NASCENT2: Generic Near-Storage Sort Accelerator for Data Analytics on SmartSSD

SAHAND SALAMAT, UC San Diego
HUI ZHANG and YANG SEOK KI, Samsung Semiconductor Inc.
TAJANA ROSING, UC San Diego

As the size of data generated every day grows dramatically, the computational bottleneck of computer systems has shifted toward storage devices. The interface between the storage and the computational platforms has become the main limitation due to its limited bandwidth, which does not scale when the number of storage devices increases. Interconnect networks do not provide simultaneous access to all storage devices and thus limit the performance of the system when executing independent operations on different storage devices. Offloading the computations to the storage devices eliminates the burden of data transfer from the interconnects. Near-storage computing offloads a portion of computations to the storage devices to accelerate big data applications. In this article, we propose a generic near-storage sort accelerator for data analytics, NASCENT2, which utilizes Samsung SmartSSD, an NVMe flash drive with an on-board FPGA chip that processes data in situ.

NASCENT2 consists of dictionary decoder, sort, and shuffle FPGA-based accelerators to support sorting database tables based on a key column with any arbitrary data type. It exploits data partitioning applied by data processing management systems, such as SparkSQL, to breakdown the sort operations on colossal tables to multiple sort operations on smaller tables. NASCENT2 generic sort provides 2× speedup and 15.2× energy efficiency improvement as compared to the CPU baseline. It moreover considers the specifications of the SmartSSD (e.g., the FPGA resources, interconnect network, and solid-state drive bandwidth) to increase the scalability of computer systems as the number of storage devices increases. With 12 SmartSSDs, NASCENT2 is 9.9× (137.2×) faster and 7.3× (119.2×) more energy efficient in sorting the largest tables of TPCC and TPCH benchmarks than the FPGA (CPU) baseline.

CCS Concepts: • **Hardware** → **Hardware accelerators**; *Emerging architectures*; • **Computer systems organization** → Reconfigurable computing;

ACM Reference format:
Sahand Salamat, Hui Zhang, Yang Seok Ki, and Tajana Rosing. 2022. NASCENT2: Generic Near-Storage Sort Accelerator for Data Analytics on SmartSSD. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 16 (January 2022), 29 pages.
https://doi.org/10.1145/3472769

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC Global Research Collaboration (GRC) grant, and also NSF grants 1730158, 1911095, 2003279, and 1826967.
Authors' addresses: S. Salamat and T. Rosing, CSE Department, UC San Diego, La Jolla, CA 92093; emails: {sasalama, tajana}@ucsd.edu; H. Zhang and Y. Seok Ki, Samsung Semiconductor Inc., San Jose, CA 95134; emails: {w.hzhang86, yangseok.ki}@samsung.com.

1936-7406/2022/01-ART16 $15.00
https://doi.org/10.1145/3472769

1 INTRODUCTION

With the explosive growth of data, processing this data has become the cornerstone of many big data use cases, such as database and data analytics applications [1, 2]. As the size of the stored data increases, the cost of loading and storing the data overweighs the computation cost and diminishes performance. In some applications, such as database, graph processing, machine learning, and statistical analysis, more than half of the execution time is spent on data transfer that shows the impact of data communication on overall performance [3, 4]. The rapid development of **solid-state drives (SSDs)** has shifted the bottleneck of data transfer time from the magnetic disks (i.e., seek and rotational latency) to interconnect bandwidth and operating system overhead [5, 6].

PCIe, the de facto I/O interconnect in conventional computer systems [7], provides limited simultaneous access to the storage devices, which limits the scalability of the system when independent operations are called on different storage devices in parallel. Limited scalability and the low performance of the interconnect bus increase the gap between the performance capacity of storage devices and the interconnection buses [4, 8] that obliges us to move the computations closer to where the data is stored [9]. The recent advancements in near storage computing devices have made the computational storage devices more powerful and easier to deploy in computer systems [5, 10–13].

Near-storage computing offloads a portion of computations to storage drives to accelerate big data applications such as machine learning [14, 15] and database [13, 16–20]. Accordingly, new devices have been developed to bring the computation power into the flash storage devices, such as NGD Systems [12], ScaleFlux [11], and Samsung's SmartSSD [10]. NGD Systems developed computational storage with a multi-core ARM processor to perform in situ computations in NVMe storage devices. ScaleFlux has developed computational storage devices with built-in GZIP compression/decompression as well as customizable database engine accelerators. SmartSSD is an NVMe flash drive with an on-board FPGA chip that processes data in situ. The FPGA, as the computation node of SmartSSD, provides a high degree of parallelism with affordable power consumption and reconfigurability to implement versatile applications. FPGAs run parallelizable applications faster with less power compared to the general processing cores (host processor) [21–26]. The advantage of using SmartSSD over conventional storage devices is thus twofold; offloading tasks to near-storage nodes increases overall performance by bridging the interconnect gap, and the FPGA as an accelerator further boosts the applications with low power consumption [27].

Many of the database operations are principally read intensive such that in some applications, 90% of the total execution time is spent on I/O read [28]. In these applications, the system's performance is limited by the interconnect bandwidth; thus, these applications can be significantly accelerated by offloading the operations to storage devices [4, 17, 29]. Therefore, recent data processing management systems aim to offload the query processing to storage drives to the greatest possible extent to minimize data transfer between the host and storage [13, 18, 28, 30]. Unlike compute-intensive applications, I/O-bound applications do not benefit from high-performance host processors, as their performance is limited by the host-to-storage bandwidth. Therefore, offloading I/O-bound applications to computational storage devices releases the host resources to execute more compute-intensive tasks. As the size of the real-world databases is growing, storing databases requires multiple storage devices. Even though database tables may be significantly large, data processing management systems, such as SparkSQL, partition data into multiple partitions and break down the operations into multiple independent operations on the partitions, where each partition is usually less than hundreds of megabytes. Although these independent operations can be executed in parallel, host processors cannot fully utilize the partitioning opportunity due to the storage-to-host bandwidth limitation. However, in computational storage devices, each storage device has its own computation resource directly connected to the SSD; hence, it executes the independent operations in situ without occupying the storage-to-host interconnect.

Sort operations are widely used in database query processing as a stand-alone operation or as the backbone of more complex database operations such as merge-join, distinct, order-by, and group-by, to name a few [31]. In sorting a database table, all the columns are sorted based on a single column, dubbed *key column*. After sorting the key column, the rest of the table needs to be shuffled accordingly. Most of the FPGA-based accelerators only support sorting integer arrays due to the high complexity of sorting non-integer arrays in general and string arrays in particular [32]. However, sorting a table based on non-integer columns is commonly used in databases. Figure 4 shows that TPCH benchmark queries require sorting tables based on 16 integer key columns and 12 non-integer key columns.

Data processing management systems often use data encoding to compress the stored data. Dictionary encoding is a lossless one-to-one compression method, commonly used in database systems, that replaces attributes from a large domain with small numbers [22, 33, 34]. Since many columns in database tables are highly repetitive, dictionary encoding effectively replaces the attributes with a low-bit-width data type to compress the columns. In the TPCH benchmark, 48% of all columns can be dictionary encoded, and 78% of the columns that will be sorted in different queries can be dictionary encoded (shown in Section 3.3).

In this article, we propose NASCENT2, a near-storage sort accelerator for data analytics using SmartSSDs that has FPGA-based accelerators (kernels) to execute *sort*, *shuffle*, and *dictionary decoding* operations. NASCENT2 sort kernel is based on bitonic sort, which is highly parallelized on FPGA to deliver high performance. The shuffle kernel reorders the table rows based on the sorted column, which is an IO-intensive operation. Therefore, NASCENT2 is designed to maximize bandwidth utilization. If the table is stored in the encoded format, the NASCENT2 dictionary decoder kernel decodes the key column. Then the sort kernel sorts the key column, and the shuffle kernel reorders the table according to the sorted key column. NASCENT2 extends our previous work [35] by supporting decoding variable-length data types and sorting non-integer columns.

Our earlier work [35] is limited to decoding only fixed-length variables and sorting integer columns. However, variable-length columns (e.g., string columns) are an integral part of databases, and dictionary encoding shows better compression in longer data types such as string. Moreover, sorting non-integer columns is crucial due to its abundance in database queries. For instance, in the TPCH benchmark, 42% of the key columns are non-integer, as shown in Section 3.3. Sorting columns with different data types requires different sort kernels, which is not possible due to SmartSSD limited resources. NASCENT2 proposes a novel generic sort method to sort that supports both integer and non-integer columns. It sorts dictionary encoded data by utilizing the sort and the generic dictionary decoder kernels and leveraging the fact that dictionary encoding represents columns with any data type with integer values. NASCENT2 inherently addresses the data transfer issue by carrying out computations near the storage system and exploiting storage-level parallelism. It also embraces an FPGA-friendly implementation of dictionary decoding, sort, and shuffle operations, all of which are parallelized to utilize FPGA available resources and maximize the performance. A summary of the contributions of the article is listed as follows:

- We present NASCENT2, a near-storage accelerator to bring the computations closer to the storage devices by leveraging SmartSSD.
- We propose a novel FPGA-friendly architecture for *bitonic sort* to highly benefit from FPGA parallelism. The proposed architecture is scalable to sort various data size, outputs the sorted indices, and can be scaled based on the available resources of the FPGA.

- Data processing management systems often use dictionary encoding to compress the data. NASCENT2 consists of a novel generic dictionary decoder kernel that supports both fixed- and variable-length data types. The kernel is highly optimized to maximize the SSD-to-FPGA bandwidth utilization.
- Sorting non-integer columns is crucial in processing database queries. We propose a novel generic sort method that supports sorting integer and non-integer columns. It utilizes the proposed dictionary decoder kernel to support sorting non-integer key columns. It shows 2× (3.2×) speedup and 15.2× (23.8×) energy reduction in sorting a string (floating-point) column as compared to CPU.
- Shuffling is the critical step of database sort and is I/O bounded. NASCENT2 accomplishes table sort using the shuffle kernel that fully utilizes the SSD bandwidth to maximize the performance of sorting database tables. We modify the storage pattern of the table to benefit from the regular memory patterns in both shuffle and sort kernels.
- Our evaluations on different table sizes show that NASCENT2 on SmartSSD is 9.9× faster and 7.3× more energy efficient than the same accelerator on conventional architectures comprising a stand-alone FPGA and storage devices where the FPGA is connected to the system through a PCIe bus. NASCENT2 also shows 137.2× speedup and 119.2× energy reduction as compared to the CPU baseline.

The rest of the article is organized as follows. In Section 2, we present the related work on near-storage computing and FPGA-based accelerators for sort operations, and introduce the bitonic sort. In Section 3, we introduce the architecture of the SmartSSD. We present the overall architecture of NASCENT2 and our hardware-software strategy as well as the architectural trade-offs. We also elaborate on the proposed architecture for dictionary decoder, sort, and shuffle kernels. In Section 4, we evaluate the performance of our proposed NASCENT2 on a system consisting of multiple SmartSSDs, and finally, Section 5 offers concluding thoughts.

2 RELATED WORK AND BACKGROUND

2.1 Near-Storage Computing

Previous studies on near-storage computing generally can be categorized as works that propose (a) novel architectures, (b) emulation and/or analysis frameworks that investigate the performance of near-storage systems, and (c) application-oriented case studies that evaluate the efficiency of select applications mapped to specific near-storage systems.

Ruan et al. [8] introduced INSIDER, a computational storage platform equipped with an FPGA drive controller. INSIDER also provides software abstractions to abstract the offloaded operations with file operations. It reduces the required modifications in applications host code to enable offloading the operations on the computational storage. Lee et al. [36] propose ExtraV, an acceleration platform that consists of an FPGA-based "accelerator function unit" that is connected to the storage devices and communicates with the processor and its main memory using a coherent interface. The accelerator executes graph traversal functions that are central to various graph algorithms. IBM's Netezza is a near-storage computing architecture that utilizes FPGAs to reduce the size of the data stream as early as possible by filtering out extraneous data while the data streams out of the storage [30]. The platform supports four functions on the FPGA, namely compress, project, restrict, and visibility, with the capability of expanding to further database operations. Jun et al. [37] propose a homogeneous cluster of host servers that utilize SSDs with in situ processing capability. They used an FPGA directly connected to the storage device to implement the processing core and the controllers required to communicate with the host and network. The proposed system showed that using computational storage devices is more beneficial than using a system with larger DRAM in improving the storage bandwidth and reducing the access latency.

As the computational storage devices are in the early stages, Ruan and Cong [4] provide an emulation platform to estimate the extent an application can benefit by offloading operations on FPGA-enabled computational storage devices. Speaking of the software support, Gu et al. [38] present Biscuit, a near-storage processing framework that provides C++ APIs to allow users to develop, distribute, and load data-intensive applications on the host and storage devices. Reis et al. [28] examine the efficiency of near-storage systems by evaluating the expected performance of particular database operations, namely scan, filter, and project, that are offloaded to storage devices equipped with ARM core as the computation element. Wang et al. [13] explore offloading the list intersection database operation, which is the core of many applications, such as search engines on computational storage devices. Pei et al. [39] offload regular expression (regex) search (a searching algorithm that looks for specific patterns in unstructured data) on computational storages. The accelerator performs a regex search while a file is being transferred to the host.

Cho et al. [40] propose a computational storage prototype that utilizes a multi-core CPU inside the SSD controller. They use MapReduce to exploit storage-level parallelism. MapReduce is a programming model developed by Google that can process parallelizable big data applications on distributed systems [41]. Similarly, Park et al. [42] explore offloading the *map* step of MapReduce to the computational storage devices. It implements the *map* function inside the SSD firmware and then integrates it into Hadoop. Jo et al. [17] present a database system that integrates query offloading to the computational storage devices. The work reduces data transfer between the storage and the CPU by pushing down the filter operation to the computational storage devices and solely transferring the filtered data to the CPU for further processing. The works of Do et al. [15] and Torabzadehkashi et al. [43] propose computational storage devices equipped with multi-core CPUs that are able to run operating systems such as Linux to reduce the efforts for executing the existing applications on computational storage devices. The work of Do et al. [15], to highlight the benefits of near-storage computing, runs various machine learning applications on the computational storage device. The work of Torabzadehkashi et al. [43] deploys and accelerates MapReduce on computational storage devices.

Utilizing high-performance CPUs does not effectively speed up data-intensive workloads that are typical of in-storage processing. These workloads inherently have simple but highly parallelizable computations. Cho et al. [44] propose a new SSD architecture equipped with a **graphics processing unit (GPU)**. They utilize a GPU to provide higher parallelism compared to computational storage devices with CPUs. They provide APIs based on the MapReduce framework to offload data-intensive operations to the GPU of the computational storage devices. Equipping computational storage devices with general-purpose processors (embedded CPUs and GPUs) provides generality at the cost of limited performance and energy efficiency improvement compared to customized hardware accelerators. FPGAs have been exploited in computational storage devices to provide high performance and energy efficiency while providing reconfigurability. Woods et al. [45] use an FPGA in the data path between the host CPU and the storage device to execute *predicate* and *group-by* operations. However, since the FPGA occupies the storage-to-host bandwidth, the framework lacks scalability when the number of storage devices increases.

2.2 FPGA-Based Sort Acceleration

Several works have attempted to accelerate various sort algorithms, such as bubble sort [46, 47], insertion sort [48], heap sort [49], radix sort [50, 51], bitonic sort [35], and merge sort [52–54] on FPGAs [55, 56]. The work of Mueller et al. [56] evaluates the performance of various sorting algorithms on FPGAs, including even-odd [57] and the bitonic sorting network [58], as well as traditional bubble and insertion sorts. In the work of Mahony and Popovici [59], the resource utilization and power consumption of various sort algorithm are studied. Although bitonic sort has a slightly higher computation complexity ($O(n \log^2 n)$) compared to common sorting algorithms (i.e., $O(n \log n$ in merge- and quick-sort), results show that bitonic sort can run faster than the common sort algorithms thanks to its high, FPGA-friendly parallelism.

Chen et al. [60] propose an FPGA-based accelerator for sorting large datasets. They randomly sample the input array and partition it to smaller buckets based on the distribution of data where all the elements in the i^{th} bucket are smaller than or equal to the elements of the $i + 1^{\text{th}}$ bucket assuming the dataset is being sorted in ascending order. Then, each segment can be sorted independently. The random sampling step is resource intensive and slow, as it requires random access to the input data. Thus, they offload the sampling step to the host CPU, which limits the scalability and applicability of the accelerator. The work of Jun et al. [61] proposed an in-storage accelerator for sorting terabyte scale datasets. It proposed a hierarchical sorting architecture that first sorts sub-arrays of the dataset that fits into the FPGA on-chip memory and writes the sorted pages into the FPGA DRAM. It reads multiple partially sorted pages from the DRAM and merges them into sorted super pages written into the storage device. Eventually, it merges super pages in storage to sort the dataset globally. Although the size of the real-world database tables is in the order of terabytes, data processing management systems usually partition the database into smaller partitions. Additionally, the size of the table is significantly larger than the size of the key column that is going to be sorted. Therefore, NASCENT2 aims to deliver high performance for megabyte scale arrays.

The work of Samardzic et al. [52] proposes an adaptive merge tree sorting accelerator. It considers the FPGA available resources as well as the off-chip memory configurations into account to maximize the performance of sorting by choosing the optimal parameters of merge trees. It focuses on modeling the performance of different configurations of merge trees to find the optimal configuration.

Our earlier work [35] proposes a near-storage accelerator for database sort on SmartSSD that performs independent table sort on multiple storage devices simultaneously. It accelerates dictionary decoding, sort, and shuffle operations on SmartSSD. However, the dictionary decoder kernel was limited to fixed-length variables, whereas dictionary encoding is more effective on encoding strings, as it replaces long strings with a small number. Most of the FPGA-based sort accelerators only support numeric arrays and specifically sorting integer arrays. The work of Asiatici et al. [32], to the best of our knowledge, is the only one that supports sorting string arrays. It presents a hybrid CPU-FPGA system for accelerating string sort based on Super Scalar String Sample Sort (S^5) [62]. In S^5, strings are classified into k buckets and then each bucket can be sorted independently. S^5, based on each bucket's size, uses different sorting algorithms (e.g., multi-key quicksort, insertion sort, sequential S^5) to maximize performance. The work of Asiatici et al. [32] accelerates the classifying step as well as the multi-key quicksort algorithm on FPGA and executes the rest of the steps on the host CPU.

This article, NASCENT2, extends our previous work [35] by proposing a novel and generic dictionary decoder architecture that handles both fixed- and variable-length data types. The earlier work [35] is limited to sorting integer columns, whereas sorting non-integer columns is crucial in database queries. In this work, we additionally propose a novel generic sort method that leverages the generic dictionary decoder kernel to support sorting non-integer columns. In real-world databases, the performance of the table sort usually is not dominated by the sort operation. Therefore, unlike state-of-the-art FPGA-based sort accelerators that try to fully utilize the FPGA resources to maximize the sorting performance, in this work we propose a platform that consists

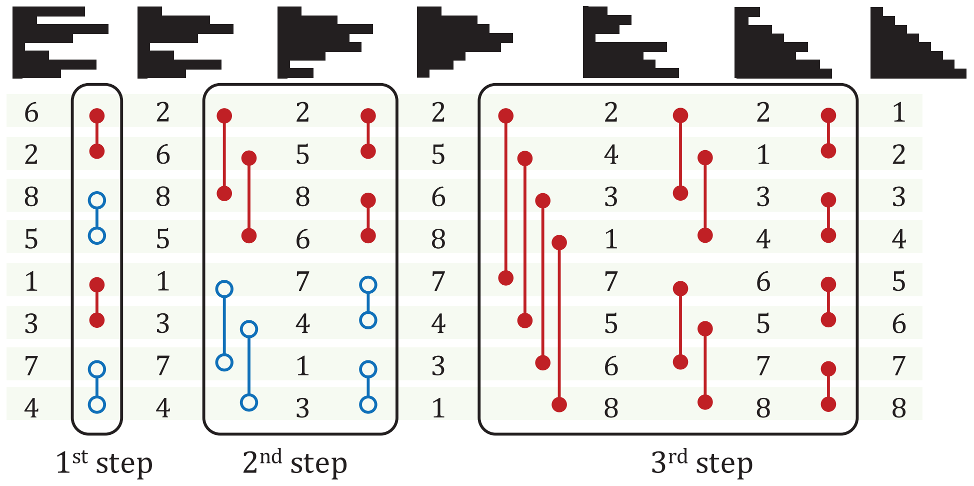

Fig. 1. Example of bitonic sort algorithm steps for an array of eight elements.

of dictionary decoder, shuffle, and sort kernels that maximize the performance of sorting the entire table, including dictionary decoding, sorting the key column, and shuffling the table. Our proposed bitonic sort kernel has a flexible architecture that keeps enough FPGA resources for the other two kernels while providing high sorting performance. The NASCENT2 generic sort method, to the best of our knowledge, is the first FPGA-based generic sort accelerator. In general, NASCENT2 increases the scalability of the system in the presence of multiple storage devices compared to a system with a stand-alone FPGA. NASCENT2 is calibrated to maximize the storage bandwidth utilization as we offload the computations to the storage devices. In contrast to the previous FPGA-based sort accelerators that try to maximize the performance by fully utilizing the DRAM-to-FPGA bandwidth, our goal is maximizing the storage bandwidth utilization, which is lower than the DRAM bandwidth. We tackle the I/O bottleneck by prudently allocating the FPGA resources for dictionary decoder kernel, multiple shuffle kernels, and the sort kernel.

2.3 Bitonic Sort Background

Bitonic sort, proposed by Batcher [63], is a sorting network that can be run in parallel. In a sorting network, the *number* of comparisons and the *order* of comparisons are predetermined and data independent. Having a predefined number and order of comparisons, bitonic sort can be efficiently parallelized on FPGAs by utilizing a fixed network of comparators. Bitonic sort first converts an arbitrary sequence of numbers into multiple bitonic sequences. Merging two bitonic sequences creates a longer bitonic sequence and proceeds until sorting the entire input sequence. A sequence of length n is a bitonic sequence if there is an i ($1 \leq i \leq n$) such that all the elements before the i^{th} are sorted ascending and all the elements after that are sorted descending—that is,

$$x_1 \leq x_2 \leq \cdots \leq x_i \geq x_{i+1} \geq \cdots \geq x_n. \tag{1}$$

Figure 1 shows the steps to sort an example input sequence of length $n = 8$ that consists of $\frac{n}{2}$ bitonic sequences of length 2. The initial unsorted sequence passes through a series of comparators that swap two elements to be in either increasing (red/filled circles) or decreasing (blue/unfilled circles) order. The output of the first step is $\frac{n}{4}$ bitonic sequences each of length 4. Applying a bitonic merge on these $\frac{n}{4}$ sequences creates $\frac{n}{2}$ bitonic sequences. The output sequence after applying $\log_2 n$ bitonic merge produces the sorted sequence.

Generally, in the bitonic merge at the i^{th} step (starting from $i = 1$), $\frac{n}{2^i}$ bitonic sequences of length 2^i are merged to create $\frac{n}{2^{i+1}}$ bitonic sequences of length 2^{i+1}. The i^{th} bitonic merge step itself consists of i sequential sub-steps of element-wise comparison (e.g., in Figure 1, the last/third rectangle is the third step and has three sequences of comparisons). In the first sub-step of the i^{th} step, element k is compared with the element $k + 2^{i-1}$, whereas the first 2^i elements are sorted in ascending order and the next 2^i elements are sorted in descending order (the sorting direction changes after every 2^i element). In the aforementioned example, in the first sub-step of the last/third step, the 1^{st} element (has a value of 2) is compared with the $1 + 2^{3-1} = 5^{\text{th}}$ element (with a value of 7). Generally, in the j^{th} sub-step ($1 \leq j \leq i$) of the i^{th} main step, element k is compared with the element $k + 2^{i-j}$. Thus, in the second sub-step of the third step, the first element (with a value of 2) is compared to the $1 + 2^{3-2} = 3^{\text{rd}}$ element (which has a value of 3 that is updated in the first sub-step).

3 NASCENT2 DESIGN

Data analytics tasks are primarily constrained by disk performance, as operations on a database require a tremendous amount of data. Data encoding is frequently used to compress the table stored in the storage system. Dictionary encoding is a popular encoding method widely used in database management systems. It maps a key to each unique value and replaces values with their corresponding keys. Dictionary encoding is used when the number of unique elements is limited; therefore, dictionary encoding compresses data by representing each value with fewer bits. Unlike byte-oriented compression methods (i.e., gzip, snappy, run-length encoding) that require decompression before query execution, dictionary encoding supports parallel decoding, and in situ query processing for many operations [33]. NASCENT2 is able to execute database sort directly on the dictionary encoded data. However, if the key column is of the integer (32-bit or 64-bit) type, NASCENT2 first decodes the data and then sorts the table due to slightly better performance. In this section, we explain NASCENT2 integer sort and NASCENT2 generic sort and their building FPGA kernels. In the rest of the article, *sort* and *integer sort* are used interchangeably, whereas *generic sort* refers to sorting a key column with any data type. After sorting the table, NASCENT2 reorders the rows of the table based on the sorted key column. NASCENT2 consists of three kernels—dictionary decoder, sort, and shuffle—to execute each step. It performs all the computations on SmartSSD to eliminate host-storage communication. In the following sections, we introduce the architecture of SmartSSD, explain the overall architecture of NASCENT2, and propose the NASCENT2 generic sort method. We then propose the architecture of NASCENT2 sort, shuffle, and dictionary decoder accelerators in Sections 3.4 through 3.6.

3.1 SmartSSD Architecture

Figure 2 demonstrates the general architecture of SmartSSD. It consists of general SSD components, including the SSD controller and NAND array, in addition to an FPGA accelerator, FPGA DRAM, and PCIe switch to set up the communication between the NAND array and the FPGA. The link between the FPGA and the SSD provides direct communication between them and the host. The SSD used by SmartSSD is a 4-TB one connected to a Xilinx KU15P Kintex UltraScale FPGA (with 523K look-up tables and 1,045K flip-flops) through a PCIe Gen3 x4 bus interface.

The host processor is connected to SmartSSD through a PCIe Gen3 x4 bus and is able to issue common SSD commands such as SSD read/write requests to the SSD controller through the SSD driver. Furthermore, the CPU is also able to issue FPGA computation and FPGA DRAM read/write requests via the FPGA driver. In addition to host-driven commands, SmartSSD supports data movement over the internal data path between its NVMe SSD and the FPGA by using the FPGA DRAM and on-board PCIe switch, which we refer to as **peer-to-peer (P2P)** communication. As shown

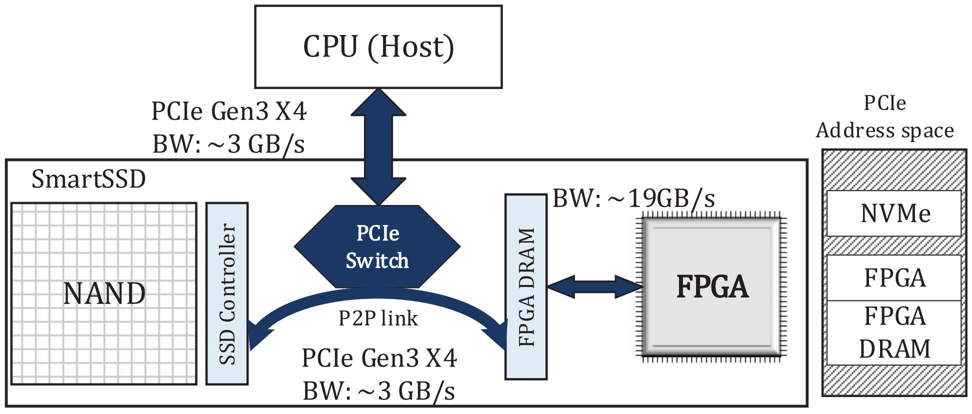

Fig. 2. Overview of SmartSSD architecture.

in Figure 2, FPGA DRAM is exposed to the host PCIe address space so that NVMe commands can securely stream data to FPGA via the P2P communication. P2P brings the computations close to where the data is permanently residing, thereby reducing or eliminating the host-to-storage and the host-to-accelerator PCIe traffic as well as related round-trip latencies and performance degradations. SmartSSD provides a development environment and runtime stack such as the runtime library, API, compiler, and drivers to implement the FPGA-based designs.

3.2 NASCENT2 Overall Architecture

In conventional storage systems, the host processor communicates with the storage devices, reads the data to the memory hierarchy, and performs computations. When an accelerator is present in the system, either the host reads the data from the storage system and transfers it to the accelerator's memory or the accelerator may have a P2P communication with the storage devices to read the data directly. In the former case, the data should pass through the host memory to reach the accelerator's memory (FPGA DRAM in this concept). Thus, the latency of transferring the data through the host is significantly larger than when the accelerator reads the data directly. In addition, P2P communication between the accelerator and the storage devices, unlike the former case, does not occupy the host resources for data transfer.

Although current FPGAs support P2P communication with storage devices, such an architecture still suffers from performance scalability when data is stored in multiple storage devices. Real-world databases need multiple devices to store the data. These databases are larger than what current commodity hardware platforms can cope with. Thus, data processing management systems partition the data into smaller chunks such that the computation nodes can execute the computations on each partition independently and in a timely affordable manner. Thereafter, the management systems combine the result of each partition to generate the final result. Assuming the data is stored in $\mathcal{M}$ SSDs, the tables of each SSD can be divided into a certain number of partitions. To sort the entire database, we can sort all the partitions of each SSD and merge them through a merge tree. Locally sorting each partition is independent of the other partitions; therefore, we can locally sort different partitions in parallel. Our focus is on the partition-level acceleration of sorting the data, as it is the backbone of the main computation.

In sorting a database table, NASCENT2 performance goal is fully utilizing the storage bandwidth. Therefore, parallelizing multiple partitions on a single SSD is not beneficial, as it does not increase the performance, since in this case the FPGA would need to frequently switch between the

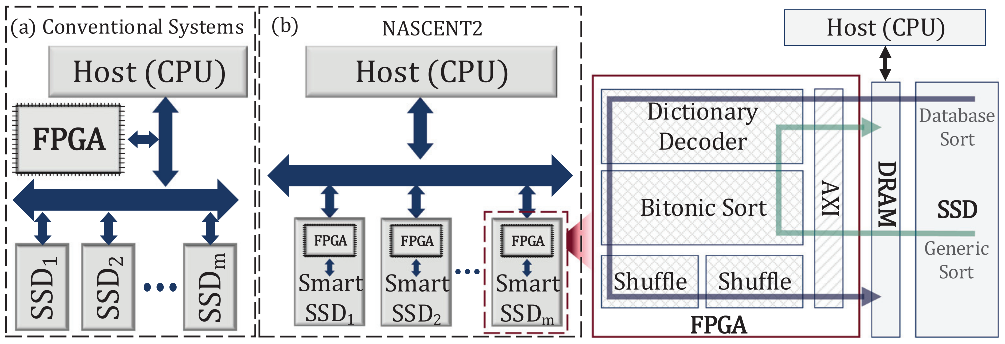

Fig. 3. The overall architecture of NASCENT2 (right) as compared to the conventional systems equipped with an FPGA accelerator (left). The blue arrow represents the data flow of the database sort, and the green arrow shows that of the generic sort.

partitions, as it cannot simultaneously access different partitions. Thus, NASCENT2 parallelizes the computations in SSD level (shown in Figure 3), which is not possible in conventional architecture. In conventional architectures, the FPGA is connected to the storage devices using a PCIe bus that cannot provide simultaneous access to multiple SSDs. NASCENT2 deploys SmartSSDs that enable P2P connection between the FPGA and the SSD. Each SmartSSD, therefore, can sort an SSD-level partition independent of the others, which significantly accelerates the overall system performance as the number of storage devices grows.

Figure 3(b) shows the building kernels of NASCENT2. It utilizes these kernels to perform database sort. In Figure 3, the blue arrow shows the sequence of operations to sort a database table. Additionally, the green arrow shows the flow of data for the generic sort method proposed to sort non-integer columns. As NASCENT2 consists of multiple different kernels, it deals with a trade-off between allocating resources to these kernels. The dictionary decoder kernel is able to saturate the storage-to-FPGA bandwidth in decoding fixed-length variables, and in decoding variable-length data types, operations cannot be parallelized, as the lengths of the decoded outputs are not known before decoding; thus, NASCENT2 meets the performance goal by instantiating a single dictionary decoder kernel. A single shuffle kernel cannot fully utilize the SSD-to-FPGA bandwidth due to the fact that, although in NASCENT2 we have proposed a new table storage format that enables reading a row in a sequential pattern, reading the next row still requires random memory access, which has a high latency. Therefore, we aim to maximize the total input bandwidth (SSD-to-FPGA bandwidth) for all the shuffle kernels to maximize the performance.

Due to the fact that the shuffle operation is I/O intensive and the size of the table is significantly larger than the size of the key column, the performance of the shuffle operation is determinative of the overall performance. Thus, we instantiate multiple instances of the shuffle kernel (as shown in Figure 3) to fully leverage the storage-to-FPGA bandwidth and a single instance of the dictionary decoder kernel and dedicate the rest of the resources to the sort kernel. Based on our evaluations, we found out that we can fully utilize the storage-to-FPGA bandwidth in the shuffle and dictionary decoder kernel while still having sufficient resources to have a high-throughput sort. The sort kernel uses a considerable portion of FPGA BRAMs to store the arrays and provide parallelism. Additionally, the dictionary decoder kernel requires on-chip memory to store the dictionary table locally to provide high throughput. Therefore, the NASCENT2 dictionary decoder primarily uses FPGA **Ultra RAMs (URAMs)** to balance the overall resource utilization of NASCENT2.

NASCENT2 program is split between a host program and the FPGA-based kernels with a communication channel between them. NASCENT2 provides OpenCL APIs for users to execute

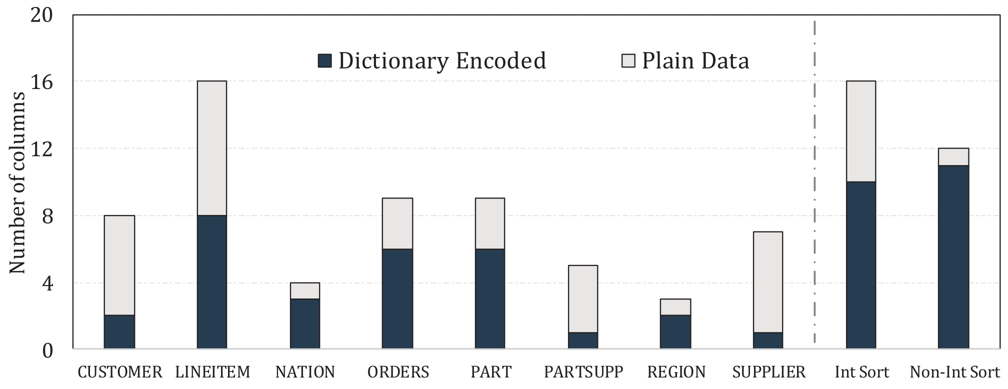

Fig. 4. The number of sort operations in different queries and the number of columns that can be dictionary encoded.

database sort on SmartSSD. The host program runs on the host CPU and calls OpenCL APIs to invoke the FPGA-based kernel run on the FPGA of SmartSSD. The API calls, managed by Xilinx Runtime Library (XRT), are used to process transactions between the host program and the FPGA on SmartSSD. These communications include transferring the control signals to/from the NASCENT2 kernels as well as transferring the data from/to the host CPU to/from the FPGA DRAM. In the case of a P2P communication between the FPGA and the SSD, the host program only sends the address and the size of data in the storage to the FPGA. Then the FPGA directly initiates the data transfer from the SSD to the FPGA kernels.

3.3 Generic Sort in NASCENT2

Database queries frequently require sorting non-integer columns. Sorting different data types require different hardware and even different operations, such as in string sort. Figure 4 shows the number of columns in each table of the TPCH [64] benchmark that have less than 2^{16} unique values and can be dictionary encoded as well as the total number of columns in each table. As illustrated in the figure, on average, 48% of the columns can be dictionary encoded. Additionally, some of the plain format columns are not used in query processing, such as the comment columns. Hence, database management systems frequently use dictionary encoding to store the data. NASCENT2 leverages the fact that key columns are usually dictionary encoded to perform both integer and non-integer sort, so-called generic sort, using the existing kernels in NASCENT2. Figure 4 also represents the frequency of calling the sort function on integer and non-integer columns in all of the 22 queries of TPCH. In all queries, 16 integer columns and 12 non-integer columns need to be sorted. As the number of non-integer key columns is comparable to the number of integer columns, supporting sorting non-integer columns is crucial for accelerating the database sort. As illustrated in the figure, for different queries, 16 integer columns will be sorted, among which 10 can be dictionary encoded. TPCH queries also need to sort 12 non-integer columns, of which 11 of those columns can be dictionary encoded. Developing a dedicated sort kernel for each data type requires an immense amount of resources, more than the total available resources on SmartSSD. However, most of the database columns can be dictionary encoded since most of the columns have limited unique values, specifically columns that need to be sorted (key columns). NASCENT2 not only supports all the integer sorts but also supports 11 out of the 12 non-integer sorts by utilizing the existing kernels with negligible overhead.

NASCENT2 proposes a novel method to sort dictionary encoded columns by leveraging the proposed sort and dictionary decoder kernels. Note that even for applications where the size of

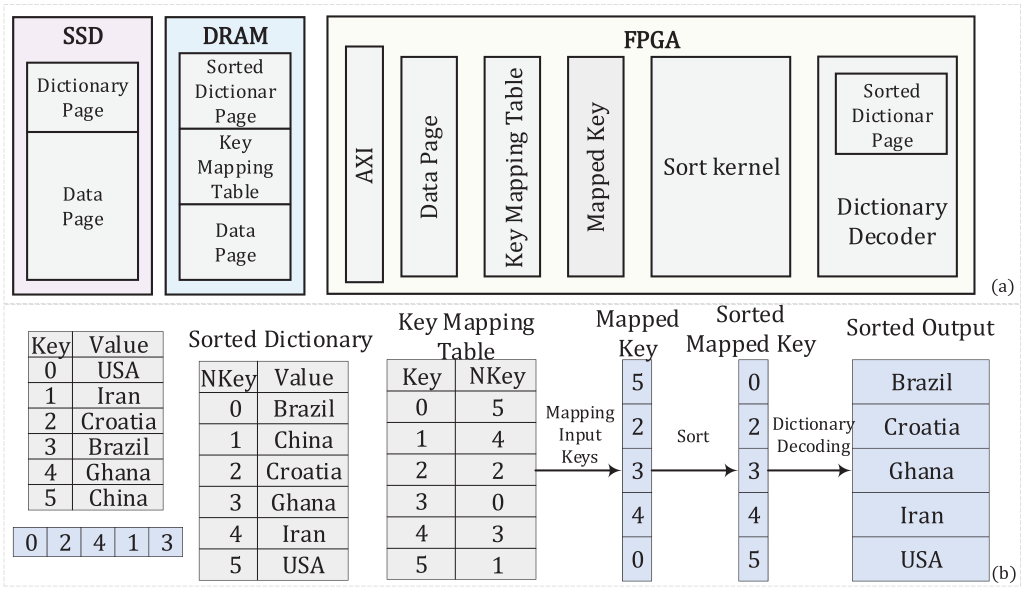

Fig. 5. (a) Architecture of the NASCENT2 generic sort. (b) Example of sorting an array of string.

the dictionary table is larger than the FPGA available on-chip memory, NASCENT2 is still able to perform the generic sort by offloading the dictionary decoding, which is the very last step, to the host CPU. Figure 5(a) shows the architecture of the NASCENT2 generic sort method. The table is stored in the storage system, and the key column is dictionary encoded. The rest of the columns can be stored in either dictionary encoded or in plain format. First, NASCENT2 reads the dictionary page of the key column on the host server. Generally, the size of the dictionary page is significantly smaller than the size of the data page. The dictionary page's size is at most 128 KB (2^{16} 16-bit elements). Hence, NASCENT2 takes advantage of the host server to sort the dictionary table due to the efficiency of sorting small arrays on CPUs. The host server sorts the dictionary table based on the values and assigns the index of each value in the sorted dictionary page as its new key, called *NKey*. The host server also generates the "key mapping table" that maps the original dictionary page's keys to the NKeys. The sorted dictionary page and the key mapping table are used in the generic sort method. NASCENT2 uses the key mapping table to map the input data to the "Mapped Key array." In this mapping, the keys' order is the same as the order of the sorted values. For instance, if we sort a column in ascending order, greater keys correspond to greater values in the sorted data.

The host program transfers both the sorted dictionary page and key mapping table to the FPGA DRAM. NASCENT2 reads the data page directly from the storage device to eliminate the overhead of transferring the data page through the host server. NASCENT2 loads the key mapping table and then streams the data page into the FPGA. NASCENT2 then maps the input keys to NKeys using the key mapping table. It then initiates the sort kernel to sort the Mapped Key array, which is analogous to sorting the original data page since the order of the NKeys is the same as the order of the values in the sorted data. NASCENT2 sorts the Mapped Key array and writes it into the "sorted Mapped Key array." It uses the dictionary decoder kernel to decode the sorted Mapped Key array to the sorted array in the original data type.

Figure 5(b) shows an example of sorting a column of strings. The string column is dictionary encoded, and the data page is stored in the storage as {0, 2, 4, 1, 3} along with the dictionary page.

In this example, for the sake of simplicity, we use a small column; however, in real-world applications, the data page size is significantly larger than the dictionary page. The host server sorts the dictionary page and generates the key mapping table as shown in Figure 5(b). In the original dictionary encoded data, string *USA* is mapped to key 0. After sorting the dictionary table, the value *USA* is the last element between all the values. The key mapping table maps key 0 to NKey 5. NASCENT2 reads the data page ($\{0, 2, 4, 1, 3\}$) and maps each element to its corresponding to generate the Mapped Key array = $\{5, 2, 3, 4, 0\}$. It then sorts the Mapped Key array. It uses the sorted dictionary table to decode the sorted Mapped array to the original data type. For instance, key 3 in the original dictionary page corresponds to the dictionary value *Brazil*. Since Brazil comes first in the sorted data, NASCENT2 maps key 3 to NKey 0. Then in the sorted Mapped Key array, the element 0 becomes the first element. NASCENT2 decodes the sorted Mapped Key array, so the first element would be decoded to *Brazil*.

3.4 NASCENT2 Sort Kernel

To sort a database table, NASCENT2 begins with sorting the *key* column. As mentioned earlier, the sequence of operations in bitonic sort are predefined, data independent, and parallelizable. Therefore, NASCENT2 takes advantage of FPGA characteristics to accelerate the bitonic sort. The input sequence is stored in the FPGA DRAM, also referred to as off-chip memory. Then NASCENT2 streams the input sequence into the FPGA through the AXI ports, with an interface data width of 512 bits (sixteen 32-bit integers). The AXI port writes the data to the *input buffer*, which has a capacity of $\mathcal{P} = 2^m$ integer numbers. To have a regular sort network, without lack of generality $\mathcal{P}$, the size of bitonic sort kernel, is a power-of-2 number (we can use padding if the total data elements is not a multiple of $\mathcal{P}$). If $\mathcal{P}$ is greater than 16, it takes multiple cycles to fill the input buffer. Whenever the input buffer fills up, it passes the buffered inputs to the $\mathcal{P}$-sorter module.

$\mathcal{P}$-sorter is implemented in parallel and consists of $\log_2 \mathcal{P}$ steps. The module is highly pipelined to meet the timing requirement of FPGA and able to provide a throughput of one sorted sequence (of size $\mathcal{P}$) per cycle. As explained in Section 2.3, the first step in the $\mathcal{P}$-sorter compares elements of even indices ($2k$-indexed elements) with their successor element. Thus, the first step requires $\frac{\mathcal{P}}{2}$ **compare-and-swap (CS)** modules. During the second step, it first compares and swaps the elements with indices $4k$ with $4k + 2$, and $4k + 1$ with $4k + 3$. Afterward, it compares and swaps $2k$ elements with $2k + 1$ elements of the updated array (see Figure 1). Therefore, the second step in the $\mathcal{P}$-sorter requires $\frac{\mathcal{P}}{2} + \frac{\mathcal{P}}{2} = \mathcal{P}$ instances of the CS module. Analogously, the i^{th} step in the $\mathcal{P}$-sorter where $1 \leq i \leq \log_2 \mathcal{P}$ needs $i \times \frac{\mathcal{P}}{2}$ CS modules. The total number of required CS modules for the $\mathcal{P}$-sorter can be estimated as follows:

$$n_{CS} = \frac{\mathcal{P}}{2} + \left(2 \times \frac{\mathcal{P}}{2}\right) + \cdots + \left(\log \mathcal{P} \times \frac{\mathcal{P}}{2}\right) \simeq \frac{\mathcal{P}}{4} \log^2 \mathcal{P}. \tag{2}$$

NASCENT2 orchestrates the sort operation on the entire data by leveraging the $\mathcal{P}$-sorter modules and FPGA's fast on-chip memory, called **block RAMs (BRAMs)**. First, when sorting every $\mathcal{P}$ elements, $\mathcal{P}$-sorter toggles between ascending and descending orders. The sorted output of $\mathcal{P}$-sorter modules are written into the *sequence memory*, which consists of two sub-memory blocks, say M_1 and M_2, that are made up of FPGA BRAMs. Initially the ascending and descending sorts are respectively written in M_1 and M_2 (see step 1 in Figure 6(b)). Each row of M_1 and M_2 contains $\mathcal{P}$ elements that together form a bitonic row (as the first half is ascending and the second half is descending) in the sequence memory with a length of $2\mathcal{P}$. Note that by *row*, we mean adjacent placements of items in a sequence, not necessarily a physical row of a BRAM that can just fit one or two integers. Since the $2\mathcal{P}$ sequence is just a single bitonic array, using a merging procedure

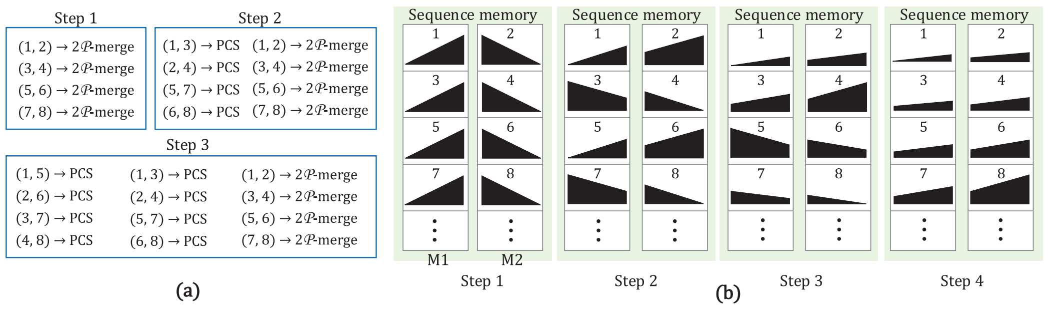

Fig. 6. (a) NASCENT2 scheduling to sort the sequence memory. (b) The content of the memory at each step.

similar to the last (third) step of Figure 1, the $2\mathcal{P}$ bitonic array can be sorted using $P \times \log(2\mathcal{P})$ CS units.

Figure 6(a) lists the steps of merging the results of $\mathcal{P}$-sorters, and Figure 6(b) illustrates the results after each step. Indeed, merging the results of $\mathcal{P}$-sorters is itself a bitonic-like procedure but on *sorted arrays* rather than scalar elements. In other words, similar to step 1 in bitonic sort, step 1 in Figure 6(a), (b) merges the adjacent *arrays*. Step 2 of Figure 6 also is similar to the second step of the simple bitonic sort that compares and swaps every item i with item $i+2$ using **parallel compare-and-swap (PCS)** units, followed by comparing item i with item $i+1$ in the modified array. Thus, we can consider the entire sort as intra-array followed by inter-array bitonic sort. When NASCENT2 accomplishes sorting the entire sequence memory, it writes it back into the off-chip DRAM and uses the same flow to fetch and sort another chunk of the input sequence repetitively and then merges them to build larger sorted chunks.

To provide the required bandwidth for the parallelization, each of the M_1 and M_2 memory blocks use the $\mathcal{P}$ column of BRAMs in parallel, so $\mathcal{P}$ integers can be fetched at once (the data width of FPGA BRAMs is 32 bit or one integer). In addition, in each memory block, $\mathcal{L}$ rows of BRAMs are placed vertically (e.g., in Figure 6(b), $\mathcal{L} = 8$), so the results of $\mathcal{L}$ sorters can be compared simultaneously. The number of BRAMs and their capacity in terms of 32-bit integers number can be formulated as follows.

$$\begin{aligned} n_{\text{BRAM}} &= 2 \times \mathcal{P} \times \mathcal{L} \\ C_{\text{BRAMs}} &= 1024 \times 2 \times \mathcal{P} \times \mathcal{L} \end{aligned} \tag{3}$$

Note that BRAMs have a 1024 (depth) × 32 bit (width) configuration. At each iteration, $C_{\text{BRAMs}} = 2048\mathcal{P}\mathcal{L}$ integers are sorted and written back to the off-chip DRAM.

To sort a database table, the rest of the table rows have to be reordered based on the sorted key column's indices, called *sorted indices*. Thus, we also need to generate the sorted indices that will later be used by the shuffle kernel to sort the entire table. To this end, when reading the input sequence from the DRAM, we assign an index to each element and store the indices in an index memory that has the same capacity as the sequence memory. When reading from the sequence memory and feeding inputs to the $\mathcal{P}$-sorter, NASCENT2 reads the corresponding index and concatenates to the value. The CS units of $\mathcal{P}$-sorters perform the comparison merely based on the value part of the concatenated elements, but the entire concatenated element, if required, will be swapped. NASCENT2 therefore stores the sorted indices in the DRAM as well.

Figure 7 demonstrates a tangible implementation of the discussed steps of the bitonic sort kernel. The $\mathcal{P}$-sorter module sorts chunks of $\mathcal{P}$ elements and stores in the following sequence memory. The M_1 memory group stores the ascending sorts while M_2 stores the descending sorted elements. There are $\mathcal{P}$ BRAMs at every row of the M_1 (and M_2) memory, so the sorted $\mathcal{P}$ elements are

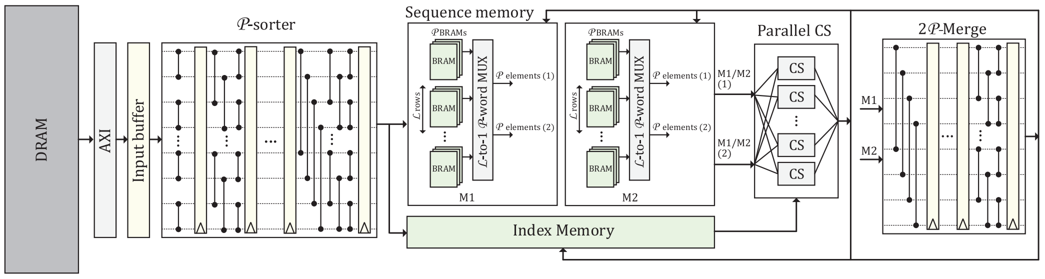

Fig. 7. Architecture of the NASCENT2 bitonic sort kernel.

partitioned element-wise for subsequent parallel operations. In the PCS sub-steps, two $\mathcal{P}$-element arrays from the same memory (either M_1 or M_2, e.g., arrays 1 and 3 from M_1, or 2 or 4 from M_2 shown in Figure 6(a)) are fetched, whereas in the last sub-step (i.e., merge), a $\mathcal{P}$-element array from M_1 and another from M_2 are fetched and sorted/merged. In our architecture, this is enabled using $\mathcal{L}$-to-1 multiplexers that are connected to all $\mathcal{L}$ BRAM groups and select up to two *arrays* from each M_1 and M_2. As shown in the architecture, the PCS and merge modules' outputs are written back in the sequence memory to accomplish the next steps.

3.5 NASCENT2 Shuffle Kernel

After sorting the key column and generating the sorted indices array, NASCENT2 uses the shuffle kernel to reorder the table rows. To do so, the shuffle kernel reads the first element of the sorted indices array, which is a row index in the original table. Then it reads all the entries of the original row that the index points to and writes it as the first row of the new sorted table. Analogously, to generate the i^{th} row of the sorted table, NASCENT2 reads the i^{th} element of the *sorted indices* sequence. The index represents the index of the row in the original table. Thus, we can formulate the mapping between the original table and the sorted one as follows.

$$\text{SortedTable}[i] = \text{OriginalTable}(\text{SortedIndices}[i]) \tag{4}$$

Evidently, the shuffle kernel does not perform any computation; hence, the kernel's performance is bounded by the memory access time. Storing the tables in the storage therefore directly affects the performance of the kernel. Typically, tables are stored in either column-wise or row-wise format. In the column-wise format, elements of every column are stored in consecutive memory elements, which is shown in Figure 8(a). In the row-wise format, all the elements of a row are placed in successive memory elements (see Figure 8(b)). Consecutive memory elements can be transferred to the FPGA from its DRAM in the burst mode, significantly faster than scattered (random) access.

Storing the table in column-wise format results in sequential/burst memory access pattern in the sort kernel (since it needs access to the consecutive elements of the key column, denoted by C_k in Figure 8). However, the shuffle kernel will have random access patterns (as the shuffle kernel needs access to the consecutive elements of the same row, which are placed distantly in the column-wise arrangement). Analogously, storing the table in row-wise format enables sequential access patterns to read a single row (suitable for the shuffle kernel) but reading the next row (required in sort kernel) issues random memory access. To optimize the access patterns of both kernels, NASCENT2 uses a hybrid technique for storing the table in the storage. As shown in Figure 8(c), we store the key column (C_k) column-wise while the rest of the table is stored in row-based format. Therefore, both kernels can benefit from sequential memory access. Note that NASCENT2 natively supports all row-wise, column-wise, and the proposed data layout. However, the proposed data

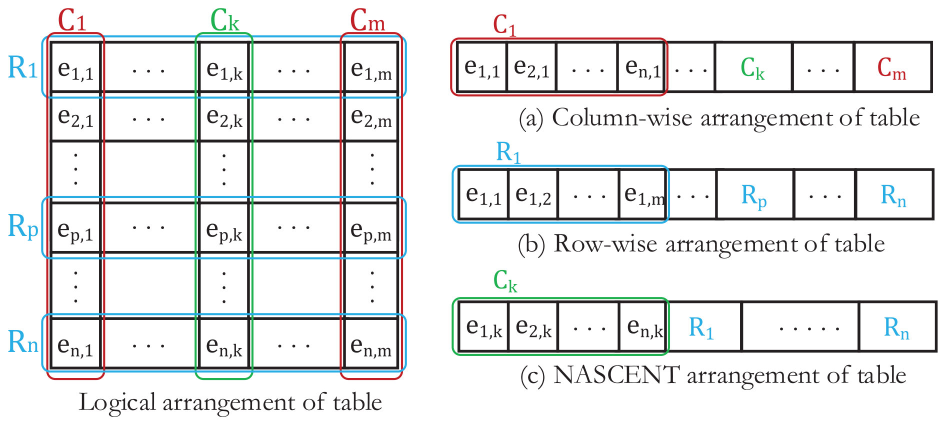
C1
Ck
Cm
R1
Rp
Rn
e1,1
e1,k
e1,m
e2,1
e2,k
e2,m
ep,1
ep,k
ep,m
en,1
en,k
en,m
Logical arrangement of table
e2,1
en,1
(a) Column-wise arrangement of table
e1,2
e1,m
(b) Row-wise arrangement of table
e2,k
en,k
(c) NASCENT arrangement of table

Fig. 8. Storing the table in column-wise (a), (b) row-wise (b), and our proposed (c) format.

layout provides higher performance since both row-wise and column-wise data layouts impose significantly higher numbers of random memory access than the proposed data layout.

3.6 NASCENT2 Dictionary Decoder Kernel

Dictionary encoding is one of the most widely used compression techniques, used either as a standalone compression technique [33] or as a step combined with other compression techniques such as in Parquet [22].

Dictionary encoding is a lossless compression technique that maps each "value" to a "key." As entries in a column usually have the same data type and repetitive values, dictionary encoding is usually applied to each column independently. Applying dictionary encoding on a column is beneficial when the size of the encoded data plus the size of the dictionary table is smaller than the original data—in other words, when the number of entries in the column is significantly higher than the number of unique values (U) in the column. Each unique value is represented by a k-bit key, where $k = log_2(U)$. Database management systems only apply dictionary encoding when the size of the encoded data plus the dictionary table size is considerably smaller than the original data. Dictionary encoding is more effective on columns with large data types, such as strings. NASCENT2 provides a generic dictionary decoder that supports input with various bit widths and outputs with different data types (both fixed- and variable-length data types) and bit widths that can be configured during the runtime.

NASCENT2 uses the dictionary decoder kernel during the database sort flow if the key column is stored in dictionary encoded format. Figure 9(a) shows the proposed architecture of the NASCENT2 dictionary decoder. The NASCENT2 dictionary decoder first reads the "dictionary page," which is stored along with the encoded data, from global memory or directly from the storage system. It stores the dictionary page in FPGA local memories to provide fast access to decode the inputs. In decoding variable-length data types, the length of the dictionary values may be different. Thus, to maximize the utilization of the FPGA on-chip memories, NASCENT2 concatenates the dictionary values and stores them in the dictionary table. The NASCENT2 dictionary table consists of R rows where each row is L_{Max} bytes. L_{Max} sets the upper bound for the length of the longest dictionary value in bytes. In other words, the NASCENT2 dictionary decoder can execute dictionary decoding operations where the lengths of the dictionary values are less than or equal to L_{Max} bytes.

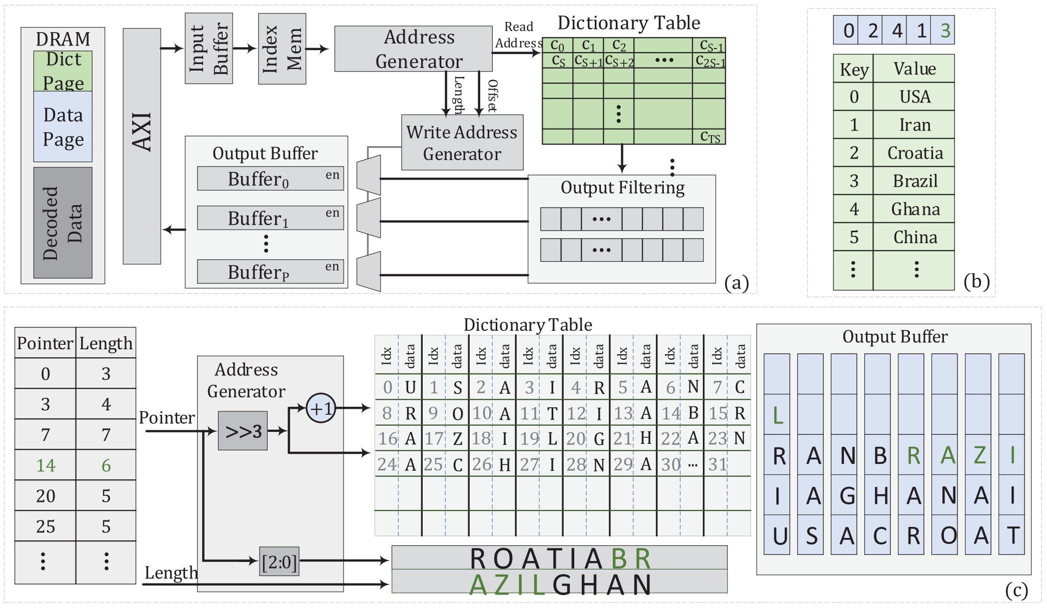

Fig. 9. (a) Architecture of the NASCENT2 dictionary decoder. (b) Example of the dictionary page and data page. (c) Intermediate values in the NASCENT2 dictionary decoder kernel running the example of (b).

Since the NASCENT2 dictionary decoder concatenates the dictionary values, bytes of a dictionary value may split into two consecutive rows of the dictionary table. Each row's length is greater than or equal to the length of every dictionary value; consequently, each value is either stored in a single row or split into two consecutive rows. To find the location and length of the value corresponding to every key, the NASCENT2 dictionary decoder constructs the "index memory" while loading the dictionary table. The index memory stores the *byte address* and the *length* of every dictionary value in the dictionary table. For each value, the byte address is the accumulation of all the previous lengths, and the length represents the size of the value in bytes.

The NASCENT2 dictionary decoder, after loading the dictionary, and constructing the index memory, streams the data page using the AXI interface. For the sake of design simplicity and AXI compatibility, the NASCENT2 dictionary decoder limits the input bit widths, k, to power-of-2 numbers greater than 8. AXI interface reads the encoded data page elements and writes them in the "input buffer." The NASCENT2 dictionary decoder also gets an input from the user, which determines if the output data type is of a fixed-length or variable-length data type. If the output is of a variable-length data type, the input keys may be associated with values with different bit widths. Thus, parallelizing the decoding process is not possible, whereas decoding fixed-length data types is parallelized to maximizing performance.

NASCENT2 at every clock cycle reads a key from the input buffer (it reads multiple keys in case of decoding fixed-length data types) and looks up the byte address and length of the corresponding value in the dictionary table from the index memory. As there is multiple access to both index memory and dictionary table in every clock cycle, NASCENT2 uses on-chip memory to store these two tables. The index memory outputs the byte address of the first byte of the value in the dictionary table as well as the length of the value. NASCENT2 uses the byte address to find the row address of the dictionary table that contains the value. A dictionary value is either entirely stored in a row or split into two consecutive rows. Therefore, for each key, the address generator outputs the row address that contains the first byte of the value and the next row. To get the row address, it shifts the byte address to right for $log_2(L_{Max})$ bits, as shown in Equation (5). NASCENT2 then uses the row address and row address +1 to read the dictionary value.

$$\begin{aligned} Row\ address &= byte\ address >> log_2(L_{Max}) \\ Offset &= byte\ address[log_2(L_{Max}) - 1 : 0] \end{aligned} \tag{5}$$

NASCENT2 reads two rows of the dictionary table and writes them into the "output filtering" module. The output filtering uses the byte address along with the length of the value to find and filter the value corresponds to the input key. As shown in Equation (5), the byte address $[L_{Max} - 1 : 0]$ is used as the offset in the output filtering module to get the first byte of the value from the two rows, read from the dictionary table. It then outputs the value and writes it into multiple parallel buffers of the "output buffer" module.

In the output buffer module, NASCENT2 uses multiple parallel buffers to increase the output bandwidth utilization and consequently increase the performance by writing multiple bytes into the FPGA DRAM in each cycle through an AXI port. The bit width of each AXI transaction directly affects the bandwidth utilization and consequently the performance of dictionary decoding. The bit width of each transaction can be up to 512 bits, equal to 64 bytes. Therefore, the NASCENT2 dictionary decoder uses parallel buffers to utilize the AXI interface more efficiently. The output buffer module concatenates the consecutive values and writes them into B parallel buffers, and whenever all the B buffers have at least an element in them, it transfers the B bytes into the FPGA DRAM. B will be set as the minimum of L_{Max} and 64, because in each cycle, the output of the output filtering module will be at most L_{Max} bytes and providing a higher number of buffers than L_{Max} does not increase the performance. However, when $L_{Max} > 64$, the AXI interface can only read up to 64 bytes per cycle; having more buffers increases the kernel's complexity.

Figure 9(b) and (c) show an example of the NASCENT2 dictionary decoder for a dictionary page of strings. The dictionary page and the input page are shown in Figure 9(b). In this example, based on the dictionary page's values, the maximum length of the values is set to 8 bytes, so each row of the dictionary table contains 8 bytes ($L_{Max} = 8$). Figure 9(c) shows sample contents of the memory and buffer at runtime. NASCENT2 constructs the content of the index memory while reading the dictionary page. The first value in the dictionary page is *USA*, which starts at address 0 and is 3 bytes long. The next value, *Iran*, starts at address 3 in the dictionary table and is 4 bytes long. NASCENT2 loads the dictionary table and constructs the index memory as explained. To decode the last input (3), for instance, the byte address and the length parameters are 14 and 6 respectively. The first byte of the corresponding value, *Brazil*, starts at address 14. The address generator shifts the byte address to the right for 3 bits ($byte\ address >> 3 = 1$), namely the corresponding value is in rows 1 and 2 of the dictionary table. NASCENT2 reads rows 1 and 2, as shown in the following, from the dictionary table and writes them into the output filtering module.

$$Row[1] = ROATIA\mathbf{BR}$$

$$Row[2] = \mathbf{AZIL}GHAN$$

Bytes [2:0] of the byte address are used as the offset from the first byte of the first read row. The value starts at byte *offset* and ends after *length* bytes. In this example, the offset is equal to 6, and the length is equal to 6. Hence, the value is from bytes 6 to 11 of the read rows. The filtering module extracts the value from the read rows and writes it into parallel output buffers. Note that decoding fixed-length data types is a special case of variable-length decoding where the output bit width of all values is the same. Therefore, NASCENT2 parallelizes the operations and decodes multiple input keys per cycle.

Table 1. Characteristics of SmartSSD Resources and the Breakdown of NASCENT2 Kernels' Resource Utilization

	LUT	BRAM	URAM	DSP	DRAM	D2FPGA BW	S2FPGA BW	Storage
SmartSSD Resources	522K	984	128	1959	4 GB	19 GB/s	3 GB/s	4 TB
Sort	39%	74%	0%	1%	–			
Dictionary decoder	1%	0%	85%	0%	–			
Shuffle (6 kernels)	3%	2%	0%	1%	–			
Platform	32%	14%	0%	0%	–			
NASCENT2	75%	90%	85%	2%	–			

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We implement the dictionary decoder, sort, and shuffle kernels on the FPGA available on SmartSSD to evaluate the efficiency of NASCENT2. Each SmartSSD consists of a 4-TB SSD directly connected to a Kintex UltraScale+ FPGA, XCKU15P, through a PCIe Gen3 x4 bus. Table 1 summarizes the available resources of SmartSSD. In this table, D2FPGA BW stands for DRAM-to-FPGA bandwidth, and S2FPGA BW indicates the SSD-to-FPGA bandwidth (the bandwidths are also shown in Figure 2). The table also shows the resource utilization of each kernel. NASCENT2 kernels are written in C++ and optimized to deliver high performance. The kernels are synthesized using the Vivado High-Level Synthesis (HLS) tool and integrated with the host code using Xilinx Vitis Accel 2019.2. The host code is written in OpenCL, which is responsible for initiating the kernels and passing the tables' location in the storage. The SmartSSD FPGA has a P2P communication with the SSD, and all communications between the storage and the FPGA will happen internally without involving the host. To measure the performance of the entire database sort and also individual kernels, we used OpenCL event profiling. We report end-to-end execution times, including the P2P communication between the FPGA and the SSD in the SmartSSD to transfer the data, and the computation time, unless otherwise stated. To evaluate the energy efficiency of NASCENT2, we measure the power consumption of the FPGA, including its off-chip DRAM, without including the power of SSD since we use the same SSD for all deployments.

4.2 Kernel Evaluation

In this section, we compare NASCENT2 with the CPU-based sort baseline to specifically examine the performance of NASCENT2's sort kernel architecture. For the baseline CPU sort, we use the quicksort implementation developed in the C++ standard library (std::qsort()), which is generally considered as one of the fastest single-threaded sort algorithms. We next compare NASCENT2's performance against the multi-threaded block sort from the Boost library (boost::sort::block_indirect_sort()) [65]. We also developed a multi-threaded implementation of the shuffle and dictionary decoding using the *pthread* library. The software implementations run on the Intel Core i7-8700 processor (12 threads) with a clock frequency of up to 4.6 GHz.

Table 1 shows the resource utilization of each kernel. As explained in Section 3.2, NASCENT2 utilizes a single dictionary decoder and six independent shuffle kernels to saturate the storage-to-FPGA bandwidth. It dedicates the rest of the resources to the sort kernel. Both sort and dictionary decoder kernels utilize FPGA on-chip memories. The sort kernel uses FPGA BRAMs while the dictionary decoder kernel stores the dictionary tables in FPGA URAMs to balance the resource utilization. The *Platform* row shows the resources used to implement the AXI interfaces and streaming buffers used by NASCENT2 kernels. As illustrated in the table, FPGA on-chip memory is a limiting resource. The performance of the dictionary decoder and shuffle kernels is limited by the

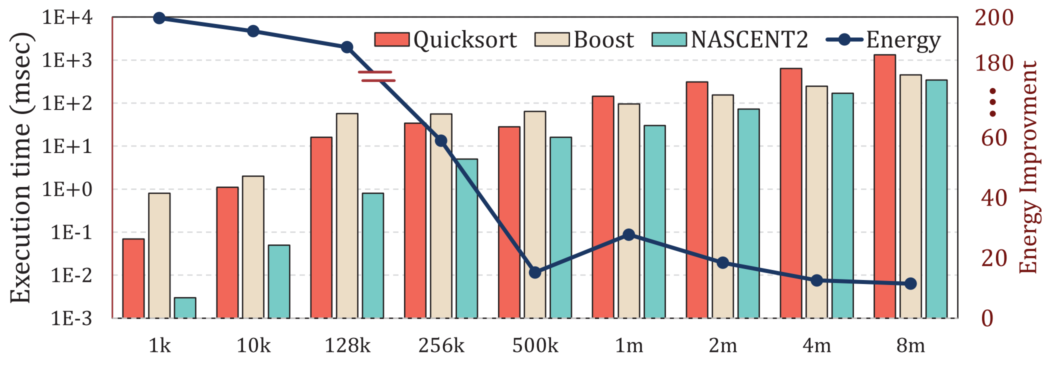

Fig. 10. Execution time and relative energy efficiency of the NASCENT2 *sort kernel* compared to the quicksort and multi-threaded block sort from the Boost library running on CPU when the data is available in the DRAM of CPU and FPGA. The Y axis is in logarithmic scale.

storage-to-host bandwidth. Thus, having an FPGA with a larger on-chip memory only increases the performance of the sort kernel, whereas using a more advanced interface between the FPGA and the SSD can increase the performance of both shuffle and dictionary decoder kernels.

Sort Kernel. Figure 10 compares the performance and the energy efficiency of NASCENT2's sort kernel with quick-sort and parallel block sort from the Boost library (referred to as *Boost* in the figure) on CPU when the data is available in the DRAM memory of both CPU and FPGA. The execution time includes reading the input array from the DRAM, sorting the array, and writing the sorted array into the platform DRAM. The input sequences are randomly generated with the lengths of 1,000 elements (1K) to 8,000,000 elements (8M). As NASCENT2 will be integrated with data processing management systems such as SparkSQL, our goal is to provide higher performance in tables smaller than hundreds of megabytes, which is the typical partition size in SparkSQL. In such tables, the key column has typically less than 8M elements. Therefore, we only compare NASCENT2 with software baselines for arrays smaller than 8M elements. The sort kernel of NASCENT2 consistently delivers higher performance than the CPU implementations. NASCENT2 sort kernel fits up to 128K elements inside the FPGA on-chip BRAM blocks. Therefore, input sequences smaller than 128K elements are sorted in a single iteration.

For a larger number of inputs, NASCENT2 sorts the first 128K elements, writes them back to the DRAM, and fetches another 128K of data until it (partially) sorts the entire input sequence. Eventually, the sort kernel merges the sorted chunks stored in the DRAM. Because the DRAM communication is slower than reading from the on-chip BRAMs, the relative performance improvement of the sort kernel shrinks for inputs larger than 128K elements (from 20× in the case of sorting 128K elements to 6.8× for sorting 256K elements). The quicksort implementation shows a better performance than the multi-threaded block sort for arrays smaller than 256k elements. However, for larger arrays, the block sort provides better performance than quicksort. NASCENT2 sort kernel's performance improvement hovers around ∼ 1.5× for inputs with larger than 1M elements compared to the block sort from the Boost library. SmartSSD is using a relatively small and low-power ∼ 7.5W FPGA. NASCENT2 shows 199× improvement of energy consumption for sorting inputs of 1K elements. With the reduction of the speed-up in larger sequences, the energy improvement saturates at ∼ 14.1× for inputs larger than 1M elements compared to the block sort.

Figure 11 compares the performance of NASCENT2's sort kernel and the CPU baseline when the data resides in the SSD. When the data is available in the DRAM, CPU can readily prefetch a major portion of the inputs into the cache and thereby has better performance compared to when it reads

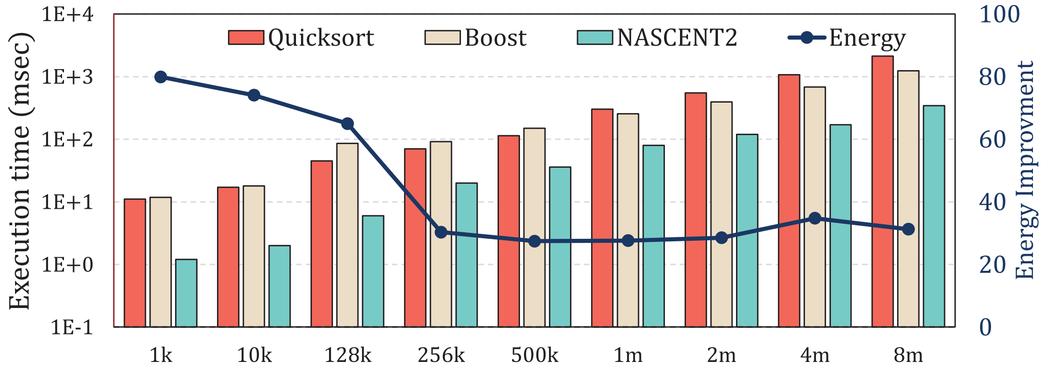

Fig. 11. Execution time and relative energy efficiency of the NASCENT2 *sort kernel* compared to the quicksort and multi-threaded block sort from the Boost library running on the CPU baseline when the data is stored in the storage devices.

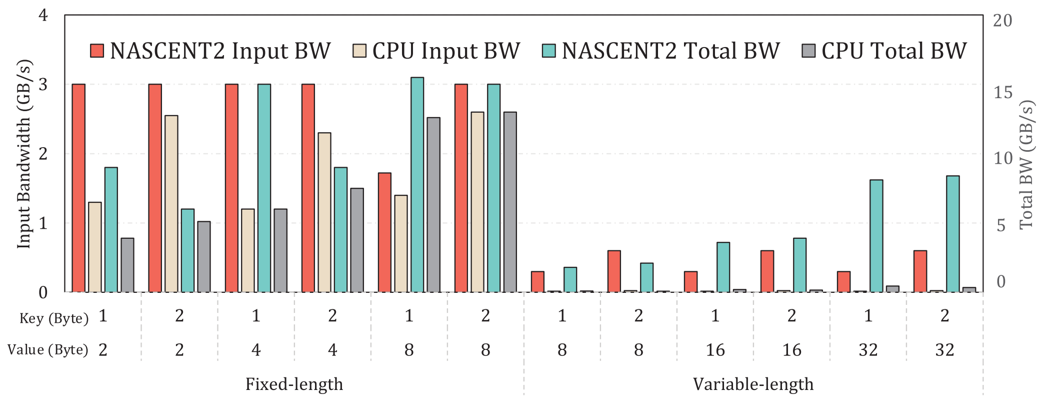

Fig. 12. Input and total bandwidth of the NASCENT2 *dictionary decoder kernel* for 2-, 4-, and 8-byte fixed-length data types and variable-length data types with maximum length 8, 16, and 32 bytes for input bit widths of 1 and 2 bytes.

the data from the SSD, for which the SSD-to-DRAM latency cannot be hidden since it is larger than the computation latency. Thus, the NASCENT2 sort kernel is faster when both platforms read the data from storage. The sort kernel of NASCENT2 is 9.2× faster for 1K data chunks and saturates at ∼ 3.6× when reading and sorting 8M elements. The energy consumption (excluding the SSD energy) also similarly increases from 31.2× to 80×.

Dictionary Decoder Kernel. Figure 12 shows the performance of the NASCENT2 dictionary decoder kernel as compared to multi-core execution of the dictionary decoder on CPU for 1- and 2-byte data pages and outputs with 2, 4, and 8 byte widths. Both NASCENT2 and CPU implementation directly read the data page and dictionary page from the storage system and temporarily store them into the device DRAM, decode the input, and write the decoded data into the device DRAM. Since the dictionary decoding is only beneficial when the bit width of the encoded values is less than the original value, we only consider 1- and 2-byte inputs. Note that if the size of the dictionary becomes greater than 64k unique elements (2-byte inputs), the database management system will not use dictionary encoding and stores the plain data. As the dictionary decoding is an

I/O-bounded application, we measured the performance as the input bandwidth from the storage devices to the computing platform (SmartSSD or CPU). The performance target is fully utilizing the SSD bandwidth to the computing platform, capped at 3 GB/s. Additionally, the total bandwidth shows the DRAM to FPGA/CPU bandwidth, including reading the data page and writing the decoded data to the DRAM. In Figure 12, the left axis shows the input bandwidth and the right axis shows the total bandwidth from DRAM to the computing platform.

The NASCENT2 dictionary decoder provides higher performance in decoding fixed-length data types. When the lengths of the decoded elements are fixed, NASCENT2 parallelizes the dictionary decoding, by instantiating multiple copies of the dictionary table, to maximize the performance. When the lengths of the decoded elements are not fixed, NASCENT2 decodes a single element per cycle. In fixed-length decoding, the NASCENT2 dictionary decoder kernel in all the cases, except for the 1-byte input and 8-byte output case, achieves 3 GB/s input bandwidth, which saturates the SSD-to-FPGA bandwidth. When the data page is 1-byte encoded data, and the values are 8-byte data, the output size would be 8× of the input; consequently, the total bandwidth reaches the maximum DRAM-to-FPGA bandwidth to write the decoded values. Therefore, it cannot saturate the input bandwidth due to the DRAM-to-FPGA bandwidth limitation. In this case, the kernel achieves 1.8-GB/s SSD-to-FPGA bandwidth. The multi-core CPU implementation of the dictionary decoder is unable to saturate the CPU-to-SSD bandwidth in most cases. In fixed-length data types, the number of dictionary decodings per second, when running on CPU, is independent of the input bit width (1- and 2-byte inputs) and consequently of the dictionary size since the dictionary access has constant time as the dictionary tables can fit into the CPU cache. Therefore, the input bandwidth for 2-byte inputs is double that for the 1-byte inputs.

In decoding variable-ength data types, the NASCENT2 dictionary decoder kernel decodes an element per cycle, thereby achieving 300-MB/s input bandwidth for 1-byte input data and 600 MB/s for 2-byte input data. In Figure 12, we tested the performance of both the NASCENT2 and CPU dictionary decoder for randomly generated string sequences. We used strings with maximum length of 8, 16, and 32 bytes to show the effectiveness of NASCENT2 in decoding string values with different lengths. The total memory bandwidth depends on the size of the output data (since the length of the strings are different). As shown in Figure 12, for string values with maximum length 8 bytes, NASCENT2 achieves 1.8-GB/s total bandwidth. In decoding strings with maximum length 32 bytes, NASCENT2 achieves 8.1-GB/s total bandwidth. Since the number of dictionary decoding per second is constant, the total bandwidth only depends on the bit width of the inputs and outputs. The CPU implementation of variable-length dictionary decoding cannot be parallelized, and thus it achieves lower performance than the fixed-length decoding. On average, the NASCENT2 dictionary decoder provides 1.4× higher input bandwidth in fixed-length decoding as compared to the CPU implementation. NASCENT2 shows higher performance improvement in decoding variable-length values, and it provides 21.4× higher input bandwidth compared to the CPU implementation.

NASCENT2 uses the dictionary decoder for both integer and generic sort. In integer sort, to eliminate the host involvement, the dictionary decoder first decodes the data and then the sort kernel sorts the decoded data. Figure 13 shows the breakdown of the execution time of sorting an integer column stored in the dictionary encoded format in the storage system. The figure shows two cases when 8-bit numbers are decoded to 32-bit and 64-bit numbers. First, the NASCENT2 dictionary decoder kernel decodes the data to the 32-bit and 64-bit numbers, and then it sorts the decoded column. The NASCENT2 sort kernel can sort 64-bit long numbers with minimal changes in the CS modules. Due to FPGA resource limitation, both 32-bit and 64-bit NASCENT2 sort kernels utilize the same amount of BRAMs; therefore, the 64-bit sort kernel fits up to 64k *long* (64-bit) numbers in the on-chip memory, as opposed to fitting 128k 32-bit elements. For larger input arrays,

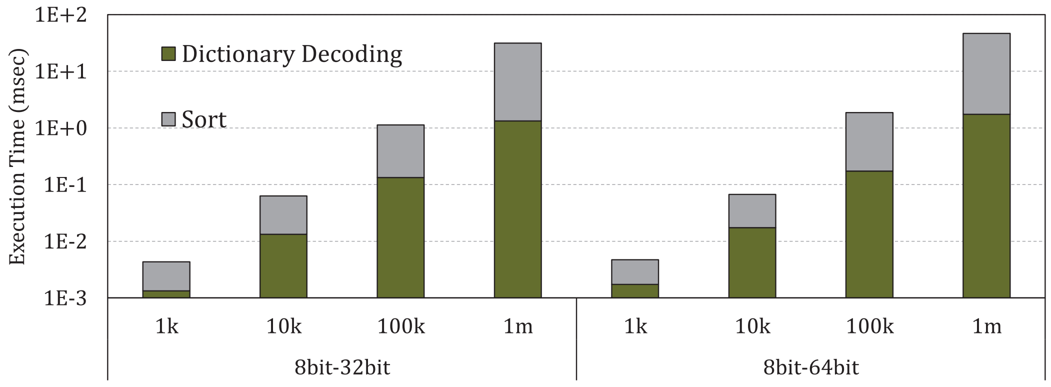

Fig. 13. Execution time for dictionary decoding and sorting an integer column with different numbers of rows when the stored data are 8-bit numbers and the outputs are 32-bit or 64-bit numbers. The *Y* axis is logarithmic.

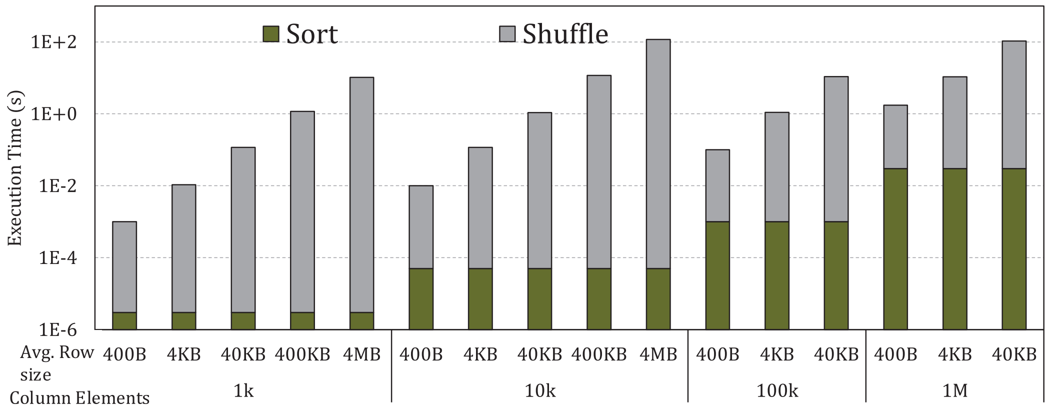

Fig. 14. Execution time for sorting tables with different numbers of rows and columns. The *Y* axis is logarithmic.

the NASCENT2 sort kernel uses off-chip DRAM to store the partially sorted arrays. For sorting input arrays smaller than 64k elements, both 32-bit and 64-bit NASCENT2 sort kernels deliver the same performance. For larger input sizes, the 64-bit NASCENT2 sort kernel provides slightly lower performance than the 32-bit sort kernel due to higher DRAM access. As illustrated in Figure 13, the execution time of the NASCENT2 dictionary decoder kernel linearly increases with the size of the input array since the dictionary decoder kernel performance is data independent.

Shuffle Kernel. Figure 14 shows the breakdown of the execution time of NASCENT2 when sorting database *tables* of various sizes when the plain data is stored in the storage system. We generated static tables with a different number of rows and columns from 1k to 1M. Note that the content of the columns is *not* limited to integer types and can be any type of variable or strings. For tables with 100k and 1M rows, we only considered 1k and 10k columns, as otherwise the table size becomes larger than the typical size of the partitions. For tables with the same number of rows, the sort kernel takes exactly the same time since the bitonic sort execution time is data independent. For a given number of rows, the execution time of the shuffle kernel increases with the number

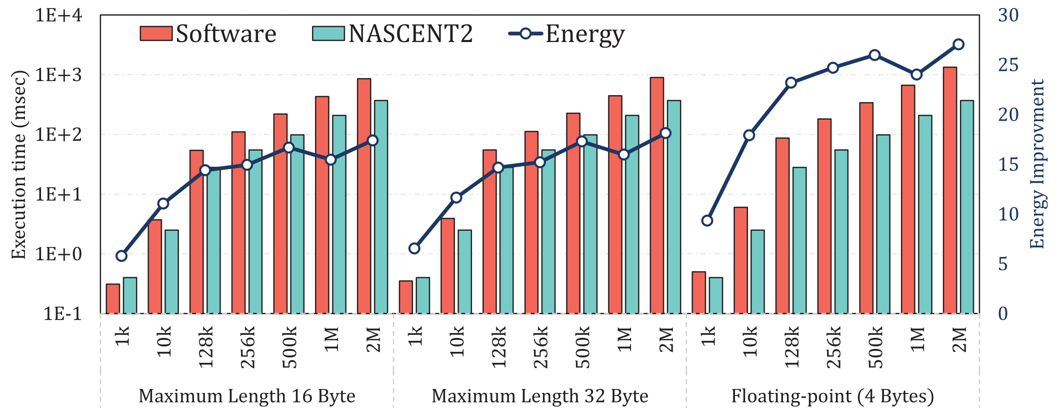

Fig. 15. Execution time for sorting string columns with different numbers of rows for two different maximum lengths of the strings. The *Y* axis is logarithmic.

of columns. Due to the fact that the overall size of the table is significantly larger than the size of the input sequence to the sort kernel (which deals with one column, i.e., the key column), the execution time of the shuffle kernel dominates the total time. The shuffle kernel fully utilizes the bandwidth of the PCIe bus from the SSD to the FPGA to minimize the shuffling time. Thus, the execution time of NASCENT2 increases almost linearly with the size of the table.

Generic Sort. To evaluate the performance of NASCENT2 generic sort, we used 4-byte floating-point and string columns with different sizes from 1,000 (1k) elements to 2,000,000 (2M) elements. In string columns, each element is a randomly generated string with maximum string lengths of 16 and 32 bytes. Columns are dictionary encoded and stored as 16-bit integers, namely the dictionary page has 64,000 unique elements. Figure 15 shows the execution of sorting the columns on CPU and NASCENT2 generic sort (left axis). It also shows the performance improvement and energy efficiency of NASCENT2 over the CPU baseline (right axis). Note that for the CPU implementation, we first decode the column and then use Python built-in string sort function to sort the decoded column. NASCENT2 generic sort shows slightly less performance when the data page size is relatively small compared to the dictionary page. For a key column with 1k elements, NASCENT2 is 20% slower than sorting the column on CPU, due to NASCENT2's overhead of reading the dictionary page and sorting it on the host server. Although NASCENT2 is slightly slower in sorting a column with 1k elements, it still increases the energy efficiency by 5.8×. For columns bigger than 1k elements, which is more often in real-world databases, NASCENT2 provides higher performance than the CPU baseline. On average, it delivers 2× higher performance and 15× higher energy efficiency than the CPU baseline. In sorting floating-point columns, NASCENT2 delivers 3.2× speedup and 23.8× energy efficiency as compared to sorting the column on CPU. The efficiency of the proposed generic sort comes from the fact that NASCENT2 sorts the integer key columns and then decodes the sorted column while the CPU is first decoding the column and then sorts a column of strings. As illustrated in the figure, the execution time of NASCENT2 is independent of the maximum string length of the column elements, whereas the execution time of the CPU baseline is greater for the case with maximum string length of 32 bytes.

4.3 System Evaluation

To evaluate the scalability of NASCENT2, in Figure 16 we show the execution time of the CPU, typical FPGA-equipped systems (see Figure 3), and NASCENT2 when the number of SSD instances

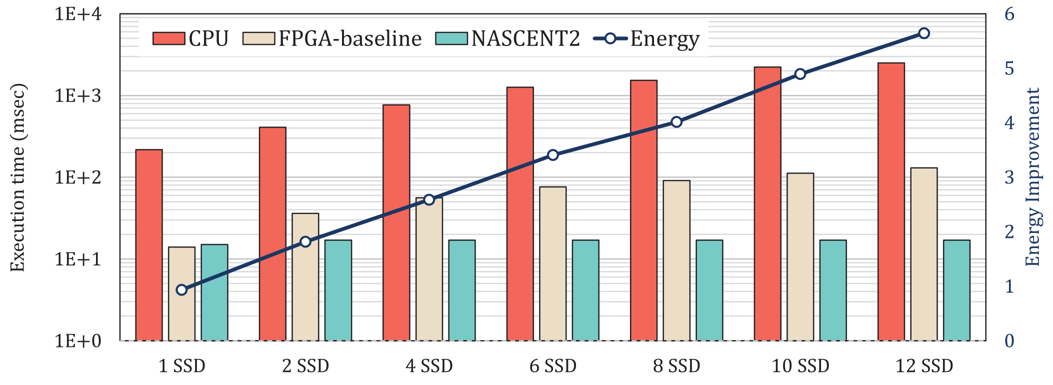

Fig. 16. Execution time of NASCENT2 as compared to the CPU and FPGA baseline for sorting 1024×1024 tables, each stored in an SSD. The left Y axis is in logarithmic scale.

increases from 1 to 12 (12 SSDs is the limitation incurred by the number of slot counts of the host machine). For the CPU baseline, we use quicksort (as it is faster than block sort for 1k arrays) and multi-threaded implementation of shuffle and dictionary decoder. Each SSD contains a table with 1024 rows and an average row size of 4 KB, which is a typical table size in SparkSQL partitions. Originally, the key columns consist of 32-bit integer numbers, but the key column in the storage system is stored as 16-bit dictionary encoded elements. Although in real-world applications different SSDs would sort different sizes of tables, here we assume all the tables have the same dimensions and size. As we showed in Figure 14, the execution time of NASCENT2 increases linearly with the size of the table (for a specific number of rows) since it fully utilizes the SSD-to-FPGA bandwidth. Each SSD contains multiple tables that are going to be sorted. Note that the bitonic sort's performance is data independent, and sort operations on different SSDs are executed independently. Thus, we can assume that all SSDs contain the same table without loss of generality.

As Figure 16 reveals, the FPGA-equipped system baseline and SmartSSD are both faster than the CPU baseline. The bottleneck of all the platforms is the storage bandwidth, and the memory hierarchy of the processor increases the execution time. Comparing the FPGA-equipped system with SmartSSD, when the system has only one storage device, the stand-alone FPGA shows slightly better performance as it is larger than the SmartSSD's FPGA, so it contains more kernels.[1] Nevertheless, as the number of storage devices increases, the execution time of NASCENT2 remains the same, as it sorts the tables independently. The CPU and FPGA baselines, however, are not able to parallelize the operations on different SSDs, and consequently their runtime increases linearly with the number of SSDs. In SmartSSD, every storage device is equipped with an FPGA, so it consumes more power than a conventional SSD. However, the power consumption of the SSD is higher than the FPGA's power, which shrinks the per-device overhead of SmartSSD. In Figure 16, we also show the energy efficiency of NASCENT2 versus the FPGA-equipped system (FPGA baseline). As the number of storage devices increases, both the performance and energy efficiency of NASCENT2 also improve. With 12 SmartSSDs, NASCENT2 is 7.6× (147.2×) faster and 5.6× (131.4×) more energy efficient than the FPGA (CPU) baseline.

Figure 17 shows the efficiency of NASCENT2 compared to an FPGA-equipped system and a multi-threaded software implementation on CPU. The speedup and energy efficiency of

[1]The baseline FPGA-equipped system uses from the Xilinx's Alveo U250 with 1,728K LUTs (compared to 391K in SmartSSD's FPGA) 64 GB of DRAM, 77 GB/s of DRAM-to-FPGA bandwidth, and on-chip BRAMs totaling 57 MB (compared to 16 MB in SmartSSD's FPGA).

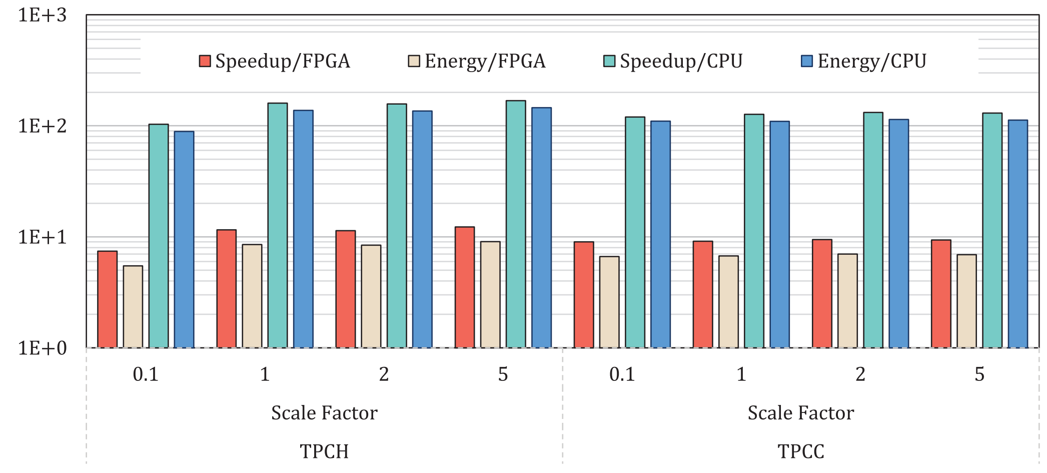

Fig. 17. Execution time of NASCENT2 compared to the FPGA-equipped baseline storage when sorting multiple copies of the largest table of TPCH and TPCC benchmarks on 12 storage devices. The *scale factor* denotes the scaling up of the number of benchmark rows.

NASCENT2 compared to the FPGA-equipped (CPU SW) system is denoted as Energy/FPGA (Energy/CPU) and Speedup/FPGA (Speedup/CPU), respectively. In this experiment, we sort a copy of the largest tables of TPCH (line-item) and TPCC (order-line) benchmarks in each SSD [64, 66]. As explained earlier, the performance of sorting the table of the same size is data independent, so having multiple copies of the same benchmark is analogous to having same-size tables with different entries. We evaluate the performance of NASCENT2 when sorting the largest tables on the TPCH and TPCC benchmarks for four different scale factors $\{0.1, 1, 2, 5\}$ that scale the number of rows. We limit our experiments to scale factor 5 since, in most cases, data processing management systems partition each table into small partitions that can be executed independently. The *lineitem* table in the TPCH benchmark with scale factor of 5 is $\sim 3.2GB$. Partition sizes are rarely larger than this. Compared to the FPGA baseline, NASCENT2 is on average 9.2× faster and has 6.8× less energy consumption when using 12 SmartSSDs to run the TPCC benchmark and 10.6× faster and 7.8× lower energy consumption for the TPCH benchmark. NASCENT2 shows higher performance improvement in executing shuffle operation. The average size of the rows in the TPCH table is greater than in the TPCC table, so the performance improvement of NASCENT2 for the TPCH benchmark is slightly higher than that for the TPCC benchmark. The shuffle operation is more dominant in the TPCH benchmark. NASCENT2 shows roughly constant improvement as the table scales. The performance of sort kernel does not scale linearly, so the overall improvement, which is dominated by shuffling performance, is near-constant. NASCENT2 compared to multi-threaded execution of table sort on CPU shows 127.4× and 147.1× speedup as well as 111.5× and 126.9× energy reduction in TPCC and TPCH benchmarks, respectively.

5 CONCLUSION AND FUTURE WORK

In this article, we present NASCENT2, a near-storage sort accelerator for data analytics on SmartSSD based on the bitonic sort. The proposed generic sort method supports sorting the table based on integer and non-integer key columns. It shows 2× (3.2×) speedup and 15.2× (23.8×) energy reduction in sorting a string (floating-point) column as compared to CPU. Moreover, NASCENT2 tackles the data transfer limitations in current interface connections between storage devices and computation platforms. NASCENT2 comprises FPGA-based accelerators with specific kernels to accelerate dictionary decoder, sort, and subsequent shuffling operations to sort a database table. NASCENT2 increases the scalability of computer systems by enabling simultaneous operations on different storage devices. With 12 SmartSSDs, NASCENT2 is 9.9× faster and 7.3× more energy efficient than the same accelerator on conventional architectures comprising a stand-alone FPGA and storage devices. NASCENT2 also shows 137.2× speedup and 119.2× energy reduction as compared to sorting the database table on the host CPU. In our future work, we are going to integrate NASCENT2 with SparkSQL, which is one of the most widely used data processing management systems. In this work, we showed end-to-end execution of table sort on SmartSSD with two orders of magnitude energy reduction, which highlights the potential of significant speedup and energy reduction in end-to-end execution of a query when NASCENT2 is integrated into SparkSQL.

ACKNOWLEDGMENTS

We thank the anonymous reviewers whose valuable comments and suggestions helped improve and clarify this work.

REFERENCES

[1] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. 2003. *Database Management Systems*. Vol. 3. McGraw-Hill, New York, NY.

[2] Dinesh Kumar and Mihir Narayan Mohanty. 2019. A survey: Classification of big data. In *Cognitive Informatics and Soft Computing*. Springer, 299–306.

[3] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jing, and Steven J. Swanson. 2015. *Gullfoss: Accelerating and Simplifying Data Movement Among Heterogeneous Computing and Storage Resources*. Department of Computer Science and Engineering, University of California.

[4] Zhenyuan Ruan and Tong He Jason Cong. 2019. Analyzing and modeling in-storage computing workloads on EISC—An FPGA-based system-level emulation platform. In *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'19)*. IEEE, Los Alamitos, CA, 1–8.

[5] Gunjae Koo, Kiran Kumar Matam, I. Te, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. IEEE, Los Alamitos, CA, 219–231.

[6] Mahsa Bayati, Janki Bhimani, Ronald Lee, and Ningfang Mi. 2019. Exploring benefits of NVMe SSDs for BigData processing in enterprise data centers. In *Proceedings of the 2019 5th International Conference on Big Data Computing and Communications (BIGCOM'19)*. IEEE, Los Alamitos, CA, 98–106.

[7] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 327–341.

[8] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. {INSIDER}: Designing in-storage computing system for emerging high-performance drive. In *Proceedings of the 2019 Annual Technical Conference (USENIX ATC'19)*. 379–394.

[9] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review* 32, 5 (1998), 81–91.

[10] Samsung. n.d. SmartSSD. Retrieved May 27, 2020 from https://samsungsemiconductor-us.com/smartssd/.

[11] ScaleFlux. n.d. Home Page. Retrieved May 27, 2020 from http://www.scaleflux.com/.

[12] ARM. 2019. *White Paper: Smarter Data Storage—A Guide to Computational Storage on ARM*. Technical Report. ARM.

[13] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–7.

[14] Cristian Zambelli, Riccardo Bertaggia, Lorenzo Zuolo, Rino Micheloni, and Piero Olivo. 2019. Enabling computational storage through FPGA neural network accelerator for enterprise SSD. *IEEE Transactions on Circuits and Systems II: Express Briefs* 66, 10 (2019), 1738–1742.

[15] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, et al. 2020. Cost-effective, energy-efficient, and scalable storage computing for large-scale AI applications. *ACM Transactions on Storage* 16, 4 (2020), 1–37.

[16] Keith Chapman, Mehdi Nik, Behnam Robatmili, Shahrzad Mirkhani, and Maysam Lavasani. 2019. Computational storage for big data analytics. In *Proceedings of 10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19)*.

[17] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.

[18] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.

[19] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[20] Alberto Lerner, Rana Hussein, André Ryser, Sangjin Lee, and Philippe Cudré-Mauroux. 2020. Networking and storage: The next computing elements in exascale systems? *IEEE Data Engineering Bulletin* 43 (2020), 60–71.

[21] André DeHon. 2000. The density advantage of configurable computing. *Computer* 33, 4 (2000), 41–49.

[22] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. 2020. SmartSSD: FPGA accelerated near-storage data analytics on SSD. *IEEE Computer Architecture Letters* 19, 2 (2020), 110–113.

[23] Sahand Salamat, Behnam Khaleghi, Mohsen Imani, and Tajana Rosing. 2019. Workload-aware opportunistic energy efficiency in multi-FPGA platforms. *arXiv preprint arXiv:1908.06519* (2019).

[24] Sparsh Mittal. 2020. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Computing and Applications* 32, 4 (2020), 1109–1139.

[25] Sahand Salamat, Sumiran Shubhi, Behnam Khaleghi, and Tajana Rosing. 2021. Residue-Net: Multiplication-free neural network by in-situ no-loss migration to residue number systems. In *Proceedings of the 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC'21)*. IEEE, Los Alamitos, CA, 222–228.

[26] Sahand Salamat, Mohsen Imani, and Tajana Rosing. 2020. Accelerating hyperdimensional computing on FPGAs by exploiting computational reuse. *IEEE Transactions on Computers* 69, 8 (2020), 1159–1171.

[27] Sahand Salamat, Hui Zhang, Joo Hwan Lee, and Yang Seok Ki. 2021. System and method for hierarchical sort acceleration near storage. US Patent App. 16/821,811. April 29, 2021.

[28] Veronica Lagrange Moutinho dos Reis, Harry Li, and Anahita Shayesteh. 2020. Modeling analytics for computational storage. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 88–99.

[29] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogrifDB: Balancing I/O and GPU bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.

[30] Phil Francisco. 2014. IBM PureData system for analytics architecture. *IBM Redbooks* 2014, 1–16.

[31] Goetz Graefe. 2006. Implementing sorting in database systems. *ACM Computing Surveys* 38, 3 (2006), 10–es.

[32] Mikhail Asiatici, Damian Maiorano, and Paolo Ienne. 2020. FPGAs in the datacenters: The case of parallel hybrid super scalar string sample sort. In *Proceedings of the 2020 IEEE 31st International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'20)*. IEEE, Los Alamitos, CA, 133–140.

[33] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. 2019. Mostly order preserving dictionaries. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE'19)*. IEEE, Los Alamitos, CA, 1214–1225.

[34] Ionut Boicu. 2019. *Adaptive On-the-Fly Compressed Execution in Spark*. Master's Thesis. Vrije Universiteit Amsterdam.

[35] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-storage acceleration of database sort on SmartSSD. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[36] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. 2017. ExtraV: Boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1706–1717.

[37] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, and Shuotao Xu. 2016. BlueDBM: Distributed flash storage for big data analytics. *ACM Transactions on Computer Systems* 34, 3 (2016), 1–31.

[38] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 153–165.

[39] Shuyi Pei, Jing Yang, and Qing Yang. 2019. REGISTOR: A platform for unstructured data processing inside SSD storage. *ACM Transactions on Storage* 15, 1 (2019), 1–24.

[40] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active disk meets flash: A case for intelligent SSDs. In *Proceedings of the 27th International ACM Conference on Supercomputing*. 91–102.

[41] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113.

[42] Dongchul Park, Jianguo Wang, and Yang-Suk Kee. 2016. In-storage computing for Hadoop MapReduce framework: Challenges and possibilities. *IEEE Transactions on Computers.* Early access, July 28, 2016.

[43] Mahdi Torabzadehkashi, Siavash Rezaei, Ali HeydariGorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Computational storage: An efficient and scalable platform for big data and HPC applications. *Journal of Big Data* 6, 1 (2019), 1–29.

[44] Benjamin Y. Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. 2013. XSD: Accelerating MapReduce by harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing.*

[45] Louis Woods, Zsolt Istvan, and Gustavo Alonso. 2013. Hybrid FPGA-accelerated SQL query processing. In *Proceedings of the 2013 23rd International Conference on Field Programmable Logic and Applications.* IEEE, Los Alamitos, CA, 1.

[46] Ashrak Rahman Lipu, Ruhul Amin, Md. Nazrul Islam Mondal, and Md. Al Mamun. 2016. Exploiting parallelism for faster implementation of Bubble sort algorithm using FPGA. In *Proceedings of the 2016 2nd International Conference on Electrical, Computer, and Telecommunication Engineering (ICECTE'16).* IEEE, Los Alamitos, CA, 1–4.

[47] Dwi Marhaendro Jati Purnomo, Ahmad Arinaldi, Dwi Teguh Priyantini, Ari Wibisono, and Andreas Febrian. 2016. Implementation of serial and parallel bubble sort on FPGA. *Jurnal Ilmu Komputer dan Informasi* 9, 2 (2016), 113–120.

[48] Rui Marcelino, Horácio Neto, and Joao M. P. Cardoso. 2008. Sorting units for FPGA-based embedded systems. In *Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems.* 11–22.

[49] Wojciech M. Zabołotny. 2011. Dual port memory based heapsort implementation for FPGA. In *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2011*, Vol. 8008. International Society for Optics and Photonics, 80080E.

[50] Bashar Romanous, Mohammadreza Rezvani, Junjie Huang, Daniel Wong, Evangelos E. Papalexakis, Vassilis J. Tsotras, and Walid Najjar. 2020. High-performance parallel radix sort on FPGA. In *Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'20).* IEEE, Los Alamitos, CA, 224–224.

[51] Xingyu Liu and Yangdong Deng. 2014. Fast radix: A scalable hardware accelerator for parallel radix sort. In *Proceedings of the 2014 12th International Conference on Frontiers of Information Technology.* IEEE, Los Alamitos, CA, 214–219.

[52] 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 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20).* IEEE, Los Alamitos, CA, 282–294.

[53] Chi Zhang, Ren Chen, and Viktor Prasanna. 2016. High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'16).* IEEE, Los Alamitos, CA, 148–155.

[54] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, and Norman May. 2020. Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA. In *Proceedings of the 16th International Workshop on Data Management on New Hardware.* 1–7.

[55] Dirk Koch and Jim Torresen. 2011. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 45–54.

[56] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting networks on FPGAs. *VLDB Journal* 21, 1 (2012), 1–23.

[57] Amirshahram Hematian, Suriayati Chuprat, Azizah Abdul Manaf, and Nadia Parsazadeh. 2013. Zero-delay FPGA-based odd-even sorting network. In *Proceedings of the 2013 IEEE Symposium on Computers and Informatics (ISCI'13).* IEEE, Los Alamitos, CA, 128–131.

[58] Ren Chen, Sruja Siriyal, and Viktor Prasanna. 2015. Energy and memory efficient mapping of bitonic sorting on FPGA. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 240–249.

[59] Aidan O. Mahony and Emanuel Popovici. 2018. Power analysis of sorting algorithms on FPGA using OpenCL. In *Proceedings of the 2018 29th Irish Signals and Systems Conference (ISSC'18).* IEEE, Los Alamitos, CA, 1–6.

[60] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. FPGA-accelerated samplesort for large data sets. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 222–232.

[61] Sang-Woo Jun, Shuotao Xu, and Arvind. 2017. Terabyte sort on FPGA-accelerated flash storage. In *Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17).* IEEE, Los Alamitos, CA, 17–24.

[62] Timo Bingmann and Peter Sanders. 2013. Parallel string sample sort. In *Proceedings of the European Symposium on Algorithms.* 169–180.

[63] Kenneth E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference.* 307–314.

[64] TPC. n.d. TPC-H. Retrieved January 21, 2021 from http://www.tpc.org/tpch/.

[65] C++ Libraries. Home Page. Retrieved April 2, 2021 from https://www.boost.org/.

[66] TPC. n.d. TPCC Benchmark. Retrieved May 27, 2020 from http://www.tpc.org/tpcc/.

Received January 2021; revised June 2021; accepted June 2021