerical computations, combinatorial search, operations research, and commercial computing are often based on sorting [30]. Moreover, sorting the retrieval results from PageRank, HillTop, and HITS (Hypertext Induced Topic Search) in a reasonable time is a significant challenge [31]. Not only sorting integer values is important but also several applications need sorting real-valued data, which is not as simple as integer values. For example, Kim et al. [32] exploits integer arithmetic on floating-point data to reduce the execution time.

B. Sorting Algorithms

Quicksort. Quicksort was first introduced by Sir C. A. R. Hoare in 1961 [33]. Quicksort is based on the divide-and-conquer paradigm that resolves a complex problem by constantly dividing it into simpler subproblems until it reaches a point where the solution to the subproblems becomes trivial. The algorithm starts with a *Partition* phase in which a *bound* element (or a *pivot*) is selected from the given array as a dividing line for partitioning it into two smaller *segments* (or sub-arrays). At the end of the *Partition* phase, if the size of sub-arrays is less than a *cut-off* amount, i.e., the solution becomes trivial, the sub-arrays may then be sorted by known methods; or even by employing programs specialized for sorting arrays containing less than cut-off elements. Conversely, if the sub-arrays are fairly large, the partitioning process continues for further division of the sub-arrays into even smaller ones. The time complexity of Quicksort is reported to be of the order of $O(n^2)$ and $O(n \log n)$, for the worst and average cases, respectively [34].

Mergesort. Mergesort was suggested by John von Neuman as early as 1945 [34]. Similar to Quicksort, Mergesort also employs the divide-and-conquer paradigm, and it recursively sorts a given array of elements. As the name of the algorithm represents, Mergesort consists of a merge algorithm and recursive calls. The merge algorithm takes two or more non-empty sorted arrays and outputs a final array that is also sorted. Generally, Mergesort first divides the input array into multiple sub-arrays, each containing only a single element, by recursive calls; a subarray that contains only a single element is considered to be sorted. Then, it repeatedly merges the subarrays until there remains only one array, which is the sorted output array [34].

Radixsort. Radixsort method employs a different scheme compared to the previous sorting algorithms as it looks through the individual digits of elements to perform a digit-inspection process. For d-digit elements, starting from the most significant digit (MSD) to the least significant digit (LSD), the algorithm sorts the elements, considering only one digit at a time, in a way that all the elements that have smaller digits appear on the left-hand side of elements with larger digits. By iterating this process from d-1 to 0, the input array will be sorted. Radixsort may also be applied in the opposite digit direction (i.e., from LSD to MSD) [34].

Heapsort. Heapsort is based on the heap data structure, which is a complete binary tree of data points. The maximum or minimum value is always located at the root of the heap tree. During each iteration of the algorithm, Heapsort removes the root node from the array and substitue the root node with the last element of the array and reheap the array [35]. The time complexity of Heapsort is $O(n \log n)$.

C. Design Challenges and Opportunities

1) Memory Bandwidth Requirements: First, not all sort algorithms exhibit the same bandwidth requirements. Figure 1 shows the number of accesses served by a memory system below the on-die cache for Mergesort (M/S), Quicksort (Q/S), and Radixsort (R/S). We consider two memory configurations for this analysis: one with an unlimited bandwidth and the other with an off-chip memory interface [36].[1] Increasing the workload size on the bandwidth-unlimited system results in a higher number of memory accesses (Figure 1(a)), which may be influenced by the number of processor cores (Figure 1(b)). In real memory systems, however, the bandwidth is limited. Figure 1 (c) shows the sustained memory bandwidth is more restricted as the number of cores varies from 1 to 64.

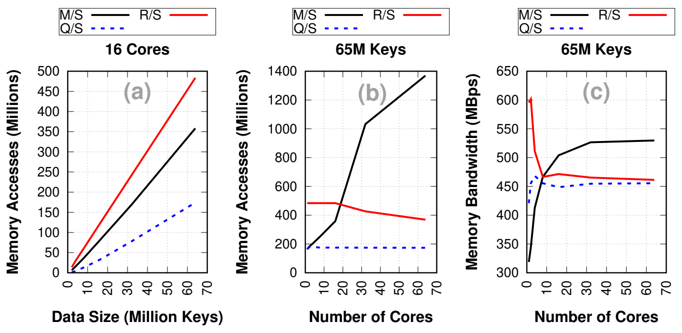


Fig. 1. Bandwidth requirements for sort algorithms.

Second, the performance of sort algorithms is sensitive to the available memory bandwidth. Figure 2 shows the throughput of the sort algorithms, in terms of million keys per second (MKps), on three systems with different available bandwidths. For this analysis, in addition to the unlimited and off-chip bandwidths, we consider a high bandwidth memory system with an in-package DRAM [37]. In an ideal memory system with unlimited bandwidth (a), R/S outperforms both Q/S and M/S at the cost of exerting significant data movement on the memory interface. This superiority, however, is taken by Q/S in the realistic memory systems with limited bandwidth–i.e., the in-package (b) and off-chip (c) memories.

[1]Section VI provides the detailed system configuration.

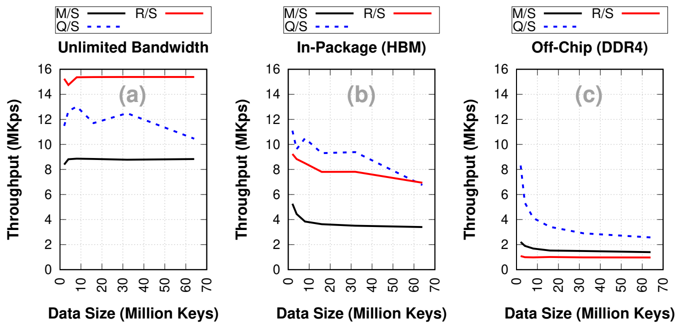


Fig. 2. Impact of available bandwidth on performance.

2) Opportunities and Potentials: The above analyses on sorting algorithms indicate that (1) the bandwidth requirement scales linearly with the size of working set and (2) the throughput of sorting is limited by bandwidth. Similar observations have been made by the prior work on StreamBox-HBM [38], where a sort-merge based algorithm for streaming computation outperforms hash-join based approaches for in-package memories. The prior work shows that throughput of GroupBy, one of the key kernels in streaming computation, increases linearly with increasing the number of cores for HBM, while it stagnates beyond 16-cores for DRAM. Thus, if there is way to remove this bandwidth bottleneck for sorting, performance can be massively improved.

One of the main reasons sorting requires a large bandwidth lies at the heart of the algorithms (i.e., comparison). In naive terms, worst-case sorting requires all possible pairwise comparison of values. Even though more sophisticated algorithms such as Quicksort, Radixsort, and Heapsort improve the bandwidth efficiency, still they don't solve the underlying issue, which is access to pairs of values in memory. In contrast, RIME enables large-scale in-situ bitwise comparison that massively improves the bandwidth efficiency by eliminating unnecessary data movement on the memory interface. Given the immense applications large-scale data sorting has, this can potentially massively accelerate sorting kernels as part of large-scale data processing applications.

D. Memristive Array Structure

Memristive technology has been promoted as an alternative to the conventional memories due to their scalability, non-volatility, and being free of leakage power. Moreover, they have shown unique capabilities for efficient in-memory processing. In particular, resistive RAM (RRAM) is one of the most promising memristive devices under commercial development that shows great potential for building main memory systems [21]. Numerous cell architectures have been proposed in the literature that optimize RRAM for better reliability, density, and computational capabilities. 1R crosspoints are denser, but lacking isolated access to individual rows and columns [40]. As the proposed in-situ approach requires isolated column access a 1T1R memory cell (Figure 3) is preferred over the 1R crosspoint.

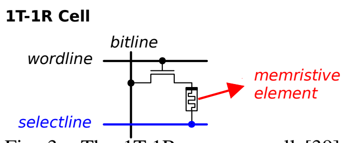


Fig. 3. The 1T-1R memory cell [39] used for RIME.

III. Design Overview

A. In-Memory Min/Max Computation

Inspired by the prior work on bit-serial median filters [41, 42], we design a new algorithm for computing the minimum (or maximum) of any N numbers in k serial steps, where k is the number of bits used for representing each number. The proposed algorithm is applicable to signed/unsigned fixed-point and floating-point number formats.

1) Unsigned Fixed-Point Numbers: We consider α integer bits and β fraction bits to represent unsigned fixed-point numbers. Every k-bit number is represented in the form of $\overline{b_{\alpha-1}\cdots b_0 \bullet b_{-1}\cdots b_{-\beta}}$, where b_is are the binary digits and $k = \alpha + \beta$. (Typically, β is set to 0 for representing pure integer numbers.) The value of each number is computed by $\sum_{i=-\beta}^{\alpha-1} 2^i b_i$. Therefore, a number with more leading 0s produces a smaller value; for example, 0001.11 is less than 0010.00. We employ this simple principle to design a bit-serial algorithm for finding the minimum (or maximum) of multiple numbers in a *set*. As shown in Algorithm 1, starting from the most significant bit position (i.e., $k-1$), we follow a k-step algorithm to examine the binary values of all bit positions (i.e., pos). At every step, some of the non-minimum (or non-maximum) values may be removed from the set. First, we search for 1 at the current bit position (pos) to form a selection of matching numbers (sel). The selected numbers are removed from the set only if the *set* and *sel* are not equal. As a result, all the final remaining numbers in the set have the minimum value.

Algorithm 1 Find the minimum of unsigned fixed-points

```
1:  $set \leftarrow \{\text{all numbers}\}$
2:  for $pos$ in $(k-1, \cdots, 1, 0)$ do
3:      $sel \leftarrow \varnothing$
4:      for all $num \in set$ do
5:          if $num_{pos} = 1$ then $sel \leftarrow sel \cup \{num\}$
6:          end if
7:      end for
8:      if $sel \neq set$ then $set \leftarrow set - sel$
9:      end if
10: end for
```

Figure 4 shows how the proposed algorithm finds the minimum of 5 unsigned fixed-point numbers with $\alpha = 3$ and $\beta = 2$. First, the most significant bit of all numbers are compared with 1 and the matching numbers (i.e., 4.00 and 6.50) are excluded from the set (Step 1). We then compare the second most significant bit of the remaining numbers with 1. As we find no matches, none of the numbers is removed from the set during Step 2. We repeat this process for the next bit position during Step 3. As all the remaining numbers have a matching 1 in the third bit position, none of the numbers should be excluded from the set. During Steps 4 and 5, the next matching numbers, respectively 1.75 and 1.25, are excluded from the set. Finally, the remaining number in the set (i.e., 1.00) represents the minimum value of the given numbers.

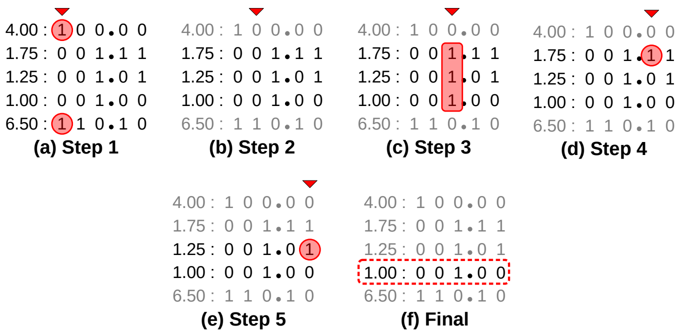


Fig. 4. Illustrative example of finding the minimum of 5 unsigned fixed-point numbers.

2) Signed Fixed-Point Numbers: We use the two's complement format for representing signed fixed-point numbers in the form of $\overline{sb_{\alpha-2}\cdots b_0 \bullet b_{-1}\cdots b_{-\beta}}$, where s is the sign bit. Every signed fixed-point value may be computed by $-2^{\alpha-1}s + \sum_{i=-\beta}^{\alpha-2} 2^i b_i$. Similar to unsigned values, having 0s in more significant b_is results in a smaller value. However, a 1 in the sign bit position makes the value negative. To support signed numbers, we change Algorithm 1 to search for matching 0s (instead of 1s) in the first iteration of the loop ($pos = k - 1$). Therefore, the proposed algorithm can exclude all the positive values from the set during Step 1 if a mix of positive and negative numbers is given. If only positive numbers are present, corresponding to the case where all bit-values in the first iteration of the loop being zero, the operation proceeds to search for matching 0s in the succeeding iterations to find the minimum magnitude

3) Floating-Point Numbers: The IEEE standard for floating-point arithmetic (IEEE 754) proposes a three-segment layout for real-valued numbers comprising one sign bit (s), a multi-bit exponent (e), and a multi-bit fraction (f). Every floating-point value may be computed by $(-1)^s \times (1 + f) \times 2^{e-b}$, where b is a positive bias added to the exponent.

Similar to signed fixed-point numbers, at the sign bit position, the algorithm searches for 0s to remove from the set. A constant offset is added to the exponent bits, but that doesn't change the monotonic relationship between the actual exponent and the magnitude of the value represented in the exponent bits. This makes it such that there is virtually no difference in the algorithm between signed fixed-point numbers and floating point numbers.

Figure 5 shows an example for finding the minimum of 3 numbers in a hypothetical 8-bit floating-point format similiar to IEEE 754, with 4 mantissa bits and 3 exponent bits. At the first step, the sign bit is checked. At the second step, given that all values in the first (sign) column were not zero, the algorithm searches for 1s to find the number with the maximum possible magnitude. After step 4, only one selected value remains, and that is the minimum value.

B. Rank/Sort/Merge Operation

In-memory min/max computation can significantly alleviate the bandwidth costs of large-scale ranking, sorting, and

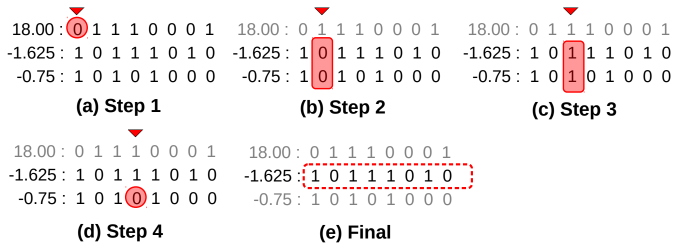


Fig. 5. Illustrative example of finding the minimum of 3 floating-point numbers.

merging operations.

1) Sorting: As established before, conventional sort algorithms require a significant memory bandwidth due to their complex access patterns for comparing pairs of data points. Depending on the type of algorithm, the complexity of memory bandwidth for sorting N data points may vary between $O(NlogN)$ and $O(N^2)$ for large data sets [43].[2] Our proposed hardware/software approach lowers the bandwidth complexity of sort operations to $O(N)$ eliminating the unnecessary data movement for finding min/max of given data points. From the software point of view, the proposed sort operation is carried out similar to reading data from an array of values. For a specific data range in memory, every access can provide the next minimum value of the array. Therefore, repeating this process N times results in an ordered stream of data from memory to the processor. First, the in-memory min/max computer is initialized for a new data range in the memory. On every sort access, the next minimum value of the data range is computed and sent to the processor. Also, the newly found data is flagged for exclusion from the data range for the next sort accesses. The exclusion flags remain until the hardware is initialized for a new sort operation or the data memory is released through APIs provided (explained in Section V).

2) Ranking: Similar to sorting, conventional data ranking algorithms consume a significant memory bandwidth. A natural way of finding the kth ordered item of N numbers is to repeat a sort algorithm until reaching the kth min/max of the numbers. This approach may result in a bandwidth complexity of $O(kN)$. Using the proposed in-memory min/max finder, we can decrease the bandwidth costs of finding the kth ordered value in a data range to k accesses, which indicates a bandwidth complexity of $O(k)$. For a given k, the in-memory hardware repeats min/max computation for k iterations until the desired value is found.

3) Merging: A merge operation refers to combining two (or more) data sets into a single ordered set of data. The resultant set may include all members of the input sets or only data points that exists in all input sets (a.k.a., merge-join in databases). Having a sorting algorithm at its heart, the bandwidth complexity of merging is the same as that of sorting. The conventional merge operations require a bandwidth complexity as low as $O(NlogN)$, where N is the size of the resultant merged data; whereas, our proposed hard-

[2]The memory bandwidth complexity significantly reduces for small data sets that entirely or partially fit in the on-chip cache.

ware/software solution reduces this complexity down to $O(N)$. To support fast merge operations, the in-memory hardware implements concurrent min/max computation on multiple data ranges. Figure 6 shows how to merge two data sets (A and B) into a stream of ordered numbers. After initializing the data ranges, software reads the first minimum value from each data sets (i.e., 1 and 4). The smaller min value is 1 from A, which is selected for the output stream. As a replacement for this value, the next min value is read from A and the min selection process repeats until all the values from both sets are accessed. In the case of a merge-join operation, the output stream will only include the min values that exist in both sets (i.e., 5).

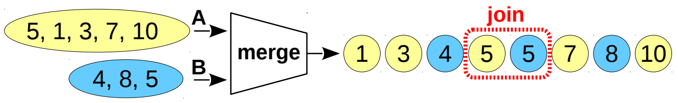


Fig. 6. Illustrative example of merging two data sets.

IV. Proposed Architecture

Integrating a min/max compute logic in memory chips may introduce significant overheads in terms of performance, energy, and memory capacity. To minimize the overheads, we propose minimal changes to the periphery and organization of conventional memristive arrays for in-situ value ranking.

A. Memristive In-Situ Ranking

As explained in Section III-A, the key operation for bit-serial min/max computation is a repetitive search for *bit value* (1 or 0) within individual columns of a data array. At every step, the outcome of the search is a match vector indicating which rows of the array should be excluded from the data set. As a result, the memory array needs to support two new operations for bitwise column search and selective row exclusion. To enable these new operations, we choose the conventional 1T1R structure explained in Section II-D. As shown in Figure 7, we propose extra control mechanism at each memristive array to enable wordline activation selectively and match vector generation on the selectlines, iteratively.

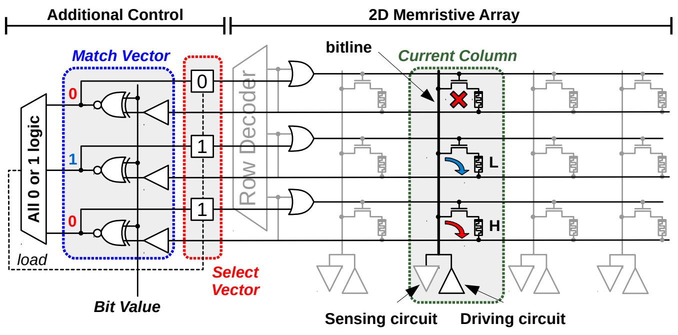


Fig. 7. Enabling selective wordline activation and match vector generation.

1) Bitwise Column Search: Every column search is performed on a set of selected cells within a particular column of the array. A *select vector*, connected to all the wordlines, determines the selected cells for each column search operation. Initially, all memristive rows containing the data points are selected by the select vector (Section IV-B2). The data points are represented with multi-bit values stored in single-level memory cells, where each cell represents a single bit of a value. Therefore, each column of the array includes a bit value from multiple data points. For every bitwise search, the bitline driver of the current column is first enabled to make a read current flow through the bitline. The bitline is connected to the binary cells that represent 0 and 1 using high (H) and low (L) resistance states, respectively. Ideally, the bitline currents reach the selectlines (recall Section II-D) only after passing through the selected cells that represent 1 with their low resistance states. In practice, however, the cells with high resistance state pass current too. But the significance of current flowing though each memristive cells is inversely proportional to its resistive state. To make a near-ideal situation for bitwise search, we choose memristive devices that provide a large dynamic range of resistance states (i.e., R_H is much bigger than R_L). By sensing the selectlines, we perform a column read at the array periphery. As shown in Figure 7, the result undergoes a bitwise XNOR with the reference bit value, a 1-bit search key, to generate a *match vector*.

2) Selective Row Exclusion: Row exclusion is performed through loading the generated match vector into the select vector, where more 1s may be turned into 0s. Therefore, we reduce the number of selected rows for the next iteration of bit-serial min/max computation. To ensure only non-minimal values are excluded from the data set (Section III-A), the newly generated match vector is only loaded into the select vector if at least one of the selected cells differs from the others. As shown in Figure 7, the All 0 or 1 Logic block generates the load signal for the select vector latches. At the end of each column operation, the contents of the select vector are updated only if the load signal is driven high.

B. Memory Organization

1) Mat Architecture: One major component of the additional circuit for computing min/max within memory arrays is the sensing circuit at each selectline for producing the match vector. In a conventional memristive array, the sensing circuits are connected to the bitlines for reading a row of the array, whereas the proposed column search operation needs the sensing circuits at the selectlines. We propose a physical structure for the memristive arrays that enables sharing the sensing circuits between read and column search operations. Figure 8 shows how every four arrays within a mat share the sensing and driving circuits for row read, column search, and row write operations. At the center of each mat, a controller is employed to operate the sense amps and drivers appropriately for the received read, write, and column search commands. All four memristive arrays are active during each mat command to perform a bit parallel access. The outcome of each column search is a binary signal indicating if at least one of the mat arrays requires a row exclusion. This signal is then sent upstream to the chip controller for further process.

2) Chip Organization: Building upon the proposed mat structure, we design a memristive chip capable of storing data points and performing in-situ min/max computation. Every chip comprises a controller and multiple banks that are

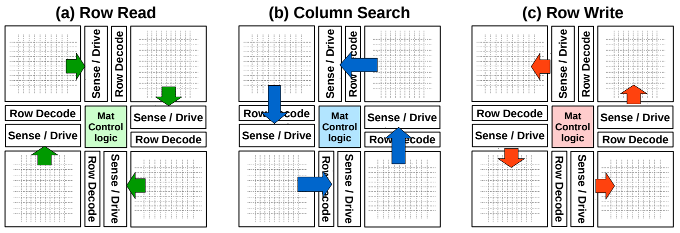


Fig. 8. Sharing sense and drive circuits for row read (a), column search (b), and row write (c) operations.

connected using a data/index H-tree. The banks are further divided into subbanks, which are similarly connected using an internal data/index H-tree. Each subbank comprises multiple mats and a selector to keep only one active mat per access.

Multi-Mat Management. Each mat is designed to compute the min/max value of the data points independently. Though, not all data may fit in a single mat. Therefore, a multi-mat management is necessary to enable computing the min/max value of larger data sets. Along these lines, we design a special data/index tree that transfers data and address in both directions between the chip controller and mats. (In the conventional memory architecture, interconnection trees are typically used for sending address and control in one direction only, from the controller to memory arrays.) This capability is necessary for two reasons: (1) the memory location of the result is needed after every min/max computation and (2) a global knowledge about all data points is necessary to accurately perform a column exclusion across multiple mats. Figure 9 shows an example that needs global knowledge for a multi-mat exclusion. A column search command is sent to 3 mats for finding 1s at the second most significant bit of all the data points. The local search results are zero, all, and one matches in Mats 0, 1, and 2 respectively. Following the local mat computation steps, Mats 0 and 1 should not exclude any data points due to having zero and all matches. This, however, results in not excluding the numbers in Mat 1, which is wrong. Instead, in a multi-mat row exclusion, all 3 mats need to be checked for the row exclusion needs. Each mat, after comparison, returns 2 signals to the controller, one value being the output of the "all 0 or all 1" logic, and the other one indicating if there was a 1 in the column. In the example case, the mat returns 00, 01 and 10 to the controller for mats 0, 1, and 2 respectively. Based on this, the controller decides which rows to exclude in computation. To realize this mechanism efficiently, we use a specialized data/index tree that ORes all the exclusion signals from the mats to signal the chip controller for a required update to the select vectors.

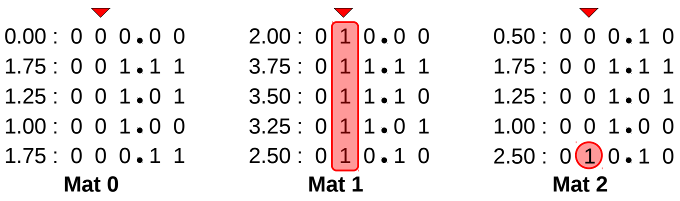


Fig. 9. Multi-mat row exclusion.

Tree-based Index Computation. Upon completing a min/max operation, which finds the global min/max output across the span of selected mats, the data/index tree is expected to compute the memory address of the output. We design the data/index tree to act as a priority encoder that selects only one min/max value per bitwise column search. The outcome of each min/max computation is a multi-bit index progressively produced in the data/index tree and sent upstream from the arrays. Each bit of the index is generated by one of the tree nodes along the path. Figure 10 shows an example index generation for 16 arrays across 4 memristive mats, where arrays 2, 7, and 12 contain the min/max value. Each mat generates a binary signal (i.e., E) indicating if it contains the min/max value[3] and an initial index (i.e., A) representing which array/row has the min/max value.[4] At every node of the tree, A_i and E_i signals that form the two children are combined to generate A_n and E_n. E_n is a binary signal computed by ORing the same signals produced by the children. A_n is, however, a multi-bit value produced through concatenating a most significant bit to a selected index from the children. A_0 is selected if E_0 is 1; otherwise, we choose A_1. The additional bit is computed by $\overline{E_0} \wedge E_1$.

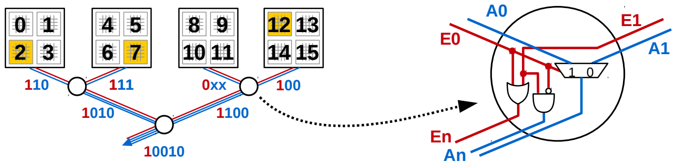


Fig. 10. Calculating the address of the minimum value in H-trees.

The output of index-reduction per column-wise computation is sent to the chip controller, which performs per-mat global select vector update. The process continues till either we reach LSB or only 1 selected value is left across all selected mats. Once we reach LSB, and multiple selected values are left, all selected values are the same minimum value in the dataset. The output of the index reduction tree at this stage would correspond to the array with the lowest address having the mimimum value which ensures stable sort of data.

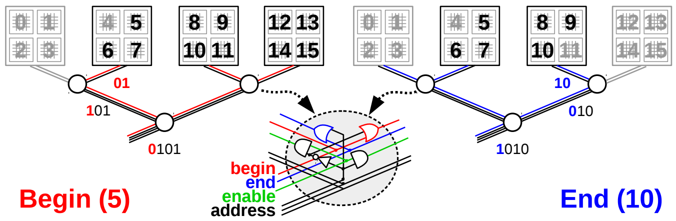


Fig. 11. Initializing the select vectors via H-trees.

Select Vector Initialization. To make use of the proposed in-situ accelerator, it is necessary to initialize the select vector of all memristive arrays containing the data points prior to the iterative Min/Max computation. We allow the application to determine an address range for every initialization at the software level (Section V). The process is then completed in hardware by sending the *begin* and *end* of the address range

[3]The exclusion signal is reused for this purpose.
[4]Priority is always given to the smaller indices.

to the data/index tree. From root to leaves, the *begin* and *end* exclude all tree branches with addresses respectively below and above the address range. At the memristive arrays of the remaining branches, the select bit of each row is set to 1 if it is within the range; otherwise, it is set to 0. Figure 11 shows how the *begin* and *end* signals, specifying an address range from 5 to 10, are sent to the memory arrays. Every node of the tree relays signals to one or both of its children based a certain address bit. This way, we use a fast and efficient way for initializing select bits prior to in-situ computation.

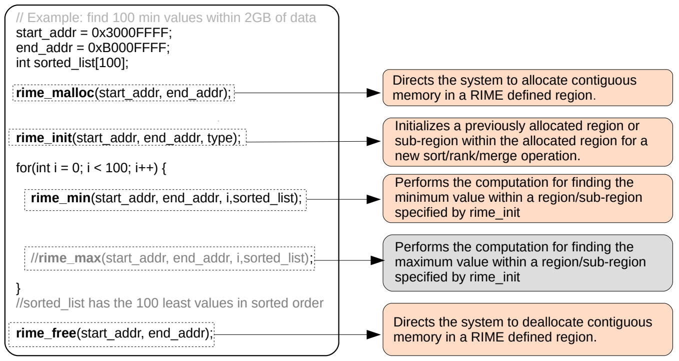
V. Software-Hardware Interface

In this paper, we use a DDR4 [36] interface to enable fast and efficient byte-addressable communication between software and the proposed accelerator. Depending on the application requirements, modules of the proposed memristive architecture may be included in the system for storage and in-memory data ranking purposes. A small fraction of the address space visible to software within every chip is mapped to an internal RAM array, and is used for implementing the data buffers and the configuration parameters. Software configures the on-chip data layout and initiates the optimization by writing to a memory mapped control register. Both memory configuration and data transfer accesses are performed through ordinary DDR4 reads and writes. This is made possible by making all accesses to the accelerator in-order strong-uncacheable.

DIMM Organization. To support large-scale data ranking problems whose working set does not fit within a single chip, it is possible to interconnect multiple RIME chips under a dual in-line memory module (DIMM) [44]. Moreover, a system may include multiple DIMMs for larger data. Each DIMM is configured to be either used in the RIME mode or normal storage mode, decided at the system boot time. Runtime reconfigurability between the RIME and normal storage modes is not allowed owing to constraints imposed by the tree-based index reduction architecture (more details are provided in Section V). Each DIMM is equipped with control registers, data buffers, and a controller. The controller receives the DDR4 commands, data, and address bits from the external interface, and orchestrates the necessary data movement and computation among all of the chips on the DIMM.

Software Support. The proposed system provides a userspace API library for efficient utilization of the in-memory processing capabilities by the user. The API enables applications to (1) allocate memory in the accelerator, (2) configure hardware prior to each computation, and (3) compute the min/max of the dataset. Any allocated memory in the accelerator may be used as normal with load and store instructions, within the constraints imposed by the tree based index reduction method.

The various functions offered as part of the RIME API, along with usage, are shown as part of an example code snippet in Figure 12. We design `rime_init()`, `rime_min()`, and `rime_max()` as part of the API to initialize the hardware and compute the minimum and maximum values, respectively. The 3rd argument of `rime_init()` function call specifies the data type stored in the memory. `rime_min()` and `rime_max()` have an argument that takes a pointer to the target sorted array. The main operation of `rime_init()` is configuring the data/index tree and setting the operational mode of the chip controller. Also, `rime_init()` allows for defining a sub-region within a region defined by `rime_malloc()` for min/max operation through appropriate arguments. This capability enables user flexibility in controlling the ranges of data in consideration for specific operations.



```
// Example: find 100 min values within 2GB of data
start_addr = 0x3000FFFF;
end_addr = 0xB000FFFF;
int sorted_list[100];

rime_malloc(start_addr, end_addr);

rime_init(start_addr, end_addr, type);

for(int i = 0; i < 100; i++) {

  rime_min(start_addr, end_addr, i,sorted_list);

  //rime_max(start_addr, end_addr, i,sorted_list);
}
//sorted_list has the 100 least values in sorted order

rime_free(start_addr, end_addr);
```

Directs the system to allocate contiguous memory in a RIME defined region.

Initializes a previously allocated region or sub-region within the allocated region for a new sort/rank/merge operation.

Performs the computation for finding the minimum value within a region/sub-region specified by rime_init

Performs the computation for finding the maximum value within a region/sub-region specified by rime_init

Directs the system to deallocate contiguous memory in a RIME defined region.

Fig. 12. Example code snippet for forming a sorted list.

Memory Allocation for RIME. The tree-based reduction connects multiple physically contiguous mats in a sub-bank (Section IV-B2). This makes it necessary that large chunks of contiguous virtual memory be allocated to contiguous mats of physical memory to efficiently utilize the tree-based reduction method. Moreover, reservation of such virtual pages onto a contiguous physical space on demand could become impossible due to physical memory fragmentation. This necessitates that there exist no fragmented physical region allocation to virtual space when `rime_malloc` is called. RIME ensures this requirement through a driver that avoids fragmentation prone allocation on the RIME defined address spaces. The driver has tunable parameters to specify the number of pages that should be reserved on startup during an `mmap` call, and the number of additional pages to reserve when the initially reserved block gets full (similar to many malloc implementations). When the available reserved blocks are all taken, the driver reserves additional contiguous physical memory and expands the existing allocated memory region.

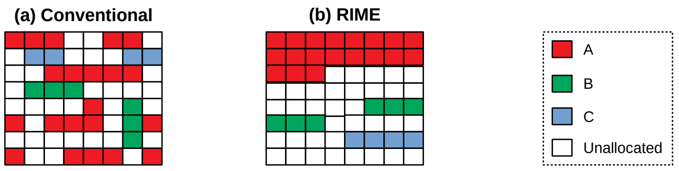


Fig. 13. Physical page allocation for malloc in the normal storage mode (a) and RIME defined DIMMs (b).

Such a difference in memory allocation between the RIME defined and normal storage regions is highlighted in Figure 13. Each small square in the figure denotes a physical page. There are three instances of malloc calls (A, B and C), each of different size, within a region reserved by `mmap` for the normal storage mode (a) and the contiguous RIME mode (b). In the conventional case of memory allocation, a virtually contiguous address space of multiple pages may not be mapped to physically noncontiguous pages. In such a case, it is highly inefficient to perform RIME operations because the system cannot exploit its reduction tree to efficiently compute the minimum value index for every column-wise comparison. In the RIME defined regions, physical pages of each malloc are contiguous, thereby utilizing the tree reduction, efficiently.

One drawback of the contiguous physical page allocation is that if the size of `rime_malloc` request exceeds the size of any physically unallocated contiguous space in the RIME region, memory allocation for that malloc is not possible. This is accounted for in the `rime_malloc` implementation, which returns a null pointer in such cases. Therefore, the user can try using `rime_free` to free up unnecessary allocated memory within the RIME region and try memory allocation again. It is notable that a contiguous physical region is only necessary if the DIMM address space should be used for RIME computation. In the case of using the DIMM for normal memory purposes, the conventional allocation mechanism is sufficient.

Address Mapping and Multi-DIMM Support. DRAM address mappings may be interleaved at fine granularity across channels to exploit further parallelism during block transfer. A RIME DIMM does not allow for such address mapping (Section V): assume two 1GB single-DIMM channels (RIME 0 and RIME 1); the address space `0x00000000–0x3FFFFFFF` maps to RIME 0 and `0x40000000–0x7FFFFFFF` maps to RIME 1.[5] Each *rime_min*/*rime_max* call is accompanied by the starting/ending addresses of the target data range as arguments (Figure 12). Therefore, all the chips within each RIME DIMM are configured for the operational address ranges. If the data spans more than one channels, the API sends multiple such commands to the RIME DIMMs.

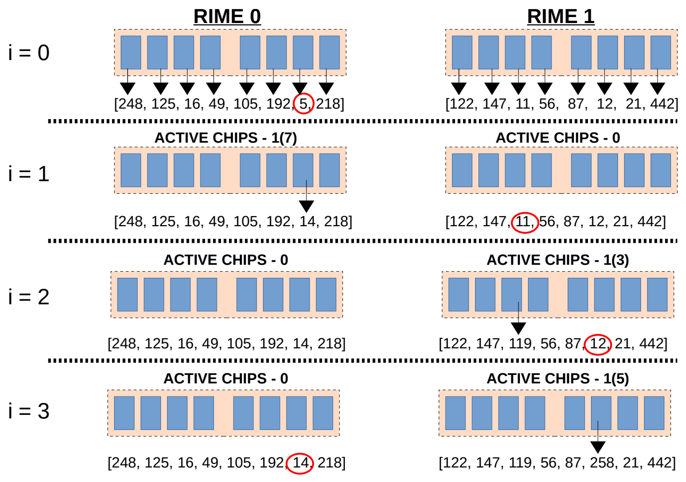


Fig. 14. Example of sorting in case of data spread across multiple channels

Figure 14 shows an example two channel RIME system, where each channel has 8 chips. Four iterations of the for-loop in Figure 12 are shown. During the first iteration, software activates all the chips across all DIMMs to receive a single min/max value from each chip.[6] The library buffers these values and performs a comparison in CPU to find the absolute min/max value (circled in the figure). Next, only the chip that had the minimum value in the previous iteration will be active, which returns a new min/max value to replace the previous one. For example, at i = 1, the chip in RIME 1 which earier computed the minimum value of 5 needs to compute a new minimum value This process will continue for 100 iterations to find the 100 minimum values. The extra buffered values are discarded when a new `rime_init()` is called for the same address range/sub-range, and proceeding *rime_min*/*rime_max* follow the same approach.

VI. Experimental Setup

A. Architecture

Based on the prior work on ESESC [45], we develop a QEMU-based cycle accurate simulator to model a multicore out-of-order processor. For the baseline systems, we interface the processor to cycle accurate components for an off-chip main memory using DDR4 DRAM [36] and an eight-vault HBM [37]. To realize the proposed API and software support, we modify QEMU for an extended version of memkind library [46] that enables special memory allocation and in-memory ranking. Table I shows the simulation parameters.

TABLE I
Simulation parameters.

	Core Type	64 4-issue cores, 2 GHz, 256 ROB entries
Cache	**Instruction L1**	32KB, direct-mapped, 64B block, hit/miss: 2/2
	Data L1	32KB, 4-way, LRU, 64B block, hit/miss: 2/2, MESI
	Shared L2	8MB, 16-way, LRU, 64B block, hit/miss: 15/12
HBM	**Memory Configuration**	2KB row buffer, 2GB DDR4-2000, Channels/Ranks/Banks: 4/8/8
	Timing (CPU cycles)	tRCD:44, tCAS:44, tCCD:16, tWTR:31, tWR:4, tRTP:46, tBL:4, tCWD:61, tRP:44, tRRD:16, tRAS:112, tRC:271, tFAW:181
Main	**Memory Configuration**	8KB row buffer, 8Gb DDR4-1600 chips, Channels/Ranks/Banks: 4/2/8
	Timing (CPU cycles)	tRCD:44, tCAS:44, tCCD:16, tWTR:31, tWR:4, tRTP:46, tBL:10, tCWD:61, tRP:44, tRRD:16, tRAS:112, tRC:271, tFAW:181
RIME	**Memory Configuration**	Channels/Chips/Banks/Subbanks: 1/8/64/64, 1Gb DDR4-1600 compatible chips, 512x512 SLC subarrays, die area: 20.54mm^2
	Timing and Power	tRead: 4.3ns, tWrite: 54.2ns, tCompute: 282.5ns, vRead: 1V, vWrite: 2V, vCompute: 1V, compute energy/chip: 51.3nJ

B. Circuits

We model the data array, sensing circuits, drivers, mat controller, and interconnect elements using SPICE predictive technology models [47] of NMOS and PMOS transistors at 22nm. To estimate the area, delay, dynamic energy, and leakage power of proposed memristive system, we perform circuit simulations for the building blocks using Cadence (SPECTRE) [48]. Then, we use the resistive memory parameters provided by the prior work [49] to evaluate the read/write/compute voltages, area, delay, and energy of the data arrays. All the additional gates, latches, and the control logic are synthesized using the Cadence Encounter RTL Compiler [50] with FreePDK [51] at 45nm. The results are then scaled down to a 22nm memory technology node. All

[5]The bit location 2^{30} is used to extract the DIMM address.

[6]The chip controller excludes this value from the range.

the SRAM units for the tables and data buffers at the chip controller are evaluated using CACTI 6.5 [52]. To estimate the system power/energy, we use the cycle-accurate simulator in coordination with McPAT [53] for the processor die, Micron power calculator [54] for the main memory, and prior work on HBM memories [55] for the in-package memory architecture.

The overheads associated with the additional circuitry is measured through modeling with CACTI 6.5 [52]. The match vectors incur a 3% area overhead per mat. Including all additional latches, control logic, tree reduction and multiplexers, each mat has an 8% area overhead and 5% die overhead.

C. Workloads

In addition to various sorting kernels (i.e., mergesort, quicksort, radixsort, and heapsort), we develop two versions of six applications for execution on the proposed RIME architecture and the conventional multicore CPU with in-package and off-chip memory systems. All of the workloads are compiled with GCC using the -O3 parameter for the MIPS64 ISA.

GroupBy. Scalar aggregate and GroupBy are two types of aggregates often used to summarize a large set of records for strategic decision making. In particular, GroupBy refers to generating a set of groups for a given table[7] [4], which is a key operator for decision support systems, database, and big data processing [56]. In GroupBy, the whole table is split into several groups depending on a specific key. Then, functions such as filtering, aggregation, and transformation are applied within each group. Finally, the groups are merged or joined to create a new table. Sorting is at the heart of modern large-scale GroupBy functions [38]. We devise a key-value database using quick sort (Q/S) for the GroupBy application to achieve the highest throughput.

MergeJoin. Sort MergeJoin is a key operation in database systems, which refers to combining records from several tables. Numerous proposals have been made to accelerate MergeJoin through parallelism [57] or FPGA accelerators [58]. For the key-value database, we devise a MergeJoin that sorts two large tables to generate a new table that includes only items that exist in both input tables.

Kruskal's and Prim's Algorithms. Minimum spanning tree (MST) is a crucial concept in graph theory. It plays a key role in a broad domain of applications, including vehicular ad-hoc network (VANET) [59], multi-level Steiner tree [60], touring problems, VLSI layout, network organization, and rail transit network [61]. Kruskal's and Prim's algorithms are two main tools for forming MST a given graph. In Kruskal's algorithm, all the graph edges are sorted from low weight to high. Then, the graph edges are iteratively added to the output MST. Prim starts from a vertex and iteratively finds a local vertex with the minimum cost to include in the output MST.

Dijkstra's Algorithm. It finds the shortest paths from a source graph node to all other nodes. The algorithm needs data to be sorted first. This algorithm is very common in network routing protocols [62–65]. We devise a program that iteratively finds a vertex with the minimum distance from the source node. The algorithm is similar to Prim's algorithm. However, Prim provides a minimum spanning tree, but Dijkstra prepares a shortest path tree.

A*-Search Algorithm. A*-search is a smart algorithm for path finding and graph traversal, which is commonly used for finding the shortest path from one point to another in a graph with multiple obstacles. The algorithm plays a significant role in robotics, web-based maps, virtual reality systems, geographic information system, and games [66–68]. We realize a 2D binary matrix representing the obstacles with 0 and non-obstacles with 1. The algorithm is then to find a path from the source to a destination only through non-obstacle paths.

Strict Priority Queue. In the priority queue, data is arranged in descending/ascending order based on their priority. Every dequeue operation results in removing the minimum/maximum entry of the queue based on their values. We use the heap structure for the baseline priority queue application. Numerous algorithms in network for routing and congestion management are based on strict priority queue [69].

From the above workloads, Dijkstra's, Kruskal's, and Prim's algorithms work with IEEE-754 floating-point values, while the rest of the workloads are of type integer. Note that if the dataset uses fixed-point, it is processed by RIME in the fixed point mode; if the dataset uses floating-point, it is processed in the floating-point mode. No data conversion is required.

VII. Evaluations

A. Performance

Sorting. Figure 15 shows the throughput of various sort algorithms, in terms of million keys per second (MKps), using RIME and the baseline systems when the data size varies from 0.5-65M keys. RIME achieves a superior performance over both the baselines for all the evaluated data sizes. As compared with the off-chip baseline, the in-package memory offers a higher memory bandwidth, which results in average throughput gains of 2.4× (M/S), 2.3× (Q/S), 8.1× (R/S), and 1.9× (H/S). In contrast, RIME lowers the bandwidth complexity of sorting via in-situ computation, thereby gaining 30.2× (M/S), 12.4× (Q/S), 50.7× (R/S), and 26× (H/S) average throughputs.

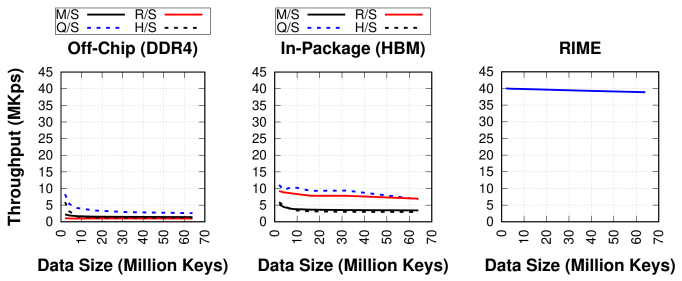


Fig. 15. Throughput of the evaluated sorting algorithms.

Merging. GroupBy and MergeJoin heavily rely on sorting key-value entries. As shown in Figure 16, for the range of evaluated

[7]Whereas, in scalar aggregates, the whole table is grouped and a single value is produced.

data sizes, the HBM implementation of GroupBy achieves $1.1 - 2\times$ better performance than off-chip DRAM. Whereas, RIME improves performance by $5.4 - 23.1\times$. Similarly, the HBM version of MergeJoin performs $1.1 - 2\times$ better than the off-chip DRAM baseline; while, RIME improves performance by $5.6 - 24.1\times$.

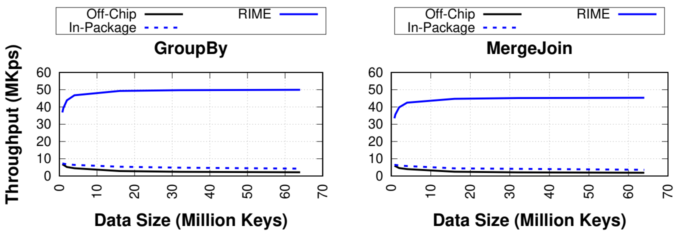


Fig. 16. Throughput of merge and join algorithms for various sizes.

Ranking. Figure 17 shows the throughput of various algorithms based on data ranking. RIME improves performance significantly. For Kruskal, the HBM implementation achieves $2.8 - 3.7\times$ of the off-chip performance; while, RIME gains $8.5 - 20.9\times$. Similarly for Dijkstra, the performance gains over the off-chip baseline are $1.2 - 2.2\times$ and $7.5 - 17.2\times$ for HBM and RIME, respectively. Such performance gains for RIME are enabled due to the significant reduction in memory bandwidth requirements. We observe similar trends in Prim and A*-Search. The performance gains for Prim are $2 - 4.4\times$ and $6.3 - 14.3\times$ for the HBM and RIME systems. As compared to the off-chip system, A*-Search on HBM and RIME respectively achieve the $1 - 1.1\times$ and $2.3 - 23\times$ performance of the off-chip DRAM baseline.

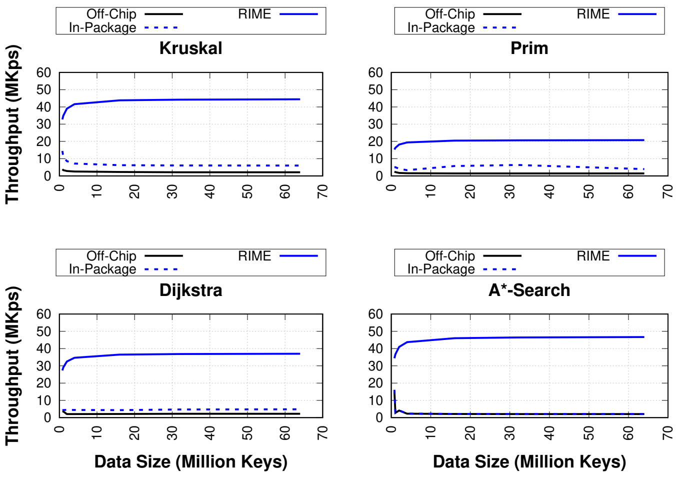


Fig. 17. Throughput of graph algorithms for various sizes.

Across all the evaluated applications, we found the performance of ranking with RIME pretty insensitive to data size. However, when executing an application, multiple ranking operations may be carried out in between frequent RIME initialization and application phases that are sensitive to data size. Therefore, we observe a stagnation in throughput of RIME as data size increases for most of the evaluated workloads.

Strict Priority Queuing. We evaluate the strict priority queuing using a packet processing workload, where two threads are used for adding and removing packets to a buffer. On every remove, a packet with the minimum key value is removed from the queue. To model various loads and packet rates, we assess performance for a range of initial buffer sizes (0.5-65M packets) and various ratios of packet add to remove (i.e., R). Figure 18 shows the throughput of removing packets from the buffer for RIME and the baseline systems. We use a heap structure for the baseline priority queues, which need heap maintenance at both insert and remove operations. Therefore, increasing the buffer size and add-to-remove ratio results in a lower throughput for the baseline HBM and off-chip system. In contrast, RIME achieves a constantly high throughput due to using ordinary memory writes for adding packets to the queue and low complexity accesses for removing packets from the buffer. Across all the evaluated sizes and rate, RIME gains $6.1 - 43.6\times$ better performance than the both HBM and off-chip baselines.

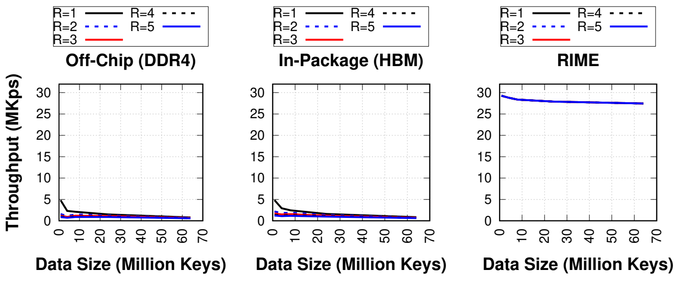


Fig. 18. Throughput of strict priority queue for various packet rates and sizes.

B. Power and Energy

The proposed software library (V) for controlling RIME DIMMs ensure a peak power of 1W for all the evaluated applications. For system energy evaluation, we execute all the evaluated workloads for 65M keys. Figure 19 shows the system energy consumed by the HBM and RIME systems normalized to the off-chip baseline. The HBM system consumes an average of 24% more energy than off-chip for A*-Search and Strict Priority Queues. This is mainly due to (1) similar execution times of the two baselines and (2) the additional static power consumed by the in-package and off-chip memories in the HBM baseline. As for the other applications, HBM can significantly reduce the execution time, thereby decreasing the system energy by about 40%. RIME achieves average energy reductions of 94% (Kruskal), 92% (Dijkstra), 91% (Prim), 95% (GroupBy), 95% (MergeJoin), 94% (A*-Search), and 96% (Strict Priority Queuing).

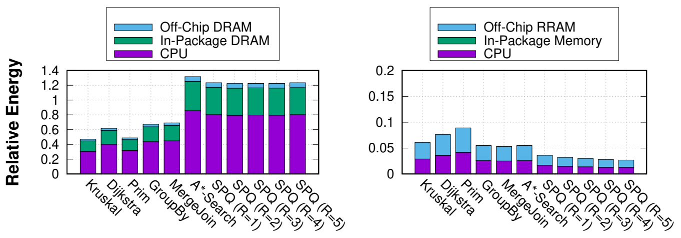


Fig. 19. System energy for various applications (65M keys).

C. Lifetime

RRAM devices can endure a finite number of writes ranging from 10^6 to 10^{12} [70–72]. We assess the impact of this finite endurance on the lifetime of the proposed memristive system. Notice that wear is only induced by writing to the memristive arrays. Therefore, we need to track the number of writes performed per memory locations during the execution of each workload. Notably, unlike the conventional sort algorithms, RIME does not require any data swap during the sort iterations. Therefore, no additional writes to the memristive cells are necessary during a RIME sort operation. Also, it is notable that the initialization and exclusion are performed to the flag bits implemented in CMOS latches. By tracking the total number of writes per second carried out during the execution of all applications, we first identify a block with the highest write frequency. Then, we compute the lifetime assuming that the most frequently written block keeps getting writes at the same rate until it stops working. Based on this study for 10^8 writes, we expect at least 376 years lifetimes for the evaluated applications.

VIII. Related Work

Sorting with SIMD and GPU Accelerators. Software approaches have been proposed to exploit SIMD instructions for utilizing data-level parallelism in sorting applications [11]. Inoue et al. [11] present a sorting algorithm based on a multi-way Mergesort that reduces the cache misses through avoiding random accesses for rearrangement of data. CloudRAMSort performs large-scale distributed sorting by using SIMD and multicore architectures [73]. Hou et al. [13] propose a segmented sort mechanism for load balanced processing on GPUs by combining or splitting data segments of different sizes. Stehle et al. [16] propose a hybrid Radixsort that reduces the amount of memory transfer in GPUs. Satish et al. [74] present an implementation of Radixsort and Mergesort on many-core GPUs by considering fine-grained parallelism and minimal global communication. Several algorithms have been devised to improve the performance of modern databases using parallel multicore processing, SIMD instructions, GPUs, and ASIC [75]. Albutiu et al. [57] develop a parallel sort-merge algorithm to minimize the query response times in databases. Chhugani et al. [15] present a multi-threaded SIMD implementation of Mergesort based on the binary tree to better utilize bandwidth . The existing software solutions rely on improving performance through reducing data movement between the processor and memory. However, they don't solve the fundamental issue, which is the need for accessing data from memory. In contrast, RIME enables in-situ ranking with no need to transfer data from memory to the processor.

Sorting with FPGA and ASIC Accelerators. Kobayashi et al. [76] propose an FPGA-based sorting accelerator that receives data from a host PC through PCIe bus and sends the sorted data back. Bonsai [77] proposes an adaptive merge-tree based sorting solution to optimize FPGA sorting performance across all scales of data (MB to GB). Zhang et al. [78] develop a CPU-FPGA heterogeneous platform for Mergesort. Casper et al. [79] present a hardware design for selection, merge join, and sorting for database applications. Specialized hardware has been also considered for accelerating sort operations [14]. These solutions still require moving data from memory to the accelerator prior to sorting. Whereas, RIME enables both storing and sorting within the same memory arrays with no need for moving data elements.

Ranking Accelerators. Numerous techniques have been proposed in the literature to speedup median computation. Kumar et al. [80] present a hardware implementation for computing the median of 25 integers in three clock cycles at 394 Mhz. Szanto et al. [81] propose a hierarchical histogram based median filter in GPUs for parallel applications. Sindhu et al. [82] design a comparator for fast sorting and ranking data. Venkatappareddy et al. [83] propose a methodology to employ the binary median filter for polynomial expressions. Lin et al. [84] propose a 1D comparison-free bit-level median filter by cascading different median units. Rupesh et al. [85, 86] examine a data clustering accelerator based on in-situ median calculation in RRAM. Unlike the existing solutions, the proposed memory system is capable of accelerating sort and general ranking operations rather only finding the median.

In-Memory Processing. In the literature, numerous accelerators have been proposed for in memory processing that aim at reducing data movement between the processor and memory through performing computation on the memory chips. Computational RAM [87] builds a system, where SIMD pipelines are placed next to the memory arrays for in-memory computation. A similar approach is proposed by Parallel PIM [88] to perform SIMD operations in memory. Active Pages [89] propose a microprocessor that includes additional logic circuit in DRAM chips for in-memory computation. FlexRAM [90] and intelligent RAM (IRAM) [91] are other examples for in-memory processing that have been evaluated on different technologies. However, RIME focuses on fast and efficient in-memory ranking for a different class of applications.

IX. Conclusions

Large scale sorting is a fundamental operation for future data intensive applications. This paper characterized the bandwidth and throughput requirements of large-scale sorting workloads and identified the primary reason for poor performance of sorting. As an effective solution for the bandwidth problem in sorting applications, we examined a novel memory system capable of data ranking in memory. The proposed architecture exhibits significant potentials for orders of magnitude performance and energy-efficiency gains for the future large scale data processing.

X. Acknowledgement

The authors would like to thank the anonymous reviewers for their useful feedback. This work was supported in part by the National Science Foundation (NSF) under Grant CCF-1755874.

References

[1] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[2] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[3] P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, 2015.

[4] D. Taniar, C. H. Leung, W. Rahayu, and S. Goel, *High-performance parallel database processing and grid databases*, vol. 67. John Wiley & Sons, 2008.

[5] E. Kovacs and I. Ignat, "Clustering with prototype entity selection compared with k-means," *Journal of Control Engineering and Applied Informatics*, vol. 9, no. 1, pp. 11–18, 2007.

[6] V. Jugé, "Adaptive shivers sort: An alternative sorting algorithm," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1639–1654, SIAM, 2020.

[7] M. D. MacLaren, "Internal sorting by radix plus sifting," *Journal of the ACM (JACM)*, vol. 13, no. 3, pp. 404–411, 1966.

[8] H. H. Goldstine, J. Von Neumann, and J. Von Neumann, "Planning and coding of problems for an electronic computing instrument," 1947.

[9] C. A. R. Hoare, "Algorithm 64: quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, 1961.

[10] R. Bordawekar, D. Brand, M. Cho, B. R. Konigsburg, and R. Puri, "Radix sort acceleration using custom asic," May 24 2018. US Patent App. 15/857,770.

[11] H. Inoue and K. Taura, "Simd-and cache-friendly algorithm for sorting an array of structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1274–1285, 2015.

[12] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 351–362, 2010.

[13] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on gpus," in *Proceedings of the International Conference on Supercomputing*, pp. 1–10, 2017.

[14] S. Haas, S. Scholze, S. Höppner, A. Ungethüm, C. Mayr, R. Schüffny, W. Lehner, and G. Fettweis, "Application-specific architectures for energy-efficient database query processing and optimization," *Microprocessors and Microsystems*, vol. 55, pp. 119–130, 2017.

[15] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core simd cpu architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[16] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 417–432, 2017.

[17] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle timesten: An in-memory database for enterprise applications.," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, 2013.

[18] M. P. (Intel), *An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing*. Intel, January 2016. https://software.intel.com/en-us/blogs/2016/01/20/an-intro-to-mcdram-high-bandwidth-memory-on-knights-landing.

[19] P. Behnam and M. N. Bojnordi, "Redcache: reduced dram caching," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[20] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[21] P. Behnam, A. P. Chowdhury, and M. N. Bojnordi, "R-cache: A highly set-associative in-package cache using memristive arrays," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pp. 423–430, IEEE, 2018.

[22] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, pp. 1–16, 2020.

[23] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–13, IEEE, 2016.

[24] T. P. Xiao, C. H. Bennett, B. Feinberg, S. Agarwal, and M. J. Marinella, "Analog architectures for neural network acceleration based on non-volatile memory," *Applied Physics Reviews*, vol. 7, no. 3, p. 031301, 2020.

[25] A. Pal Chowdhury, P. Kulkarni, and M. Nazm Bojnordi, "Mb-cnn: memristive binary convolutional neural networks for embedded mobile devices," *Journal of Low Power Electronics and Applications*, vol. 8, no. 4, p. 38, 2018.

[26] M. N. Bojnordi and E. Ipek, "The memristive boltzmann machines," *IEEE Micro*, vol. 37, no. 3, pp. 22–29, 2017.

[27] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.

[28] G. Graefe, "Implementing sorting in database systems," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, pp. 10–es, 2006.

[29] Z. Wang, L. Tian, D. Guo, and X. Jiang, "Optimization and analysis of large scale data sorting algorithm based on hadoop," *arXiv preprint arXiv:1506.00449*, 2015.

[30] "sorting applications." https://algs4.cs.princeton.edu/25applications/. Accessed: 2020-03-12.

[31] L. Z. Xiang, "Research and improvement of pagerank sort algorithm based on retrieval results," in *2014 7th International Conference on Intelligent Computation Technology and Automation*, pp. 468–471, IEEE, 2014.

[32] C. Kim, S. Yoon, and D. Kim, "Fast sort of floating-point data for data engineering," *Advances in Engineering Software*, vol. 42, no. 1-2, pp. 50–54, 2011.

[33] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, p. 321, July 1961.

[34] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[36] "Jedec standard: Ddr4 sdram," *JEDEC Solid State Technology Association*, 2012.

[37] S. JEDEC, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[38] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox-hbm: Stream analytics on high bandwidth hybrid memory," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 167–181, 2019.

[39] M. Zangeneh and A. Joshi, "Design and optimization of nonvolatile multibit 1t1r resistive ram," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 8, pp. 1815–1828, 2014.

[40] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, IEEE, 2015.

[41] P.-E. Danielsson, "Getting the median faster," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 71–78, 1981.

[42] I. Hatirnaz, F. Gurkaynak, and Y. Leblebici, "Realization of a programmable rank-order filter architecture using capacitive threshold logic gates," in *ISCAS'99. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI (Cat. No. 99CH36349)*, vol. 1, pp. 435–438, IEEE, 1999.

[43] D. H. Yoon and F. Petrini, "Hourglass: A bandwidth-driven performance model for sorting algorithms," in *Supercomputing* (J. M. Kunkel, T. Ludwig, and H. W. Meuer, eds.), (Cham), pp. 93–108, Springer International Publishing, 2014.

[44] "Ddr4 sdram registered dimm design specification," *JEDEC Solid State Technology Association*, 2014.

[45] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 448–459, IEEE, 2013.

[46] "Memkind."

[47] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm design exploration," in *International Symposium on Quality Electronic Design*, 2006.

[48] "Spectre circuit simulator." http://www.cadence.com/products/cic/

spectre_circuit/pages/default.aspx.

[49] M. Wu, Y. Lin, W. Jang, C. Lin, and T. Tseng, “Low-power and highly reliable multilevel operation in ZrO_2 1t1r rram,” *IEEE Electron Device Letters*, vol. 32, no. 8, pp. 1026–1028, 2011.

[50] “Encounter RTL compiler.” http://www.cadence.com/products/ld/rtl_compiler/.

[51] “Free PDK 45nm open-access based PDK for the 45nm technology node.” http://www.eda.ncsu.edu/wiki/FreePDK.

[52] S. Wilton and N. Jouppi, “CACTI: An enhanced cache access and cycle time model,” vol. 31, pp. 677–688, May 1996.

[53] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *International Symposium on Computer Architecture*, 2009.

[54] “Micron ddr4 power calculator.” https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm.

[55] M. O’Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, “Fine-grained dram: Energy-efficient dram for extreme bandwidth systems,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 ’17, (New York, NY, USA), pp. 41–54, ACM, 2017.

[56] S. Chaudhuri and K. Shim, “Including group-by in query optimization,” in *VLDB*, vol. 94, pp. 354–366, 1994.

[57] M.-C. Albutiu, A. Kemper, and T. Neumann, “Massively parallel sort-merge joins in main memory multi-core database systems,” *arXiv preprint arXiv:1207.0145*, 2012.

[58] M.-T. Xue, Q.-J. Xing, C. Feng, F. Yu, and Z.-G. Ma, “Fpga-accelerated hash join operation for relational databases,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2019.

[59] J. J. Kponyo, Y. Kuang, E. Zhang, and K. Domenic, “Vanet cluster-on-demand minimum spanning tree (mst) prim clustering algorithm,” in *2013 International Conference on Computational Problem-Solving (ICCP)*, pp. 101–104, IEEE, 2013.

[60] R. Ahmed, F. D. Sahneh, S. Kobourov, and R. Spence, “Kruskal-based approximation algorithm for the multi-level steiner tree problem,” *arXiv preprint arXiv:2002.06421*, 2020.

[61] T. Liang, H. Liu, and Y. Tan, “Research on the gravity planning model of prefecture city rail transit network,” in *E3S Web of Conferences*, vol. 145, p. 02005, EDP Sciences, 2020.

[62] B. Musznicki, M. Tomczak, and P. Zwierzykowski, “Dijkstra-based localized multicast routing in wireless sensor networks,” in *2012 8th International Symposium on Communication Systems, Networks & Digital Signal Processing (CSNDSP)*, pp. 1–6, IEEE, 2012.

[63] I. Koutsopoulos, E. Noutsi, and G. Iosifidis, “Dijkstra goes social: Social-graph-assisted routing in next generation wireless networks,” in *European Wireless 2014; 20th European Wireless Conference*, pp. 1–7, VDE, 2014.

[64] F. Yue-zhen, L. Dun-min, W. Qing-chun, and J. Fa-chao, “An improved dijkstra algorithm used on vehicle optimization route planning,” in *2010 2nd international conference on computer engineering and technology*, 2010.

[65] C. Liu, Y. Li, W. Cheng, and G. Shi, “An improved multi-channel aodv routing protocol based on dijkstra algorithm,” in *2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pp. 547–551, IEEE, 2019.

[66] S. Bandi and D. Thalmann, “The use of space discretization for autonomous virtual humans (video session),” in *Proceedings of the second international conference on Autonomous agents*, pp. 336–337, 1998.

[67] J. Yao, C. Lin, X. Xie, A. J. Wang, and C.-C. Hung, “Path planning for virtual human motion using improved a* star algorithm,” in *2010 Seventh international conference on information technology: new generations*, pp. 1154–1158, IEEE, 2010.

[68] B. M. ElHalawany, H. M. Abdel-Kader, A. TagEldeen, A. E. Elsayed, and Z. B. Nossair, “Modified a* algorithm for safer mobile robot navigation,” in *2013 5th International Conference on Modelling, Identification and Control (ICMIC)*, pp. 74–78, IEEE, 2013.

[69] D. Medhi and K. Ramasamy, *Network routing: algorithms, protocols, and architectures*. Morgan Kaufmann, 2017.

[70] C. Cheng, A. Chin, and F. Yeh, “Novel ultra-low power rram with good endurance and retention,” in *VLSI Technology (VLSIT), 2010 Symposium on*, pp. 85–86, June 2010.

[71] H. Akinaga and H. Shima, “Resistive random access memory (reram) based on metal oxides,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[72] C.-W. Hsu, I.-T. Wang, C.-L. Lo, M.-C. Chiang, W.-Y. Jang, C.-H. Lin, and T.-H. Hou, “Self-rectifying bipolar tao x/tio 2 rram with superior endurance over 10 12 cycles for 3d high-density storage-class memory,” in *VLSI Technology (VLSIT), 2013 Symposium on*, pp. T166–T167, IEEE, 2013.

[73] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, “Cloudramsort: fast and efficient large-scale distributed ram sort on shared-nothing cluster,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 841–850, 2012.

[74] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, IEEE, 2009.

[75] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, “In-memory big data management and processing: A survey,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

[76] R. Kobayashi and K. Kise, “A high performance fpga-based sorting accelerator with a data compression mechanism,” *IEICE Transactions on Information and Systems*, vol. 100, no. 5, pp. 1003–1015, 2017.

[77] N. Samardzic, W. Qiao, V. Aggarwal, M. F. Chang, and J. Cong, “Bonsai: High-performance adaptive merge tree sorting,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 282–294, 2020.

[78] C. Zhang, R. Chen, and V. Prasanna, “High throughput large scale sorting on a cpu-fpga heterogeneous platform,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 148–155, IEEE, 2016.

[79] J. Casper and K. Olukotun, “Hardware acceleration of database operations,” in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 151–160, 2014.

[80] V. Kumar, A. Asati, and A. Gupta, “Low-latency median filter core for hardware implementation of $5\times$ 5 median filtering,” *IET Image Processing*, vol. 11, no. 10, pp. 927–934, 2017.

[81] P. Szántó and B. Fehér, “Hierarchical histogram-based median filter for gpus,” *Acta Polytechnica Hungarica*, vol. 15, no. 2, 2018.

[82] E. Sindhu and K. Vasanth, “Vlsi architectures for 8 bit data comparators for rank ordering image applications,” in *2019 International Conference on Communication and Signal Processing (ICCSP)*, pp. 0087–0093, IEEE, 2019.

[83] P. Venkatappareddy, B. Lall, C. Jayanth, K. Dinesh, and M. Deepthi, “Novel methods for implementation of efficient median filter,” in *2017 14th IEEE India Council International Conference (INDICON)*, pp. 1–5, IEEE, 2017.

[84] C. Lin, W.-T. Chen, Y.-C. Chou, and P.-Y. Chen, “A novel comparison-free 1d median filter,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2019.

[85] Y. K. Rupesh, P. Behnam, G. R. Pandla, M. Miryala, and M. N. Bojnordi, “Accelerating k-medians clustering using a novel 4t-4r rram cell,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 12, pp. 2709–2722, 2018.

[86] Y. K. Rupesh and M. N. Bojnordi, “Large scale data clustering using memristive k-median computation,” in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 374–374, IEEE, 2017.

[87] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, “Computational ram: implementing processors in memory,” *IEEE Design Test of Computers*, vol. 16, pp. 32–41, Jan 1999.

[88] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: the terasys massively parallel pim array,” *Computer*, vol. 28, pp. 23–31, Apr 1995.

[89] M. Oskin, F. T. Chong, and T. Sherwood, “Active pages: a computation model for intelligent memory,” in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pp. 192–203, Jun 1998.

[90] Y. Kang, W. Huang, S. M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, “Flexram: Toward an advanced intelligent memory system,” in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 5–14, Sept 2012.

[91] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent ram,” *IEEE Micro*, vol. 17, pp. 34–44, Mar. 1997.