OnSort: An $\mathcal{O}(n)$ Comparison-Free Sorter for Large-Scale Dataset With Parallel Prefetching and Sparse-Aware Mechanism

Muxuan Gao, Juntao Jiang, Shuangming Lei, Huifeng Wu, *Member, IEEE*, Jun Chen, and Yong Liu, *Member, IEEE*

***Abstract*—This brief proposes OnSort, a parallel comparison-free sorting architecture with $\mathcal{O}(n)$ time complexity, utilizing the SRAM structure to support large-scale datasets efficiently. The performance of existing comparison-free sorters is limited by uneven value distribution and variable element numbers. To address these issues, we introduce a parallel prefetching strategy to accelerate the indexing process and a sparse-aware mechanism to narrow the indexing search range. Furthermore, OnSort implements streaming execution through a pipelined design, thereby optimizing the previously overlooked latency of the counting phase. Experimental results show that, under the configuration of sorting 65,536 16-bit data elements, OnSort achieves a 1.97× speedup and a 22.6× throughput-to-area ratio compared to the existing design. The source code is available at https://github.com/gmx-hub/OnSort.**

***Index Terms*—Comparison-free sort, large-scale sorting, hardware acceleration, energy-efficient design, streaming execution.**

I. Introduction

Sorting, as one of the most fundamental operations in the field of computer science, is widely used in various applications such as database management [1], computer graphics [2], image processing [3], and scientific computing [4]. Despite years of optimization, traditional software-based sorting algorithms [5], [6], [7] still face performance bottlenecks, especially when handling large datasets and low-latency requirements. Hardware acceleration technologies have emerged as a promising approach to overcoming these limitations.

Received 5 March 2025; revised 21 April 2025; accepted 14 May 2025. Date of publication 16 May 2025; date of current version 1 July 2025. This work was supported in part by the National Natural Science Foundation of China under Grant U21A20484, and in part by the Zhejiang Provincial Natural Science Foundation of China under Grant LQN25F030018. This brief was recommended by Associate Editor C. Meinhardt. *(Corresponding authors: Jun Chen; Yong Liu.)*

Muxuan Gao, Juntao Jiang, Shuangming Lei, and Yong Liu are with the College of Control Science and Engineering, Zhejiang University, Hangzhou 310027, China (e-mail: muxuangao@zju.edu.cn; juntaojiang@zju.edu.cn; shuangminglei@zju.edu.cn; yongliu@iipc.zju.edu.cn).

Huifeng Wu is with the Institute of Intelligent and Software Technology, Hangzhou Dianzi University, Hangzhou 310018, China (e-mail: whf@hdu.edu.cn).

Jun Chen is with the National Special Education Resource Center for Children With Autism, Zhejiang Normal University, Hangzhou 311231, China, and also with the School of Computer Science and Technology, Zhejiang Normal University, Jinhua 321004, China (e-mail: junc.change@gmail.com).

Digital Object Identifier 10.1109/TCSII.2025.3570797

Hardware sorting architectures can be classified into comparison-based and comparison-free methods. For the former, bitonic sorting architectures and their variants [8], [9], [10] achieve high performance through parallel streaming execution. However, the performance gains come at the cost of area, as their implementation relies on many comparators and on-chip storage, making them unsuitable for large-scale datasets. Merge tree-based architectures [11], [12], [13] offer better area efficiency by using smaller bitonic networks to merge pre-sorted data bundles, but this leads to performance degradation on large-scale datasets.

For comparison-free methods, [14] proposed a radix sort acceleration architecture integrated into GPUs. However, it suffers from poor scalability and requires a large amount of memory. Reference [15] introduces a comparison-free sorting algorithm with time complexity of $\mathcal{O}(n)$, ingeniously transforming the sorting task into a classification operation. Reference [16] improves the algorithm in [15] by introducing a bidirectional sorting structure, which enables parallel sorting of the high-index and low-index parts to enhance performance. Although [15] and [16] achieve $\mathcal{O}(n)$ time complexity, they still have many limitations. First, these architectures lack practical applicability for large-scale datasets due to the area overhead caused by the large number of counting and indexing registers. Second, their performance is suboptimal due to the lack of full streaming support and limited parallelism. Finally, uneven numerical distribution and variable element numbers lead to inefficient performance. For instance, the actual sorting cycles of [15] and [16] range from $2N$ to $3N$ and from $1.5N$ to $2N + (N/2) - 2$, depending on the value distribution.

This brief proposes OnSort, a sorting architecture for large-scale datasets that combines the advantages of comparison-free sorters [15], [16] while overcoming their limitations. The main contributions can be described as follows:

- We propose OnSort, a parallel hardware sorting architecture with $\mathcal{O}(n)$ time complexity, utilizing the SRAM structure to support large-scale datasets efficiently.
- We introduce a parallel prefetching strategy to accelerate the indexing process and a sparse-aware mechanism to narrow the indexing search range, making the performance largely independent of the value distribution.
- We further propose a high-performance version that supports streaming execution through a pipelined design, with the actual sorting cycles approximating N.

Algorithm 1: Counting Phase of Comparison-Free Sort

```
Input: Unsorted Array (UA): list
Output: Count Array (CA): list
begin
    Initialize CA to zero;
    for $i \in [0, N)$ do
        CA[UA[$i$]] = CA[UA[$i$]] + 1;
    end
end
```

Algorithm 2: Indexing Phase of Comparison-Free Sort

```
Input: Count Array (CA): list
Output: Sorted Array (SA): list
begin
    $i$, $oidx$ = 0;
    while $i < 2^K$ do
        if $CA[i] > 0$ then
            SA[$oidx$] = $i$;
            $oidx = oidx + 1$;
            CA[$i$] = CA[$i$] -1;
        end
        else if $CA[i] == 0$ then
            $i = i + 1$;
        end
    end
end
```

II. Comparison-Free Sorting Algorithm

A. Comparison-Free Sort

Assume the input is an unsorted array (UA) of N elements, where each element is K-bit wide with values ranging from 0 to $2^K - 1$, and at most L duplicate elements. The comparison-free sort is divided into 2 phases: counting and indexing.

1) *Counting*: In this stage, a count array (CA) is obtained with 2^K elements, each with a width of $log_2(L)+1$ to represent the occurrence count of its corresponding input element. Each element in the UA is used as an address to query the CA, and the queried count is incremented by 1. The counting phase is illustrated in Algorithm 1.

2) *Indexing*: In this stage, the sorted array (SA) is produced in descending or ascending order according to the configuration. For simplicity, we take the ascending order as an example. CA is indexed in the range from 0 to $2^K - 1$. If the indexed count is nonzero, the address is recorded in the SA and that count is decremented by 1. If the indexed count is zero, indicating that this address does not exist in the UA, it is skipped. The indexing phase is illustrated in Algorithm 2.

B. Performance Analysis

Assuming the comparison-free algorithm is configured to sort a maximum of N elements, where $N = 2^K$, and that one indexing process in the Algorithm 2 takes one clock cycle. There are two scenarios in the algorithm that can lead to inefficient performance.

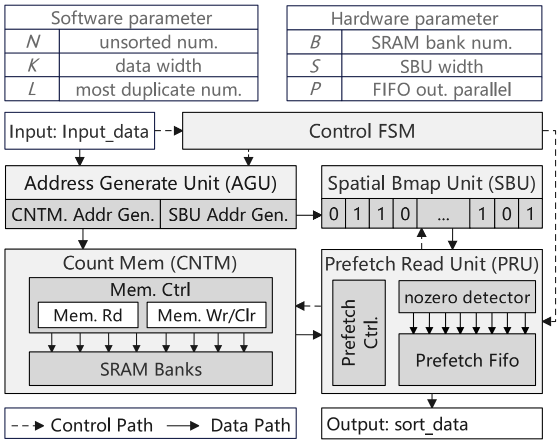

Fig. 1. The architecture and parameters of resource-efficient design.

1) Uneven distribution of sorted values. The performance of the comparison-free algorithm depends on the data distribution. The best case occurs when no duplicate values exist among the N elements. Each count in CA is non-zero, and all indices are valid, allowing the indexing phase to be completed in N cycles. The worst case is that all N elements have the same value. Only one count in CA is non-zero, while the rest are zero, leading to many invalid indices. We define the regions in CA where the count is 0 as sparse regions. Specifically, in Algorithm 2, lines 4-10 iterate N times, and lines 12-14 iterate $N-1$ times to handle the indexing of sparse regions, resulting in the indexing phase consuming $2N - 1$ cycles. Considering the counting and indexing phases, the sorting cycles range from $2N$ to $3N-1$. In general, sparse regions in CA result in performance degradation.

2) The variability in the number of elements. Variability refers to the difference in the number of elements sorted in each sorting task. In hardware design, it is necessary to support the maximum number of elements an application might encounter, denoted as N. In practice, the number of elements in a sorting task ranges from 1 to N. The sorting cycles of a comparison-free algorithm do not scale linearly with the number of elements. For example, if the input consists of N distinct elements, the indexing phase requires N cycles. However, if the input contains only two elements, the indexing phase still requires N cycles due to invalid indices. Therefore, variability in the sorting task leads to suboptimal performance.

III. Acceleration Structure Implementation

This section presents the hardware architecture of OnSort. We propose a resource-efficient design to address performance inefficiency and support larger-scale datasets. Furthermore, we introduce a high-throughput version that enables streaming execution to enhance performance. The details are provided in the following subsections.

A. Resource-Efficient (RE) Design

The resource-efficient (RE) architecture of OnSort is shown in Fig. 1. The components of OnSort are described as follows.

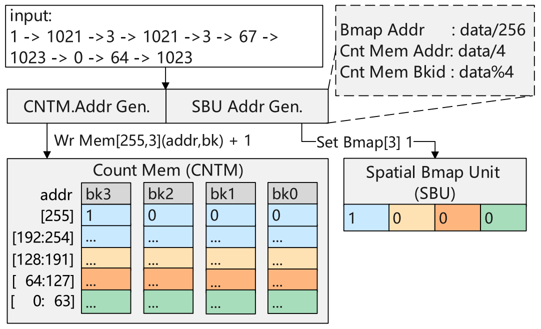

Fig. 2. An example illustrating the operation of counting phase.

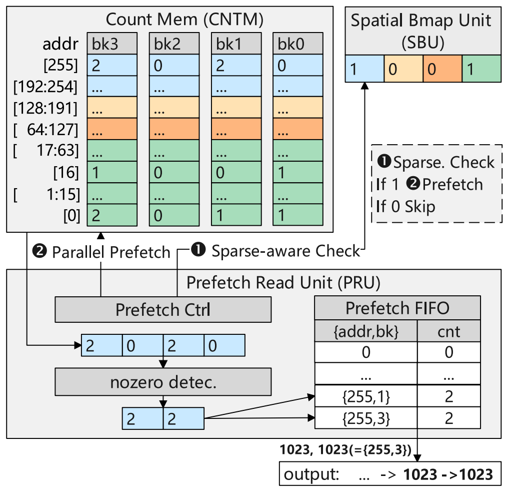

Fig. 3. An example illustrating the operation of indexing phase.

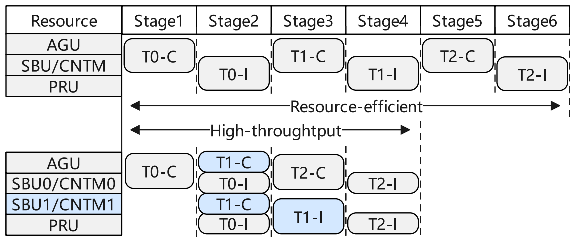

Fig. 4. The pipeline design of OnSort. TX-C and TX-I represent the counting and indexing phases of different sorting tasks, respectively.

1) Control Finite State Machine (FSM): It controls the switching of the counting and indexing phases.

2) Count Mem (CNTM): CNTM is designed with an SRAM-based structure to support large-scale datasets, as SRAM features a moderate area and power growth rate. It consists of B banks, each of which is a dual-port SRAM. The depth of each bank is $2^K/B$, with a width of $log2(L) + 1$. In the counting phase, input elements are used as addresses to read data from the memory, which is then incremented by one and written back. In the indexing phase, CNTM is accessed in parallel with a degree of B to enhance performance, followed by a post-read clearing operation.

3) Spatial Bmap Unit (SBU): The SBU is designed to support a sparse-aware mechanism. The SBU is implemented as a bitmap with a bit width of S. The entire value range 2^K is divided into several segments. Each bit records whether the corresponding data segment is a sparse region. A bit value of 0 indicates that all counts in the corresponding data segment are 0, and its indexing can be skipped.

4) Address Generate Unit (AGU): The AGU is responsible for generating addresses for SBU and CNTM.

5) Prefetch Read Unit (PRU): PRU is designed to accelerate the indexing process through parallel prefetching. The prefetch control unit accesses CNTM with a parallelism of B. The retrieved data is processed by a nonzero value detector to eliminate bubbles, and nonzero values along with their addresses are stored in the first-in-first-out (FIFO). This process continues until the FIFO is full, effectively removing invalid indices caused by the sparse region. Meanwhile, sorted data is output from the FIFO with a parallelism degree of P.

B. Workflow of Sorting Process

Here is an example illustrating how the comparison-free sorting algorithm is implemented in OnSort. Assuming the hardware configuration: $N = 1024$, $K = 10$, $B = 4$, $P = 2$, $S = 256$. As shown in Fig. 2, in the counting phase, AGU generates the addresses for CNTM and SBU based on the input element (1023). SBU [3] is set to 1, indicating that CNTM [255:192] is not a sparse region. CNTM [255, 3] is incremented by 1, recording that 1023 has appeared once. In this way, the input array [1, 1021, 3, 1021, 3, 67, 1023, 0, 64, 1023] is stored in CNTM.

Once all input elements are counted, the indexing phase is switched via the Control FSM. As shown in Fig. 3, PRU iteratively queries the SBU and CNTM in the indexing phase. The indexing addresses for SBU and PRU range from 3 to 0 and 255 to 0, respectively. We employ a sparse-aware mechanism to identify sparse regions and skip their indexing, thereby narrowing the indexing range. In the first step, PRU checks SBU. If an SBU entry is enabled, the corresponding addresses are indexed from CNTM in the second step. Conversely, the indexing of the CNTM will be skipped. For example, CNTM [255:192] is retrieved because SBU [3] is enabled, while the indexing of CNTM [191:128] is skipped because SBU [2] is disabled. This approach allows for coarse-grained avoidance of accessing invalid regions. Each access to CNTM is performed with a parallelism of B. A non-zero detection circuit filters out meaningful values from the retrieved data, which are then stored in the FIFO along with their corresponding addresses. For example, invalid values in CNTM [255] are removed, and the sparse region CNTM [192:254] is rapidly indexed with a parallelism of 4. Meanwhile, the addresses and bankid are output count times with a parallelism of 2.

C. High Throughput Design (HT)

As demonstrated in Algorithm 1 and Algorithm 2, data dependencies exist between the counting and indexing phases, which pose challenges for streaming execution and lead to suboptimal performance. We propose a high-throughput version by duplicating modules such as CNTM and SBU to eliminate data dependencies. The FSM controls switching hardware resources through multiplexers according to the

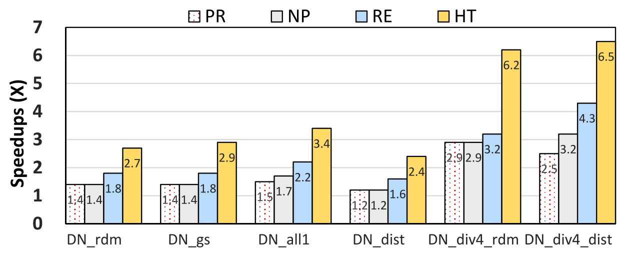

Fig. 5. The ablation study of OnSort: speedups of variants against the naive.

TABLE I
THE COMPARISON OF AREA, POWER AND MAXIMUM FREQUENCY

Design		$N(=2^K)$			
		1024	8192	16384	65536
Area (mm^2)	[15]	0.12	1.25	2.71	9.83
	[16]	0.09	0.88	1.75	6.48
	[10]	**0.03**	0.53	-	-
	Our(RE)	0.04	**0.07**	**0.10**	**0.29**
	Our(HT)	0.06	0.11	0.17	0.57
Power (W)	[15]	0.25	1.48	3.11	11.6
	[16]	0.16	0.97	2.01	7.56
	[10]	0.05	0.70	-	-
	Our(RE)	**0.05**	**0.09**	**0.11**	**0.24**
	Our(HT)	0.08	0.18	0.20	0.46
F_{max} (GHz)	[15]	**1.64**	**1.51**	**1.48**	1.18
	[16]	1.62	1.49	1.47	1.18
	Our(RE)	1.54	1.47	1.45	**1.43**
	Our(HT)	1.51	1.43	1.41	1.39

task phase. This approach enables a pipeline design for the counting and indexing stages across multiple tasks, as shown in Fig. 4. The counting latency of one task can be hidden by the indexing phase of the previous task. This optimization becomes more impactful when handling many tasks, as it helps amortize the overhead of the epilogue and prologue. However, this improvement comes with the trade-off of area overhead.

IV. EXPERIMENT AND RESULTS

A. Evaluation Settings

Hardware Implementation. We design a cycle-accurate simulator to evaluate the performance of OnSort and verify its correctness through a Verilog RTL implementation. The simulator's performance aligns perfectly with the RTL implementation. We use the Synopsys Design Compiler to estimate the chip area and total power consumption under the TSMC 28nm process.

Datasets. A diverse set of datasets is employed to evaluate OnSort under various conditions. These datasets include four types of data distributions: DN_{gs, rdm, dist, all1}, where N represents the maximum number of elements supported by the hardware, and gs and rdm refer to Gaussian and random distributions, respectively. DN_dist and DN_all1 represent datasets where all element values are distinct and all element values are the same, respectively. Additionally, we introduced variability to simulate real-world applications, such as DN_div2 and DN_div4, representing datasets containing $N/2$ and $N/4$ elements. The final dataset is a combination of these data distributions and variability.

TABLE II
THE COMPARISON OF PERFORMANCE

Design		$N(=2^K)$			
		1024	8192	16384	65536
Time (μs)	[15]	1.48	12.8	26.2	131.4
	[16]	1.07	9.26	18.8	93.2
	[10]	0.95	8.15	16.6	83.3
	Our(RE)	1.00	8.37	16.9	68.7
	Our(HT)	**0.68**	**5.76**	**11.6**	**47.3**
Throughput / area (GB/s/mm^2)	[15]	6.73(1.00)	0.77(1.00)	0.38(1.00)	0.10(1.00)
	[16]	12.4(1.84)	1.52(1.97)	0.81(2.13)	0.20(2.00)
	[10]	**41.8(6.21)**	2.87(3.72)	-	-
	Our(RE)	29.8(4.43)	**21.2(27.5)**	**15.7(41.3)**	**6.12(61.2)**
	Our(HT)	29.1(4.32)	19.6(25.5)	13.5(35.5)	4.52(45.2)

B. Ablation Studies

To demonstrate that the components of the proposed architecture enhance performance in various scenarios, we conducted ablation studies. The accelerator is configured with parameters $N = 65536$, $B = 8$, $S = 256$, and $K = 16$. Our naive version is a resource-efficient architecture without parallel prefetching, sparse-aware mechanism, and parallel output capabilities. We evaluate four variants of OnSort to decouple the contribution proposed in this brief. (1) PR: includes parallel prefetching strategy without sparse-aware mechanism and parallel output capabilities. (2) NP: includes parallel prefetching strategy and sparse-aware mechanism without parallel output capabilities. (3) RE: a fully functional resource-efficient architecture with P set to 2. (4) HT: the high-throughput architecture.

Fig. 5 presents the speedup comparison of different variants across multiple datasets. The PR variant achieves an average speedup of 1.81× over the baseline. These optimizations stem from accelerating the indexing phase via parallel prefetching, which allows skipping fine-grained sparse regions. Additionally, on the DN_all1 and DN_div4_dist datasets, NP outperforms PR by 1.13× and 1.28×, respectively. This is because these datasets contain large sparse regions, and the sparse-aware mechanism effectively narrows the indexing range. The parallel output capability of FIFO amplifies the performance gains from our parallelized prefetch mechanism, resulting in a 1.26× speedup for RE compared to NP. Notably, the NP and RE versions only accelerate the indexing phase without optimizing the counting phase. By interleaving counting latency with the preceding indexing phase via inter-task pipelining, HT further improves performance by 1.62× over RE. HT ultimately achieves 4.02× speedup compared to the baseline.

C. Performance Comparison

To demonstrate the superiority of our architecture, we compare the performance of OnSort with the architecture presented in [10], [15], and [16]. Our architecture is configured with parameters $B = 8$, $S = 256$, $P = 2$, and $L = 1024$. Reference [15] is a comparison-free sorter with $\mathcal{O}(n)$ time complexity, and [16] introduces bidirectional structure to improve this architecture. Reference [10] is the state-of-the-art design, constructed with a bitonic sorter and numerous bidirectional insertion sorting units (BISU), achieving higher energy efficiency and area efficiency. For the experiments involving large-scale datasets that are unavailable in this brief, we reproduce and synthesize [15] and [16] under the same setting and estimate the results for [10] based on the reported data. These sorters are implemented using a large number of registers or comparators. References [15] and [16] require at least $3N$ and $2N$ registers, respectively, while [10] requires $N/2$ BISUs, each containing several registers and comparators. As a result, these sorters exhibit poor area and power performance on large-scale datasets, as shown in Table I. In contrast, our SRAM-based design features moderate area and power growth rates. Compared to [10] and [16], RE achieves up to 95% and 87% area reduction and up to 97% and 87% power reduction, respectively.

We compare OnSort with the state-of-the-art design across four datasets in Table II. Reference [10] achieves a sorting cycle of $1.5N$, while [15] and [16] range from $2N$ to $3N$ and $1.5N$ to $2N+(N/2)-2$, depending on value distribution. This dependency arises from redundant indexing caused by sparse regions in CA, preventing optimal sorting cycles in practice. To address this, OnSort's sparse-aware mechanism checks signals marking sparse regions, skipping coarse-grained invalid indexing. Parallel prefetching accelerates indexing, while nonzero detection eliminates redundant fine-grained indexing by preventing bubbles. With these optimizations, RE and HT achieve sorting cycles of approximately $1.5N$ and N on the DN_rdm dataset, delivering the shortest sorting time. To evaluate sorting time and area, we present the throughput-area ratio in Table II, and the maximum frequency (F_{max}) during evaluation is shown in Table I. Since the frequency for the corresponding configuration is not provided in [10], we set it to the highest among the designs (which may exceed the actual frequency). The average throughput-area ratios of RE and HT are 16.6× and 13.6× higher than [16], respectively.

We further compare the sorting cycles of [15] and [16] across datasets with different distributions and element numbers, as shown in Table III. As previously analyzed, [15] and [16] exhibit significant performance variations across different data distributions, whereas our design is largely independent of data distribution, achieving a sorting cycle approximating N. Moreover, [15] and [16] do not exhibit linear optimization in sorting cycles for DN_div2_rdm and DN_div4_rdm. This is because they still index the entire search space despite the significant redundancy within it. In contrast, the sparse-aware mechanism narrows the indexing range, while parallel prefetching accelerates the remaining indexing process. As a result, RE achieves 1.94× and 2.44× speedups for DN_div2_rdm and DN_div4_rdm, respectively, compared to [16]. In addition, previous designs overlooked the optimization of the counting phase, while HT hides this phase's latency through streaming execution, achieving the fewest sorting cycles across all listed datasets.

TABLE III
THE COMPARISON OF SORTING CYCLES (IN THOUSANDS)

Design	Dataset ($N=65536$, $K=16$)				
	gs	all1	dist	rdm	
	DN	DN	DN	DN_div2	DN_div4
[15]	163(2.47)	197(2.98)	131(1.98)	105(3.18)	83(4.61)
[16]	115(1.74)	131(1.98)	99(1.50)	69(2.09)	50(2.78)
Our(RE)	102(1.55)	99(1.50)	99(1.50)	54(1.64)	34(1.89)
Our(HT)	**66(1.00)**	**66(1.00)**	**66(1.00)**	**33(1.00)**	**18(1.00)**

V. CONCLUSION

In this brief, we propose OnSort, an SRAM-based, comparator-free sorting architecture with $\mathcal{O}(n)$ time complexity for large-scale datasets. We introduce a parallel prefetching strategy to accelerate the indexing process and a sparse-aware mechanism to narrow the indexing search range. Additionally, we enable full streaming execution through a pipeline design. Experimental results demonstrate that our architecture achieves high performance with minimal area overhead.

REFERENCES

[1] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2014, pp. 151–160.

[2] Z. Li, T. Wang, and Y. Deng, "Fully parallel kd-tree construction for real-time ray tracing," in *Proc. 18th Meeting ACM SIGGRAPH Symp. Interact. 3D Graph. Games*, 2014, p. 159.

[3] A. Gabiger-Rose, M. Kube, R. Weigel, and R. Rose, "An FPGA-based fully synchronized design of a bilateral filter for real-time image denoising," *IEEE Trans. Ind. Electron.*, vol. 61, no. 8, pp. 4093–4104, Aug. 2014.

[4] L. Njejimana et al., "Design of a real-time FPGA-based data acquisition architecture for the LabPET II: An APD-based scanner dedicated to small animal PET imaging," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 5, pp. 3633–3638, Oct. 2013.

[5] K. Sujatha, P. N. Rao, A. A. Rao, V. Sastry, V. Praneeta, and R. K. Bharat, "Multicore parallel processing concepts for effective sorting and searching," in *Proc. Int. Conf. Signal Process. Commun. Eng. Syst.*, 2015, pp. 162–166.

[6] G. Capannini, F. Silvestri, and R. Baraglia, "Sorting on GPUs for large scale datasets: A thorough comparison," *Inf. Process. Manag.*, vol. 48, no. 5, pp. 903–917, 2012.

[7] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-sort: A new parallel sorting algorithm for multi-core SIMD processors," in *Proc. 16th Int. Conf. Parallel Architect. Compil. Techn. (PACT)*, 2007, pp. 189–198.

[8] R. Chen and V. K. Prasanna, "Computer generation of high throughput and memory efficient sorting designs on FPGA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3100–3113, Nov. 2017.

[9] A. Norollah, D. Derafshi, H. Beitollahi, and M. Fazeli, "RTHS: A low-cost high-performance real-time hardware sorter, using a multidimensional sorting algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 7, pp. 1601–1613, Jul. 2019.

[10] Y.-R. Chen, C.-C. Ho, W.-T. Chen, and P.-Y. Chen, "A low-cost pipelined architecture based on a hybrid sorting algorithm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 71, no. 2, pp. 717–730, Feb. 2024.

[11] P. Papaphilippou, C. Brooks, and W. Luk, "An adaptable high-throughput FPGA merge sorter for accelerating database analytics," in *Proc. 30th Int. Conf. Field-Program. Logic Appl. (FPL)*, 2020, pp. 65–72.

[12] S. Mashimo, T. Van Chu, and K. Kise, "High-performance hardware merge sorter," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2017, pp. 1–8.

[13] H. W. Oh, J. Park, and S. E. Lee, "DL-Sort: A hybrid approach to scalable hardware-accelerated fully-streaming sorting," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 71, no. 5, pp. 2549–2553, May 2024.

[14] X. Liu, S. Li, K. Fang, Y. Ni, Z. Li, and Y. Deng, "RadixBoost: A hardware acceleration structure for scalable radix sort on graphic processors," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2015, pp. 1174–1177.

[15] S. Abdel-Hafeez and A. Gordon-Ross, "An efficient O (N) comparison-free sorting algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 6, pp. 1930–1942, Jun. 2017.

[16] W.-T. Chen, R.-D. Chen, P.-Y. Chen, and Y.-C. Hsiao, "A high-performance bidirectional architecture for the quasi-comparison-free sorting algorithm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 4, pp. 1493–1506, Apr. 2021.