Hypersort: High-performance Parallel Sorting on HBM-enabled FPGA

Soundarya Jayaraman
National Institute of Technology
Tiruchirappalli, India
soundarya.jayaraman1@gmail.com

Bingyi Zhang
University of Southern California
Los Angeles, CA, USA
bingyizh@usc.edu

Viktor Prasanna
University of Southern California
Los Angeles, CA, USA
prasanna@usc.edu

***Abstract*—Accelerating sorting on FPGA has been extensively studied by leveraging the fine-grained data parallelism of FPGAs. However, with the optimized hardware pipelines, the performance of sorting algorithms is bounded by the off-chip memory bandwidth. The integration of high-bandwidth memory (HBM) on FPGAs offers significantly more off-chip memory bandwidth compared with traditional DDR memory, which enables new opportunities for accelerating sorting.**

In this paper, we develop Hypersort, a hardware accelerator to accelerate sorting on HBM-enabled FPGA. We use columnsort to merge HBM channels. To support the data communication patterns of Columnsort, we propose several optimizations to reduce external memory (HBM) traffic and hide data communication latency to further improve the overall throughput. We implement our accelerator on a state-of-the-art HBM-enabled FPGA. Experimental results show that our implementation achieves overall sorting throughput of 34 GB/s, which is up to 14.8×, 4.73× and 2.18× faster than the state-of-the-art implementations on CPU, FPGA with external DDR and HBM-enabled FPGA, respectively. The proposed approach demonstrates higher efficiency for merging sorted arrays in HBM channels compared with the state-of-the-art implementation on HBM-enabled FPGA.

***Index Terms*— Parallel Sorting, Columnsort, High-Bandwidth Memory (HBM), FPGA**

I. Introduction

SORTING is a key problem in a broad range of applications, including big data applications, digital signal processing, bioinformatics, compression of grid functions, and large-scale scientific computing [1], [2], [3], [4], [5], [6]. Field Programmable Gate Array (FPGA) is a promising hardware platform for sorting due to its fine-grained data parallelism and customizable memory access pattern [7], [8], [9], [10], [11], [12], [13], [14]. While there are many FPGA accelerators proposed for efficiently executing sorting algorithms, the sorting algorithms are inherently bounded by the available memory bandwidth, preventing them from being further accelerated [8], [9], [10]. Integration of High-Bandwidth Memory (HBM) on FPGA offers new opportunities for improving sorting performance.

High Bandwidth Memory (HBM) is a type of memory device that uses 3D-stacking techniques providing up to 460 GB/s memory bandwidth on a single FPGA device (e.g., Xilinx Alveo U280 [15]). FPGA vendors (e.g., AMD Xilinx, Intel) exploit the Through Silicon Via (TSV) technology to integrate the programmable logic and HBM into a single chip to reduce the memory access latency from HBM. Compared with traditional DDR memory on FPGAs (e.g., Xilinx Alveo U250, U200 [16]) which have up to 77 GB/s, HBM provides 6x more memory bandwidth (up to 460 GB/s).

Despite the massive memory bandwidth of HBM, we face non-trivial design challenges in achieving high-throughput sorting on HBM-enabled FPGA. Unlike traditional DDR memory of FPGA which has up to four memory channels, the HBM memory on FPGA (e.g., Xilinx Alveo U280) has a large number of memory channels (up to 32), which imposes significant challenges not only on the efficiency of sorting algorithm, but also on the scalability of the hardware design. On the one hand, a scalable and load-balanced parallel sorting algorithm should be identified to efficiently exploit the massive memory bandwidth of the HBM. Prior work [17] maps the bucket sort algorithm on HBM-enabled FPGA. However, the uneven data distribution can lead to load imbalance in piplining bucket sort. On the other hand, a hardware-efficient design should be developed to utilize the massive number of memory channels of HBM. Such a design will involve efficient hardware pipelines and interconnection network to communicate data among the HBM channels. For example, in the high-performance design in prior work [18], the resource requirement can quickly exceed the available resources of state-of-the-art FPGA to effectively use the massive data parallelism of HBM.

In this paper, we develop Hypersort, a hardware accelerator to accelerate sorting on HBM-enabled FPGA. We use columnsort to merge HBM channels. To support the data communication patterns of Columnsort (See Section V), we propose various optimizations to further improve the performance. Our main contributions are:

- We propose a hardware design on HBM-enabled FPGA with well-optimized computation pipelines and interconnection network to efficiently execute Columnsort.
- We propose algorithm-specific optimizations that can further improve the overall sorting throughput (Section V), including (1) Optimized all-to-all personalized communication, (2) Fake shifting, (3) Data caching, (4) Step overlapping.
- Through detailed analysis and performance modeling for the proposed mapping technique, we demonstrate that Columnsort based design has higher efficiency of merging

the sorted arrays in the HBM channels compared with merge sort based designs.
- We conduct experiments on Xilinx Alveo U280. The proposed design achieves an overall sorting throughput of 34 GB/s with $14.8\times$, $4.73\times$ and $2.18\times$ speed-up over the state-of-the-art implementations on CPU, FPGA with external DDR, and HBM-enabled FPGA respectively.

II. Background and Related Work

A. Problem Definition

Sorting an array is to rearrange the array in ascending or descending order. The input and output are defined as follow:

- **Input:** The input array is arranged in a 2-D mesh (Figure 1 shows an example) with the columns of data stored in HBM channels. The input array can be fully stored in the HBM channels of FPGA. Note that the input array can have any data distribution. HBM is also used to store the intermediate results.
- **Output:** The sorted data is stored in HBM channels in column major order.

B. HBM-enabled FPGA

A HBM device has multiple memory channels. Each channel is connected to a memory controller and can be accessed through an AXI port. For example, Xilinx Alveo U280 has 2 HBM stacks with each stack having 16 pseudo channels, and a peak memory bandwidth of 460 GB/s with total memory capacity of 8 GB. Table I illustrates the specifications of each pseudo-channel (PC) of Xilinx Alveo U280 with sequential access pattern [15], [17]. The AXI port of a HBM PC can output a 512-bit data packet per clock cycle.

TABLE I: Attributes of a single pseudo channel of the HBM on Xilinx Alveo U280

Parameter	Description
Memory capacity	256 MB
Read-only bandwidth	13 GB/s
Write-only bandwidth	13.1 GB/s
Ideal bandwidth	14.4 GB/s
Read latency	289 ns
Write latency	151 ns

C. Columnsort

The classic Leighton's Columnsort algorithm [19] sorts a N-element input sequence represented as a 2-D mesh (See an example in Figure 1) of size $r \times s$, where r denotes the number of rows and s denotes the number of columns. Columnsort algorithm has a restriction that $r \geq 2 \cdot (s-1)^2$ and s is a divisor of r. For an array of any size, data padding can be used to satisfy the above restriction easily. Columnsort algorithm performs the following eight steps explained with a toy example in Figure 1.

- **Step 1 (Sort along the column):** Each column is sorted independently.
- **Step 2 (Transpose):** The elements of the matrix are accessed in column major order and written back in row major order.
- **Step 3 (Sort along the column):** Each column is sorted independently.
- **Step 4 (Untranspose):** The elements of the matrix are accessed in row major order and written back in column major order.
- **Step 5 (Sort along the column):** Each column is sorted independently.
- **Step 6 (Shift forward by $\frac{r}{2}$ positions):** The columns are shifted forward by $[\frac{r}{2}]$ positions and $-\infty$ and $+\infty$ are inserted at the beginning and end respectively such that the matrix size changes from $r \times s$ to $r \times (s+1)$.
- **Step 7 (Sort along the column):** Each column is sorted independently.
- **Step 8 (Shift backward by $\frac{r}{2}$ positions):** The columns are shifted backward by $\frac{r}{2}$ positions such that the matrix size changes from $r \times (s+1)$ to $r \times s$.

Note that the Steps 1, 3, 5 and 7 involve the sorting within each column. Intuitively, we can implement s parallel processors to sort s columns concurrently and each column can be stored in the local memory of a processor. Steps 2, 4, 6 and 8 involve the data communication among these processors. To mitigate the height and divisibility restrictions in Columnsort, [20] presents 2 modifications to Columnsort - subblock columnsort and slabpose columnsort. Both the modifications can relax the height restriction to $r \geq 4s^{\frac{3}{2}}$ when s is a perfect square and s is a divisor of r. While subblock columnsort can further eliminate the divisibility condition at the cost of a tighter height restriction, $r \geq 6s^{\frac{3}{2}}$.

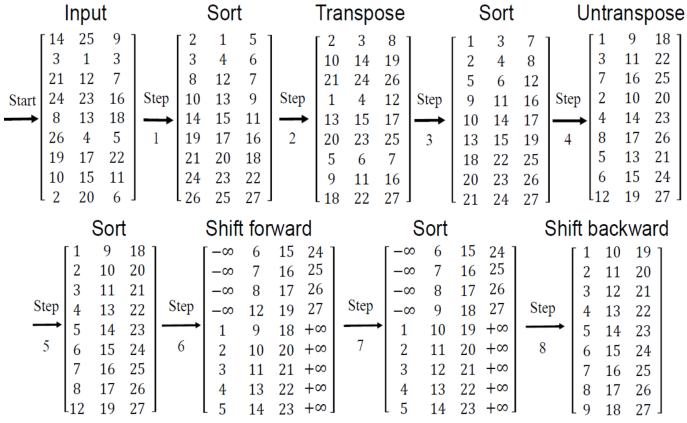

Fig. 1: The eight steps of the Columnsort algorithm explained with a toy example

D. Related Work

In this section, we discuss two major categories of related work. First, there have been many works done on analysis and optimisation of Columnsort algorithm. [21] discusses the design and implementation of Columnsort as an efficient out-of-core sorting algorithm. However, their implementation is not parallelized and they utilise asynchronous I/O and communication calls. Non-overlapping of sorting and communication stages, and the additional time spent for disk read/write increases the latency.

Second, some recent works [1], [22], [23], [24] explore HBM to accelerate various computation tasks on FPGA. Saturating the entire bandwidth offered by the 32 HBM channels is challenging as the scalability of the design may be restricted by the availability of on-chip resources. For example, HiSparse [22], a linear algebra accelerator on HBM could only harness 18 HBM channels as their scalability was limited by routability issues and resource contention. Topsort [1] is a sorting accelerator for executing parallel two-phase merge sort on HBM-enabled FPGAs, achieving the overall throughput of 15.6 GB/s. Although [1] efficiently utilizes the bandwidth of HBM in the 1^{st} phase, their main drawback is that they can only utilize 25% of the HBM bandwidth in the 2^{nd} sorting phase due to the inherent limitations of merge sort algorithm and on-chip resource availability.

III. Merging Sorted HBM Channels

Sorting on HBM-enabled FPGA involves two major stages: (1) sorting the subarray (column) within each HBM channel, and (2) merging the sorted subarrays of all the HBM channels. Sorting the subarray within each HBM channel is similar to sorting an array within a DDR channel, for which many algorithms and hardware designs [7], [8], [10] have been proposed. However, merging the sorted subarrays of all the HBM channels can dominate the overall performance. A major challenge is that the data parallelism provided by all the HBM channels is massive, and directly scaling the prior FPGA designs [8], [10] to match the data rate of all the HBM channels can lead to large hardware consumption. Moreover, the merge tree based approach [1] is data-dependent resulting in bandwidth under-utilization during merging HBM channels. In this section, we analyze bitonic sort, radix sort, merge sort, and columnsort for merging the sorted subarrays of HBM channels. For simplicity, we assume there are p HBM channels and each HBM channel can read/write l data per cycle. To fully utilize the bandwidth of HBM channels, the hardware design should match the data rate of HBM ($p \times l$ data elements per cycle). For example, the HBM on Xilinx Alveo U280 has $p = 32$ HBM channels and each HBM channel can read/write $l = 16$ 32-bit data per cycle.

Bitonic Sort: Bitonic sorting is an efficient algorithm for sorting the data within a single DDR/HBM channel, since the data rate of a single DDR/HBM channel is limited, for example, $l = 16$. To execute Bitonic sort algorithm, Bitonic sorting network is used [7], which has the hardware complexity of $\mathcal{O}(n \log^2 n)$[1], where n is the data rate of off-chip memory denoting the number of input/output data per cycle. For a single HBM channel, the required hardware complexity is $l \times \log^2(l)$. If we directly scale the Bitonic sorting network to match the data rate of p HBM channels, the required hardware complexity is $pl \log^2(pl)$.

Radix Sort: It is an non-comparative algorithm as it groups data elements based on the radix [25]. It does not require hardware comparators. However, the algorithm is neither load balanced nor data oblivious. Since the algorithm depends on the values of the data elements, the bucketing and communication patterns are data dependant. In many real-world applications, skewed data distribution can lead to severe load imbalance, thus degraded performance.

Merge Sort: It uses divide-and-conquer strategy to perform large-scale sorting. Prior work [1] uses a merge tree with Bitnoic merging unit as the basic component for merging the p HBM channels. The hardware design of [1] can efficiently sort data in each individual HBM channel. However, [1] does not exploit all the memory bandwidth when merging the p HBM channels since matching the data rate of p HBM channels will result in large amount of hardware resources. Therefore, [1] builds a small merge tree that only utilizes 25% of the HBM bandwidth for merging $p = 32$ HBM channels.

Columnsort: To support the communication patterns of Columnsort (Steps 2, 4, 6, and 8), we can implement an interconnection network for communicating data among p HBM channels. The interconnection network does not require hardware comparators. For example, [17] develops a large-scale butterfly network for communicating data among p parallel HBM channels which is demonstrated to be lightweight and hardware-efficient. Moreover, with careful optimizations, we are able to completely hide overhead/latency of the data communication steps (See Section V).

To summarize, Columnsort algorithm has the following benefits for merging p HBM channels on FPGA: (1) the required hardware resource for computation units grows linearly w.r.t. p. Only an extra interconnection network is needed which can be made hardware efficient. (2) The sorting steps (Steps 1, 3, 5 and 7) and data communication among columns (Steps 2, 4, 6 and 8) can fully utilize the memory bandwidth of p HBM channels. (3) The number of data communication steps (Steps 2, 4, 6 and 8) is fixed and does not depend on the array size and the number of HBM channels p. (4) Columnsort does not make any assumptions regarding the data distribution. This leads to load-balanced implementation.

IV. Algorithm and Implementation

A. Overview of the proposed system

Our objective is to map Columnsort algorithm on HBM-enabled FPGA. Suppose the HBM memory has $2p$ pseudo channels (PCs). We divide the $2p$ PCs into p HBM groups with each *HBM group* having 2 PCs. For example, AMD Xilinx Alveo U280 has 32 parallel pseudo-channels (PCs) which can be divided into 16 groups using the proposed approach. For a given input array of size N, we convert it into a 2-D mesh of size $\frac{N}{p} \times p$. This 2-D mesh has p columns with each column having N/p elements.

Columnsort involves eight steps (Section II-C). For Steps 1, 3, 5 and 7, each column is sorted using a processing element (PE) that is connected to a HBM group. For Steps 2, 4, 6 and 8, there is an interconnection network (IN) that performs data communication among HBM groups. The overall architecture is shown in Figure 2. Moreover, we perform algorithm-specific

[1]The logarithms are to base 2 unless the base is specified.

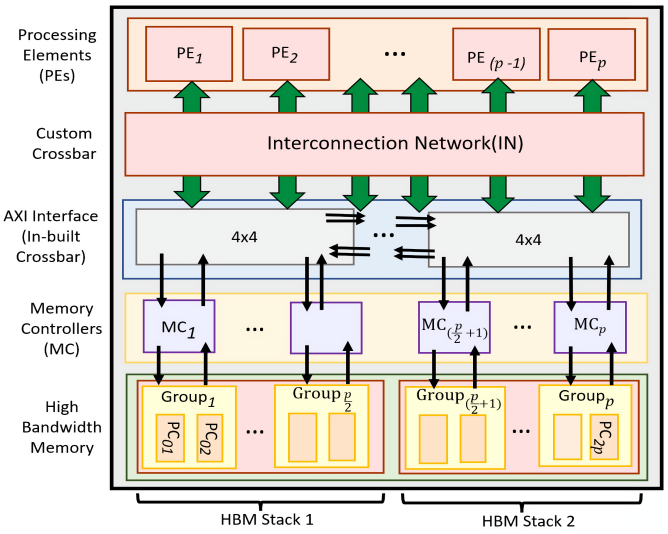

Fig. 2: Overall architecture of the proposed hardware design

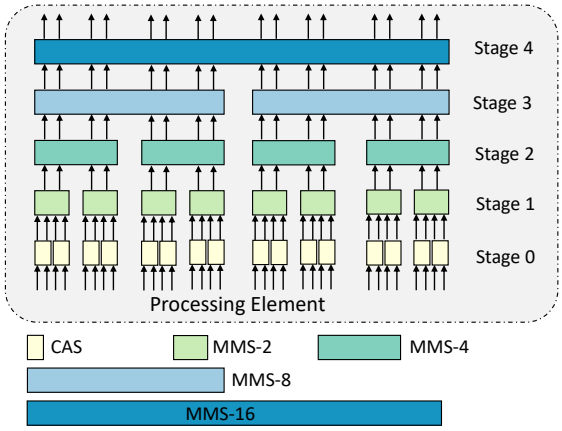

Fig. 3: Diagram of a Processing Element (PE) that can output 16 data per cycle

optimizations discussed in Section V to improve performance and compare our design with the state-of-the-art work in Section VII-D.

B. Algorithm on HBM-enabled FPGA

The parallel Columnsort algorithm on HBM-enabled FPGA is described in Algorithm 1. The p columns of the input 2-D mesh is stored in p HBM groups respectively. For sorting steps (Steps 1, 3, 5 and 7), each PE sorts a column in a HBM group. p PEs sort p columns concurrently. The interconnection network performs data communication in Steps 2, 4, 6 and 8.

Algorithm 1 Parallel Columnsort on HBM-enabled FPGA

Input: The size of the array: N; The number of HBM groups: p
Output: Sorted array

1: Rearrange the input array into a 2-D mesh of r rows and p columns: $r \leftarrow \frac{N}{p}$, $s \leftarrow p$
2: Assign column i to group$_i$ $(1 \leqslant i \leqslant p)$ ▷ Preprocessing
3: **for** i = 1 to p **Parallel do** ▷ Step 1
4: PE_i sorts column i in group i
5: Perform *transpose* using interconnection network ▷ Step 2
6: **for** i = 1 to p **Parallel do** ▷ Step 3
7: PE_i sorts column i in group i
8: Perform *untranspose* using interconnection network ▷ Step 4
9: **for** i = 1 to p **Parallel do** ▷ Step 5
10: PE_i sorts column i in group i
11: Perform *shift forward* using interconnection network ▷ Step 6
12: **for** i = 2 to p **Parallel do** ▷ Step 7
13: PE_i sorts column i in group i
14: Perform *shift back* using interconnection network ▷ Step 8

C. Hardware Modules

Processing Element (PE): As shown in Figure 2, each HBM group is connected to a Processing Element (PE). A PE is implemented as a Merge Tree (MT) (Figure 3) to perform sorting of the column that is stored in a HBM group. In the Merge Tree, each node except the leaf node is implemented as a Massive Merge Sorter (MMS) unit [12] (See Figure 4). We use MMS-l to denote the MMS unit that can output l data per cycle. The leaf node is implemented using a Compare-and-Swap (CAS) unit. We use MT-l to denote the Merge tree that can output l data per cycle. A MT-l can merge l sorted subarrays in parallel. A MT-l consists of $\log_2(l)+1$ stages (from stage 0 to stage $\log_2(l)$) with each stage having single or multiple MMS units. The stage i $(1 \leqslant i \leqslant \log_2(l))$ has $\frac{l}{2^i}$ MMS-2^i units. We use BM-l to denote the Partial Bitonic Merger of $2l$-intput and l-output. A MMS-l unit has two Partial Bitonic Mergers (BM), each reading $2l$ inputs and generating l outputs per cycle: BM$_L$-l and BM$_S$-l. The architectures of BM$_L$-l and BM$_S$-l are shown in Figure 5. The basic component in BM is the Compare-And-Swap (CAS) Unit. BM$_L$-l or BM$_S$-l has $\log(2l)$ stages. While stage 1 has l CAS units, each of other stages has $\frac{l}{2}$ CAS units. Therefore, a MMS-l consists of $l\log(2l)+l$ CAS Units as BM$_L$-l has $\frac{l}{2}\log(2l)+\frac{l}{2}$ CAS units and BM$_L$-l also has $\frac{l}{2}\log(2l)+\frac{l}{2}$ CAS units.

A sorting step (Steps 1, 3, 5, 7) consists of single or multiple sorting *passes*. In one sorting pass, the input elements are streamed from the HBM channels into the leaf buffers of the MT, and the output elements are streamed back into the other HBM channel of the same group. For one sorting pass, the entire input array is read and written from/to HBM once. For example, each group has $\frac{N}{p}$ unsorted elements stored in the HBM channel initially. In the first pass, the MT reads $\frac{N}{p}$ unsorted sub-arrays containing 1 element each from the HBM channel, merges them into $\frac{N}{p*l}$ sorted sub-arrays containing l-elements each and writes back to the HBM channel. Similarly, in the second pass, $\frac{N}{p*l}$ sub-arrays of length l-elements each is streamed into the BMT and $\frac{N}{p*l^2}$ sorted sub-arrays with l^2 elements each is written back to the HBM channel. Thus, using MT-l, it takes $\log_l(\frac{N}{p})$ passes to sort $\frac{N}{p}$ elements.

Interconnection network (IN): The interconnection network is used to permute the data elements in Steps 2 and 4, which have all-to-all data communication pattern. While the built-in crossbar [26] can possibly be used, it has degraded all-to-all

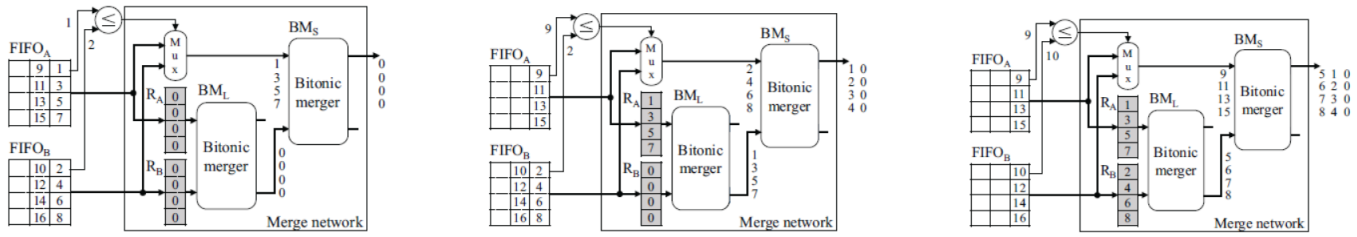

Fig. 4: Diagram of MMS-4 [12] which consists of a BM_L-4 and a BM_S-4

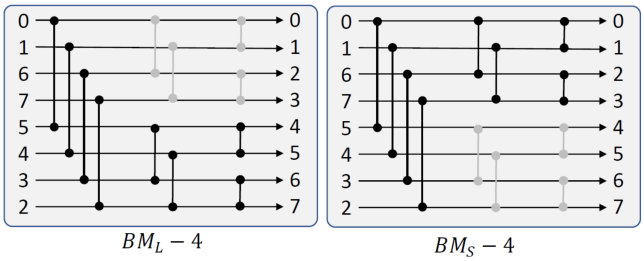

Fig. 5: Diagram of Bitonic Mergers: BM_L-4 and BM_S-4. The hardware switches in grey color are not needed.

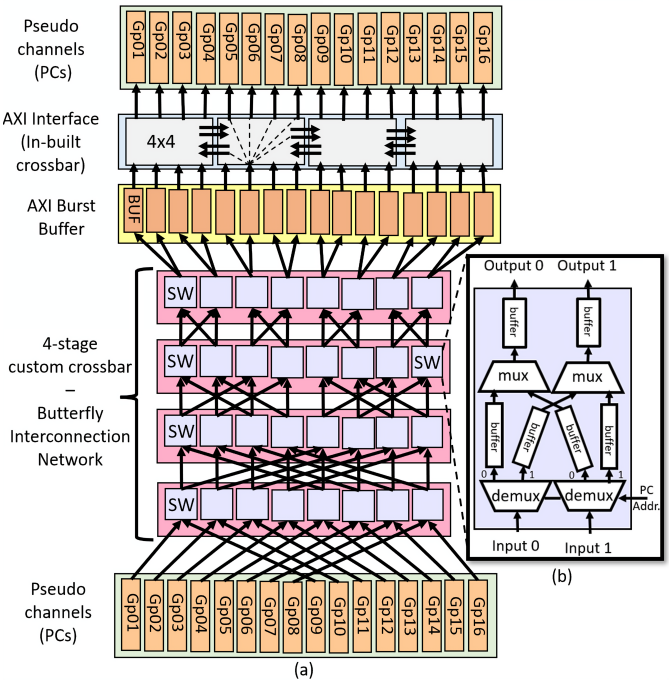

Fig. 6: Overall architecture of the four-stage butterfly interconnection network

data communication throughput due to lateral communications (See [26] for lateral communication). Therefore, we exploit an optimized custom crossbar [17] which is implemented as a multi-stage butterfly interconnection network. As depicted in Figure 6, the fundamental component of the interconnection network is the 2×2 switch. To establish all-to-all communication, we implement $\log p$ stages with $\frac{p}{2}$ 2×2 switching units per stage. Figure 6 illustrates the topology of the custom crossbar. The 2×2 switching unit is constructed as a Mux-Demux switch. Each 2×2 switch reads 2 data elements from the source PC/previous stage switch and writes to the output ports of the destination PC/next stage switch in a round robin fashion.

V. Optimizations

In this section, we introduce algorithm-specific optimizations to improve the sorting throughput and reduce hardware resource consumption. We develop techniques to hide data communication latency by exploiting on-chip memory which also reduces the HBM traffic.

Workload reduction:

Steps 1, 3, 5 and 7 involve sorting the data within each column. [21] depicts that the data should be communicated to the correct column but its placement within the column is immaterial. Hence, by permuting sorted data from each column to contiguous locations in the the destination column, we can reduce the number of sorting passes in the subsequent sorting step. In Step 1, each PE needs to sort a column from scratch. But in Steps 3 and 5, each column already has p sorted sub-arrays, having $\frac{N}{p^2}$ elements each. While Step 7 has 2 sorted sub-arrays, comprising $\frac{N}{2*p}$ elements each. Hence, Steps 3 and 5 require $\lceil \log_l(p) \rceil$ passes, and Step 7 only requires 1 pass to complete sorting.

Optimized all-to-all personalized communication: Step 2 (transpose) and Step 4 (untranspose) involve basic data communication primitive – *all-to-all personalized communication* [27]. For this, each column/group sends $\frac{N}{p^2}$ elements to every other column/group. This all-to-all personalized communication is performed by the interconnection network (IN). However, without careful scheduling, the all-to-all personalized communication can easily lead to congestion in the butterfly network. We proposed an optimized data communication scheduling for Step 2 and Step 4, as shown in Algorithm 2. Using the proposed scheduling, no more than one HBM group will send data to a HBM group concurrently, which eliminates the congestion.

Fake shifting: Steps 6 and 8 involve shifting each data element

Algorithm 2 Scheduling of all-to-all personalized communication for Step 2 (transpose) and Step 4 (untranspose)

Input: HBM group$_i$ ($1 \leqslant i \leqslant p$) has messages $\{\boldsymbol{M}[i][j] : 1 \leqslant j \leqslant p\}$ ($\boldsymbol{M}[i][j]$ contains $\frac{N}{p^2}$ array elements of column i that need to be sent from HBM group$_i$ to HBM group$_j$).

1: **for** i = 1 to p **parallel do**
2: **for** j = 1 to $p-1$ **do**
3: group$_i$ send $\boldsymbol{M}[i][(i+j)\%p]$ to group$_{(i+j)\%p}$

by $\frac{r}{2}$ positions in forward and backward directions respectively. This data shifting can lead to extra memory traffic and requires an additional HBM group to store the extra column. To address the above issue, we propose the *fake shifting* technique. The key idea is that instead of actually performing the shift operation at Step 6 (See Figure 1), we pretend that the shifting is already done for Step 6 and each PE maintains two pointers (physical addresses) that point to the two subarrays of a column. The subarrays are stored in two adjacent HBM groups since the shifting is not performed. Then, in Step 7, the PE directly fetches the subarrays from two HBM groups and merges them into a single sorted array/column. The sorted array/column will be stored back to the corresponding HBM group of that PE. After Step 7, no backward shift (Step 8) is needed since the whole array has already been sorted. Using the proposed fake shifting technique, Steps 6, 7 and 8 are combined into a single Step, which reduces the memory passes (of Steps 6, 7 and 8) from 3 to 1, thus dramatically reducing the memory traffic.

Data caching: For implementing Step 1, the on-chip memory of FPGA can be used to store the intermediate results which can reduce the total memory traffic. Suppose each PE has an on-chip memory of size $S_{\text{on-chip}}$. For the pass 1 of Step 1, every time, the PE fetches $S_{\text{on-chip}}$ unsorted elements from the HBM and stores them in the on-chip memory. When the $S_{\text{on-chip}}$ data elements are fully sorted on-chip, each PE sends the $S_{\text{on-chip}}$ elements back to HBM. Therefore, using the on-chip memory, we can obtain sorted subarrays of size $S_{\text{on-chip}}$ in a single pass. The number of passes for Step 1 can be reduced from $\log_l(\frac{N}{p})$ to $\log_l(\frac{N}{p}) - \log_l(S_{\text{on-chip}}) + 1$ (l is the data parallelism of the merge tree in a PE, See Section IV-C).

Increasing data parallelism in pass 1 of Step 1: Since we exploit data caching optimization for Step 1, for the first pass of Step 1, we store $S_{\text{on-chip}}$ unsorted elements in the on-chip memory. The on-chip memory has higher memory bandwidth than HBM. Therefore, we can exploit the higher memory bandwidth of on-chip memory to increase the data parallelism of a PE to accelerate pass 1 of Step 1. For example, increasing the data parallelism of the Merge Tree in a PE from l to $2l$ can lead to $2\times$ speedup for pass 1 of Step 1.

Step overlapping: The computation steps (Steps 1, 3, 5 and 7) and the communication steps (Steps 2, 4, 6 and 8) can be overlapped. The adjacent computation step and communication step can be overlapped to hide the latency of the communication steps. Specifically, (1) the last pass of Step 1 and Step 2 can be overlapped. When the PE performs the last pass of Step 1, the sorted results can be directly sent to the corresponding HBM group, which is same as performing Step 2. (2) Similarly, Steps 3 and Step 4 can be overlapped. (3) For Steps 6 and Step 8, since we already use the fake shifting technique, Steps 6 and Step 8 are combined into Step 7. Through step overlapping, the latency of Steps 2 and Step 4 can be completely hidden.

VI. Performance and Resource Consumption Modeling

We make the following assumptions for our modeling:

- Input array has size N.
- A single CAS unit in the Merge Tree (MT) consumes hardware resource: S_{CAS}. A single switch in the butterfly network consumes hardware resource: S_{switch}.
- A MT in a PE can output l_{MT} data per cycle and a single HBM PC can also input/ouput l_{PC} data per cycle. To fully exploit the memory bandwidth of HBM, l_{MT} should be greater than or equal to l_{PC}. In our design, $l_{\text{MT}} \geqslant l_{\text{PC}}$.
- The total number of HBM groups is p (p is a power of 2; each HBM group contains two HBM PCs); The total number of PEs is p. For example, for Xilinx Alveo U280, $p = 16$.
- A single PE has an on-chip memory of size $S_{\text{on-chip}}$.

A. Performance modeling

We estimate the performance based on the metrics defined below:

- **Latency:** It is the total execution time taken to sort the input array of size N. We assume the input and output are stored in the HBM.
- **Throughput:** Throughput, measured in GB/s, is the size of the input sequence sorted per second [8].

$$\text{Throughput} = \frac{\text{Size of the input sequence}}{\text{Latency}} \tag{1}$$

Modeling of throughput: The throughput of i^{th} Step (β_{Step_i}) is defined as the size of the input sequence divided by the total execution time taken to complete the i^{th} Step. The overall sorting throughput $\beta_{overall}$ of parallel sorting with 8 Steps can be represented as a function of sorting throughput of the individual Steps is given by Equation 2.

$$\beta_{overall} = \frac{1}{\frac{1}{\beta_{Step_1}} + \frac{1}{\beta_{Step_2}} + ... + \frac{1}{\beta_{Step_8}}} \tag{2}$$

Due to the proposed step overlapping technique, Step 1 and Step 2 are combined into a single step denoted as Step$_{1-2}$. Step 3 and Step 4 are combined into a single step denoted as Step$_{3-4}$. Step 6, Step 7, and Step 8 are combined into a single step denoted as Step$_{6-7-8}$. Therefore, the overall throughput after step overlapping is:

$$\beta_{overall} = \frac{1}{\frac{1}{\beta_{Step_{1-2}}} + \frac{1}{\beta_{Step_{3-4}}} + \frac{1}{\beta_{Step_5}} + \frac{1}{\beta_{Step_{6-7-8}}}} \tag{3}$$

TABLE II: The performance of merging p HBM groups of various designs

Design	Algorithm	# of passes	Data Parallelism	Execution time (clock cyles) for merging p HBM channels
Serial Merge	Merge Sort	1	1	N
Topsort [1]	Merge Sort	1	$4 \times l_{\text{PC}}$	$T_{\text{MergeSort}} = \frac{1}{4} \times \frac{N}{l_{\text{PC}}}$
Our work	Columnsort	$2\lceil \log_{l_{\text{PC}}}(p) \rceil + 1$	$p \times l_{\text{PC}}$	$T_{\text{Columnsort}} = \frac{2\lceil \log_{l_{\text{PC}}}(p) \rceil + 1}{p} \times \frac{N}{l_{\text{PC}}}$

Number of passes: One memory pass is defined as the process of reading the entire array from HBM and writing the entire array back to HBM. During the reading and writing process, the accelerator performs sorting/merging operations for rearranging the array elements. Due to data caching and step overlapping, the number of passes for Step_{1-2} is:

$$NP_{1-2} = \log_{l_{\text{pc}}}(\frac{N}{p}) - \log_{l_{\text{pc}}}(S_{\text{on-chip}}) + 1 \quad (4)$$

Due to the workload reduction optimization, both Step_{3-4} and Step 5 require $\lceil \log_{l_{\text{PC}}}(p) \rceil$ passes each and Step_{6-7-8} requires 1 pass. Therefore, the total number of passes for all the steps is:

$$NP_{\text{total}} = \log_{l_{\text{pc}}}(\frac{N}{p}) - \log_{l_{\text{pc}}}(S_{\text{on-chip}}) + 2\lceil \log_{l_{\text{PC}}}(p) \rceil + 2 \quad (5)$$

Note the Step_{1-2} is to sort the data in each HBM group, and Step_{3-8} is to merge the HBM groups.

Efficiency of merging HBM channels: Merging HBM channels denotes merging p sorted arrays in p HBM group (each HBM group has one sorted array) into one single sorted array in column-major order (See Section II-C. In the proposed design, the total number of passes for merging the HBM group is:

$$NP_{3-8} = NP_{3-4} + NP_5 + NP_{6-7-8} = 2\lceil \log_{l_{\text{PC}}}(p) \rceil + 1 \quad (6)$$

We compare the efficiency of merging p HBM channels with serial merge and the approach in Topsort [1]. For serial merge, the architecture is a merge tree with p leaf nodes that merges p HBM channels which can output 1 data per cycle. We use the serial merge as the baseline for Topsort and our approach. The comparison results are shown in Table II. Topsort [1] implements extra sorting units for merging p HBM groups. Although Topsort only uses 1 pass for merging HBM groups, it only has the data parallelism of $4 \times l_{\text{PC}}$ which is achieved through adding extra computation units. Hypersort requires $2\lceil \log_{l_{\text{PC}}}(p) \rceil + 1$ passes and can fully utilize all the $p \times l_{\text{PC}}$ data parallelism. The comparison of execution time for merging p HBM groups between Topsort and our work is shown in Figure 8. Our columnsort based design is more efficient for merging p HBM groups than the merge sort based design [1] for $p \geqslant 12$ when $l_{pc} = 16$.

B. Resource modeling

Suppose each PE has a merge tree MT-l, there are $\log(2l_{\text{MT}}) \times (\log(2l_{\text{MT}}) + 2) \times l_{\text{MT}}$ CAS units. Therefore, the total amount of hardware resources of p parallel PEs is:

$$S_{\text{PE}} = p \times \log(2l_{\text{MT}}) \times (\log(2l_{\text{MT}}) + 2) \times l_{\text{MT}} \times S_{\text{CAS}}. \quad (7)$$

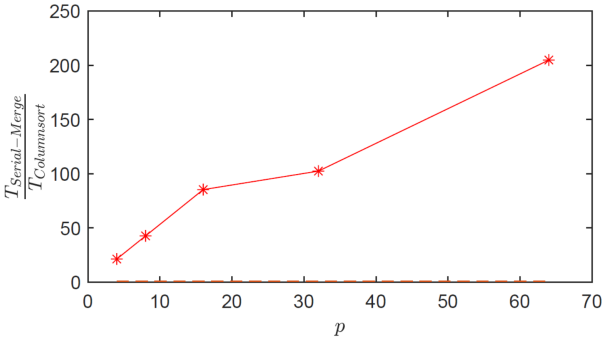

Fig. 7: Comparison of execution time between Serial Merge and Hypersort for merging $p = 4, 8, 16, 32, 64$ HBM groups ($l_{pc} = 16$)

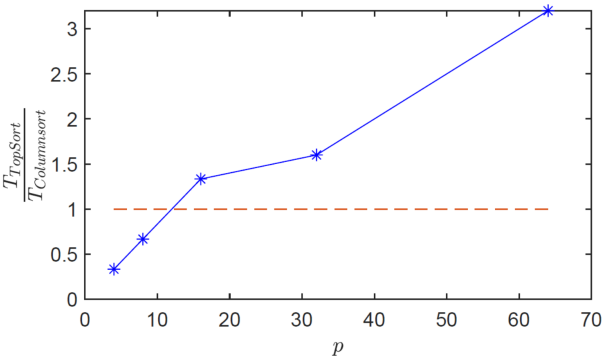

Fig. 8: Comparison of execution time between Topsort and Hypersort for merging $p = 4, 8, 16, 32, 64$ HBM groups ($l_{pc} = 16$)

The total amount of hardware resources of the interconnection network is:

$$S_{\text{IN}} = \log(p) \times \frac{p}{2} \times l_{\text{PC}} \times S_{\text{switch}} \quad (8)$$

Therefore, the total hardware resources consumption of our design is:

$$S_{\text{total}} = S_{\text{PE}} + S_{\text{IN}} \quad (9)$$

According to the above analysis, the resource consumption of PEs, $S_{\text{PE}} = \mathcal{O}(pl\log^2(l))$ grows linearly with the number of HBM channels p. The interconnection network is also resource-efficient $\mathcal{O}(pl\log(p))$ with respect to p (See [17] for the detailed evaluation for the resource efficiency of the butterfly network).

VII. Experiments

A. Implementation Details

We implement our design using Verilog HDL on Xilinx Alveo U280 FPGA Board as the target platform. We implement 16 PEs ($p = 16$) to utilize the 32 HBM channels. Each HBM channel can input/output 16 32-bit data per cycle ($l_{\text{PC}} = 16$). We use BRAM for the implementation of data buffers in the switches of the interconnection network. Further, we use URAM [26], a high capacity on-chip memory for the implementation of the data caching technique discussed in Section V. We perform Synthesis and Place & Route using Vivado 2021.1. The reported resource utilization is shown in Table IV. We also build a cycle-accurate simulator for evaluation. The reported throughput is obtained through simulation on the input data.

B. Benchmarks and Baselines

Benchmark: We evaluate our design using arrays up to 4 GB (half the memory capacity of HBM). The entire input sequence is stored in the HBM.

Baseline Sorters: Table III shows the platform specifications of the baselines with which we compare our design. Section VII-(D) evaluates the performance of our design and compares our work with the state-of-the-art parallel sorting implementations on CPU and FPGA.

C. Resource Utilization

We implement $p = 16$ parallel PEs with each PE having a Merge Tree (MT) that accepts 16 input elements ($l = 16$) and outputs 16 elements per clock cycle. For the Interconnection Network, we implement a four-stage butterfly network. Table IV shows the breakdown of the resource utilization. The total resource consumption of the entire design comprises of the resource consumption of 16 PEs and butterfly interconnection network.

D. Breakdown Analysis

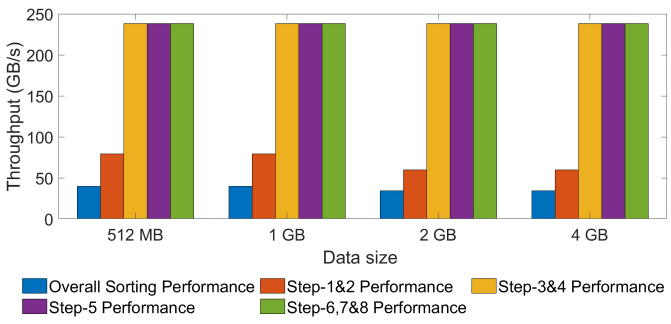

Fig. 9: The overall sorting performance for data sizes 512 MB, 1 GB, 2 GB and 4 GB. The overall sorting throughput is calculated as the total data size divided by the total latency.

The performance of each Step is shown in Figure 9. Note that the breakdown analysis only includes 4 Steps – Step_{1-2}, Step_{3-4}, Step_{5} and Step_{6-7-8} as explained in Section VI-A. Since HBM access is half-duplex, in a HBM group, one channel is used for reading and the other channel is used for writing data. The overall performance of our algorithm sorting 4 GB data with a data width of 32 bits per data element is 34 GB/s where, Step_{1-2} has sorting throughput of 59.6 GB/s and the other Steps have a sorting throughput of 238.4 GB/s. The overall throughput of our design is bounded by the performance of the Step with the lowest throughput (Step_{1-2}). We observe that for 4 GB data, Step_{1-2} runs for 6 passes and each pass reads and writes to the HBM channels simultaneously.

E. Impact of Optimizations

We perform ablation study to demonstrate the impact of various proposed optimizations:

Fake Shifting Through this optimization, we circumvent the communication overhead for $\frac{2*N}{16^2}$ cycles which will otherwise be used for streaming the data elements in Steps 6 and 8. Thus, we achieve nearly 20% speed-up in overall sorting throughput.

Data caching We observed that the on-chip memory in Alveo U280 can accommodate data elements upto 32 MB i.e., 2 MB data elements/group. The data parallelism for the Sorting Step-1 can be scaled up by increasing the number of parallel Merge Trees (MT) until the on-chip resources become the bottleneck. Given the available on-chip resources, we can scale up the data parallelism of a MT up to 32 and realize around 30% improvement in overall sorting throughput.

Step overlapping Through step-overlapping, consecutive Steps can be overlapped, which can effectively reduce overall latency. Based on our experiments, by adopting techniques to hide latency, we obtain up to 22.2% improvement in sorting throughput compared with implementing Columnsort without overlapping consecutive Steps.

F. Efficiency of Merging HBM channels

While TopSort [1] is very efficient for sorting the data in each HBM channel, their overall throughput [1] is bounded by the performance in phase 2. In phase 2 of [1], some merge trees are idle and it uses only 25% of the HBM bandwidth thereby limiting the overall sorting throughput of phase 2 to be 38 GB/s. In our design, we implement Steps 3-8 of columnsort instead of the final merging phase (Phase-2). (1) We have fixed number of sorting passes involved in Steps 3-8, which is 3 (as $p = 16$). (2) Steps 3-8 of Columnsort involves repeated sorting and data movement (transpose, untranspose and shifting) within the groups which can fully utilise the HBM bandwidth. Therefore, the proposed columnsort based design is efficient for merging the sorted arrays in parallel HBM channels, and it does not dependent on input data.

G. Scalability Analysis

Table VI shows the resource breakdown analysis on varying l_{MT} and p at a design frequency of 250MHz.

Scalability in Input Size: We study the scalability of our design across a range of input size varying from 512MB to 4GB as shown in Figure 9. Figure 9 shows a performance drop when data size is increased from 1GB to 2GB because 2GB

TABLE III: Comparison with state-of-the-art sorters. Dashes ('-') indicates no reported result.

Sorter	Platform	Peak Bandwidth	Algorithm	Off-chip Memory	Frequency	Overall Throughput
PARADIS [28]	CPU – Intel Xeon (E7-8837)	-	Radix sort	DDR memory	2.66 GHz	2.3 GB/s
Bonsai [8]	FPGA – Virtex UltraScale+ VU9P FPGA	32 GB/s	Merge sort	4 DDR channels	250 MHz	7.1 GB/s
Samplesort [10]	CPU+FPGA – Xilinx UltraScale+ VU9P FPGA	32 GB/s	Bucket sort	4 DDR channels	250 MHz	7.2 GB/s
TopSort [1]	FPGA– Xilinx Alveo U280 FPGA	460 GB/s	Merge sort	32 HBM channels	214 MHz	15.6 GB/s
Hypersort	FPGA – Xilinx Alveo U280 FPGA	460 GB/s	Columnsort	32 HBM channels	250 MHz	34 GB/s

TABLE IV: Resource utilization of the proposed design on Xilinx Alveo U280 Board ($l_{\text{MT}} = 32$)

Component	**LUTs**	**FFs**	**BRAMs**	**URAMs**
Available	1,300K	2,600K	2016	960
16 PEs	688K	917K	0	960
Percentage	52.93%	35.27%	0%	100%
Interconnection Network	189K	305K	248	0
Percentage	14.54%	11.73%	12.3%	0%
Total	877K	1222K	248	960
Percentage	67.5%	47.02%	12.3%	100%

TABLE V: Impact on the throughput with increased data parallelism and clock frequency

Parameters	Frequency	Throughput
$l_{\text{MT}} = 16$	200MHz	19.07 GB/s
$l_{\text{MT}} = 16$	250MHz	23.84 GB/s
$l_{\text{MT}} = 32$	250MHz	34 GB/s

takes 6 sorting passes in Step_{1-2} while sorting 1GB takes 5 passes.

Scalability in Data Parallelism l_{MT} (See Section V): Figure 10 shows that overall sorting throughput increases superlinearly with linear increase in data parallelism. However, scaling in performance is bounded by the limited on-chip resources.

Scalability in Number of HBM groups p: When we fix the size of input array, we observe a linear increase in overall sorting throughput with increasing of p as shown in Figure 11.

TABLE VI: Overall resource utilization on Xilinx Alveo U280 Board when varying l_{MT} and p

	LUTs	FFs	BRAMs	URAMs
$p = 16$, $l_{\text{MT}} = 16$	430K (33.07%)	626K (24.07%)	248 (12.3%)	960 (100%)
$p = 16$, $l_{\text{MT}} = 32$	877K (67.5%)	1222K (47.02%)	248 (12.3%)	960 (100%)
$p = 8$, $l_{\text{MT}} = 32$	759K (58.38%)	1039K (39.7%)	93 (4.7%)	960 (100%)
$p = 4$, $l_{\text{MT}} = 32$	712K (54.8%)	956K (36.75%)	35 (1.6%)	960 (100%)

H. Comparison with the state-of-the-art

To evaluate the performance of our design, we compare it with the state-of-the-art work. Table III lists the platform specifications of the baselines and the comparison results. We compare our work with the following baselines:

- PARADIS [28], a state-of-the-art CPU sorter, implements the parallel in-place radix sort algorithm to achieve the overall sorting throughput of 2.3 GB/s. They propose speculative permutation and distributive-adaptive load balancing method to minimize the overall sorting time. PARADIS suffers from performance degradation due to the load imbalance caused by highly skewed data distribution.
- Samplesort [10], an FPGA sorter implementing bucket sort, offers overall sorting throughput of 7.2 GB/s. It uses the CPU for initial coarse-grained partitioning operation (sampling and bucketing) which limits its scalability and efficiency.
- Bonsai [8] presents an adaptive sorting solution implemented on Amazon AWS EC2 F1 instance to achieve a throughput of 7.1 GB/s. However, the scalability of the merge units is restricted by the DRAM bandwidth.

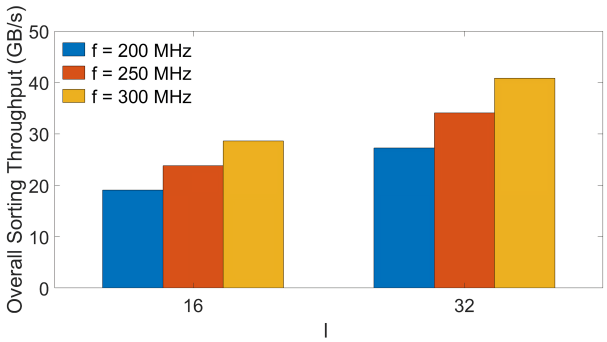

Fig. 10: The overall sorting performance when varying data parallelism (l_{MT}) and frequency (through simulation) with $p = 16$

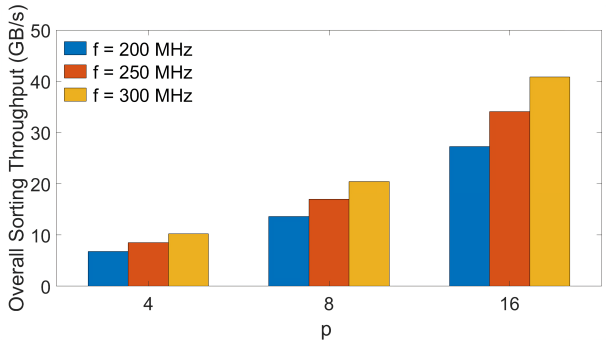

Fig. 11: The overall sorting performance when varying the number of HBM groups (p) and frequency (through simulation)

- TopSort [1] is the state-of-the-art accelerator on HBM-enabled FPGA. It executes a 2-phase merge sorting algorithm using a parallel merge tree, achieving an overall throughput of 15.6 GB/s. However, the 2^{nd} sorting phase in TopSort has idle merge trees near the root of the merge units which restricts the throughput. Moreover, additional resources are required to implement the extra logic to form wider merge trees in phase 2.

Hypersort achieves $2.18\times$ speed-up in overall sorting throughput compared to TopSort [1].

Hypersort exhibits better overall sorting performance primarily due to the proposed optimizations: workload reduction and step overlapping which effectively reduce latency and improve the overall sorting throughput while data caching, fake shifting and implementation of customized crossbar alleviate memory congestion to enhance bandwidth utilization. Columnsort algorithm's inherent load balancing and logic reuse property further reduces resource consumption and improves the scalability of our design. Thus we observe that Hypersort is $14.8\times$ and $4.73\times$ faster than state-of-the-art sorters implemented on CPU [28] and FPGA with external DDR [8].

VIII. Conclusion and Future Work

In this paper, we presented a novel approach to map Columnsort algorithm using well-optimized computation pipelines and hardware-efficient interconnection network on HBM-enabled FPGA to achieve high-throughput sorting. We further adopted several algorithm-specific optimizations to enhance the sorting throughput. Experimental results show our optimized design yields $14.8\times$, $4.73\times$ and $2.18\times$ speed-up compared with state-of-the-art implementations on CPU, FPGA with external DDR and HBM-enabled FPGA. The main advantage of the proposed columnsort based design is that it can efficiently merge the sorted arrays in parallel HBM channels. We expect the columnsort based design can achieve higher speedup on the future FPGA devices with more HBM channels. We plan to improve the design along the following directions:

Algorithm optimization: We plan to apply columnsort for sorting the data within each HBM channel. We can view the stored data in a HBM channel to have multiple columns. Thereby, we can use a smaller number of comparators with hardware-efficient interconnection to sort the data within each column, which can further reduce hardware consumption and increase the scalability of the design.

Hardware optimization: The interconnection network is the key to perform the data communication for columnsort. In this paper, we exploit the optimized butterfly network in a prior work [17] which is implemented using High-level Synthesis (HLS). In future, we plan to develop optimized interconnection network using Verilog HDL which can further reduce the hardware consumption.

Acknowledgment

This work is supported by the National Science Foundation (NSF) under grants CCF-1919289 and OAC-2209563. Soundarya Jayaraman was supported by the IUSSTF-Viterbi Program by the Viterbi School of Engineering, University of Southern California (USC) and the Indo-US Science and Technology Forum (IUSSTF).

References

[1] Weikang Qiao, Licheng Guo, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. Topsort: A high-performance two-phase sorting accelerator optimized on hbm-based fpgas. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–1, 2022.

[2] Vasily Kasilov, Pavel Drobintsev, and Nikita Voinov. High-performance genome sorting program. *Procedia Computer Science*, 193:464–473, 2021. 10th International Young Scientists Conference in Computational Science, YSC2021, 28 June – 2 July, 2021.

[3] Seth H. Pugsley, Arjun Deb, Rajeev Balasubramonian, and Feifei Li. Fixed-function hardware sorting accelerators for near data mapreduce execution. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 439–442, 2015.

[4] Ryohei KOBAYASHI and Kenji KISE. A high performance fpga-based sorting accelerator with a data compression mechanism. *IEICE Transactions on Information and Systems*, E100.D(5):1003–1015, 2017.

[5] I.D. Scherson and S. Sen. Parallel sorting in two-dimensional vlsi models of computation. *IEEE Transactions on Computers*, 38(2):238–249, 1989.

[6] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, page 417–432, 2017.

[7] Sruja Siriyal Ren Chen and Viktor 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*, pages 240–249, 2015.

[8] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. Bonsai: High-performance adaptive merge tree sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 282–294, 2020.

[9] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. An adaptable high-throughput fpga merge sorter for accelerating database analytics. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 65–72, 2020.

[10] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. Fpga-accelerated samplesort for large data sets. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 222–232, 2020.

[11] Sang-Woo Jun, Shuotao Xu, and Arvind. Terabyte sort on fpga-accelerated flash storage. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–24, 2017.

[12] Makoto Saitoh, Elsayed A. Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. 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)*, pages 197–204, 2018.

[13] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. Cost-effective and high-throughput merge network: Architecture for the fastest fpga sorting accelerator. 44(4):8–13, 2017.

[14] Dirk Koch and Jim 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*, page 45–54, 2011.

[15] Xilinx. Alveo u280 data center accelerator card data sheet. https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds963-u280.pdf.

[16] Xilinx. Alveo u200 and u250 data center accelerator cards data sheet. https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds962-u200-u250.pdf.

[17] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. Hbm connect: High-performance hls interconnect for fpga hbm. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 116–126, 2021.

[18] Ren Chen, Sruja Siriyal, and Viktor 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*, pages 240–249, 2015.

[19] Tom Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, page 71–80, 1984.

[20] Geeta Chaudhry and Thomas H. Cormen. Stupid columnsort tricks. 2003.

[21] Geeta Chaudhry, Thomas H. Cormen, and Leonard F. Wisniewski. Columnsort lives! an efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, page 169–178, 2001.

[22] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru 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*, page 54–64, 2022.

[23] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[24] Yang Yang, Sanmukh R Kuppannagari, and Viktor K Prasanna. A high throughput parallel hash table accelerator on hbm-enabled fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 148–153. IEEE, 2020.

[25] Xilinx. Non-comparison algorithms. https://pages.cs.wisc.edu/~paton/readings/Old/fall01/LINEAR-SORTS.html#youtry1.

[26] Xilinx. Alveo U280 Data Center Accelerator Card-User guide.

[27] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

[28] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. Paradis: An efficient parallel algorithm for in-place radix sort. *Proc. VLDB Endow.*, 8:1518–1529, 2015.