FPGA-Accelerated Samplesort for Large Data Sets

Han Chen, Sergey Madaminov, Michael Ferdman, Peter Milder
Stony Brook University
{han.chen.2,peter.milder}@stonybrook.edu,{smadaminov,mferdman}@cs.stonybrook.edu

ABSTRACT

Sorting is a fundamental operation in many applications such as databases, search, and social networks. Although FPGAs have been shown very effective at sorting data sizes that fit on chip, systems that sort larger data sets by shuffling data on and off chip are bottlenecked by costly merge operations or data transfer time.

We propose a new technique for sorting large data sets, which uses a variant of the samplesort algorithm on a server with a PCIe-connected FPGA. Samplesort avoids merging by randomly sampling values to determine how to partition data into non-overlapping *buckets* that can be independently sorted. The key to our design is a novel parallel multi-stage hardware partitioner, which is a scalable high-throughput solution that greatly accelerates the samplesort partitioning step. Using samplesort for FPGA-accelerated sorting provides several advantages over mergesort, while also presenting a number of new challenges that we address with cooperation between the FPGA and the software running on the host CPU.

We prototype our design using Amazon Web Services FPGA instances, which pair a Xilinx Virtex UltraScale+ FPGA with a high-performance server. Our experiments demonstrate that our prototype system sorts 2^{30} key-value records with a speed of 7.2 GB/s, limited only by the on-board DRAM capacity and available PCIe bandwidth. When sorting 2^{30} records, our system exhibits a 37.4x speedup over the widely used GNU parallel sort on an 8-thread state-of-the-art CPU.

CCS CONCEPTS

• **Theory of computation → Sorting and searching**; • **Hardware → Hardware accelerators**.

KEYWORDS

sorting; samplesort; partitioning; FPGA

ACM Reference Format:
Han Chen, Sergey Madaminov, Michael Ferdman, Peter Milder. 2020. FPGA-Accelerated Samplesort for Large Data Sets. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375304

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375304

1 INTRODUCTION

Sorting data has always been one of the fundamental computing operations. In the early days of computing, studies showed that 25% of all CPU time was spent on sort [13]. Today, application complexity has increased and data sets have grown to unprecedented sizes, with sort remaining a critical component of the dominant datacenter workloads such as large-scale data warehouses, databases, and web search. At datacenter scale, there is a constant need for large-scale sorting operations [9, 10], with performance and cost of running sort being a major consideration of application design.

The popularity of the sort operation has led to the development of many sorting algorithms with different properties, including the ubiquitous *quicksort* [11, 12] and *mergesort* [5]. However, fundamentally, the von Neumann architecture of modern CPUs is not well-suited for sort, regardless of algorithm, as CPUs are unable to leverage the vast available parallelism of the comparison operations. Moreover, sort is challenging not only for CPUs, but also for accelerators. Although sorting accelerators are able to exploit ample internal parallelism, sorting large data sets requires partitioning them into *buckets* and then aggregating the sorted buckets, an operation that has traditionally been bottlenecked on the memory bandwidth available to the accelerator.

Although many sorting accelerators have been proposed in the literature, none are able to simultaneously support large data sets and achieve high performance. Streaming sorting networks such as Spiral provide extremely fast sorters whose data set sizes are limited by the capacity of the on-chip memories (BRAMs and URAMs) available on the FPGA [24]. Accelerators that tackle larger sorting tasks divide large data sets into smaller buckets and combine the resulting buckets by performing multi-way merges either in software [23] or in hardware [16–19, 21, 22]. However, merge-based techniques also fall short due to their high resource costs and multiple memory round-trips required by the merge implementations. To achieve high performance on sorting large data sets, we therefore pursue an approach that avoids merging.

We propose a new technique for using FPGAs to accelerate the samplesort algorithm, which avoids merging by partitioning data into buckets that can be sorted and concatenated together. The key to our design is a novel parallel multi-stage hardware partitioner, which consists of multiple partitioner cores operating in a pipeline. Each partitioner stage splits data into approximately equal-sized buckets of non-overlapping data (i.e., where all values in the ith bucket are guaranteed to be less than all values in the $(i+1)$th bucket, but values within a bucket are not necessarily sorted). The buckets are stored by the partitioner in the off-chip memory accessible to the FPGA, and then read back from off-chip memory by the next partitioner stage, or by a parallel sorting network [24] that writes the final sorted sequence back to the host over PCIe.

Our partitioner approach is fundamentally more scalable than prior work that was based on merging sorted buckets. The main difference between a mergesort based system and a samplesort based system is that the samplesort approach uses partitioning before sorting, whereas mergesort uses merging after sorting. The benefit of the partitioning approach is that each record is independent, whereas the merging process must compare records to each other. This independence allows us to construct an inexpensive and high-throughput partitioner that can split data across a large number of buckets. By virtue of the partitioner's cost-efficiency, we are able to construct a multi-stage pipelined partitioning system that handles a large number of buckets while maintaining high throughput.

Using samplesort for FPGA-accelerated sorting provides several advantages over mergesort, while also presenting a number of new challenges that we address with cooperation between the FPGA and the software running on the host CPU. First, samplesort relies on an initial random sampling to ensure that buckets are of approximately equal size. Although slow and resource-intensive for an FPGA, initial random sampling is well-suited to run on a high-performance CPU; we therefore perform it in software. Second, although random sampling ensures that buckets will be of approximately equal size with high probability, occasional outliers are possible, with some buckets being larger than expected by the hardware. In this case, our hardware implementation detects this rare event and triggers a software function to re-sort the bucket in parallel with the FPGA sorting the subsequent buckets. Furthermore, our experimental results show that these outliers are extremely rare in practice (e.g., 1 out of 2^{18} buckets in a typical test). Finally, even with the efficient partitioning hardware design, the data set size that the FPGA can sort is still limited by the FPGA on-board DRAM capacity (64 GB in our prototype). To scale to larger data sets, we rely on the CPU to perform an initial coarse-grained partitioning operation, decomposing data into portions that fit within this constraint.

We designed and evaluated our prototype hardware and software system using an Amazon F1 FPGA instance, comprising an 8-thread CPU connected via PCIe to a Xilinx Virtex UltraScale+ (XCVU9P) FPGA with 64 GB of on-board DDR4 memory. Our prototype operates on a data set of 2^{30} 14-byte records (10-byte key and 4-byte index), which is larger than any data set running on prior high-throughput FPGA sorting systems, and demonstrates that our system can saturate PCIe bandwidth in a single round trip from host memory to FPGA and back to host memory with a speed of 7.2GB/s. The size of the data set that the prototype system can process at this rate in hardware is only limited by the on-board DRAM capacity. Overall, we demonstrate a 37.4x speedup over the widely used GNU parallel sort on an 8-thread state-of-the-art CPU when sorting 2^{30} records.

The rest of this paper is organized as follows. In Section 2, we present our motivation for choosing samplesort for FPGA acceleration of sorting. In Section 3, we introduce our approach to accelerating samplesort and describe our hardware-software strategy. Section 4 presents a detailed view of our hardware architecture. Section 5 describes the hardware partitioner and Section 6 describes the structure of the hardware sorter. Section 7 presents the implementation of the hardware and software in our prototype system. Lastly, Section 8 evaluates our prototype system on an Amazon F1 instance and Section 9 offers concluding thoughts.

2 MOTIVATION

Given the rich history of sorting algorithms and implementations, it may initially be surprising that a high-performance FPGA design for sorting large data sets did not previously exist, requiring us to develop a novel technique. We therefore begin our discussion of sorting large data sets on FPGAs by explaining the motivation behind the various parts of our design and how they were influenced by the prior work in the field.

In the literature, there are many examples of accelerator designs that focus on high-throughput hardware structures for sorting on-chip data. These techniques effectively leverage the compute parallelism available on FPGAs to yield extremely high-performance implementations. However, the performance of these accelerators relies on the massive on-chip memory bandwidth of BRAMs and URAMs, which fundamentally limits the size of the data sets that can be sorted to ones that can fit into these low-capacity on-chip memories. Prominent examples of this approach are the Spiral streaming sorting networks [24], which represent a class of flexible sorting hardware structures that can scale to different cost-performance trade-offs. Using Spiral [24] to generate a high-throughput sorter yields high performance (e.g., 32 GB/s throughput), but can only accommodate data sets of up to 2^{17} records before running out of on-chip storage on a Xilinx Virtex UltraScale+ VU9P FPGA. Although alternative accelerator designs (including other Spiral configurations) can trade some performance to handle slightly larger data sets, accelerators such as those built by Spiral are inherently limited by the total available on-chip memory. Despite streaming sorting networks being unable to handle our target data set sizes (2^{30} records), they provide a useful building block for larger sorters. For our prototype, we use the Spiral generator to produce the sorting unit that we incorporate into our design.

To handle large data sets, the records to be sorted must be divided into smaller *buckets*, whose size is determined by the maximum practical size of the underlying sorting unit. The buckets can be independently sorted while their contents are on chip, but all buckets must be temporarily stored in off-chip memory (i.e., DRAM) so that they can be later combined to form the final single sorted sequence. A natural approach to combine the buckets is by having software merge them, using the CPU to stream through the buckets and selecting the lowest record among the buckets at each step [23]. However, although this approach is capable of accelerating sorting of data sets that do not fit into on-chip FPGA memories, using software to perform a multi-way merge quickly becomes the bottleneck of the system. As a result, using a software merger to combine buckets yields only a modest increase in the data set size that can be practically sorted by this approach, allowing sorting of fewer than 2^{25} records with 4-byte keys, or even fewer records with the 10-byte keys that we target. Notably, this is the only full-system sorter design capable of handling large data sets that we were able to find in the literature, yet it falls short of being able to sort our target data sets.

The limits of software merging have been recognized, and FPGA-based solutions for merging buckets stored in off-chip memory have been investigated recently. For example, the FIFO merge sorter and tree merge sorter structures were demonstrated by sorting a stream of approximately 2^{29} values produced by a random-number

generator [14]. This approach demonstrated a practical hardware merger implementation capable of merging 2^7 buckets. Unfortunately, the throughput achieved by these structures (sorting 4-byte keys) was only 667 MB/s, which falls below the performance of sorting on a non-accelerated multi-core CPU. Another design, the merge sorter tree, was shown to merge up to 4096 buckets [22]. However, the throughput and peak frequency of the merge sorter tree are constrained by its feedback path. The resulting design can produce only one record per cycle at 150MHz, severely limiting its performance. More efficient hardware parallel merge trees were proposed to improve the feedback path and output multiple records per cycle [16, 18, 19, 21]. Most recently, the Fast Lightweight Merge Sorter [17] was demonstrated with 2-way merge that produces 64 records per cycle. Overall, we found that all existing hardware merger designs share common pitfalls. To achieve high performance, mergers require large amounts of FPGA resources, which limits the number of buckets that can be merged and leaves little room for a sorting unit to sort the individual buckets. Moreover, mergers that achieve high throughput are unable to concurrently merge many buckets in a single pass, losing performance due to requiring multiple round-trips to off-chip DRAM to perform a full sort. As a result, although designs for hardware mergers are available, none have been used to build an FPGA-based accelerator for sorting large data sets that could compete with a modern CPU.

Because of the limitations of hardware mergers, our aim was to develop a sorting system for large data sets that combines buckets without merging in software or hardware. Our search led us to explore the less well-known samplesort algorithm as the foundation of our implementation [8]. Samplesort begins by partitioning the records to be sorted into non-overlapping buckets, such that the elements within each bucket are in random order, but the relative sorted order of the buckets is known. The buckets are then individually sorted and the sorted results are written directly into the appropriate location in the output sequence. The output location of each sorted bucket is precisely known after completing the initial partitioning step, because the relative ordering of the buckets and the number of elements in each bucket are known at that point.

The key strength of samplesort arises due to its approach for selecting *splitters*, the keys that determine bucket boundaries used by the partitioning step. Traditional bucket sort implementations were considered and abandoned by prior work [14] because they suffer from radically varying bucket sizes that depend on the distribution of the input data. Conversely, we pursued samplesort because, with high probability, it selects bucket boundaries such that the buckets are all of approximately equal size, making the algorithm amenable to implementation with fixed bucket capacities. In the rare event that a bucket exceeds its target capacity, our design informs the software about the overflow, allowing it to recover by re-sorting the overflowing bucket.

Finally, we note that, unlike merging, where elements from all buckets must be compared against each other, the partitioning process can treat each record completely independently. This makes the partitioning process well-suited for implementation as a set of independent parallel streaming accelerators that are relatively inexpensive in terms of FPGA resource consumption, as no comparisons are required among concurrently processed elements and all comparators can be easily pipelined without any feedback paths.

Algorithm 1: Samplesort Algorithm.

1 **Function** *SampleSort(*$R[r_1, r_2, ..., r_{2^N}], k, P$*)*
2 Select $k * 2^P$ samples from R. $S = [S_1, S_2, ..., S_{k*2^P}]$;
3 Sort the samples S;
4 Select $2^P - 1$ splitters,
$[s_0, s_1, ..., s_{2^P-2}] = [S_k, S_{2k}, ..., S_{k(2^P-1)}]$;
5 **foreach** $r \in R$ **do**
6 Put r into jth bucket b_j such that $s_{j-1} < r \leq s_j$;
7 Return concatenate(Sort(b_0), Sort(b_2), ..., Sort(b_{k-1}));

As a result, our partitioner design easily achieves the target, approaching ~ 8 GB/s throughput with data arriving via the PCIe Gen3 x16 interface when sorting 2^{30} records with 10-byte keys.

3 FPGA ACCELERATION OF SAMPLESORT

This section presents a high-level overview of our novel FPGA/CPU technique for sorting large data sets. First, Section 3.1 introduces the samplesort algorithm and Section 3.2 details our strategy for accelerating it using hardware and software. Then, Section 3.3 discusses how we use software to enable further scalability.

3.1 Samplesort

Samplesort is a sorting algorithm often used in parallel processing systems [2, 4, 15, 20]. The algorithm, illustrated as pseudocode in Algorithm 1, operates in three stages: sampling, partitioning, and sub-sorting. First, the sampling stage (lines 2–3), samples a portion of the input records to estimate the distribution of the data set; it chooses a set of records and sorts them. Next, the partitioning stage (lines 4–6) selects a set of these records as *splitters*, which are used to define the ranges of non-overlapping *buckets*. Records are partitioned into buckets by comparing each record with the values of the splitters. Then, the sub-sorting stage (line 7) sorts each bucket and concatenates them to produce the final sorted sequence. Because buckets are independent, sorting them is an independent parallelizable operation.

Because efficient parallelization relies on the buckets having similar size, a key consideration is in the size of the random sample used for estimating the data distribution. Larger sample sizes result in better estimates of the data distribution and less variability in the bucket sizes. To partition data into 2^P buckets, we choose $k \times 2^P$ random records as candidate splitters, where k is called the oversampling ratio. This results in a calculable upper bound on the probability that a bucket grows beyond a given size [6]. In practice, choosing an appropriate k ensures that there is a near-zero probability that any given bucket will be larger than a desired upper bound; in the rare instances where this upper bound is exceeded, the situation can be detected and corrected by re-sorting that bucket, albeit at a small cost to performance. Our prototype uses $k = N$, similar to CPU-based samplesort implementations, where 2^N is the total number of records to sort.

3.2 Hardware-Software Samplesort Strategy

Typical hardware sorters, such as streaming sorting networks [24], require storing all input, output, and intermediate data on chip.

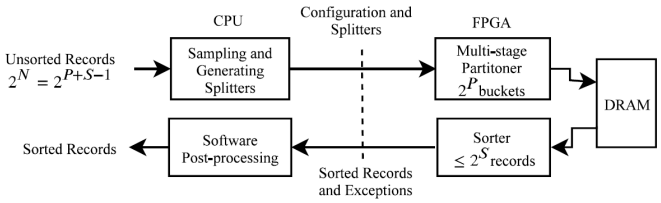

Figure 1: CPU-FPGA samplesort strategy.

Algorithm 2: FPGA Accelerated Samplesort Algorithm.

Function *FPGASampleSort*($R[r_1, r_2, ..., r_{2^N}], k, P$)
- Select $k * 2^P$ samples from R. $S = [S_1, S_2, ..., S_{k*2^P}]$;
- Sort the samples S;
- Select $2^P - 1$ splitters, $[s_0, s_1, ..., s_{2^P-2}] = [S_k, S_{2k}, ..., S_{k(2^P-1)}]$;
- Send splitters and configuration to FPGA;
- Send records R to FPGA;
- Wait for FPGA to return the result to R;
- Re-sort any oversized buckets;
- Return R;

These sorters are therefore very fast, but the amount of data they can sort is necessarily limited by the amount of available on-chip memory. To sort larger data sets, we propose a new strategy to implement samplesort using hardware and software. The system first breaks down input records into buckets (each of which can fit in on-chip memory), and then sorts each bucket to form the final result.

This process is illustrated in Figure 1 and Algorithm 2. First, the system begins with $2^N = 2^{P+S-1}$ records in host memory. Software samples the records and sorts the sample to create splitters. The splitters, records, and configuration information are streamed from the CPU to the FPGA via PCIe. Inside the FPGA, a hardware partitioning module partitions the records into 2^P buckets; each bucket holds an average of 2^{S-1} and a maximum of 2^S records. The FPGA stores these buckets in the FPGA's on-board DRAM. Each bucket is then read from the DRAM into a streaming sorting network; the maximum bucket size 2^S is chosen to allow all data to fit on-chip, ensuring that each bucket can be sorted efficiently. After sorting, the buckets are concatenated and streamed back to the host. Lastly, the CPU will correct any problems caused by occasional oversized buckets (those with $> 2^S$ records).

As an example, the prototype system we describe and evaluate in Section 7 targets $2^{P+S-1} = 2^{30}$ records in hardware, with $P = 18$ and $S = 13$ (that is, the partitioner splits data into up to 2^{18} buckets and the sorter sorts buckets of up to 2^{13} records). Our results show that the capacity of the prototype system (i.e., the values of N, S, and P) is limited only by the size of the on-board DRAM capacity and not the available FPGA resources.

3.3 Scalability

The number of records to be sorted by the proposed hardware-software system is inherently scalable. The hardware control logic

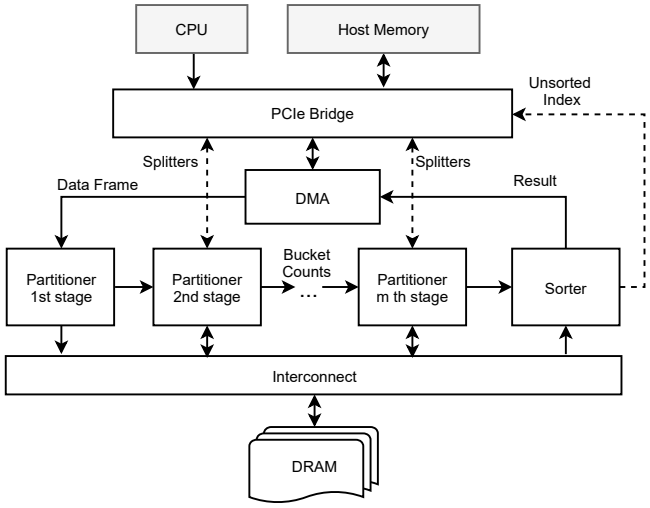

Figure 2: Overview of the sorting system.

is designed to be flexible to efficiently accommodate smaller data set sizes, dynamically adjusting the amount of partitioning required to ensure high throughput. Specifically, when sorting a data set of $< 2^N$ records, the system will reduce the number of buckets to maintain the average bucket size of 2^{S-1} records. This is important because smaller buckets would lead to under-utilization of the hardware sorter and an unneeded loss of performance. Furthermore, our system handles any number of records; in this paper, 2^N is chosen for readability of formulas and results. Both our partitioner and sorter can accept arbitrary-size input.

A different challenge is observed when the data set grows to sizes larger than the FPGA's on-board DRAM capacity. That is, the hardware system can sort $2^N = 2^{P+S-1}$ records, but N is limited by the on-board memory. With sufficient DRAM, N can be quite large (e.g., our prototype system can sort up to $2^N = 2^{30}$ records in hardware); beyond that, further scalability requires extending our approach through software. For larger data sets, the system can perform an initial layer of partitioning using the CPU. For example, if one wants to sort a data set of 2^{H+N} records with a hardware system that sorts 2^N, the CPU will first sample and partition this dataset into 2^H buckets of average size 2^N. Now, these 2^H buckets can stream to the FPGA to be sorted, and their results concatenated in software. Although the extra CPU processing will increase the runtime of the system, it can allow further scalability up to the host's DRAM capacity.

4 HARDWARE ARCHITECTURE

In this section, we describe the overall hardware architecture of our FPGA-accelerated samplesort system. Later, Sections 5 and 6 will provide a detailed look at the internals of the partitioners and the sorter subsystem, respectively.

Figure 2 presents an overview of our design. At the top of the diagram, the CPU communicates with the FPGA through PCIe. The CPU configures a hardware DMA module to stream data between host memory and hardware accelerators. The accelerators operate

as a pipeline, including multiple partitioners (which operate in multiple *stages*, each partitioning the data into an increasing number of buckets) and a sorter (which will sort each bucket). Input records are streamed into the first partitioner; from there, they will pass through the following partitioner cores and the sorter, eventually streaming back to the host memory. The intermediate data in each stage are stored in the off-chip DRAM, while associated metadata needed for control are directly forwarded to the next stage. The dashed lines in Figure 2 show memory-mapped channels which allow direct access between the host memory/CPU and the partitioners and sorter. These channels allow partitioners to retrieve their splitters from host memory and allow the the sorter to inform the software of any oversized buckets that must be re-sorted.

Each sorting task is packaged into a data frame and streamed to the first partitioner. A data frame contains a descriptor with configuration information, the set of splitters for the first stage partitioner, and the records to be partitioned and (eventually) sorted. The first partitioner processes the data frame descriptor and forwards additional configuration data to the rest of partitioners and the sorter. After configuration, the first stage partitioner partitions the data from the input stream into 2^{P_0} buckets and stores them in off-chip memory. Each of the remaining partitioners will read the previous stage's results from DRAM, partition the bucket into finer-grained buckets, and write the resulting buckets back to the on-board memory while sending the bucket sizes to the next stage. After going through the requisite number of partitioning stages, the data set in DRAM has been partitioned into $2^P = 2^{P_0+P_1+\ldots+P_{m-1}}$ buckets.

The sorter then reads buckets from memory, sorts each one, and outputs the sorted records. The stream of sorted results is sent to the host memory by the hardware DMA system. Meanwhile, if there are buckets larger than the hardware sorter's capacity, the sorter indicates this to the host by writing to a predefined location in the host memory.

5 PARTITIONING HARDWARE

This section presents our novel hardware partitioning system. First, we describe the design of each partitioning core. Then we show how we use multiple partitioning cores to form a larger multi-stage partitioning system.

5.1 Parallel Partitioner

A partitioner takes as input a stream of records and a set of splitters that define the boundaries of the desired buckets. The partitioner then compares each record with the splitters to determine in which bucket each record should be placed. Lastly, the partitioner moves each record to a location in memory that corresponds to its bucket. The partitioning can be parallelized to increase throughput (e.g., to match the rate that records arrive via PCIe). Figure 3 illustrates a top-level view of the partitioner, which consumes w records in each clock cycle, distributing data to w parallel cores (which each consume one record per cycle).

Before the partitioner can begin receiving records, it must take in the set of splitters, which are then stored in on-chip memory in each parallel core. Input records are then received and aligned by a dispatcher that distributes them to the w parallel partitioning

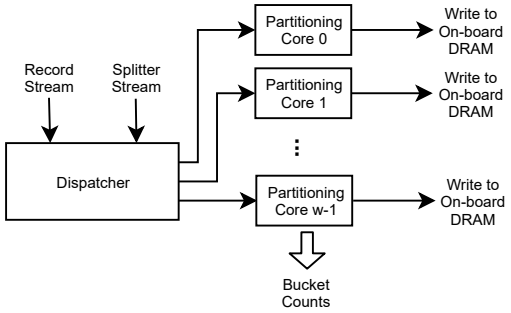

Figure 3: Overview of the partitioner.

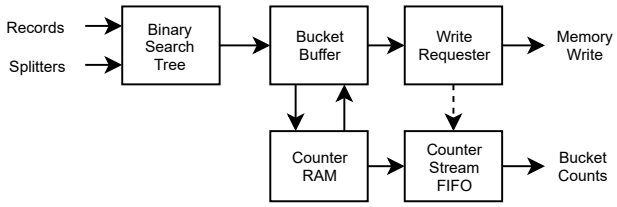

Figure 4: Partitioning core structure.

cores, each of which has its own independent internal buffer. Once all input records have been processed, the cores flush their internal pipelines and transmit their bucket counts to the next stage.

Figure 4 illustrates the design of each of the parallel partitioning cores. Each core consumes one record per cycle and will partition records into 2^P buckets. Each core includes: a binary search tree (BST) module, which is responsible for determining the correct bucket for each record; a bucket buffer, which temporarily holds values on chip until a burst write to DRAM can be performed; a write requester, which initiates the memory burst write; and a counter stream FIFO, which informs the next hardware stage of the number of records in each bucket. For clarity, Figure 4 omits control logic, which is used to communicate between modules, stall the pipeline if needed, and handle any back-pressure received on its outputs.

The BST is the partitioning core's key component. It is a pipeline that accepts and produces one record per cycle; it determines into which bucket the system should write each record. Its structure is shown in Figure 5. To partition into 2^P buckets, the BST includes P stages; each represents one level of the tree. Figure 6 shows the internals for each stage, specifically showing the m-th stage, which will hold 2^m splitter values in the Splitter RAM.

As an example, consider the first stage of the pipeline ($m = 0$), which holds a single splitter value. By comparing its input record with its splitter value, the stage will determine if the input record belongs in the top half or the bottom half of the buckets. This decision is encoded as a one bit index, which is passed (along with the unmodified record) to the next stage. Continuing the example, the Splitter RAM in the second stage ($m = 1$) holds two splitter values; it will use the one-bit index produced from the previous stage to determine which of the two splitter values to compare with. This comparison again yields one index bit, which is now concatenated with the index from the previous stage. To generalize,

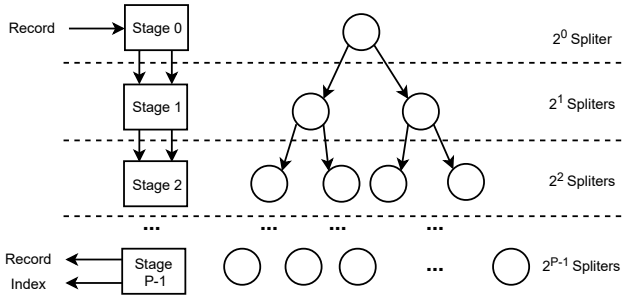

Figure 5: The structure of the multi-stage binary search tree.

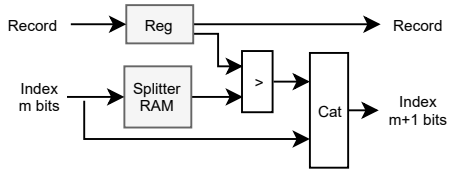

Figure 6: The mth stage in the binary search tree.

stage m uses an m-bit index to determine which of its 2^m splitters to compare with, resulting in an $m+1$ bit output index. After all P stages, the final P bit index represents which of the 2^P buckets this record belongs in.

On each cycle, the BST outputs a record and its corresponding P bit index. Naively, one could immediately write this record to the correct location in DRAM. However, this would cause many small DRAM writes, wasting precious bandwidth. Instead, the partitioning core (Figure 4) employs a bucket buffer, constructed using on-chip memories, to accumulate records until a burst write to DRAM can be performed. The buffer comprises 2^P small memories (one for each bucket). When the buffer has collected enough records for a burst write, it forwards the data to the write requester, which initiates a DRAM write. Lastly, the counter RAM tracks the bucket counts—the number of records in each bucket. At the end of processing a data frame, the partitioning core flushes the bucket buffer to the DRAMs and flushes the bucket counts to the counter stream FIFO. When the write requester determines that all DRAM writes have completed, it allows the bucket counts to be output.

As seen in Figure 3, the partitioner uses w parallel cores; to allow them to work independently, we allow each to write to its own private region of DRAM. This implies that a bucket's values are spread among w locations in DRAM. Our prototype uses $w = 4$ and four DRAM controllers, enabling one-to-one mapping. For systems where w is greater than the number of DRAM controllers, the partitioning cores require arbitration logic.

5.2 Multi-stage Partitioning

To maximize the supported data set size, it is desirable for the hardware partitioning system to partition data into as many buckets as possible. However, we observe that the partitioner's bucket buffer (as seen in Figure 4) grows linearly with the number of partitions 2^P. That is, each time we increase P by one to double the number of partitions, we also double amount of on-chip memory required by

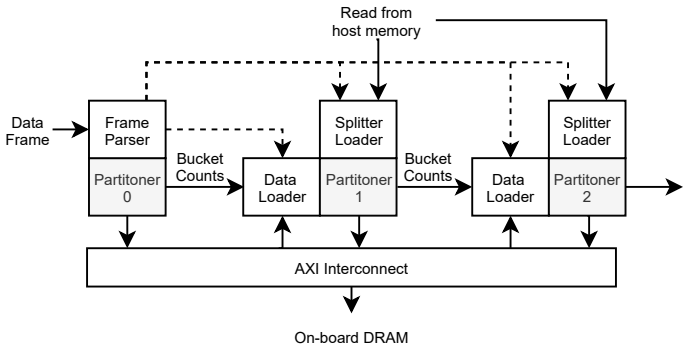

Figure 7: Multi-stage partitioning pipeline.

the partitioner. To overcome this challenge, we observe an interesting opportunity to trade off-chip DRAM bandwidth to reduce the on-chip memory cost by replacing a single large partitioner with multiple smaller ones.

For example, consider two cases: a 2^8 partitioner versus a pipeline of two 2^4 partitioners. Both will result in the same number of buckets and have the same throughput. The former will partition data in a single pass, while the latter will need an extra DRAM round-trip between stages: first it will split data into 2^4 buckets, storing each bucket in DRAM. Then, its second stage will read the buckets from DRAM and partition each into 2^4 smaller buckets, yielding a total of $2^4 \times 2^4 = 2^8$ buckets. The difference between these two cases is in the size of the bucket buffers: the 2^8 partitioner requires 16x more on-chip memory than each of the 2^4 partitioners. Overall, the two-stage partitioner requires one-eighth of the buffers of the single-stage version. However this comes at a cost: the two-stage partitioner requires 2x more bandwidth as the data must make an extra round trip on and off chip.

We generalize this idea to an m-stage partitioning system, where the first stage partitions data into 2^{P_0} buckets, the second stage partitions each of those buckets into 2^{P_1} smaller buckets, and so on. Overall, records are partitioned into $2^P = 2^{P_0+P_1+\ldots+P_{m-1}}$ buckets. As m (the number of stages) increases, the on-chip buffering costs will decrease but the DRAM bandwidth requirement grows larger, allowing the designer to choose an appropriate balance given available resources.

Figure 7 shows an example 3-stage partitioner. The data frame including configuration, splitters, and records is streamed from the host memory into a frame parser module, which configures each partitioner stage. The first stage partitioner partitions data into 2^{P_0} buckets and stores the buckets in the on-board DRAM. Then, it passes the bucket counts (which indicate the size of each bucket it just produced) directly to the next partitioner, which reads the buckets from DRAM and repeats the process.

In our prototype system, the FPGA board has four DRAM channels which (based on our measurements) can provide approximately 27GB/s bandwidth (bi-directional transfer). This is much higher than the PCIe bandwidth (approximately 8GB/s) that we use to provide records to the partitioner. This means that even with two partitioners sharing the DRAM, the partitioner's throughput is still limited by PCIe bandwidth. For this reason, our prototype uses two $P = 9$ partitioners to partition data into $2^{9+9} = 2^{18}$ buckets.

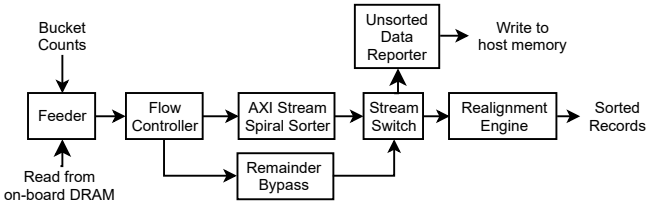

Figure 8: Sorting subsystem architecture.

6 SORTING SUBSYSTEM

After partitioning, 2^P buckets (with an average of 2^{S-1} records each) are stored in the FPGA's off-chip DRAM. Next, the sorting module reads buckets from memory and sorts them by streaming them through a parallel sorting network.

The architecture of our sorting module is shown in Figure 8. Its key component is a modified version of a streaming sorting network (SSN) from the Spiral Sorting Network IP Generator [24], which produces high-throughput sorters based on user-provided parameters.[1] Notably, the SSN has several strict input/output and timing restrictions, which we have loosened to allow the design to work more effectively as part of our larger samplesort accelerator. Specifically, our system requires flexibility in the number of elements to sort (because bucket sizes are not uniform) and it must allow the pipeline to stall due to back-pressure on its output or delays on its input. To provide this functionality, we added skid buffers into the SSN's internal pipeline to allow it to stall if presented with backpressure on its output, and surrounded the SSN with several additional modules seen in Figure 8. These modules, described in detail below, provide additional buffering, padding, and control; they allow the sorter to tolerate any I/O delays, to operate on any number of records $\leq 2^S$, and to report oversized buckets ($> 2^S$ records) to software.

An interesting implication arises from the disparity between the average bucket size (2^{S-1} records) and the maximum supported size (2^S records). Previously, we defined w as the number of records that our partitioner can produce/consume per cycle. Intuitively, the average throughput of the sorting subsystem should also consume w records per cycle to match. We observe that an SSN that natively sorts a maximum of 2^S records at $2w$ records per cycle will exhibit a throughput of w records per cycle when sorting our average bucket (2^{S-1} records). To account for this, we design the components of the sorting subsystem to process $2w$ words per cycle.

In addition to the SSN itself, this sorting subsystem consists of several additional components, which we describe below.

Feeder. The feeder module is responsible for reading buckets from the FPGA's on-board DRAM and preparing them for the SSN. First, the feeder must know the number of records stored in each bucket (*bucket counts*); this information is received from the hardware partitioner, and stored locally. The feeder then issues memory read requests, buffers them, and forms the results into a stream of $2w$ records per cycle.

Flow Controller. The Spiral SSN is designed to operate on buckets of a fixed size (2^S records); a new bucket cannot begin entering the SSN until its input has consumed all 2^S. However the dynamic nature of our system means that typically we will have fewer valid records to input to it. In this case, the flow controller is responsible for stalling the input stream until the SSN is ready to begin operating on a new bucket. To do so, the flow controller counts the number of records passing it and stalls the stream when needed. Additionally, the flow controller has another important task: if it observes an oversized bucket ($> 2^S$ records), it will output the first 2^S records to SSN, while redirecting extra records to the remainder bypass module.

[1] We will use the term "SSN" to describe the Spiral-generated streaming sorting network IP and the term "sorting subsystem" to refer to the system seen in Figure 8 that includes the SSN and other components.

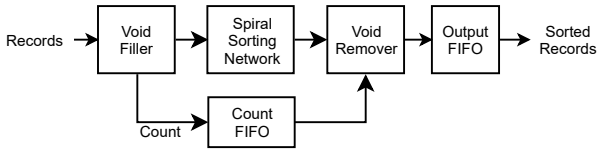

Figure 9: The structure of the AXI Stream Spiral sorter.

AXI-Stream Spiral Sorter. The AXI Spiral Sorter, shown in Figure 9, wraps the Spiral SSN with AXI stream interfaces and an output FIFO. We modified the Spiral sorting network by adding stall signals and skid buffers, which can allow the pipeline to safely stall without requiring that a single high-fanout stall signal reach every internal register.

Because the SSN requires a full set of 2^S records to operate, we have added a "void filler" module that pads the input stream with artificial records, which will be removed from its output using a "void remover" module.

Other Sorting Subsystem Modules. If a bucket's size is larger than 2^S, the remainder bypass module will be used to allow the overflowing words to bypass the sorter; they are stored in a FIFO. The Stream Switch can then append these unsorted records at the end of the sorted portion of the bucket. Such oversized buckets will be handled by the Unsorted Data Reporter, which writes the start and end indices of the unsorted sequence into host memory, allowing the application software to re-sort those rare buckets that hardware could not completely sort. Lastly, the realignment engine concatenates the sorted sequences of each bucket into a single AXI stream and converts them to the bus width used by the output AXI stream.

7 IMPLEMENTATION

In this section, we describe the hardware and software implementation of our prototype system.

7.1 Hardware Implementation

To evaluate our design, we created a prototype targeting the Amazon AWS F1 [1] instance which uses a Xilinx Virtex UltraScale+ FPGA (xcvu9p-flgb2104-2-i). This system pairs the FPGA with a private 64 GB DDR4 memory (split across four channels) and a dedicated PCIe gen3 x16 interface to the host. For implementation, we used Xilinx Vivado version 2017.4.

Our prototype, which runs at 250MHz, uses two-stage hardware partitioning (each stage partitioning data into 2^9 buckets, i.e., $P = 18$) and an $S = 13$ sorter; therefore it can sort sequences up to $2^N = 2^{P+S-1} = 2^{30}$ records. Each record is a (record-key, record-index)

Table 1: Resource utilization.

Name	LUT	Flip Flop	BRAM	URAM
Amazon AWS Shell	221,727	296,609	303.5	43
DMA and Interconnect	79,444	100,422	18.5	0
Partitioner	95,628	72,159	188	128
Sorter	232,738	319,741	571.5	29
Total utilization	629,537	788,931	1,081.5	200
Available	1,181,768	2,363,536	2,160	960
Percentage	53.3%	33.4%	50.1%	20.8%

pair with a 10-byte key and 4-byte index. (Records are sorted based on their keys.) The system's parallelism is set to $w = 4$ records; at 250MHz this allows the prototype to approach the system's upper bound on PCIe bandwidth (as we will demonstrate in Section 8).

We implemented all modules in the partitioner and sorter systems as hand-written Verilog, except the Spiral streaming sorting network, which was generated by Spiral [24] and modified as explained in Section 6. The AWS Shell provides AXI4 memory mapped interfaces for accessing four DRAM controllers and host memory. It includes hardware logic for the DRAM controllers and the PCIe endpoint. The DMA and interconnect are implemented with Xilinx standard IP: AXI DMA v7.1 and AXI SmartConnect v1.0.

Table 1 shows the resource utilization of the prototype's hardware components. We note that the partitioner and sorter collectively consume a relatively small percentage of the FPGA's overall area (27.8% LUTs, 16.58% registers, 35.16% BRAM and 16.35% URAM). Even with other components like the AWS shell and the interconnect, there is still significant room to increase parallelism (i.e., increase w) or scale the partitioner and sorter to larger sizes (i.e., increase S and/or P). However, our experimental results (in Section 8 below) show that the system already saturates PCIe bandwidth, meaning increased hardware parallelism will not improve overall performance. Furthermore, the hardware system's current capacity of $2^N = 2^{30}$ is already the maximum that will fit within the board's 64GB of DRAM; although we have the FPGA logic available to increase N, it would not be possible without increasing the DRAM size.

7.2 Software Implementation

In addition to the hardware design, our prototype includes all necessary software components to interact with the FPGA design and perform the portions of the algorithm allocated to the CPU.

First, a user-space application performs the functions previously described in Algorithm 2: sampling data, generating splitters needed for partitioning, performing software partitioning (only for datasets $> 2^{30}$ records), and correcting any oversized buckets. The software also manages memory allocation for the data transfers between host memory and the FPGA, and makes `ioctl()` system calls to the kernel driver (described below) to perform DMA transfers.

Next, we use a hardware driver to manage data transfer between host memory and FPGA board, and to provide an application programming interface for the user to issue sorting tasks. The hardware driver controls the FPGA's on-chip hardware DMA module; its registers are mapped into the kernel space, and the driver translates the virtual addresses of pointers to physical addresses, which can be then used by the hardware DMA. The software sends commands to the FPGA's DMA engine via PCIe to initiate transfers between host memory and the FPGA. Furthermore, the kernel driver manages the transfers, tracking the amount of data sent and received between the host and FPGA.

Table 2: Amazon AWS F1 instances. "vCPU" refers to the number of available CPU threads.

Name	FPGAs	vCPU	RAM (GB)	Price ($/hr)
f1.2xlarge	1	8	122	1.650
f1.16xlarge	8	64	976	13.200

To reduce overhead, the driver uses a zero-copy mechanism, which directly sets up DMA transfers between user buffers and the hardware device. This removes the CPU overhead of copying data between kernel space and user space and lets the user set up the FPGA DMA transfer through `ioctl()` calls. The user-space code does this by passing the pointer and length of a data array to the kernel. The driver then looks up the page tables related to the pointer and finds the physical addresses of all of the pages related to this data array. Lastly the driver sets up the DMA transfer using these physical addresses.

The zero-copy method requires that data should not be modified or swapped out during the DMA transfer. This can be done by locking the data in the memory or storing the data in "huge pages," which are pre-allocated, persistent, and will not be swapped under memory pressure. In our system, we use 2 MB huge pages to reduce the number of DMA descriptors needed to transfer data.

8 EVALUATION

In this section, we evaluate our FPGA-accelerated samplesort system in terms of its latency and throughput. First, Section 8.1 describes the experimental setup. Then, Sections 8.2 and 8.3 evaluate the system's latency and throughput, respectively. Because no prior work on FPGA sorting is able to operate on datasets of this size at high throughput, we benchmark against parallel software sorters running on multi-threaded CPUs. Lastly, Section 8.4 shows that our single-FPGA prototype even outperforms a highly parallel CPU system with 64 threads.

8.1 Experimental Setup

To evaluate our design, we characterize our prototype on the Amazon AWS F1 [1] FPGA instance *f1.2xlarge*, which is described in Table 2. Amazon also provides a larger system *f1.16xlarge*, which we use in some comparisons only for its larger host RAM capacity and its greater number of CPU threads. The *f1.16xlarge* system includes eight FPGA boards (each identical to the FPGA board in the *f1.2xlarge*, as described in Section 7.1), but all reported results only require the use of a single FPGA.

The data sets used for our experiments are generated by the `gensort` record generator from the Datamation sorting benchmark [7], where each record is 100 bytes with a 10 byte key and 90 byte value. To save memory and PCIe bandwidth, we use a common technique [3] that translates records into (record-key, record-index) pairs with a 4-byte index, which points to the location of the record

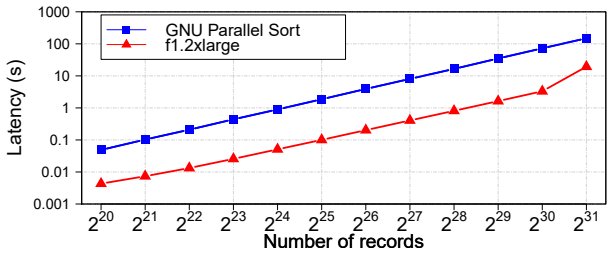

Figure 10: The latency of sorting a single task.

in the input file. We generate data sets of size 2^{20} to 2^{31} records, approximately 14 MB to 28 GB of data, which is limited by the host memory capacity of the *f1.2xlarge* instance. For all experiments, we measure the time elapsed when sorting memory-resident key-index pairs.

To the best of our knowledge, there are no existing high throughput hardware designs that can process datasets of this magnitude, making direct comparison with other FPGA designs impossible. Instead, we benchmark against GNU parallel sort, a widely-adopted parallel software sorting program, running on state-of-the-art CPUs.

8.2 Latency of Sorting a Single Task

First, we evaluate our system's performance by measuring the time (latency) of sorting a given number of records, and we compare this with the latency of GNU parallel sort. We measure the latency as the elapsed time for a single sorting task, which includes sampling and preparing splitters in software, transmission of the data between the host and the FPGA, partitioning and sorting on the FPGA, transmission of the data back to the host, and the final software post-processing step to fix any oversized buckets. Our system can sort sequences up to 2^{30} records entirely in hardware (that is, without additional software partitioning); above this size, the dataset cannot fit into the FPGA's DRAM, requiring the CPU to perform an initial partitioning step as described in Section 3.3 to break the dataset into smaller buckets.

Figure 10 and Table 3 show the latency of sorting 2^{20} through 2^{31} records with our FPGA-accelerated system (*f1.2xlarge*) and with our software benchmark (GNU Parallel Sort) running on the 8-thread CPU, and Figure 11 shows the speedup of our system relative to software. We observe an 11.3x to 21.9x performance increase when the data set can be sorted entirely by the FPGA ($\leq 2^{30}$ records), and a 7.7x increase for 2^{31} records (where the host CPU must perform an initial partitioning step before the FPGA can begin).

Figure 12 shows a breakdown of how our system's work is divided between the CPU and the FPGA. This breakdown explains the observed latency trends. At 2^{20} records, the CPU takes 33% of the total runtime for memory allocation, sampling and generating the splitters. As the data set grows, the percentage of time consumed by the CPU decreases. We observe this is caused by a nearly constant time for allocating memory. During memory allocation, the CPU allocates memory for splitters and makes a system call to the kernel driver to obtain the physical address of this memory. Most of this time is spent on the context switch between user space and kernel space. Therefore, the portion of time consumed by the CPU

Table 3: Comparison of the latency of sorting a single task using GNU sort versus FPGA-accelerated samplesort.

Number of records	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
GNU sort time (s)	0.049	0.10	0.21	0.44	0.89	1.86
FPGA sort time (s)	0.0043	0.0074	0.013	0.026	0.051	0.10
Speedup	11.3x	13.9x	15.7	17.1	17.6	18.5
Number of records	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}
GNU sort time (s)	3.90	8.05	16.51	31.91	72.52	149.25
FPGA sort time (s)	0.20	0.41	0.81	1.63	3.31	19.36
Speedup	19.3x	19.8x	20.3x	21.4x	21.9x	7.7x

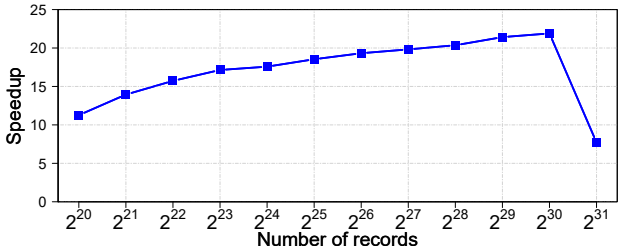

Figure 11: Speedup of FPGA-accelerated samplesort relative to GNU Sort for sorting a single task

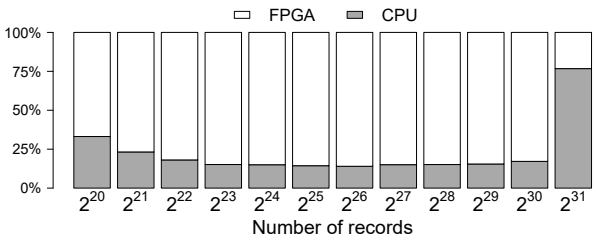

Figure 12: The proportion of runtime consumed by the FPGA and CPU when sorting a single task of the given size.

drops to approximately 15% of the total time (dominated by the time for generating splitters). However, when the data set grows larger than 2^{30}, the CPU must perform one additional step of partitioning, leading the CPU to consume 76% of the total runtime.

We also observe that the CPU takes a small amount of time for generating splitters and post-processing oversized buckets (those that are larger than 2^S records and therefore cannot be completely sorted in hardware). When sorting 2^{30} records, generating splitters takes 0.5s (16.9% of total time), while post-processing consumes only 0.0019s (0.05% of total time). Only one out of 2^{18} total buckets is oversized, yielding probability $\sim 4 \times 10^{-6}$.

8.3 Throughput of Sorting a Batch of Tasks

Our system's partitioners and sorter are fully pipelined, allowing them to concurrently process multiple independent sorting tasks, yielding increased throughput when sorting a batch of tasks. In this scenario, the first-stage partitioner can operate on one task

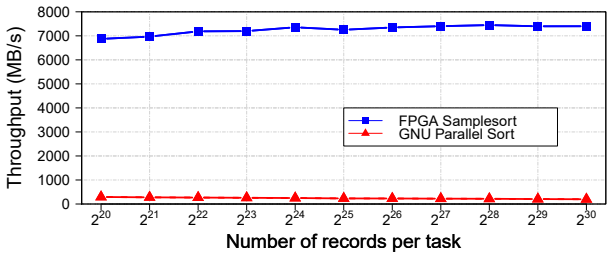

Figure 13: Average throughput of sorting 16 tasks.

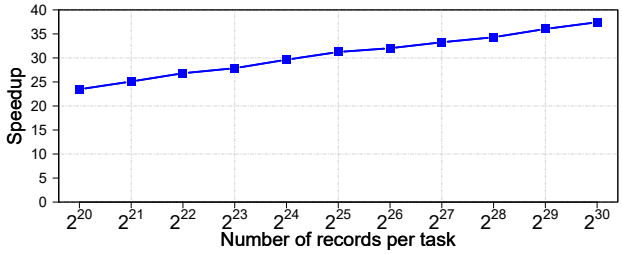

Figure 14: Speedup of FPGA-accelerated samplesort relative to GNU Sort for sorting 16 tasks.

while the second-stage partitioner and sorters can operate on another. Additionally, the CPU and FPGA runtime can overlapped, improving throughput further.

To evaluate the system's throughput, we measured the total time consumed when sorting 16 independent sorting tasks. However, this scenario also requires 16 times more RAM on the host CPU than a single task would require; the *f1.2xlarge* AWS instance only has enough RAM to run this test on tasks of up to 2^{26} records. Therefore, we used the larger AWS instance *f1.16xlarge*, whose host has 976GB RAM (8x higher than *f1.2xlarge*) and identical FPGA hardware.[2] The larger RAM in *f1.16xlarge* is sufficient to test a batch of 16 tasks of up to 2^{30} records each.

Figure 13 shows the measured throughput for our system and for GNU sort running on 8 threads, and Figure 14 shows the corresponding speedup. We observe that the FPGA system's throughput varies only slightly with task size (on average 7.2 GB/s, effectively saturating the system's PCIe bandwidth) indicating that the throughput is limited by data transfer time over PCIe. The FPGA's speedup grows as the number of records increases, culminating in a 37.4x speedup when sorting tasks of 2^{30} records.

8.4 Comparison to Highly-Parallel CPU System

The performance comparison in Section 8.2 was performed using the CPU and FPGA resources in Amazon's *f1.2xlarge* instance, whose CPU supports eight parallel threads. As a final confirmation of the benefits of our accelerator, we compare our system to a much more extreme test: a larger instance (*f1.16xlarge*) with 64 CPU threads. The higher parallelism available from the larger number of CPU cores substantially increases the speed of the software sort implementation. We measured the performance of GNU Parallel Sort for the data set from 2^{20} to 2^{30} records on the large instance using 64 threads.

[2] Although *f1.16xlarge* instances include 64 CPU threads and eight FPGA boards, for consistency, we use only 8 threads and one FPGA in this comparison. Where possible (2^{20} through 2^{26} records), we ran experiments on both instances and ensured that differences were negligible.

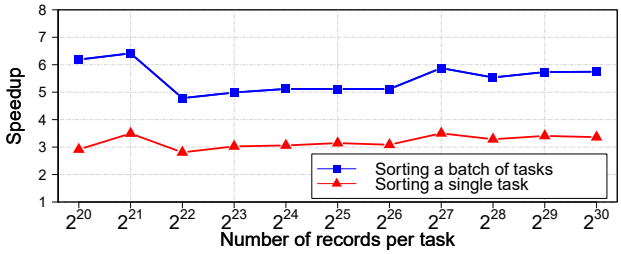

Figure 15: Speedup of FPGA-accelerated samplesort relative GNU sort running on 64 threads.

Figure 15 shows the speedup of our FPGA-accelerated samplesort system relative to GNU Parallel Sort using 64 threads. We observe that our single-FPGA system still can achieve considerable improvement even relative to this massively more powerful CPU system: 5.7x higher throughput and 3.4x lower latency (both at 2^{30} records).

9 CONCLUSIONS

Although prior work has shown that FPGAs can efficiently sort a data set that fits in on-chip memory, existing approaches exhibit a lack of scalability to large data sizes, typically limited by the time and memory bandwidth required for merging sorted subsequences.

In this work, we proposed a new technique for using an FPGA to accelerate the samplesort algorithm. Our method uses a novel multi-stage hardware partitioner combined with a streaming sorter to provide a high throughput solution for sorting large data sets. In our system, hardware in the FPGA cooperates with software running on the host CPU to enable scalability and efficient sorting.

We implemented and tested a prototype of our sorting system on an Amazon AWS F1 FPGA instance with a Xilinx UltraScale+ VU9P FPGA. Our prototype demonstrates an average throughput of 7.2 GB/s when sorting tasks of 2^{30} records, effectively saturating the PCIe bandwidth of the system. No prior research has explored sorting data sets of this magnitude with high throughput hardware. Compared to multi-threaded software (GNU parallel sort on 8 threads), our system demonstrates a 21.9x improvement in latency (sorting 2^{30} records) and a 37.4x throughput improvement when sorting a batch of tasks (2^{30} records per task).

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation through research grants CCF-#1452904 and CCF-#1725543. We would also like to thank Amazon Web Services, Inc for providing AWS credits to run the experiments on the F1 instances.

REFERENCES

[1] Amazon. [n.d.]. Amazon EC2 F1 Instances. Retrieved September 15, 2019 from https://aws.amazon.com/ec2/instance-types/f1/

[2] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. 2015. Practical Massively Parallel Sorting. In *Proceedings of the 27th ACM on Symposium*

on Parallelism in Algorithms and Architectures (SPAA '15). ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/2755573.2755595

[3] D. A. Bell. 1958. The Principles of Sorting. *Comput. J.* 1, 2 (Jan. 1958), 71–77. https://doi.org/10.1093/comjnl/1.2.71

[4] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/113379.113380

[5] Coenraad Bron. 1972. Merge Sort Algorithm [M1]. *Commun. ACM* 15, 5 (May 1972), 357–358. https://doi.org/10.1145/355602.361317

[6] David J. DeWitt, Je Key F. Naughton, and Donovan A. Schneider. 1991. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. IEEE, 280–291. https://doi.org/10.1109/PDIS.1991.183115

[7] Anon et al. 1985. A Measure of Transaction Processing Power. *Datamation* 31, 7 (April 1985), 112–118. http://dl.acm.org/citation.cfm?id=13900.18159

[8] William D. Frazer and Archie C. McKellar. 1970. Samplesort: A Sampling Approach to Minimal Storage Tree Sorting. *J. ACM* 17, 3 (July 1970), 496–507. https://doi.org/10.1145/321592.321600

[9] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees *(CIDR '03)*.

[10] Goetz Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3, Article 10 (Sept. 2006). https://doi.org/10.1145/1132960.1132964

[11] Charles A.R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961), 321–. https://doi.org/10.1145/366622.366644

[12] Charles A.R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (Jan. 1962), 10–16. https://doi.org/10.1093/comjnl/5.1.10

[13] Donald E. Knuth. 1998. *The Art of Computer Programming, Vol. 3: Sorting and Searching (2nd. ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[14] Dirk Koch and Jim Torresen. 2011. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/1950413.1950427

[15] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. 2010. GPU sample sort. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS) (IPDPS '10)*. IEEE, 1–10. https://doi.org/10.1109/IPDPS.2010.5470444

[16] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. 2017. High-Performance Hardware Merge Sorter. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (FCCM '17)*. IEEE, 1–8. https://doi.org/10.1109/FCCM.2017.19

[17] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. 2018. FLiMS: Fast Lightweight Merge Sorter. In *2018 International Conference on Field-Programmable Technology (FPT) (FPT '18)*. IEEE, 78–85. https://doi.org/10.1109/FPT.2018.00022

[18] Makoto Saitoh, Elsayed A. Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. 2018. A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (FCCM '18)*. IEEE, 197–204. https://doi.org/10.1109/FCCM.2018.00038

[19] Makoto Saitoh and Kenji Kise. 2018. Very Massive Hardware Merge Sorter. In *2018 International Conference on Field-Programmable Technology (FPT) (FPT '18)*. IEEE, 86–93. https://doi.org/10.1109/FPT.2018.00023

[20] Edgar Solomonik and Laxmikant V. Kalè. 2010. Highly scalable parallel sorting. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS) (IPDPS '10)*. IEEE, 1–12. https://doi.org/10.1109/IPDPS.2010.5470406

[21] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. 2016. Parallel hardware merge sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (FCCM '16)*. IEEE, 95–102. https://doi.org/10.1109/FCCM.2016.34

[22] Takuma Usui, Thiem Van Chu, and Kenji Kise. 2016. A Cost-Effective and Scalable Merge Sorter Tree on FPGAs. In *2016 Fourth International Symposium on Computing and Networking (CANDAR) (CANDAR '16)*. IEEE, 47–56. https://doi.org/10.1109/CANDAR.2016.0023

[23] Chi Zhang, Ren Chen, and Viktor Prasanna. 2016. High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (IPDPSW '16)*. IEEE, 148–155. https://doi.org/10.1109/IPDPSW.2016.117

[24] Marcela Zuluaga, Peter Milder, and Markus Püschel. 2016. Streaming Sorting Networks. *ACM Trans. Des. Autom. Electron. Syst.* 21, 4, Article 55 (May 2016), 30 pages. https://doi.org/10.1145/2854150