2021 IEEE 39th International Conference on Computer Design (ICCD)

2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00080

PSACS: Highly-Parallel Shuffle Accelerator on Computational Storage

Chen Zou
Computer Science
University of Chicago
Chicago, IL, USA
chenzou@uchicago.edu

Hui Zhang
Memory Soluation Lab
Samsung Semiconductor, Inc.
San Jose, CA, USA
w.hzhang86@samsung.com

Andrew A. Chien
Computer Science
University of Chicago
Chicago, IL, USA
achien@cs.uchicago.edu

Yang Seok Ki
Memory Soluation Lab
Samsung Semiconductor, Inc.
San Jose, CA, USA
yangseok.ki@samsung.com

***Abstract*—Shuffle is an indispensable process in distributed online analytical processing systems to enable task-level parallelism exploitation via multiple nodes. As a data-intensive data reorganization process, shuffle implemented on general-purpose CPUs not only incurs data traffic back and forth between the computing and storage resources, but also pollutes the cache hierarchy with almost zero data reuse. As a result, shuffle can easily become the bottleneck of distributed analysis pipelines.**

Our PSACS approach attacks these bottlenecks with the rising computational storage paradigm. Shuffle is offloaded to the storage-side PSACS accelerator to avoid polluting computing node memory hierarchy and enjoy the latency, bandwidth and energy benefits of near-data computing. Further, the microarchitecture of PSACS exploits data-, subtask-, and task-level parallelism for high performance and a customized scratchpad for fast on-chip random access.

PSACS achieves 4.6x−5.7x shuffle throughput at kernel-level and up to 1.3x overall shuffle throughput with only a twentieth of CPU utilization comparing to software baselines. These mount up to 23% end-to-end OLAP query speedup on average.

***Index Terms*—shuffle, accelerator, computational storage, SmartSSD, OLAP**

I. Introduction

As the extractor of task-level parallelism for multiple nodes to exploit, shuffle is an essential process in distributed data analytics systems [1]–[5]. Shuffle at each node would first trigger the materialization of the intermediate results, which are then partitioned and grouped as local shuffle output and stored into local storage. These shuffle output would be fetched by different nodes across the cluster for later-stage tasks.

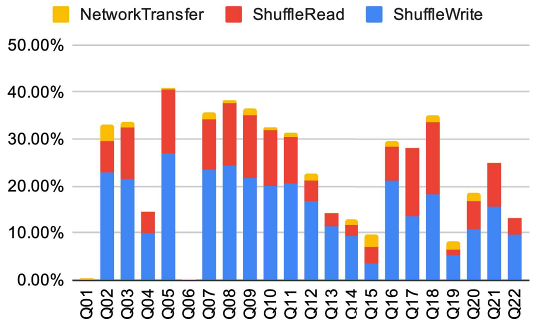

Fig. 1: Shuffle bottleneck in OLAP systems

However, multiple batches of partitioned data are spilled out into storage drives during shuffle. These spilled data need to be fetched back into the main memory to be grouped together to enable larger access granularity for efficient data transfer over the network. This process incurs traffic back and forth between compute and storage as well as ill use of the memory hierarchy, resulting in cache thrashing and page swapping.

These issues make shuffle a performance bottleneck as shown in Figure 1. The measurement was taken on four SSD-equipped 44-SkylakeSP-Core servers connected through 1GE network with jvm-profiler [6]. The workloads are all of the 22 TPC-H [7] queries with scaling-factor 1000 (i.e. raw table sizes totaling roughly 1000 GB) implemented with Spark v3.0.1 [3]. Shuffle tasks can take up to 40 percent CPU time of the entire query execution.

Different from existing work on software optimizations [8], [9], network fabric improvement [10], [11] or function-specific acceleration of partitioning [12], [13], sorting [14] or aggregation [15], we look into PSACS, a highly-Parallel hardware Shuffle Accelerator employing the Computational Storage paradigm (shown in Figure 2) targeting the distributed OLAP workload. PSACS excels in shuffle performance by exploiting task-, subtask- and data-level parallelism. It also employs tiled shuffling and a customized scratchpad for efficient memory accesses at different levels of the memory hierarchy, addressing the aforementioned issues.

In summary, our contributions are three-folded:

- Designed the first shuffle accelerator on computational storage. Offloading shuffle to computational storage constrains shuffle-related accesses inside storage, and thus largely reduces CPU's cache thrashing.
- Designed PSACS microarchitecture exploiting task-,

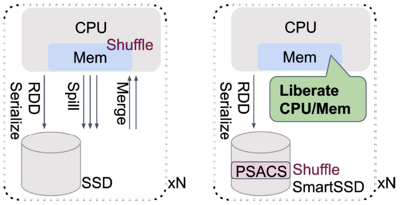

Fig. 2: System architecture w/o and w computational storage

978-1-6654-3219-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICCD53106.2021.00080

subtask- and data- level parallelism and customized scratchpads for high performance random accesses.
- Showcased PSACS performance benefits, host CPU utilization reductions as well as synergy supports for output redistribution of PSACS.

The rest of the paper is organized as follows. In Section II, we discuss backgrounds for shuffle and SmartSSD. In Section III, we bring out algorithms and approaches behind PSACS. In Section IV, detailed PSACS architecture is discussed. We provide evaluation methodology and evaluation results of PSACS's implementation on SmartSSD in Section V. Related work is covered in Section VI. We discuss the summary and future work in Section VII.

II. Background

A. Shuffle in distributed systems

Shuffle is an indispensable operation in distributed computing systems [1]–[5]. It redistributes data across the computing nodes according to meaningful repartitioning to enable the exploitation of partition-level parallelism in distributed systems. After shuffle, the computation to carry out on each partition would be independent, allowing concurrent computations on multiple cores across multiple nodes.

Let us use the famous word counting problem that is first coined by MapReduce [5] as an example (see also Figure 3). First, the map tasks in each node would independently and thus concurrently map the words assigned to the node to tuples: (*word*, 1). In order to utilize each node (and each core) to perform independent aggregation, tuples with the same *word* must end up in the same node. This is where shuffle comes into play. Tuples have to be shuffled on *word* (shuffle key) to form multiple partitions but ensuring the tuples with the same shuffle key go into the same partition. Then, the reduction on counts for each partition could be executed concurrently. This same-key-same-partition property is what we call 'meaningful' partitioning. The most used scheme for partitioning is to use a single hash function on each shuffle key to determine its destination partition. If the same hash function is used by all the nodes or cores performing local partitioning, the same-key-same-partition property is guaranteed.

Shuffle generally has three steps as shown in Figure 3:

- Partition: Apply partitioning scheme to the records to determine their destination partitions.
- Group: Group records by their destination partitions for efficient access during Distribution.
- Distribute: Distribute records based on their destinations.

Only the Distribute step involves communications across nodes. Each node (each core) carries out the Partition step and Group step on its own without synchronizations or communications with other nodes (cores). These properties shape the notion of 'stage' and 'task' in distributed OLAP systems, as shown in Figure 4. A shuffle process spans two stages, with Partition and Group step (together called as ShuffleWrite) at end of the prior stage, and the Distribute step is done at the beginning of the following stage through pulling rather than pushing (thus called ShuffleRead). The compute results of a prior stage are persistently stored into storage during ShuffleWrite for reliability and failure-recovery. Any failure of a stage would only require recomputation from the ShuffleWrite results of the previous stage stored in storage, rather than starting from the beginning.

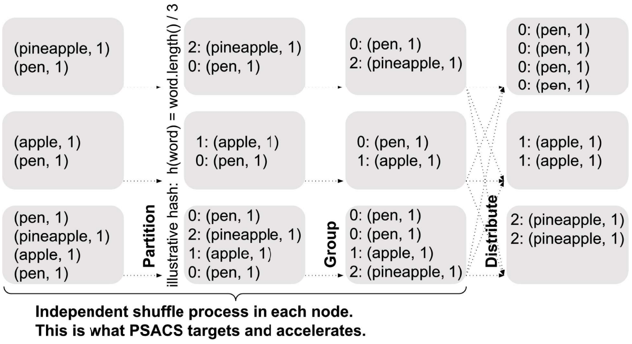

Fig. 3: Shuffle steps. WordCount on three nodes as an example

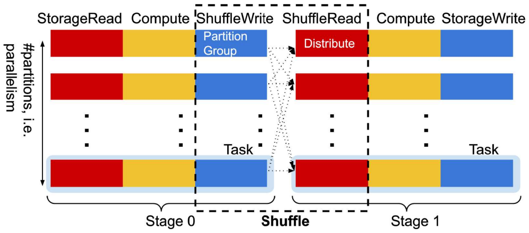

Fig. 4: Shuffle separates two stages of local compute

B. SmartSSD: a computational storage drive

Recent rapid increases in storage device capacity and bandwidth have shifted scaling and cost bottlenecks in the modern HPC or cloud data center architecture to the CPU and interconnect. Computational storage is a thread of efforts trying to address this changing balance. It adds computation and acceleration capabilities inside the storage, moves computation closer to the storage to reduce data movement, and enables autoscaling of the computation with storage capacity and bandwidth.

SmartSSD [16] is one of the first industry products in the direction of computational storage. As shown in Figure 5,

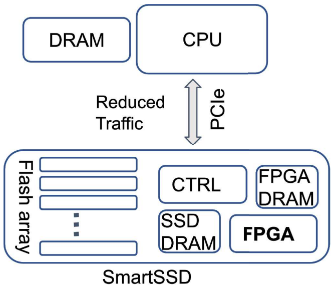

Fig. 5: SmartSSD: a computational storage drive

SmartSSD integrates a Xilinx FPGA chip onto the solid-state drive, next to its flash array which possesses excessive bandwidth as opposed to what an SSD exposes. Although FPGA is still a peer to the PCIe endpoint of the SSD, FPGA and SSD PCIe endpoint share the same PCIe switch. This is an essential step toward opening up the storage and allowing near-data computation. The FPGA can get access to the data in storage without passing them through the host memory, CPU or PCIe root complex, which opens up to early-filtering and data reduction opportunities to address the aforementioned computation and interconnect bottlenecks.

III. Shuffle acceleration Approach

A. System architecture

We envision a system architecture following Figure 2. Every node in the system features a PSACS-enabled storage device. Map tasks are applied to the records (e.g. rows in Dataframes in the case of Spark) by the CPU of each node before streaming into the computational storage DRAM. There, PSACS would perform shuffle Partition and Group for local map outputs, which also constrains shuffle traffic inside storage. Results are then persistently stored into the storage media (e.g. flashes) for durability and failure recovery (see end of Section II-A). At the same time, CPU(s) are free to compute for other map tasks, given the ample task-level parallelism in OLAP workload. PSACS interfaces to Spark, specifically, with a new shuffle manager wrapping the PSACS shuffle kernel on SmartSSD FPGA. Data are written from SmartSSD FPGA to flashes via PCIe peer-to-peer transfers.

Recall that shuffle Partition and Group do not involve communications across nodes (see the end of Section II-A), shuffle acceleration carried out in one PSACS-enabled storage device is independent of other nodes and PSACS devices. Further, results after Partition and Group to be fetched by each node for next-stage computation are contiguous in storage. It is easy to form a block-level prefetch/compute pipeline to hide network latency of next block under the computation of the current block. Thus, the latency added by Shuffle Distribute is negligible as shown in Figure 1. If each node enjoys shuffle acceleration by PSACS, collectively, the shuffle process across the whole distributed OLAP system is accelerated.

In the following section, we will describe PSACS's approach to accelerate shuffle's Partition and Group steps.

B. Partitioning acceleration approach

Partitioning assigns a new partition label for each record to be shuffled. And this label determines the shuffle destination. We take a hash approach for Partitioning considering its generality of not requiring shuffle keys to have a total order relation. Different from hash approaches used in Cryptography, collision resistance is not a priority. Rather, each partition should get a near-even allocation of shuffling records. The hash function we considered is a variant of folding hash, as shown in Figure 6. We additionally zigzag the bits in a shuffle key to improve robustness and evenness. Our PSACS design is not dependent on this chosen hash. Different hash modules or even other partitioning schemes can be easily swapped in.

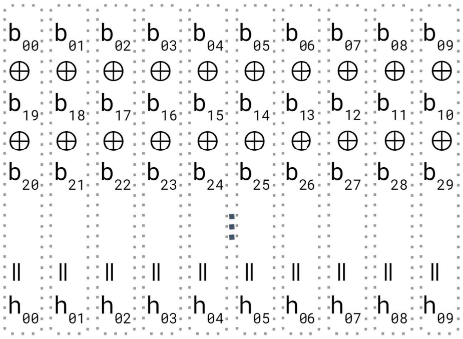
b_{00} b_{01} b_{02} b_{03} b_{04} b_{05} b_{06} b_{07} b_{08} b_{09}
⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕
b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10}
⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕
b_{20} b_{21} b_{22} b_{23} b_{24} b_{25} b_{26} b_{27} b_{28} b_{29}
⋮
|| || || || || || || || || ||
h_{00} h_{01} h_{02} h_{03} h_{04} h_{05} h_{06} h_{07} h_{08} h_{09}

Fig. 6: Zigzag-fold hash. b_i is the i-th bit of shuffle key. h_j is the j-th bit of output. Only 10-bit output situation is drawn

C. Grouping acceleration approach

Shuffle Grouping is most challenging because it performs data reorganization among a huge working set with scarce data reuse opportunities. There are two most used approaches: bucketing and sorting.

Bucketing-based grouping allocates a bucket for each destination partition. After assigned a partition label by the partitioning scheme, each record is inserted into the corresponding bucket. However, without efficient memory management in a computational storage environment, we will end up allocating memory capacity for each bucket based on an extreme scenario to avoid overflow, resulting in huge memory wastes and aggravated spilling to storage drives. Even with a memory allocator like the one in Caribou [17], bucketing based-solution does not scale well as it presents a linear scaling of bucket size requirement for on-chip SRAM over the number of destination partitions (i.e. #cores in a distributed OLAP system). It would be infeasible for a scaled-out distributed OLAP system. This is the root cause of cache thrashing and low performance in existing software shuffle implementations when opting for bucket-based shuffle grouping (BypassMergeSortShuffleWriter in the case of Spark [3]). And this is also the drawbacks of existing single-node-oriented hardware partitioning accelerators [12], [13] for only support either a small number of partitions (e.g. up to 256 [13]) or fixed short 8-Byte rows [12].

Sort-based grouping arranges the records by sorting them based on assigned partition labels. The resources required do not grow with the number of partitions needed to support, making the approach suitable for a scaled-out distributed OLAP system. Further optimization is: instead of sorting the records (each could be large) directly, sort the record pointers (each is small, e.g. 8B) with the partition label and perform gather for each sorted pointer to replace the pointer with the actual record. This reduces the data copy operations from O(logN) to O(1) per record, improving grouping performance for the OLAP case which features not-so-short records. However, the gathering process features random accesses over a large working-set and almost none temporal locality. This would trigger cache thrashing if not treated carefully.

We opt for sort-based grouping for its scalability. We address the issue of random access during the gathering process through the use of a customized scratchpad and exploit data-level parallelism for long record gathering, both of which would be described in detail in Section IV.

D. Distribute considerations

In terms of the Distribute step for data that is already Partitioned and Grouped, our core consideration is to enable better compression before transmitting over the network to enjoy higher effective bandwidth (multiplied by compression ratio) and reduce network traffic. This brings both performance and energy benefits, especially for the scenario of disaggregated compute and storage resource pools for ease of management and separate scaling where network connecting the two pools is potentially busy.

For this purpose, our PSACS acceleration interface is carefully designed to accept and emit column-major table slices. Although PSACS does not contain a compression module, the columnar interface design opens up to the opportunities of better compression and higher benefits, as the similarities among the data in the same column are higher [18].

IV. PSACS Architecture

A. Microarchitecture of PSACS

The PSACS microarchitecture (depicted in Figure 7) features the following modules:

- FSM: Controls the shuffle process.
- Reader: Manages a scratchpad as on-chip random access buffers for the table data. Accept the prefetch of table data from DRAM into this buffer. Serves the data to Gather upon random access by the table RowID.
- Partitioner: Accept shuffle keys streamed from the reader. Map keys via hash (Section III-B) to partition ID(PID).
- Sorter: Sort tuple (PID, RowID) from Partitioner by PID and stream the sorted tuples into Gather.
- Gather: Gather rows from Reader by RowIDs in sorted tuple streams from Sorter through random accessing a scratchpad in Reader.
- Writer: Stream the rows from Gather to DRAM.
- Indexer: Generate an index identifying the portions of each partition across shuffle batches.
- Merger: Merge according to indexes, grouping the records going to the same partition across batches. Merger also split each row into different fields for columnar store.

Please recall that we employ hash-based partitioning and sort-based grouping with optimization of first pointer sorting and then gathering as discussed in Section III-C. Partitioner implements our hash-based partitioning approach, while Sorter and Gather collectively implement our optimized sort-based grouping approach. The Reader and Writer are memory hierarchy modules that manage a customized random-access scratchpad and a write-buffer respectively, and handle the implementation of the computational storage DRAM access protocol. The Indexer and Merger relate to the tiled shuffling which we will discuss in Section IV-B.

We will walk through each module with the word counting example we first used in Section II-A. Please see green texts in Figure 7. First, records in (*word*, 1) are prefetched into the scratchpad by the Reader to prepare for later random access by the Gather. Second, during this process, the shuffle keys (*word*) are further streamed into Partitioner and produced tuples of (PID, RowID). PID is a partition ID. RowID is a record pointer (An index to a table row in the OLAP case). Third, tuples of (PID, RowID) are streamed into Sorter and sorted by PID. This is the indirect sorting we discussed in Section III-C. Fourth, Gather would replace RowID in the sorted tuple stream with the actual record contents through random accessing Reader's scratchpad with RowID as the address. Fifth, the Writer would accept (PID, Record(*word*, 1)) tuples and translates to AXI packets to write these outputs to computational storage DRAM.

Because gathering is performed in the order of associated sorted PIDs, the output records, which are streamed from Gather through Writer into computational storage DRAM, are grouped (or to be more specific, sorted) by PIDs.

B. Tiled shuffling tailoring for memory hierarchy

As discussed in Section I, software shuffle on CPUs suffer from the ill use of the memory hierarchy. In contrast, PSACS efficiently utilizes each level of the memory hierarchy through optimizing the overall shuffle via tiling. Tiling also enables PSACS to support shuffle tasks in different sizes independent from computational storage DRAM size through divide-and-conquer.

Shuffle is first carried out in batches, where the size of each batch is limited by the on-chip memory resources of PSACS. The on-chip memory is used as a scratchpad (in Reader) to provide fast and efficient random access as needed by the Gather. Once enough batches of records are shuffled, the Merger is triggered to further group shuffle results going to the same partition across different batches through reorganizing the contents in the computational storage DRAM, as shown in Figure 8. For each partition, the Merger (more specifically AddrGen) checks the indexes generated by Indexer for each batch, fetches the records assigned to that partition from DRAM and concatenates them together. The merged shuffle results are then written into storage media for persistency and failure recovery. In our implementation on SmartSSD, this is done through PCIe peer-to-peer transfer between FPGA DRAM and SSD DRAM.

Merging carried out by the Merger would make shuffle Distribution (i.e. ShuffleRead in next-stage tasks) mostly sequential reads over distributed storage, largely increases the efficiency and reduces the latency, as shown in Figure 1.

Depending on the needs of the workloads, there may be another level of grouping that merges shuffle results externally on disk by rearranging the disk pages. It is even possible to tap into the flash translation layer [19] and reorganize the mapping to achieve virtual merging. Because of the embracing

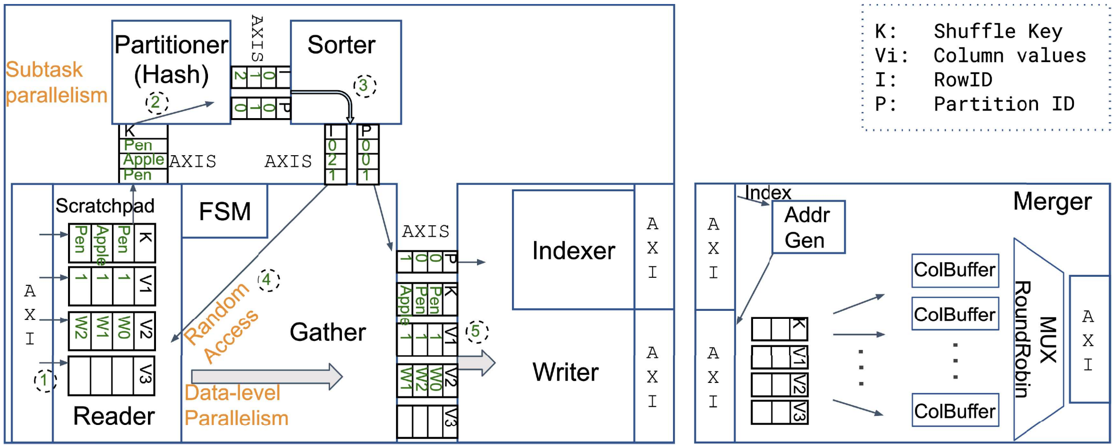

Fig. 7: Microarchitecture diagram of PSACS

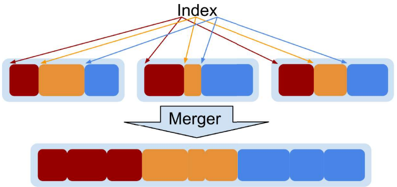

Fig. 8: Merging across multiple shuffled batches

of the computational storage paradigm, our PSACS system architecture is flexible to support this as a future extension.

C. Scratchpad memory for efficient random access

As discussed in Section III-C, in our improved sort-based Group, we first sort the record pointers (short, 8B) by the shuffle key and then gather the records (long, variable) to arrange the records by the order from sorting. Since each row of the table is only gathered once, there is no temporal locality during this process. Further, as the gathering process features random accesses to a large working set, the spatial locality is also scarce. Thus, the employment of cache for the shuffle Group step would only incur latency and energy overhead of fetching more data (cache line size) onto the chip and wasting precious on-chip memory resources. As a result, we opt for a customized scratchpad memory specifically serving the random access from the Gather. The scratchpad memory is a flat memory without tag arrays (as there is no locality). The Reader manages the data in the scratchpad memory by prefetching the table data. The Reader also performs clever layout and alignment for different columns with different widths to support efficient streaming access needed by the Partitioner for shuffle keys, and high-speed wide random access needed by the Gather.

D. Parallelism and their exploitations

There is plenty of parallelism in the shuffle process, and our PSACS microarchitecture enables us to exploit them. First, the shuffle grouping subtask features the data-level parallelism as the whole record (or in the OLAP case, the whole row) needs to be reordered by the sorted pointers (RowIDs in OLAP). The scratchpad memory managed by the reader features a wide data path for random access by the Gather, exploiting data-level parallelism. Second, there is subtask-level parallelism in shuffle workloads. The data prefetching management in the reader could be overlapped with the partitioning in Partitioner and the pointer sorting process in Sorter. To enable the exploitation, we prioritize the prefetching of the shuffle keys and streaming in the shuffle keys to the partitioner at the same time. With keys fetched, the prefetching of other values could be overlapped with partitioning of keys and the pointer sorting. Finally, there is task-level parallelism to exploit. There are multiple partitions of local map results to be shuffled, because map tasks are carried out in parallel by multiple cores in a single node. The multi-processing paradigm could be employed by duplicating PSACS to shuffle different partitions in an embarrassingly parallel manner. Although in our implementation, we are limited to one PSACS kernel on SmartSSD due to resource limits (mostly limited by on-chip memory size).

E. Columnar output for better compression and redistribution

As discussed in Section II-A, we aim to provide synergy support for the shuffle Distribution by enabling better compressibility through a column-major output format.

The Merger module contains the transpose functionality to achieve this. Each column is assigned a unique buffer (ColBuffer) before writing to computational storage DRAM. Transpose is delayed until merging for maximum exploitation of data-level parallelism in Partition and Group steps.

V. Evaluation

A. Methodology

As discussed in Section III-A, the shuffle Partition and Group steps are independently accelerated without cross-node communication, and the added latency from shuffle Distribute are really low. Thus, we evaluate PSACS in a single-node

TABLE I: Resource utilization of PSACS on SmartSSD

	LUT	FF	BRAM	URAM	DSP
Avail	522720	1045440	984	128	1968
Used	61917	80885	433	64	0
Util	11.85%	7.74%	44.00%	50.00%	0.00%

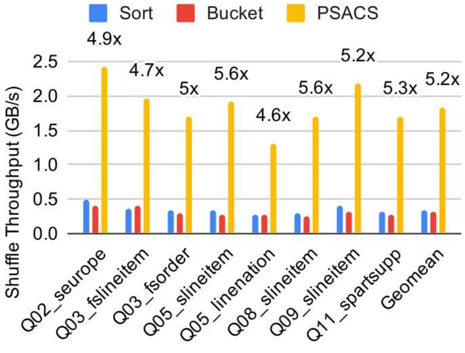

Fig. 9: Shuffle kernel performance

setting to reduce uncontrolled variables. A server with a 44-core SkylakeSP CPU, 128GB DRAM and a PCIe-attached SmartSSD as discussed in Section II-B is used. We opted for SmartSSD for the simplicity of conducting evaluations. We believe that results are transferrable to other FPGA-based computational storage platforms.

The workload is the most data-intensive (see Figure 1) shuffle kernels (named with 'Qid_tablename') extracted from Apache Spark v3.0.1 [3] implementation of TPC-H queries [7]. The number of partitions that shuffle would output is set to be 1024, which corresponds to a cluster with dozens of nodes.

We compare PSACS against two hand-optimized software shuffle kernels, each implementing one grouping algorithm as described in Section III-C, and named as 'Sort' and 'Bucket'. The two kernels very much resemble the BypassMergeSortShuffle and SortShuffle from Spark.

B. PSACS implementation on SmartSSD

We implemented PSACS on a SmartSSD with a KU15 FPGA via System Verilog and Xilinx Vitis. We adapt a merge sorter [20] for the Sorter module to achieve $O(Nlog_2N)$ time complexity for sorting. The scratchpad for random access is implemented with UltraRAMs to hold a large working set.

PSACS is synthesized to run at 221 MHz. The overall resource utilization is summarized in Table I.

C. Shuffle kernel performance of PSACS

First, we compare kernel-level performance. All three approaches are implemented with a single thread. And the performance here does not consider the latency of data preparation or storing results into the storage media.

Figure 9 shows the kernel-level shuffle performance of PSACS and its software counterparts. The performance here is the geometric mean of the throughput over all the tasks from the corresponding shuffle workload. Our PSACS approach features 4.6x−5.7x higher kernel-level shuffle performance than its software counterparts. The performance comes from efficient parallelism exploitation and the use of customized scratchpad that eliminates thrashing.

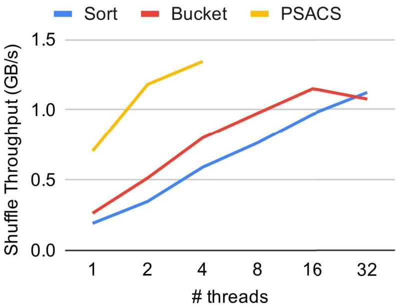

Fig. 10: Multithreading and system-level pipeline scaling for shuffle throughput on Q05 slineitem

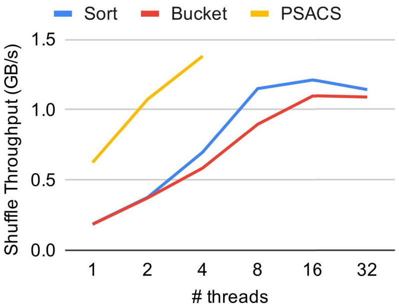

Fig. 11: Multithreading and system-level pipeline scaling for shuffle throughput on Q11 spartsupp

D. Overall shuffle performance of PSACS

In this section, we consider the shuffle workload from the perspective of an overall system. Both data preparation before shuffle and results writing to storage media after shuffle are included. Further, software baselines can employ up to 32-thread multithreading to exploit task-level parallelism. Falling short of resources on SmartSSD FPGA for task-level parallelism, at any time, there is only one shuffle kernel processing shuffle tasks in PSACS. But the PSACS approach would exploit system-level pipelining to absorb data preparation and results storing latency into the latency of PSACS shuffle acceleration. This is only up to 4-threads because of the diminishing returns. Thus, at most four cores are used from the host CPU by PSACS. much less than the 32 cores in the two software baselines.

We first investigate how multithread scaling would affect the performance. Figure 10 and 11 show detailed scaling for shuffle workloads from two TPC-H queries. The performance of the software-based methods saturates around 16 threads because of the increasing cache thrashing. On the other hand, system-level pipelining significantly improves overall shuffle throughput for PSACS through the latency hiding.

Figure 12 shows the overall shuffle performance of PSACS, with comparisons to the two software baselines on the

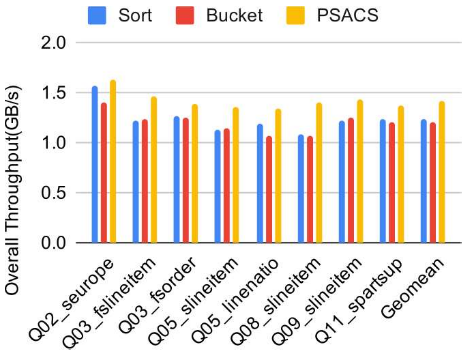

Fig. 12: Overall shuffle performance on shuffle-intensive TPC-H queries

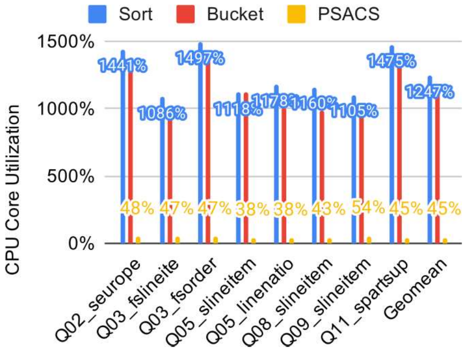

Fig. 13: CPU core utilizations for shuffle on shuffle-intensive TPC-H queries

TPC-H shuffle-intensive queries. And Figure 13 depicts the corresponding CPU core utilization. [1]. Please notice that our PSACS approach features only one accelerator kernel implemented on a single SmartSSD. But our PSACS approach outperforms the up to 32-way multithread software shuffle approaches by 10% - 31% as in Figure 12. PSACS also features more than 20 times lower CPU core utilization as in Figure 13, which liberates the host CPU and memory for other map tasks that is not yet computed.

Further, this low utilization could lead to significant scaling-up opportunities, as the CPU has enough headroom to drive dozens of computational storage drives. And because of our computational storage approach, the shuffle acceleration capability of PSACS scales with the storage, thus this scaling wouldn't hit a storage bottleneck as is the case of the software-based shuffle approaches.

E. End-to-end query performance

We further extrapolated end-to-end TPC-H query performance, through instrumenting Spark event traces considering both shuffle throughput improvement by PSACS acceleration and early start of new tasks when host CPU and memory is freed from shuffle through offloading to PSACS inside computational storage.

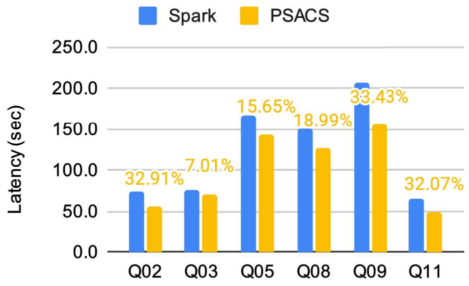

Fig. 14: End-to-End TPC-H query latency

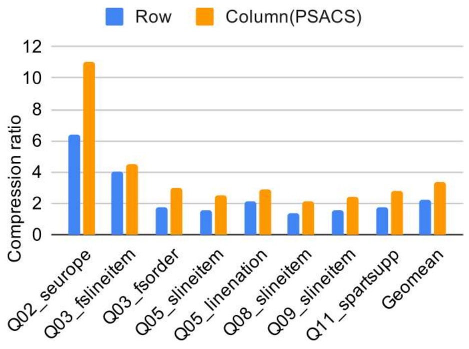

Fig. 15: Compression ratio: row-major VS column-major

Unmodified Spark 3.0.1 is used as the baseline to compare with PSACS. Figure 14 shows that PSACS delivers significant end-to-end query speedup, averaging at 23%. The speedup comes from PSACS's shuffle acceleration exploiting multiple levels of parallelism and more efficient use of the host CPU and caches.

F. Column-major for better compression in redistribution

In this section, we evaluate how much compression ratio improvement the column-major format could bring, for a glimpse on further multiplicative-bandwidth benefits if future implementation of PSACS includes a compression module. We measure the compression ratio achieved when compressing using LZ4 on shuffle outputs in UnsafeRow (row-major) format [21] and the compression ratio of the same shuffle outputs but in parquet [22] which also uses LZ4 compression but compresses in column-major format.

Figure 15 shows the overall compression ratio we can achieve for row-major and column-major shuffle output format with LZ4 compression. The column-major approach of PSACS can achieve on average 1.5x better compression than row-major approach, demonstrating effective synergy support for the Distribute step of a shuffle process. In other words, the column-major format could on average enable 1.5x higher efficient network bandwidth for shuffle Distribute.

VI. Related Work

Single-node partition acceleration. There is FPGA acceleration work for partitioning in both academia [12]

[1] 16-thread scenario for two software-based approaches as their performance tops around there during multithreading scaling

and industry [13]. They target single-node partitioning to enable multi-core parallelism for later operators. Bucket-based grouping method (see Section III-C) is used, which requires linear scaling of on-chip FPGA memory (BRAM) over the number of destination partitions. PSACS tries to address the scalability issues of these single-node approaches to advance the shuffle acceleration frontier for the distributed systems.

Specific-function acceleration in OLAP. Cereal [23] employs a customized serialization format and exploits parallelism among serialization tasks through customization and acceleration in memory hierarchy. SortAcc [14] accelerates sort functions in map tasks via a fixed-function accelerator. CASM [15] enables across-chip traffic reduction in multi-core system by enabling local aggregation with collaborative accelerators. All these techniques could be integrated on the host-side (i.e. around CPU) of PSACS system to accelerate map tasks that produce records to be shuffled, while PSACS accelerates shuffle in computational storage.

Network transfer fabric improvement. SparkRDMA [10] and SparkPMoF [11] drive performance improvement of the redistribution step (see Section II-A) through the use of better network fabrics. PSACS focuses on the partitioning and grouping steps. These techniques are orthogonal to PSACS and could be integrated together.

Software optimization for shuffle. Riffle [8] employs a new shuffle merge policy to improve shuffle performance. The policy merges small shuffle outputs or skips merging large shuffle outputs, thus optimizing the storage IO as well as network transfer patterns. On the other hand, Intel proposes an in-memory shuffle [9] software architecture. It constrains shuffle traffic inside a disaggregated non-volatile memory [24] pool. PSACS align with the big ideas of constraining shuffle traffic, but inside computation storage. This also opens up new possibility for software optimizations.

VII. Summary and Future Work

We proposed PSACS, the first shuffle accelerator addressing the shuffle bottlenecks of distributed OLAP systems.

Our PSACS approach employs the rising computational storage paradigm. Shuffle is offloaded to the storage-side PSACS accelerator to avoid polluting computing node memory hierarchy and enjoy the latency, bandwidth and energy benefits of near-data computing. Further, the microarchitecture of PSACS exploits data-, subtask-, and task-level parallelism for high performance and a customized scratchpad for fast and efficient on-chip random access.

These innovations lead to 4.6x−5.7x throughput improvements at the kernel level. Even when comparing to multi-threading software baselines with up to 32 threads, single-kernel PSACS on SmartSSD achieves up to 30% overall shuffle throughput improvement with only a twentieth of the CPU utilization. These translate to on average 23% end-to-end query latency reduction, comparing with Spark.

Our future work includes the full integration of the PSACS into Spark, supporting more partitioning schemes and columnar-encoding and decoding inside storage.

References

[1] "Big query." [Online]. Available: https://cloud.google.com/bigquery

[2] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[4] "Apache hadoop." [Online]. Available: https://hadoop.apache.org/

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] "Uber jvm profiler." [Online]. Available: https://github.com/uber-common/jvm-profiler

[7] "Tpc-h, a decision support benchmark." [Online]. Available: http://www.tpc.org/tpch/

[8] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[9] "Remote shuffle." [Online]. Available: https://github.com/Intel-bigdata/OAP/tree/master/oap-shuffle/remote-shuffle

[10] "Sparkrdma shufflemanager plugin." [Online]. Available: https://github.com/Mellanox/SparkRDMA

[11] "Spark-pmof: Rpmem extension for spark shuffle." [Online]. Available: https://github.com/Intel-bigdata/Spark-PMoF

[12] K. Kara, J. Giceva, and G. Alonso, "Fpga-based data partitioning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 433–445.

[13] "Vitis database library." [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html

[14] S. H. Pugsley, A. Deb, R. Balasubramonian, and F. Li, "Fixed-function hardware sorting accelerators for near data mapreduce execution," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015, pp. 439–442.

[15] A. Addisie and V. Bertacco, "Collaborative accelerators for in-memory mapreduce on scale-up machines," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 747–753.

[16] 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.

[17] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1202–1213, 2017.

[18] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 553–564.

[19] Y. Kang, R. Pitchumani, P. Mishra, Y.-s. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 144–154. [Online]. Available: https://doi.org/10.1145/3319647.3325831

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

[21] "Spark unsaferow." [Online]. Available: https://github.com/apache/spark/blob/master/sql/catalyst/src/main/java/org/apache/spark/sql/catalyst/expressions/UnsafeRow.java

[22] "Apache parquet." [Online]. Available: https://parquet.apache.org/

[23] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A specialized architecture for object serialization with applications to big data analytics," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 322–334.

[24] "Optane memory." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html