FANS: FPGA-Accelerated Near-Storage Sorting

Weikang Qiao*, Jihun Oh†, Licheng Guo*, Mau-Chung Frank Chang*, Jason Cong*
* University of California, Los Angeles, USA † Samsung Electronics, Hwasung, Korea
wkqiao2015@ucla.edu, ozfoot.oh@samsung.com, lcguo@cs.ucla.edu,
mfchang@ee.ucla.edu, cong@cs.ucla.edu

***Abstract*—Large-scale sorting is always an important yet demanding task for data center applications. In addition to powerful processing capability, high-performance sorting system requires efficient utilization of the available bandwidth of various levels in the memory hierarchy. Nowadays, with the explosive data size, the frequent data transfers between the host and the storage device are becoming increasingly a performance bottleneck. Fortunately, the emergence of near-storage computing devices gives us the opportunity to accelerate large-scale sorting by avoiding the back and forth data transfer. Near-storage sorting is promising for extra performance improvement and power reduction. However, it is still an open question of how to achieve the optimal sorting performance on the existing near-storage computing device.**

In this work, we first perform an in-depth analysis of the sorting performance on the newly released Samsung SmartSSD platform. Contrary to the previous belief, our analysis shows that the end-to-end sorting performance is bound by not only the bandwidth of the flash, but also the main memory bandwidth, the configuration of the sorting kernel and the intermediate sorting status. Based on our modeling, we propose FANS, an FPGA accelerated near-storage sorting system which selects the optimized design configuration and achieves the theoretically maximum end-to-end performance when using a single Samsung SmartSSD device. The experiments demonstrate more than 3× performance speedup over the state-of-art FPGA-accelerated flash storage.

I. Introduction

Sorting is one of the fundamental computational challenges for data center applications. For example, many relational database system operations such as `order by`, `group by` and `sort-merge join` rely on high-performance sorting. In the era of data explosion, large-scale sorting is becoming a non-trivial performance bottleneck in data centers. As a result, designing an efficient large-scale sorting solution is crucial for the data center system architects.

Sorting large-scale data relies on *external sorting* [1], where storage devices such as hard disks or flashes are used to store the intermediate data and final results [2]. This is in sharp contrast to the widely studied *internal sorting* problem where the entire data can fit into the system main memory (DRAM). One of the most common external sorting algorithms is *external merge sort* [1]. This algorithm consists of two phases: the sorting phase and the merging phase. In the sorting phase, data are first sorted into intermediate chunks that can fit into the processor's main memory. In the following merging phase, those sorted chunks are merged and written into the external storage through one or multiple merging passes to be the final result. The external merge sort algorithm is both computationally intensive and data intensive. On one hand, the sorting phase relies on a high-performance processor to sort the data. On the other hand, the merging phase moves the data frequently between the processor and the external storage.

The emerging near-storage computing devices bring new opportunities to accelerate external merge sort. Instead of first copying the data to the host side, the co-processor placed right next to the storage could process the data in the drive directly, thus it can perform the sorting tasks more efficiently. Specifically, we find that the newly released Samsung SmartSSD is a suitable candidate for near-storage acceleration of large-scale sorting. Samsung SmartSSD is a programmable computational storage platform that integrates an FPGA into the same package as the flash. With Samsung SmartSSD, we can accelerate the sorting phase using customized FPGA accelerators while avoiding the expensive data transfer between the host and the flash in the merging phase.

Although FPGAs have been widely used to accelerate various sorting algorithms, most of the previous works only target on-chip sorting [3], [4], [5], [6], [7] or DRAM-scale sorting [8], [9], [10], [11], [12], [13]. While achieving competitive performance when sorting small-scale data, these design intuitions cannot be directly employed in the near-storage computing scenarios, where the data are exchanged among multiple levels in the memory hierarchy. Specifically, without considering the low bandwidth of the storage and having an efficient scheme to hide the storage access latency, a high-performance DRAM sorter will be futile.

There are only a few previous studies on FPGA-accelerated external merge sort that also involve the flash [14], [15]. For example, [14] builds an FPGA-accelerated flash storage but only considers the bandwidth of the flash as the system bottleneck. In comparison, we present a more comprehensive analysis and show a variety of influencing factors such as the FPGA size, the DRAM bandwidth and even the size of the intermediate sorted chunks. Another work [15] presents a scalable sorting solution that is able to sort data in a range from megabytes to terabytes. When sorting terabyte data, the solution in [15] is based on an ideal computational storage, which has non-trivial discrepancy compared to the existing near-storage devices, such as Samsung SmartSSD used in this work. Section V-C will discuss the reason why their results cannot be directly applied.

In this work, we propose FANS, an FPGA-accelerated near-storage sorting solution that achieves remarkable end-to-end sorting performance when targeting Samsung SmartSSD. First,

through a comprehensive analysis of the sorting system based on the memory hierarchy of SmartSSD, we identify that in addition to the flash bandwidth, there are multiple key factors that determine the overall performance, including the merge sort kernel configuration, the FPGA DRAM specification and the sorted chunk size after the sorting phase. Based on our analysis, we provide an optimized sorting solution for Samsung SmartSSD. Our solution has a distinct sort phase and merge phase, and each phase is configured separately to maximize the entire sorting performance. Different phases can be activated at runtime by reprogramming the FPGA. The experiments demonstrate that our near-storage sorting system achieves more than 3x performance speedup over the previous state-of-the-art of FPGA-accelerated flash storage [14]. The summary of the contributions of the paper is listed as follows:

- We propose a novel analytical framework to model the overall performance of the FPGA-accelerated external merge sort system and reveal that various factors could be the bottlenecks of the end-to-end system performance.
- We implement a high-performance end-to-end sorting system on Samsung SmartSSD, which contains a distinct sort phase and merge phase. Our system offers an efficient FPGA-flash communication scheme to overlap the FPGA sorting with the flash access and allows for independent optimization of each phase through FPGA reprogramming at runtime.
- Our sorting acceleration system with optimized design configuration achieves more than 3x speedup over the previous state-of-the-art.
- We provide valuable architectural insights based on our analysis and experiments to help vendors further improve their near-storage computing devices.

The rest of the paper is organized as follows: Section II reviews the common elements of FPGA sorting systems and introduces Samsung SmartSSD. Section III explains the system architecture of FANS. Section IV discuss the performance analysis and optimized system configuration for Samsung SmartSSD. In Section V, we present the performance evaluation of FANS. The conclusion is in Section VI.

II. Background

A. Sorting on FPGA

Sorting acceleration on FPGA has been a hot topic in recent years. We briefly discuss the techniques that FANS adopts from the previous researches here.

The *compare-swap element* is the basic building block for hardware sorters. It compares two input values and swaps them into the correct ordering [8]. A compare-swap element usually contains a comparator and a 2-input multiplexer, which is suitable to be implemented using the Look-Up Tables (LUTs) on the FPGAs. With compare-swap elements, designers can develop the more complex *parallel merger* and the *merge tree*, which are described below.

1) Parallel Merger: This sorting unit takes two sorted arrays of numbers as input and then merges them into one final sorted array. The parallel merger can be built from a pipeline of multiple compare-swap elements. For clarification, in this work a *k-input merger* denotes the parallel merger that merges two arrays of length k. The bitonic merger (Figure 1) is one of the most widely used parallel mergers.

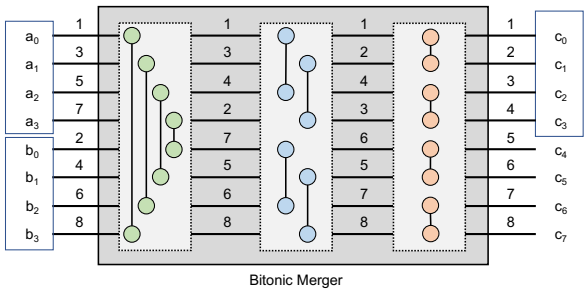

Fig. 1: An example of 4-input bitonic merger: each vertical line that connects two dots is a compare-swap element and the compare-swap elements in the same box are processed in the same cycle. c_{0-3} will be the outputs.

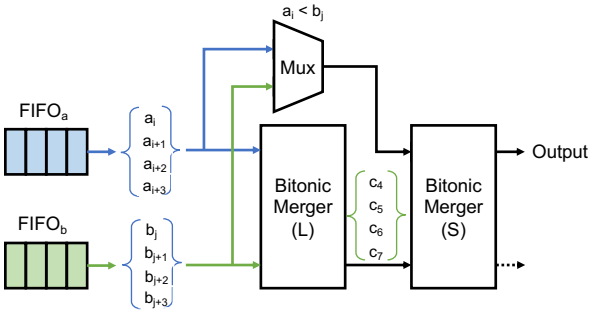

Fig. 2: The topology of a 4-input MMS merger.

If the two sorted input sequences are longer than k, the k-input merger has to be reused multiple times with extra control signals to ensure the outputs are in order. For example, in Figure 1, c_{4-7} represent the largest four elements of a_{0-3} and b_{0-3}. In the next cycle, c_{4-7} need to be sent back to the input of the same bitonic merger and merged with either a_{4-7} or b_{4-7} to create the second 4-element sorted tuples. The feedback paths from the output ports of the bitonic merger to its input ports often cause severe routing congestion and limit the scalability of the merger implementation on FPGAs [7].

To overcome the problem, [7] proposes a high-performance parallel merger called *MMS* that uses two bitonic mergers, as shown in Figure 2. The intuition is that the original feedback path for c_{4-7} can be calculated through another unfolded bitonic merger (L). In fact, the larger half outputs from the bitonic merger (L) is exactly the c_{4-7} and they are fed to another bitonic merger (S) with the next 4 elements from either input sequence a or b. The proof of the functional correctness can be found in [7]. In this work we use the same design methodology of MMS to construct the mergers for FANS [1].

[1]The mergers in [6], [16] can be alternatives to MMS, here we omit the comparison as this work focuses on implementing efficient end-to-end near-storage sorting systems instead of optimizing the fundamental merger blocks

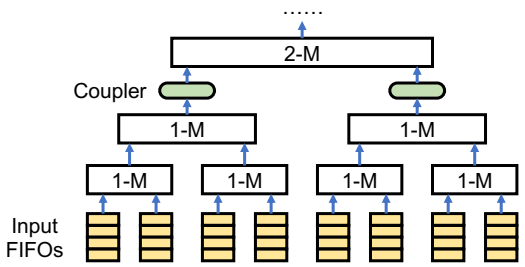

Fig. 3: An example of a parallel merge tree: *1-M* and *2-M* represent the 1-input merger and the 2-input merger. Different input sequences are stored in the different input FIFOs. The coupler is used to match the output rate of the lower-level merger to the input rate of the merger at the next level.

TABLE I: Model parameters used in [15]

	Symbol	Definition
Input Param.	N	Number of records in array
	r	Record width in bytes
Hardware Param.	β_{DRAM}	Bandwidth of the DRAM
Merge Tree Param.	f	Design frequency
	p	Merge tree throughput
	l	Number of leaves

2) Merge Tree: Using several levels of multi-input mergers, a parallel merge tree can be formed to merge multiple sorted sequences simultaneously [4], [15]. Figure 3 shows a merge tree that is able to process 8 sorted sequences. The root of the merge tree is a 2-input merger, which means the merge tree outputs 2 elements per cycle. Since we can uniquely identify a merge tree by (p, l), where p refers to the root throughput and l is the number of leaves [15], we denote the example in Figure 3 as a merge tree (2, 8).

To construct a merge tree (p, l), one can start from level 0 (the root) and use a set of $p/2^t$-input mergers for the t-th level. Note that the output throughput of the mergers at level $t+1$ can be half of the input width for mergers at level t. In this case, we insert a *coupler module* in between the two levels, which gathers the outputs from the low-level merger and matches the throughput rate. If l is larger than p, the merge tree uses 1-input mergers from level $log(p)$ to level $log(l)$.

B. Performance Modeling on Merge Tree

The performance of a merge tree that targets DRAM-scale data can be modeled based on the tree configuration (p, l) and the characteristic (e.g., bandwidth) of the used DRAM [15]. [15] proposes a detailed performance model to analyze the kernel performance of the merge tree, which we find of great use and is adopted in our analysis of the merge tree kernel. Table I lists the necessary parameters that used in [15] to determine the DRAM-scale sorting time. Assume there are N unsorted elements initially stored in the DRAM, the merge tree kernel will recursively merge them in multiple passes until the N elements are completely sorted.

During the first pass, the merge tree merges the original unsorted input data from the DRAM and produces l-element sorted subsequences back into the DRAM. In the second pass, the l-element sorted subsequences are fed into the merge tree again and form l^2-element sorted subsequences, et al. In general, the total number of passes needed to sort N elements is $\lceil \log_\ell N \rceil$. On the other hand, since the merge tree outputs p r-byte elements per cycle to the DRAM, the effective merge tree throughput is $\min\{pfr, \beta_{\text{DRAM}}\}$. Therefore, the total time (i.e. the latency) to sort N elements in the DRAM is:

$$\text{Latency} = \frac{Nr \cdot \lceil \log_\ell N \rceil}{\min\{pfr, \beta_{\text{DRAM}}\}}. \quad (1)$$

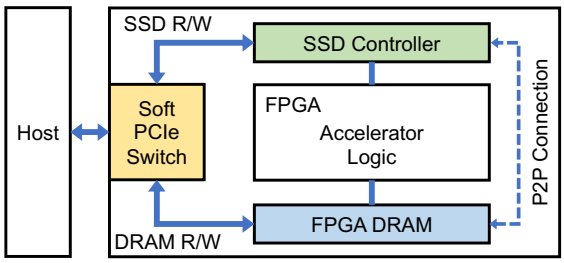

Fig. 4: High-level diagram of Samsung SmartSSD.

C. Samsung SmartSSD

Samsung SmartSSD is the industry's first programmable computational storage, where a Xilinx KU15P FPGA is integrated with a 3.84 TB NAND flash into the same package in the U.2 format [17]. With SmartSSD, many of the computational tasks can be offloaded from the host to the FPGA, which is next to the flash. By reducing the data transfer between the host and the flash, we could potentially save the host-drive bandwidth and boost the kernel performance [18].

As seen in Figure 4, the datapath between the host and the drive is through PCIe Gen3.0×4 [19]. The FPGA inside SmartSSD is equipped with a 4 GB DRAM, which is exposed to the FPGA kernels as its internal DRAM and to the host as a common memory area (CMA). The CMA is exposed to the host address space as a PCIe Base Address Register (BAR) and can be mapped into an application address space using buffer allocation. Once mapped, the host can initiate *peer-to-peer (P2P) transfers* between the SSD and the FPGA DRAM. Note that although the P2P transfer is initiated by the host, the actual data transfer will directly go through the PCIe link connecting the SSD controller and the FPGA DRAM without host involvement.

III. System Architecture of FANS

In this section, we present the system architecture of FANS. First we show the internal architecture of the sorting kernel that works with the FPGA DRAM. Then we illustrate how we improve the end-to-end system performance that takes into consideration the performance of the merge tree sorting kernel and the flash drive access bandwidth.

We specifically split the entire merge sort process into two phases: the sorting phase and the merging phase. In the sorting phase, the data is sorted into DRAM-scale subsequences and the merging phase assembles the subsequences into the final

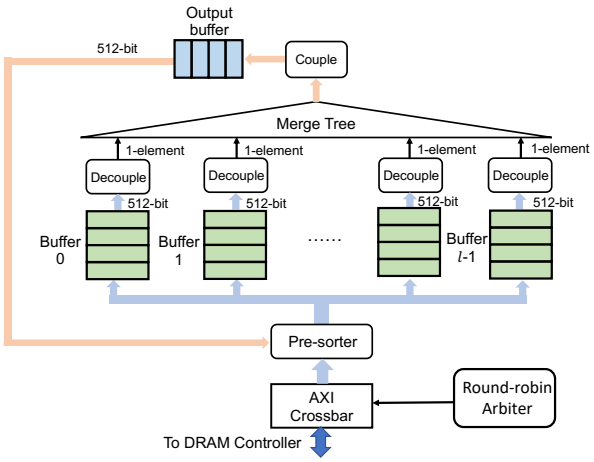

Fig. 5: The internal architecture of the sorting kernel.

output. Note that in the merging phase, the size of the merged output is usually larger than the FPGA DRAM capacity and we have to write the partially merged sequences directly back onto the flash when the DRAM capacity has been fully occupied.

A. Sorting Kernel

One of the key components in the FANS system is the merge tree kernel implemented on the FPGA. We adopt the merge tree design described in Section II-A2 and uses the same definition of (p, l) to denote the tree configuration. A detailed micro-architecture of our merge tree kernel is shown in Figure 5.

To make full use of the DRAM bandwidth, we rely on the *burst* mode in the AXI protocol to read data into each leaf in the merge tree and write back the output from the tree root. The burst sizes of the read and the write operations are 1KB and 4KB respectively. Each AXI transaction is 512-bit wide, which consists of multiple input data elements depending on the element width. The *decouple* unit splits the AXI transactions into individual records. Each leaf has its own buffer to store the burst transactions. In order to hide the DRAM access latency, the buffer size is set to be able to accommodate at least 2 full AXI bursts. To make sure that each leaf merger has equal priority in receiving data from the DRAM, we check the available space of each leaf's input buffer in a round-robin fashion: whenever a leaf buffer has enough space to accommodate another burst input, we will dynamically initiate an additional AXI read burst request for it.

Meanwhile, it would be beneficial that we start from some sorted data chunks instead of the entirely unsorted data [12]. As a result, we place a pre-sorter after the data that the AXI bus reads from the DRAM. The pre-sorter is implemented using a bitonic sorting network that sorts the AXI read data in order before distributing them into the leaf buffers. Since the bitonic sorting network is fully pipelined, the pre-sorting step will not affect the throughput of the overall design.

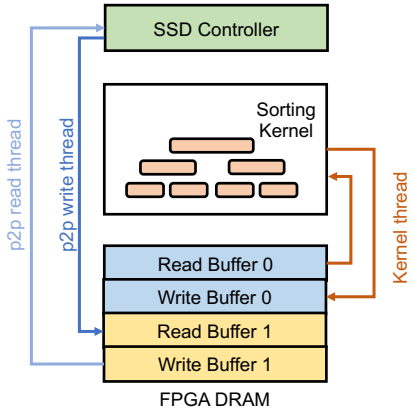

Fig. 6: The double buffering scheme with the host-side multi-threading in the sorting phase

B. Sorting Phase of FANS

Initially the original data are stored entirely in the flash and we fetch them into the FPGA DRAM in batches. We use the merge tree sorting kernel described in the previous sub-section to sort the data in one or more passes. Assume the configuration of the sorting kernel is (p_1, l_1) where p_1 refers to the merge tree throughput and l_1 is the tree leaf number. After each pass, the partially sorted chunks will grow by l_1 in size correspondingly. Note that since the bandwidth of the FPGA DRAM is higher than the flash and the FPGA DRAM is closer to the kernel, a natural design choice is to sort the data for multiple passes in the FPGA DRAM before we write them back to the flash.

Since the end-to-end execution time of the sorting phase includes both the sorting time and flash access time, we apply the double buffering technique [20] to improve the overall performance. As shown in Figure 6, the FPGA DRAM is split into two sets and each set contains two buffers of equal size. The merge tree kernel works with a single set each time: in one pass, it reads the data from the read buffer 0 and write to the write buffer 0; in the next pass, it reads the partially sorted data from the write buffer 0 and write to the read buffer 0, etc. At the same time, the write buffer 1 sends the sorted results of the previous batch back to the flash and the read buffer 1 fetches the next batch of data to be sorted from the flash. Please note that although the P2P transfer can stream data directly between the FPGA DRAM and the flash, it still relies on the host to issue the transfer commands. In FANS, we use three threads on the host side to overlap the data transfer and FPGA kernel execution.

C. Merging Phase of FANS

The merging phase requires the merge tree kernel to support additional host-FPGA communication feature, which will be explained below. As a result, we use another merge tree kernel and assume the tree configuration is (p_2, l_2) in this case. Since the size of the output data in this phase will exceed the capacity of the FPGA DRAM, we explicitly reserve l_2 input buffers in the FPGA DRAM, each for a sorted chunk of data that is

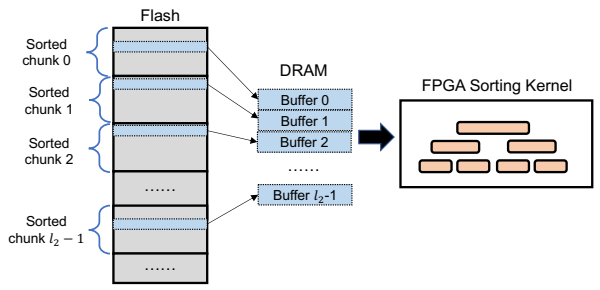

Fig. 7: The illustration of using the DRAM as buffers for the flash in the merging phase.

stored in the flash. There is also an output buffer to store the merged output data, and the size of the output buffer is equal to the sum of the input buffers. A detailed example is shown in Figure 7. At the beginning of the merging phase, the first batch of the sorted chunk 0 is fetched into the buffer 0 in the FPGA DRAM and the first batch of the sorted chunk 1 is also fetched into the buffer 1 in the FPGA DRAM, etc. After all of the buffers are filled with data, the merge tree kernel starts to merge these data into a larger sorted chunk and write it back to the FPGA DRAM. If the data in a buffer have been consumed by the kernel, then we will fetch another batch of data from its corresponding sorted chunk in the flash to the same buffer again. For example, in Figure 7 we will fetch the second batch from the sorted chunk 0 to the buffer 0 after the first batch is fully fed into the merge tree.

Depending on the relative order of different data batches, some buffers will be emptied earlier than others. To make the merging result correct, we have to suspend the merge tree kernel, fetch the next batches for those empty buffers and then resume the merge tree kernel again. To support this feature, the merge tree kernel needs some modifications:

- All of the on-chip storage elements that store the intermediate data and states (e.g., registers and FIFOs) should not be reset unless specified after the kernel is resumed.
- The merge tree kernel should keep track of the amount of data that have been sent into each leaf node. Whenever a leaf node has consumed the same amount of data as the FPGA DRAM buffer size, it means the corresponding buffer in the FPGA DRAM is already empty and the merge tree kernel should be suspended.
- The FPGA needs to message the host which buffers are already empty when the kernel is suspended, so that the host can issue the P2P transfer commands for the corresponding buffers.

To hide the latency of the flash storage access and improve the end-to-end performance, we utilize the same double buffering technique as the one in the sorting phase.

IV. Performance Modeling of FANS

In this section, we combine the external memory model and extend the sorting kernel analysis from Section II-B to analyze the performance of FANS. In addition to the parameters summarized in Section II-B, we list a few more parameters that we will take into consideration in Table II.

TABLE II: Extra parameters for FANS

Symbol	Definition
n	Number of elements the pre-sorter can sort
M	Size of sorted chunk after the sorting phase
β_{read}	Read bandwidth of the flash
β_{write}	Write bandwidth of the flash
(p_1, l_1)	Merge tree configuration of the sorting phase
(p_2, l_2)	Merge tree configuration of the merging phase

We show that the bottlenecks of the two phases may come from the FPGA DRAM bandwidth, the flash bandwidth and even the choice of intermediate sorted chunk size M after the sorting phase. Based on our analysis, we give an optimized configuration for the current Samsung SmartSSD device. Our analysis also serves as an in-depth case study that could help the vendors improve their near-storage computing products.

A. Sorting Phase

In the sorting phase, the initial input data produced by the pre-sorter is in a sorted chunk of size n and then we feed them into the merge tree of (p_1, l_1) through multiple passes. We use M to denote the size of the final sorted output after the sorting phase. Thus the number of passes is $\lceil \log_{l_1}(M/n) \rceil$. Using the equation 1 in Section II-B, the total time for sorting a batch of M data can be expressed as:

$$t_M = \frac{Mr \cdot \lceil \log_{l_1}(M/n) \rceil}{\min\{pfr, \beta_{\text{DRAM}}\}}. \tag{2}$$

Assume the available FPGA resources allow us to properly scale the merge tree to saturate the DRAM bandwidth, then the kernel performance β_{kernel} is :

$$\beta_{\text{kernel}} = \frac{Mr}{t_M} = \frac{\beta_{\text{DRAM}}}{\lceil \log_{l_1}(M/n) \rceil} \tag{3}$$

In the sorting phase, since we overlap the flash read and write with the kernel execution, the actual performance of sorting the data is

$$\beta_{\text{sort}} = \min\{\beta_{\text{read}}, \beta_{\text{kernel}}, \beta_{\text{write}}\}$$

Therefore, the total time for sorting the entire data of size N would be:

$$t_1 = \frac{N}{M} \cdot \frac{M}{\beta_{\text{sort}}} = \frac{N}{\min\{\beta_{\text{read}}, \beta_{\text{kernel}}, \beta_{\text{write}}\}} \tag{4}$$

B. Merging Phase

In this case, the data in the DRAM will go through the merge tree by one pass and the throughput of the merge tree kernel itself is $\min\{pfr, \beta_{\text{DRAM}}\}$. The end-to-end merging performance is also dependent on the flash read and write bandwidth, and since we use double buffering to hide the flash access latency, the actual merging performance will be

$$\beta_{\text{merge}} = \min\{\beta_{\text{read}}, \min\{pfr, \beta_{\text{DRAM}}\}, \beta_{\text{write}}\}$$

Since the read and write bandwidth of the flash are smaller than those of the FPGA DRAM and one can easily implement

a merge tree that saturates the flash access bandwidth, the merging performance can be simplified as:

$$\beta_{\text{merge}} = \min\{\beta_{\text{read}}, \beta_{\text{write}}\} \tag{5}$$

We use (p_2, l_2) to denote the configuration of the merge tree in the merging phase. The merging phase starts with partially sorted chunk of size M, so we need $\lceil \log_{l_2}(N/M) \rceil$ more passes to get the entire N size data to be sorted. As a result, we derive the total time of the merging phase as follows:

$$t_2 = \lceil \log_{l_2}(N/M) \rceil \cdot \frac{N}{\min\{\beta_{\text{read}}, \beta_{\text{write}}\}} \tag{6}$$

C. Optimized Configuration for Samsung SmartSSD

The above analysis on the two phases applies for any external merge sort implementations when using FPGA as the hardware accelerator. For Samsung SmartSSD we can get an optimized sorting configuration based on its physical characteristics. Inside SmartSSD, the SSD controller is connected to the FPGA through the PCIe Gen3.0×4 links, which has a theoretical 4 GB/s full-duplex bandwidth. Our profiling shows that the actual bandwidths for continuous read and write are both around 3 GB/s. Meanwhile, the FPGA is equipped with a single DRAM, whose theoretical bandwidth is 16 GB/s and the actual bandwidth is around 14 GB/s in half-duplex.

For the sorting phase, we notice that the kernel performance is lower than the flash read or write bandwidth. There are three reasons for the phenomenon. First, there is only one DRAM bank connecting to the FPGA kernel and the effective DRAM read and write bandwidth β_{DRAM} is roughly 7 GB/s. Second, sorting random data into DRAM-scale chunks (e.g. hundreds megabytes) usually takes more than 2 passes, as constructing a merge tree with a larger number of leaves will require more FPGA resources than available. Finally, due to the limit of the SmartSSD system, the data from the flash must be first written to the FPGA DRAM and then be read into the sorting kernel, which will further reduce the effective DRAM bandwidth for the sorting kernel. As a result, the total time for the sorting phase in Samsung SmartSSD is in equation 7

$$t_1 = \frac{N \cdot \lceil \log_{l_1}(M/n) \rceil}{\beta_{\text{DRAM}} \cdot (1-\gamma)} \tag{7}$$

where γ represents the DRAM degradation factor and indicates that P2P transfers also consume DRAM bandwidth.

The analysis of the merging phase for Samsung SmartSSD stays the same and we use β_{read} to denote the minimum number of the flash read and write bandwidth. Then we derive the total time of both phases in equation 8.

$$t_{\text{total}} = \frac{N \cdot \lceil \log_{l_1}(M/n) \rceil}{\beta_{\text{DRAM}} \cdot (1-\gamma)} + \lceil \log_{l_2}(N/M) \rceil \cdot \frac{N}{\beta_{\text{read}}} \tag{8}$$

We can see that, for the current Samsung SmartSSD device, both the communication bandwidth of the slow storage device and the merge tree configuration on the FPGA could be the bottlenecks to the end-to-end performance. Our modeling provides more comprehensive analysis compared to the previous belief that only the flash bandwidth could be the weakpoint [14]: ensuring that the merge tree's throughput saturates the DRAM's bandwidth is not enough, one must take the tree leaf number and the intermediate sorted chunk size after the sorting phase into consideration to best overlap the merge tree kernel execution and the flash access.

TABLE III: Detailed Configuration for Samsung SmartSSD.

Component	LUT	Flip Flop	BRAM
Available	326679	668532	647
$(p_1 = 2, l_1 = 64)$	160543	243145	127
Utilization	49%	36%	20%
$(p_2 = 1, l_2 = 64)$	152646	246865	127
Utilization	47%	37%	20%
Sorted chunk size M		256MB	

To get good sorting performance, we need to select both the appropriate merge tree configuration (p, l) and the intermediate sorted chunk size M.

- Firstly, since the FPGA DRAM and the flash bandwidth is relatively small, we only need to choose the minimal p_1 and p_2 required to saturate the FPGA DRAM bandwidth and the flash bandwidth. For example, when each data element takes 16 bytes and assume the merge tree kernel is running at 250 MHz, a merge tree with $p_1 = 2$ for the sorting phase and another merge tree with $p_2 = 1$ for the merging phase is optimal.
- Secondly, with the tree throughput p determined, we try to build the merge trees with the maximum number of leaves l such that the on-chip resources allow, as increasing l_1 and l_2 could always be beneficial.
- Finally, the choice of M should also be appropriate. Although we get the optimal value of M from solving equation 8, we provide the intuition as below. In the sorting phase, we fetch all of the data from the storage to the FPGA kernel only once and the performance is bounded by the kernel execution. If reducing M can reduce the number of passes that the sorting kernel spends on each unsorted data batch, the sorting phase's performance will be improved. On the other hand, the merging phase always runs at a speed that saturates the flash access bandwidth. But, if increasing M could reduce the number of passes that the data travels between the flash and the on-chip merge tree, the merging phase's performance could also be improved. Considering that the number of passes (the logarithm item in the equation 8) needs to be rounded into discrete integers, M is chosen as the minimum number that will not increase the number of merging passes.

The configuration that we actually achieve on Samsung SmartSSD is shown in Table III and we leave the detailed explanation in Section V-B.

D. Architectural Insights

Our performance model is more than a guide to determine the optimal design choices for the current Samsung SmartSSD devices. In fact, the near-storage device vendors can also benefit from our study and foresee the potential performance gain from the architectural advancement.

- The drive's bandwidth plays an important role in the end-to-end execution of the merging phase. While the current solid-state drives use the four-lane PCIe Gen3 links as the primary interconnection to the host and the FPGA, the technology scaling that incorporates the next generation's PCIe links will boost the merging performance.
- While the capacity of the FPGA DRAM does not impact the performance, its bandwidth is the primary bottleneck in the sorting phase. We anticipate that the high-bandwidth memory (HBM), which features smaller capacity but much higher bandwidth will be a better candidate to perform sorting tasks (e.g. [21]) in near-storage computing devices.
- The requirement that SSD data must first be transferred to the FPGA DRAM further reduces the performance in the sorting phase. In contrast, allowing the PCIe data to be directly sent to the FPGA kernel as in [22] could remove the DRAM degradation factor γ in equation 8.

V. Evaluation of FANS

In this section, we evaluate the overall performance of FANS. First, Section V-A describes the experimental setup. Then, Section V-B shows the detailed merge tree configuration, the performance breakdown of FANS's execution and the impact of P2P transfers on the effective FPGA DRAM bandwidth. Third, Section V-C compares FANS with previous works and illustrates why FANS achieves better performance. Lastly, we quantitatively evaluate the benefits that arise from the near-storage characteristics of Samsung SmartSSD.

A. Experimental Setup

1) System Setup: We perform our experiments on Samsung SmartSSD with the Xilinx OpenCL runtime. We prepare the bitstreams of the two phases in advance and reprogram the FPGA at runtime to switch between different bitstreams. The host-side multi-threading is implemented through POSIX Threads (pthreads). The sorting kernel is developed using Verilog and is synthesized and implemented using Xilinx Vitis 2019.2. We also tune the kernel so it can run at a minimum frequency of 230MHz.

2) Benchmark: The benchmark used in our experiments is generated from the public Terasort benchmark [23], where each record is 100 bytes with a 10-byte key and a 90-byte value. To save the memory and PCIe bandwidth, we use the same method in [14] that converts the 90-byte value into the 6-byte index. That is, the record we actually sort is 16 bytes with a 10-byte key and a 6-byte index. We generate 2^{33} records, which is 128 GB in size and store them into SmartSSD in advance.

B. Performance of FANS

1) Merge Tree Configuration: We implement the merge tree as Section IV-C suggests: for the sorting phase, we set the tree throughput p_1 to 2; for the merging phase, we set the tree throughput p_2 to 1. We also use a pre-sorter that is able to sort 4 elements every cycle in the sorting phase.

TABLE IV: Performance breakdown when sorting 128GB data

Phase	Time (s)
Sorting phase	85
Merging phase	100
Reprogram	4
Total	189

Then we maximize the corresponding leave number l_1 and l_2 that on-chip FPGA resources allow and make sure the design does not have a routing failure. Although the advertised AXI burst size is 4KB to achieve the peak DRAM performance, we find that reducing the AXI burst size to be 1KB does not hurt the DRAM read performance. Therefore, we issue the AXI burst in a granularity of 1KB and set the size of each leaf node buffer to be 2KB, which allows 2 outstanding AXI bursts to be performed on-the-fly. Furthermore, to relieve the routing congestion, we use LUTRAMs instead of BRAMs to implement most of the leaf node buffers. Finally, we find the maximum l_1 and l_2 to be 64, respectively. After fixing l_1 and l_2, the intermediate sorted chunk size M is also determined to be 256MB. A detailed resource utilization of the merge tree kernels is listed in Table III.

2) Performance Breakdown: The performance of each phase of FANS when sorting 128GB data is shown in Table IV. In the sorting phase, the time spent is around 85 seconds, meaning the average performance is 1.5GB/s. The performance is much less than the roughly 3GB/s flash read and write bandwidth and it matches the analysis in Section IV-A that the merge sort kernel is the bottleneck in this phase.

In the merging phase, FANS takes 2 passes to merge the 256MB sorted subsequences into the final sorted results. Ideally the end-to-end performance for phase 2 is half of the flash read and write bandwidth, which is around 1.5GB/s. The actual measurement of the performance is around 1.3GB/s. Although the gap in between is close, we believe it is from the overhead that halts the merge tree kernel too many times in a cumbersome way. In fact, the merge tree kernel is suspended whenever one of the DRAM buffers becomes empty.

3) Evaluation of the DRAM degradation factor γ: Section IV-C shows that the direct P2P transfers between the flash and the FPGA DRAM reduce the effective bandwidth for the merge sort kernel and thus harm the performance of the sorting phase. Section IV-D anticipates that if the FPGA can access the flash data directly from the PCIe transaction packets instead of from its DRAM, we can remove the DRAM degradation factor γ in equation 7. To validate the claim, we do another experiment that loads 256MB data onto the FPGA in advance and we measure the pure kernel performance of the merge tree kernel in the absence of P2P transfers.

As shown in Table V, the pure merge sort kernel without P2P transfers is 20% faster than the actual performance we observe in the sorting phase. This indicates that if the FPGA kernel is able to directly stream data into the flash via PCIe links, we can get another 20% performance speedup in the sorting phase.

TABLE V: Evaluation of γ

Kernel	Performance (GB/s)
With P2P	1.5
Without P2P	1.8

TABLE VI: Comparison between FANS and [14].

	Item	[14]	FANS
Hardware setup	DRAM BW. (GB/s)	16	16
	DRAM Capacity (GB)	1	4
	Flash Read BW. (GB/s)	2.4	3
	Flash Write BW. (GB/s)	2	3
Design Config.	No. of Leaf Mergers	16	64
	Frequency (MHz)	125	250
	Sorted chunk size M (MB)	512	256
Performance	Sorting Performance (GB/s)	0.21	0.68 (3.2×)

C. Comparison with Previous Work

[14] builds an FPGA-accelerated flash storage system where a Xilinx Vertex 7 FPGA is coupled with a custom flash expansion card via two FMC ports. Using the system, [14] is able to sort 10^{10} 16-byte key-index pairs or 150GB of data in 700 seconds. In contrast, FANS sorts 128GB of data in 190 seconds. Since the amount of time it takes to sort a dataset is linear to the size as long as the number of merging passes does not change, we can use the average performance, or the total size of data divided by the entire sorting time, to make a fair comparison. In terms of the average performance, FANS achieves 3.2× speedup over the sorting system in [14].

Table VI summarizes the difference of hardware and design choices between [14] and FANS. We notice that Samsung SmartSSD has a higher flash access bandwidth, which enables FANS to access the flash at a faster speed of 1.5×. The different design choices further broaden the performance gap: on the one hand, FANS allows us to use a merge tree configuration with a 4× larger number of leaves l to reduce the number of passes it takes to sort the entire data. On the other hand, FANS selects a smaller intermediate sorted chunk size M and uses the pre-sorter to decrease the number of passes that the kernel takes to access the FPGA DRAM, thus improving the performance in the sorting phase.

[15] presents an adaptive sorting solution that adapts the on-chip computational resources to the available off-chip memory bandwidth to optimize the sorting performance. While its model-based analysis motivates this work, the external sorting solution presented in [15] is based on a theoretical model of near-storage devices and is not applicable to Samsung SmartSSD. Specifically, [15] assumes a flash connected to four DRAM banks and the flash bandwidth is as much as 8GB/s. To fully utilize the flash bandwidth in the sorting phase, [15] proposes to have four merge sort kernels working with the four DRAM banks separately and each kernel has the same throughput as the flash. The four merge sort kernels are connected in a pipelined fashion so that the output of the current kernel will be fed as the input into the next kernel. While this method achieves the best flash bandwidth utilization in the described storage system, it cannot be used in the current near-storage devices such as Samsung SmartSSD. First, due to the cost and power consideration, Samsung SmartSSD has only one DRAM bank and pipelining merge sort kernels in this case is impossible. Second, the bandwidth of the current flash is only 4GB/s. Even if we have multiple DRAM banks, running those merge sort kernels with the same throughput of the flash will not fully utilize the DRAM bandwidth (e.g. 16 GB/s in half-duplex mode), thus making the system performance sub-optimal. Since the hardware in [15] and FANS is quite different, we omit the performance comparison.

TABLE VII: Effective communication bandwidth between the FPGA and the flash without P2P transfers.

Direction	Performance (GB/s)
flash to FPGA	1.5
FPGA to flash	1.3

D. Performance Benefits from near-storage acceleration

The feature of integrating the FPGA into the flash package and allowing P2P transfers without interfering the host gives another level of system gain. To quantitatively compare the gain, we perform a check-experiment using SmartSSD but relying on the host to access the flash storage as well as the FPGA: in this case the FPGA is utilized as a conventional accelerator and does not have the direct access to the flash.

As shown in Table VII, the involvement of the host reduces the effective bandwidth between the FPGA and the flash from 3GB/s to 1.5GB/s. Using the analysis in Section IV-A and Section IV-B, we anticipate that the performance bottleneck in the sorting phase will be shifting from the merge tree kernel to the effective FPGA-flash bandwidth and the performance of the merging phase is going to be directly reduced by 2×.

Besides, a main benefit from near-storage computing platforms is that it frees CPU cycles and reduces the DRAM usage on the host side. Although we cannot directly see this benefit from measuring the end-to-end sorting performance in this work, one can derive a qualitative analysis of the host-side gain using the same methodology found in [24].

VI. Conclusion

In this work, we propose FANS: an end-to-end FPGA accelerated near-storage sorting solution that is able to sort hundreds of gigabytes data on a single Samsung SmartSSD. FANS is built on top of a concrete performance model that analyzes the performance bottlenecks of the near-storage sorting process and indicates that the flash access bandwidth, the FPGA DRAM bandwidth, the configuration of merge tree kernel and the intermediate sorted chunk size all impact the end-to-end sorting performance. Through the optimized sorting configurations, FANS achieves 3.2× performance speedup over the previous FPGA-accelerated flash storage.

Acknowledgements

We acknowledge the support from the CRISP Program and the CDSC industrial partners, including Samsung (Samsung Electronics Co. Award# 20201855) and Mentor Graphics (CDSC Consortium fund). We also thank Nikola Samardzic and Kwan Huen for helpful discussions.

REFERENCES

[1] D. E. Knuth, *Art of Computer Programming: Sorting and Searching*, 2nd ed. Addison-Wesley Professional, 1998.

[2] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat, "TritonSort: A Balanced Large-Scale Sorting System," in *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[3] R. Chen, S. Siriyal, and V. Prasanna, "Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA," in *Proceedings of the 23th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2015.

[4] W. Song, D. Koch, M. Lujan, and J. Garside, "Parallel Hardware Merge Sorter," in *Proceedings of the 24th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2016.

[5] S. Mashimo, T. V. Chu, and K. Kise, "High-Performance Hardware Merge Sorter," in *Proceedings of the 25th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2017.

[6] P. Papaphilippou, C. Brooks, and W. Luk, "FLiMS: Fast Lightweight Merge Sorter," in *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[7] M. Saitoh, E. A. Elsayed, T. V. Chu, S. Mashimo, and K. Kise, "A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath," in *Proceedings of the 26th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2018.

[8] D. Koch and J. Torresen, "FPGASort: A High Performance Sorting Architecture Expoiting Run-time Reconfiguration on FPGAs for Large Problem Sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2011.

[9] J. Casper and K. Olukotun, "Hardware Acceleration of Database Operations," in *Proceedings of the 22th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2014.

[10] K. Manev and D. Koch, "Large Utility Sorting on FPGAs," in *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[11] H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "FPGA-Accelerated Samplesort for Large Data Sets," in *Proceedings of the 28th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2020.

[12] P. Papaphilippou, C. Brooks, and W. Luk, "An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics," in *Proceedings of the 30th international conference on Field-Programmable Logic and applications (FPL)*, 2020.

[13] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs," in *Proceedings of the 29th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2021.

[14] S.-W. Jun, S. Xu, and Arvind, "Terabyte Sort on FPGA-Accelerated Flash Storage," in *Proceedings of the 25th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2017.

[15] N. Samardzic, W. Qiao, V. Aggarwal, M.-C. F. Chang, and J. Cong, "Bonsai: High-Performance Adaptive Merge Tree Sorting," in *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2020.

[16] E. A. Elsayed and K. Kise, "Towards an Efficient Hardware Architecture for Odd-even Based Merge Sorter," in *Proceedings of the 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2019.

[17] "SmartSSD Computational Storage Drive: Installation and User Guide," https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/1_0/ug1382-smartssd-csd.pdf.

[18] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD," in *IEEE Computer architecture letters*, 2020.

[19] M. Jackson and R. Budruk, *PCIe Express Technology*, 1st ed. MindShare Inc., 2012.

[20] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth Optimization Through On-Chip Memory Restructuring for HLS," in *Proceedings of the 54th Design Automation Conference (DAC)*, 2017.

[21] Y. kyu Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "HBM Connect: High-Performance HLS Interconnect for FPGA HBM," in *Proceedings of the 29th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2021.

[22] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[23] "Sort Benchmark," http://www.ordinal.com/gensort.html.

[24] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and Arvind, "AQUOMAN: An Analytic-Query Offloading Machine," in *Proceedings of the 53th IEEE/ACM international symposium on microarchitecture (MICRO)*, 2020.