TopSort: A High-Performance Two-Phase Sorting Accelerator Optimized on HBM-based FPGAs

Weikang Qiao, *Student Member, IEEE,* Licheng Guo, Zhenman Fang, *Member, IEEE,* Mau-Chung Frank Chang, *Life Fellow, IEEE* and Jason Cong, *Fellow, IEEE*

Abstract—The emergence of high-bandwidth memory (HBM) brings new opportunities to boost the performance of sorting acceleration on FPGAs, which was conventionally bounded by the available off-chip memory bandwidth. However, it is nontrivial for designers to fully utilize this immense bandwidth. First, the existing sorter designs cannot be directly scaled at the increasing rate of available off-chip bandwidth, as the required on-chip resource usage grows at a much faster rate and would bound the sorting performance in turn. Second, designers need an in-depth understanding of HBM's characteristics to effectively utilize the HBM bandwidth. To tackle these challenges, we present TopSort, a novel two-phase sorting solution optimized for HBM-based FPGAs. In the first phase, 16 merge trees work in parallel to fully utilize 32 HBM channels' bandwidth. In the second phase, TopSort reuses the logic from phase one to form a wider merge tree to merge the partially sorted results from phase one. TopSort also adopts HBM-specific optimizations to reduce resource overhead and improve bandwidth utilization. TopSort can sort up to 4 GB data using all 32 HBM channels, with an overall sorting performance of 15.6 GB/s. TopSort is 6.7× and 2.7× faster than state-of-the-art CPU and FPGA sorters.

Index Terms—Sorting, merge sort, hardware acceleration, high-bandwidth memory, memory-centric design, FPGA, floorplan.

1 Introduction

Sorting is one of the fundamental computation kernels in many big data applications and there have been continuous efforts on designing high-performance sorting accelerators on FPGAs [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Most of these sorting accelerators are based on the multi-way merge tree sort algorithm [1], [2], [3], [4], [5], [6], [7], [8], [9], due to its massive data parallelism and regular memory access patterns. Before the maturing of the high-bandwidth memory (HBM) technology, these sorters were usually implemented on DRAM-based FPGAs and bounded by the off-chip memory bandwidth [7], [8]. For example, Bonsai [7] presented the state-of-the-art merge tree based sorter on the AWS F1 datacenter FPGA and achieved an overall sorting performance of 5.8 GB/s with a merge tree's throughput of 32 GB/s;[1] it could sufficiently scale up the number of parallel merge units until the memory bandwidth becomes the bottleneck.

The emergence of HBM-based FPGAs has brought the potential to further boost the performance of the sorting accelerators by offering a much higher off-chip memory bandwidth. For example, the recent HBM-based datacenter FPGAs report a peak memory bandwidth of about 420 GB/s, which is roughly 6× of that on a similar DRAM-based datacenter FPGA [16], [17]. A natural question is: if we simply port and scale the state-of-the-art merge tree designs onto an HBM-based FPGA, can we get 6× higher merge tree throughput? Unfortunately, the answer is no due to two nontrivial challenges.

On the one hand, with the tremendous off-chip bandwidth increase, the bottleneck of sorting acceleration would shift from off-chip memory to the available on-chip resources. As will be analyzed in Section 3, for a merge tree-based sorter that can output p elements per cycle, the required off-chip bandwidth increases linearly with p, while the required on-chip resources increase much faster at the rate of $\theta(plog^3(p))$. Unfortunately, for an HBM-based FPGA (e.g., Xilinx Alveo U280 FPGA [18]) that provides roughly 6× more bandwidth than a similar DRAM-based FPGA (e.g., AWS F1 FPGA), it only provides merely 1.2× more on-chip resources. Indeed, our analysis shows that a single merge tree accelerator can not be scaled to use more than $1/4$ of the available HBM bandwidth.

On the other hand, it is nontrivial to fully utilize the HBM bandwidth in an accelerator design. Typically, the HBM is composed of 32 small channels. The accelerator has to rely on multiple memory controllers to access those HBM channels in parallel to maximize the memory bandwidth, at the expense of on-chip resources. Moreover, there could easily be contention between multiple channel accesses due to HBM's internal channel switching, which would degrade the effective bandwidth. Therefore, it requires delicate memory access control to improve the bandwidth utilization and reduce resource overhead. Finally, the HBM stacks are physically connected to a huge multi-die FPGA's bottom die only, making it difficult to spread the resource utilization across multiple FPGA dies to achieve desirable timing closure. To the best of our knowledge, none of the published HBM-

- *Weikang Qiao and Mau-Chung Frank Chang are with the Department of Electrical and Computer Engineering, University of California, Los Angeles, CA, 90025.*
  E-mail: wkqiao2015@ucla.edu, mfchang@ee.ucla.edu
- *Licheng Guo and Jason Cong are with the Department of Computer Science, University of California, Los Angeles, CA, 90025.*
  E-mail: {lcguo,cong}@cs.ucla.edu
- *Zhenman Fang is with the School of Engineering Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada.*
  E-mail: zhenman@sfu.ca

1. The definitions of a merge tree's throughput and the overall sorting performance are presented in Section 2 and Section 3.

based accelerator designs [19], [20], [21], [22], [23] is able to fully utilize the entire bandwidth of the 32 HBM channels.

In this work, we present TopSort, a high-performance two-phase sorting accelerator specialized for HBM-based FPGAs. TopSort avoids the excessive resource consumption of directly scaling a single giant sorter's throughput by novelly splitting the complete sorting process into two separate merge phases with smaller sorters and reusing the resources between the two phases. In the first phase, TopSort employs 16 parallel small merge tree kernels, each of which sorts a portion of the input sequence with two HBM channels (one for reading unsorted inputs and the other for writing sorted outputs). In this phase, the effective merge tree throughput is equal to the bandwidth of 16 HBM channels. In the second phase, TopSort merges the sorted results from all HBM channels into one final sorted sequence. To reduce the resource consumption, this phase reuses 4 merge tree kernels from phase one to form a wider merge tree with a 4× higher throughput, since merge trees with different throughput share a similar operation pattern.

To improve the effective HBM bandwidth utilization, we profile the corresponding HBM channel characteristics and carefully optimize the merge tree's memory access pattern for each phase, including optimizing its data layout and burst access, as well as reducing the usage of HBM controllers. Besides, the novel merge tree reuse architecture allows us to easily improve the design frequency through coarse-grained floorplanning of each separate merge tree kernel. We floorplan TopSort by evenly distributing the design across the entire FPGA logic regions, based on an efficient resource model that considers the accelerator's design complexity, the available resources of each FPGA die, the limitation of cross-die signals, and the overhead of HBM controllers.

When implemented on the Xilinx Alveo U280 FPGA, TopSort runs at 214 MHz and achieves an overall performance of 15.6 GB/s, which is 6.7× and 2.7× faster than state-of-the-art CPU and FPGA sorters [7], [24]. TopSort can sort up to 4 GB data at a time, which is half of the total HBM capacity. Although this work includes a number of device-specific optimization for the HBM-based U280 FPGAs from Xilinx, the general two-phase reused-based merge sort architecture is applicable to all HBM-based FPGAs across different vendors.

We summarize the contributions of TopSort as below:

1. A novel two-phase merge tree based sorter optimized on HBM-based FPGAs, which fully utilizes the HBM bandwidth and alleviates the on-chip resource bottleneck by intelligently reusing multiple merge trees between two phases.
2. Techniques and insights for HBM-specific optimizations, including the data layout, burst access, and HBM controller optimizations, as well as the floorplanning strategy.
3. Experimental results that demonstrate the superior 15.6 GB/s sorting performance and show TopSort is 6.7× and 2.7× faster than state-of-the-art CPU and FPGA sorters.

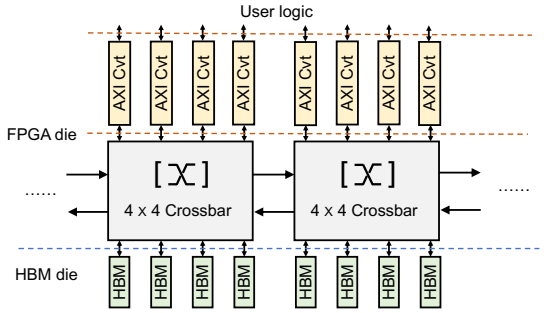


Fig. 1: Connections from user logic to HBM channels. AXI Cvt is short for AXI Rate Converter. Lateral connections between nearby crossbars enable AXI to access HBM channels that are not located in the same group.

2 Background Review

The merge tree sorting algorithm is favored for FPGA-based sorters due to its massive data parallelism, less control overhead and regular memory access patterns [1], [2], [3], [4], [5], [6], [7], [8], [9]. In this section, we first introduce the HBM-based FPGAs. Then we give an overview of the hardware merge units and the existing DRAM-based merge tree sorting accelerators.

2.1 HBM-Based FPGAs

HBM achieves higher bandwidth than DDR4 DRAMs by stacking multiple small DRAM dies together and is one of the most promising candidates enabling memory-centric designs [25]. Taking Xilinx U280 board as an example, the board is equipped with 2 HBM stacks and each stack contains 16 pseudo channels [18]. Figure 1 exhibits the connections between the user logic and the HBM channels. Each HBM channel can be accessed through a 256-bit wide AXI interface running at 450 MHz. The vendor tool will by default implement AXI rate converters in the HBM Memory Subsystem (HMSS) to adapt the original 256-bit AXI interfaces to 512-bit AXI interfaces running at 225 MHz to the user logic. In the case of routing congestion which happens if the on-chip resources are over-utilized or the design is not well pipelined, both the HBM-side AXI frequency and the user-side design frequency will be reduced and the available HBM bandwidth will be degraded.

There is no global crossbar to allow 32 AXI interfaces to access the 32 HBM channels at the same time. Instead, the 32 HBM channels are physically bundled into 8 groups and each group contains 4 adjacent channels joined by a built-in 4×4 crossbar, as is shown in Figure 1. The crossbar provides full connectivity within the group. Meanwhile, each AXI interface at the user side can still access any HBM channels outside its group. The data will sequentially traverse through each of the lateral connections until it reaches the crossbar connecting to the target channel.

2.2 Hardware Merge Unit

A hardware merge unit takes two sorted sequences of elements as its inputs and then merges them into one sorted sequence. Specifically, an E-rate hardware merge unit takes E input elements from its two E-element wide inputs and

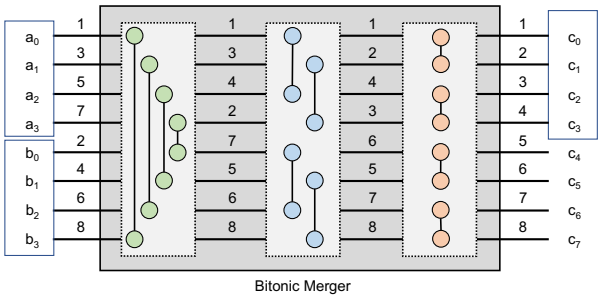


Fig. 2: An example of 4-rate bitonic merge unit: each vertical line that connects two dots is a compare-swap cell and the compare-swap cells in the same box are processed in the same cycle. c_{0-3} will be the outputs after 3 cycles.

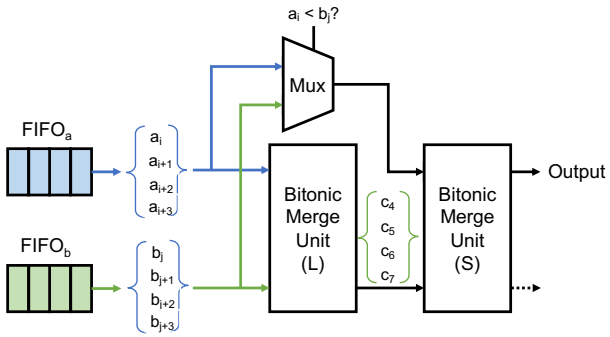


Fig. 3: The topology of a 4-rate MMS merge unit. Registers in the datapath are omitted for simplicity.

outputs E sorted elements every cycle. The *compare-swap cell* is the basic building block for hardware merge unit, which compares two elements' values and swaps them into the correct ordering [1]. A compare-swap cell usually contains a comparator and a 2-input multiplexer, which is suitable to be implemented using the Look-Up Tables (LUTs) on the FPGAs. Using a pipeline of multiple compare-swap cells, designers can develop the hardware merge unit. The bitonic merge unit shown in Figure 2 is one of the most widely used hardware merge units.

If the two sorted input sequences are longer than E, the E-rate merge unit has to be time multiplexed (i.e., executed multiple rounds) with extra control signals to ensure the outputs are in order. For example, in Figure 2, 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 side of the same bitonic merge unit and merged with either a_{4-7} or b_{4-7} to create the second 4-element sorted outputs. The feedback paths from the output side of the bitonic merge unit to its input side mean the merge unit has to wait for extra cycles before it can do the next merge operation. In other words, the initial interval (II) is equal to the number of pipeline stages in the bitonic merge unit.

To improve the merging performance with an II equal to 1, several optimization techniques have been proposed [11], [12], [13], [14]. [12] proposes an E-rate streaming merge unit called *MMS* that outputs E elements every cycle using two bitonic merge units, as shown in Figure 3. The intuition is

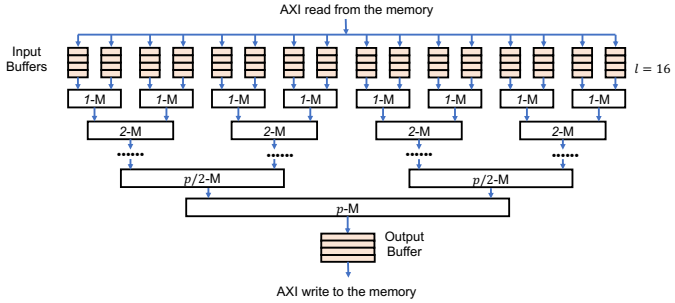


Fig. 4: An example of the merge tree ($p = 16$, $l = 16$), where an E-M box denotes an E-rate hardware merge unit. Note that l can be larger than p, e.g., to make a merge tree ($p = 16$, $l = 32$), one can have another layer of 32 1-M merge units on top of the current leaf layer.

that the larger half outputs of the original bitonic merge unit can be first calculated through the bitonic merge unit (L) and are later fed into the bitonic merge unit (s) with delayed inputs from either sequence a or b. In this work, we use the same topology of MMS to construct the hardware merge unit, as it is free of feedback paths and thus helps alleviate the design routing congestion. However, we may use other efficient merge units as well, such as FLiMSj from [26].

2.3 DRAM-based Merge Tree Sorting Accelerator

Using a combination of various hardware merge units with different rates, we can build a complete binary tree that consumes l unsorted input sequences concurrently at its leaves and outputs p sorted elements at its root every cycle [4], [6], [7], [8]. Figure 4 shows the architecture of a merge tree [7]. Such a merge tree can be uniquely denoted by (p, l), where p refers to the **merge tree's throughput** and l is **the number of leaves**.

In one sorting *pass*, the input elements are streamed from the off-chip memory into the leaf buffers and then through the merge tree, and the output elements are streamed back into the off-chip memory. Assume there are N unsorted elements initially stored in the off-chip memory, these N elements need to be streamed into the merge tree for multiple passes and after each pass, the size of the partially sorted sequence grows exactly by l times. During the first pass, the merge tree reads these N sub sequences (each containing one element) from the off-chip memory, merges them into N/l sorted sub sequences, each containing l sorted elements, and writes them back to the off-chip memory. In the next pass, the l-element sorted sub sequences are fed into the merge tree again to get N/l^2 sorted sequences of length l^2. These steps are repeated until the last pass, where l sorted sequences of length N/l are processed by the merge tree to form a complete N-element sorted sequence. One can easily derive that the total number of passes is $\lceil \log_\ell N \rceil$.

3 Scalability Analysis of a Single Merge Tree

In this section, we explain why the single merge tree employed on existing DRAM-based FPGAs cannot be directly scaled to efficiently work on HBM-based FPGAs.

The throughput of a single merge tree can be derived using the analytical model from [7]. First, the merge tree with l leaves takes $\lceil \log_\ell N \rceil$ passes. Second, the merge tree outputs p elements every cycle to the off-chip memory, where p is directly reflected by the off-chip memory bandwidth β_{memory} allocated to the writing operations. Generally, half of the system bandwidth is allocated for writing and the other half is for reading. We define the **overall sorting performance** as the number of bytes of unsorted elements divided by the time it takes to get the completely sorted elements. Assuming there are unlimited on-chip resources, we have the overall sorting performance $\beta_{overall}$ in Equation 1.

$$\beta_{overall} = \frac{\beta_{memory}}{\lceil \log_\ell N \rceil} \tag{1}$$

Clearly, the on-chip resources are finite. Next, we analyze the on-chip resource requirement in scaling the merge tree-based sorter. The merge tree in Figure 4 can be viewed as two sub merge trees whose root rates are $p/2$ each, plus a merge unit whose rate is p. Since a merge unit is a variation of a parallel sorting network, such as bitonic or even-odd sorting network, and such a p-rate merge unit consumes $plog^2(p)$ number of comparison operators [11], [12], the number of comparators, $L(p)$, required when scaling p can be summarized in Equation 2.

$$L(p) = 2L(p/2) + \theta(plog^2(p)) \tag{2}$$

Based on this equation, we can derive that the total number of logic elements (i.e., comparators) for the complete binary tree is $\theta(plog^3(p))$ [27]. Considering that migrating from 4 DRAM channels in [7] to 32 HBM channels increases the available off-chip bandwidth by nearly 8×, it is not possible to directly scale a single merge tree's throughput to catch up with the increase of the HBM bandwidth.

In fact, we find that the root throughput of a single merge tree can only be scaled to match 4 HBM channels (for write operations) on the Xilinx Alveo U280 board, due to the on-chip resource limitation and the routing congestion issue shown in Section 6.7. Since the read operations of the tree also occupies the same amount of HBM bandwidth, the bandwidth utilized in such a merge tree is at most equal to 8 HBM channels. In other words, such a merge tree only uses 25% of the HBM bandwidth in each pass.

4 TopSort Methodology & Architecture

In this section, we present the methodology of TopSort. First, we illustrate the idea of the two-phase sorting. Second, we show how TopSort reuses the logic between two phases to alleviate the resource contention and improve the performance.

4.1 Two Phases in TopSort

Since the required resources grow super-linearly when directly scaling a single merge tree, TopSort chooses to split the N unsorted elements evenly into k parts and have k smaller merge trees work in parallel. Each merge tree sorts N/k elements and is configured to have a throughput that saturates the bandwidth of a single HBM channel. This method has better scalability as its resource consumption grows linearly with the number of trees k. However, the k sorted sequences still need to be merged into the final sorted sequence. This can be done by streaming the k sequences into another merge tree for the final merge. We refer to the process during which k merge trees work in parallel as phase 1, and the subsequent final merge as phase 2.

If each merge tree in phase 1 has l_{phase1} number of leaves and a single HBM channel's bandwidth is $\beta_{channel}$, the average performance in phase 1 is given in Equation 3. The fraction part is a single merge tree's performance when sorting N/k elements, and there are k such merge trees in parallel.

$$\beta_{phase1} = k \cdot \frac{\beta_{channel}}{\lceil \log_{l_{phase1}}(N/k) \rceil} \tag{3}$$

Let p_{phase2} denote the merge tree's throughput in phase 2. As long as the merge tree has k or more leaves, phase 2 will require only *one* pass, e.g., if phase 2's merge tree has k leaves, then each of the k sorted sequences from phase 1 will be fed into one of the corresponding k leaves once. According to the analysis in Section 3, the merge tree of phase 2 only exploits a portion of the HBM bandwidth due to on-chip resource constraints. Therefore, the performance β_{phase2} of phase 2 is proportional to p_{phase2}.

The overall sorting performance is in Equation 4.

$$\beta_{overall} = \frac{1}{\frac{1}{\beta_{phase1}} + \frac{1}{\beta_{phase2}}} \tag{4}$$

This highlights that the resource balance between the parallel merge trees of phase 1 and the final merge tree of phase 2 plays an important role in the overall sorting performance. For example, we may have 16 merge trees in the first phase to fully utilize all 32 HBM channels and achieve the best performance. But if the remaining resources only allow us to build a final merge tree whose throughput is equal to the bandwidth of a single channel, then the overall sorting performance will be limited by the final merge, no matter how much better the memory bandwidth utilization we achieve in phase 1. One may wonder if we can reprogram the FPGA in phase 2 to have a wider tree for final merging, but the reprogramming overhead takes several seconds [9] and thus is not practical in this case.

4.2 Merge Tree Reuse in TopSort

To resolve the resource contention between phases 1 & 2, TopSort proposes to reuse the merge units in phase 1 to build the final merge tree. This is based on the observation that a merge tree can be decomposed as sub merge trees plus some extra levels of merge units. For example, a merge tree whose throughput is p in Figure 4 consists of four merge trees of throughput $p/4$, plus two $p/2$-rate merge units and one p-rate merge unit, as shown in Figure 5. In other words, if we want to build a merge tree whose throughput is equal to 4 HBM channels' bandwidth in phase 2, we only need extra resources to build the 3 more merge units. The rest of the resources including the input buffers can be reused from four of the merge trees in phase 1. Figure 6 illustrates the detailed dataflow of the merge tree reuse mechanism in the proposed two-phase approach, which starts with four $p/4$-rate merge trees. If the current phase is phase 1, then the four output

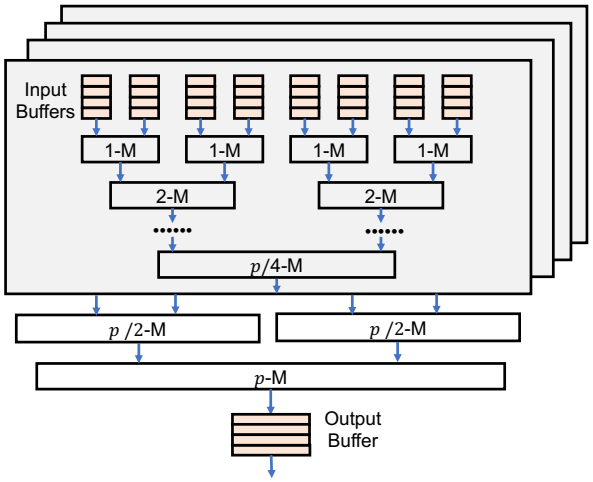


Fig. 5: Merge tree decomposition: four $p/4$-rate merge trees can be combined to build a larger p-rate merge trees.

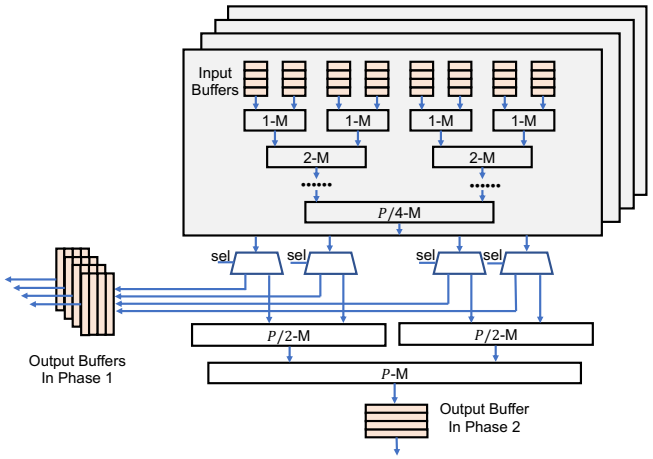


Fig. 6: Detailed dataflow of the merge tree reuse in the proposed two-phase approach: the four demuxes (trapezoids in the figure) are controlled by the same sel signal, which indicates whether the current phase is phase one or phase two.

streams from the roots of the four trees will be directed by the four de-multiplexers into the output buffers on the left side before they are written into the corresponding HBM channels. If the current phase is phase 2, then the four output streams from the four $p/4$-rate merge trees will be switched to flow into the two $p/2$-rate merge units. Later on, the outputs of the two $p/2$-rate merge units are fed into the final p-rate merge unit, thus forming a p-rate merge tree. In this way, the p-rate merge tree in phase 2 is built by reusing the four $p/4$-rate merge trees in phase 1 plus two $p/2$-rate merge units and one p-rate merge unit. The memory write operations in phase 2 will be discussed in detail in Section 4.6.

With this novel merge tree reuse scheme, conceptually, TopSort is able to integrate 16 merge trees to saturate all 32 HBM channels' bandwidth in phase 1 and another merge tree whose throughput saturates 4 HBM channels' bandwidth in phase 2 onto one FPGA. Please note this merge tree reuse approach shows a general architecture that could be easily scaled with different hardware configurations. Given the available off-chip bandwidth (e.g., the number of HBM channels) and on-chip resources (e.g., available BRAMs and LUTs), we can easily adapt (either scale up or down) the number of the parallel merge trees in phase 1, the size (i.e., number of tree leaves l) of each merge tree in phase 1 and the final merge tree in phase 2. The reason for TopSort to select the current configuration will be explained in Section 4.4.

4.3 Architecture to Support Logic Reuse

The overall architecture of TopSort is shown in Figure 7. There are 32 HBM channels available, so TopSort uses 16 merge tree kernels in phase 1. Given a total of N unsorted elements, we split them into 16 sequences, each of which contains $N/16$ elements and is stored in one HBM channel. Then each of the 16 trees will stream the unsorted sequence from one HBM channel and stream the merged (sorted) sequence to another HBM channel, as described in Section 2.3. We configure each tree to have 16 leaves and output 64 bytes per cycle at its root to saturate the bandwidth of a single HBM channel. The leaf number for each merge tree is chosen to be 16 is because the leaf buffers cannot exceed

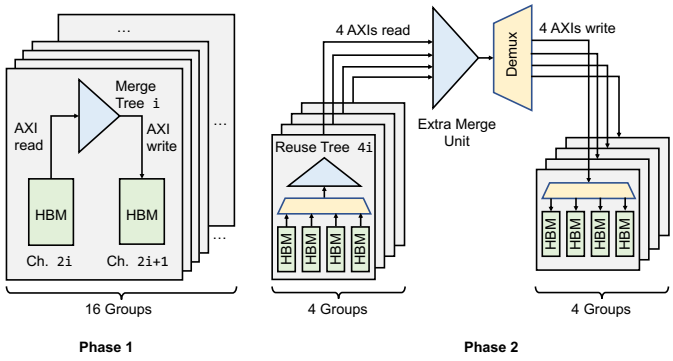


Fig. 7: The overall architecture of the two phases in TopSort. The merge trees rely on AXI interfaces to access HBM channels. The role of demux is covered in Section 4.6.

the available on-chip BRAM limit. Assuming each element is 64-bit, the merge tree will output 8 elements every cycle.

In phase 2, TopSort reuses 4 merge trees from phase 1 and adds two levels of merge units to form a wider merge tree. This merge tree has 64 leaves and outputs 256-byte elements per cycle. Since there are 16 sequences sitting in 16 HBM channels after phase 1, we split each sequence into 4 segmented sub sequences. Then each of the 64 sub sequences is fed into one of the 64 leaves for one pass to get the final sorted results.

In phase 2, the input buffers and each merge unit of the four reused merge trees stay exactly the same as phase 1. We only need to change the memory addresses and the length of the sequences that the AXI interfaces read from each channel for each of the 64 leaves. The change of the write behavior of phase 2 is discussed in Section 4.6.

4.4 Optimization of The Design Choices

As mentioned in Section 4.3, there can be multiple design choices when we select the appropriate TopSort configurations, given a specific HBM-based FPGA board. Now we follow the

TABLE 1: Parameters associated with TopSort models

	Symbol	Definition
Input Param.	N	Number of records in array
	r	Record width in bytes
Hardware Param.	β_{chan}	Bandwidth of a single HBM channel
	C_{BRAM}	On-chip memory capacity in bytes
	C_{LUT}	Number of on-chip logic units
Merge Tree Param.	f	Design frequency
	p_1	Throughput of the merge tree in phase 1
	l_1.	Leaf number of the merge tree in phase 1
	k	Number of the merge trees in phase 1
	β_1	Aggregate performance of phase 1
	p_2	Throughput of the merge tree in phase 2
	l_2.	Leaf number of the merge tree in phase 2
	β_2	Performance of phase 2
	β	Overall performance of the two-phase sort
	b	Size of HBM read/write bursts in bytes

same methodology in [7], [9] and illustrate how to select the best configuration by performing a comprehensive analysis on the performance and resource utilization of TopSort.

4.4.1 Performance Model

Table 1 lists the associated parameters with TopSort. If there are k parallel merge trees in phase 1 and each merge tree has a throughput of p_1 and leaf number of l_1, the average performance of phase 1 can be derived from Equation 3 and is rewritten in Equation 5. Note that the merge tree throughput only needs to saturate the off-chip memory bandwidth and further scaling the merge tree throughput does not improve the effective throughput, we use $p_1 fr$ to represent the actual throughput of each merge tree in phase 1 and make sure p_1 is selected so that $p_1 fr$ is not larger than the available off-chip memory bandwidth to each merge tree.

$$\beta_1 = k \cdot \frac{p_1 fr}{\lceil \log_{l_1}(N/k) \rceil} \tag{5}$$

In the second phase, the final merge tree merges the k sorted sequences into the final result. Since the merge tree in this phase is built on top of the k merge trees in phase 1 and there are $k \times l_1$ leaves available from the merge trees in phase 1, the leaf number of the merge tree in phase 2 is guaranteed to be large than k. As a result, it will require only one pass for this merge tree to merge the k sorted sequences together. Besides, according to the scalability analysis in the previous section, the single merge tree in this phase only exploits a portion of the HBM bandwidth due to the on-chip resource constraints. Therefore, the performance of phase 2 is directly reflected by the tree throughput in phase 2, as shown in Equation 6:

$$\beta_2 = p_2 fr. \tag{6}$$

The overall performance of TopSort is thus calculated by dividing the problem size by the overall time spent in both phases in Equation 7:

$$\beta = \frac{Nr}{\frac{Nr}{\beta_1} + \frac{Nr}{\beta_2}} = \frac{fr}{\frac{\lceil \log_{l_1}(N/k) \rceil}{k \cdot p_1} + \frac{1}{p_2}} \tag{7}$$

4.4.2 Resource Model

Equation 8 lists the on-chip LUT utilization of the merge trees in the whole design. We use m_{2^n} to represent the number of LUTs consumed by a 2^n-rate merge unit, which can be derived by synthesizing the standalone module. At depth n of a merge tree, there are 2^n merge units in total. Thus, the first item on the right hand side in Equation 8 represents the LUTs utilization of k merge trees, each of which has a configuration of (p_1, l_1). Since the final merge tree in phase 2 is built on top of the merge trees in phase 1, we only need to add $\log \frac{p_2}{p_1} + 1$ extra levels of merge units, which correspond to the second item on the right hand side in Equation 8.

$$\text{LUT} = k \cdot \sum_{n=0}^{\log \ell_1} 2^n (m_{\lceil p_1/2^n \rceil}) + \sum_{n=0}^{\log \frac{p_2}{p_1}} 2^n (m_{\lceil p_2/2^n \rceil}). \tag{8}$$

On the other hand, we need to access the HBM channels in burst mode to make sure the HBM channels operating at their peak bandwidth. As each of the l_1 input leaves of the merge trees in phase 1 corresponds to separate segments on the HBM channels, each leaf requires a separate input FIFO for storing the burst reads. Therefore the on-chip memory usage is at least $k \times b \times l_1$. Please also note that the merge tree in phase 2 can fully reuse the input FIFOs of phase 1's trees for its own leaves, the on-chip memory usage is thus given as follows:

$$SRAM = k \cdot b \cdot \ell_1 \tag{9}$$

4.4.3 Design Space Exploration on the Xilinx U280 Board

There are various combinations of k, (p_1, l_1), (p_2, l_2); each of which has a different performance and requires different amount of on-chip resources. To get the configuration that delivers the best performance, we apply the constraints of the available resources to the performance and resource utilization models derived above. Figure 8 lists the performance and resource utilization of various configurations when sorting (32-bit key, 32-bit value) pairs on the Xilinx U280 board. Note that there are roughly 1 million LUTs available on the Xilinx U280 board and it is recommended that the maximum LUT usage is no more than 70%, otherwise the design may not be routable. The configuration we adopt in TopSort is marked by the red circle in Figure 8, which sets k to be 16, (p_1, l_1) to be (8, 16), and (p_2, l_2) to be (32, 64). There is one more point next to our current configuration point, which consumes nearly the same

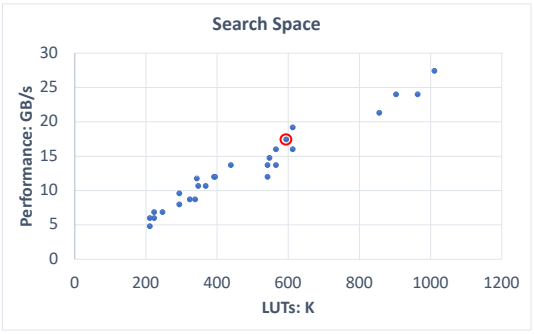


Fig. 8: Search space of different configurations when sorting (32-bit key, 32-bit value) pairs on the Xilinx U280 board. Each dot represents a possible configuration of k, (p_1, l_1), (p_2, l_2). The X-axis indicates the LUT utilization and the Y-axis indicates the theoretical performance of each configuration. The dot marked by a red circle is the actual configuration we choose in TopSort.

amount of LUTs while achieving higher performance. That configuration sets k to be 2, (p_1, l_1) to be (32, 128), and (p_2, l_2) to be (64, 256). However, the design of this configuration is hard to be evenly distributed across the FPGA dies to achieve routable solutions, which is in contrast to our current configuration that allows for coarse-grained floorplanning (see Section 5.4).

4.5 Tuning Sorted Sequence Size from Phase 1

One hidden requirement for a merge tree to output p elements per cycle is that, on average, it has p leaves out of its total l leaves providing elements every cycle. Section 4.3 mentions that each of the 16 sequences in 16 HBM channels after phase 1 needs to be divided into 4 segmented sub sequences so that there are 64 sub sequences fed into the 64 leaves of the tree in phase 2. If each sequence is entirely sorted after phase 1, then the 4 sub sequences have the relation that elements in sub sequence 0 are always smaller than elements in sub sequence 1, and so on. When such 4 sub sequences are fed into 4 leaves, only 1 of the 4 leaves will feed the element to the merge tree at any cycle. For each of the reused merge trees that has 16 leaves, this means only 4 leaves are providing elements into the tree per cycle, as shown in Figure 9. As a result, each merge tree is idle half of the time. Although each tree can output 8 elements per cycle, the average throughput of the tree is 4 elements per cycle, which is below our expectation.

To solve this issue, we need to tune the sorted sequence size from phase 1 so that at least 8 leaves of the merge tree are always actively feeding data in phase 2. This is done by directing each of the merge tree in phase 1 to sort either $N/32$ or $N/64$ elements at a time. For instance, instead of completely sorting the $N/16$-element sequence in each channel, we can get 4 $N/64$-element sorted sub sequences from phase 1 This is done by controlling the number of elements that go into the leaves in the last pass of phase 1. Since the sorted sequence size always grows by the number of leaves l, we deliver $N/1024$ elements into each leaf. At the tree root side, we get $N/64$-element sorted sequence. We repeat this process 4 times for each merge tree in phase 1 to get 4 $N/64$-element sorted sub sequences.

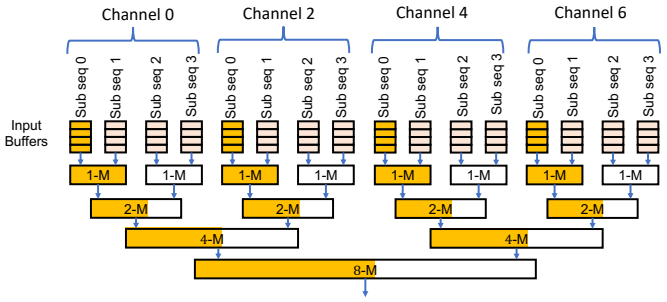


Fig. 9: The active rate of one reused tree in phase 2, if the sequence in each channel is completely sorted. Initially, only the leaves feeding sub sequence 0 from each channel are active, then are the leaves feeding sub sequences 1, and so on. Leaves marked as yellow are actively feeding. Merge units partially marked mean they are idle half of the time.

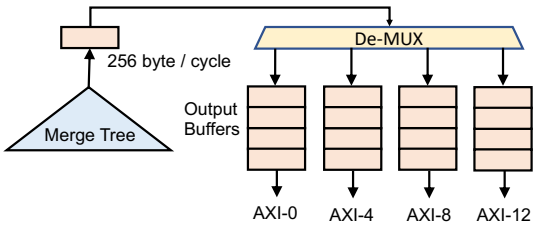


Fig. 10: TopSort's write behavior of phase 2. Writes from 4 AXI interfaces are used to match the tree's throughput.

4.6 Memory Write Pattern in Phase 2

Since the 4× wider merge tree of phase 2 outputs 256-byte sorted elements per cycle, there are 4 AXI interfaces writing to the HBM channels in parallel. The choice of the specific AXI interfaces will be explained in Section 5.2. The on-chip memory is not big enough to hold all the continuously sorted elements before they are written back to entirely fill one HBM channel. Therefore, these continuously sorted elements have to be split into batches and then written back to separate HBM channels through 4 AXI interfaces. Figure 10 depicts the details of such behavior. In our implementation, we choose the batch size to be 4 KB, e.g., we write the first 4 KB sorted elements into the first output buffer, then write the second 4 KB sorted elements into the second output buffer, and so on. The elements in the four buffers will then be written to HBM channels via the four AXI interfaces. As a result, although the elements are completely sorted, one needs to pick the 4 KB batches channel by channel to form the final sorted results.

5 HBM-Specific Optimizations

In this section, we introduce the design choices in TopSort that are coupled with specific HBM characteristics. These HBM-related optimizations are necessary to achieve a full bandwidth usage. First, we minimize the number of AXI interfaces to reduce the resource overhead. Second, we propose a dedicated data layout with spatial locality to prevent bandwidth degradation. In addition, we properly set different burst sizes for different merge trees to strike a balance between the area overhead and the bandwidth.

Finally, we floorplan and pipeline our design based on the architecture of the HBM platform for timing optimization.

5.1 Avoiding Unnecessary AXI Interfaces

According to our measurement, an AXI rate converter in Figure 1 for an HBM channel requires about 5K LUTs and about 6K Flip-Flops (FFs) on the Xilinx U280 FPGA. As a result, the naive choice of instantiating one AXI interface for each of the 32 HBM channels will result in about 320K LUTs, which is more than 40% of the available LUTs on the bottom die where the HBM resides. Such high resource overhead will squeeze the space for user logic and cause severe routing issues.

To address this issue, we utilize the internal crossbar among HBM channels. For each merge tree of phase 1, we only need one AXI module to read from and write to two adjacent HBM channels. This way halves the AXI area overhead and effectively reduces the routing congestion in the bottom die of the FPGA.

5.2 Data Layout Optimization

Although each AXI interface from the user side can access any of the 32 HBM channels, non-local data accesses could cause contention over the lateral connections between channel groups, thus leading to bandwidth decrease. To access an HBM channel outside the group, the data must occupy and traverse through each of the lateral connections until it reaches the destination. Two memory accesses requiring the same lateral connection will cause a conflict and one will be blocked.

Therefore, we must carefully arrange for the data placement to minimize out-of-group channel accesses and avoid conflicts in lateral connections. Table 2 summarizes the target HBM channels for each AXI interface. In the first phase, merge tree i uses AXI-i to only read/write a pair of nearby HBM channels $2i$ and $2i+1$, where i ranges from 0 to 15. In the second phase, the merge tree $4i$ uses AXI-$4i$ to read/write between channels $8i+2j$ and channels $8i+2j+1$, where i & j both range from 0 to 3.

TABLE 2: Correspondence between each user side AXI and the HBM channels it accesses in TopSort, i=0-3.

AXI NO.	HBM Channel NO.
AXI-$4i$	$8i$ - ($8i$+7)
AXI-$(4i+1)$	($8i$+2) - ($8i$+3)
AXI-$(4i+2)$	($8i$+4) - ($8i$+5)
AXI-$(4i+3)$	($8i$+6) - ($8i$+7)

Figure 11 illustrates the access pattern of the first four AXI interfaces in different colors. AXI-1, 2 and 3 will only access the two channels within its crossbar group. Although the AXI-$4i$ in the second phase needs to access channels outside of its local group, our data layout guarantees that different AXI interfaces will not cause conflicts in lateral connections.

5.3 Burst Size Optimization

For TopSort, each of the AXI interfaces will always access the HBM in burst mode. We need to properly choose the burst sizes because a large burst requires excessive buffers while a small burst may not fully utilize the bandwidth.

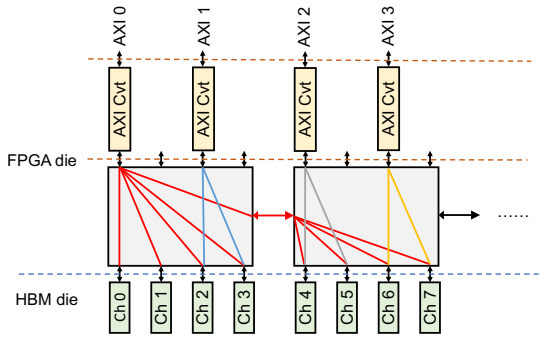


Fig. 11: Usage of the first 4 AXI interfaces. The same behavior is repeated for rest of the 12 AXI interfaces.

In order to select proper burst sizes, we profile the HBM performance based on the memory access pattern of each AXI interface. As shown in Fig. 11, In phase one, each AXI reads from one channel and writes to the adjacent channel; In phase two, AXI-0 reads from four even/odd channels in a round-robin way and writes to the other four nearby odd/even channels.

We define reading from m HBM channels and writing to their nearby m channels as pattern $m\times m$. The patterns of phase 1 and phase 2 are thus 1×1 and 4×4 by this definition. For completeness, we also perform tests for patterns 2×2 and 8×8. Then we measured the achieved bandwidth when using different AXI burst sizes as seen in Figure 12.

Based on our experiments, any AXI burst size from 512 B to 4 KB can achieve the peak HBM bandwidth for the 1×1 pattern, which is used by merge trees in phase 1. However, for the 4×4 inter-crossbar memory access pattern used in phase 2, the AXI burst size needs to be maximized to 4 KB to achieve the best bandwidth. Therefore, we set the burst size to be 1 KB for the 12 merge trees that are not reused in phase 1, and 4 KB for the 4 merge trees that are reused in phase 2.

Minimizing the burst sizes under the bandwidth constraints could significantly save resources because not all buffers are implemented in BRAM. For each merge tree, each leaf node requires a 512-bit wide input buffer (see Figure 4) to hold the read bursts from AXI interfaces. When implemented using BRAM, each buffer consumes at least 7.5 BRAMs on Xilinx FPGAs. Since we have 16 merge trees and each tree has 16 leaves, the FPGA does not have enough BRAMs to implement all the buffers. So, about $3/4$ of the buffers are implemented using LUT-based shift registers. While BRAM-based buffers are insensitive to the burst size, the area of a LUT-based buffer is directly proportional to the buffer depth. A single LUT can implement a 1-bit shift register with a depth of 32 and 512 LUTs can hold two AXI bursts of 1 KB. Reducing the bursts size from 4 KB to 1 KB directly reduces the LUTs consumption by $4\times$. Further reducing the burst size from 1 KB to 512 B does not save more LUTs because 2 bursts of 512 B still require 512 LUTs. The actual sorting results with different burst sizes in TopSort is further discussed in Section 6.5.

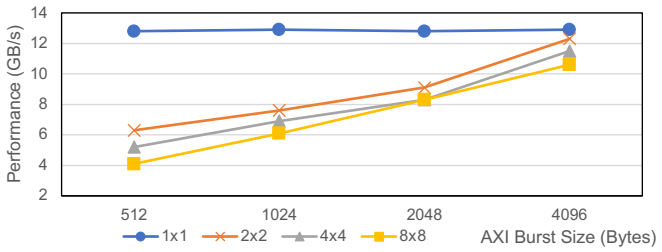


Fig. 12: HBM channel performance of inter-crossbar & intra-crossbar behaviors when varying the AXI burst size. The number of outstanding bursts is fixed at 32.

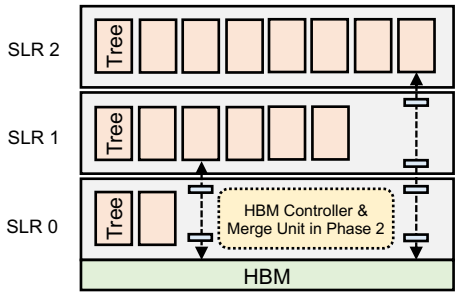


Fig. 13: The floorplan of TopSort on the Xilinx U280 FPGA. Each Super Logic Region (SLR) is a single FPGA die slice in the Stacked Silicon Interconnect (SSI) FPGA device

5.4 Floorplanning Optimization

Thanks to the parallel sorting in the first phase and merge tree reuse in the second phase, we are able to separate each of the merge tree kernels independently and apply a simple yet efficient resource model to better distribute these kernels across the three FPGA dies, as shown in Figure 13. Please note that the merge tree kernels in the first phase only expose simple AXI interfaces and each of the merge tree has the same size. This feature allows us to easily implement a coarse-grained floorplanning with the granularity of the separate merge tree kernel. In contrast, if we simply scale the single merge tree and want to make use of all of the FPGA dies, it would requires significantly more engineering efforts to manually split the design logics and fit them into each die.

Considering that the sorted output from the merge tree of phase 2 will be directly written to the HBM channels, a natural choice is to place the extra merge units on the bottom die to minimize the signal crossing. After that, we will migrate as many merge tree kernels to the mid and top dies, to save more spaces in the bottom die to alleviate the routing issues.

We derive a simple resource model to solve this floorplanning problem. Let L be the resource consumption of each merge tree, u_i be the number of merge trees and a_i be the available resources on the i-th die, w be the width of an AXI interface, and W be the available signals crossing two neighbor dies. Then the problem can be formulated as *maximizing* $(u_1 + u_2)$ under the following constrains:

$$\begin{cases} (u_1 + u_2) \cdot w \leq W & (1) \\ u_1 \cdot L \leq a_1 & (2) \\ u_2 \cdot L \leq a_2 & (3) \end{cases}$$

where constraint (1) limits the AXI connections from all merge trees in both die 1 and 2 to die 0 do not exceed the available amount of signals crossing die 0 and 1, and constrains (2) and (3) make sure the merge trees placed on die 1 and 2 do not exceed each die's available resource.

By solving the above problem, we place eight merge trees on the top die and six merge trees on the middle die. There are another two merge trees remaining on the bottom die; we select these two merge trees as merge tree 0 and merge tree 8. The detailed layout are presented in Section 6.3.

Finally, to improve the timing, the signals crossing dies need to be pipelined with registers [28]. In general, we add 2 stages of pipeline registers when a signal crosses one die and add 4 stages of pipeline registers when it crosses two dies.

6 Experimental Results

6.1 Experimental Setup

We perform all of the experiments on the Xilinx U280 FPGA board. The kernel is developed using System Verilog and is synthesized and implemented using Xilinx Vitis 2020.2. We use the pblock method [29] to place the design modules during the floorplanning. The input data are in a key-value pair format, each with a 32-bit key and a 32-bit value. The keys used in the experiments have a uniform distribution. To ensure the keys are fully randomized, we first generate N records whose keys are incremented from 1 to N, then we use the $random\_shuffle()$ function from the python library to shuffle the records. The sorted results are validated by checking whether their keys are from 1 to N.

6.2 TopSort Configuration, Resource Utilization & Power Consumption

We implement 16 merge trees in total, each with 16 leaves and outputs 8 elements (each 8 bytes) per cycle. In phase 1, TopSort uses all 16 merge trees. In phase 2, TopSort reuses 4 of the merge trees to form a wider merge tree that can sort 32 elements (each 8 bytes) per cycle.

The resources utilization of the dynamic region[2] is listed in Table 3, including our TopSort kernel and HBM controllers. TopSort consumes 54.6% LUTs, 35.5% FFs and 56% BRAMs.

Table 4 shows the resource breakdown of each type of the merge trees. For those trees that are not reused (e.g., merge tree 1), each of them takes less than 3% of the available resources. This is consistent with the observation in Section 3 that it takes significantly less resources for a single merge tree to saturate one HBM channel than multiple HBM channels. Meanwhile, a reused tree (e.g., merge tree 4) requires more LUTs because its input buffers need to

2. The FPGA is partitioned into two regions, with the dynamic region reserved for user logic and the static region used for infrastructure IPs.

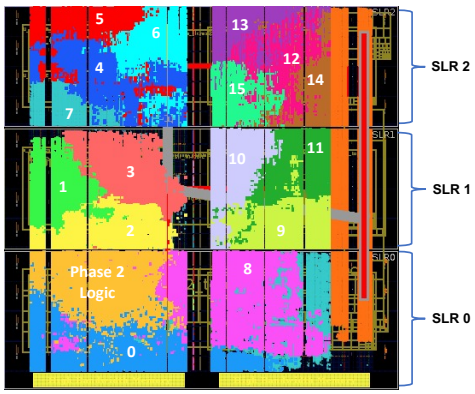


Fig. 14: Layout of TopSort implemented on the Xilinx U280 FPGA board. Number 0-15 label the 16 merge trees of phase 1. The orange part in the bottom die represents the extra logic used to form the wider merge tree of phase 2.

accommodate larger data bursts. As for the extra 3 merge units used in phase 2, they takes much more resources than a single merge tree, which validates the theoretical analysis that directly scaling the tree throughput requires the resources to grow superlinearly.

TABLE 3: Resources utilization and percentage of the TopSort kernel and HBM controllers. DSPs are used to calculate the addresses of AXI read and write.

Components	LUTs	Flip-Flops	BRAMs	DSPs
Available	1,066,848	2,144,089	1,487	8,484
TopSort kernel	582,244	750,505	834	84
Percentage	54.6%	35.5%	56.0%	1.0%
HBM Controllers	87,848	113,814	4	0
Percentage	8.2%	5.3%	0.2%	0.0%

TABLE 4: Resources utilization of individual merge trees. Here we pick one merge tree from each type since they have slightly different resource consumption.

Components	LUTs	Flip-Flops	BRAMs	DSPs
Merge tree 1	28,788	39,905	37.5	2
Percentage	2.7%	1.9%	2.5%	0.02%
Merge tree 4	39,195	39,459	60	10
Percentage	3.7%	1.8%	4.0%	0.1%
Extra logic of phase 2	60,239	102,864	144	20
Percentage	5.6%	4.8%	9.7%	0.2%

Table 5 summarizes the power consumption of TopSort obtained from the Xilinx Power Estimator (XPE) [30]. We expect that only 51.4 Watt of the total on-chip power is needed to enable TopSort to deliver the peak performance.

6.3 Design Layout & Frequency

Figure 14 shows the final placements of all merge trees and the extra phase 2 logic. Our floorplanning and pipelining effectively reduces local routing congestion, so that TopSort could achieve 214 MHz in user clock and 414 MHz in the HBM control clock. Without our floorplanning, Vivado failed in routing even with the highest optimization level.

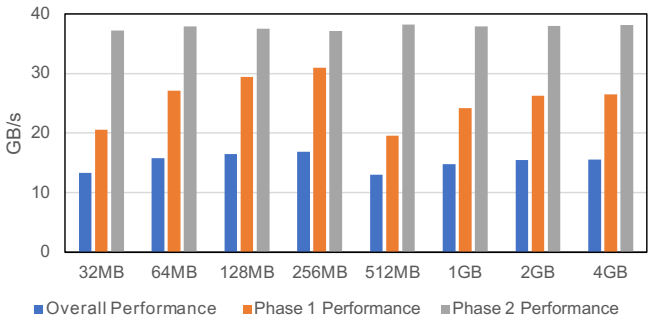


Fig. 15: Sorting performance with different data sizes. The overall performance is calculated through dividing the data size by the total sorting execution time.

6.4 Overall Sorting Performance

Figure 15 presents the performance of sorting data ranging from 32 MB to 4 GB in size. The maximal data size that we can handle is half of the HBM capacity, which is 4 GB in the Xilinx U280 FPGA. This is because half of the HBM channels are used for reading and the other half for writing. For datasets whose sizes are larger than 4 GB, another layer of memory with larger capacity such as the host DRAM or FPGA DRAM must be presented: we may need to divide the entire dataset into multiple smaller pieces that can fit into the HBM to get them independently sorted. Then we may merge these sorted pieces of data through another pass. But this is out of the scope of this work and we omit the further discussion here. When sorting 4 GB of data, TopSort achieves an overall performance of 15.6 GB/s. Please note that 4 GB is half of the HBM capacity and the maximal size that we are able to sort. Sorting 4GB data has a performance of 26.5 GB/s in the first phase and 38 GB/s in the second phase.

Figure 15 also shows a performance drop when the data size is increased from 256 MB to 512 MB. This is because sorting 512 MB data takes 6 passes in phase 1 while sorting 256 MB data takes 5 passes. In fact, we choose to direct each of the merge trees in phase 1 to get two $N/32$-element sorted sub sequences, as illustrated in Section 4.5. Since the merge trees in phase 1 have 16 leaves, each of them can work with at most $2 \times 16^5 = 2^{21}$ elements within 5 passes. There are 16 merge trees working in parallel and each element is 8-byte wide, giving a total data size to be $2^{21} \times 16 \times 8 = 256$ MB. Similarly, data with a size larger than 256 MB but no more than than 4 GB requires 6 passes in phase 1.

Besides, sorting 512 MB to 4 GB data requires the same number of passes in phase 1, but the sorting performance still varies, which is similar for sorting 32 MB to 256 MB data. This is because some input leaves of the merge trees may not be fully active in the last pass of phase 1, similar to what is illustrated in Figure 9. For instance, in the case of sorting 512 MB data, each merge tree in phase 1 needs to get two $512/16/2 = 16$ MB sorted sub sequences, or equivalently 2^{21}-element sorted sub sequences. However, we always get $16^5 = 2^{20}$-element sorted chunks after the first 5 passes. That means the 2^{21}-element sub sequence to be sorted contains two 2^{20}-element sorted chunks. During the last pass, each of the two sorted chunks is divided into 8 continuous portions and each portion is fed into one merge tree leaf. In our design, portion

TABLE 5: Power consumption of TopSort.

TopSort kernel:	41.6 W	Dynamic:	46.8 W
HBM controller:	9.8 W	Static:	4.6 W
Total:	51.4 W	Total:	51.4 W

1 of the first sorted chunk goes to leaf 1, portion 2 of the first sorted chunk goes to leaf 2, and portion 8 of the first sorted chunk goes to leaf 8. The correspondence of the 8 portions from the second sorted chunk to leaf 9-16 is the same. In this case, since the records from leaf 1 are always smaller than the records from leaf 2 and the records from leaf 2 are always smaller than the records from leaf 3, etc, only one of the leaf 1-8 is active all the time: e.g., the merge tree will always fetch the records from leaf 1 until it fully consumes portion 1, then it always fetches the records from leaf 2 until portion 2 is fully consumed, so on and so forth. The same situation happens for leaf 9-16. As a result, only 2 leaves out of the 16 leaves are active, which cannot sustain the merge tree to output 8 elements every cycle and leads to a lower performance. Similarly, if the data size is 1 GB, then each merge tree in phase 1 gets the sorted sub sequences with a size of 32 MB, which contains four 2^{20}-element sorted chunks. Each of the sort chunks will be divided into 4 portions, which will feed 4 leaves. As a result, 4 leaves out of the 16 leaves are active in the last pass, which gives better performance than the case of sorting 512 MB data. Likewise, in the cases of sorting 2 GB and 4 GB data, 8 and 16 leaves out of the 16 leaves are active, since the tree outputs 8 elements per cycle, the merge trees in both cases are fully active. That's why sorting 2GB and 4GB data has quite similar overall performance, which is higher than that of sorting 512MB or 1GB data.

Based on our measurement, we are able to utilize at least 318 GB/s of HBM bandwidth in the first phase, where we run 6 passes and each pass reads and writes to HBM channels simultaneously. Thus the average HBM bandwidth utilized is at least $26.5 \times 6 \times 2 = 318$ GB/s. In fact, the used bandwidth is even higher because we have not account for the idle time of the merge units, which needs to be reset after merging two sorted sequences. In the initial pass, the length of the sorted sequences is 1, so the merge units have to be reset frequently.

6.5 Sorting Performance with Different Burst Sizes

Figure 16 compares the performance of two burst size choices. Both tests set the burst size as 1 KB in phase 1. In the second phase, we set the burst sizes to be 1 KB and 4 KB. The results show that reading in 4KB bursts in phase 2 gives better performance, which matches the profiling results in Section 5.3 that the designers need to use 4KB AXI bursts to maximize the HBM channels' bandwidth when they use one AXI to access multiple HBM channels.

6.6 Comparison with State-of-the-art CPU & FPGA Sorters

TopSort is an in-memory sorter [31], which stores the entire input in main memory such as an HBM or a DRAM. An in-memory sorter has much higher overall sorting performance than an in-storage sorter, which requires a second-level storage such as SSD to sort a larger dataset and thus is bounded

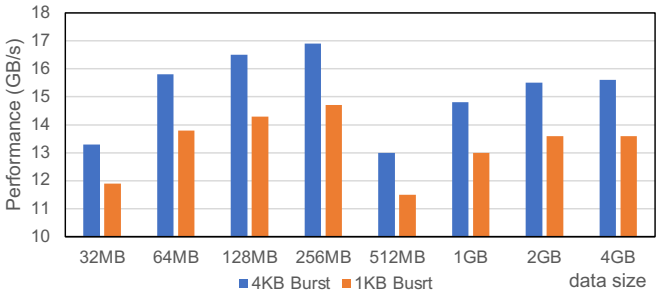


Fig. 16: Overall sorting performance with the same AXI burst size in phase one but different AXI burst sizes in phase two.

TABLE 6: Comparison with existing CPU based in-memory sorters.

	Algorithm	Overall Sorting Perf
C++ std::sort [32]	Merge sort	0.27 GB/s
VLDB'15 [24]	Radix sort	2.3 GB/s
TopSort	Merge tree sort	15.6 GB/s

by the slow I/O [5], [9]. In fairness, we compare TopSort against several CPU and FPGA in-memory sorters here. Table 6 compares TopSort with two existing CPU based in-memory sorters. We first optimize the standard C++ std::sort() function on the Intel Xeon E5-2680 CPU by fully utilizing 32 threads and get an overall sorting performance of 0.27 GB/s when sorting 4GB of data. Compared to it, TopSort achieves 57.7× speedup. The state-of-the-art CPU in-memory sorter in [24] can sort 4GB of data in 1.7 second, achieving an overall sorting performance of about 2.3 GB/s. Compared to it, TopSort achieves 6.7× speedup.

Table 7 lists the existing FPGA in-memory sorters. The best-performing DRAM-based FPGA sorter in [7] can fully utilize 4 DDR4 DRAM channels on the AWS F1 datacenter FPGA. When sorting 4 GB data, it achieves an overall sorting performance of 5.8 GB/s. Comparing to it, TopSort achieves 2.7× speedup. Besides, we also list the absolute merge tree throughput, which is the number of elements (p) that the merge tree can output per cycle multiplied by the design frequency and the element's width in bytes, to show the utilization of the off-chip memory bandwidth of different merge tree designs. TopSort has an entire tree throughput of 219 GB/s in phase 1 and saturates the 420 GB/s HBM bandwidth (HBM is half-duplex so half of its bandwidth is for reading data and the other half is for writing data). In phase 2, TopSort still has 1.7× and 14× higher merge tree throughput than that of [7], [8], respectively.

One may find that the overall sorting performance improvement of TopSort over [7] is much less than the 6× off-chip bandwidth increase. This is because the on-chip resources of the Xilinx U280 FPGA is merely enriched by about 20% compared to the AWS F1 FPGA. As analyzed in

TABLE 7: Performance comparison with existing FPGA based in-memory sorters.

	Algorithm	Off-Chip Memory Used	Merge Tree Throughput	Design Frequency	Overall Sorting Perf.
FPGA'20 [15]	Sample sort	4 DDR4 channels	N/A	250 MHz	4.3 GB/s
FPL'20 [8]	Merge tree sort	1 DDR4 channel	4 GB/s	214 MHz	0.2 GB/s [3]
ISCA'20 [7]	Merge tree sort	4 DDR4 channels	32 GB/s	250 MHz	5.8 GB/s
TopSort	Merge tree sort	32 HBM channels	219 GB/s (phase 1) 55 GB/s (phase 2)	214 MHz	15.6 GB/s

Section 2, the on-chip resources become the new bottleneck to linearly scale the overall sorting performance. While TopSort could address this issue to some extent, its overall performance is still bound by the merge tree in phase 2. That is why only 2.7× speedup is reported.

Table 8 compares the resource utilization of TopSort with the existing FPGA in-memory sorters. Since the performance and resource utilization are different among various sorters, we introduce a more unified metric called the resource efficiency, which is defined as dividing the overall sorting performance of a sorter in GB/s by its LUT utilization in the unit of 100K LUTs. For example, TopSort has an overall sorting performance of 15.6 GB/s and utilizes roughly 5.8×100K LUTs, thus its resource efficiency is 15.6/5.8 = 2.7 GB/s/100K LUTs. Table 8 shows that TopSort has the best resource efficiency, due to the two major design choices. First, TopSort chooses to use multiple smaller merge trees to independently work with the HBM channels in the first phase, which provides a linear performance improvement while avoiding the super-linear growth of resource utilization shown in Section 3. Second, the logic reuse in the second phase further saves the overall resource utilization. Since the on-chip resource utilization of a design is directly related to its power consumption, the resource efficiency can also serve as a metric to indicate the sorter's power efficiency. As a result, we believe that TopSort also delivers better power efficiency than the existing FPGA sorters.

TABLE 8: Comparison of resource utilization for various FPGA based in-memory sorters. The resource efficiency of the sorter is defined as the overall sorting performance in GB/s divided by the LUT utilization in the unit of 100K LUTs.

	LUTs	Flip-Flops	BRAMs	Resource Efficiency
FPGA'20 [15]	328,366	391,900	760	1.3
FPL'20 [8]	44,000	61,000	64	0.5
ISCA'20 [7]	287,672	768,906	960	2.0
TopSort	582,244	750,505	834	2.7

6.7 Comparison with Scaled Single Merge Tree

We also compare TopSort to a single giant merge tree, which is scaled to output 32 elements per cycle and has a throughput equal to writing 4 HBM channels, i.e., the same as the merge tree throughput of phase two in TopSort. Unfortunately, we are not able to route this single merge tree, as it requires significantly more engineering efforts of manual placement optimization compared to TopSort. The reason why it is easier to floorplan TopSort than to floorplan single merge tree is explained in Section 5.4. Nonetheless, we can still estimate its performance, assuming it achieved the same design frequency as TopSort.

The best estimation is that the single merge tree has the same number of leaves as the sum of the 16 merge tree leaves in phase one of TopSort, which is 256. This means sorting 4 GB data still requires 4 passes. Since the throughput of the merge tree is the same as TopSort's throughput in phase 2, which is 38 GB/s, the overall sorting performance of the single merge tree is thus at most 38/4 = 9.5 GB/s, much less than the 15.6 GB/s achieved in TopSort.

6.8 Sensitivity to the Input Data

The methodology and architecture of TopSort can be adopted to sort different datasets with different kinds of keys or (key, value) pairs. Figure 17 shows the overall sorting performance when sorting two kinds of datasets with uniformly random distributions. One kind of them is (32-bit key, 32-bit value) pairs, as mentioned above. The other kind is pure 32-bit keys. In the case of sorting 32-bit keys, we still use 16 merge trees in phase 1, each having 16 leaves and output 16 elements per cycle at its root. At the leaf layer of each merge tree, we have 8 2-rate merge units. After the first phase, each merge tree will generate 2 $N/32$-element sorted sub sequence. In phase 2, we still reuse 4 merge trees from phase 1 to form a wider tree that has 64 leaves and outputs 64 elements per cycle. We can see that both cases report similar overall sorting performance, this is because they have the same number of passes when sorting the same amount of data, despite their record width is different.

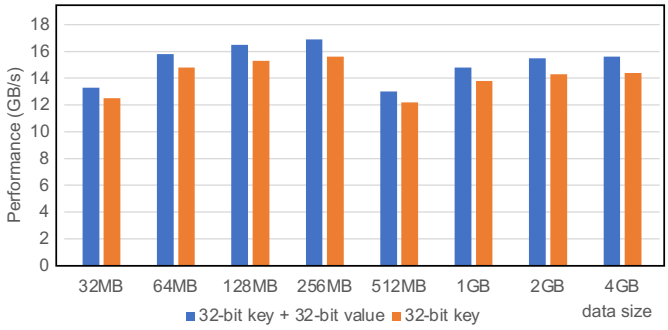


Fig. 17: Overall sorting performance when sorting (32-bit key, 32-bit value) pairs and 32-bit keys. The performance variation is due to the difference in the design frequencies.

We also evaluate the performance of TopSort using different key distributions. In order to generate various key distributions with different skew, we adopt the benchmark from [33], which uses the Shannon entropy to measure the key distribution. The idea is that skewed keys can be generated by performing *bitwise AND* operations on the adjacent keys which originally have a uniformly random distribution. For instance, an entropy of

3. [8] reported 49× speedup over the C++ std::sort() implementation on an ARM Cortex A53 core when sorting 256 KB data. We re-implement C++ std::sort() on the same CPU and multiply the performance when sorting 32 MB data by 49× to estimate the value as listed.

32 bits for 32-bit keys means the keys are uniformly random distributed. On the other hand, an entropy of 0 bit indicates all the keys are the same.

Figure 18 shows the overall performance of TopSort when sorting vastly different skewed data. We observe that TopSort is insensitive to the data skew. Specifically, TopSort achieves the highest performance of 16.4 GB/s when sorting data that share the same key. This is because we design each merge unit so that it always fetches records in a round-robin fashion from both input ports if the keys from the two input ports are the same. In the case of sorting data that have the same key, any E-rate merge unit at any level of the merge tree is guaranteed to first consume E records from its left-side input port, then consume E records from its right-side input port, etc. In this way, the merge tree is always fully active during the sorting process.

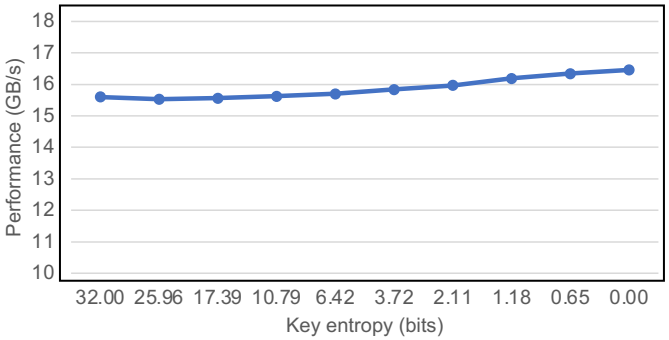


Fig. 18: Overall sorting performance when sorting 4 GB data of (32-bit key, 32-bit value) pairs with different key distributions.

7 Related Work

7.1 Sorting Acceleration on FPGAs

Sorting acceleration on FPGAs has been a widely studied topic in recent years. Table 9 summarizes the features of the existing FPGA-based sorting designs. [10], [34] proposed two bitonic sorters that could sort kilobyte-scale elements by designing huge on-chip bitonic sorting networks, but they could not sort data whose size is larger than several megabytes. [15] designed a Sample sort accelerator, which included a CPU to perform the sampling and an FPGA to do the bucket partition as well as small-scale sorting for each bucket. [2], [35] focused on the implementation of the naive merge sort, which involved only one merge unit and always merged two sequences into one sequence. Compared with the merge tree sort that merges multiple sequences at a time, this simpler merge sort based solution had much lower sorting performance. All the other related works fell into the category of the merge tree sort.

The existing works related to the merge tree-based acceleration are split into two directions. One direction focuses on optimizing the resource consumption of the merge unit itself, at the center of which is a variation of parallel sorting networks such as bitonic or even-odd sorting networks [11], [12], [13], [14]. TopSort benefits from these works in that using efficient merge units may always reduce the on-chip resource consumption of the merge tree. For instance, we believe that adopting the newly proposed merge unit called FLiMSj from [26] may further reduce the resource consumption of TopSort. In this work, we adopt the merge units from [12], which contains two bitonic mergers, but rewrite the whole control logic to make the merge unit has the pure streaming behavior.

The other direction investigates how to design efficient merge trees that interact with various memory layers such as DRAM and storage [5], [6], [7], [8], [9]. These works all used a single tree to sort the entire data, with the best optimization to be scaling the single merge tree's throughput to match the off-chip DRAM bandwidth. Compared with them, TopSort is the first HBM-based FPGA sorter, which involves a more challenging problem where the off-chip memory bandwidth is improved by at least 6× while the on-chip resources only increase by 20%. In this case, the previous single merge tree solution could not be further scaled to improve the bandwidth utilization or deliver better performance. In contrast, TopSort adopts a distinct two-phase solution where multiple merge trees work in parallel in phase 1 to fully utilize the HBM bandwidth and phase 2 novelly reuses the merge trees from phase 1 to form a wide merge tree to further improve the overall performance under the limited resource constraint.

7.2 HBM-Specific Optimizations

As for HBM-specific optimizations, [22] presents an HLS design that applies a similar floorplanning strategy and achieves a design frequency of 237 MHz when using 18 HBM channels. The majority of the recent HBM-based accelerators [19], [20], [23] are HLS designs but are not able to get more than 190 MHz and use more than 28 HBM channels. [21] implements a hash join accelerator that uses 32 HBM channels while running at 250 MHz and its random memory accesses are through 256-bit wide AXI interfaces. However, using a 256-bit wide AXI interface can only utilize half of the available HBM bandwidth at most. Compared to it, our work relies on continuous memory reads and writes, which requires the AXI interface to be 512-bit wide to maximize the memory performance and consumes significantly more on-chip resources for the AXI rate converters as well as the AXI burst buffers. Besides, we need to carefully optimize the data layout to avoid access conflicts in the lateral connections of the built-in AXI crossbars, which is never revealed in those applications with random memory access patterns. We believe that our work provides valuable insights on how to optimize HBM-based accelerator designs, especially HLS-based designs that are struggling to fully utilize the HBM bandwidth with a high design frequency.

8 Conclusion

In this work, we present TopSort, a high-performance two-phase sorting accelerator specialized for HBM-based FPGAs. Our analysis shows that the sorter performance on HBM-based FPGAs is bounded by the limited on-chip resources. To achieve better performance, TopSort proposes a novel two-level sorting solution with smaller merge trees. In the first phase, TopSort can fully utilize all the HBM bandwidth. In the second phase, TopSort reuses the logic from the first phase to avoid the resource contention. Moreover, TopSort adopts several HBM-specific optimizations to further reduce the resource overhead and improve the bandwidth utilization. Finally, it also employs coarse-grained

TABLE 9: A summary of the features of the existing FPGA-based sorter/merger designs. A "Yes" in the Merger column indicates the corresponding work optimizes the merge unit as the building block for the merge tree sort. A "Yes" in the Complete sorter column means the work performs the real experiments to completely sort the data and the Data size column as well as the Memory type column show the actual size of the data it sorts and the memory it works with, respectively. The Analytical model column lists whether the work does comprehensive analysis on either the sorting performance or the resource utilization.

	Merger	Complete sorter	Data size	Memory type	Algorithm	Analytical model
DAC'12 [34]	N/A	Yes	KB	On-chip SRAM	Bitonic sort	Yes
FPGA'15 [10]	N/A	Yes	KB	On-chip SRAM	Bitonic sort	Yes
FCCM'16 [4]	Yes	Yes	KB	On-chip SRAM	Merge tree sort	Yes
FPGA'16 [3]	No	Yes	KB	On-chip SRAM	Multiple	No
FPGA'11 [1]	Yes	Yes	MB	DRAM	Multiple	No
MEMOCODE'08 [35]	No	Yes	MB	DRAM	Merge sort	No
FPGA'14 [2]	No	Yes	MB	DRAM	Merge sort	No
FPL'20 [8]	No	Yes	MB	DRAM	Merge tree sort	Yes
FPGA'20 [15]	N/A	Yes	GB	DRAM	Sample sort	No
FPT'18-1 [6]	Yes	Unknown	GB	DRAM	Merge tree sort	Yes
ISCA'20 [7]	No	Yes	GB	DRAM	Merge tree sort	Yes
FCCM'17-1 [5]	No	Yes	TB	DRAM & Flash	Merge tree sort	No
FCCM'21 [9]	No	Yes	TB	DRAM & Flash	Merge tree sort	Yes
FCCM'17-2 [11]	Yes	No	N/A	N/A	N/A	N/A
FCCM'18 [12]	Yes	No	N/A	N/A	N/A	N/A
FPT'18-2 [13]	Yes	No	N/A	N/A	N/A	N/A
MCSoc'19 [14]	Yes	No	N/A	N/A	N/A	N/A
TC'22 [26]	Yes	No	N/A	N/A	N/A	N/A
TopSort	No	Yes	GB	HBM	Merge tree sort	Yes

floorplanning to achieve better time closure. TopSort is the first HBM-based FPGA accelerator that can fully utilize all the HBM channel bandwidth and achieves 6.7× and 2.7× speedup over state-of-the-art CPU sorter and DRAM-based FPGA sorter.

ACKNOWLEDGMENT

This work is partially supported by CRISP, one of six JUMP centers and the CDSC industrial partners, including Samsung and Siemens Mentor Graphics. We also thank Zhe Chen for his invaluable support in measuring the CPU sorting performance on the ARM Cortex A53 core.

REFERENCES

[1] D. Koch and J. Torresen, "FPGAsort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2011.

[2] 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.

[3] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis," in *Proceedings of the 24th ACM/SIGDA international symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[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.-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.

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

[7] 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.

[8] 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.

[9] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, "Fans: Fpga-accelerated near-storage sorting," in *Proceedings of the 29th IEEE international symposium on Field-programmable custom cumputing machines (FCCM)*, 2021.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking," in *2021 International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21, 2021, p. 105–115.

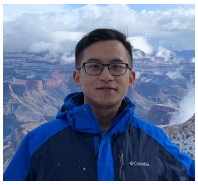
[17] Y. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: high-performance hls interconnect for fpga hbm," in *Proceedings of the 29th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2021.

[18] Xilinx. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds963-u280.pdf

[19] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

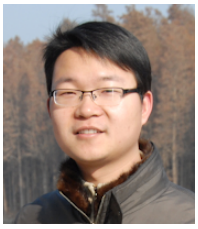
[20] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth

memory based accelerator for general-purpose sparse matrix-vector multiplication," *arXiv preprint arXiv:2111.12555*, 2021.
[21] J. Wirth, J. A. Hofmann, L. Thostrup, C. Binnig, and A. Koch, "Scalable and flexible high-performance in-network processing of hash joins in distributed databases," in *Proceedings of the 2021 International Conference on Field-Programmable Technology (FPT)*, 2021, pp. 1–9.
[22] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022.
[23] Y. Chi, L. Guo, and J. Cong, "Accelerating sssp for power-law graphs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022.
[24] M. Cho, D. Brand, and R. Bordawekar, "PARADIS: An efficient parallel algorithm for in-place radix sort," in *Very Large Data Bases (VLDB)*, 2015.
[25] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 113–124.
[26] P. Papaphilippou, W. Luk, and C. Brooks, "Flims: a fast lightweight 2-way merger for sorting," *IEEE Transactions on Computers*, pp. 1–1, 2022.
[27] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *ACM SIGACT News*, vol. 12, no. 3, pp. 36–44, 1980.
[28] 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.
[29] Xilinx. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2020_2/ug938-vivado-design-analysis-closure-tutorial.pdf
[30] Xilinx. [Online]. Available: https://docs.xilinx.com/r/en-US/ug440-xilinx-power-estimator/Overview?tocId=1dgXNCIFAtTUfWCaTUG~ZA
[31] D. E. Knuth, *Art of Computer Programming: Sorting and Searching*, 2nd ed. Addison-Wesley Professional, 1998.
[32] C++ Standard Library. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/sort
[33] K. Thearling and S. Smith, "An improved supercomputer sorting benchmark," in *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992.
[34] M. Zuluaga, P. Milder, and M. Püschel, "Computer generation of streaming sorting networks," in *DAC Design Automation Conference 2012*, 2012, pp. 1241–1249.
[35] K. Fleming, M. King, M. C. Ng, A. Khan, and M. Vijayaraghavan, "High-throughput pipelined mergesort," in *2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, 2008, pp. 155–158.



Weikang Qiao received his B.S. degree in Information and Communication Engineering from Zhejiang University and his M.S. degree in Electrical Engineering from UCLA. Currently, he is a fifth-year Ph.D. student in the Electrical and Computer Engineering department at UCLA. His research focuses on customized accelerator architecture designs and performance modeling across various memory hierarchies, such as DRAM, High-bandwidth Memory (HBM) and SSDs. He is an IEEE student member.

Licheng Guo received his B.S. degree in Electrical Engineering from Zhejiang University in 2018. Currently he is a 4th-year Ph.D. student in the UCLA CS department. His research focus on co-optimizing HLS compilers (from C++ to RTL) and physical design tools (from RTL to hardware) to improve the circuit maximal frequency and reduce the compilation time.



Zhenman Fang received his PhD degree in Computer Science from Fudan University, China in 2014. He did his postdoc at UCLA from 2014 to 2017, and worked as a Staff Software Engineer at Xilinx, San Jose, from 2017 to 2019. Currently, Zhenman is an Assistant Professor in School of Engineering Science, Simon Fraser University, Canada. His recent research focuses on customizable computing with specialized hardware acceleration, including emerging application characterization and acceleration, novel accelerator-rich and near-data computing architecture designs, and corresponding programming, runtime, and tool support. He is a member of the IEEE and ACM.

Mau-Chung Frank Chang is the Wintek Chair in Electrical Engineering and Distinguished Professor at UCLA. He pioneered the development of the world's first multi-gigabit/sec data converters in heterojuction bipolar technologies; first mm-Wave Radio-on-Chip with Digitally Controlled on-chip Artificial Dielectric (DiCAD) for broadband frequency tuning and on-chip sensing/actuating with cautious feedback control for achieving self-diagnosis and self-healing capabilities. He realized the first CMOS frequency synthesizers up to terahertz spectra and demonstrated tri-color and 3-dimensional CMOS active imagers at the (sub)-mmWave spectra based on a time-encoded digital architecture. He is a Member of the US National Academy of Engineering, a Fellow of US National Academy of Inventors, an Academician of Academia Sinica of Taiwan, and a Lifetime Fellow of IEEE. He was honored with the IEEE David Sarnoff Award in 2006 for developing and commercializing GaAs HBT and BiFET power amplifiers, which dominated smartphones transmitter worldwide production throughout the past 2.5 decades.

Jason Cong received his B.S. degree in computer science from Peking University in 1985, his M.S. and Ph. D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1987 and 1990, respectively. Currently, he is the Volgenau Chair for Engineering Excellence (and former department chair) at the UCLA Computer Science Department, with joint appointment from the Electrical Engineering Department. Dr. Cong's research interests include novel architectures and compilation for customizable computing and quantum computing. He has over 500 publications in these areas, including 16 best paper awards, three 10-Year Most Influential Paper Awards, and three papers in the FPGA and Reconfigurable Computing Hall of Fame. He was elected to an IEEE Fellow in 2000, an ACM Fellow in 2008, a member of the National Academy of Engineering in 2017, and a Fellow of the National Academy of Inventors in 2020.