Sorting it out in Hardware: A State-of-the-Art Survey

AMIR JALILVAND, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States
FAEZE S. BANITABA, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States
S. NEWSHA ESTIRI, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States
SERCAN AYGUN, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States
M. HASSAN NAJAFI, Department of Electrical, Computer, and Systems Engineering, Case Western Reserve University, Cleveland, United States

Sorting is a fundamental operation in various applications and a traditional research topic in computer science. Improving the performance of sorting operations can have a significant impact on many application domains. Much attention has been paid to hardware-based solutions for high-performance sorting. These are often realized with application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs). Recently, in-memory sorting solutions have also been proposed to address the movement cost issue between memory and processing units, also known as the Von Neumann bottleneck. Due to the complexity of the sorting algorithms, achieving an efficient hardware implementation for sorting data is challenging. A large body of prior solutions is built on compare-and-swap (CAS) units. These are categorized as *comparison-based* sorting. Some recent solutions offer *comparison-free* sorting. In this survey, we review the latest works in the area of hardware-based sorting. We also discuss the recent hardware solutions for *partial* and *stream* sorting. Finally, we discuss some important concerns that need to be considered in the future designs of sorting systems.

CCS Concepts: • **Hardware** → *Integrated circuits*; *Hardware accelerators*; • **Theory of computation** → **Sorting and searching**; • **General and reference** → **Surveys and overviews**; • **Computer systems organization** → *Processors and memory architectures*; *Embedded hardware;*

Additional Key Words and Phrases: Comparison-based sorting, comparison-free sorting, hardware-based sorting, in-memory sorting, partial sorting

This work is supported in part by the National Science Foundation under grants #2019511, #2339701, and a generous gift from NVIDIA.

Authors' Contact Information: Amir Jalilvand, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States; e-mail: mr.amirhosseinjalilvand@gmail.com; Faeze S. Banitaba, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States; e-mail: faeze.banitaba@louisiana.edu; S. Newsha Estiri, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States; e-mail: seyedehnewsha.estiri1@louisiana.edu; Sercan Aygun, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, United States; e-mail: sercan.aygun@louisiana.edu; M. Hassan Najafi, Department of Electrical, Computer, and Systems Engineering, Case Western Reserve University, Cleveland, Ohio, United States; e-mail: mhassan.najafi@case.edu.

ACM 1084-4309/2025/06-ART52
https://doi.org/10.1145/3734797

ACM Reference Format:
Amir Jalilvand, Faeze S. Banitaba, S. Newsha Estiri, Sercan Aygun, and M. Hassan Najafi. 2025. Sorting it out in Hardware: A State-of-the-Art Survey. *ACM Trans. Des. Autom. Electron. Syst.* 30, 4, Article 52 (June 2025), 31 pages. https://doi.org/10.1145/3734797

1 Introduction

Today, the data volume has increased significantly in many application domains. Processing data at the terabyte and petabyte levels has become routine. Processing large volumes of data is challenging and is expected to remain at an upward rate [27]. Sorting is one of the substantial operations in computer science performed for different purposes, from putting data in a specific order, such as *ascending* or *descending*, to finding the minimum and maximum values, finding the median, and partial sorting to find the top-m greatest or smallest values. As Figure 1 shows, sorting is used in many application domains, from data merging to big data processing [52, 64], database operations [25], especially when the scale of files/data is very large, robotics [24, 81], signal processing (e.g., sorting radar signals) [35, 51], and wireless networks [38].

Sorting time-series data according to their timestamps holds critical importance in numerous **artificial intelligence** (**AI**) applications, such as forecasting and anomaly detection [1], where the sequential occurrence of events is of paramount significance [103]. Wireless sensor network applications often incorporate genetic algorithms, with the "**Non-dominated Sorting Genetic Algorithm** (**NSGA**)" being a commonly employed and efficient approach requiring sorting [23]. Additionally, wireless networks necessitate the implementation of sorting algorithms that are both energy-optimal and energy-balanced, such as enhanced sorting algorithms [84]. The concept of sorting also extends to robotic visual tasks. Much like traditional scalar sorting, sorting items based on attributes like color, shape, or other features within a robot's perceived environment constitutes a tangible engineering application of sorting [77]. In the field of robotics, object sorting is a significant task. Particularly in computer vision applications, sorting objects by robots based on their perceived environment is challenging [11]. Another intriguing application is to control greenhouse climatic factors through sorting networks [72]. Some researchers adopt an external sorting methodology to sort large-scale datasets. External sorting serves as a solution for sorting vast datasets that cannot fit into the primary memory of a computing platform. Instead, it utilizes additional memory elements like hard disk drives, employing a sort-and-merge strategy [17]. Sorting has also found novel applications in signal processing. This extends from theoretical scalar sorting to sorting tasks in real-world signal processing. An illustrative example is radar signal sorting, a recent and intricate sorting challenge in the context of multi-function radar systems [20].

Improving the sorting speed can have a significant impact on all these applications. Many software- and hardware-based solutions have been proposed in the literature for high-performance sorting. Software-based solutions rely on powerful single/multi-core and **graphics processing unit** (**GPU**)-based processors for high performance [83]. Much attention has been paid to hardware sorting solutions, especially for applications that require very high-speed sorting [28, 54, 60]. These have been implemented using either **application-specific integrated circuits** (**ASICs**) or **field-programmable gate arrays** (**FPGAs**). Depending on the target applications, the hardware sorting units vary greatly in how they are configured and implemented. The number of inputs can be as low as nine for some image processing applications (e.g., median filtering [26], compression [100]) to tens of thousands [28, 60]. The data inputs have been binary values, integers, or floating-point numbers ranging from 4- to 256-bit precision.

Hardware cost and power consumption are the dominant concerns with hardware implementations. The total chip area is limited in many applications [19]. As fabrication technologies continue

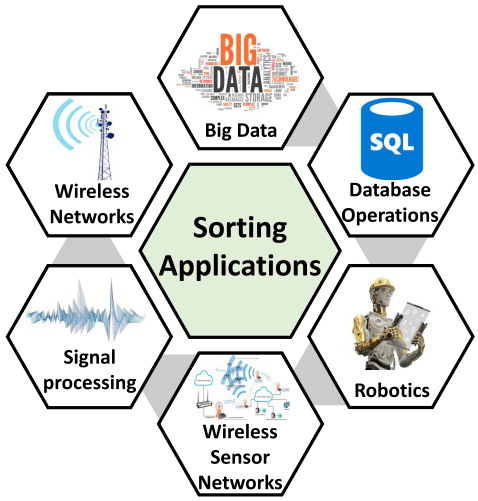
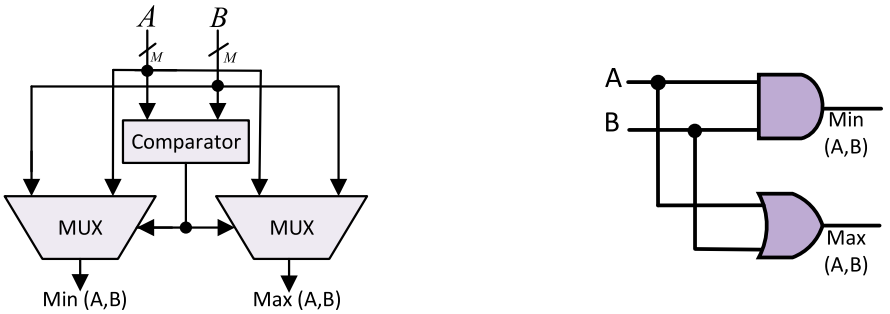


Fig. 1. Common applications of sorting: *Big Data* [16, 56, 63], *Database operations* [6, 33, 89], *Robotics* [11, 34, 77, 78], *Wireless Sensor Networks* [10, 55, 82, 84, 88], *Signal Processing* [20, 43, 51, 79, 93], and *Wireless Networks* [80].



(a) (b)

Fig. 2. The hardware designs for CAS operation in [60]. (a) Non-unary *CAS* operation, (b) Unary *CAS* block.

to scale, keeping chip temperatures low is an important goal since leakage current increases exponentially with temperature. Power consumption must be kept as low as possible. Developing low-cost, power-efficient hardware-based solutions to sorting is an important goal [60].

There is a large body of work on the design of customized sorting hardware. These works seek to utilize the hardware resources fully and to provide a custom, cost-effective hardware sorting engine. Developing hardware-efficient implementations for sorting algorithms is challenging, considering the complexity of these algorithms [2, 28]. A significant amount of hardware resources is spent on comparators and memory elements including large global memories, complex pipelining, and complicated local and global control units [2].

Many of the prior hardware solutions are built on basic **compare-and-swap** (**CAS**) units that compare pairs of data and swap if needed. These solutions are categorized as *comparison-based* sorting. As shown in Figure 2(a), each basic CAS unit is conventionally implemented with a binary

Table 1. Comparison between the Existing Surveys for Hardware Solutions of Sorting

Article	Year	Sorting Networks	Comparison-based Sorting	Comparison-free Sorting	Partial Sorting	In-memory Sorting
Jmaa et al. [8]	2019	❍	●	❍	❍	❍
Skliarova [85]	2021	●	●	❍	❍	❍
Ali [4]	2022	◗	◗	◗	❍	❍
This work		●	●	●	●	●

❍ → not covered, ◗ → partially covered, ● → fully covered

comparator and two **multiplexer (MUX)** units [60]. Sorting networks of CAS units are frequently used for fast and parallel hardware sorting. Their inherent parallelism enables them to achieve sorting at a considerably faster rate than the fastest sequential software-based sorting algorithms. However, CAS-based hardware solutions can exhibit high hardware costs in certain scenarios, such as when implementing specific designs like bitonic sort. While these solutions are highly efficient for some applications, they may come with tradeoffs, particularly in hardware complexity. Additionally, alternative approaches can present other challenges, such as lower throughput, highlighting the importance of selecting the appropriate solution based on the application requirements [73].

In the last few years, some *comparison-free/quasi-comparison-free* sorting solutions have been proposed to address the challenges with *comparison-based* sorting designs. We will discuss these novel solutions in Section 2.2.

Complete sorting sorts all items (N) of a list. *Partial* sorting has also been a popular sorting variant. Unlike complete sorting, partial sorting returns a list of the M smallest or largest elements in order where $M < N$ [28, 74]. The cost of partial sorting is often substantially less than complete sorting when the number of sub-sorting attempts is small compared to N. The other elements (above the M smallest ones) may also be sorted as in-place partial sorting or discarded, which is common in streaming partial sorts [15].

Despite many recent works in hardware-assisted sorting, no recent survey reviews the latest developments in this area. Studying the literature, we found three surveys discussing prior hardware-based sorting designs. These are compared in Table 1. Jmaa et al. [8] compare the performance of the hardware implementations of popular sorting algorithms (i.e., `Bubble Sort`, `Insertion Sort`, `Selection Sort`, `Quick Sort`, `Heap Sort`, `Shell Sort`, `Merge Sort`, and `Tim Sort`) in terms of execution time, standard deviation, and resource utilization. They synthesized the designs on a Zynq-7000 FPGA platform. Skliarova [85] reviewed different implementation approaches for network-based hardware accelerators for *sorting*, *searching*, and *counting* tasks. Ali [4] looked closely at comparison-based and comparison-free hardware solutions for sorting. As in-memory and partial sorting are relatively emerging topics, these previous surveys do not cover them. Motivated by this, this work reviews the latest hardware solutions for complete, partial and in-memory sorting, covering both comparison-based and comparison-free approaches. Table 2 summarizes and classifies the important works we study in this article.

To enhance the comparison of state-of-the-art hardware-based sorting solutions, we have organized them into categories based on their features. We first reviewed the complete sorting methods using comparison-based and comparison-free approaches; these are more on sorting techniques, like partial sorting, in the following section. While the platforms such as FPGA, ASIC, GPU, CPU, and in-memory designs are discussed in these sections, there will be a separate section to highlight the emerging in-memory computing, a promising direction in addition to available conventional platforms. Figure 3 navigates the reader through this overall sectioning. Section 2 focuses on complete sorting solutions, which may involve comparing elements or employing comparison-free

Table 2. Prior Art for Hardware Solutions of Sorting

	Reference	Year	Idea	Design
Comparison Based	Farmahini et al.[28]	2013	Sorting using hierarchical smaller blocks	♠
	Lin et al.[49]	2017	Pointer-like iterative architecture	♠
	Najafi et al.[60]	2018	Bit-stream-based and time-encoded unary design	♠
	Norollah et al.[62]	2019	Consecutive normal and reverse sorting	✣
	Jelodari et al.[41]	2020	Vertex indexing in graph representation of inputs	✣
	Papaphilippou et al. [65, 66]	2020	Recursive parallel merge tree	✣
	Preethi et al.[69]	2021	Clock gating techniques to improve power consumption	♠
	Prince et al.[70]	2023	Sorting weighted stochastic bit-streams	♠
Comparison Free	Abdel-Hafeez et al. [2]	2017	One-hot weight representation	♠
	Bhargav et al. [9]	2019	FSM module for sorting	♠
	Chen et al.[18]	2021	Bidirectional architecture improving sorting cycle [9]	♠ / ✣
	Sri et al.[86]	2022	Improving the boundary-finding module of [18]	✣
	Ray et al. [73]	2022	Parallel Comparison-free Hardware Sorter	✣
	Jalilvand et al.[39]	2022	Comparison-free sorting based on unary computing	♠
Partial Sorting	Yu et al. [97]	2011	Spike sorting hardware	✣
	Campobello et al. [14]	2012	Maximum - minimum finder	✣
	Subramaniam et al. [87]	2017	Median finder	✣
	Korat et al. [42]	2017	Odd-even merge sorting	✣
	Valencia et al. [91]	2019	Minimum distance calculation for spike sorting	✣
	Zhang et al. [101]	2021	Min-max sorting architecture	✷
	Yan et al. [95]	2021	Determining a certain M largest or smallest numbers	✣
In-Memory	Wu et al.[94]	2015	Data sorting in flash memory (NAND flash-based)	❖
	Samardzic et al. [75]	2020	Bonsai Sorting on CPU-FPGA by DRAM-scale sorting	✣ / ❖
	Li et al. [48]	2020	Parallel sorting via hybrid memory cubes	▲
	Prasad et al. [68]	2021	Memristor-based data ranking and min/max computation	✦
	Chu et al. [22]	2021	Detecting partially ordered for cost reduction	✪
	Riahi Alam et al. [3]	2022	High-performance and energy-efficient data sorting	★
	Yu et al. [98]	2022	Column-skipping algorithm for memristive memory	♣
	Zokaee et al.[104]	2022	Sorting large datasets based on sample sort	✦
	Lenjani et al. [46]	2022	Optimizing external sorting for NVM-DRAM hybrid storage	✦
	Liu et al. [50]	2023	Minimizing NVM write operations	❖
	Gao et al. [29]	2024	Uniform recursive structure	✦

♠→ ASIC, ✣→ FPGA, ✷→ DRAM, ✦→ In-Memory, ♣→ Memristor-based, ★→ Adaptive Memristor, ✪→ NVM-Based, ▲→ In-Logic-Layer Based, ❖→ In-Memory Friendly.

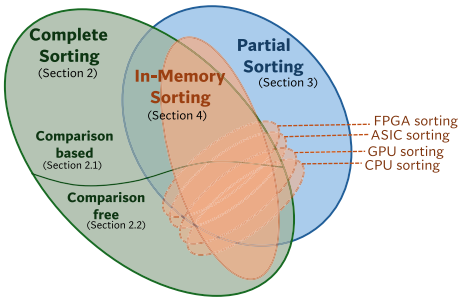


Fig. 3. Three main sections of this survey: *Complete* and *Partial Sorting* for the techniques of sorting, and *In-Memory Sorting* for platform-aware sorting.

methods. Section 3 discusses partial sorting techniques, and Section 4 explores in-memory sorting solutions. While these categories can have high overlaps, we organized them in a manner to provide readers with a clear overview of the topic. Section 5 discusses open challenges and future works. Finally, Section 6 concludes the article.

Table 3. Comparative Analysis of Sorting Architectures (Discussed in Section 2.1), Highlighting Key Experimental Metrics Such as Hardware, Complexity, Latency, Energy Efficiency, Scalability, and Optimization Focus

Ref.	Hardware	Time Complexity	Space Complexity	Latency	Energy / Power	Dataset / Application
[28]	65-nm	$O(\log N \log M)$	$O(N \log^2 M)$	17.64 ns for 16 to 4 max selection unit	N/A	M largest/smallest elements with no order
[49]	90-nm	$O(N \log N)$	$O(\log^2 M)$	1/2 to 1/3 compared to conventional methods	13.60 mW for 32-bit 16-to-4 unit	Real-time processing
[60]	45-nm	$O(N \log^2 N)$	Up to 91% reduction in hardware cost for large networks	100× increase due to unary processing, mitigated with time-encoding to process >1000 HD images/sec	Up to 92% area and power savings for large networks	Sorting for IoT applications, image processing (median filtering)
[62]	FPGA (Virtex7)	O(N)	$O(N)$	$\approx 75ns$ for $N = 64$, bit-width = 64	$\approx 5W$ for $N = 64$, bit-width = 64	Real-time and big-data processing
[41]	FPGA (Cyclone IV)	$O(1)$ for small number of inputs	$O(N^2)$	2 cycles for $N = 8$	N/A	Real-time processing
[65]	FPGA (MPSoc)	$O(N/p)$	$O(N)$	N/A	N/A	Large Datasets
[69]	90-nm	$O(N^2)$	N/A	N/A	Dynamic power reduced by 47.5% at 50 MHz	Sorting for data centers and IoT applications
[70]	45-nm	$O(N)$	$O(N)$	50% latency reduction due to Lock-and-Swap (LAS) units	3.8%–93% energy reduction compared to prior designs	Sorting networks for large-scale applications

2 Complete Sorting Methods

Complete sorting involves arranging all elements of a dataset in a specific order, such as ascending or descending. This is particularly valuable in applications where the entire dataset must be organized, such as generating business intelligence reports, performing statistical analyses, or optimizing database indexing for efficient query performance. In machine learning, sorting features or training datasets can enhance algorithm efficiency, which is especially critical for on-edge learning applications requiring dedicated sorting hardware. Larger systems, including financial platforms, e-commerce operations, and inventory management systems, also rely on complete sorting to enable accurate ranking, reporting, and decision-making. These processes are further enhanced by implementing efficient sorting in hardware, such as GPUs or powerful FPGAs, in collaboration with server systems.

We begin by reviewing recent works on complete sorting, which processes all the data to sort them in an ascending or descending order. We divide our discussion into two categories: *comparison-based* and *comparison-free* sorting.

2.1 Comparison-based

Table 3 provides a detailed comparison of the comparison-based sorting architectures, highlighting their advantages and tradeoffs. Farmahini et al. [28] proposed a comparison-based design that employs efficient techniques for constructing high-throughput, low-latency sorting units using smaller building blocks in a hierarchical manner. Their design includes N-to-M *sorting* and *max-set-selection* units. They extensively discuss the structure, performance, and resource requirements of these units. Despite its primary focus on integer numbers, their design efficiently accommodates two's complement and floating-point numbers, as the comparators utilized in their **compare-and-exchange** (**CAE**) blocks can be substituted accordingly.

Some sorting applications do not need to sort all input data. Instead, the application may only require the identification of the M largest or M smallest values from a set of N inputs. These algorithms are called *partial sorters* and will be discussed later in this survey. In an N-to-M max-set-selection unit used in the sorting designs of [28], only the M largest inputs are required in no specific order.

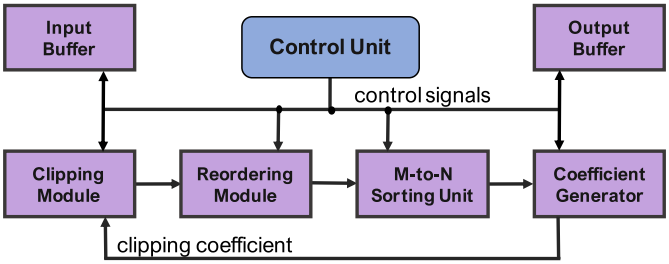


Fig. 4. Lin et al. iterative sorting system. An iterative architecture is designed by repeatedly employing a smaller sorting unit to process streaming input data. In addition to the application of the low-power sorting module, the design also incorporates an adaptive clipping mechanism and a reordering module. Source: [49].

Lin et al. [49] proposed a hardware acceleration architecture for real-time sorting of M out of N inputs. Their design benefits from moving indexes instead of data, called a *pointer-like* design. They reduce power consumption by reducing switching activities and signal transitions while maintaining high throughput. Their sorting approach has a time complexity of $O(N \log^2 N)$ for the iterative design (we assume the input buffer size is the same as input size, N). The primary contributor to power consumption is the switching activities of registers. To effectively reduce power, they recommend modification to the **register transfer level** (**RTL**) design. Notably, signal transitions increase when the input dataset is larger or when the bit width of the input sample is significant. They propose incorporating additional registers to represent the position of each input sample. So, only the indexes need to be migrated from register to register. When N inputs are present, the complete index can be represented using only $\log_2 N$ bits, irrespective of whether the bit-widths are 8-bit, 16-bit, or more. While modifications may increase the total cell area, they substantially reduce dynamic power dissipation.

Executing the sorting process using a single module is impractical for large input datasets, as it requires high I/O bandwidth and a large cell area. To mitigate this issue, Lin et al. [49] proposed reusing smaller sorting units as the core module and combining these small units with other control units to implement an iterative architecture. Figure 4 shows their proposed architecture. Users have the flexibility to select different sorting units as the core module, enabling them to tradeoff throughput for resource constraints.

Najafi et al. [59] developed an area- and power-efficient hardware design for complete sorting based on ***unary* computing** (**UC**). They convert the data from binary to unary bit-streams to sort them in the unary domain. Their approach replaces the conventional complex design of the *CAS* unit implemented based on binary radix with a simple unary-based design made of simple standard AND and OR gates. Figure 2(b) demonstrates how a *CAS* block is implemented in the unary domain. When two unary bit-streams of equal length are connected to the inputs, an AND gate yields the minimum value, whereas an OR gate produces the maximum value. An overhead of this unary design is the cost of converting data from binary to unary representation. However, this conversion overhead is insignificant compared to the cost savings in the computation circuits. They report an area and power saving of more than 90% for implementing a 256-input complete sorting network. The unary design of [60] consists of simple logic gates independent of data size. The length of bit-streams controls the computation accuracy. The longer the bit-stream, the higher the accuracy. However, processing long, unary bit-streams can significantly increase latency with the sorting approach outlined in [60]. This causes runtime overhead compared to the conventional binary process. While the latency may be tolerated in many applications, they introduce a time-based, unary design to mitigate the latency issue. They encode the input data to pulse-width modulated

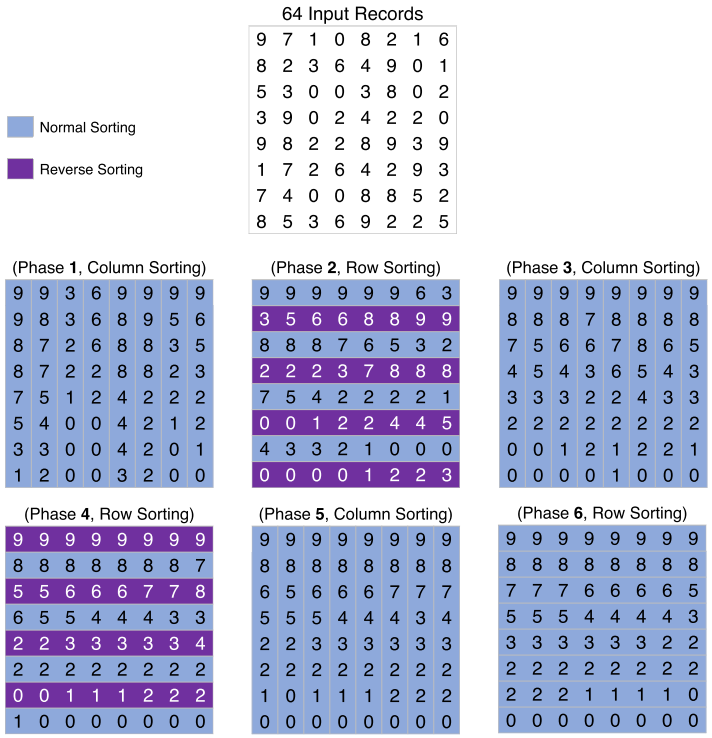


Fig. 5. The MDSA with 64 input records, forming an 8×8 matrix. Source: [62].

signals. The data value is determined by the duty cycle in this approach. The time-based approach significantly reduces the latency at the cost of slight accuracy loss.

Prince et al. [70] combined the bit-stream capabilities of **stochastic computing** (**SC**) with binary weighting, reducing the latency of bit-stream-based sorting. The approach offers good scalability and cost-efficiency compared to SC and traditional binary methods, making it an efficient solution for sorting tasks. They use a weighted bit-stream converter to generate weighted bit-streams for an adaptable sorting network. Unlike conventional SC bit-streams, each bit in the weighted bit-streams retains its weight as a standard binary value. This conversion reduces the number of bits in SC from 2^N to N for N-bit precision, resulting in a substantial reduction in latency and energy consumption by shifting from exponential to linear representation. They propose a new **lock-and-swap** (**LAS**) unit to sort weighted bit-streams. Their LAS-based sorting network can determine the result of comparing different input values early and then map the inputs to the corresponding outputs based on shorter weighted bit-streams.

Norollah et al. [62] presented a novel **multidimensional sorting algorithm** (**MDSA**) and its corresponding architecture, a **real-time hardware sorter** (**RTHS**), to efficiently sort large sequences of records. MDSA reduces the required resources, enhances memory efficiency, and has a minimal negative impact on execution time, even when the number of input records increases. MDSA divides large sequences of records into smaller segments, which are then sorted separately. As shown in Figure 5, the MDSA algorithm consists of six consecutive phases and two modes: normal and reverse sorting. The sorting network organizes the records in descending and ascending order for normal and reverse modes, respectively. In each phase, the sorting networks are fed by a group of input records to sort independently.

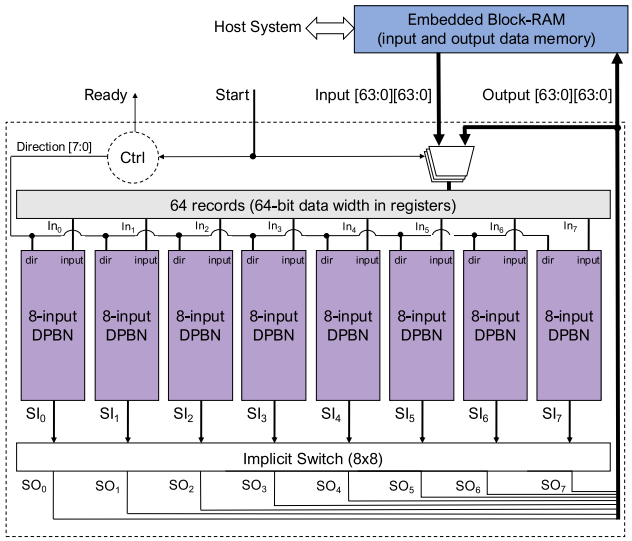


Fig. 6. RTHS architecture for 8×8 matrix records. Source: [62].

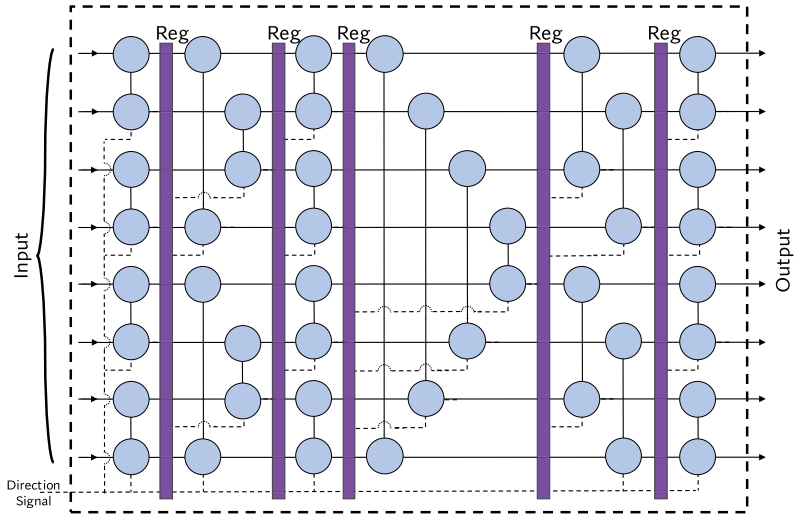


Fig. 7. DPBN unit for 8 inputs. Input and Output have the size of [7:0][63:0], and The direction signal indicates the mode for sorting: normal or reverse. Source: [62].

The authors in [62] claim that their sorting method is more beneficial for resource conservation (memory efficiency) while providing high performance. Figure 6 shows the complete architecture of RTHS. This design uses pipelining to reduce the critical path in **dual-mode pipeline bitonic networks (DPBNs)**. Figure 7 shows a DPBN unit for 8 inputs. The number of pipeline stages in a DPBN is directly proportional to its number of steps, which can be computed by $(1/2\log_2(N)(\log_2(N)+1))$, where N is the number of inputs. The implicit switch is done by fixed wiring and so is entirely static. This hardwired switching does not require additional routing resources and has minimal overhead.

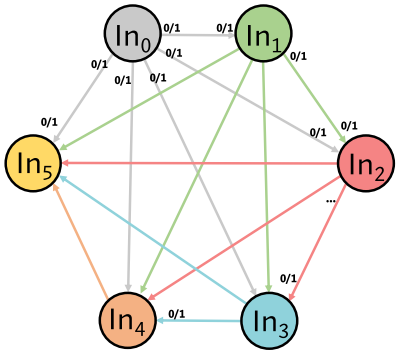


Fig. 8. An example of mapping unsorted input data to a graph. Each input, represented by a vertex, is linked to all other vertices through directed edges, indicating a directed, fully connected graph. Source [41].

Jelodari et al. [41] proposed a low-complexity sorting network design, which maps the unsorted input data to a graph. In this graph, the vertices represent inputs and are fully connected through directed edges, as shown in Figure 8. This structure allows comparing all inputs with each other through the directed edges connecting their corresponding vertices. At each end of any graph edge, the corresponding vertex is tagged by 0 or 1. The tags of the vertices connected by an edge are always complementary. The outgoing tag "1" means the source vertex is greater than or equal to the sink vertex. The sum of the tags assigned to each vertex indicates the position of the corresponding input data in the sorted output.

Papaphilippou et al. [66], [65] introduced a merge sorter tailored for small *list*s, which can merge sublists recursively. This feature sets their solution apart from most large-scale sorters, often reliant on pre-sorted sublists or established hardware sorter modules. Their design bridges the gap between high-throughput and many-leaf sorters by merge sorters, allowing bandwidth, data, and payload width customization. They assess the applicability of their solution in their specialized in-house context, specifically for database analytics. This involved calculating the count of distinct values per key (group) from a dataset comprising key-value pairs. They integrated a fully-pipelined high-throughput stream processor seamlessly with the sorter's output, enabling real-time result generation. Their streamlined process eliminates the need for temporary data storage, exemplifying task-pipelining for efficient data processing. Figure 9 shows their setup. They incorporate a **fast, lightweight merge sorter** (**FLiMS**) as a key component within their parallel merge/sort tree. The FLiMS unit combines two separate -already sorted- lists. The design is characterized by p linear sorters, where p signifies the degree of parallelism being employed. Each individual linear sorter has a length of N/p, with N representing the total merge capacity or the size of the sorted chunk. This architectural arrangement sorts an input dataset comprising N elements while adeptly merging already sorted lists of varying lengths.

Preethi et al. [69] investigated the use of a clock gating technique to design low-power sorters. The `bubble sort`, `bitonic sort`, and `odd-even sorting` algorithms are redesigned to make them low-power using a clock gating technique. The implementation results showed that clock gating can reduce the dynamic power consumption of sorters by 47.5% with no significant impact on the performance.

2.2 Comparison-free

In recent years, comparison-free sorting has gained significant attention due to its potential to overcome the limitations of traditional comparison-based methods. Unlike comparison-based sorting,

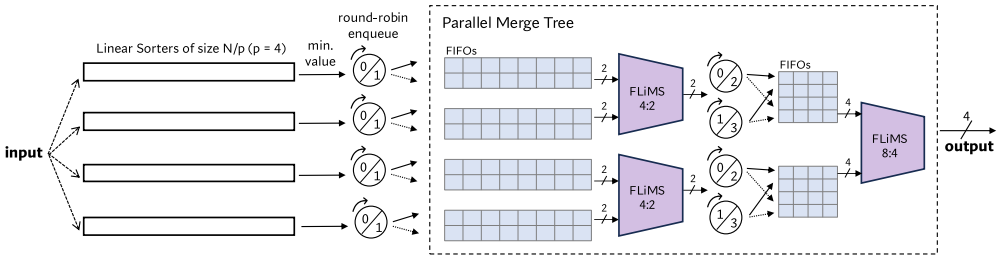


Fig. 9. The high-throughput sorting system for sorting data quickly while merging them efficiently at a parallelism degree of $p = 4$. This structure possesses the capability to perform both the sorting of an input containing N elements and the merging of N sorted lists of variable lengths. The 2-bit counter is incremented whenever a new sorted chunk is flushed to the "Parallel Merge Tree." This plays a vital role in the FLiMS (fast lightweight merge sorter) system [65], ensuring correct sorting prioritization for independently sorted chunks. Source: [66].

which relies on compare-and-swap operations to determine the order of elements, comparison-free sorting eliminates the need for direct element comparisons. This approach leverages alternative techniques, such as unary computing, one-hot encoding, and state-based processing, to achieve efficient and scalable sorting. These methods are particularly advantageous in hardware implementations, where minimizing complexity, latency, and power consumption is critical. The surge in research and development of comparison-free sorting is driven by its compatibility with emerging paradigms like in-memory computing, which addresses the Von Neumann bottleneck by performing operations directly in memory. This reduces energy consumption and data movement overhead, making comparison-free sorting ideal for energy-constrained environments, such as mobile or edge devices. Furthermore, these methods achieve lower latency and greater scalability, enabling efficient processing of large datasets in high-throughput applications. Comparison-free sorting also avoids the extensive switching activities inherent in comparison-based methods, thereby reducing dynamic power dissipation. Sorting networks like bitonic or merge sort, while robust, introduce significant latency due to their sequential comparison layers. By contrast, comparison-free approaches, such as parallel sorters, achieve efficient sorting with fewer resource demands, making them an advanced choice for modern hardware-oriented systems. In this section, we will provide an overview of these advancements. Table 4 provides a detailed comparison of sorting architectures in this group, highlighting their advantages and tradeoffs.

Abdel-Hafeez and Gordon [2] proposed a comparison-free sorting algorithm for large datasets. The method operates on the elements' one-hot weight representation, a unique count weight associated with each of the N elements. The input elements are inserted into a binary matrix of size $N \times 1$, where each element is k bits. Concurrently, the input elements are converted to a one-hot weight representation and stored in a one-hot matrix of size $N \times H$. In this matrix, each stored element is of size H-bit, and $H = N$ gives a one-hot matrix of size N-bit×N-bit. The one-hot matrix is transposed to a transpose matrix of size $N \times N$, which is multiplied by the binary matrix to produce a sorted matrix. An example of this method is illustrated in Figure 11. The total number of sorting cycles is linearly proportional to the number of input data elements N. The architecture of [2] is a high-performance and low-area design for hardware implementation.

Bhargav and Prabhu [9] later proposed an algorithm for comparison-free sorting using **finite-state machines (FSMs)**. Their FSM consists of six states that describe the functionality of a comparison-free sorting algorithm dealing with N inputs. Their proposed design shows 53% and 68% savings in area and power consumption compared to the design proposed in [2].

Table 4. Comparative Analysis of Sorting Architectures (Discussed in Section 2.2), Highlighting Key Experimental Metrics Such as Hardware, Complexity, Latency, Energy Efficiency, Scalability, and Optimization Focus

Ref.	Hardware	Time Complexity	Space Complexity	Latency	Energy	Area	Dataset / Application
[2]	90-nm	$O(N)$	$O(N)$	$4-6\mu s$ for $N=1024$	$6\mu s * 1.6mW$	750000 transistor	Database visualization
[9]	90-nm	$O(N)$	N/A	Not explicitly mentioned	N/A	$52739\mu m^2$	Sorting of large datasets
[18]	90-nm	$O(N)$	$O(N)$	Reduced due to bidirectional sorting	$1.61nj$ for $N=1024$	$605\mu m^2$	Sorting of large and Gaussian-distributed data
[86]	45-nm	$O(N)$	$O(N)$	21% increase compared to [18]	32% increase compared to [18]	31% saving compared to [18]	Sorting of large datasets
[30]	FPGA (Virtex5)	$O(N)$	$O(N)$	5.3 to $15.5ns$ per element	Reasonable for tested dataset sizes	2016 LUTs	Pseudorandom, random, and sorted datasets
[73]	45-nm	$O(N)$	N/A	Average 3 cycles for 256-input design	N/A	Up to 81% reduction in area	Simulation datasets with random inputs
[39]	45-nm	$O(N)$	$O(N)$	Linear delay with smaller clock periods	Efficient energy utilization due to parallel clusters	Efficient resource utilization	Custom data for real-time applications
[96]	130-nm	$O(N)$	$O(N^2)$	$1ns$ per input	N/A	N/A	Image processing

Chen et al. [18] improve the number of sorting cycles, which range from $[2N$ to $2N+2K-1]$ to $[1.5N$ to $2N+(\frac{2^k}{2})-2]$. Their proposed architecture improves the performance of the unidirectional architecture in [2] by reducing the total number of sorting cycles via bidirectional sorting along with two auxiliary modules. One of the auxiliary modules is *boundary finding*, which is designed to record the maximum and minimum values of the input data for the high-index part (max H and min H) and the low-index part (max L and min L). The high-index part processes the upper half of the data range (e.g., $2^{K/2}$ to 2^K-1), while the low-index part processes the lower half (e.g., 0 to $2^{K/2}-1$), where K is the bit width of the data elements. Sorting tasks are performed concurrently in both parts, significantly reducing sorting cycles. Boundary values for each part (maxH, minH, maxL, minL) are calculated during the count mode to narrow the search range, enhancing efficiency.

As shown in Figure 10, the boundary values are stored in four K-bit registers where K is the bit-width of input data. In the initial state of the circuit, the values of max H, min H, max L, and min L are set to $2K/2$, $2K-1$, 0, and $(2K/2)-1$, respectively. A *binary finding* module shortens the range for index searching by finding the boundaries of the range. Bidirectional sorting allows the sorting tasks to be conducted concurrently in the high- and low-index parts of the architecture. Sri et al. [86] reduce the area, delay, and power consumption of the design of [18] by improving the boundary finding module. Key modifications include eliminating the traditional count and selection modes in favor of a single count variable, streamlining the boundary finding module by removing AND gates and multiplexers, and optimizing the index storing process by replacing flag registers with counter variables. These changes result in significant reductions in power consumption, area usage, and delay. Simulation and synthesis results, using Verilog HDL and Cadence Genus (45 nm technology), show that the proposed architecture achieves up to 46% lower power consumption, 31% reduced area, and 21% lower delay compared to existing designs.

Ray and Ghosh [73] developed an architecture for parallel comparison-free sorting that sorts N, n-bit elements consuming linear worst case sorting latency of $O(N)$ clock cycles utilizing p clusters, with p as the parallelism degree. based on a model presented earlier in [30]. This work

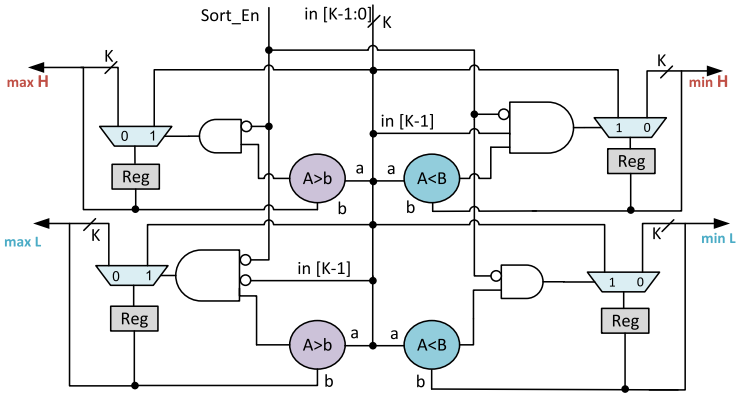


Fig. 10. Architecture of the boundary founding module. Source [18].

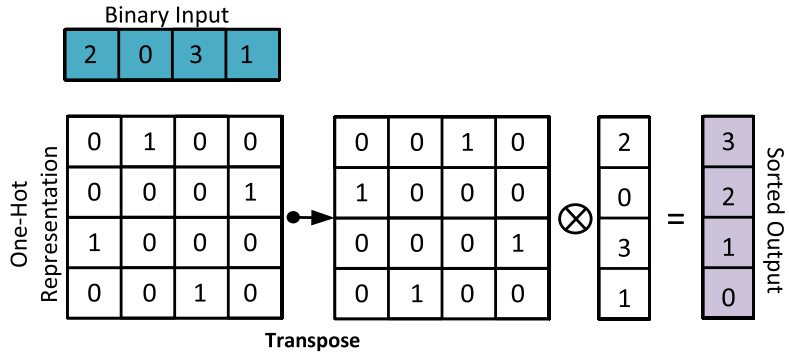


Fig. 11. Example of sorting four input data with the method of [2].

sorts N data elements completely by utilizing N iterations with a speed-up of $\frac{n}{\lceil \frac{n}{p} \rceil + p}$ compared to non-parallel architectures.

Jalilvand et al. [39] proposed a fast and low-cost comparison-free sorting architecture based on UC. Similar to [40, 57], their method iteratively finds the index of the maximum value by converting data to left-aligned unary bit-streams and finding the first "1" in the generated bit-streams. Figure 12 shows the high-level architecture. The architecture includes a sorting engine, a controller, and a multiplexer. The design reads unsorted data from the input registers and performs sorting by finding the address of the maximum number at each step. Figure 13 shows the architecture of the sorting engine. The sorting engine contains simple logic and converts data to right-aligned unary bit-streams. It returns the index of the bit-stream corresponding to the maximum value. This is done by finding the bit-stream that produces the first "1". The design also employs a controller that gets a duplication sign signal from the sorting engine and puts the following value to the output sorted register.

Finally, Yoon [96] proposed a sorting engine based on the radix-2 sorting algorithm. Their sorting engine avoids comparison by creating and distributing data into buckets according to the radix-2 sorting.

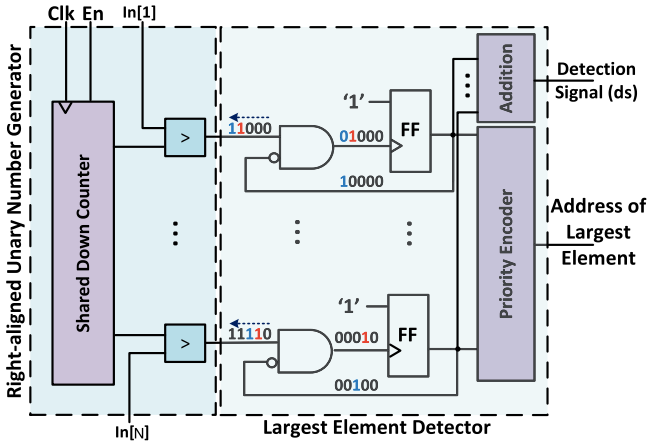


Fig. 12. High-level architecture of the comparison-free unary sorter. Source: [39].

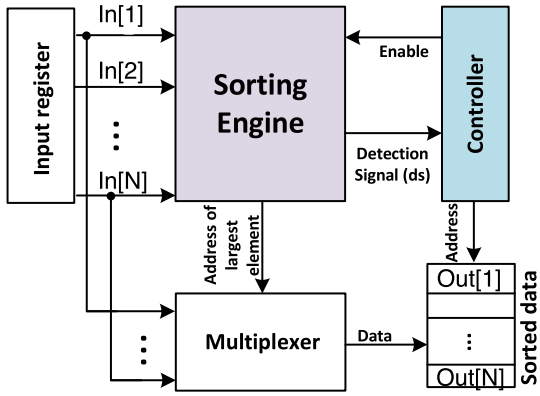


Fig. 13. The *Unary Sorting Engine* Source: [39].

3 Partial Sorting

Partial sorting is primarily used to sort the top-M largest or smallest values out of N elements, where $M < N$. Partial sortings have been used to determine the minimum and maximum values [74], find more than one relative maximum and minimum (max-set min-set selection) [28], merge of partially sorted data, and approximate partial sorting [99]. Finding the minimum and maximum values among a set of data has been particularly an important target of partial sorting. FPGA has been a popular platform for implementing this type of sorting in hardware.

A comprehensive comparison of various hardware implementations is presented in Table 5, showcasing their respective time complexity, latency, energy consumption, and application domains.

Partial sorting is ideal for applications where only a small portion of the data is relevant, such as partial ranking systems in search engines, recommendation engines targeting the most or least popular items or users, competitive ranking, and brain-inspired spike processing systems. It is particularly suited for approximate computing applications, where processing a subset of elements

Table 5. Comparative Analysis of Various Hardware Implementations for Partial Sorting Methods, Highlighting Experimental Data: Time and Space Complexities, Latency, Energy Consumption, Area Utilization, and Target Applications or Datasets

Ref.	Hardware	Time Complexity	Space Complexity	Latency	Energy/Power	Area	Dataset/Application
[97]	FPGA	N/A	N/A	Learning: 5.6 ms	6.4 mW (10-bit), 8.6 mW (16-bit)	777 Slices, 41 BRAM (10-bit) / 1113 Slices, 65 BRAM (16-bit)	Real-time multi-channel neural signals
[14]	FPGA	O(1) to O(log(n))	$O(n^2)$ to $O(n^2)$	N/A	N/A	N/A	Sorting networks (SNs) with min/max circuits
[87]	FPGA	O(N)	O(1)	8 cycles	N/A	1552 CLB, 344 DFF, 152 LUT	3×3 window median filters
[42]	FPGA	< O(n)	O(n)	358-370 MHz	N/A	136-5487 LUTs, 181-8267 Regs	Vector-based parallel sorting
[91]	FPGA	Template matching	N/A	68 clock cycles	64 nW @ 24 kHz (ASIC)	0.3 mm^2 (ASIC) / 4880 Slice Registers, 6628 LUTs (FPGA)	Real-time single/multi-channel neural spike sorting
[101]	In-DRAM (Max-PIM)	O(1)	Constant	1 cycle (for XNOR operation)	0.44 mJ for dataset T10I4D100K	Similar area as Ambit and DRISA-1T1C	Big data applications, sorting, graph processing
[95]	FPGA	N/A	N/A	60 cycles (32768-to-128 sort)	N/A	78,190 Slices, 456 BRAM	Partial sorting of large datasets, e.g., 256-to-32, 32768-to-128

is sufficient. In larger systems, partial sorting benefits task scheduling, priority queues, and real-time data processing by prioritizing critical or urgent elements instead of sorting the entire dataset. This makes it especially advantageous in memory-constrained or real-time environments, where full sorting may be too resource-intensive. Embedded systems, on-edge processing, external sorting, and streaming algorithms can efficiently handle large data volumes or continuous streams using partial sorting. For instance, neural network batch processing can leverage partial sorting in stochastic gradient descent optimizations, where k-fold cross-validation may focus on partially sorted batches while unsorted batches undergo standard training. Similarly, applications in image processing or sensor data analysis can utilize partial sorting to detect significant features or data points without the need to sort the entire dataset, enabling faster and more resource-efficient computations.

Yan et al. [95] proposed an architecture for determining the k largest or smallest numbers on FPGA. Their work allows selecting two min/max subsets with a **real-time hardware partial sorter** (**RTHPS**) structure consisting of even-odd swap blocks, a bitonic sorting network, and parallel swap blocks. Korat et al. [42] proposed a sorting algorithm that partially sorts the odd and even parts in a vector structure. Their method guarantees a linear time complexity with $O(n)$. The hardware unit includes two multiplexers and a comparator, which is responsible for ordering input pairs. Their FPGA-based hardware design implemented on a Xilinx VIRTEX-7 VC707 FPGA consumes 136 LUTs and 181 registers with a working frequency of 370 MHz when sorting eight inputs.

Median sorting is another practice of partial sorting with wide application in image processing, particularly for image enhancement. Various hardware designs for median filtering have been proposed in the literature. Subramaniam et al. [87] proposed a hardware design for finding a dataset's median value. They employ selective comparators to locate the median, allowing for partial sorting with fewer elements than conventional designs that necessitate a fully sorted list. CAS operations are obtained using a comparator and two 2-to-1 `MUX`s. They implement the design on an FPGA (Xilinx FPGA Virtex 4 XC4VSX25) and evaluate it using an image processing case study [87]. Using a pipelined architecture, Cadenas et al. [12, 13] proposed a median filtering architecture using accumulative parallel counters. Najafi et al. [60] further implemented a low-cost median filtering

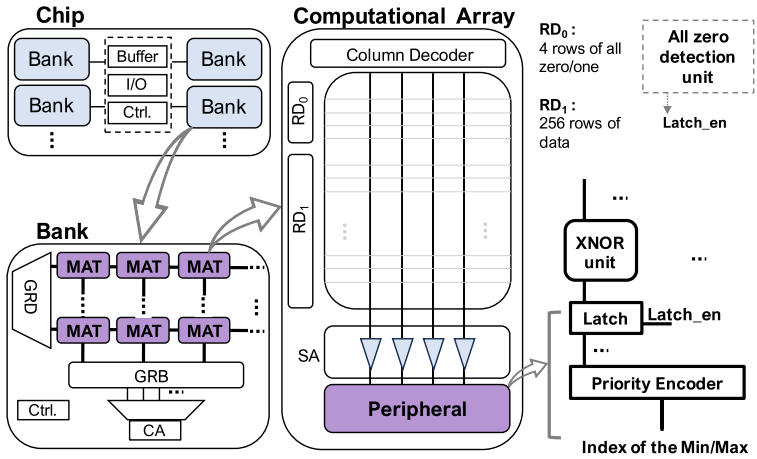


Fig. 14. The MIN-MAX PIM architecture preserves the original memory hierarchy with each DRAM chip divided into multiple banks. Source: [101].

design based on UC by converting data to unary bit-streams and processing them in the unary domain using simple standard AND and OR gates. Finally, Riahi Alam et al. [3] proposed a binary and a unary architecture for energy-efficient median filtering completely in memory.

Finding the maximum and minimum values is one of the current topics in in-memory computing applications. Zhang et al. [101] proposed an in-memory min-max sorting architecture in DRAM technology for fast and big data applications. In Figure 14, each bank comprises **multiple memory matrices** (**MATs**), which are essentially DRAM subarrays. The design supports Ambit [5, 76] logic and instructions with enhanced support for the **Dual Row Activation** (**DRA**) mechanism, thus providing compatibility with XNOR operations. The Computational Array includes (i) two-row decoders, (ii) one column decoder, (iii) modified logic sense-amplifier, (iv) one latch per bit-line, (v) pseudo-OR gate, and (vi) one priority encoder. Sorting and graph processing applications are provided with an architecture that produces results 50 times faster than a GPU. This architecture includes two-row decoders, a one-column decoder, a modified logic sense amplifier with a **typical sense amplifier** (**TSA**), one latch per bit-line, a pseudo-OR gate, and one priority encoder (for the resultant index of minimum and maximum locations).

Campobello et al. [14] discuss sorting networks' complexity and propose a multi-input maximum finder circuit. Their design finds the maximum value using an XNOR comparator, a zero catcher (via Q-port feedbacked D flip-flip), a buffer with enable for each input, an OR gate, and a D-flip-flop.

Partial sorting can also be used as an intermediary tool to help understand data, e.g., to find outliers [53]. This includes the complex task of *spike sorting* in brain-inspired computing. Spike sorting encompasses algorithms designed to identify individual spikes from extracellular neural recordings and classify them based on their shapes, attributing these detected spikes to their respective originating neurons. This sorting process differs from conventional sorting as it involves machine learning-related steps such as detection, feature extraction, and classification. Instead of straightforward scalar sorting, spike sorting resembles the segmentation of patterns within brain signal pulses [102]. Spike sorting involves partial sorting for tasks such as early learning termination, outlier analysis, and spike activity thresholding.

In the literature, spike sorting for a unit activity may encompass partial sorting to separate multi-unit activity into distinct groups of single-unit activity [31]. The segmentation of spike data plays a

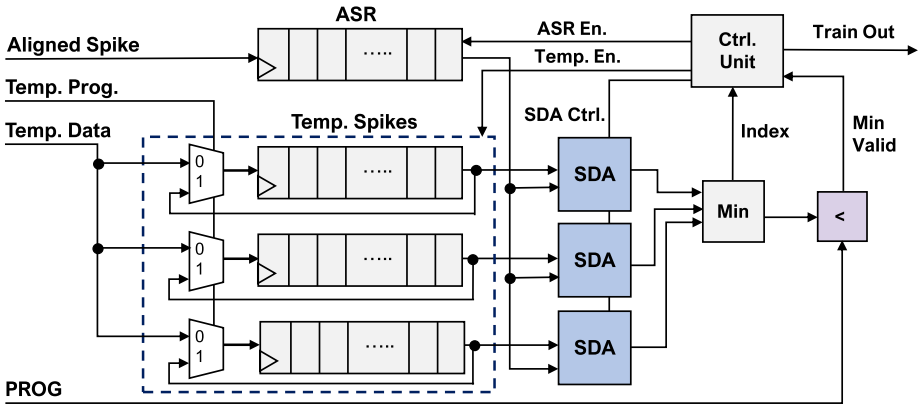


Fig. 15. Template matching-based architecture for spike sorting. Source: [91].

significant role in distinguishing specific activities within the overall spike data. Within the domain of spike processing, some studies underscore partial sorting for outlier analysis of the spikes [53], while others commend it for thresholding operations [21]. Valencia and Alimohammad [91] implement a hardware module for spike sorting architecture. Their design incorporates a template matching unit to compute the minimum distance between spikes during the spike sorting process. Figure 15 depicts the spike sorting design, which relies on template checks and minimum distance calculations. In Figure 15, the aligned spike is directed into an **Aligned Shift Register (ASR)** module, which has been set up for parallel input and serial output. The values stored in the templates and the ASR module are then transferred into some **Squared Difference Accumulator** (**SDA**) units. These SDA units are used to calculate and accumulate the squared differences between the spike waveform preserved in the ASR and templates. The MIN unit identifies and conveys the minimum value to the comparator, along with the index of the minimum value, which is then passed on to the Control Unit. A substantial reduction of raw data to sorted spikes is achieved by transmitting only those sorted spikes (in a partial sorting manner) that match a small set of frequently encountered waveforms. Such advancement in hardware-powered sorting is expected to open new research avenues in emerging machine-learning models, particularly brain-inspired computing.

4 In-Memory Sorting

In addition to complete and partial sorting methods, we also explore in-memory sorting as an emerging approach, given the growing prominence of in-memory computing. While some methods from the complete and partial sorting categories already incorporate memory hierarchies, the ones implemented with in-memory architectures will be revisited and further discussed in this section. Table 6 provides a systematic overview of the recent in-memory sorting architectures, comparing their performance across critical metrics such as time complexity, space complexity, latency, and energy consumption. The compilation highlights the diverse approaches in hardware design, ranging from NAND flash-based SSDs to advanced memristive memory arrays, demonstrating the ongoing innovation in computational sorting techniques.

The conventional sorting approach wastes a significant portion of the total processing time and energy consumption for transferring data between the memory and processing unit. Most prior sorting designs are implemented based on this Von Neumann architecture with separate

Table 6. Comprehensive Comparative Analysis of In-memory Sorting Architectures Considering Experimental Data

Ref.	Hardware	Time Complexity	Space	Latency	Energy	Dataset/ Application
[94]	NAND flash-based SSD	$O(N)$	$O(N)$	Dependent on record size (e.g., 25μs read, 200μs write)	N/A	Variable-length records, databases
[75]	FPGA	$O(N \log N)$	$O(N)$	4-32 GB with $2.3 - 2.5x$ speedup	$3.3x$ better bandwidth efficiency	Datasets from MB to TB
[48]	Hybrid Memory Cube (HMC)	$O(N \log N)$	$O(N)$	Parallel in-memory sorting, reduced data movement	Improved energy efficiency	Large-scale data processing
[68]	Memristive Memory Arrays	$O(N)$	$O(1)$	$12.4 - 50.7x$ throughput gains	N/A	Database, graph analytics, network processing
[22]	Non-Volatile Memory (NVM)	$O(N \log N)$	$O(N)$	Efficient sorting, reduced NVM write operations	N/A	Partially sorted datasets, hybrid memory
[3]	Memristive Memory	$O(N \log N)$	$O(N)$ and $O(2^N)$	Reduced processing time	$37x$ (binary), $138x$ (unary) energy reduction	Efficient in-memory sorting
[98]	Memristive Memory with Near-Memory Circuit	$O(N)$	$O(N)$	Column-skipping, up to $4.08x$ speedup	$3.39x$ energy efficiency	Scalable sorting across memory banks
[104]	Skyrmion Racetrack Memory (SRM)	$O(N \log N)$	$O(N)$	In-memory sorting, reduced data movement	Energy-efficient data transfer	Large-scale datasets in data centers
[46]	Digital Processing Elements (PEs)	$O(N \log N)$	$O(N)$	Cooperative resource sharing	Improved through algorithm/hardware co-optimization	Scalable in-memory sorting
[50]	Non-Volatile Memory	$O(N \log N)$	$O(N)$	Customized sorting for NVM	N/A	Non-volatile memory applications
[29]	Network of Processing Elements (PEs)	$O(N)$	$O(N)$	Low latency due to design	Linear dependency with data bit-width and size	High performance, low latency applications

memory and processing units [32]. **In-memory computation** (**IMC**) –aka **processing-in-memory** (**PIM**)– is a promising solution to address this data movement bottleneck. In this processing approach, the chip memory is used for both storage and computation [90]. With that in mind, *in-memory sorting* has been suggested [3, 68], especially considering the unique characteristics of **non-volatile memories** (**NVMs**), which position them as a strong contender for efficient sorting within memory.

NVMs are particularly suited for in-memory sorting due to their high density, enabling the storage of large datasets within a compact memory footprint, and their non-volatility, which ensures data persistence during power interruptions. Furthermore, the inherent bitwise parallelism of NVMs allows for efficient comparison and manipulation of data, while their low static power consumption minimizes energy usage. However, challenges such as limited endurance and higher write latencies pose significant constraints, as frequent write operations can degrade device reliability and necessitate algorithmic and architectural optimizations. IMC-based sorters address these challenges by leveraging the strengths of NVMs to eliminate the Von Neumann bottleneck, reducing data movement overhead and enabling massive parallelism directly within memory arrays. This results in significantly lower latency and energy consumption, particularly for large-scale and energy-constrained applications. Nonetheless, the choice of hardware for sorting depends on specific application requirements. While NVMs excel in scenarios demanding high-density storage and non-volatility, other hardware, such as GPUs, FPGAs, and ASICs, may outperform NVMs in certain contexts. GPUs are well-suited for highly parallel tasks with irregular data access, FPGAs offer customizable, low-power solutions for real-time and large-scale tasks, and ASICs provide efficiency for fixed-function tasks in dedicated applications. Moreover, volatile memories like DRAM or SRAM may be preferable for latency-sensitive applications where endurance and non-volatility are less critical. By balancing these tradeoffs, IMC-based sorting architectures demonstrate their potential to revolutionize hardware efficiency in sorting operations.

Chu et al. [22] proposed an NVM-friendly sorting algorithm called "NVMSorting". NVMSorting is a modification of the MONTRES algorithm [45], a sorting algorithm resembling merge sort, designed for flash memory. MONTRES aims to enhance performance by minimizing I/O operations and reducing the generation of temporary data during sorting. It includes a run generation phase and a run merge phase, employing optimized block selection, continuous run expansion, and on-the-fly merging for efficient data organization. NVMSorting can detect partially ordered runs by using a new concept called *natural run* to reduce the sorting cost. A natural run consists of multiple blocks. The items within each block do not need to be sorted, but the items between any two consecutive blocks are ordered. In the first step, the algorithm searches for the input data's partially ordered runs (i.e., natural runs). The next step is the *run generation*, based on merge-on-the-fly and run expansion mechanisms. DRAM is divided into two sections: (I) workspace for the natural runs and (II) workspace for the other input data. Chu et al. use the NVM's byte-addressable capability to merge the runs. Their evaluations show that NVMSorting is more efficient than the traditional merge sorting algorithms in terms of execution time (t) and number of NVM writes (w). However, if the dataset is entirely random, NVMSorting can perform similarly to MONTRES, hybrid sort, and external sort [92].

Li et al. [48] proposed a PIM architecture called IMC-Sort to perform parallel sort operations using a **hybrid memory cube** (**HMC**). As shown in Figure 16, IMC-Sort comprises sorting units specifically designed to operate within each HMC vault's logic layer. The control unit of the HMC vault is enhanced with some logic to carry out the sorting process. The sorting units in IMC-Sort can parallel access and utilize the HMC crossbar network to communicate with one another.

In an "*Intra-vault merging*" step, they utilize a *chunking* technique to accommodate a range of input sequence lengths using a fixed number of CAS units and a fixed input permutation unit. They divide the sequence into chunks of a specific size determined by the number of CAS units. Then, they sort the chunks. Finally, the sorted values are merged into a single sorted sequence. On the other hand, an "*Inter-vault merging*" step combines the sorted values or sequences from all vaults to produce a globally sorted sequence. IMC-Sort delivers 16.8× and 1.1× speedup and 375.5× and 13.6× reduction in energy consumption compared to the widely used CPU implementation and a state-of-the-art near-memory custom sort accelerator [28, 71].

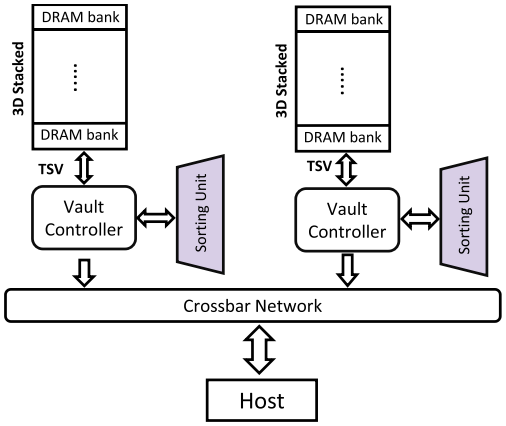


Fig. 16. Overall architecture of the IMC-Sort. A single stack HMC vault is composed of several DRAM banks that are linked to the logic layer via TSV. Source: [48].

Riahi Alam et al. [3] proposed the first in-array (in-memory) architectures for high-performance and energy-efficient data sorting completely in memory using memristive devices. They introduce two different architectures. The first architecture, "*Binary Sorting*," is based on the conventional weighted binary representation, while the second architecture, "*Unary Sorting*," is based on the non-weighted unary representation. Both of these sorting designs achieve a significant reduction in the processing time compared to prior off-memory binary and unary sorting designs. They used the memristor technology based on the *stateful logic* in which the input and output are presented as the state of input and output memristors. In stateful logic, values are stored and maintained within memristive switches through their resistance states. These switches store logic values and perform logical operations, exhibiting both memory and computational capabilities [36]. They implement the boolean operations with **memristor-aided logic** (**MAGIC**) [44] in a crossbar implementation. Each MAGIC logic gate utilizes memristors as inputs containing previously stored data and additional memristor functions as the output. Parallel architectures such as CAS-based sorting networks can be executed efficiently within the memory using these IMC logic operations [3].

In the first design, the memory is split into multiple partitions to enable parallel execution of different CAS operations of each bitonic CAS stage. The number of partitions indicates the number of CAS units that can run in parallel. The first two inputs of each partition are sorted using a basic sorting operation. Then, the maximum value of each basic sort operation is copied to another partition determined by the sorting network. The second design is a complete unary sort system that follows the same approach as the binary implementation but represents and processes the data in the *unary* domain with uniform unary bit-streams [61]. The comparison operations are implemented in this design based on a basic unary sorting unit. Their performance evaluation results show a significant latency and energy consumption reduction compared to the conventional off-memory designs. On average, their in-memory binary sorting resulted in a 14× reduction in latency and a 37× reduction in energy consumption. On the other hand, the average latency and energy reductions for the in-memory unary sorting design were much more significant, at 1200× and 138×, respectively. Further, they implemented two in-memory binary and unary designs for Median filtering based on their developed in-memory basic sorting units. Their results showed an energy reduction of 14× (binary) and 5.6× (unary) for a 3 × 3-based image processing system and 3.1× and 12× energy reduction for binary and unary median filtering, respectively, for a 5 × 5-based image processing system compared to their corresponding off-memory designs.

Today's systems often face memory bandwidth constraints that can limit their performance. The available memory bandwidth can enormously impact the efficiency of the sorting algorithms. To overcome the bandwidth problem in large-scale sorting applications, Prasad et al. [68] proposed an iterative in-memory min/max computation technique. They applied a novel mechanism called "RIME," which enhances bandwidth efficiency by enabling extensive in-situ bitwise comparisons. RIME eliminates unnecessary data movement on the memory interface, improving performance. They provide an API library with significant control over essential in-situ operations like ranking, sorting, and merging. With RIME, users can efficiently manage and manipulate data. To perform bit-serial min/max operation, they execute an iterative search for bit value (1 or 0) within individual columns of a data array using a 1T1R memristive memory. Each search iteration generates a match vector to identify which rows in the array should be eliminated from the dataset. The memory array must be capable of performing two additional operations, namely bitwise column search and selective row exclusion.

The algorithm examines the binary values of all bit positions, beginning from the most significant bit position in a set of numbers. This process is carried out using a M-step algorithm, during which some of the non-minimum or non-maximum values may be removed from the set at each step. At each step, a selection of matching numbers is formed by searching for "1" at the current bit position. The selected numbers are removed from the set only if the set and selection are unequal. This results in all the final remaining numbers in the set having the minimum value. By eliminating the unnecessary data movement for finding min/max of given data, their sorting operation obtains a bandwidth complexity of $O(N)$. With the suggested in-memory min/max locator, the costs of accessing bandwidth when searching for the M^{th} value in a range of data decrease to M operations, which shows a bandwidth complexity of $O(M)$. Their simulation results on a group of advanced parallel sorting algorithms demonstrate a significant increase in throughput ranging from 12.4× to 50.7× when using RIME.

Yu et al. [98] improve the speed and performance of Prasad et al.'s design by proposing a column-skipping algorithm that keeps track of the column read conditions and skips those that are leading 0's or have been processed previously (see Figure 17). A *bank manager* enables column-skipping for datasets stored in different banks of the memristive memory. For detecting and skipping redundant column reads, the algorithm records the k most recent row exclusion states and their corresponding column indexes, which can be reloaded to avoid repeating these states.

To tackle the sorting challenges of large-scale datasets, Zokaee et al. [104] proposed Sky-Sorter, a cutting-edge sorting accelerator powered by **Skyrmion Racetrack Memory** (**SRM**). Sky-Sorter leverages the unique capabilities of SRM, enabling the storage of 128 bits of data within a single racetrack. Sky-Sorter adopts the `sample sort` algorithm, which encompasses four essential steps: sampling, splitting marker sorting, partitioning, and bucket sorting. First, it employs a random sampling technique to estimate the distribution of the dataset. This sampled subset is then sorted, and specific records are selected as splitting markers. The markers are crucial for defining the boundaries of non-overlapping buckets. The next step involves partitioning, where all records, excluding the splitting markers, are allocated to appropriate buckets based on their relationship to the markers. Lastly, each bucket is sorted individually, and the results are concatenated to produce the final sorted sequence. Bucket sorting, known for its high parallelizability, is the key to this algorithm's efficiency, with the distribution of bucket sizes playing a crucial role in maintaining balance. It is important to evenly distribute records across all buckets to ensure a balanced distribution and avoid load imbalances in bucket sorting. Larger random sampling sizes contribute to more accurate estimates of the data distribution and less variability in bucket sizes. The algorithm ensures that the probability of any bucket exceeding an upper size limit is nearly zero. In rare

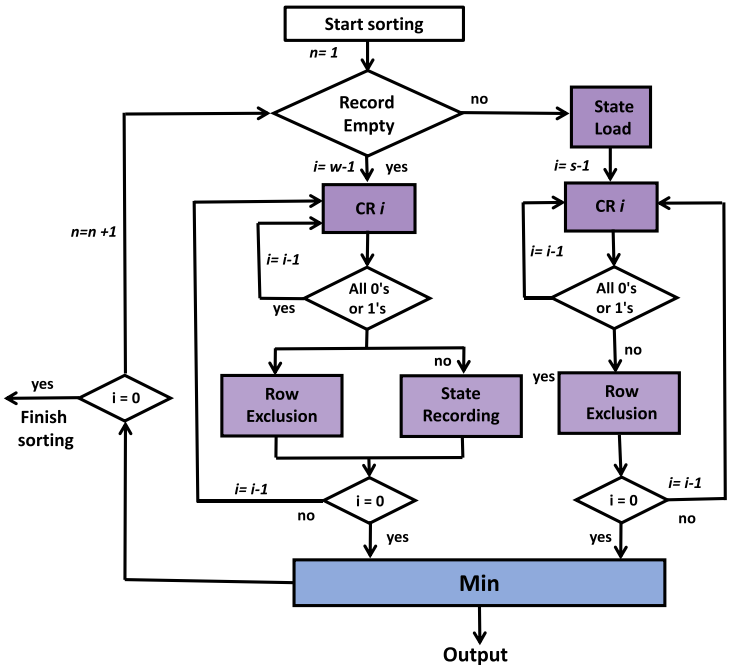


Fig. 17. Iterative min search with proposed column-skipping algorithm. Source: [98].

cases where a bucket size surpasses this threshold, the algorithm resamples splitting markers to maintain uniformity in bucket sizes. The fundamental cell structure of SRM is composed of four integral parts. These components encompass two injectors devoted to creating skyrmions, a detector designed for precisely detecting skyrmions, a nano track to facilitate the controlled motion of these skyrmions, and peripheral circuits that support and coordinate the functionality of the entire cell. The authors claim that Sky-Sorter improves the throughput per Watt $\sim 4\times$ over prior FPGA-, **Processing Near Memory** (**PNM**)-, and PIM-based accelerators when sorting with a high bandwidth memory DRAM, a DDR4 DRAM, and an SSD.

Liu et al. [50] proposed LazySort, an external sorting algorithm tailored to the NVM-DRAM hybrid storage architecture. LazySort leverages NVM's byte-addressable feature and locally ordered data to minimize write operations to NVM. It comprises two stages: Run generation and Merge. To enhance efficiency, they introduce an optimization strategy known as RunMerge for the merge stage. RunMerge intelligently merges non-intersecting data blocks based on the range of index table records, reducing the total number of runs and memory usage. To validate the performance, they established a real NVM-DRAM experimental platform and conducted comprehensive experiments. The results showed LazySort's superior time performance and significantly reduced NVM write operations. Compared to traditional external sorting algorithms relying on HDD/SSD, LazySort reduced sorting time by 93.08% and minimized NVM write operations by 49.50%. This design then addresses an important need for efficient external sorting methods for NVM-DRAM hybrid storage.

Lenjani et al. [46] proposed *Pulley*, an algorithm/hardware co-optimization technique for in-memory sorting. Pulley uses 3D-stacked memories. They employ Fulcrum [47] for the baseline PIM architecture. Fulcrum inputs data into a single-word **arithmetic logic unit** (**ALU**) sequentially and enables operations that involve data dependencies and operations based on a predicate. In

Fulcrum, every pair of subarrays has three row-wide buffers called *Walkers*. In the radix sorting proposed in Fulcrum, all buckets have the same length, and a bucket in each pass can always fit in one subarray. For efficient sorting of large data using Fulcrum, Lenjani et al. modified the design by calculating the exact length of each bucket and the position of each key within that bucket. In the first step, the keys of each processing unit are sorted locally. This step dichotomizes the keys into two buckets (*Bucket0* and *Bucket1*). The **subarray-level processing unit** (**SPU**) starts *Bucket0* from the bottom of the space and fills it upward, and starts *Bucket1* from the end of the space and fills it downward. In the next step, each SPU generates the histogram values of the first 256 buckets iteratively, and all SPUs reduce the histogram values of each of the 256 buckets in the lowest sub-array. In Pulley, each vault's core in the logic layer performs a prefix-sum on all the shared sub-arrays in the vaults. Then, the cores in the vault aggregate their prefix-sum arrays. They evaluate Pulley in 1-device and 6-device settings, where each device has four stacks of 8-GB memories. Compared to IMC-Sort, Pulley has a lower working frequency.

Wu and Huang [94] introduced a novel sorting technique tailored explicitly to NAND flash-based storage systems, aiming to optimize performance and efficiency. They propose a record rearrangement and replacement method for unclustered sorting, which involves scanning sorted tags to efficiently rearrange records and minimize unnecessary page reads during the process. They introduce a strategic decision rule to harness the advantages of both clustered and unclustered sorting approaches. This rule categorizes records based on their length and then selects the most appropriate sorting method (clustered or unclustered) for each category, followed by merging the sorted results. They reuse data to reduce page writes by detecting content similarities in the output buffer and marking logical addresses in the address translation table for potential reuse. They provide a comprehensive I/O analysis, comparing the performance of clustered sorting, unclustered sorting, MinSort, and FAST in terms of page reads and writes. Finally, they implement and test the proposed methods on real hardware, including an Intel SSD and a Hitachi HDD, demonstrating significant performance improvements compared to traditional external sorting methods.

Samardzic et al. [75] introduced "Bonsai," an adaptive sorting solution that leverages merge tree architecture to optimize sorting performance across a wide range of data sizes, from megabytes to terabytes, on a single CPU-FPGA server node. Bonsai's adaptability is achieved by considering various factors, including computational resources, memory sizes, memory bandwidths, and record width. It employs analytical performance and resource models to configure the merge tree architecture to match the available hardware and problem sizes. Their approach can enhance sorting efficiency on a single FPGA while also being used as a foundation for potential use in larger distributed sorting systems. Bonsai's primary objective is to minimize sorting time by selecting the optimal adaptive merge tree configuration based on the hardware, merger architecture, and input size. They demonstrate the feasibility of implementing merge trees on FPGAs, highlighting their superior performance across various problem sizes, particularly for DRAM-scale sorting. Bonsai achieves significant speedup over CPU, FPGA, and GPU-based sorting implementations, along with impressive bandwidth efficiency improvements, making it an appealing solution for adaptive sorting.

Most sorting algorithms utilizing dedicated processors are designed solely based on the parallelization of the algorithm, lacking considerations of specialized hardware structures. A systolic array is a specialized hardware architecture designed for efficient computation, particularly in parallel processing. It consists of a network of processors that rhythmically compute and pass data through the system, allowing for high levels of data reuse and minimizing the need for extensive memory access. The processors in a systolic array operate in a synchronized manner, which enhances throughput and computational efficiency. Researchers in [29, 37] propose a Systolic Sorter Array, implemented by a uniform recursive structure that unifies the computational logic of the

systolic array, which is highly parameterized in terms of data size, bit width, and type. In [29], they utilize an array with a capacity of $N + 2$ to manage the sorting sequence. This algorithm operates concurrently on two groups: the LEFT and RIGHT arrays. The sorting process alternates between two states: **Left In, Right Out** (**LIRO**) and **Left Out, Right In** (**LORI**). LIRO, where the LEFT array inputs data while the RIGHT array outputs sorted results, and LORI, which reverses this functionality. The algorithm employs a binary grouping strategy, mapping pairs of data to facilitate comparisons and sorting. It begins with initializing the array with a logical maximum value, and during the LIRO state, inputs are compared to assign values accordingly. As the state transitions to LORI, the algorithm continues to sort and output the data iteratively, achieving a total of $2N$ operations for sorting a single sequence. Overall, the SSA algorithm is structured to ensure efficient parallel execution for simultaneous sorting of two independent sequences. To address large-scale sorting scenarios, the authors in [37] propose an enhanced merge tree structure, the MC-merge tree, which flexibly adjusts concurrency levels and expands sorting capacity while maintaining high throughput. Experimental results demonstrate significant performance gains, achieving a maximum speedup of 73.17x and a throughput of 25.6 Gb/s compared to state-of-the-art algorithms.

5 Open Challenges

Although significant strides have been made in the field of hardware sorting, numerous challenges persist, warranting further research and innovation. In this section, we explore the ongoing challenges within the research on hardware-assisted sorting. Addressing these challenges can result in sorting solutions that are more efficient in different aspects, from performance to footprint area, power, and energy consumption. These challenges are elaborated on in the following sections.

5.1 Algorithmic Considerations

With recent research opportunities and emerging sorting solutions such as in-memory and partial sorting, future research needs to explore potential avenues for radically novel sorting architectures, from algorithmic considerations to hardware-level enhancements. For instance, when developing new sorting algorithms, it is crucial to commence with an initial argument considering a time complexity of $O(n)$. Assessing the evolution of sorting architectures, an emerging trend involves using RAM devices for a new sorting approach known as *stream sorting* [67, 105]. Stream sorting takes N data words as input and produces p words per clock cycle across N/p clock cycles. The sorter achieves a throughput of w if it operates in a fully streaming manner, implying no waiting time between consecutive input sets. Without a fully stream network, the throughput will be less than p words per cycle. We anticipate that one of the pivotal challenges lies in devising algorithms tailored specifically for hardware design, addressing pipeline and parallel processing concerns. Solutions such as stream sorting represent cutting-edge approaches for achieving a more efficient design right from the initial stages, optimizing both memory utilization and time complexity.

5.2 Power and Energy Efficiency

The issue of power usage holds significant importance in current and future hardware designs. Given that sorting designs are being incorporated into a range of embedded and power-limited systems, the reduction of power consumption takes on a vital role. Future works must delve into innovative strategies for ultra-low-power hardware. These could encompass advanced clock gating, dynamic voltage scaling, and enhanced management of data transfer to curtail the energy consumption tied to the implementation of sorting designs. Additionally, by loosening accuracy demands and taking advantage of approximate computing techniques, hardware has the capacity to execute computations with fewer resources.

5.3 Resource Limitations

Hardware designs must operate within the boundaries defined by accessible resources such as registers, memory, and processing units. Striving to optimize the utilization of these resources while upholding performance is challenging, especially when dealing with intricate sorting algorithms that exhibit diverse computational demands. Lin et al. [49] provide a tradeoff between throughput and resources. UC-based solutions (e.g., [3, 60]) have successfully achieved hardware sorting designs with extremely simple digital logic. However, they achieved this at the cost of an exponential increase in latency. Developing future sorting systems based on such emerging computing systems that operate on simple data representations [7, 58] is a promising path forward.

5.4 Latency vs. Throughput Tradeoff

Designing hardware sorting systems necessitates finding the right compromise between latency (the duration of a single sorting operation) and throughput (the number of sorting operations completed within a specific period). Designers must achieve an optimum point based on the application expectations and hardware constraints.

5.5 Parallelism

Sorting algorithms encompass repetitive and regular processes that hold the potential for improvement with parallelization and pipelining. Nonetheless, implementing efficient parallel/pipelined hardware architectures (e.g., [65]) and the oversight of data inter-dependencies can intricate these endeavors. Striking a harmonious equilibrium amidst diverse processing units while upholding synchronization and communication can pose a considerable challenge. PIM solutions hold significant promise for the highly parallel execution of future sorting architectures.

5.6 Adaptation

Numerous practical applications demand data sorting in dynamic and ever-evolving streams. Crafting hardware-based sorting designs capable of adeptly managing these dynamic inputs in real-time presents a multifaceted difficulty. It is imperative for researchers to delve into adaptive algorithms capable of flexibly adapting to shifting input patterns. This adaptability should ensure sustained, efficient sorting performance while minimizing any notable additional workload.

5.7 Customization

Hardware sorting designs may need to be customized for specific applications or environments. This requires flexibility in the design process (e.g., [49, 65]) to accommodate different requirements. From different data types to various data precisions (i.e., bit-width), size of the dataset, and hardware constraints (e.g., area and power budget), achieving the best performance may require customized hardware. However, the higher design time and cost of implementing customized hardware must also be considered.

5.8 Data Movement and Memory Access

Optimal memory access is pivotal for sorting algorithms, and hardware architectures must strive to curtail data transfer and cache-related inefficiencies. Sorting entails frequent data comparisons and exchanges, introducing the potential for irregular memory access patterns. Effectively handling these access patterns is imperative to avert potential performance bottlenecks. The problem aggregates in big data applications where the sorting engine is expected to sort a large set of data.

5.9 Technology Scaling

Hardware designs might necessitate adjustments to accommodate technological shifts, such as advancements in the semiconductor manufacturing process. Designers must meticulously evaluate the potentials and consequences of technological scaling on factors such as performance, area, power and energy usage, and various design parameters.

6 Conclusion

Sorting is one of the crucial operations in computer science, widely used in many application domains, from data merging to big data processing, database operations, robotics, wireless sensor networks, signal processing, and wireless networks. A substantial body of work is dedicated to designing hardware-based sorting. In this survey, we reviewed the latest developments in hardware-based sorting, encompassing both comparison-based and comparison-free solutions. Comparison-based solutions tend to incur high hardware costs, particularly as the volume and precision of data increase. Comparison-free solutions have recently been proposed to overcome the challenges associated with compare-and-swap-based sorting designs. We reviewed recent hardware solutions for partial sorting and stream sorting, which are used to sort the top-k largest or smallest values of the dataset. We also studied the latest emerging in-memory solutions for sorting operations. Finally, we outlined the challenges in developing future hardware sorting, aiming to provide readers with insights into the next generation of sorting systems.

References

[1] Bilal Abbasi, Jeff Calder, and Adam M. Oberman. 2018. Anomaly detection and classification for streaming data using PDEs. *SIAM J. Appl. Math.* 78, 2 (Jan 2018), 921–941. DOI: https://doi.org/10.1137/17m1121184

[2] Saleh Abdel-Hafeez and Ann Gordon-Ross. 2017. An efficient O(N) comparison-free sorting algorithm. *IEEE Transactions on Very Large Scale Integration Sys.* 25, 6 (2017), 1930–1942. DOI: https://doi.org/10.1109/TVLSI.2017.2661746

[3] Mohsen Riahi Alam, M. Hassan Najafi, and Nima Taherinejad. 2022. Sorting in memristive memory. *Association for Computing Machinery Journal on Emerging Technologies in Computing Systems* 18, 4 (2022), 1–21.

[4] Rouf Ali. 2022. Hardware solution to sorting algorithms: A review. *Turkish Journal of Computer and Mathematics Education* 13, 2 (2022), 254–272.

[5] Shaahin Angizi and Deliang Fan. 2019. Accelerating Bulk Bit-Wise X(N)OR operation in processing-in-DRAM platform. Retrieved from https://arxiv.org/abs/1904.05782

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. 1997. High-performance sorting on networks of workstations. *SIGMOD Rec.* 26, 2 (1997), 243–254. DOI: https://doi.org/10.1145/253262.253322

[7] Sercan Aygun, Mehran Shoushtari Moghadam, M. Hassan Najafi, and Mohsen Imani. 2023. Learning from hypervectors: A survey on hypervector encoding. Retrieved from https://arxiv.org/abs/2308.00685

[8] Yomna Ben Jmaa, Rabie Ben Atitallah, David Duvivier, and Maher Ben Jemaa. 2019. A comparative study of sorting algorithms with FPGA acceleration by high level synthesis. *Comp. y Sistemas* 23, 1 (2019), 213–230.

[9] T. A. S. Bhargav and E. Prabhu. 2019. Power and area efficient FSM with comparison-free sorting algorithm for write-evaluate phase and read-sort phase. In *Advances in Signal Processing and Intelligent Recognition Systems: 4th International Symposium SIRS 2018, Bangalore, India.* Springer, 433–442.

[10] J. L. Bordim, K. Nakano, and H. Shen. 2002. Sorting on single-channel wireless sensor networks. In *Proceedings of the 2002 International Symposium on Parallel Architectures, Algorithms, and Networks.* IEEE Computer Society, Los Alamitos, CA, USA, 0153. DOI: https://doi.org/10.1109/ISPAN.2002.1004275

[11] Hoang-Dung Bui, Hai Nguyen, Hung Manh La, and Shuai Li. 2020. A deep learning-based autonomous robot Manipulator for sorting application. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC).* 298–305. DOI: https://doi.org/10.1109/IRC.2020.00055

[12] J. O. Cadenas, G. M. Megson, and R. S. Sherratt. 2015. Median filter architecture by accumulative parallel counters. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 7 (2015), 661–665. DOI: https://doi.org/10.1109/TCSII.2015.2415655

[13] J. O. Cadenas, G. M. Megson, R. S. Sherratt, and P. Huerta. 2012. Fast median calculation method. *Electronics Letters* 48, 10 (2012), 558–560.

[14] Giuseppe Campobello, Giuseppe Patanè, and Marco Russo. 2012. On the complexity of min–max sorting networks. *Information Sciences* 190 (2012), 178–191. DOI: https://doi.org/10.1016/j.ins.2011.12.008

[15] J. M. Chambers. 1971. Algorithm 410: Partial sorting. *Communications of the ACM* 14, 5 (1971), 357–358. DOI: https://doi.org/10.1145/362588.362602

[16] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. FPGA-accelerated samplesort for large data sets. In *Proceedings of the Association for Computing Machinery/SIGDA Int. Symp. on Field-Prog. Gate Arrays* (Seaside, CA, USA). 222–232. DOI: https://doi.org/10.1145/3373087.3375304

[17] Wenhan Chen, Yang Liu, Zhiguang Chen, Fang Liu, and Nong Xiao. 2020. External sorting algorithm: State-of-the-art and future directions. *IOP Conf. Series: Materials Science and Engineering* 806, 1 (2020), 012040. DOI: https://doi.org/10.1088/1757-899X/806/1/012040

[18] Wei-Ting Chen, Ren-Der Chen, Pei-Yin Chen, and Yu-Che Hsiao. 2021. A high-performance bidirectional architecture for the quasi-comparison-free sorting algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 4 (2021), 1493–1506. DOI: https://doi.org/10.1109/TCSI.2020.3048955

[19] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.* 1, 2 (2008), 1313–1324. DOI: https://doi.org/10.14778/1454159.1454171

[20] Kun Chi, Jihong Shen, Yan Li, Yunjie Li, and Sheng Wang. 2021. Multi-function radar signal sorting based on complex network. *IEEE Signal Processing Letters* 28 (2021), 91–95. DOI: https://doi.org/10.1109/LSP.2020.3044259

[21] Breanne P. Christie, Derek M. Tat, Zachary T. Irwin, Vikash Gilja, Paul Nuyujukian, Justin D. Foster, Stephen I. Ryu, Krishna V. Shenoy, David E. Thompson, and Cynthia A. Chestek. 2014. Comparison of spike sorting and thresholding of voltage waveforms for intracortical brain-machine interface performance. *J Neural Eng* 12, 1 (2014), 016009.

[22] Zhaole Chu, Yongping Luo, Peiquan Jin, and Shouhong Wan. 2021. NVMSorting: Efficient sorting on non-volatile memory. In *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering* .

[23] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. DOI: https://doi.org/10.1109/4235.996017

[24] Mohammadreza Lalegani Dezaki, Saghi Hatami, Ali Zolfagharian, and Mahdi Bodaghi. 2022. A pneumatic conveyor robot for color detection and sorting. *Cognitive Robotics* 2 (2022), 60–72. DOI: https://doi.org/10.1016/j.cogr.2022.03.001

[25] Thanh Do, Goetz Graefe, and Jeffrey Naughton. 2023. Efficient sorting, duplicate removal, grouping, and aggregation. *ACM Transactions on Database Systems* 47, 4 (2023), 35 pages. DOI: https://doi.org/10.1145/3568027

[26] H. H. Draz, N. E. Elashker, and Mervat M. A. Mahmoud. 2023. Optimized algorithms and hardware implementation of median filter for image processing. *Circuits, Systems, and Signal Processing* 42 (2023), 5545–5558. DOI: https://doi.org/10.1007/s00034-023-02370-x

[27] Wei Fan and Albert Bifet. 2013. Mining big data: Current status, and forecast to the future. *SIGKDD Explor. Newsl.* 14, 2 (2013), 1–5. DOI: https://doi.org/10.1145/2481244.2481246

[28] 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. DOI: https://doi.org/10.1109/TC.2012.108

[29] Teng Gao, Lan Huang, Shang Gao, and Kangping Wang. 2024. SSA: A uniformly recursive bidirection-sequence systolic sorter array. *IEEE Transactions on Parallel and Distributed Systems* 35, 10 (2024), 1721–1734. DOI: https://doi.org/10.1109/TPDS.2024.3434332

[30] Surajeet Ghosh, Shaon Dasgupta, and Sanchita Saha Ray. 2019. A comparison-free hardware sorting engine. In *Proceedings of the 2019 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 586–591.

[31] Sarah Gibson, Jack W. Judy, and Dejan Marković. 2012. Spike sorting: The first step in decoding the brain: The first step in decoding the brain. *IEEE Signal Processing Magazine* 29, 1 (2012), 124–143. DOI: https://doi.org/10.1109/MSP.2011.941880

[32] M. D. Godfrey and D. F. Hendry. 1993. The computer as von Neumann planned it. *IEEE Annals of the History of Computing* 15, 1 (1993), 11–21. DOI: https://doi.org/10.1109/85.194088

[33] Goetz Graefe. 2006. Implementing sorting in database systems. *Association for Computing Machinery Computing Surveys* 38, 3 (2006), 10–es. DOI: https://doi.org/10.1145/1132960.1132964

[34] Di Guo, Huaping Liu, and Fuchun Sun. 2023. Audio–visual language instruction understanding for robotic sorting. *Robotics and Autonomous Systems* 159 (2023), 104271. DOI: https://doi.org/10.1016/j.robot.2022.104271

[35] Wenbo Guo and Shuguo Li. 2021. Fast binary counters and compressors generated by sorting network. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29, 6 (2021), 1220–1230. DOI: https://doi.org/10.1109/TVLSI.2021.3067010

[36] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *Proceedings of the 2018 IEEE/Association for Computing Machinery International Conference on Computer-Aided Design* . IEEE, 1–7.

[37] Lan Huang, Teng Gao, Yang Feng, and Kangping Wang. 2025. MCSSA: A stream-based multiconcurrency systolic sorting array combining merge tree. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 4 (2025), 1340–1353. DOI: https://doi.org/10.1109/TCAD.2024.3471903

[38] Andrey Ivanov, Dmitry Yarotsky, Maria Stoliarenko, and Alexey Frolov. 2018. Smart sorting in massive MIMO detection. In *Proceedings of the 2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications*. 1–6. DOI: https://doi.org/10.1109/WiMOB.2018.8589084

[39] Amir Hossein Jalilvand, Seyedeh Newsha Estiri, Samaneh Naderi, M. Hassan Najafi, and Mohsen Imani. 2022. A fast and low-cost comparison-free sorting engine with unary computing: Late breaking results. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1390–1391. DOI: https://doi.org/10.1145/3489517.3530615

[40] Amir Hossein Jalilvand, M. Hassan Najafi, and Mahdi Fazeli. 2020. Fuzzy-logic using unary bit-stream processing. In *Proceedings of the 2020 IEEE International Symposium on Circuits and Systems*. IEEE, 1–5.

[41] Parham Taghinia Jelodari, Mojtaba Parsa Kordasiabi, Samad Sheikhaei, and Behjat Forouzandeh. 2020. An O(1) time complexity sorting network for small number of inputs with hardware implementation. *Microprocessors and Microsystems* 77 (2020), 103203. DOI: https://doi.org/10.1016/j.micpro.2020.103203

[42] Uday A. Korat, Pratik Yadav, and Harshil Shah. 2017. An efficient hardware implementation of vector-based odd-even merge sorting. In *Proceedings of the IEEE 8th UEMCON*. https://doi.org/10.1109/UEMCON.2017.8249010

[43] C. Kotropoulos, M. Pappas, and I. Pitas. 2002. Sorting networks using L/sub p/ mean comparators for signal processing applications. *IEEE Transactions on Signal Processing* 50, 11 (2002), 2716–2729. DOI: https://doi.org/10.1109/TSP.2002.804069

[44] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. 2014. MAGIC–Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[45] Arezki Laga, Jalil Boukhobza, Frank Singhoff, and Michel Koskas. 2017. MONTRES: Merge ON-the-Run external sorting algorithm for large data volumes on SSD based storage systems. *IEEE Trans. Comput.* 66, 10 (2017), 1689–1702. DOI: https://doi.org/10.1109/TC.2017.2706678

[46] Marzieh Lenjani, Alif Ahmed, and Kevin Skadron. 2022. Pulley: An algorithm/hardware co-optimization for in-memory sorting. *IEEE Computer Architecture Letters* 21, 2 (2022), 109–112.

[47] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture* . IEEE, 556–569.

[48] 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*. 45–50.

[49] Shih-Hsiang Lin, Pei-Yin Chen, and Yu-Ning Lin. 2017. Hardware design of low-power high-throughput sorting unit. *IEEE Trans. Comput.* 66, 8 (2017), 1383–1395.

[50] Yang Liu, Yang Ou, Wenhan Chen, Zhiguang Chen, and Nong Xiao. 2023. LazySort: A customized sorting algorithm for non-volatile memory. *Information Sciences* 641 (2023), 119137. DOI: https://doi.org/https://doi.org/10.1016/j.ins.2023.119137

[51] Chia-Yu Lu and Chang-Ming Wu. 2011. A hardware design approach of sorting for FlexRay-based clock synchronization. In *2011 IEEE/SICE International Symposium on System Integration (SII'11)*. 1400–1405. https://doi.org/10.1109/SII.2011.6147654

[52] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. 2023. Faster sorting algorithms discovered using deep reinforcement learning. *Nature* 618, 7964 (2023), 257–263.

[53] Partha Mitra and Hemant Bokil. 2008. *Observed Brain Dynamics*. Oxford University Press, New York, NY.

[54] Guillermo Montesdeoca, Víctor Asanza, Kevin Chica, and Diego H. Peluffo-Ordóñez. 2022. Analysis of sorting algorithms using a WSN and environmental pollution data based on FPGA. In *Proceedings of the 2022 International Conference on Applied Electronics*. 1–4. DOI: https://doi.org/10.1109/AE54730.2022.9920090

[55] Mahmoud Moshref, Rizik Al-Sayyed, and Saleh Al-Sharaeh. 2021. An enhanced multi-objective non-dominated sorting genetic routing algorithm for improving the QoS in wireless sensor networks. *IEEE Access* 9 (2021), 149176–149195. DOI: https://doi.org/10.1109/ACCESS.2021.3122526

[56] J. I. Munro and M. S. Paterson. 1980. Selection and sorting with limited storage. *Theoretical Computer Science* 12, 3 (1980), 315–323. DOI: https://doi.org/10.1016/0304-3975(80)90061-4

[57] Mohammadhassan Najafi, Amir Hossein Jalilvand, and Mahdi Fazeli. 2021. Method and Architecture for Fuzzy-Logic Using Unary Processing. US Patent App. 17/340,834.

[58] M. Hassan Najafi, Devon Jenson, David J. Lilja, and Marc D. Riedel. 2019. Performing stochastic computation deterministically. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 12 (2019), 2925–2938. https://doi.org/10.1109/TVLSI.2019.2929354

[59] M. Hassan Najafi, David J. Lilja, Marc Riedel, and Kia Bazargan. 2017. Power and area efficient sorting networks using unary processing. In *2017 IEEE Intern. Conf. on Computer Design (ICCD'17)*. 125–128. DOI: https://doi.org/10.1109/ICCD.2017.27

[60] M. Hassan Najafi, David. J. Lilja, Marc D. Riedel, and Kia Bazargan. 2018. Low-cost sorting network circuits using unary processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 8 (2018), 1471–1480. DOI: https://doi.org/10.1109/TVLSI.2018.2822300

[61] M. Hassan Najafi, David J. Lilja, Marc D. Riedel, and Kia Bazargan. 2018. Low-cost sorting network circuits using unary processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 8 (2018), 1471–1480.

[62] Amin Norollah, Danesh Derafshi, Hakem Beitollahi, and Mahdi Fazeli. 2019. RTHS: A low-cost high-performance real-time hardware sorter, using a multidimensional sorting algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 7 (2019), 1601–1613.

[63] Max OrHai and Christof Teuscher. 2011. Spatial sorting algorithms for parallel computing in networks. In *Proceedings of the 2011 5th IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*. 73–78. DOI: https://doi.org/10.1109/SASOW.2011.10

[64] Sulin Pang, Jiaqi Wang, and Xiaoshuang Yi. 2022. Application of loan lost-linking customer path correlated index model and network sorting search algorithm based on big data environment. *Neural Comput. Appl.* 35, 3 (April 2022), 2129–2156. DOI: https://doi.org/10.1007/s00521-022-07189-2

[65] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. 2018. FLiMS: Fast lightweight merge sorter. In *Proceedings of the 2018 International Conference on Field-Programmable Technology*. IEEE, 78–85.

[66] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. 2020. An adaptable high-throughput FPGA merge sorter for accelerating database analytics. In *Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications*. 65–72. DOI: https://doi.org/10.1109/FPL50879.2020.00021

[67] M. L. Petrović and V. M. Milovanović. 2021. A chisel generator of parameterizable and runtime reconfigurable linear insertion streaming sorters. In *Proceedings of the 2021 IEEE 32nd International Conference on Microelectronics* . 251–254. DOI: https://doi.org/10.1109/MIEL52794.2021.9569153

[68] Ananth Krishna Prasad, Morteza Rezaalipour, Masoud Dehyadegari, and Mahdi Nazm Bojnordi. 2021. Memristive data ranking. In *Proceedings of the 2021 IEEE International Symposium on High Performance Computer Architecture* . Seoul, South Korea, 440–452.

[69] Preethi Preethi, K. G Mohan, K. Sudeendra Kumar, and K. K. Mahapatra. 2021. Low power sorters using clock gating. In *Proceedings of the 2021 IEEE International Symposium on Smart Electronic Systems*. 6–11. DOI: https://doi.org/10.1109/iSES52644.2021.00015

[70] Brady Prince, M. Hassan Najafi, and Bingzhe Li. 2023. Scalable low-cost sorting network with weighted bit-streams. In *Proceedings of the 2023 24th International Symposium on Quality Electronic Design*. IEEE, 1–6.

[71] Seth H. Pugsley, Arjun Deb, Rajeev Balasubramonian, and Feifei Li. 2015. Fixed-function hardware sorting accelerators for near data mapreduce execution. In *Proceedings of the 2015 33rd IEEE International Conference on Computer Design*. IEEE, New York, NY, USA, 439–442.

[72] Blanca C. López Ramírez, Giovanni Guzmán, Wadee Alhalabi, Nareli Cruz-Cortés, Miguel Torres-Ruiz, and Marco Moreno-Ibarra. 2018. On the usage of sorting networks to control greenhouse climatic factors. *International Journal of Distributed Sensor Networks* 14, 2 (2018), 1550147718756871. DOI: https://doi.org/10.1177/1550147718756871

[73] Sanchita Saha Ray and Surajeet Ghosh. 2023. k-Degree parallel comparison-free hardware sorter for complete sorting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 5 (2023), 1438–1449. DOI: https://doi.org/10.1109/TCAD.2022.3207689

[74] Artjom Rjabov. 2016. Hardware-based systems for partial sorting of streaming data. In *Proceedings of the 2016 15th Biennial Baltic Electronics Conference*. 59–62. DOI: https://doi.org/10.1109/BEC.2016.7743728

[75] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-performance adaptive merge tree sorting. In *Proceedings of the 2020 Association for Computing Machinery/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 282–294.

[76] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th IEEE/Association for Computing Machinery MICRO*. 273–287.

[77] Abu Salman Shaikat, Suraiya Akter, and Umme Salma. 2020. Computer vision based industrial robotic arm for sorting objects by color and height. *Journal of Engineering Advancements* 1, 04 (2020), 116–122. DOI: https://doi.org/10.38032/jea.2020.04.002

[78] Beining Shang, Richard Crowder, and Klaus-Peter Zauner. 2014. Swarm behavioral sorting based on robotic hardware variation. In *Proceedings of the 2014 4th International Conference On Simulation And Modeling Methodologies, Technologies and Applications*. 631–636. DOI: https://doi.org/10.5220/0005111006310636

[79] Shyue-Horng Shiau and Chang-Biau Yang. 2000. A fast sorting algorithm and its generalization on broadcast communications. In *Proceedings of the Computing and Combinatorics*. Ding-Zhu Du, Peter Eades, Vladimir Estivill-Castro, Xuemin Lin, and Arun Sharma (Eds.), 252–261.

[80] Shyue-Horng Shiau and Chang-Biau Yang. 2006. Generalization of sorting in single hop wireless networks. *IEICE Trans. Inf. Syst.* 89-D, (2006), 1432–1439. Retrieved from https://api.semanticscholar.org/CorpusID:8143837

[81] Shahriar Shirvani Moghaddam and Kiaksar Shirvani Moghaddam. 2023. A threshold-based sorting algorithm for dense wireless sensor systems and communication networks. *IET Wireless Sensor Systems* 13, 2 (2023), 37–47.

[82] Shahriar Shirvani Moghaddam and Kiaksar Shirvani Moghaddam. 2023. A threshold-based sorting algorithm for dense wireless sensor systems and communication networks. *IET Wireless Sensor Systems* 13, 2 (2023), 37–47. DOI: https://doi.org/10.1049/wss2.12048

[83] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. 2018. Survey of GPU based sorting algorithms. *Int. J. Parallel Program.* 46, 6 (2018), 1017–1034. DOI: https://doi.org/10.1007/s10766-017-0502-5

[84] Mitali Singh and Viktor K. Prasanna. 2003. Energy-optimal and energy-balanced sorting in a single-hop wireless sensor network. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications* DOI: https://doi.org/10.1109/PERCOM.2003.1192726

[85] Iouliia Skliarova. 2022. A survey of network-based hardware accelerators. *Electronics* 11, 7 (2022). DOI: https://doi.org/10.3390/electronics11071029

[86] Gorrepati Chaithra Sri, S. Arpitha Kopparthi Veera Hanuma, Nuthalapati Harshita, and Sonali Agrawal. 2022. An efficient quasi comparison-free bidirectional architecture for sorting algorithm. In *Proceedings of the 2022 IEEE 3rd Global Conference for Advancement in Technology*. 1–8. DOI: https://doi.org/10.1109/GCAT55367.2022.9972111

[87] J. Subramaniam, J. K. Raju, and D. Ebenezer. 2017. Fast median-finding word comparator array. *Elect. Lett.* 53, 21 (2017), 1402–1404. DOI: https://doi.org/10.1049/el.2017.1811

[88] Yifeng Tang, Dechang Huang, Rong Li, and Zhaodi Huang. 2022. A non-dominated sorting genetic algorithm based on voronoi diagram for deployment of wireless sensor networks on 3-D terrains. *Electronics* 11, 19 (2022). DOI: https://doi.org/10.3390/electronics11193024

[89] W. Teich and H. Ch. Zeidler. 1983. Data handling and dedicated hardware for the sort problem. In *Proceedings of the Database Machines*. H.-O. Leilich and M. Missikoff (Eds.), Springer, Berlin, 205–226.

[90] Pedro Trancoso. 2015. Moving to memoryland: In-memory computation for existing applications. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. 1–6.

[91] Daniel Valencia and Amirhossein Alimohammad. 2019. An efficient hardware architecture for template matching-based spike sorting. *IEEE Transactions on Biomedical Circuits and Systems* 13, 3 (2019), 481–492. DOI: https://doi.org/10.1109/TBCAS.2019.2907882

[92] Stratis D. Viglas. 2014. Write-limited sorts and joins for persistent memory. *Proceedings of the VLDB Endowment* 7, 5 (2014), 413–424.

[93] Jiawei Wang, Changbo Hou, and Fuxin Qu. 2017. Multi-threshold fuzzy clustering sorting algorithm. In *Proceedings of the 2017 Progress In Electromagnetics Research Symposium*. 889–892. DOI: https://doi.org/10.1109/PIERS.2017.8261869

[94] Chin-Hsien Wu and Kuo-Yi Huang. 2015. Data sorting in flash memory. *Association for Computing Machinery Transactions on Storage* 11, 2 (2015), 1–25.

[95] Di Yan, Wei-Xing Wang, Lei Zuo, and Xiao-Wei Zhang. 2021. A novel scheme for real-time max/min-set-selection sorters on FPGA. *IEEE Transactions on Circuits and Systems II: Express Briefs* 68, 7 (2021), 2665–2669. DOI: https://doi.org/10.1109/TCSII.2021.3058245

[96] Myungchul Yoon. 2022. A novel architecture of asynchronous sorting engine module for ASIC design. *Journal of Semiconductor Technology and Science* 22, 4 (2022), 224–233.

[97] Bo Yu, Terrence Mak, Xiangyu Li, Fei Xia, Alexandre Yakovlev, Yihe Sun, and Chi-Sang Poon. 2011. Real-time FPGA-based multichannel spike sorting using hebbian eigenfilters. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 1, 4 (2011), 502–515. DOI: https://doi.org/10.1109/JETCAS.2012.2183430

[98] Lianfeng Yu, Zhaokun Jing, Yuchao Yang, and Yaoyu Tao. 2022. Fast and scalable memristive in-memory sorting with column-skipping algorithm. In *Proceedings of the 2022 IEEE International Symposium on Circuits and Systems* . IEEE, 590–594.

[99] Bilgiday Yuce, H. Fatih Ugurdag, Sezer Gören, and Gunhan Dundar. 2013. A fast circuit topology for finding the maximum of N k-bit numbers. In *Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic*. 59–66. DOI: https://doi.org/10.1109/ARITH.2013.35

[100] Ali Zafari, Atefeh Khoshkhahtinat, Piyush Mehta, Mohammad Saeed Ebrahimi Saadabadi, Mohammad Akyash, and Nasser M Nasrabadi. 2023. Frequency disentangled features in neural image compression. In *Proceedings of the 2023 IEEE International Conference on Image Processing*. IEEE, 2815–2819.

[101] Fan Zhang, Shaahin Angizi, and Deliang Fan. 2021. Max-PIM: Fast and efficient max/min searching in DRAM. In *Proceedings of the 58th Design Automation Conference DAC*. San Francisco, California, 211–216. DOI: https://doi.org/10.1109/DAC18074.2021.9586096

[102] Tim Zhang, Corey Lammie, Mostafa Rahimi Azghadi, Amirali Amirsoleimani, Majid Ahmadi, and Roman Genov. 2022. Toward a formalized approach for spike sorting algorithms and hardware evaluation. *2022 IEEE MWSCAS* (2022). DOI: https://doi.org/10.1109/mwscas54063.2022.9859357

[103] Xiaojian Zhang, Hongyin Zhang, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2023. Backward-sort for time series in apache IoTDB. In *Proceedings of the 2023 IEEE 39th Intern. Conf. on Data Engineering*. 3196–3208. DOI: https://doi.org/10.1109/ICDE55515.2023.00245

[104] Farzaneh Zokaee, Fan Chen, Guangyu Sun, and Lei Jiang. 2022. Sky-Sorter: A processing-in-memory architecture for large-scale sorting. *IEEE Trans. Comput.* 72, 2 (2022), 480–493.

[105] Marcela Zuluaga, Peter Milder, and Markus Püschel. 2016. Streaming sorting networks. *Association for Computing Machinery Transactions on Design Automation of Electronic Systems* 21, 4 (2016), 30 pages. DOI: https://doi.org/10.1145/2854150

Received 29 May 2024; revised 24 January 2025; accepted 5 April 2025