HBM Connect: High-Performance HLS Interconnect for FPGA HBM

Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong
Computer Science Department, University of California, Los Angeles
{ykchoi,chiyuze}@cs.ucla.edu,{wkqiao2015,nikola.s}@ucla.edu,cong@cs.ucla.edu

ABSTRACT

With the recent release of High Bandwidth Memory (HBM) based FPGA boards, developers can now exploit unprecedented external memory bandwidth. This allows more memory-bounded applications to benefit from FPGA acceleration. However, fully utilizing the available bandwidth may not be an easy task. If an application requires multiple processing elements to access multiple HBM channels, we observed a significant drop in the effective bandwidth. The existing high-level synthesis (HLS) programming environment had limitation in producing an efficient communication architecture. In order to solve this problem, we propose HBM Connect, a high-performance customized interconnect for FPGA HBM board. Novel HLS-based optimization techniques are introduced to increase the throughput of AXI bus masters and switching elements. We also present a high-performance customized crossbar that may replace the built-in crossbar. The effectiveness of HBM Connect is demonstrated using Xilinx's Alveo U280 HBM board. Based on bucket sort and merge sort case studies, we explore several design spaces and find the design point with the best resource-performance trade-off. The result shows that HBM Connect improves the resource-performance metrics by 6.5X–211X.

KEYWORDS

High Bandwidth Memory, high-level synthesis, field-programmable gate array, on-chip network, performance optimization

ACM Reference Format:
Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431920.3439301

1 INTRODUCTION

Although field-programmable gate array (FPGA) is known to provide a high-performance and energy-efficient solution for many applications, there is one class of applications where FPGA is generally known to be less competitive: memory-bound applications. In a recent study [8], the authors report that GPUs typically outperform FPGAs in applications that require high external memory bandwidth. The Virtex-7 690T FPGA board used for the experiment reportedly has only 13 GB/s peak DRAM bandwidth, which is much smaller than the 290 GB/s bandwidth of the Tesla K40 GPU board used in the study (even though the two boards are based on the same 28 nm technology). This result is consistent with comparative studies for earlier generations of FPGAs and GPUs [9, 10]—FPGAs traditionally were at a disadvantage compared to GPUs for applications with low reuse rate. The FPGA DRAM bandwidth was also lower than the CPUs—Sandy Bridge E5-2670 (32 nm, similar generation as Virtex-7 in [8]) has a peak bandwidth of 42 GB/s [21].

But with the recent emergence of the High Bandwidth Memory 2 (HBM2) [15] FPGA boards, there is a good chance that future FPGAs can compete with GPUs to achieve higher performance in memory-bound applications. HBM benchmarking works [19, 27] report that Xilinx's Alveo U280 [28] (two HBM2) provides HBM bandwidth of 422–425 GB/s, which approaches that of Nvidia's Titan V GPU [23] (650 GB/s, three HBM2). Similar numbers are reported for Intel's Stratix 10 MX [13] as well. Since FPGAs already have advantage over GPUs in terms of its custom datapath and the custom data types [10, 22], enhancing external memory bandwidth with HBM could allow FPGAs to accelerate a wider range of applications.

The large external memory bandwidth of HBM originates from multiple independent HBM channels (e.g., Fig. 1). To take full advantage of this architecture, we need to determine the most efficient way to transfer data from multiple HBM channels to multiple PEs. It is worth noting that the Convey HC-1ex platform [2] also has multiple (64) DRAM channels like the FPGA HBM boards. But unlike Convey HC-1ex PEs that issue individual FIFO requests of 64b data, HBM PEs are connected to 512b AXI bus interface. Thus, utilizing the bus burst access feature has a large impact on the performance of FPGA HBM boards. Also, the Convey HC-1ex has a pre-synthesized full crossbar between PEs and DRAM, but FPGA HBM boards require programmers to customize the interconnect.

Table 1: Effective bandwidth of memory-bound applications on Alveo U280 using Vivado HLS and Vitis tools

Application	PC #	KClk (MHz)	EffBW (GB/s)	EffBW/PC (GB/s)
MV Mult	16	300	211	13.2
Stencil	16	293	206	12.9
Bucket sort	16	277	65	4.1
Merge sort	16	196	9.4	0.59

In order to verify that we can achieve high performance on an FPGA HBM board, we have implemented several memory-bound

FPGA '21, February 28-March 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439301

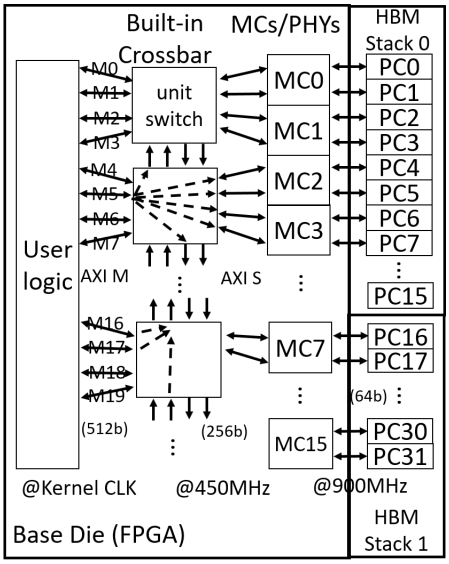


Figure 1: Alveo U280 Architecture

applications on Alveo U280 (Table 1). We were not able to complete the routing for all 32 channels, so we used the next largest power-of-2 HBM pseudo channel (PC) of 16. The kernels are written in C (Xilinx Vivado HLS [31]) for ease of programming and a faster development cycle [17]. For dense matrix-vector (MV) multiplication and stencil, the effective bandwidth per PC is similar to the board's sequential access bandwidth (Section 4.1.1). Both applications can evenly distribute the workload among the available HBM PCs, and their long sequential memory access pattern allows a single processing element (PE) to fully saturate an HBM PC's available bandwidth.

However, the effective bandwidth is far lower for bucket and merge sort. In bucket sort, a PE distributes keys to multiple HBM PCs (one HBM PC corresponds to one bucket). In merge sort, a PE collects sorted keys from multiple HBM PCs. Such an operation is conducted in all PEs—thus, we need to perform all PEs to all PCs communication. Alveo U280 provides an area-efficient built-in crossbar to facilitate this communication pattern. But, as will be explained in Section 6.1, enabling external memory burst access to multiple PCs in the current high-level synthesis (HLS) programming environment is difficult. Instantiating a burst buffer is a possible option, but we will show this leads to high routing complexity and large BRAM consumption (details to be presented in Section 6.2). Also, shared links among the built-in switches (called lateral connections) become a bottleneck that limits the effective bandwidth (details to be presented in Section 4.2).

This paper proposes HBM Connect—a high-performance customized interconnect for FPGA HBM board. We first evaluate the performance of Alveo U280 built-in crossbar and analyzes the cause of bandwidth degradation when PEs access several PCs. Next, we propose a novel HLS buffering scheme the increases the effective bandwidth of the built-in crossbar and consumes fewer BRAMs. We also present a high-performance custom crossbar architecture to remove the performance bottleneck from lateral connections. As will be demonstrated in the experimental section, we found that it is sometimes more efficient to completely ignore the built-in crossbar and only utilize our proposed customized crossbar architecture. The proposed design is fully compatible with Vivado HLS C syntax and does not require RTL coding.

The contribution of this paper can be summarized as follows:

- A BRAM-efficient HLS buffering scheme that increases the AXI burst length and the effective bandwidth when PEs access several PCs.
- An HLS-based solution that increases the throughput of a 2×2 switching element of customized crossbar.
- A design space exploration of customized crossbar and AXI burst buffer that finds the best area-performance trade-off in HBM many-to-many unicast environment.
- Evaluation of the built-in crossbar on Alveo U280 and analysis of its performance.

The scope of this paper is currently limited to Xilinx's Alveo U280 board, but we plan to extend it to other Xilinx and Intel HBM boards in the future.

2 BACKGROUND

2.1 High Bandwidth Memory 2

High Bandwidth Memory [15] is a 3D-stacked DRAM designed to provide a high memory bandwidth. There are 2~8 HBM dies and 1024 data I/Os in each stack. The HBM dies are connected to a base logic die using Through Silicon Via (TSV) technology. The base logic die connects to FPGA/GPU/CPU dies through an interposer. The maximum I/O data rate is improved from 1 Gbps in HBM1 to 2 Gbps in HBM2. This is partially enabled by the use of two pseudo channels (PCs) per physical channel to hide the latency [13, 16]. Sixteen PCs exist per stack, and we can access PCs independently.

2.2 HBM2 FPGA Platforms and Built-In Crossbar

Intel and Xilinx have recently released HBM2 FPGA boards: Xilinx's Alveo U50 [29], U280 [28], and Intel's Stratix 10 MX [13]. These boards consist of an FPGA and two HBM2 stacks (8 HBM2 dies). The FPGA and the HBM2 dies are connected through 32 independent PCs. Each PC has 256MB of capacity (8 GB in total).

In Stratix 10 MX (early-silicon version), each PC is connected to the FPGA PHY layer through 64 data I/Os that operates at 800MHz (double data rate). The data communication between the kernels (user logic) and the HBM2 memory is managed by the HBM controller (400MHz). AXI4 [1] and Avalon [14] interfaces (both with 256 data bitwidth) are used to communicate with the kernel side. The clock frequency of kernels may vary (capped at 450MHz) depending on its complexity. Since the frequency of HBMCs is fixed to 400MHz, rate matching (RM) FIFOs are inserted between the kernels and the memory controllers.

In Xilinx Alveo U280, the FPGA is composed of three super logic regions (SLRs). The overall architecture of U280 is shown in Fig. 1 The FPGA connects to the HBM2 stacks on the bottom SLR (SLR0). The 64b data I/Os to the HBM operate at the frequency of 900 MHz (double data rate). The data transaction is managed by the HBM

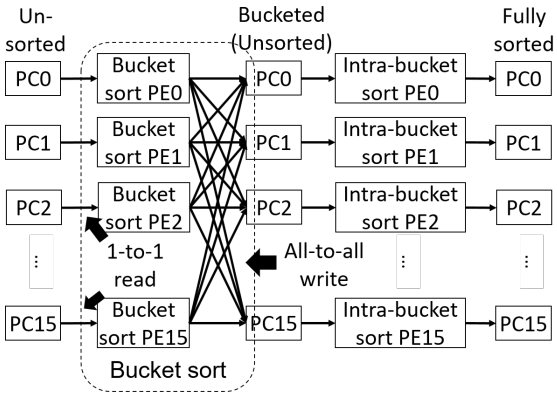


Figure 2: Bucket sort application

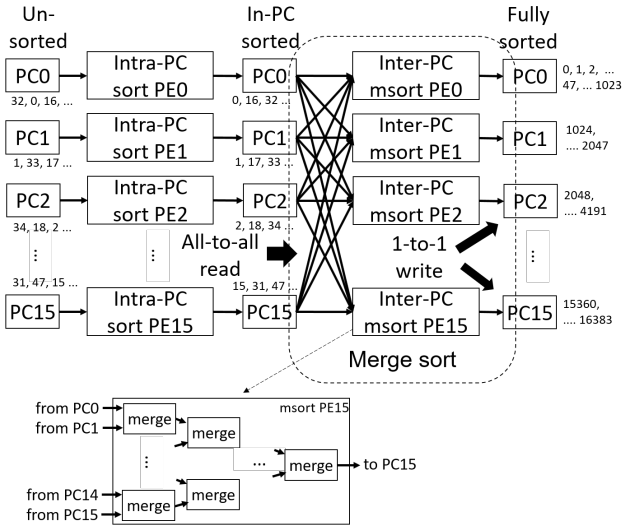


Figure 3: Merge sort application

memory controllers (MCs). A MC communicates with the user logic (kernel) via a 256b AXI3 slave interface running at 450 MHz [30]. The user logic has a 512b AXI3 master interface, and the clock frequency of the user logic is capped at 300 MHz. The ideal memory bandwidth is 460 GB/s (= 256b * 32PCs * 450MHz = 64b * 32PCs * 2 * 900MHz).

Four user logic AXI masters can directly communicate with any of the four adjacent PC AXI slaves through a fully-connected unit switch. For example, the first AXI master (M0) has direct connections to PCs 0–4 (Fig. 1). If an AXI master needs to access non-adjacent PCs, it can use the lateral connections among the unit switches—but the network contention may limit the effective bandwidth [30]. For example in Fig. 1, M16 and M17 AXI masters and the lower lateral AXI master may compete with each other to use the upper lateral AXI slave for communication with PC 0–15. Note that each AXI master has a connection to four PC AXI slaves and two lateral connections (see M5 in Fig. 1).

The thermal design power (TDP) of Alveo U280 is 200W. Note that Alveo U280 also has traditional DDR DRAM—but we decided not to utilize the traditional DRAM because the purpose of this paper is to evaluate the HBM memory. We refer readers to the work in [19, 27] for comparison of HBM and DDR memory and also the work in [20] for optimization case studies on heterogeneous external memory architectures.

2.3 Case Studies

In order to quantify the resource consumption and the performance of HBM interconnect when PEs access multiple PCs, we select two applications for case studies: bucket sort and merge sort. A bucket sort PE *writes* to multiple PCs, and a merge sort PE *reads* from multiple PCs. These applications also have an ideal characteristic of accessing each PC in a sequential fashion—allowing us to analyze the resource-performance trade-off more clearly.

2.3.1 Bucket Sort. Arrays of unsorted keys are stored in input PCs. A bucket PE sequentially reads the unsorted keys from each input PC and classify them into different output buckets based on the value of the keys. Each bucket is stored in a single HBM PC, and this allows a second stage of sorting (e.g., with merge sort) to work independently on each bucket. Several bucket PEs may send their keys to the same PC—thus, all-to-all unicast communication architecture is needed for write as shown in Fig. 2. Since the keys within a bucket does not need to be in a specific order, we combine all the buckets in the same PC and write the keys to the same output memory space.

Since our primary goal is to analyze and explore the HBM PE-PC interconnect architecture, we make several simplifications on the sorter itself. We assume a key is 512b long. We also assume that the distribution of keys is already known, and thus we preset a splitter value that divides the keys into equal-sized buckets. Also, we do not implement the second-stage intra-bucket sorter—the reader may refer to [3, 12, 25, 26] for high-performance sorters that utilize the external memory.

We limit the number of used PCs to 16 for two reasons. First, we were not able to utilize all 32 PCs due to routing congestion (more details in Section 4.1.1). Second, we wanted to simplify the architecture by keeping the number of used PCs to the power of two.

2.3.2 Merge Sort. In contrast to the bucket sort application which sends the data to a PC bucket before sorting within a PC bucket, we can also sort the data within a PC first and then collect and merge the data among different PCs. Fig. 3 demonstrates this process. The intra-PC sorted data is sent to one of the PEs depending on the range of its value, and each PE performs merge sort on the incoming data. Each PE reads from 16 input PCs and writes to one PC. This sorting process is a hybrid of bucketing and merge sort—but for convenience, we will simply refer to this application as merge sort in the rest of this paper.

This application requires a many-to-many unicast architecture between PCs and PEs for data read, and a one-to-one connection is needed for data write. It performs both reading and writing in a sequential address. We make a similar simplification as we did for the bucket sort—we assume 512b key and equal key distribution, and we omit the first-stage intra-PC sorter.

```
                        <kernel file>
1  void bsort(ap_uint<512>* pe0_pc0, pe0_pc1, ... pe0_pc15){
2    #pragma HLS INTERFACE m_axi port=pe0_pc0 bundle=M0
3    #pragma HLS INTERFACE m_axi port=pe0_pc1 bundle=M0
4    ......
5    #pragma HLS INTERFACE m_axi port=pe0_pc15 bundle=M0
6             //all 16 ports mapped to the same AXI master M0
7  #pragma HLS dataflow
8    key_write( pe0_pc0, pe0_pc1, .. pe0_pc15 );
9  }
10
11 void key_write( hls::stream< ... > & in_fifo,
12             ap_uint<512>* pe0_pc0, pe0_pc1, ... pe0_pc15){
13   while(...){
14 #pragma HLS pipeline II=1
15     {data, bucket_id} = in_fifo.read();
16     switch( bucket_id ){
17       case 0 : pe0_pc0[addr0++] = data; break;
18       case 1 : pe0_pc1[addr1++] = data; break;
19       ...
20       case 15 : pe0_pc15[addr15++] = data; break;
21 } }

                          <Makefile>
22 CLFLAGS += --sp bucket_1.pe0_pc0:HBM[0]        //M0 to PC0
23 CLFLAGS += --sp bucket_1.pe0_pc1:HBM[1]        //M0 to PC1
24 ......
25 CLFLAGS += --sp bucket_1.pe0_pc15:HBM[15]     //M0 to PC15
```

Figure 4: Conventional HLS coding style to send keys to multiple output PCs (buckets) using the built-in crossbar

2.4 Conventional HLS Programming for Bucket Sort

We program the kernel and host in C using Xilinx's Vitis [33] and Vivado HLS [31] tools. We employ dataflow programming style (C functions executing in parallel and communicating through streaming FIFOs) for kernels to achieve high throughput with small BRAM consumption [31].

Alveo U280 and Vivado HLS offer a particular coding style to access multiple PCs. An example HLS code for bucket sort is shown in Fig. 4. The output write function `key_write` reads an input data and data's bucket ID (line 15), and it writes the data to the function argument that corresponds to the bucket ID (lines 17 to 20). We can specify the output PC (bucket ID) of the function arguments in Makefile (lines 22 to 25). Notice that a common bundle (`M0`) was assigned to all function arguments (lines 2 to 5). A *bundle* is a Vivado HLS concept that corresponds to an AXI master. That is, `key_write` uses a single AXI master `M0` and the built-in crossbar to distribute the keys to all PCs.

Although easy-to-code and area-efficient, this conventional HLS coding style has two problems. First, while accessing multiple PCs from a single AXI master, data from different AXI masters will frequently share the lateral connections and reduce the effective bandwidth (more details in Section 4.2). Second, the bucket ID of a key read in line 15 may differ in the next iteration of the while loop. Thus, Vivado HLS will only use an AXI burst length of one for each key write. This also degrades the HBM effective bandwidth (more details in Section 6.1). In the following sections, we will examine solutions to these problems.

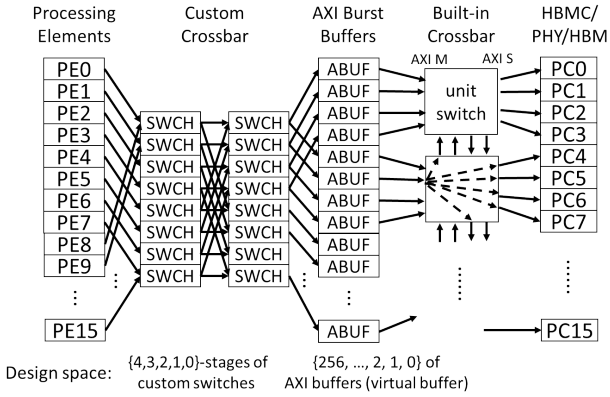


Figure 5: Overall architecture of HBM Connect and the explored design space

3 DESIGN SPACE AND PROBLEM FORMULATION

Let us denote a PE that performs computation as PE_i ($0 <= i < I$) and an HBM PC as PC_j ($0 <= j < J$). PE is a coarse-grain computational unit composed of a single function and may contain multiple fine-grain computational units inside its function. PE_i transfers $data_{ij}$ to PC_j. If PE_i makes no communication with PC_j, $data_{ij}$ equals 0. We denote the averaged effective bandwidth of transferring $data_{ij}$ as BW_{ij}. The total effective bandwidth of the system BW is equal to the summation of BW_{ij} for all (i, j).

We make the following assumptions. First, the kernel is written in a dataflow style, where functions execute in parallel and communicate through streaming FIFOs. Second, we read or write $data_{ij}$ from/to PC_j in a sequential address (see Section 2.3 for examples). Third, PEs reads and writes data every cycle (II=1) if its input/output FIFOs are not empty or full. Fourth, the pipeline depth of PEs is negligible compared to the total execution time t_{TOT}.

In Fig. 5, we show the design space of the HBM Connect. It consists of a custom crossbar, an AXI burst buffer, and a built-in AXI crossbar.

The purpose of the custom crossbar is to partly replace the functionality of the built-in AXI crossbar and increase the effective bandwidth. We employ a multi-stage butterfly network for a reason we will explain in Section 5.1. As a design space, we may use $CXBAR$ = 0, 1, 2, ..., log(16) stages of custom crossbar.

An AXI burst buffer is needed to enable burst access in the built-in crossbar (more details in Section 6.1). The design space of the AXI buffer size is $ABUF$ = 0, 1, 2, 4, ... 128, 256.

The aim of this work is to find a PE_i-PC_j interconnect architecture (among all i's and j's) that has a good trade-off between the data transmission time and the interconnect resource usage. For quantitative evaluation, we use metrics that are similar to the inverse of the classic area-delay-square product (AT^2) metric. Specifically, we divide the squared value of the effective HBM bandwidth by LUT (BW^2/LUT), FF (BW^2/FF), or BRAM ($BW^2/BRAM$). These metrics intuitively match a typical optimization goal of maximizing the effective bandwidth while using as small resource as possible. The effective bandwidth term is squared with an assumption that the

HBM boards will be more popular for memory-bound applications—that is, the bandwidth is a more important criteria than the resource consumption in the HBM boards.

The problem we solve in this paper is formulated as:
Given $data_{ij}$, find a design space (CXBAR, ABUF) that minimizes BW^2/LUT.

Metric BW^2/LUT in the formulation may be replaced with metrics BW^2/FF or $BW^2/BRAM$. The choice among the three metrics will depend upon the bottleneck resource of the PEs.

We will explain the details of the HBM Connect major components in the following sections. Section 4 provides an analysis of the built-in crossbar. The architecture and optimization of the custom crossbar is presented in Section 5. The HLS-based optimization of the AXI burst buffer will be described in Section 6.

4 BUILT-IN CROSSBAR AND HBM

This section provides an analysis of the built-in AXI crossbar and HBM. The analysis is used to estimate the effective bandwidth of the built-in interconnect system and guide the design space exploration. See [4] for more details on our memory access analysis. We also refer readers to the related HBM benchmarking studies in [18, 19, 27].

4.1 Single PC Characteristics

We measure the effective bandwidth when a PE uses a single AXI master to connect to a single HBM PC. We assume that the PE is designed with Vivado HLS.

4.1.1 Maximum Bandwidth. The maximum memory bandwidth of the HBM boards is measured with a long (64MB) sequential access pattern. The experiment performs a simple data copy with read & write, read only, and write only operations. We use the Alveo's default user logic data bitwidth of 512b.

A related RTL-based HBM benchmarking tool named Shuhai [27] assumes that the total effective bandwidth can be estimated by multiplying the bandwidth of a single PC by the total number of PCs. In practice, however, we found that it is difficult to utilize all PCs. PC 30 and 31 partially overlap with the PCIE static region, and Vitis was not able to complete the routing even for a simple traffic generator for PC 30 and 31. The routing is further complicated by the location of HBM MCs—they are placed on the bottom SLR (SLR0) and user logic of memory-bound applications tends to get placed near the bottom. For this reason, we used 16 PCs (nearest power-of-two usable PCs) for evaluation throughout this paper.

Table 2: Maximum effective per-PC memory bandwidth with sequential access pattern on Alveo U280 (GB/s)

Read & Write	Read only	Write only	Ideal
12.9	13.0	13.1	14.4

Table 2 shows the measurement result. The effective bandwidth per PC is similar to 13.3 GB/s measured in RTL-based Shuhai [27]. The result demonstrates that we can obtain about 90% of the ideal bandwidth. The bandwidth can be saturated with read-only or write-only access.

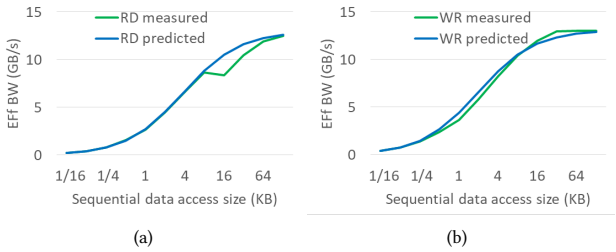


Figure 6: Effective memory bandwidth per PC (a single AXI master accesses a single PC) with varying sequential data access size (a) Read BW (b) Write BW

4.1.2 Short Sequential Access Bandwidth. In most practical applications, it is unlikely that we can fetch such a long (64MB) sequential data. The bucket PE, for example, needs to write to multiple PCs, and there is a constraint on the size of write buffer for each PC (more details in Section 6). Thus, each write must be limited in length. A similar constraint exists on the merge sort PE's read length.

HLS applications require several cycles of delay when making an external memory access. We measure the memory latency LAT using the method described in [5, 6] (Table 3).

Table 3: Read/write memory latency

	Read lat	Write lat
Total	289 ns	151 ns

Let us divide $data_{ij}$ into $CNUM_{ij}$ number of data chunks sized $BLEN_{ij}$:

$$data_{ij} = CNUM_{ij} * BLEN_{ij} \quad (1)$$

The time $t_{BLEN_{ij}}$ taken to complete one burst transaction of length $BLEN_{ij}$ to HBM PC can be modeled as [7, 24]:

$$t_{BLEN_{ij}} = BLEN_{ij}/BW_{max} + LAT \quad (2)$$

where BW_{max} is the maximum effective bandwidth (Table 2) of one PC, and LAT is the memory latency (Table 3).

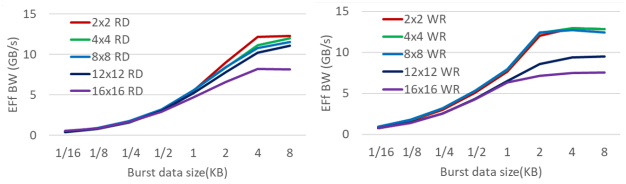
Then the effective bandwidth when a single AXI master accesses a PC is:

$$BW_{ij} = BLEN_{ij}/t_{BLEN_{ij}} \quad (3)$$

Fig. 6 shows the comparison between the estimated effective bandwidth and the measured effective bandwidth after varying the length ($BLEN_{ij}$) of sequential data access on a single PC. Note that the trend of the effective bandwidth in this figure resembles that of other non-HBM, DRAM-based FPGA platforms [5, 6].

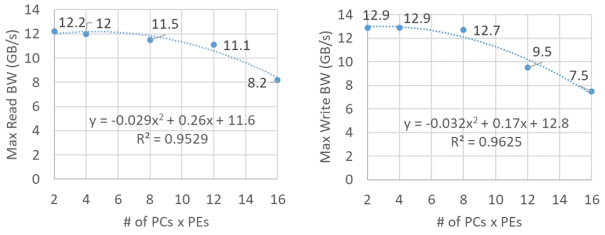
4.2 Many-to-Many Unicast Characteristics

In this section, we consider the case when multiple AXI masters access multiple PCs in round-robin. Since each AXI master access only one PC at a time, we will refer to this access pattern as many-to-many unicast. We vary the number of PCs accessed by AXI masters. For example, in the many-to-many write unicast test with (AXI masters × PCs) = (2×2) configuration, AXI master M0 writes to PC0/PC1, M1 writes to PC0/PC1, M2 writes to PC2/PC3, M3 writes to PC2/PC3, and so on. AXI masters access different PCs in round



(a) (b)

Figure 7: Many-to-many unicast effective memory bandwidth among 2~16 PCs (a) Read BW (b) Write BW



(a) (b)

Figure 8: Maximum bandwidth (BW_{max}) for many-to-many unicast on Alveo U280 (GB/s) (a) Read BW (b) Write BW

robin. Another example of this would be the many-to-many read unicast test with (AXI masters × PCs) = (4×4) configuration. All M0, M1, M2, and M3 masters read from PC0, PC1, PC2, and PC3 in round robin. The AXI masters are not synchronized, and it is possible that some masters will idle waiting for other masters to finish their transaction.

Fig. 7 shows the effective bandwidth after varying the burst length and the number of PCs accessed by AXI masters. The write bandwidth (Fig. 7(b)) is generally higher than the read bandwidth (Fig. 7(a)) for the same burst length because the write memory latency is smaller than the read memory latency (Table 3). Shorter memory latency decreases the time needed per transaction (Eq. 2).

For 16×16 unicast, which is the configuration used in bucket sort and merge sort, the lateral connections become the bottleneck. For example, M0 needs to cross three lateral connections of unit switches to reach PC12–PC15. Multiple crossings severly reduces the overall effective bandwidth.

Fig. 8 summarizes the maximum bandwidth observed in Fig. 7. The reduction in the maximum bandwidth becomes more severe as more AXI masters contend with each other to access the same PC.

We can predict the effective bandwidth of many-to-many unicast by replacing the BW_{max} in Eq. 2 with the maximum many-to-many unicast bandwidth in Fig. 8. The maximum many-to-many unicast bandwidth can be reasonably well estimated (R^2=0.95 ~ 0.96) by fitting the experimentally obtained values with a second-order polynomial. The fitting result is shown in Fig. 8.

5 CUSTOM CROSSBAR

5.1 Network Topology

As demonstrated in Section 4.2, it is not possible to reach the maximum bandwidth when an AXI master tries to access multiple PCs. To reduce the contention, we add a custom crossbar.

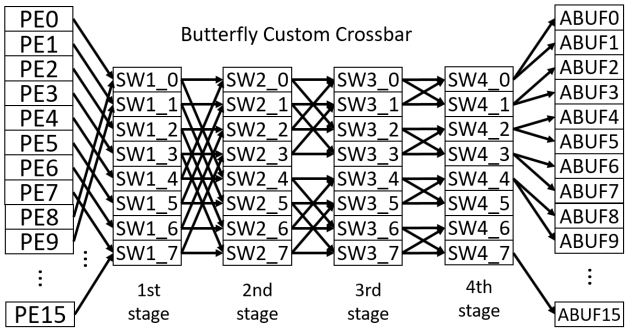


Figure 9: The butterfly custom crossbar architecture (when *CXBAR*=4)

We found that Vitis was unable to finish routing when we tried to make a fully connected crossbar. Thus, we decided to employ a multi-stage network. To further simplify the routing process, we compose the network with 2×2 switching elements.

There are several multi-stage network topologies. Examples include Omega, Clos, Benes, and butterfly networks. Among them, we chose the butterfly network shown in Fig. 9. We chose this topology because the butterfly network allows sending data across many hops of AXI masters with just a few stages. For example, let us assume we deploy only the first stage of butterfly network shown in Fig. 9. Data sent from PE0 to PC8–PC15 can avoid going through two or three lateral connections with just a single switch `SW1_0`. The same benefit applies to the data sent from PE8 to PC0–PC7. We can achieve a good trade-off between the LUT consumption and the effective bandwidth due to this characteristics. The butterfly network reduces its hop distance at the later stages of the custom crossbar. Note that the performance and the resource usage is similar to that of Omega networks if all four stages are used.

Adding more custom stages will reduce the amount of traffic crossing the lateral connection at the cost of more LUT/FF usage. If we implement two stages of butterfly as in Fig. 5, each AXI master has to cross a single lateral connection. If we construct all four stages as in Fig. 9, the AXI master in the built-in crossbar only accesses a single PC.

5.2 Mux-Demux Switch

A 2×2 switching element in a multistage network reads two input data and writes to output ports based on the destination PC. A typical 2×2 switch can send both input data to output if the data's output ports are different. If they are the same, one of them has to stall until the next cycle. Assuming the 2×2 switch has an initiation interval (II) of 1 and the output port of the input data is random, the averaged number of output data per cycle is 1.5.

We propose an HLS-based switch architecture named *mux-demux switch* to increase the throughput. A mux-demux switch decomposes a 2×2 switch into simple operations to be performed in parallel. Next, we insert buffers between the basic operators so that there is a higher chance that some data will exist to be demuxed/muxed. We implement buffers as FIFOs for simpler coding style.

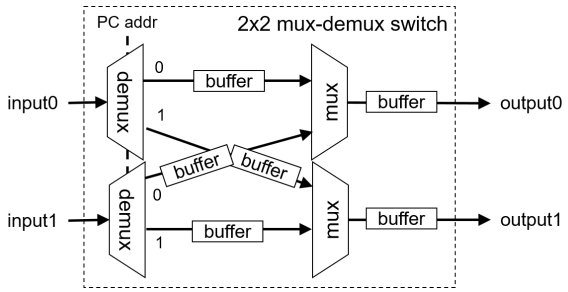


Figure 10: Architecture of mux-demux switch

Fig. 10 shows the architecture of mux-demux switch. After reading data in input0 and input1, the two demux modules independently classify the data based on the destination PC. Then instead of directly comparing the input data of the two demux modules, we store them in separate buffers. In parallel, the two mux modules each read data from two buffers in round-robin and send the data to their output ports.

As long as the consecutive length of data intended for a particular output port is less than the buffer size, this switch can almost produce two output elements per cycle. In essence, this architecture trades-off buffer with performance.

We estimate the performance of mux-demux switch with a Markov chain model (MCM), where the number of remaining buffer corresponds to a single MCM state. The transition probability between MCM states is modeled from the observation that the demux module will send data to one of the buffers with 50% probability every cycle for random input (thus reducing buffer space by one) and that the mux module will read from each buffer every two cycles in round-robin (thus increasing buffer space by one). The mux module does not produce an output if the buffer is in an "empty" MCM state. The MCM estimated throughput with various buffer sizes is provided in the last row of Table 4.

Table 4: Resource consumption (post-PnR) and throughput (experimental and estimated) comparison of typical 2×2 switch and the proposed 2×2 mux-demux switch in a stand-alone Vivado HLS test

	Typ SW	Mux-Demux SW		
Buffer size	-	4	8	16
LUT	3184	3732	3738	3748
FF	4135	2118	2124	2130
Thr (Exp.)	1.49	1.74	1.86	1.93
Thr (Est.)	1.5	1.74	1.88	1.94

We measure the resource consumption and averaged throughput after generating random input in a stand-alone Vivado HLS test. We compare the result with a typical 2×2 HLS switch that produces two output data only when its two input data's destination port is different. One might expect that a mux-demux switch would consume much more resource than a typical switch because it requires 4 additional buffers (implemented as FIFOs). But the result (Table 4) indicates that the post-PnR resource consumption is similar. This is due to the complex typical switch control circuit which compares two inputs for destination port conflict on every cycle (II=1). A mux-demux switch, on the other hand, decomposes this comparison into 4 simpler operations. Thus, the resource consumption is still comparable. In terms of throughput, a mux-demux switch clearly outperforms a typical switch.

We fix the buffer size of the mux-demux switch to 16 in HBM Connect, because it gives the best throughput-resource trade-off. Table 4 confirms that the experimental throughput well matches the throughput estimated by the MCM.

6 AXI BURST BUFFER

6.1 Direct Access from PE to AXI Master

In bucket sort, PEs distribute the keys to output PCs based on its value (each PC corresponds to a bucket). Since each AXI master can send data to any PC using the built-in crossbar (Sections 2.2), we first make a one-to-one connection between a bucket PE and an AXI master. Then we utilize the built-in AXI crossbar to perform the key distribution. We use the coding style in lines 17 to 20 of Fig. 4 to directly access different PCs.

With this direct access coding style, however, we were only able to achieve 59 GB/s among 16 PCs (with two stages of custom crossbar). We obtain such a low effective bandwidth because there is no guarantee that two consecutive keys from input PC will be sent to the same bucket (output PC). Existing HLS tools do not automatically hold the data in buffer for burst AXI access to each HBM PC. Thus, the AXI burst access is set to one. Non-burst access to HBM PC severely degrades the effective bandwidth (Fig. 6 and Fig. 7). A similar problem occurs when making a direct access for read many-to-many unicast in the merge sort.

6.2 FIFO-Based Burst Buffer

An intuitive solution to this problem is to utilize a FIFO-based AXI burst buffer for each PC [4]. Based on data's output PC information, data is sent to a FIFO burst buffer reserved for that PC. Since all the data in a particular burst buffer is guaranteed to be sent to a single HBM PC, the AXI bus can now be accessed in a burst mode. We may choose to enlarge the burst buffer size to increase the effective bandwidth.

However, we found that this approach hinders with effective usage of FPGA on-chip memory resource. It is ideal to use BRAM as the burst buffer because BRAM is a dedicated memory resource with higher memory density than LUT (LUT might be more efficient as a compute resource). But BRAM has a minimum depth of 512 [32]. As was shown in Fig. 6, we need a burst access of around 32 (2KB) to reach a half of the maximum bandwidth and saturate the HBM bandwidth with simultaneous memory read and write. Setting the burst buffer size to 32 will under-utilize the minimum BRAM depth (512). Table 5 confirms the high resource usage of the FIFO-based burst buffers.

Another problem is that this architecture scatters data to multiple FIFOs and again gathers data to a single AXI master. This further complicates the PnR process. Due to the high resource usage and the routing complexity, we were not able to route the designs with FIFO-based burst buffer (Table 5).

Table 5: Effective bandwidth and FPGA resource consumption of bucket sort with different AXI burst buffer schemes ($CXBAR = 2$)

Buf Sch	Bur Len	CX bar	FPGA Resource LUT/FF/DSP/BRAM	KClk (MHz)	EffBW (GB/s)
Direct access		2	126K / 238K / 0 / 248	178	56
FIFO	16	2	195K / 335K / 0 / 728	PnR failed	
Burst	32	2	193K / 335K / 0 / 728	PnR failed	
Buf	64	2	195K / 335K / 0 / 728	PnR failed	
HLS	16	2	134K / 233K / 0 / 368	283	116
Virt	32	2	134K / 233K / 0 / 368	286	185
Buf	64	2	134K / 233K / 0 / 368	300	180

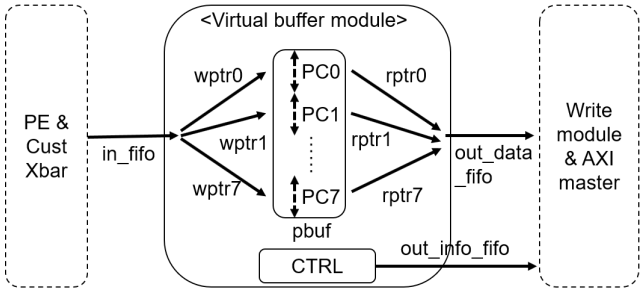


Figure 11: HLS virtual buffer architecture for 8 PCs

```
01 ap_uint<512> pbuf[8*ABUF];              //single physical buffer
02
03 while(...){
04 #pragma HLS pipeline II=1
05 #pragma HLS dependency variable=pbuf inter false
06   if( write_in_progress = 1 ){
07     write_data = pbuf[64*write_ch + rptr[write_ch]];
08                                  //read data from physical buffer
09     out_data_fifo.write(write_data);    //write to out_data_fifo
10     rptr[write_ch]++;               //increases the read pointer
11     write_num--;                    //keep track of data written
12     if( write_num == 0 ){
13       write_in_progress = 0;  //stop write when data is all sent
14     }
15   }
16   else{          //initiate a write transfer for a new target PC
17     write_ch++;
18     write_num = wptr[write_ch] - rptr[write_ch];
19           //number of write determined by the existing elements
20     out_info_fifo.write((write_ch, write_num)); //pass ctrl info
21     write_in_progress = 1;
22   }
23
24   read_success = in_fifo.read_nb(read_data);
25   if( read_success ){
26     targetPC = read_data(.....); //find targetPC from the read_data
27     pbuf[64*targetPC + wptr[targetPC]] = read_data;
28                                      //write to physical buffer
29     wptr[targetPC]++;              //increases the write pointer
30 } }
```

Figure 12: HLS code for HLS virtual buffer (for write)

6.3 HLS Virtual Buffer

In this section, we propose an HLS-based solution to solve all of the aforementioned problems: the burst access problem, the BRAM under-utilization problem, and the FIFO scatter/gather problem. The idea is to share the BRAM as a burst buffer for many different target PCs. But none of current HLS tools offer such functionality. Thus, we propose a new HLS-based buffering scheme called *HLS virtual buffer* (HVB). HVB allows a single physical FIFO to be shared among multiple virtual channels [11] in HLS. As a result, we can have a higher utilization of BRAM depth as the FIFOs for many different PCs. Another major advantage is that the HVB physically occupies one buffer space—we can avoid scattering/gathering data from multiple FIFOs and improve the PnR process.

```
vir_ch0 = 0;
for(i=0; i<BURST_LEN; i++){
#pragma HLS pipeline II=1
  data = pfifo.vfifo_read(vir_ch0);
  pe0_pc0[i] = data;
}
```

Figure 13: Abstracted HLS virtual buffer syntax (for read)

We present the architecture of HVB in Fig. 11 and its HLS code in Fig. 12. A physical buffer (`pbuf`) is partitioned into virtual FIFO buffers for 8 target PCs. The buffer for each PC has a size of $ABUF$, and we implement it as a circular buffer with a write pointer (`wptr`) and a read pointer (`rptr`). At each cycle, the HVB reads a data from textttin_fifo in a non-blocking fashion (line 24) and writes it to the target PC's virtual buffer (line 27). The partition among different PCs in `pbuf` is fixed.

Whereas the target PC for input data is random, the output data is sent in a burst for the same target PC. Before initiating a write transfer for a new target PC, the HVB passes the target PC and the number of elements in `out_info_fifo` (line 20). Then it transmits the output data in a burst as shown in lines 7 to 14. A separate write logic (omitted) receives the burst information and relays the data to an AXI master.

It implements the HVB for read operation (e.g., in merge sort) in a similar code as in Fig. 12, except that it collects the input data in a burst from a single source PC and sends output data in a round-robin fashion among different PCs.

The HVB for read operation (e.g., in merge sort) is implemented in a similar code as in Fig. 12, except that the input data is collected in a burst from a single source PC and output data is sent in a round-robin fashion among different PCs.

Table 5 shows that the overall LUT/FF resource consumption of HVB is similar to the direct access scheme. The performance is much better than the direct access scheme because we send data through the built-in crossbar in a burst. Compared to the FIFO burst buffer scheme, we reduce the BRAM usage as expected because HVB better utilizes the BRAM by sharing. Also, the LUT/FF usage has been reduced because we only use a single physical FIFO. The routing for HVB is successful because of the small resource consumption and the low complexity.

We can estimate the performance of HVB by setting BW_{max} of Eq. 2 to that of Fig. 8 and $BLEN$ to the buffer size of HVB ($ABUF$).

It is difficult for novice HLS users to incorporate the code in Fig. 12 into their design. For better abstraction, we propose using the syntax shown in Fig. 13. A programmer can instantiate a physical buffer `pfifo` and use a new virtual buffer read keyword `vfifo_read`. The virtual channel can be specified with a tag

Table 6: Effective bandwidth (on-board test), BW^2/resource metrics, and resource consumption (post-PnR) of bucket sort after varying the number of crossbar stages

Cus Xbar	AXI Xbar	Bur Len	FPGA Resource LUT/FF/DSP/BRAM	KClk (MHz)	EffBW (GB/s)	BW^2/Resource Metrics BW^2/LU	BW^2/FF	BW^2/BR
0	4	0	102K / 243K / 0 / 248	277	65	1.0	1.0	1.0
0	4	64	122K / 243K / 0 / 480	166	108	2.3	2.7	1.4
1	3	64	121K / 231K / 0 / 368	281	160	5.1	6.4	4.1
2	2	64	134K / 233K / 0 / 368	300	180	5.8	8.0	5.2
3	1	64	155K / 243K / 0 / 368	299	195	5.9	9.0	6.1
4	0	0	189K / 305K / 0 / 248	207	203	5.3	7.8	9.8

`vir_ch0`. Then an automated tool can be used to perform a code-to-code transformation from this abstracted code to the detailed implementation in Fig. 12.

7 DESIGN SPACE EXPLORATION

As explained in Section 3, we explore the design space for *CXBAR* = 0, 1, 2, ...log(16) and *ABUF* = 0, 1, 2, 4, ... 128, 256. The throughput is estimated using the methods described in Sections 4, 5, and 6. The resource is estimated by first generating few design spaces and obtaining the unit resource consumption of the components. Table 7 shows the unit resource consumption of major HBM Connect components. The BRAM consumption of HVB is estimated by multiplying the burst buffer depth and the number of supported PCs ceiled by a 512 minimum depth. Next, we estimate the number of components based on the design space and multiply it by the unit consumption.

Table 7: FPGA resource unit consumption (post-PnR) of major components (data bitwidth:512b)

	LUT	FF	DSP	BRAM
HLS AXI master	2220	6200	0	15.5
Mux-Demux switch	3748	2130	0	0
HVB *ABUF*=64, 8PCs	160	601	0	7.5
HVB *ABUF*=128, 8PCs	189	612	0	14.5

Since there are only 5 (*CXBAR*) × 9 (*ABUF*) = 45 design spaces which can be estimated in seconds, we enumerate all design spaces. The design space exploration result will be presented in Section 8.

8 EXPERIMENTAL RESULT

8.1 Experimental Setup

We use Alveo U280 board for experiment. The board's FPGA resource is shown in Table 8. For programming, we utilize Xilinx's Vitis [33] and Vivado HLS [31] 2019.2 tools.

Table 8: FPGA resource on Alveo U280

LUT	FF	DSP	BRAM
1.30M	2.60M	9.02K	2.02K

8.2 Case Study 1: Bucket Sort

In Table 5, we have already presented a quantitative resource-performance analysis when enlarging the HLS virtual buffer (after fixing the number of custom crossbar stage). In this section, we first analyze the effect of varying the number of custom crossbar stages. We fix the HLS virtual buffer size to 64 for clearer comparison.

The result is shown in Table 6. We only account for the post-PnR resource consumption of the user logic and exclude the resource consumption of the the static region, the MCs, and the built-in crossbars. BW^2/LUT, BW^2/FF, and $BW^2/BRAM$ metrics are normalized to the baseline design with no custom crossbar stage and no virtual buffer. Larger value of these metrics suggests better designs.

As we add more custom crossbar stages, we can observe a steady increase of LUT and FF because more switches are needed. Larger number of custom crossbar stages reduces the data transaction through the lateral connections and increases the effective bandwidth. But as long as more than one AXI masters communicate with a common PC through the built-in AXI crossbar, the bandwidth loss due to contention is unavoidable (Section 4.2). When the custom crossbar (4 stages) completely replaces the built-in crossbar, one AXI master communicates with only a single PC. The data received from multiple PEs is written to the same memory space because the keys within a bucket does not need to be ordered. The one-to-one connection between an AXI master and a PC removes the contention in the built-in crossbar, and we can reach the best effective bandwidth (203 GB/s). Note that this performance closely approaches the maximum bandwidth of 206 GB/s (=16 PCs * 12.9GB/s) achieved with sequential access microbenchmark on 16 PCs (Table 2).

In terms of the resource-performance metrics (BW^2/LUT, BW^2/FF, and $BW^2/BRAM$), the designs with few custom crossbar stages are much better than the baseline design with no custom crossbar. For example, the design with two stages of custom crossbar and 64 virtual buffer depth per PC is superior by factors of 5.8X/8.0X/5.2X. Even though adding more custom crossbar stages resulted in an increased resource consumption, the amount of increased effective bandwidth is far greater. This result shows that memory-bound applications can benefit by adding a few custom crossbars to reduce the lateral connection communication.

We can observe a very interesting peak in the design point that has 4 stages custom crossbar. Since this design has the most number of switches, BW^2/LUT is slightly poor (5.3) compared to a design with two custom crossbar stages (5.8). But in this design, a PE only needs to communicate with a single bucket in a PC. Thus, we can infer burst access without an AXI burst buffer and remove the HVB.

The BRAM usage of this design point is lower than others, and $BW^2/BRAM$ is superior (9.8). We can deduce that if the data from multiple PEs can be written to the same memory space and BRAM is the most precious resource, it might be worth building enough custom crossbar stages to ensure one-to-one connection between an AXI master and a PC.

Table 9: Bucket sort's design points with best BW^2/LUT and $BW^2/BRAM$ metrics (normalized to a baseline design with $CXBAR$=0 and $ABUF$=0). Y-axis is the number custom crossbar stages and the X-axis is the virtual buffer depth. The best and the second best designs are in bold.

(BW^2/LUT)

	0	16	32	64	128
0	1.0	0.9	2.6	2.3	NA
1	1.0	2.8	**6.5**	5.1	3.5
2	0.6	2.4	**6.2**	5.8	4.7
3	0.8	2.3	3.8	5.9	5.3
4	5.3	-	-	-	-

($BW^2/BRAM$)

	0	16	32	64	128
0	1.0	0.7	2.0	1.4	NA
1	1.1	2.2	5.2	4.1	2.2
2	0.7	2.1	5.5	5.2	4.2
3	1.1	2.3	3.9	**6.1**	5.5
4	**9.8**	-	-	-	-

Table 9 presents the design space exploration result with a various number of custom/built-in crossbar stages and virtual buffer sizes. We present the numbers for BW^2/LUT and $BW^2/BRAM$ metrics but omit the table for BW^2/FF because it has a similar trend as the BW^2/LUT table. In terms of $BW^2/BRAM$ metric, ($CXBAR$=4, $ABUF$=0) is the best design point for the reason explained in the interpretation of Table 6. In terms of BW^2/LUT metric, the data points with $CXBAR$=1~3 have similar values and clearly outperform data points with ($CXBAR$=0). This agrees with the result in Fig. 7(b) where the 2×2 to 8×8 configurations all have a similar effective bandwidth and are much better than the 16×16 configuration. For both metrics, the design points with $ABUF$ less than 16 are not competitive because the effective bandwidth is too small (Fig. 7). The design points with $ABUF$ larger than 64 also are not competitive because almost an equal amount of read and write is performed on each PC—the effective bandwidth cannot increase beyond 6.5 GB/s (=12.9 GB/s ÷ 2) even with a large $ABUF$.

8.3 Case Study 2: Merge Sort

Table 10 shows the design space exploration of merge sort that uses HBM Connect in its read interconnect. The absolute values of metrics BW^2/LUT and $BW^2/BRAM$ are considerably higher than that of bucket sort for most of the design points. This is because the read effective bandwidth of the baseline implementation ($CXBAR$=0,$ABUF$=0) is 9.4 GB/s, which is much lower than the write effective bandwidth (65 GB/s) of the bucket sort baseline implementation.

As mentioned in Section 4.2, the read operation requires a longer burst length than the write operation to saturate the effective bandwidth because the read latency is relatively longer. Thus the BW^2/LUT metric reaches the highest point at the burst length of 128–256, which is larger than the 32–64 burst length observed in bucket sort (Table 9). $BW^2/BRAM$ metric, on the other hand, reaches the peak at the shorter burst length of 64 because a larger $ABUF$ requires more BRAMs.

Table 10: Merge sort's design points with best BW^2/LUT and $BW^2/BRAM$ metric (normalized to a baseline design with $CXBAR$=0 and $ABUF$=0). Y-axis is the number custom crossbar stages and the X-axis is the virtual buffer depth.

(BW^2/LUT)

	0	32	64	128	256
0	1.0	64	52	NA	NA
1	1.8	82	120	100	114
2	1.7	88	149	119	168
3	1.5	86	141	154	**211**
4	12	85	137	181	**191**

($BW^2/BRAM$)

	0	32	64	128	256
0	1.0	57	34	NA	NA
1	1.6	62	66	36	25
2	1.6	66	81	42	35
3	1.6	70	**84**	60	48
4	15	70	**85**	73	46

Similar to bucket sort, replacing the built-in crossbar with a custom crossbar provides a better performance because there is less contention in the built-in crossbar. As a result, design points with $CXBAR$=4 or $CXBAR$=3 generally have better BW^2/LUT and $BW^2/BRAM$. But unlike bucket sort, the peak in $BW^2/BRAM$ for $CXBAR$=4 does not stand out—it has a similar value as $CXBAR$=3. This is because merge sort needs to read from 16 different memory spaces regardless of the number of custom crossbar stages (explained in Section 2.3.2). Each memory space requires a separate virtual channel in HVB. Thus, we cannot completely remove the virtual buffer as in the bucket sort.

9 CONCLUSION

We have implemented memory bound applications on a recently released FPGA HBM board and found that it is difficult to fully exploit the board's bandwidth when multiple PEs access multiple HBM PCs. HBM Connect has been developed to meet this challenge. We have proposed several HLS-compatible optimization techniques such as the HVB and the mux-demux switch to remove the limitation of current HLS HBM syntax. We also have tested the effectiveness of butterfly multi-stage custom crossbar to reduce the contention in the lateral connection of the built-in crossbar. We found that adding AXI burst buffers and custom crossbar stages significantly improves the effective bandwidth. We also found in the case of bucket sort that completely replacing the built-in crossbar with a full custom crossbar may provide the best trade-off in terms of BRAMs if the output from multiple PEs can be written into a single space. The proposed architecture improves the baseline implementation by a factor of 6.5X–211X for BW^2/LUT metric and 9.8X–85X for $BW^2/BRAM$ metric. As a future work, we plan to apply HBM Connect to Intel HBM boards and also generalize it beyond the two cases studied in this paper.

10 ACKNOWLEDGMENTS

This research is in part supported by Xilinx Adaptive Compute Cluster (XACC) Program, Intel and NSF Joint Research Center on Computer Assisted Programming for Heterogeneous Architectures (CAPA) (CCF-1723773), NSF Grant on RTML: Large: Acceleration to Graph-Based Machine Learning (CCF-1937599), NIH Award (U01MH117079), and Google Faculty Award. We thank Thomas Bollaert, Matthew Certosimo, and David Peascoe at Xilinx for helpful discussions and suggestions. We also thank Marci Baun for proofreading this article.

REFERENCES

[1] ARM. 2011. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite.* www.arm.com

[2] J. Bakos. 2010. High-performance heterogeneous computing with the Convey HC-1. *IEEE Comput. Sci. Eng.* 12, 6 (2010), 80–87.

[3] R. Chen, S. Siriyal, and V. Prasanna. 2015. Energy and memory efficient mapping of bitonic sorting on FPGA. In *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*. 240–249.

[4] Y. Choi, Y. Chi, J. Wang, L. Guo, and J. Cong. 2020. When HLS meets FPGA HBM: Benchmarking and bandwidth optimization. *ArXiv Preprint* (2020). https://arxiv.org/abs/2010.06075

[5] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platform. In *Proc. Ann. Design Automation Conf.* 109–114.

[6] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. 2019. In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms. *ACM Trans. Reconfigurable Technology and Systems* 12, 1 (Feb. 2019).

[7] Y. Choi, P. Zhang, P. Li, and J. Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*. 691–698.

[8] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang. 2018. Understanding performance differences of FPGAs and GPUs. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines*. 93–96.

[9] P. Cooke, J. Fowers, G. Brown, and G. Stitt. 2015. A tradeoff analysis of FPGAs, GPUs, and multicores for sliding-window applications. *ACM Trans. Reconfigurable Technol. Syst.* 8, 1 (Mar. 2015), 1–24.

[10] B. Cope, P. Cheung, W. Luk, and L. Howes. 2010. Performance comparison of graphics processors to reconfigurable logic: a case study. *IEEE Trans. Computers* 59, 4 (Apr. 2010), 433–448.

[11] W. J. Dally and C. L. Seitz. 1987. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computers* C-36, 5 (May 1987), 547–553.

[12] K. Fleming, M. King, and M. C. Ng. 2008. High-throughput pipelined mergesort. In *Int. Conf. Formal Methods and Models for Co-Design.*

[13] Intel. 2020. *High Bandwidth Memory (HBM2) Interface Intel FPGA IP User Guide.* https://www.intel.com/

[14] Intel. 2020. *Avalon Interface Specifications.* https://www.intel.com/

[15] JEDEC. 2020. *High Bandwidth Memory (HBM) DRAM.* https://www.jedec.org/standards-documents/docs/jesd235a

[16] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim. 2017. HBM (High Bandwidth Memory) DRAM technology and architecture. In *Proc. IEEE Int. Memory Workshop*. 1–4.

[17] S. Lahti, P. Sjövall, and J. Vanne. 2019. Are we there yet? A study on the state of high-level synthesis. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 898–911.

[18] R. Li, H. Huang, Z. Wang, Z. Shao, X. Liao, and H. Jin. 2020. Optimizing memory performance of Xilinx FPGAs under Vitis. *ArXiv Preprint* (2020). https://arxiv.org/abs/2010.08916

[19] A. Lu, Z. Fang, W. Liu, and L. Shannon. 2021. Demystifying the memory system of modern datacenter FPGAs for software programmers through microbenchmarking. In *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*.

[20] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. 2019. Streambox-HBM: Stream analytics on high bandwidth hybrid memory. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*. 167–181.

[21] D. Molka, D. Hackenberg, and R. Schöne. 2014. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Proc. Workshop on Memory Systems Performance and Correctness*. 1–10.

[22] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh. 2017. Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*. 5–14.

[23] Nvidia. 2020. *Nvidia Titan V.* https://www.nvidia.com/en-us/titan/titan-v/

[24] J. Park, P. Diniz, and K. Shayee. 2004. Performance and area modeling of complete FPGA designs in the presence of loop transformations. *IEEE Trans. Computers* 53, 11 (Sept. 2004), 1420–1435.

[25] M. Saitoh, E. A. Elsayed, T. V. Chu, S. Mashimo, and K. Kise. 2018. A high-performance and cost-effective hardware merge sorter without feedback datapath. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines*. 197–204.

[26] N. Samardzic, W. Qiao, V. Aggarwal, M. F. Chang, and J. Cong. 2020. Bonsai: High-performance adaptive merge tree sorting. In *Ann. Int. Symp. Comput. Architecture*. 282–294.

[27] Z. Wang, H. Huang, J. Zhang, and G. Alonso. 2020. Shuhai: Benchmarking High Bandwidth Memory on FPGAs. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines*.

[28] Xilinx. 2020. *Alveo U280 Data Center Accelerator Card User Guide.* https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1314-u280-reconfig-accel.pdf

[29] Xilinx. 2020. *Alveo U50 Data Center Accelerator Card User Guide.* https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1371-u50-reconfig-accel.pdf

[30] Xilinx. 2020. *AXI High Bandwidth Memory Controller v1.0.* https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf

[31] Xilinx. 2020. *Vivado High-level Synthesis (UG902).* https://www.xilinx.com/

[32] Xilinx. 2020. *UltraScale Architecture Memory Resources (UG573).* https://www.xilinx.com/

[33] Xilinx. 2020. *Vitis Unified Software Platform.* https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html