NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD

Sahand Salamat, Armin Haj Aboutalebi*, Behnam Khaleghi, Joo Hwan Lee*, Yang Seok Ki*, Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093 *Samsung Semiconductor Inc., San Jose, CA 95134
{sasalama, bkhaleghi, tajana}@ucsd.edu {armin.h, joohwan.lee, yangseok.ki}@samsung.com

ABSTRACT

As the size of data generated every day grows dramatically, the computational bottleneck of computer systems has been shifted toward the storage devices. The interface between the storage and the computational platforms has become the main limitation as it provides limited bandwidth which does not scale when the number of storage devices increases. Interconnect networks do not provide simultaneous accesses to all storage devices and thus limit the performance of the system when independent operations on different storage devices. Offloading the computations to the storage devices eliminates the burden of data transfer from the interconnects. Emerging as a nascent computing trend, near storage computing offloads a portion of computation to the storage devices to accelerate the big data applications. In this paper, we propose a near storage accelerator for database sort, NASCENT, which utilizes Samsung SmartSSD, an NVMe flash drive with an on-board FPGA chip that processes data in-situ. We propose, to the best of our knowledge, the first near storage database sort based on bitonic sort which considers the specifications of the storage devices to increase the scalability of computer systems as the number of storage devices increases. NASCENT improves both performance and energy efficiency as the number of storage devices increases. With 12 SmartSSDs, NASCENT is 7.6× (147.2×) faster and 5.6× (131.4×) more energy efficient than the FPGA (CPU) baseline.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; *Emerging architectures*; • **Computer systems organization** → Reconfigurable computing.

ACM Reference Format:
Sahand Salamat, Armin Haj Aboutalebi*, Behnam Khaleghi, Joo Hwan Lee*, Yang Seok Ki*, 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 (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431920.3439298

FPGA '21, February 28-March 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439298

1 INTRODUCTION

With the explosive growth of data, processing the massive amount of data has become the cornerstone of many big data use-cases such as database 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 since more than half of the execution time is spent on data transfer which highlights 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]. The PCIe provides limited simultaneous accesses to the storage devices, which limits the scalability of the system when independent operations are called on different storage devices in parallel. This issue along with low performance of the interconnect bus increase the gap between the performance capacity of storage devices and the interconnection buses [4, 6] that obliges us to move the computations closer to where the data is stored, which has been empowered by recent advances in near-storage computing devices [5, 7–12].

Near-storage computing offloads a portion of computation to the storage drive to accelerate the big data applications. Accordingly, new devices have been developed to bring the computation power into the flash storage devices, e.g., NGD Systems [9], ScaleFlux [8], and Samsung's SmartSSD [7]. 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. SmartSSD is an NVMe flash drive with an on-board FPGA chip that processes data in-situ. 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) [13–15]. Therefore, FPGAs have become an inevitable part of data centers [16–18]. The speed-up of using SmartSSD over the conventional storage devices is thus two-fold; not only offloading tasks to near-storage nodes increases the overall performance by bridging the interconnection gap, but also the FPGA as an accelerator further boosts the applications with low power consumption. Since the performance of data-intensive applications such as database management is limited by the system bandwidth, these applications can be significantly accelerated by offloading the computations to the storage drive [4, 19, 20]. Therefore, recent processing systems aim to offload the query processing

to storage drive to the greatest possible extent to minimize data transfer between the host and storage [10, 21–23]. Also, 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 release the host resources to execute more compute-intensive tasks.

As the size of the real-world databases is growing, storing databases require multiple storage devices. Database management systems partition databases into multiple partitions and breakdown the operations into multiple independent operations on the partitioned database. Although these independent operations can be executed in parallel, due to storage-to-host bandwidth limitation in I/O bound applications, host processors cannot fully utilize the partitioning opportunity. However, in computational storage devices, each storage device has its own computation resource; hence, it can perform the independent operations in-situ without occupying the storage-to-host bandwidth. In particular, sort operation is commonly used in database query processing as a standalone operation or as the backbone of more complex database operations such as merge-join, distinct, order-by, group-by, etc. [24]. When sorting a database, all the table columns are sorted based on a single column, dubbed *key column*. Due to their large number of columns, real-world databases are complicated to sort since after sorting the key column, the rest of the table needs to be shuffled accordingly. Most database management systems often use data encoding to compress the stored data into the storage devices. Being vastly used in database systems, dictionary encoding is a lossless one-to-one compression method that replaces attributes from a large domain with small numbers [14, 25, 26]. To sort the database, if the data is stored in the encoded format, the table should be decoded and then sorted.

While conventional systems cannot exploit the storage-level parallelism as they do not provide access to all the storage devices simultaneously, Computational storage devices offer independent operations on data stored in each storage device. In this paper, to sort the database tables, we propose near-storage sort using SmartSSDs that comprise FPGA-based accelerators with specific kernels to accelerate *dictionary decoding*, *sort*, and the subsequent *shuffle* operations. If the table is stored in the encoded format, the NASCENT dictionary decoding 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. NASCENT not only inherently addresses the data transfer issue by carrying out computations near the storage system but also embraces an FPGA-friendly implementation of dictionary decoding, sort, and shuffle operations. The summary of the contributions of the paper is listed as follows.

- We present NASCENT, 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.
- Database management systems often encode the data using dictionary encoding to compress the data. NASCENT consists of a dictionary decoding kernel to decode the data at the first stage of the database sort to provide the input to the sort kernel. NASCENT dictionary decoding kernel fully utilizes the SSD bandwidth.
- Shuffling is the critical step of database sort and is I/O bounded. NASCENT accomplishes table sort using the shuffle kernel which 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 NASCENT on SmartSSD is NASCENT is 7.6× faster and 5.6× 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 PCIe bus. NASCENT also shows 147.2× speedup and 131.4× energy reduction as compared to the CPU baseline.

2 RELATED WORK

Previous studies on near-storage computing generally can be categorized as works that propose (a) novel architectures, (b) emulation and/or analysis frameworks the 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.

In [6], the authors introduce 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. The authors of [27] 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 which 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 [22]. The platform supports four functions on the FPGA, viz. compress, project, restrict, and visibility, with the capability of expanding to further database operations. As the computational storage devices are in the early stages, the work in [4] provides an emulation platform to estimate the extent an application can benefit by offloading operations on FPGA-enabled computational storage devices.

From the application perspective, the work in [23] examines 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. In [10], the authors explore offloading the list intersection database operation, which is the core of many applications such as search engines on computational storage devices. The work in [28] offloads regular expression (regex) search (a searching algorithm that looks for specific patterns in unstructured data) on computational storage. The accelerator performs a regex search while a file is being transferred to the host.

Speaking of the sort algorithm, several works have attempted to accelerate various sort algorithms on FPGAs [29, 30]. The authors

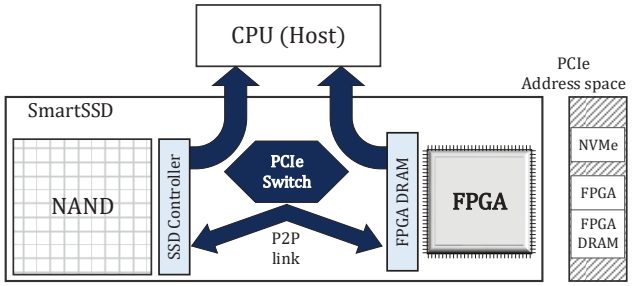

Figure 1: Overview of SmartSSD architecture.

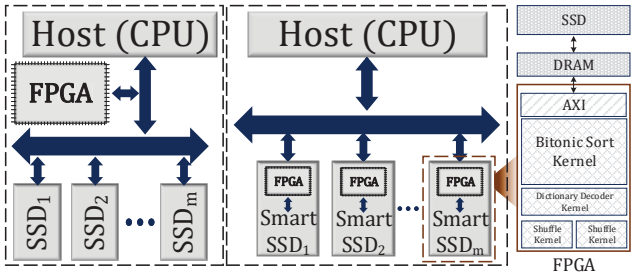

Figure 2: The overall architecture of NASCENT (right) as compared to the conventional systems equipped with an FPGA accelerator (left).

of [31] propose an FPGA-based accelerator for sorting datasets larger than the available on-chip memory of FPGAs (which makes the sort challenging as data needs to be transferred back and forth between the off-chip DRAM and on-chip block RAMs). It partitions the data to smaller chunks where all the elements in the i^{th} chunk are smaller than or equal to the elements of the $i+1^{\text{th}}$ chunk assuming the dataset is being sorted in ascending order. Therefore, each segment can be sorted independently. The work in [30] evaluates the performance of various sorting algorithms on FPGAs, including even-odd [32] and bitonic sorting network [33], as well as traditional bubble and insertion sorts [34]. 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 quicksort), their results show that, in practice, bitonic sort can run faster than the common sort algorithms thanks to its high, FPGA-friendly parallelism. Eventually, the work in [33] proposes an FPGA-based accelerator for bitonic sort. It uses a classic Clos network which is programmable to perform all the permutations required in the bitonic sort algorithm.

Compared to the previous work, to the best of our knowledge, our proposed NASCENT is the first near-storage accelerator for database sort on SmartSSD which performs independent table sort on multiple storage devices simultaneously. NASCENT increases the scalability of the system in the presence of multiple storage devices as compared to a system with a stand-alone FPGA. NASCENT is calibrated to fully utilize the storage bandwidth when executing the dictionary decoder, sort, and shuffle kernels. In contrast to the previous FPGA-based sort accelerators that target maximizing the performance by fully utilizing the DRAM-to-FPGA bandwidth, our challenge is the storage bandwidth which is lower than the DRAM bandwidth. We tackle the I/O bottleneck by prudently allocating the FPGA resources for dictionary decoding kernel, multiple shuffle kernels versus the sort kernel.

3 NASCENT DESIGN

Database systems are largely constrained by disk performance as every operation on the database requires a tremendous amount of data. A database comprises one or more tables, each with rows and columns where each entry holds a specific attribute. Data encoding is frequently used to compress the table stored in the storage system. Dictionary encoding is a common encoding method widely used in database management systems. Unlike byte-oriented compression methods (i.e. gzip, snappy, run-length encoding) that require decompression as a blocking step before query execution, dictionary encoding supports parallel decoding, and in-situ query processing [25]. Sorting a database table based on a key column requires the following three steps. Decompressing the key column, if it is stored in dictionary encoded format; sorting the key column; and reordering the rest of the table correspondingly. NASCENT consists of three types of kernels: dictionary decoding, sort, and shuffle to execute each step. NASCENT performs all the computations on SmartSSD to eliminate host-storage communication. In the following subsections, we describe the NASCENT design.

3.1 SmartSSD Architecture

Figure 1 demonstrates the general architecture of SmartSSD. It consists of the components of a general SSD, SSD controller, and NAND array, as well as an additional 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 4TB 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.

In SmartSSD, the processor 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 request and FPGA DRAM read/write requests via the FPGA driver. In addition to host-driven commands, a SmartSSD device 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 term as 'peer-to-peer (P2P) communication'. As shown in Figure 1, 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 run-time stack such as runtime library, API, compiler, and drivers to implement the FPGA-based designs.

3.2 NASCENT 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 device and

transfers it to the accelerator, or the accelerator may have a P2P communication with the storage device to directly read the data from the storage device. In the former case, the data should pass through the host memory to reach the accelerator memory (FPGA DRAM in this concept). Thus, the latency of transferring the data through the host is significantly larger than when the accelerator directly reads the data from the storage. Also, P2P communication between the accelerator and the storage devices, unlike the former case, does not occupy the host resources for data transfer. Current FPGAs support P2P communication with storage devices. Nonetheless, such an architecture still suffers from performance scalability when data is stored in multiple storage devices. Current databases need multiple devices to store the data. These databases are larger than what current commodity hardware platforms can cope with. Thus, database management systems partition the data into smaller chunks such that the computation nodes can execute the computations on each partition 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 all through the merge tree. Locally sorting each partition is independent of the other partitions; therefore, we can locally 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, NASCENT aims to fully utilize 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 partitions as it cannot simultaneously access different partitions. Thus, NASCENT parallelizes the computations in SSD-level (shown in Figure 2), which is not possible in conventional architecture. In conventional architecture, the FPGA is connected to the storage devices using a PCIe bus which cannot provide simultaneous access to multiple SSDs. NASCENT deploys SmartSSDs, each of which is directly connected to an FPGA. 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.

Since NASCENT comprises sort, shuffle, and dictionary decoder 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; thus, instantiating a single dictionary decoder kernel is sufficient to deliver the maximum performance. A single shuffle kernel cannot fully utilize the SSD-to-FPGA bandwidth due to the fact that, although in NASCENT 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 set the total input consumption rate for all the shuffle kernels to the maximum provided bandwidth of the SSD-to-FPGA to fully utilize bandwidth. 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 can be seen in Figure 2) to fully leverage

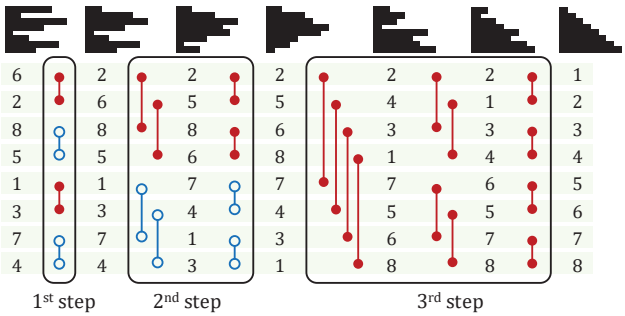

Figure 3: Example of bitonic sort algorithm steps for an array of eight elements.

the storage-to-FPGA bandwidth and a single instance of the dictionary decoder kernel and use the rest of the resources for 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 great portion of the FPGA BRAMs to store the arrays and provide the required parallelism. Additionally, the dictionary decoder kernel requires on-chip memory to store the dictionary table locally to provide high throughput. Therefore, NASCENT dictionary decoder mostly uses FPGA Ultra RAMs (URAMs) to balance the overall resource utilization of NASCENT.

3.3 Bitonic Sort

Bitonic sort, proposed in [35], 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. By merging two bitonic sequences, it 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, i.e.,

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

Figure 3 shows the steps to sort an example input sequence of length $n = 8$ which 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 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 3 the last/third rectangle is the step three and has three sequences of comparisons). In the first sub-step of the i^{th} step, element k is

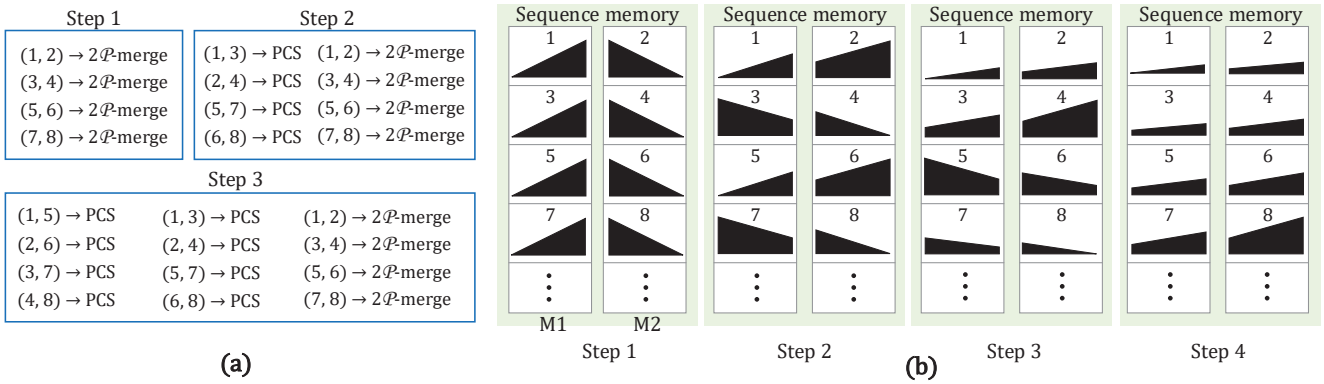

Figure 4: (a) NASCENT scheduling to sort the sequence memory, and (b) the content of the memory at each step.

compared with the element $k + 2^{i-1}$, while 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 the last/third step, the 1^{st} element (has a value of 2) is compared with the $1 + 2^{3-1} = 5^{th}$ 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 $1 + 2^{3-2} = 3^{rd}$ element (which has an value of 3 that is updated in the first sub-step).

3.4 NASCENT Sort Kernel

To sort a database table, NASCENT begins with sorting the *key* column. As mentioned earlier, the sequence of operations in bitonic sort are predefined, data-independent and parallelizable. Therefore, NASCENT takes advantage of FPGA characteristics to accelerate the bitonic sort. The input sequence is stored in the FPGA DRAM, also referred as 'off-chip memory'. Then NASCENT streams the input sequence into the FPGA through the AXI ports which has an interface data width of 512 bits (16 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-two number (we can use padding if the total data elements is not a multiple of $\mathcal{P}$). $\mathcal{P}$ 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 being able to provide a throughput of one sorted sequence (of size $\mathcal{P}$) per cycle. As explained in Section 3.3, 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 $4k + 3$. Afterwards, it compares and swaps $2k$ elements with $2k + 1$ elements of the updated array (see Figure 3). 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. Total number of required CS modules for the $\mathcal{P}$-sorter can be estimated as follows:

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

NASCENT 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 4(b)). Each row of M_1 and M_2 contains $\mathcal{P}$ elements which 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 block RAM which can just fit one or two integers. Since the $2\mathcal{P}$ sequence is just a single bitonic array, using a merging procedure similar to the last (3^{rd}) step of Figure 3, the $2\mathcal{P}$ bitonic array can be sorted using $P \times \log(2\mathcal{P})$ compare-and-swap (CS) units.

Figure 4(a) lists the steps of merging the results of $\mathcal{P}$-sorters and Figure 4(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. That is, similar to the step 1 in bitonic sort, the step 1 in Figure 4(a), (b) merges the adjacent *arrays*. Step 2 of Figure 4 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 NASCENT 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 M_1 and M_2 memory blocks use $\mathcal{P}$ 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). Also, in each memory block, $\mathcal{L}$ rows of BRAMs are placed vertically (e.g., in Figure 4(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

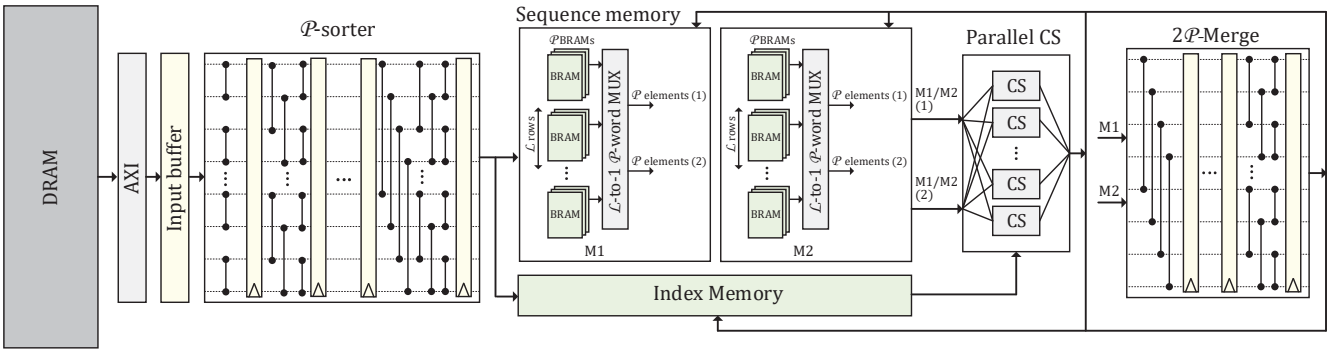

Figure 5: Architecture of the NASCENT bitonic sort kernel.

can be formulated as follow.

$$n_{\text{BRAM}} = 2 \times \mathcal{P} \times \mathcal{L}$$
$$C_{\text{BRAMs}} = 1024 \times 2 \times \mathcal{P} \times \mathcal{L} \quad (3)$$

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, NASCENT reads the corresponding index and concatenates to the value. The compare-and-swap 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. NASCENT therefore, stores the sorted indices in the DRAM, as well.

Figure 5 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 partitioned element-wise for subsequent parallel operations. In the PCS substeps 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 4(a)) are fetched while 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 NASCENT shuffle Kernel

After sorting the key column, NASCENT uses shuffle kernel to reorder the table rows. It reads the value of the first element of the *sorted* key column as well as its index in the original table (which is concatenated to the value of elements). Then it reads all the entries

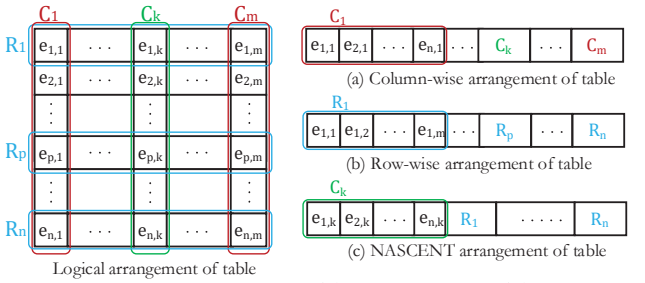

Figure 6: Storing the table in (a) column-wise, (b) row-wise, and (c) our proposed format.

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, NASCENT 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.

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

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 6(a). In the row-wise format, all the elements of a row are placed in successive memory elements (see Figure 6(b)). Consecutive memory elements can be transferred to the FPGA from its DRAM in the burst mode, significantly faster than scattered (random) accesses.

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 6). 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, NASCENT uses a hybrid technique for storing the table in the storage. As shown in Figure 6(c), we store the key column

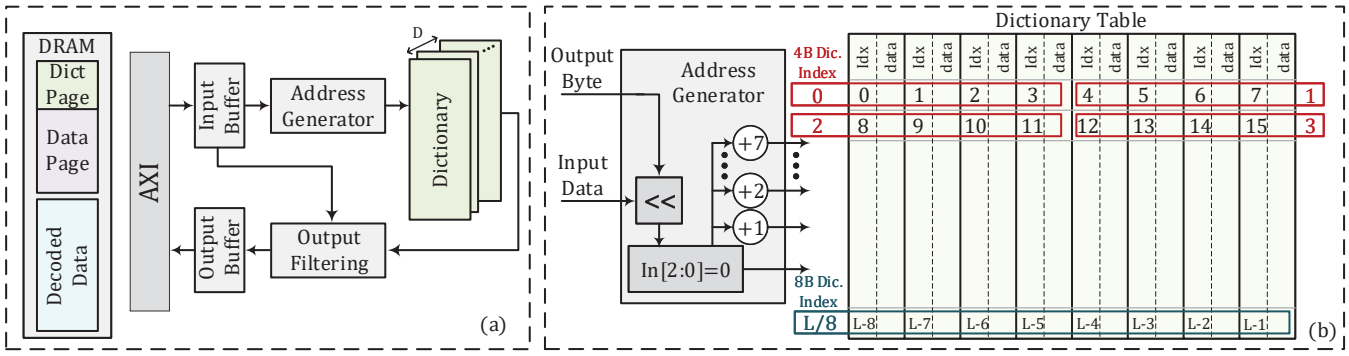

Figure 7: (a)Architecture of the NASCENT dictionary decoder, (b) the generic Byte-addressable dictionary table.

(C_k) column-wise while the rest of the table is stored in row-based format. Therefore, both kernels can benefit from sequential memory accesses.

3.6 NASCENT Dictionary Decoding Kernel

Dictionary encoding is used as a stand-alone compression technique [25] or as a step combined with other compression techniques such as in Parquet [14]. Dictionary encoding is a lossless compression technique that maps each "value" to a "key". Using dictionary encoding is beneficial when the range of the numbers is significantly higher than the number of unique values (U). Each unique n-bit value is represented by a k-bit key where $k = log_2(U)$. Dictionary encoding is beneficial when k is considerably smaller than n. Therefore, database management systems decide whether a table using dictionary encoding is favorable or simply storing and processing the plain data.

If the data is stored in the storage devices in the encoded format, to perform sort operation on the table, the data has to be decoded first. Figure 7 shows the proposed NASCENT dictionary decoder architecture. NASCENT dictionary decoder first reads the "dictionary page", which is stored along with the encoded data, from the storage device. Then it streams in the "data page", decodes it, and writes the decoded data to the FPGA DRAM. As the decoded data will be used in the sort kernel, NASCENT keeps the decoded data into the FPGA DRAM to avoid unnecessary storage accesses. The width of the input elements (k) depends on the number of unique elements in the dictionary (U), and the width of the decoded elements (n) is the same as the original data. NASCENT provides a generic dictionary decoder that supports various input and output bit widths that can be configured during the runtime.

NASCENT dictionary decoder, after loading the dictionary, streams in the data page using the AXI interface. For the sake of design simplicity and AXI compatibility, NASCENT dictionary decoder limits the input and output bit widths (n and k respectively) to power-of-two numbers greater than 8. The AXI interface reads the encoded data page elements and stores them in the "input buffer". To support different output bit widths, the dictionary table has to support the reading and writing element with different bit widths. Since there are multiple accesses to the dictionary table in each clock cycle, NASCENT uses on-chip memory to store the dictionary table. Figure 7(b) shows the dictionary table configuration stored in the FPGA on-chip memory. The Dictionary table is a Byte-addressable memory to support reading and writing elements with different bit widths. Although the proposed architecture for the dictionary decoder is able to support any fixed size of output bit width, in our application, we set the maximum output bit width to 64, i.e., each row of the dictionary table consists of 8 Bytes. In Figure 7(b), the address of each Byte is illustrated (black indices). However, when the application decodes k-bit inputs to n-bit outputs, the input element should be interpreted as the index of the table when storing n-bit elements. For instance, red indices show the table indexing when the dictionary's outputs are 4-Byte elements, and blue index shows the 8-Byte table indexing. Each encoded element in the data page is the index to the dictionary; however, since the NASCENT dictionary table is Byte addressable, NASCENT translates each input to the dictionary table address in the "address generator" module.

The address generator module maps the input element to an address to the dictionary table. For each input, it generates Byte addresses to all the elements in the row that includes the value corresponding to the input key. Therefore, for each input element, NASCENT reads an entire row of the dictionary table. To map the inputs to the Byte addresses, the address generator module shifts the input element to the left for $log_2(\frac{n}{8})$ bits. Since in NASCENT we set the maximum output bit width to 64 bits (8 Bytes), after shifting the input element, the first three bits of the shifted element are set to zero to indicate the address of the first element of the row that contains the value corresponding to the input key. To read an entire row, NASCENT generates addresses to all the 8 elements in the row. NASCENT dictionary decoder writes the entire row into the "output filtering" module where it masks the row using the input key to get the corresponding value. For instance, when the dictionary decoder outputs are 32-bit numbers (4 Bytes) and the input is an 8-bit key equal to 3, the address generator shifts the input to the left for $log_2(\frac{32}{8}) = 2$ bits, the shifted element which is the address to the first Byte of the original value is $3 \times 2^2 = 12$. Setting the first three bits of the Byte address to zero generates the address of the first Byte of the row (8). The address generator outputs addresses from $\{8, 9, \ldots, 15\}$ that includes Bytes $\{12, 13, 14, 15\}$ representing the decoded value. NASCENT dictionary decoder reads the entire Bytes, and then the output filtering module masks elements with addresses $\{12, 13, 14, 15\}$, representing the decoded element correspond to key=3. Then the output filtering selects the Bytes $\{12, 13, 14, 15\}$ and writes these 4 Bytes into the output buffer, which will be then transferred to the off-chip DRAM.

Table 1: Characteristics of SmartSSD resources.

	Storage	LUT	BRAM	DSP	DRAM	D2FPGA BW	S2FPGA BW
SmartSSD	4 TB	391 K	503	1959	4 GB	19 GB/s	3 GB/s

4 EXPERIMENTAL RESULTS

4.1 General Setup

To evaluate the efficiency of NASCENT, we implemented the dictionary decoder, sort, and shuffle kernels on the FPGA available on SmartSSD. Each SmartSSD consists of a 4TB 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. NASCENT kernels are written in C++ and optimized to deliver high performance. The kernels are synthesized using the Vivado High-Level Synthesis tool (HLS) 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 the 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 NASCENT, 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 the deployments.

4.2 Kernel Evaluation

Although a major contribution of NASCENT is carrying out the database sort operation near the storage device with a low-power accelerator and benefit from eliminating the data movement as well as releasing the processor to perform other complex query operations, to solely examine the performance of NASCENT's sort kernel architecture, we compare it with the CPU-based sort baseline. For the baseline CPU sort, we use a C++ implementation of quick-sort, which is generally considered as one of the fastest sort algorithms. The software sort runs on the Intel Core i7-8700 processor with a clock frequency of up to 4.6 GHz.

Figure 8 compares the performance of NASCENT's sort kernel and quick-sort 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 1000 elements (1K) to 8,000,000 elements (8M). The sort kernel of NASCENT consistently delivers higher performance than the CPU implementation.NASCENT sort kernel fits up to 128K elements inside the FPGA on-chip BRAM blocks. Therefore, input sequences smaller than 128K elements will be sorted in a single iteration. For larger number of inputs, NASCENT 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

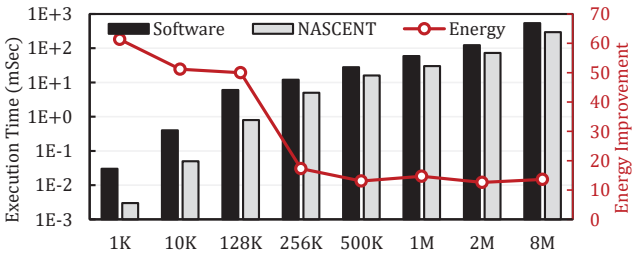

Figure 8: Execution time and relative energy efficiency of NASCENT *sort kernel* compared to the CPU baseline when the data is available in the DRAM of CPU and FPGA. The Y-axis is in logarithmic scale.

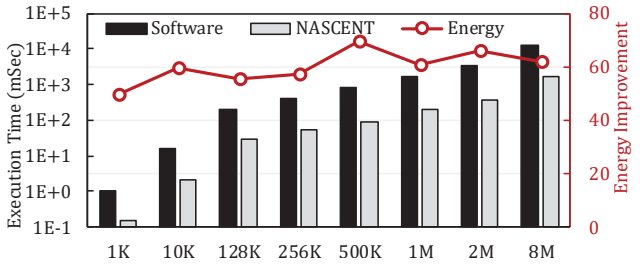

Figure 9: Execution time and relative energy efficiency of NASCENT *sort kernel* compared to the CPU baseline when the data is stored in the storage devices.

BRAMs, the relative performance improvement of the sort kernel shrinks for inputs larger than 128K elements (from 7.5× in the case of sorting 128K elements to 2.4× for sorting 256K elements). The performance improvement hovers around ~ 1.8× for inputs with larger than 1M elements. SmartSSD is using a relatively small and low-power ~ 7.5W FPGA. Therefore, NASCENT shows 61.3× 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 ~ 13.6× for inputs larger than 1M elements.

Figure 9 compares the performance of NASCENT's sort kernel and the CPU baseline when the data resides in SSD. When the data was available in the DRAM, CPU could readily prefetch a major portion of the input elements into the cache and thereby showed better performance compared to when it reads the data from the SSD, for which the SSD-to-DRAM latency cannot be hidden as it is larger than the computation latency. Thus, NASCENT sort kernel shows even better speed-up when both the platforms read the data from storage. The sort kernel of NASCENT shows 6.6× speed-up for a small 1K chunk of inputs, which saturates at ~ 8.25× when reading and sorting 8M elements. The energy consumption (excluding the SSD energy) also similarly increases from 49.7× to 61.6×.

Figure 10 shows the performance of NASCENT dictionary decoder kernel as compared to multi-core execution of the dictionary decoder on CPU for 8-bit and 16-bit data pages and outputs with 16, 32, and 64 bit widths when the data is stored in SSDs. Both NASCENT and CPU dictionary decoder kernels are reading the from the storage devices directly, temporarily store into the device DRAM, decode the input, and write the decoded data into the device DRAM. Since the dictionary decoding is only beneficial when

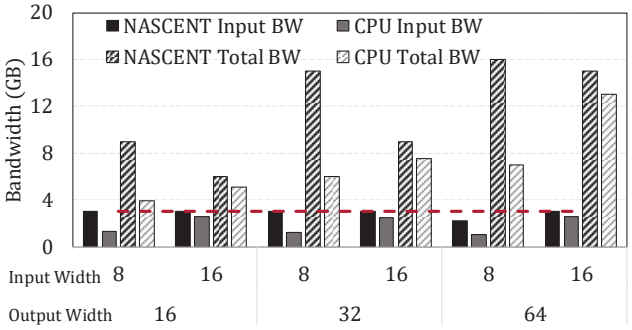

Figure 10: Input and total bandwidth of NASCENT *dictionary decoder kernel* for various range of input bit widths (8 and 16) and output bit widths (16, 32, and 64).

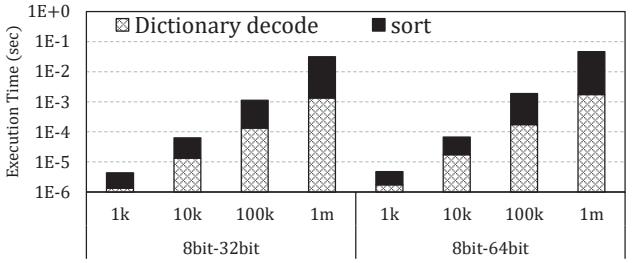

Figure 11: Execution time for dictionary decoding and sorting a column with different number of rows when the stored data are 8-bit numbers and the outputs are 32-bit or 64-bit numbers. The Y-axis is logarithmic.

the bit width of the encoded values is less than the original value, we only consider 8-bit and 16-bit inputs. Note that if the size of the dictionary becomes greater than 64k unique elements (16-bit 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, shown by the red dashed line in the figure. Additionally, the total bandwidth shows the DRAM to FPGA/CPU bandwidth, including reading the data page and writing the decoded data to the DRAM.

NASCENT dictionary decoder kernel in all the cases, except for the 8-bit input and 64-bit output case, achieves 3 GB/sec input bandwidth, which saturates the SSD-to-FPGA bandwidth. When the data page is 8-bit encoded data, and the values are 64-bit 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 saturates the input bandwidth due to the DRAM-to-FPGA bandwidth limitation. In this case, the kernel achieves 2.3 GB/sec 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. The number of dictionary decodings per second, when running on CPU, is independent of the input bit width (8-bit and 16-bit inputs) and consequently of the dictionary size. Therefore, the input bandwidth for 16-bit inputs is double that for the 8-bit inputs. Nonetheless, for smaller dictionary tables, NASCENT instantiates more copies of the dictionary to parallelize the decoding further and saturating the SSD-to-FPGA bandwidth.

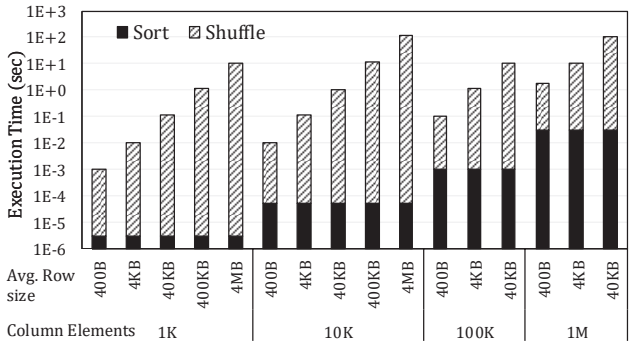

Figure 12: Execution time for sorting tables with different number of rows and columns. The Y-axis is logarithmic.

Figure 11 shows the breakdown of the execution time of sorting a 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, NASCENT dictionary decoder kernel decodes the data to the 32-bit and 64-bit numbers, and then it sorts the decoded column. Note that NASCENT sort kernel can sort 64-bit long numbers with minimal changes in the compare and swap (CS) modules. Due to FPGA resource limitation, both 32-bit and 64-bit NASCENT sort kernels utilize the same amount of BRAMs; therefore, 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, NASCENT 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 NASCENT sort kernels deliver the same performance. Nevertheless, for larger input sizes, the 64-bit NASCENT sort kernel provides slightly lower performance than the 32-bit sort kernel due to higher DRAM accesses. As illustrated in Figure 11 the execution time of the NASCENT dictionary decoder kernel linearly increases with the size of the input array since the dictionary decoder kernel performance is data independent.

Figure 12 shows the breakdown of the execution time of NASCENT 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 the content of the columns is *not* limited to integer types and can be any types 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 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 NASCENT increases almost linearly with the size of the table.

4.3 System Evaluation

In order to evaluate the scalability of NASCENT, in Figure 13 we show the execution time of the CPU, typical FPGA-equipped systems (see Figure 2) and NASCENT when the number of SSD instances increases from 1 to 12 (12 SSDs is the limitation incurred by the number of slot counts of the host machine). Each SSD contains a table with 1024 rows and an average row size of 4KB. Originally, the key columns consist of 32-bit integer numbers, but the columns in the storage system are stored as 16-bit dictionary encoded elements. While 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 12, the execution time of NASCENT 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 all the SSDs contain the same table without loss of generality.

As Figure 13 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 contains more kernels[1]. Nevertheless, as the number of storage devices increases, the execution time of NASCENT 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 13 we also show the energy efficiency of NASCENT versus the FPGA-equipped system (FPGA baseline). As the number of storage devices increases, both the performance and energy efficiency of NASCENT also improves. With 12 SmartSSDs, NASCENT is 7.6× (147.2×) faster and 5.6× (131.4×) more energy efficient than the FPGA (CPU) baseline.

Eventually, Figure 14 shows the speed-up and energy efficiency of NASCENT compared to the FPGA-equipped system when it sorts a copy of the largest table (order-line) of the TPCC benchmark in each SSD [36] (as explained earlier, the performance of sorting the table with 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 have evaluated the performance of NASCENT when sorting the on the TPCC benchmark for five different scale factors $\{1, 2, 5, 10, 20\}$ (which scales the number of rows). Compared to the FPGA baseline, NASCENT shows an average 9.2× speed-up and 6.8× energy reduction when using 12 SmartSSDs. NASCENT 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).

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

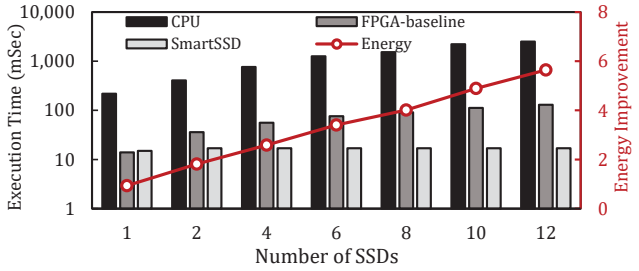

Figure 13: Execution time of NASCENT as compared to CPU and FPGA baseline for sorting 1024×1024 tables, each stored in an SSD. The Y-axis is in logarithmic scale.

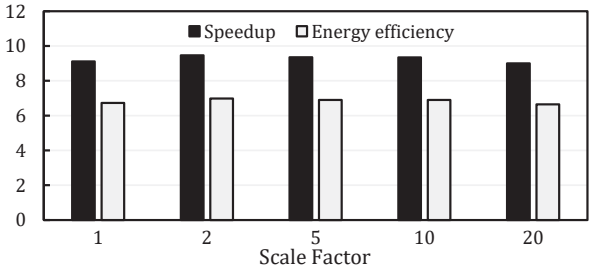

Figure 14: Execution time of NASCENT compared to the FPGA-equipped baseline storage when sorting multiple copies of the largest table of TPCC benchmark on 12 storage devices. The *scale factor* denotes the scaling up the number of benchmark rows.

5 CONCLUSION

In this paper, we present NASCENT, a near-storage accelerator for database sort on SmartSSD based on the bitonic sort. NASCENT tackles the data transfer limitations in current interface connections between storage devices and computation platforms. NASCENT comprise FPGA-based accelerators with specific kernels to accelerate dictionary decoder, sort, and the subsequent shuffling operations to sort a database table. NASCENT increases the scalability of computer systems by enabling simultaneous operations on different storage devices. With 12 SmartSSDs, NASCENT is 7.6× faster and 5.6× more energy efficient than the same accelerator on conventional architectures comprising a stand-alone FPGA and storage devices. NASCENT also shows 147.2× speedup and 131.4× energy reduction as compared to sorting the database table on the host CPU.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers whose valuable comments and suggestions helped improve and clarify this manuscript. 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 #2028040, #1730158, #1911095, #2003279 and #1826967.

REFERENCES

[1] R. Ramakrishnan, J. Gehrke, and J. Gehrke, *Database management systems*, vol. 3. McGraw-Hill New York, 2003.

[2] D. Kumar and M. N. Mohanty, "A survey: classification of big data," in *Cognitive Informatics and Soft Computing*, pp. 299–306, Springer, 2019.

[3] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jing, and S. J. Swanson, *Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources.* Department of Computer Science and Engineering, University of California . . . , 2015.

[4] Z. Ruan and T. H. J. Cong, "Analyzing and modeling in-storage computing workloads on eisc—an fpga-based system-level emulation platform," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.

[5] G. Koo, K. K. Matam, I. Te, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 219–231, IEEE, 2017.

[6] Z. Ruan, T. He, and J. Cong, "{INSIDER}: Designing in-storage computing system for emerging high-performance drive," in *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pp. 379–394, 2019.

[7] "Smartssd." https://samsungsemiconductor-us.com/smartssd/. Accessed: 2020-05-27.

[8] "Scaleflux." http://www.scaleflux.com/. Accessed: 2020-05-27.

[9] "White paper: Smarter data storage, a guide to computational storage on arm," tech. rep., Arm, 09 2019.

[10] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson, "Ssd in-storage computing for list intersection," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pp. 1–7, 2016.

[11] S. Salamat, M. Imani, S. Gupta, and T. Rosing, "Rnsnet: In-memory neural network acceleration using residue number system," in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–12, IEEE, 2018.

[12] S. H. Hashemi, J. H. Lee, and Y. S. KI, "Optimal dynamic shard creation in storage for graph workloads," June 18 2020. US Patent App. 16/274,232.

[13] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[14] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "Smartssd: Fpga accelerated near-storage data analytics on ssd," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, 2020.

[15] B. Khaleghi, S. Salamat, M. Imani, and T. Rosing, "Fpga energy efficiency by leveraging thermal margin," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 376–384, IEEE, 2019.

[16] B. Falsafi, B. Dally, D. Singh, D. Chiou, J. Y. Joshua, and R. Sendag, "Fpgas versus gpus in data centers," *IEEE Micro*, vol. 37, no. 1, pp. 60–72, 2017.

[17] S. Salamat, B. Khaleghi, M. Imani, and T. Rosing, "Workload-aware opportunistic energy efficiency in multi-fpga platforms," *arXiv preprint arXiv:1908.06519*, 2019.

[18] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 133–139, 2020.

[19] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 14, pp. 1647–1658, 2016.

[20] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[21] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1221–1230, 2013.

[22] P. Francisco, "Ibm puredata system for analytics architecture," *IBM Redbooks*, pp. 1–16, 2014.

[23] V. Lagrange Moutinho dos Reis, H. Li, and A. Shayesteh, "Modeling analytics for computational storage," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 88–99, 2020.

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

[25] C. Liu, M. Umbenhower, H. Jiang, P. Subramaniam, J. Ma, and A. J. Elmore, "Mostly order preserving dictionaries," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1214–1225, IEEE, 2019.

[26] I. Boicu, "Adaptive on-the-fly compressed execution in spark," 2019.

[27] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek, "Extrav: Boosting graph processing near storage with a coherent accelerator," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1706–1717, 2017.

[28] S. Pei, J. Yang, and Q. Yang, "Registor: A platform for unstructured data processing inside ssd storage," *ACM Transactions on Storage (TOS)*, vol. 15, no. 1, pp. 1–24, 2019.

[29] D. Koch and J. Torresen, "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*, pp. 45–54, 2011.

[30] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, 2012.

[31] H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "Fpga-accelerated samplesort for large data sets," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 222–232, 2020.

[32] A. Hematian, S. Chuprat, A. A. Manaf, and N. Parsazadeh, "Zero-delay fpga-based odd-even sorting network," in *2013 IEEE Symposium on Computers & Informatics (ISCI)*, pp. 128–131, IEEE, 2013.

[33] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 240–249, 2015.

[34] A. R. Lipu, R. Amin, M. N. I. Mondal, and M. Al Mamun, "Exploiting parallelism for faster implementation of bubble sort algorithm using fpga," in *2016 2nd International Conference on Electrical, Computer & Telecommunication Engineering (ICECTE)*, pp. 1–4, IEEE, 2016.

[35] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 307–314, 1968.

[36] "Tpcc benchmark." http://www.tpc.org/tpcc/. Accessed: 2020-05-27.