# Demystifying the Soft and Hardened Memory Systems of Modern FPGAs for Software Programmers through Microbenchmarking

ALEC LU, ZHENMAN FANG, and LESLEY SHANNON,
Simon Fraser University, Canada

Both modern datacenter and embedded FPGAs provide great opportunities for high-performance and high energy-efficiency computing. With the growing public availability of FPGAs from major cloud service providers such as AWS, Alibaba, and Nimbix, as well as uniform hardware accelerator development tools (such as Xilinx Vitis and Intel oneAPI) for software programmers, hardware and software developers can now easily access FPGA platforms. However, it is nontrivial to develop efficient FPGA accelerators, especially for software programmers who use high-level synthesis (HLS).

The major goal of this paper is to figure out how to efficiently access the memory system of modern datacenter and embedded FPGAs in HLS-based accelerator designs. This is especially important for memory-bound applications; for example, a naive accelerator design only utilizes less than 5% of the available off-chip memory bandwidth. To achieve our goal, we first identify a comprehensive set of factors that affect the memory bandwidth, including 1) the clock frequency of the accelerator design, 2) the number of concurrent memory access ports, 3) the data width of each port, 4) the maximum burst access length for each port, and 5) the size of consecutive data accesses. Then we carefully design a set of HLS-based microbenchmarks to quantitatively evaluate the performance of the memory systems of datacenter FPGAs (Xilinx Alveo U200 and U280) and embedded FPGA (Xilinx ZCU104) when changing those affecting factors, and provide insights into efficient memory access in HLS-based accelerator designs. Comparing between the typically used soft and hardened memory systems respectively found on datacenter and embedded FPGAs, we further summarize their unique features and discuss the effective approaches to leverage these systems. To demonstrate the usefulness of our insights, we also conduct two case studies to accelerate the widely used K-nearest neighbors (KNN) and sparse matrix-vector multiplication (SpMV) algorithms on datacenter FPGAs with a soft (and thus more flexible) memory system. Compared to the baseline designs, optimized designs leveraging our insights achieve about 3.5x and 8.5x speedups for the KNN and SpMV accelerators. Our final optimized KNN and SpMV designs on a Xilinx Alveo U200 FPGA fully utilize its off-chip memory bandwidth, and achieve about 5.6x and 3.4x speedups over the 24-core CPU implementations.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**.

Additional Key Words and Phrases: Datacenter FPGAs, Embedded FPGAs, Memory System, HLS, Benchmarking

**ACM Reference Format:**
Alec Lu, Zhenman Fang, and Lesley Shannon. 2022. Demystifying the Soft and Hardened Memory Systems of Modern FPGAs for Software Programmers through Microbenchmarking. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (September 2022), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' address: Alec Lu, alec_lu@sfu.ca; Zhenman Fang, zhenman@sfu.ca; Lesley Shannon, lesley_shannon@sfu.ca, Simon Fraser University, Canada.

1936-7406/2022/9-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

With the end of general-purpose CPU scaling due to power and utilization walls [15], customizable accelerators on FPGAs have gained increasing attention in modern datacenters and heterogeneous edge computing platforms due to their high flexibility, low power, high performance and energy-efficiency. In the past few years, all major cloud service providers—such as Amazon Web Services [3], Microsoft Azure [25], Alibaba Cloud [1], and Nimbix [26]—have deployed FPGAs in their datacenters. On the other hand, embedded FPGAs, relatively cheaper in cost than datacenter FPGAs, have been extensively used to bring acceleration to computational intensive in-field applications such as, real-time object detection [31], autonomous driving system [22], and machine learning inference [21]. As a result, hardware and software developers not only have access to FPGAs on the embedded System-on-Chip (SoC) boards but also can easily access FPGA computing platforms as a cloud service. Moreover, programmers can now leverage uniform development tools, such as Xilinx Vitis [39] and Intel oneAPI [20], to design their accelerators on both datacenter and embedded FPGAs. However, it is nontrivial to design efficient accelerators on these FPGAs, especially for software programmers who use high-level languages such as C/C++.

One of the critical programming challenges is to design efficient communication between the accelerator and the off-chip memory system, and between multiple accelerators. It is especially challenging for communication and/or memory bound accelerators designed in high-level synthesis (HLS) languages like HLS C/C++. For example, as observed in [14], an HLS-based design that uses 32-bit data types, which is common in software programs (e.g., int and float), only utilizes 1/16 of the off-chip memory bandwidth. Another study in [42] found that the maximum effective off-chip memory bandwidth for an HLS-based design—10GB/s for DDR3, about 83% of the theoretical peak bandwidth 12.8GB/s—can only be achieved when the memory access port width is at least 512-bit and the consecutive data access size is at least 128KB.

In fact, in this paper, we find that there are more factors (summarized in Section 3.1) affecting the effective memory system performance which an HLS-based accelerator design can achieve. For example, without careful tuning of the maximum burst access length for each memory access port, adding more ports can lead to unstable degradation of the total effective memory bandwidth that the design can achieve by up to 3x, which is shown in Figure 6(a) and explained in Section 4.3.1. Clock frequency of the accelerator design is another factor that often gets overlooked when considering the effective memory bandwidth; the best port width of a memory access port for achieving the peak effective bandwidth depends on the accelerator clock frequency, as shown in equation 4 and explained in 4.2, and can exceed 512 bits. Unfortunately, neither the HLS report [40] nor the hardware emulation (i.e., register-transfer level simulation) [39] accurately models the effective off-chip memory bandwidth under all those affecting factors.

In this paper, building upon our work presented in [24], our goal is to demystify the effective memory system performance of modern FPGA boards under a comprehensive set of affecting factors. Particularly, we look at two types of modern FPGAs platforms: datacenter FPGA cards—which usually have multiple FPGA dies and multiple DRAM or HBM banks in a single board [37, 38]; and embedded FPGA boards—which typically have a single FPGA die sharing access to a single DRAM bank with a multi-core ARM processor [36]. With a quantitative evaluation of the recent Xilinx Alveo U200 and U280 datacenter FPGA boards [37, 38], and the Xilinx ZCU104 embedded FPGA SoC boards [36], we aim to provide insights for software programmers into how to access the memory system in their HLS-based accelerator designs efficiently. In summary, this paper makes the following contributions.

1. We identify a comprehensive set of factors that affect the memory system performance in HLS-based FPGA accelerators. This includes 1) the clock frequency of the accelerator design, 2)

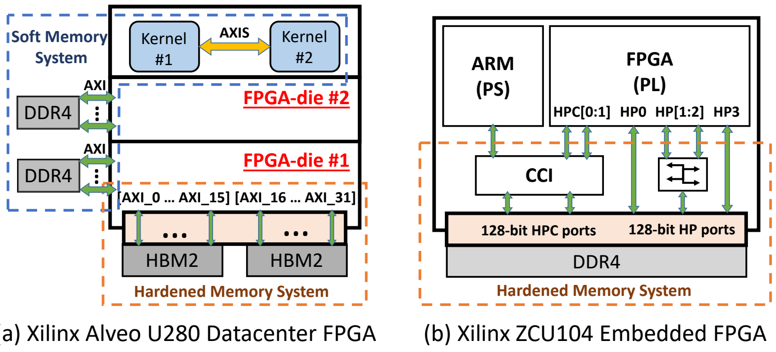

Fig. 1. Architecture layout for modern FPGAs.

the number of concurrent memory access ports, 3) the data width of each port, 4) the maximum burst access length for each port, and 5) the size of consecutive data accesses.

2. We develop a suite of open source HLS-C/C++ microbenchmarks to quantitatively evaluate the effective bandwidth, latency, and resource usage of the off-chip memory access and the accelerator-to-accelerator streaming in modern datacenter and embedded FPGAs, while systematically changing those affecting factors. Our microbenchmark suite has been open sourced at: https://github.com/SFU-HiAccel/uBench.
3. We summarize the similarities and differences between the soft (i.e., user has more control) and hardened (i.e., user has less control and can only select which memory ports to connect) memory systems typically used in datacenter FPGAs and embedded FPGAs, respectively. Our definition of soft and hardened memory systems are described in Section 2. We also discuss the trade-offs and effective approaches to leverage these memory systems.
4. We also conduct two accelerator case studies on the widely used K-nearest neighbors (KNN) [2, 8] and sparse matrix-vector multiplication (SpMV) [6, 29] algorithms to demonstrate how to leverage our results and insights during design space exploration for HLS-based accelerators. Our final optimized KNN and SpMV designs on an Xilinx Alveo U200 datacenter FPGA fully utilize the off-chip memory bandwidth of all four DRAMs, achieve 3.5x and 8.5x speedups over the baseline FPGA designs, and 5.6x and 3.4x speedups over the 24-core CPU implementations.

The rest of the paper is organized as follows. Section 2 discusses the common memory systems found on modern FPGAs and the focus of our paper. Section 3 describes the challenges and methodology behind our microbenchmark designs. Sections 4 and 5 present the results and insights on datacenter and embedded FPGAs, respectively. Section 6 summarizes the results and discusses the similarities and differences between soft and hardened memory systems. Sections 7 and 8 conduct two case studies on the KNN and SpMV accelerator designs, respectively, and demonstrate that leveraging our insights can provide up to 3.5x and 8.5x speedups over the baseline designs. Section 9 discusses previous studies on memory system evaluations and finally, Section 10 presents our concluding statements.

## 2 MODERN DATACENTER AND EMBEDDED FPGAS AND OUR FOCUS

Recently, a variety of FPGA platforms have been deployed in datacenters, such as Microsoft Catapult [28], AWS F1 instance [3], Alibaba F1 and F3 instances [1], IBM OpenCAPI [32], Intel HARP and HARPv2 platforms [4], Xilinx Alveo boards in Nimbix [26, 37, 38], and Intel Stratix X [19]. To meet the ever-increasing demand in datacenter workloads, both the computing resource and memory bandwidth provided by these FPGAs keep increasing. Typically, a single datacenter

Table 1. Off-chip memory details and theoretical bandwidth on modern datacenter and embedded FPGAs

| Off-chip memory | Bus width (bits per transfer) | Transfer rate | Theoretical peak bandwidth |
|---|---|---|---|
| U200/U280 DRAM | 64 bits | 2,400 MT/s | 19.2 GB/s |
| U280 HBM (per bank) | 64 bits | 1,800 MT/s | 14.4 GB/s |
| ZCU104 DRAM | 64 bits | 2,133 MT/s | 17.0 GB/s |

FPGA board, as shown in Figure 1(a), consists of multiple FPGA dies and multiple DRAMs and even HBMs (high-bandwidth memory).

On the other hand, due to being cost-effective and energy efficient, FPGA platforms are also widely used as edge computing solutions for time-critical computational intensive applications such as real-time object detection [31], autonomous driving system [22], and machine learning inference [21]. A modern embedded FPGA is typically integrated on an SoC-based platform, as illustrated in Figure 1(b), which is composed of the processing system (PS) and programmable logic (PL), both sharing a single DRAM. The PS is typically a multi-core ARM CPU that runs the main application program; while the PL is the FPGA for running customized hardware accelerators.

In this paper, we mainly focus on measuring the performance of off-chip memory access by accelerators using the memory-mapped AXI ports [35, 39] and on-chip accelerator-to-accelerator streaming using the AXIS streaming ports [35, 39], as shown in Figure 1. Moreover, we mainly focus on the performance under consecutive data access patterns, which are the most efficient and the most common access pattern for FPGA accelerators. The random access pattern is considered as a special case where the consecutive data access size is just one element.

The platforms evaluated in this study are the PCIe-based Xilinx Alveo U200 and U280 datacenter FPGA boards [37, 38] and the Xilinx Zynq UltraScale+ ZCU104 embedded board [36]. The U200 FPGA is similar to the AWS F1 instance setup [3], which features three separate FPGA dies and four 16GB off-chip DDR4-2400MT DRAMs. It uses a **soft memory system**, where users have more control of the AXI port connections from the accelerator to the off-chip memory, such as the number of AXI ports and the (physical) data width of each AXI port. As shown in Table 1, for each DDR4-2400MT DRAM in the U200 and U280 FPGAs, its memory bus width ($mem\_bus\_width$) is 64 bits (i.e., each transaction transfers 64 bits of data), its bus transfer rate ($transfer\_rate$) is 2,400M transfers per second, and its theoretical peak bandwidth ($theoretical\_mem\_BW$) is 19.2GB/s. The theoretical peak bandwidth can be calculated using the following equation.

$$theoretical\_mem\_BW = mem\_bus\_width \times transfer\_rate/8\text{-bit } (B/s) \tag{1}$$

where $transfer\_rate$ is twice the rate of the memory bus clock frequency, since DRAM and HBM banks are dual rated.

The U280 FPGA features not only three separate FPGA dies and two 16GB off-chip DDR4-2400MT DRAMs connected through a soft memory system, but also two 4GB HBM2 stacks that have a total of 32 256MB HBM banks. Figure 1(a) shows the architecture layout of the U280 FPGA. The HBM in the U280 FPGA uses a **hardened memory system** that runs at 450MHz: there are 32 hardened AXI ports connecting to the HBM banks and the data width of each AXI port is 256 bits (fixed). Users can only choose which HBM bank(s) to connect with their accelerator and configure the AXI port width on the accelerator side, but not on the memory system side. As shown in Table 1, for each HBM2 bank on the U280 FPGA, its memory bus width is 64 bits, its bus transfer rate is 1,800MT/s, and its theoretical peak bandwidth is 14.4GB/s. Note that the hardened HBM memory system exactly matches with this theoretical peak bandwidth, as will be explained in Equation 4.

Shown in Figure 1(b), the ZCU104 board comes with a quad-core ARM Cortex™-A53 processing system (PS), a single FPGA die (PL), and a 2GB off-chip DDR4-2400MT DRAM. Note that even though the ZCU104 board uses a DDR4-2400MT DRAM, its memory controller is downgraded to run at 1,066.67MHz, which is lower than the peak 1,200MHz of the memory bus clock frequency. Therefore, we treat its memory as a DDR4-2133MT (twice the memory controller frequency) DRAM. As shown in Table 1, for the DRAM in the ZCU104 board, its memory bus width is 64 bits, its bus transfer rate is 2,133MT/s, and its theoretical peak bandwidth is 17GB/s.

The embedded ZCU104 board uses a **hardened memory system**, where there are six hardened AXI ports, including four High Performance (non-coherent) ports (i.e., HP[0:3]) and two High Performance Coherent ports (i.e., HPC[0:1]), on the PL side, shown in Figure 1(b). Moreover, all of these six AXI ports are of 128-bit wide (hardened). Users can only choose which AXI ports to connect with their accelerator and configure the AXI port width on the accelerator side, but not on the memory system side. Although there are four HP ports, two of them (HP1 and HP2) share a single interconnect block. And thus, only three 128-bit physical (HP) ports are connected to the memory controller. The two HPC ports are connected to the DRAM memory controller through a Cache Coherent Interconnect (CCI), which is shared with the ARM processor and has two 128-bit physical ports to the memory controller.

## 3 MICROBENCHMARK DESIGN

### 3.1 Key Factors on Memory Performance

We have identified the following set of key factors that could affect the memory system performance.

1. *Kernel clock frequency*, i.e., the clock frequency of the accelerator design.
2. *Number of concurrent ports*, i.e., the number of concurrent AXI (memory-mapped) or AXIS (streaming) ports.
3. *Port width*, i.e., the bit-width of each AXI or AXIS port.
4. *Port max_burst_length*, i.e., the maximum burst access length (in terms of number of elements) for each AXI port. The derived *port max_burst_size = max_burst_length × port width.*
5. *Consecutive data access size*, i.e., the size of consecutive data accesses for each AXI or AXIS port.

In addition to the above five key factors, we also explored one more factor: the *num_outstanding_access* of an AXI port, which specifies the number of outstanding memory access requests that an AXI port can hold before stalling the system. However, we do not include it in the key factor list, because the default *num_outstanding_access* (16) that Vitis HLS [40] uses already achieves the best memory performance (results omitted). The study in [14] only considered the *port width* of a single AXI port, and another study in [42] only considered the *port width* and *consecutive data size* of a single AXI port. To the best of our knowledge, we are the first to characterize the memory system performance under this comprehensive set of factors.

### 3.2 Design Challenges and Solutions

Our goal is to figure out how those identified key factors quantitatively impact the bandwidth, latency, and resource usage of off-chip memory access by accelerators and accelerator-to-accelerator streaming, and provide insights for software programmers. Unfortunately, HLS report [40] always assumes that an accelerator can achieve the theoretical peak bandwidth at a given port width and kernel clock frequency, but ignores other factors. For register-transfer level (RTL) hardware emulation [39], it does not accurately model the off-chip memory system. Therefore, we decide to develop a set of microbenchmarks to run on real hardware for the evaluation. For each microbenchmark, we develop the FPGA kernel code using Vitis HLS-C/C++ [40] and the CPU host code using OpenCL in the Xilinx Vitis tool [39]. To ensure accuracy, we limit our microbenchmarks to only use the most primitive operations, i.e., off-chip memory read or write, addition, and streaming read or write. Note this does not include the logic introduced by the loop mechanism used in the kernels. However, several design challenges arise from this simplicity of the microbenchmarks.

```
void krnl_ubench_RD (volatile ap_int<W>* in0,
        volatile ap_int<W>* in1,
       const int data_length, ap_int<W>* sum) {
#pragma HLS INTERFACE m_axi port=in0 bundle=gmem0
        max_read_burst_length=16...
#pragma HLS INTERFACE m_axi port=in1 bundle=gmem1
        max_read_burst_length=16...
...
     volatile ap_int<W> sum_data_0, sum_data_1;

#pragma HLS DATAFLOW
     //Repeat NUM_ITERATIONS times: read data_length
             number of consecutive data
     RD_in_0: for (int i = 0; i < NUM_ITERATIONS; ++i)
          for (int j = 0; j < data_length; ++j)
               #pragma HLS PIPELINE II=1
               sum_data_0 += in0[j];
     RD_in_1: for (int i = 0; i < NUM_ITERATIONS; ++i)
          for (int j = 0; j < data_length; ++j)
               #pragma HLS PIPELINE II=1
               sum_data_1 += in1[j];
     //Return dummy sum values
     sum[0] = sum_data_0;
     sum[1] = sum_data_1;
}
```

Listing 1. Sample HLS code to characterize the off-chip memory read bandwidth with two concurrent AXI ports

```
#pragma HLS INTERFACE m_axi port=in0 bundle=gmem0...
#pragma HLS INTERFACE m_axi port=in1 bundle=gmem0...
...
     // Option 1 - array-wise access switching
     for (int j = 0; j < data_length; ++j)
          #pragma HLS PIPELINE II=1
          sum_data_0 += in0[j];
     for (int j = 0; j < data_length; ++j)
          #pragma HLS PIPELINE II=1
          sum_data_1 += in1[j];

     // Option 2 - element-wise access switching
     for (int j = 0; j < data_length; ++j)
          #pragma HLS PIPELINE II=1
          sum_data_0 += in0[j];
          sum_data_1 += in1[j];
```

Listing 2. Sample HLS code to measure the off-chip read bandwidth of two input arrays sharing a single AXI port

1. The very short execution time of the microbenchmark kernel on the FPGA is very difficult to be accurately measured from the host CPU. We solve this issue by repeating the execution of the microbenchmark code a number of times within the FPGA kernel and then getting the average execution time. This also generalizes the variability in DRAM refresh cycle.
2. The highly repetitive accesses of simple memory operations in the microbenchmark kernel are treated as dead code and can be optimized away by Vitis HLS. From our previous work in [24], we fix this in Xilinx Vitis 2019.2 [39] by adding the *volatile* qualifier for those involved variables. In this work, as we move to Xilinx Vitis 2020.2 [39], adding the *volatile* qualifier alone cannot avoid this issue. We solve this by introducing a dummy variable to return the sum of all accessed data at the end of the benchmark.
3. By default, Vitis HLS schedules multiple loops, even independent loops, to execute in sequential. This prevents the testing of the impact by multiple concurrent ports distributed across multiple loops. To solve this issue, we use the *DATAFLOW* pragma to schedule multiple independent loops to run in parallel.

## 3.3 Microbenchmarks for Bandwidth Test

In our microbenchmark suite, we evaluate two types of data communication bandwidths. One is the off-chip memory access bandwidth used by the accelerators, using the memory-mapped AXI ports [35]. The other is the on-chip accelerator-to-accelerator streaming bandwidth using the AXIS streaming ports [35]; this is a relatively new feature added since Xilinx Vitis 2019.2 [39]. By default, 300MHz and 150MHz are used as the target clock frequencies for synthesizing accelerator designs on the datacenter and embedded FPGAs, respectively. To vary the clock frequency of the designs, we specify and pass the desired kernel frequency configuration with the "*−−kernel_frequency*" compiler option to Xilinx Vitis [39].

*3.3.1 Off-Chip Memory Bandwidth with Multiple AXI Ports.* List 1 presents a sample HLS code to characterize the relation between the off-chip memory read bandwidth and the proposed factors.

1. Line 1: the input arrays *in0* and *in1* use the *ap_int<W>* type in HLS [40], where $W$ defines the AXI port width and needs to be statically set at compile time. The *volatile* keyword ensures that the memory accesses are not optimized away by Vitis HLS.
2. Line 2: the input variable *data_length* specifies the number of consecutive array elements accessed from the off-chip memory. The consecutive data access size is $data\_length \times W/8$ bytes. The output array *sum* returns the dummy sum variable values for each input arrays to avoid dead code elimination.
3. Lines 3-4: the unique bundle names such as *gmem0* and *gmem1* indicate that the input arrays *in0* and *in1* use two physical AXI ports. Each AXI port makes independent access requests to the off-chip memory. The parameter $max\_read\_burst\_length$ = 16 specifies that the maximum number of elements during burst access is 16, i.e., *port* $max\_burst\_length$ *= 16.*
4. Line 6: dummy variables are declared to store and accumulate the sum of the data read from off-chip memory. The *volatile* keyword is also needed.
5. Line 8: the *DATAFLOW* pragma is used to schedule the *RD_in_0* loop and *RD_in_1* loop to execute in parallel. Therefore, these two loops will concurrently access two AXI ports.
6. Lines 10-13: the inner loop keeps reading a $data\_length$ size of consecutive array elements from the AXI port *gmem0* in a pipelined fashion, with an initiation interval (II) of 1. The outer loop repeats this for *NUM_ITERATIONS* times to get accurate execution time. Lines 14-17 access the second concurrent port.

*3.3.2 Off-Chip Memory Bandwidth with Multiple Arguments Sharing a Single AXI Port.* List 2 presents a sample HLS code to characterize the off-chip memory bandwidth for reading multiple data arrays using a shared AXI port, under the aforementioned factors.

1. Lines 1-2: the same bundle name *gmem0* indicates that both input arrays *in0* and *in1* are read from a shared AXI port.
2. Option 1, lines 4-10: the first j loop reads the entire array *in0*, and then the second j loop reads the entire *in1*. Note since *in0* and *in1* share the same AXI port, they cannot be accessed concurrently.
3. Option 2, lines 12-16: for each iteration of the j loop, it reads one element of arrays *in0* and *in1* in an interleaved fashion. Option 2 is a widely used coding style for software programmers.

*3.3.3 Accelerator-to-Accelerator Streaming Bandwidth with AXIS Ports.* To characterize the bandwidth for the inter-accelerator streaming connection, two kernels are used to emulate a stream write and read pair. The streaming variables are of type *hls::stream*, and each streaming data item is defined using the ap_axiu<W,0,0,0> type in HLS [40], where W specifies the bit-width of the AXIS streaming port. We omit its code due to space constraints. Similarly, the number of AXIS ports, *data_length*, and port width $W$ can be adjusted to test different configurations as described in Section 3.3.1.

## 3.4 Microbenchmarks for Latency Test

In the latency test, we aim to measure the performance of the random access latency of the off-chip memory. To access the off-chip data in a random order, a random index array is generated in the CPU host code and then stored on the FPGA on-chip BRAM. Using this on-chip random index array, the input array can be accessed from the off-chip memory in a random order. Moreover, to avoid multiple memory accesses overlapping their latency with each other, we set $num\_outstanding\_access$ = 1.

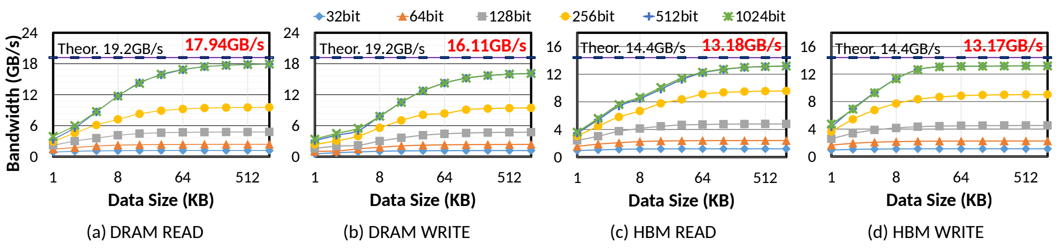

Fig. 2. Bandwidth (datacenter) for accessing a single DDR4 and HBM bank with a single AXI port at 300MHz kernel clock frequency. Note the x-axis is plotted in $log_2$ scale.

## 4 RESULTS AND INSIGHTS ON DATACENTER FPGAS

In this section, bandwidth characterizations and resource usage results for datacenter FPGAs are presented followed by a summary of design insights for both the memory mapped AXI port and the streaming port. Lastly, the latency result is presented for the memory mapped AXI port.

### 4.1 Experimental Setup

As discussed in Section 2, the platforms we evaluate are the Xilinx Alveo U200 and U280 datacenter FPGA boards [37, 38]. We built our HLS C/C++ based microbenchmarks using Xilinx Vitis 2019.2 [39] in [24], and confirm the results using Xilinx Vitis 2020.2 in this work. The FPGA kernels of these microbenchmarks run at 300MHz unless otherwise specified. The DRAM results and accelerator-to-accelerator streaming results apply to both Alveo U200 and U280 FPGAs, and the resource utilization is based on Alveo U200 FPGA.

### 4.2 Bandwidth and Resource Utilization for Single Argument AXI Port Off-Chip Access

*4.2.1 Bandwidth Results at 300MHz Kernel Clock Frequency.* For a single AXI port (and single argument) running at 300MHz *kernel clock frequency* with the default *max_burst_length* (16) and *num_outstanding_access* (16), the effective DRAM and HBM bank read and write bandwidths under different port widths and consecutive data access sizes are shown in Figure 2. A similar trend is observed from both off-chip memory read and write.

1. Both bandwidths increase almost linearly as the port width increases and flattens at 512-bit width because it is the best port width to achieve the peak effective memory bandwidth when running at 300MHz kernel clock frequency. Note that even each AXI port on the HBM memory system side is hardened to be 256-bit wide, it runs at 450MHz; that is why the effective bandwidth still flattens at 512-bit port width for the AXI port on the accelerator side (at 300MHz). We will further explain this in Section 4.2.2.
2. Both bandwidths increase as the consecutive data access size increases. The bandwidths flatten at around 128KB for DDR4 read and write and HBM read, and at 32KB for HBM write. This is because multiple sources of off-chip memory access overhead—such as the activation and precharge a DRAM row [7], page miss and address translation overhead—can be better amortized with larger consecutive data access sizes.
3. Although the theoretical peak bandwidth of the DDR4 used in this paper is 19.2GB/s, the effective peak bandwidths are 17.94GB/s for read and 16.11GB/s for write for a single AXI port. Further, they can only be achieved when the port width is 512-bit or above and the consecutive data access size is no less than 128KB. For the HBM2 used in this paper, the theoretical peak bandwidth is 14.4GB/s, yet the effective peak bandwidths are around 13.2GB/s for both read and write for

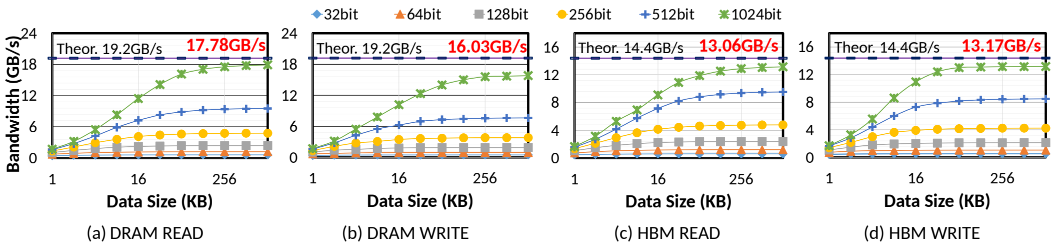

Fig. 3. Bandwidth (datacenter) for accessing a single DDR4 and HBM bank with a single AXI port at 150MHz kernel clock frequency. Note the x-axis is plotted in $log_2$ scale.

a single AXI port. This is slightly lower than the reported 13.27GB/s HBM bandwidth in [33] where RTL microbenchmark design is used.

4. For a single AXI port, increasing the *max_burst_length* and/or *num_outstanding_access* parameters beyond the default values (16) does not improve the bandwidths for DDR4 or HBM2.

### 4.2.2 *Bandwidth Results at Different Kernel Clock Frequencies.*

To evaluate the impact of kernel clock frequency on the effective off-chip bandwidth, we repeat the same test presented in Section 4.2.1, except that the accelerator kernels run at 150MHz. As shown in Figure 3, all the trends are the same as those in Figure 2, which are summarized in Section 4.2.1, except that now the bandwidths flatten at 1024-bit (instead of 512-bit) port width, which is the maximum AXI port width that Vitis HLS supports.

To understand why, we first calculate the theoretical bandwidth by a single AXI port on the accelerator side (*theoretical_port_BW*), which is determined by the AXI port width (*port_width*) and the kernel clock frequency (*kernel_freq*), as shown in the following equation:

$$theoretical\_port\_BW = port\_width \times kernel\_freq/8\text{-bit}\ (B/s) \tag{2}$$

where we assume every clock cycle *port_width* bits of data can be transferred by the AXI port. The same equation applies to calculate the theoretical streaming AXIS port bandwidth.

Since the theoretical port bandwidth has to match with the theoretical off-chip memory bandwidth (i.e., *theoretical_port_BW* == *theoretical_mem_BW*), by combining Equations 1 and 2, we derive and show that the effective AXI port width for achieving the peak off-chip memory bandwidth can be calculated with the following equation:

$$port\_width = mem\_bus\_width \times transfer\_rate/kernel\_freq \tag{3}$$

Note that in Vitis HLS [39], the AXI port width can only be configured to be power-of-two and the maximum number is 1,024. Therefore, the final optimal port width (*optimal_port_width*) can be calculated using the following equation:

$$optimal\_port\_width = min\left(pow\left(2, \left\lceil log_2 \frac{mem\_bus\_width \times transfer\_rate}{kernel\_freq} \right\rceil\right), 1024\right) \tag{4}$$

Using Equation 4, the *mem_bus_width* and *transfer_rate* parameters presented in Table 1, and the *kernel_freq* (300MHz in Figure 2 and 150MHz in Figure 3), we can calculate the optimal AXI port widths for 300MHz and 150MHz accelerator designs to be 512 and 1024 bits, respectively, which explains what we observe in Figure 2 and Figure 3.

To further validate our Equation 4, we sweep the kernel clock frequency from 100MHz to 300MHz at a 50MHz increment, for a single 512-bit AXI port design. As shown in Figure 4, the effective read and write bandwidths generally scales linearly with the kernel clock frequency. For the HBM, at 225MHz, it already achieves the peak bandwidths according to Equation 4. For the rest of this section, we will only present results running at 300MHz kernel clock frequency.

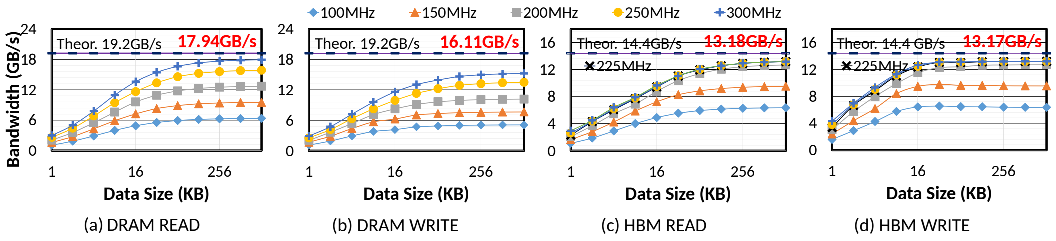

Fig. 4. Bandwidth (datacenter) for accessing a single DDR4 and HBM bank with a single 512-bit AXI port. Note the x-axis is plotted in $log_2$ scale.

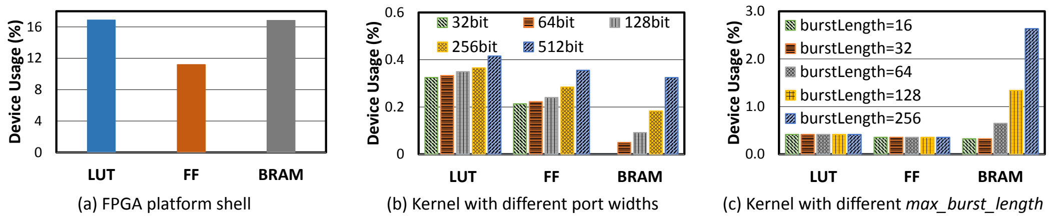

Fig. 5. U200 resource usage of a single AXI port design with different port widths and *max_burst_lengths* (port width = 512 bits).

*4.2.3 Resource Usage.* Figure 5 presents the resource usage of LUT, FF, and BRAM for the FPGA platform shell and the AXI port, under different port widths and *max_burst_length* values. Note that the consecutive data access size is a software parameter and does not change the hardware implementation. Also note that the resource utilization is similar at different kernel clock frequencies for our simple microbenchmarks. Throughout this paper, we report the post place and routing resource utilization.

1. Shown in Figure 5(a), the platform shell resource usage remains unchanged as there is no additional AXI port connection.
2. Shown in Figure 5(b), as the AXI port width increases, the LUT and FF resources have a steady small percentage increase while the BRAM usage grows linearly. The total resource increase is less than 0.3% of the entire device across all three resources.
3. Shown in Figure 5(c) where it uses 512-bit port width, as the *max_burst_length* increases, the LUT and FF resources remain unchanged, but the BRAM usage grows linearly. The reason is that the AXI port automatically buffers some data that it reads from off-chip DRAM in on-chip FIFO or BRAM [35]. This buffer size can be calculated using the following equation [40]:

$$AXI\_buf\_size = port\_width \times max\_burst\_length \times num\_outstanding\_access \tag{5}$$

This equation explains the linear increase of BRAM resource in Figure 5(b) and (c). For the 32-bit case in Figure 5(b), it uses FFs instead of BRAMs due to its small buffer size. For the $max\_burst\_length$ = 16 case in Figure 5(c), each BRAM bank is only occupied by half, and that is why it uses the same number of BRAMs as the $max\_burst\_length$ = 32 case.

## 4.3 Bandwidth and Resource Utilization for Multiple AXI Ports Off-Chip Access

*4.3.1 Bandwidths for Multiple AXI Ports in Single DRAM.* We first evaluate the bandwidth of multiple concurrent AXI ports accessing a single DRAM where the aggregated port width is 512 bits (the best port width). Figure 6 presents the results for 2 ports each 256-bit wide, 4 ports each 128-bit wide, and 8-ports each 64-bit wide.

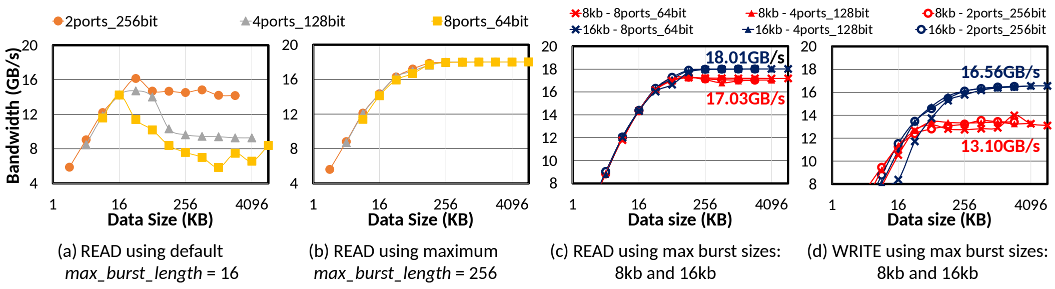

Fig. 6. Bandwidth (datacenter) for accessing a single DDR4 DRAM with multiple AXI ports and a total port width of 512-bit, at 300MHz.

1. Figure 6(a) presents the bandwidth with *max_burst_length* = 16 (HLS default). As the consecutive data access size increases, the bandwidths for 2 ports, 4 ports, and 8 ports get an unstable degradation. This is mainly because the DRAM row buffer for one AXI port access will be replaced by another AXI port access before it is fully utilized. Note there is no contention in the single port case. The more ports, the more contention on the DRAM row buffer, the more bandwidth degradation. The degraded bandwidth can be up to 3x lower than the expected bandwidth.
2. Figure 6(b) presents the bandwidth with *max_burst_length* = 256 (maximum number supported in HLS). The bandwidths for 2 ports, 4 ports, and 8 ports are almost identical to the single 512-bit port case, because they can fully utilize the DRAM row buffer locality (explained below using *max_burst_size*). However, according to Equation 5, a design with *max_burst_length* = 256 can lead to up to about 8x more usage of the on-chip BRAM resource.
3. To find out the optimal *max_burst_length* that achieves the best bandwidth and uses least amount of BRAM, we test a full set of configurations with different *max_burst_lengths* and port widths. We find this is actually determined by the derived factor *port max_burst_size* (i.e., *max_burst_length* × *port width*).
   Figure 6(c) and (d) present the effective read and write bandwidths for 2 ports, 4 ports, and 8 ports under 8Kb and 16Kb *max_burst_sizes* (we omit other *max_burst_sizes* to make the figures more clear to read). Note the effective peak read and write bandwidths for multiple AXI ports (18.01GB/s and 16.56GB/s) are slightly higher than those for a single AXI port.
   Based on the Figure 6(c) and (d), we conclude that 16Kb is the best *max_burst_size*. The reason is that the DRAM row buffer size is 512B and the page size is 4KB. A *max_burst_size* of 16Kb (2KB) is large enough to fully utilize the DRAM row buffer locality and amortize the page miss and address translation overhead.

We also measure other combinations with different number of ports and different port widths. Here is a summary of our findings for multiple AXI port designs in HLS.

1. The maximum number of AXI ports that can be connected to each single DRAM is 15.
2. To achieve the best stable bandwidth, the *max_burst_size* for all AXI ports should be set as 16Kb by setting the *max_burst_length*. For AXI ports whose port widths are under 64 bits, i.e., where 16Kb *max_burst_size* cannot be achieved, the maximum *max_burst_length* = 256 should be used.
3. As long as the best $max\_burst\_size = 16Kb$ is used, the total bandwidth scales linearly with the aggregated port width and flattens at 512-bit width. It also increases as the consecutive data access size increases and is about to flatten at 128KB size.

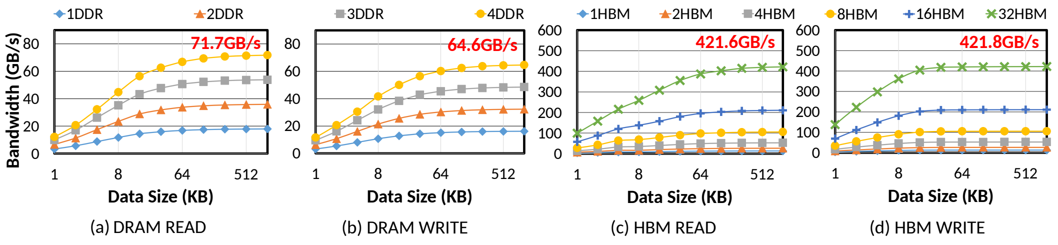

Fig. 7. Bandwidth (datacenter) for accessing multiple DDR4 and HBM banks using AXI ports at 300MHz. Note the x-axis is plotted in $log_2$ scale.

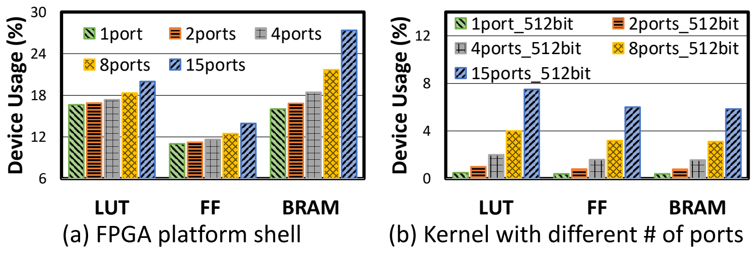

Fig. 8. U200 resource usage for multiple DRAM AXI ports.

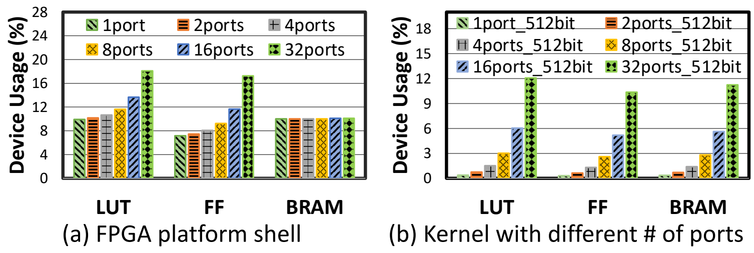

Fig. 9. U280 resource usage for multiple HBM AXI ports.

### 4.3.2 *Bandwidths for Multiple Off-Chip Memory Banks.*

Figure 7(a) and (b) present the effective off-chip bandwidths on the Alveo U200 FPGA with multiple DRAMs, each of which has a single AXI port with 512-bit width. The effective DRAM bandwidth scales linearly with the number of memory banks. The bandwidths for the four DDR4s on U200 peak at 71.7GB/s and 64.6GB/s for read and write.

Figure 7(c) and (d) present the effective off-chip bandwidths on the Alveo U280 FPGA with multiple HBM banks, where each HBM bank uses a single AXI port with 512-bit port width (on the accelerator side) at 300MHz. The effective HBM bandwidth also scales linearly with the number of memory banks. The bandwidths for the 32 HBM banks on the Alveo U280 peak at around 422GB/s for both read and write. Note that the maximum number of AXI ports supported for HBM is also 32. Therefore, a typical usage is to connect one AXI port to one HBM bank to maximize the bandwidth utilization, instead of connecting multiple AXI ports to a single HBM bank (we omit this result).

### 4.3.3 *Resource Usage for Multiple AXI Ports in Single DRAM.*

As shown in Figure 8, when the number of AXI ports connected to the DRAM increases (the maximum is 15 for a single DRAM), the resource usage increases in both the U200 FPGA platform shell and the kernel design. Between a single-port design and a 15-port design, the FPGA platform shell requires a device usage increase of 4% in LUT, 3% in FF, and 11% in BRAM. For the kernel design, the resource usage increases linearly with the number of ports.

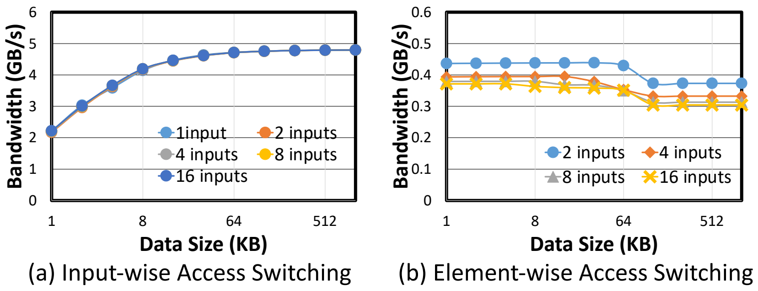

Fig. 10. Read bandwidth (datacenter) for accessing multiple input data arrays via a single shared 128-bit AXI port at 300MHz.

*4.3.4 Resource Usage for Multiple AXI Ports Accessing Multiple HBM Banks.* Figure 9 presents the resource usage for the U280 FPGA platform shell and kernel designs, where each AXI port is connected to a separate HBM bank. For the FPGA platform shell, the usage in LUT and FF scale linearly at 0.25% and 0.30% per AXI port. Nevertheless, the BRAM usage does not vary due to hardened HBM memory interconnects for the U280 FPGA. For the kernel design, the resource usage has a similar trend as the DRAM kernels in Figure 8(b), which increases linearly with the number of ports.

## 4.4 Off-Chip Bandwidth for Multiple Arguments Sharing a Single AXI Port

Figure 10 presents the read bandwidth when accessing multiple input data arrays on the DDR4 DRAM through a shared 128-bit AXI port.

1. Shown in Figure 10(a), the read bandwidth for accessing each input data array consecutively in separate loops (Option 1 in Section 3.3.2) follows the same trend as the single AXI port bandwidth (Section 4.2). This bandwidth does not vary as the number of input data array increases, because the consecutive memory burst access for each input array still retains an efficient utilization of the DRAM row buffer. Note that the bandwidth degradation issue for multiple AXI ports sharing a single DRAM in Section 4.3.1 does not apply here because multiple arguments are accessed in a sequential and consecutive way via a single AXI port. So we just need to use the default *max_burst_length*.
2. Shown in Figure 10(b), the read bandwidth for accessing multiple input data array elements in an interleaved fashion in a single loop (Option 2 in Section 3.3.2) is less than 10% of that for Option 1. This is due to the DRAM row buffer thrashing from frequent input array access switching. The bandwidth also decreases slightly as the number of input array and the array size increase. A software programmer needs to pay special attention to avoid this common coding style and choose Option 1 instead.

## 4.5 Bandwidth and Resource Utilization for Accelerator-to-Accelerator Streaming Ports

*4.5.1 Bandwidth Results.* Figure 11 presents the accelerator-to-accelerator streaming bandwidths at 300MHz kernel clock frequency. Similar to the AXI port bandwidth presented in Section 4.2.2, the streaming AXIS bandwidth scales linearly with the kernel clock frequency. We omit the results in this subsection, but include the streaming bandwidths at 150MHz for the embedded ZCU104 FPGA board in Figure 20 in Section 5.5, which are roughly half of the bandwidths shown here in Figure 11.

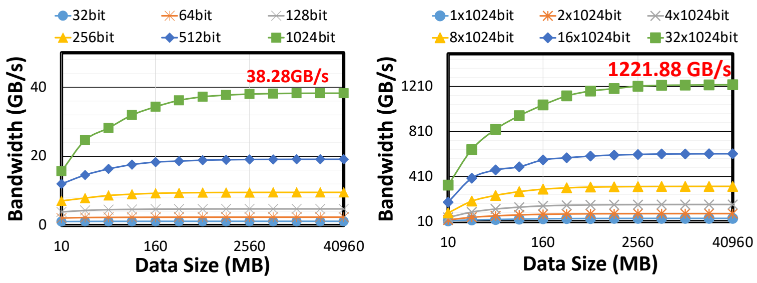

Fig. 11. Bandwidth (datacenter) for accessing multiple AXIS ports at 300MHz.

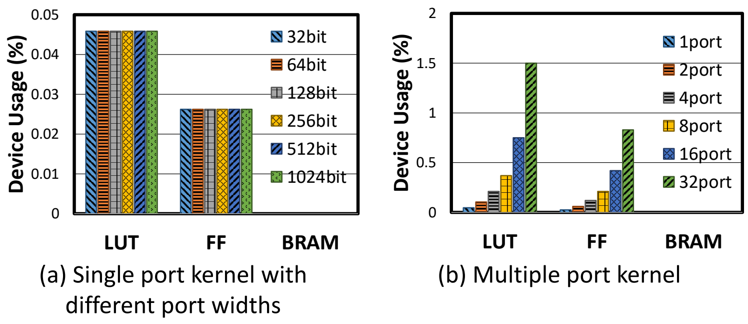

Fig. 12. U200 resource usage of a multiple AXIS ports design.

1. Shown in Figure 11(a), the effective inter-kernel streaming bandwidth scales linearly with the AXIS port width all the way to the maximum 1024-bit width that HLS supports.
2. Shown in Figure 11(a), the effective inter-kernel streaming bandwidth increases as the streaming data size increases, and flattens at around 2GB size: the peak effective bandwidth is 38.28GB/s for a single AXIS port, which is almost the same as the theoretical peak AXIS bandwidth (38.4GB/s = 1024 bit/cycle × 300MHz/8-bit B/s) calculated using Equation 2.
3. Shown in Figure 11(b), the effective inter-kernel streaming bandwidth scales linearly as the number of AXIS ports increases. In [24], with Xilinx Vitis 2019.2, this bandwidth flattens until 16 AXIS ports are used and achieves about 612.84GB/s for 16 of 1024-bit wide AXIS ports, no matter which port width is used. In this work, we observe this limitation has been addressed in Xilinx Vitis 2020.2 [39]. The streaming bandwidth now scales linearly with the number of AXIS ports as long as there is enough device resource, and can achieve 1221.88 GB/s with 32 ports.

*4.5.2 Resource Usage.* Shown in Figure 12(a), a single AXIS port has a fixed tiny resource usage regardless of its port width: it uses 0.045% of LUT and 0.026% of FF. The resource usage scales linearly as the number of AXIS ports increases, as shown in Figure 12(b). Therefore, accelerator-to-accelerator streaming is very efficient.

## 4.6 Latency for Off-Chip Access

Using our proposed microbenchmark, Table 2 summarizes the latencies for randomly accessing 4 bytes of data from the off-chip DDR4 and HBM2 banks. We use the minimum HLS setting for the AXI port: $max\_burst\_length = 2$ and $num\_outstanding\_access = 1$, to eliminate performance gain from DRAM row buffer locality and parallel memory accesses. As a result, using our microbenchmark, we find the latencies for DDR4 and HBM2. They are nearly the same: 110ns for DDR4 access and 108ns for HBM access. Our latency results represent the page-hit performance as presented in [33], because our microbenchmark uses a small data array size and repeat the test for many times. Our measured HBM latency is similar to that measured in [33] because of the efficient hardened HBM memory interconnect. However, our measured DRAM latency is higher than that measured in [33] (73.3ns) due to the overhead from the HLS-generated memory interconnect IP. Comparing with the findings from [33]'s RTL-based microbenchmarking tool, our latency results align with their page-hit result for HBM2 and their page-miss result for DDR4.

Table 2. Off-chip memory access latency (datacenter) at 300MHz

| 4 bytes of data | DDR4 | HBM2 |
|---|---|---|
| latency | 110ns / 33cycles | 108ns / 33cycles |

## 5 RESULTS AND INSIGHTS ON EMBEDDED FPGAS

In this section, bandwidth characterizations and resource usage results for embedded FPGAs are presented followed by a summary of design insights for both the memory mapped AXI port and the streaming port. In the embedded FPGA board, the AXI ports are hardened and include two major types of ports: High Performance non-coherent (HP) port and High Performance Coherent (HPC) port. Its detailed architecture layout is presented in Figure 1(b) in Section 2.

### 5.1 Experimental Setup

As discussed in Section 2, the platform we evaluate is the Xilinx Zynq UltraScale+ ZCU104 embedded board [36]. We build our HLS C/C++ based microbenchmarks using Xilinx Vitis 2020.2 [39]. The FPGA kernels of these microbenchmarks run at 150MHz unless otherwise specified.

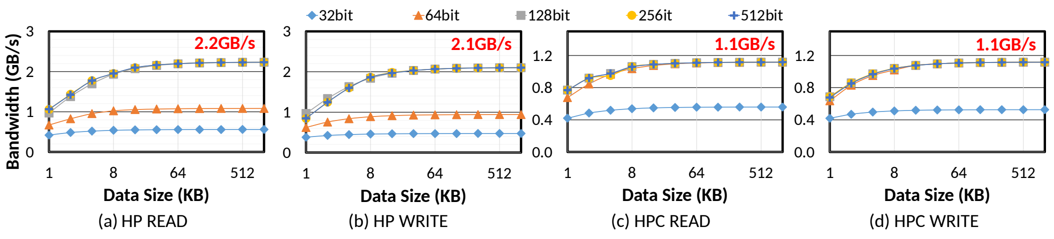

Fig. 13. Bandwidth (embedded) for a single HP or HPC AXI port accessing a DDR4 DRAM at 150MHz kernel clock frequency. Note the x-axis is plotted in $log_2$ scale.

### 5.2 Bandwidth and Resource Utilization for Single Argument AXI Port Off-Chip Access

*5.2.1 Bandwidth Results at 150MHz Kernel Clock Frequency.* For a single AXI port (and single argument) running at the default 150MHz *kernel clock frequency* with the default *max_burst_length* (16) and *num_outstanding_access* (16), the effective HP and HPC AXI port read and write bandwidths under different port widths and consecutive data access sizes are shown in Figure 13.

1. Both bandwidths increase almost linearly as the port width increases and flattens at 128-bit width via the HP port and 64-bit width through the HPC port. For HP port, this is because the AXI interface is hardened and the maximum width is fixed at 128-bit width. For HPC port, even though HPC ports have the same specifications as the HP ports (i.e., hardened AXI interface and 128-bit maximum width), the peak memory bandwidths flatten at 64-bit[1]. We believe this is due to the cache invalidation overhead from the CCI (cache-coherent interconnect).

[1]Xilinx confirms and produces the same behavior as ours.

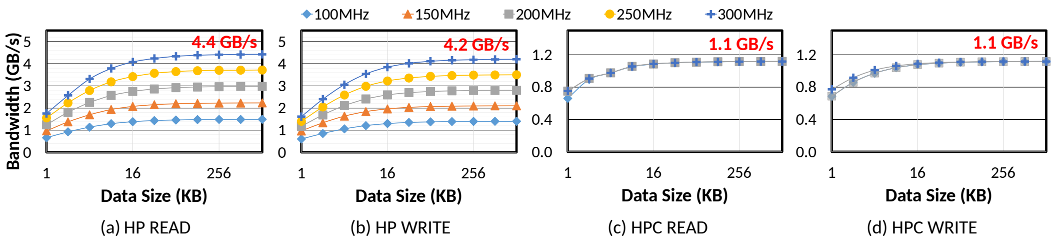

Fig. 14. Bandwidth (embedded) for a single 128-bit HP or 64-bit HPC AXI port accessing a DDR4 DRAM. Note the x-axis is plotted in $log_2$ scale.

2. Both bandwidths increase as the consecutive data access size increases. The bandwidths flatten at around 64KB for HP port read and write, and at 32KB for HPC port read and write. This is because multiple sources of off-chip memory access overhead—such as the activation and precharge a DRAM row [7], page miss and address translation overhead—can be better amortized with larger consecutive data access sizes.
3. Based on Equation 2, for a kernel that runs at 150MHz, the theoretical peak bandwidth for the 128-bit HP port and 64-bit HPC port is around 2.4GB/s and 1.2GB/s, respectively. The effective peak bandwidths of a single HP port are 2.2GB/s for read and 2.1GB/s for write. Further, these bandwidths can only be achieved when the port width is 128-bit or above, and the consecutive data access size is no less than 64KB. The effective peak bandwidths of a single HPC port for both read and write are 1.1GB/s. And these bandwidths can only be achieved when the port width is 64-bit or above, and the consecutive data access size is no less than 32KB.
4. For a single HP or HPC port, increasing the *max_burst_length* and/or *num_outstanding_access* parameters beyond the default values (16) does not improve the bandwidths.

*5.2.2 Bandwidth Results at Different Kernel Clock Frequencies.* To evaluate the impact of the kernel clock frequency on the effective off-chip bandwidth, we repeat the same test presented in Section 5.2.1 on the best performing port width of 128-bit for the HP port and 64-bit for the HPC port, while varying the kernel clock frequency from 100MHz to 300MHz at a 50MHz increment. Shown in Figure 14, a similar trend is observed from both off-chip memory read and write.

1. Shown in Figure 14(a) and (b), for the 128-bit HP AXI port, the bandwidths scales linearly with the kernel clock frequency and achieve a peak bandwidth of 4.4 GB/s for read and 4.2 GB/s for write at 300MHz. This is in line with Equation 2 as presented in Section 4.2.2. Note that according to Equation 4, the optimal AXI port width to achieve the peak effective DRAM bandwidth at 300MHz should be 512 bits; therefore, using one single hardened HP port that is 128-bit wide significantly underutilizes the DRAM bandwidth. In Section 5.3, we will present results that utilize multiple hardened HP and/or HPC ports to better utilize the DRAM bandwidth.
2. Shown in Figure 14(c) and (d), for the 64-bit HPC AXI port, the bandwidths do not change with the kernel clock frequency and always give a peak bandwidths of 1.1 GB/s for read and write. We believe this is due to the cache invalidation overhead from the CCI.

*5.2.3 Resource Usage.* Figure 15 presents the resource usage of LUT, FF, and BRAM for the FPGA platform shell and the AXI port. Note that the resource results apply to both HP and HPC ports. In addition, the consecutive data access size is a software parameter and does not change the hardware implementation. Throughout this paper, we report the post place and routing resource utilization.

1. Shown in Figure 15(a), the platform shell resource usage remains unchanged as there is no additional AXI port connection.

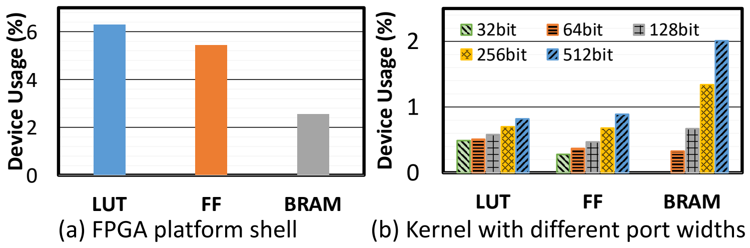

Fig. 15. ZCU104 resource usage of a single (HP or HPC) AXI port design with different port widths.

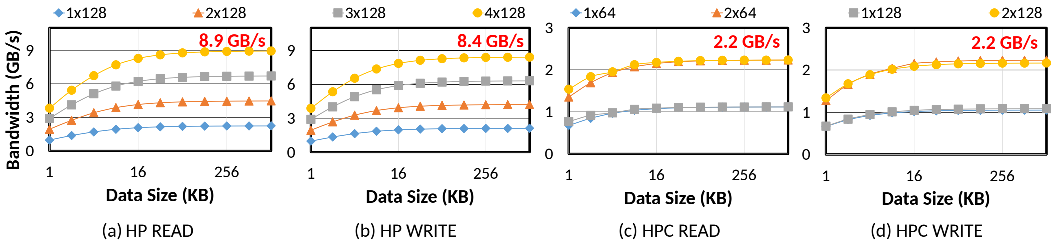

Fig. 16. Bandwidth (embedded) for accessing the off-chip DRAM with multiple AXI ports running at 150MHz. Note the HP port width is 128-bit and HPC port width is 64-bit.

2. Shown in Figure 15(b), as the AXI port (on the accelerator side, connecting to either HP or HPC port) width increases, the LUT and FF resources have a steady small percentage increase while the BRAM usage grows linearly. The total resource increase is less than 1.8% of the entire device across all three resources.
3. The BRAM usage grows linearly as the *max_burst_length* increases, as presented in Equation 5 in Section 4.2.3. We omit this result, as in the embedded FPGA, the HP and HPC AXI ports are hardened and users do not need to tune *max_burst_length* even with multiple AXI ports.

## 5.3 Bandwidth and Resource Utilization for Multiple AXI Ports Off-Chip Access

*5.3.1 Bandwidths for Multiple HP AXI Ports at 150MHz Kernel Frequency.* We first evaluate the bandwidth of multiple concurrent HP AXI ports accessing the same DRAM, each port width being 128-bit (the best port width). Figure 16(a) and (b) present the bandwidth results when the number of HP port varies from one to four at 150MHz. The bandwidths scale linearly with the number of HP ports and peak at 8.9GB/s for read and 8.4 GB/s for write with four HP ports. This is because each individual HP port is hardened in the FPGA and at 150MHz kernel clock frequency, their aggregated port width (512-bit = 4 × 128-bit) is significantly less than the effective port width (910-bit = 64-bit × 2133MT/s / 150MHz) calculated using Equation 3 for accessing the off-chip DRAM.

*5.3.2 Bandwidths for Multiple HPC AXI Ports at 150MHz Kernel Frequency.* When using multiple HPC ports concurrently, shown in Figure 16(c) and (d), the effective off-chip bandwidths also scale with the number of HPC ports and peak at 2.2GB/s for both read and write when using two 64-bit (the best effective port width shown in Section 5.2) HPC ports, at kernel clock frequency of 150MHz. This is because ZCU104 has two independently hardened HPC ports and their aggregated port width (128-bit) is significantly less than the effective port width (910-bit) for accessing the DRAM. To further confirm that 64-bit is the best effective port width for the HPC port, we also evaluate the read and write bandwidths using 128-bit HPC port, where the effective bandwidths do not improve as shown in Figure 16(c) and (d).

*5.3.3 Maximum DRAM Bandwidth using both HP and HPC AXI Ports at 150MHz Kernel Frequency.* While it might seem straightforward to derive the maximum achievable off-chip DRAM bandwidth (11.1GB/s at 150MHz) by adding the peak bandwidth of all HP ports (8.9GB/s in Figure 16(a)) and all HPC ports (2.2GB/s in Figure 16(c)), in practice, it is nontrivial to achieve (or measure, at least) this maximum achievable bandwidth. To ensure accurate bandwidth measurements, all concurrently accessed ports (i.e., all memory access loops shown in Listing 1) should start at exactly the same time and end at exactly the same time. Since the bandwidths (i.e., data transfer rates) are different between the HP and HPC ports, we need to adjust the ratio of the consecutive data size transferred between the two ports in order to match their memory access time. To do so, we apply both *static* and *dynamic* approaches.

For the *static approach*, we statically adjust the consecutive data size for the HPC port to match the access time for the HP port based on the single port bandwidth results shown in Figure 13. This approach only uses a single microbenchmark kernel, which contains all the concurrent memory access loops for HP and HPC ports. The results of this approach are shown as the solid dark blue and red lines (for HP ports plus HPC ports) in Figure 17. Note the results from the static approach may deviate from the actual bandwidth due to runtime heuristics in the memory controller when scheduling requests from multiple AXI ports.

We also apply the *dynamic approach* to compensate for the runtime scheduling heuristics in the memory controller, where two microbenchmark kernels (one for HP ports and the other for HPC ports) are used to run in concurrent and on the host side, we dynamically tune the consecutive data size for the HPC port over multiple iterations until the kernel execution times of both kernels are around the same. The results of this approach are shown as the dashed dark blue and red lines (for HP ports plus HPC ports) in Figure 17. Nevertheless, the dynamic approach also has a drawback: the Xilinx Runtime (XRT) overhead from executing and scheduling multiple kernels could cause bandwidth fluctuation.

We evaluate the aggregated read bandwidth when concurrently utilizing both HP and HPC ports with both the static and dynamic approaches at 150MHz and 300MHz kernel clock frequencies. Here we present the results at 150MHz in Figure 17(a) and we will further present the results at 300MHz in Section 5.3.4.

1. At 150MHz kernel clock frequency, the effective AXI port width to fully utilize the DRAM bandwidth is 910-bit based on Equation 3. However, in the hardened memory system of ZCU104, the best aggregated AXI port width is 640-bit using all four 128-bit HP ports and two 64-bit HPC ports. Assuming there is a linear scaling between the bandwidth and AXI port width, the best achievable bandwidth is around 12.0GB/s, i.e., 70.3% (640-bit / 910-bit) of the theoretical peak bandwidth (17GB/s) of the off-chip DRAM.
2. With the static approach, the design with four 128-bit HP ports plus one 64-bit HPC port achieves a steady 1GB/s bandwidth improvement compared to the four HP port design, due to the additional HPC port. However, for the four 128-bit HP ports plus two 64-bit HPC ports design, the bandwidth plummets after reaching a peak bandwidth of 10.4GB/s, because memory access times between the HP and HPC ports become mismatched due to runtime heuristics in the memory controller when scheduling requests from multiple AXI ports.
3. With the dynamic approach, it improves the bandwidth of the four 128-bit HP ports plus two 64-bit HPC ports design compared to the static approach for larger consecutive data sizes, with some bandwidth fluctuation caused by the XRT overhead from executing and scheduling multiple concurrent kernels. Such XRT overhead even leads to that the dynamic approach performs worse than the static approach for the four 128-bit HP ports plus one 64-bit HPC ports design. At 150MHz kernel frequency, compared to the best achievable bandwidth of 12.0GB/s,

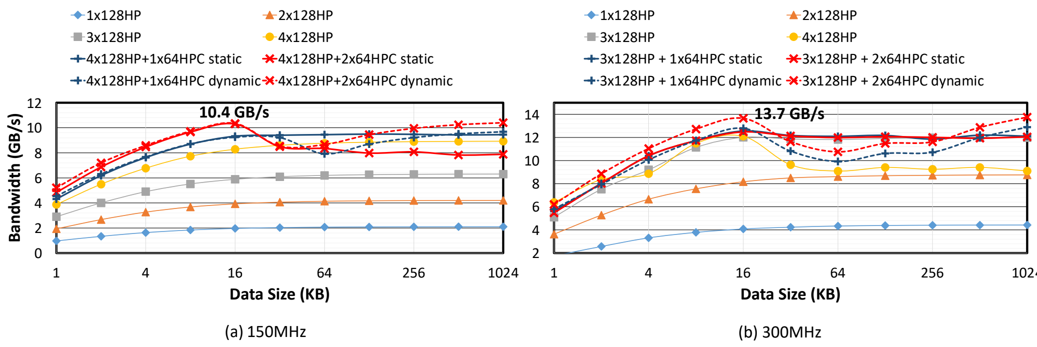

Fig. 17. Aggregated bandwidth (embedded) for reading the DRAM using multiple HP and HPC AXI ports.

the aggregated peak effective read bandwidth is around 10.4GB/s, using four 128-bit HP ports plus two 64-bit HPC ports.

*5.3.4 Maximum DRAM Bandwidth using both HP and HPC AXI Ports at 300MHz Kernel Frequency.* To evaluate the impact of kernel clock frequency on the maximum achievable bandwidths on ZCU104, we repeat the same test presented in Section 5.3.3 at 300MHz. Figure 17(b) presents the aggregated read bandwidth of utilizing both HP and HPCs ports at 300MHz, at which the effective port width becomes 455-bit (half of 910-bit for the 150MHz design) for accessing the DRAM.

1. When leveraging all four 128-bit HP ports to concurrently access the DRAM, the aggregated port width is 512-bit, which exceeds the effective port width of 455-bit and should able to fully utilize the effective DRAM bandwidth. However, from the architecture layout of ZCU104 shown in Figure 1(b), two of the HP ports (HP1 and HP2) actually share a single interconnect switch block before reaching the memory controller, which limits the bandwidth due to access contention at 300MHz. This explains why the four HP ports design (yellow trend line) even degrades the bandwidth compared to the three HP ports design (grey trend line) where only one of HP1 or HP2 is used. Since the effective DRAM port width is 455 bits and three 128-bit HP ports only utilize 384 bits, adding more HPC ports could further increase the effective bandwidth.
2. With the static approach, the bandwidth of the three 128-bit HP ports plus one or two 64-bit HPC ports design only slightly increases when each port transfers 16KB or less amount of data, and flattens at around the same bandwidth as the three HP ports design. This is due to the mismatching memory access times between the HP and HPC ports caused by runtime heuristics in the memory controller when scheduling requests from multiple AXI ports.
3. With the dynamic approach, the bandwidth further increases by adding more HPC ports compared to the three HP ports design, with some bandwidth fluctuation caused by the XRT overhead from executing and scheduling multiple concurrent kernels. The final peak effective read bandwidth using three 128-bit HP ports and two 64-bit HPC ports is around 13.7GB/s, which is about 80.6% of the theoretical peak DRAM bandwidth (17GB/s).

*5.3.5 Resource Usage for Multiple AXI Ports.* Figure 18 shows that when the number of HP ports (on the accelerator side) connected to the DRAM increases, the resource usage increases in both the ZCU104 FPGA platform shell and the kernel design. Between a single-port design and a four-port design, the FPGA platform shell requires a device usage increase of 2.9% in LUT, 3.2% in FF, and 2.9% in BRAM. For the kernel design, the resource usage increases linearly with the number of ports. A similar trend is observed for the HPC ports and we omit the results due to space constraints.

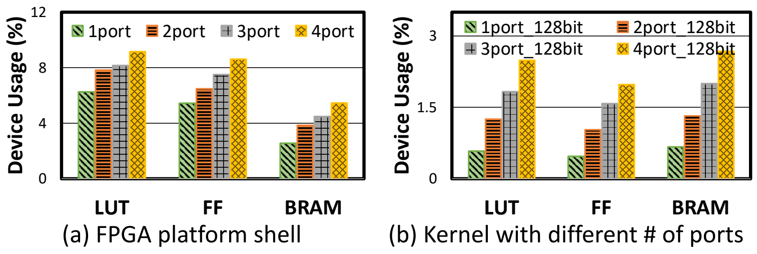

Fig. 18. ZCU104 resource usage for multiple (HP) AXI ports accessing the DRAM.

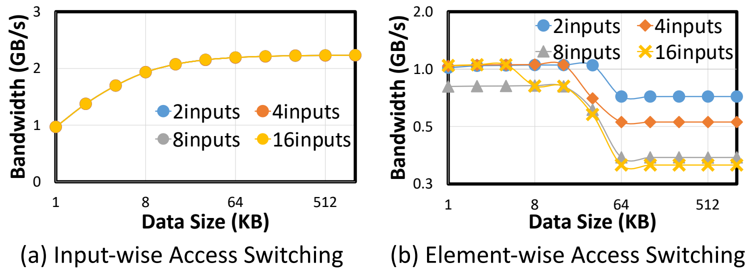

Fig. 19. Read bandwidth (embedded) for accessing multiple input arrays via a shared 128-bit HP AXI port at 150MHz.

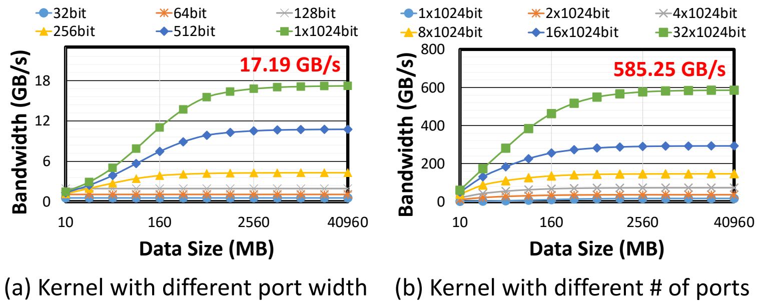

Fig. 20. Bandwidth (embedded) for accessing multiple AXIS ports at 150MHz.

## 5.4 Off-Chip Bandwidth for Multiple Arguments Sharing a Single AXI Port

Figure 19 presents the read bandwidth when accessing multiple input data arrays on the DDR4 DRAM through a shared 128-bit HP AXI port at 150MHz. A similar trend is observed as that in the datacenter FPGAs (as presented in Section 4.4), except that the bandwidth numbers are lower since here the accelerator kernels run at 150MHz. A software programmer needs to pay special attention to accessing each input data array consecutively in separate loops (Option 1 instead of Option 2 in Section 3.3.2) to better utilize the off-chip bandwidth.

## 5.5 Bandwidth and Resource Utilization for Accelerator-to-Accelerator Streaming Ports

*5.5.1 Bandwidth Results.* Figure 20 presents the accelerator-to-accelerator streaming bandwidths with kernel clock frequency of 150MHz for the single AXIS port case and multiple AXIS port case. A similar trend is observed as that in the datacenter FPGAs (as presented in Section 4.5): the effective inter-kernel streaming bandwidth scales linearly with the AXIS port width (up to 1024-bit width that HLS supports) and the number of AXIS ports. The peak effective bandwidth is 17.2GB/s for a single AXIS port, which is close to the theoretical peak AXIS bandwidth (19.2GB/s = 1024 bit/cycle

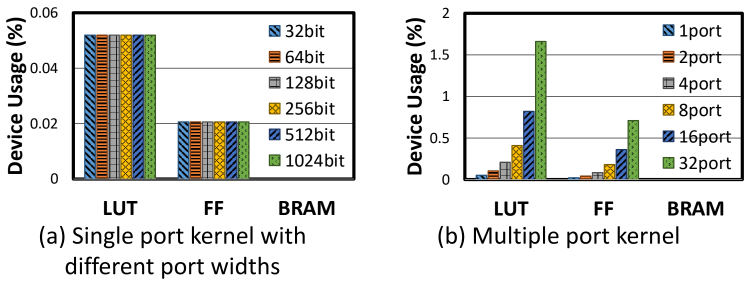

Fig. 21. ZCU104 resource usage of a multiple AXIS ports design.

× 150MHz/8-bit B/s) calculated using Equation 2. With 32 1024-bit width AXIS ports, the effective peak streaming bandwidth is about 585.2GB/s. Based on Equation 2, the streaming AXIS bandwidth scales linearly with the kernel clock frequency. We omit the bandwidth results at 300MHz in this subsection, but include the streaming bandwidths at 300MHz for the datacenter FPGA board in Figure 11 in Section 4.5, which are roughly twice of the bandwidths shown here in Figure 20.

*5.5.2 Resource Usage.* Shown in Figure 21(a), a single AXIS port has a fixed tiny resource usage regardless of its port width: it uses 0.052% of LUT and 0.021% of FF. The resource usage scales linearly as the number of AXIS ports increases, as shown in Figure 21(b). Note that AXIS ports use less resource here than that in the U200 datacenter FPGA as presented in Section 4.5.2, due to the clock frequency difference (150MHz vs 300MHz). Nevertheless, in both embedded and datacenter FPGAs, accelerator-to-accelerator streaming is very efficient and only consumes a tiny percentage of the entire FPGA resource.

### 5.6 Latency for Off-chip Access

Using our proposed microbenchmark and the same HLS setting in Section 4.6, Table 3 summarizes the latencies for randomly accessing 4 bytes of data from the off-chip DRAM via the HP and HPC AXI ports. Note that the kernel clock frequency does not affect the off-chip access latency, which involves a single memory transfer. We find that the latency for accessing the DRAM via the HP port is lower than that via the HPC port (115ns vs. 195ns). This is because the direct connection between the HP port to the memory controller is more efficient than going through a cache-coherent interconnect (CCI) for the HPC port, where cache invalidation with the ARM processor's cache occurs and adds extra latency overhead. Moreover, the DRAM latency through the HP port is comparable to the DRAM and HBM latencies on the datacenter FPGAs shown in Table 2.

Table 3. Off-chip DRAM memory access latency (embedded) at 150MHz

| 4 bytes of data | HP | HPC |
|---|---|---|
| latency | 115ns / 17cycles | 195ns / 29cycles |

## 6 DISCUSSION ON SOFT AND HARDENED MEMORY SYSTEMS ON MODERN FPGAS

Based on our quantitative studies on the five bandwidth affecting factors in Section 4 and Section 5, we summarize the similarities and differences of the memory systems between the modern datacenter (soft and users have more control) and embedded (hardened and users can only select which memory ports to connect) FPGAs, and discuss the design trade-offs to access them efficiently.

Datacenter and embedded FPGA memory systems are similar in the following aspects:

1. The effective off-chip memory bandwidth via AXI ports and on-chip accelerator-to-accelerator streaming bandwidth via AXIS ports generally scale linearly as the kernel clock frequency increases, as shown in Equation 2. And the optimal AXI port width in a soft memory system for

achieving the peak off-chip bandwidth can be calculated based on the kernel clock frequency as shown in Equation 4. In particular, for a 150MHz accelerator design that accesses a DDR4 bank on a datacenter FPGA, a 1024-bit port width should be used instead of 512-bit (which is the optimal port width for a 300MHz accelerator design) to achieve the peak bandwidth.

2. The off-chip memory bandwidth scales (almost) linearly with the number of concurrent AXI ports when there is no contention in the memory access. The on-chip accelerator-to-accelerator streaming bandwidth also scales linearly with the number of AXIS ports.
3. The off-chip memory bandwidth increases linearly as the port width increases, until it reaches either the physical limit of the AXI port width (for hardened memory system) or the optimal port width of the memory system (for soft memory system, based on Equation 4). For the on-chip accelerator-to-accelerator streaming connection, its bandwidth scales linearly with the AXIS port width until it reaches the maximum 1024-bit width that HLS supports.
4. To access the off-chip memory with a single AXI port, the effective bandwidth does not change when increasing the *max_burst_length* and *num_outstanding_access* beyond their default value (both are 16).
5. When accessing the off-chip memory in larger consecutive data size, the bandwidth improves until the data size is large enough to amortize multiple sources of off-chip memory access overhead, such as the activation and precharge a DRAM row [7], page miss and address translation overhead.

The differences between the datacenter and embedded FPGA memory systems are as follows:

1. On the evaluated ZCU104 embedded FPGA, the effective off-chip memory bandwidth via the HPC AXI port does not vary with the kernel clock frequency and flattens at 64-bit even though the hardened HPC port is 128-bit wide.
2. On the datacenter FPGAs, for a single DDR4 bank whose theoretical peak bandwidth is 19.2GB/s, with a kernel frequency of 300MHz, the effective peak read and write bandwidths are about 18.01GB/s and 16.56GB/s, which flattens at 512-bit port width. And for a single HBM2 bank whose theoretical peak bandwidth is 14.4GB/s, with a kernel frequency of 300MHz, the effective peak read and write bandwidths are about 13.18GB/s and 13.17GB/s, which also flattens at 512-bit port width. On embedded FPGAs, for the single DDR4 bank whose theoretical peak bandwidth is 17GB/s, with a 300MHz kernel frequency, the effective peak read and write bandwidths are 13.70GB/s and 13.11GB/s when using 512-bit aggregated port width across 3 HP and 2 HPC AXI ports.
3. To achieve the peak effective off-chip memory bandwidth when using multiple concurrent AXI ports, the soft memory system on datacenter FPGAs requires tuning the *max_burst_length* such that the *max_burst_size* (i.e., *max_burst_length* $\times$ *port_width*) for each AXI port equals to 16kb. For hardened memory systems such as HBM on datacenter FPGAs and DRAM on embedded FPGAs, this tuning is not needed since the AXI ports are pre-allocated on the memory side and the memory controller could be optimized for the accesses to avoid the row buffer thrashing.
4. The threshold for the consecutive data size to reach the peak effective off-chip bandwidth is around 128KB in the soft memory system, and around 64KB in the hardened memory system.

Based on our summarized similarities and differences, for most of the time, the soft memory system is preferred over the hardened memory system due to its flexibility. The soft memory system allows users to customize the number of off-chip memory AXI ports and the bit width of each port as needed to fully utilize the off-chip bandwidth, as long as it chooses the optimal aggregated port width as presented in Equation 4. For example, at a 300MHz kernel clock frequency, a single 512-bit AXI port, or two 256-bit AXI ports, or four 128-bit AXI ports, can all fully utilize the effective bandwidth of a DDR4 on datacenter FPGAs. The only thing is that users have to tune the *max_burst_length* for multi-port configurations. However, with the hardened memory system, (almost) all hardened ports need to be used to exploit the full off-chip bandwidth, which could call for significant code changes in the accelerator design; otherwise, if some hardened ports are not used by the accelerator, the off-chip bandwidth is underutilized.

In terms of resource usage, the soft memory system could have some overhead when accessing the off-chip memory with multiple concurrent ports and a large *max_burst_length* as needed. Even though, the hardened memory system may not always be the more resource efficient option. First, when an accelerator design does not use all the hardened ports, the hardware resource for those ports is wasted. On the contrary, the soft memory system can customize the ports as needed to save the resource utilization. Second, users can eliminate the resource overhead in the soft memory system by merging multiple narrow AXI ports into a single wide AXI port to access the off-chip memory, at the cost of more code changes.

In the following two sections, we present two case studies to validate the usefulness of our insights. We take the following general approach to leverage our insights in optimizing our case studies. Similar to [12, 13], we separate the computation and memory access of the accelerator design using the common load-compute-store pattern, and further balance and overlap the computation and memory access in a coarse-grained pipeline with the Ping-Pong buffer technique. To further improve the throughput of the accelerator design, we first optimize the off-chip memory access bandwidth by tuning the five impacting factors using the insights summarized above. Subsequently, we tune the pipeline initiation interval (II) and parallelization (unrolling) factor in the computation loops to balance the latencies between the memory access and computation. Lastly, we build multiple optimized design points with nearly the same execution cycles to select the best performing one with the highest clock frequency. To better demonstrate our insights, we choose to accelerate the case study designs on the Xilinx Alveo U200 datacenter FPGA that uses the more flexible soft memory system.

## 7 CASE STUDY 1: K-NEAREST NEIGHBOR

To demonstrate how to leverage our results and insights in designing an efficient HLS-based hardware accelerator on datacenter FPGAs, we conduct a case study on a K-nearest neighbors (KNN) accelerator in HLS-C/C++. The KNN algorithm [2, 8] is widely used in many applications such as similarity searching, classification, and database query search [17, 30, 41]. We study two implementations for the KNN accelerator, one is compute-bounded and the other is memory-bounded. Finally, our bandwidth-optimized design is able to leverage the entire FPGA DRAM bandwidth and achieve a 6x speedup over the 24-thread CPU implementation.

### 7.1 KNN Algorithm and Accelerator Design

*7.1.1 KNN Algorithm.* We use the software code of KNN from the widely used benchmark suite Rodinia [8]. Its pseudo code with some hardware-friendly code transformations is shown in Algorithm 1, which has two major sequential functions.

In function *Compute_Distances* (lines 4-11), for every data point in the search space, its Euclidean distance to the given input query data point is calculated. Each calculation is independent.

In function *Sort_Top_K_Neighbors* (lines 12-28), the top K nearest neighbors to the input query data point are sorted out based on the distances: $top\_K\_distance[K]$ stores the smallest distance, and $top\_K\_distance[1]$ stores the K-th smallest distance. Note that both the top K distances and their associated data point IDs have to be sorted and returned. To enable fine-grained parallelism, inside each loop iteration i (line 16), we split the compare-and-swap loop into two j loops that increment j by a step of two. The first j loop compares-and-swaps elements to their next neighbor (lines 20-23) and the second j loop compares-and-swaps elements to their previous neighbor (lines

**Algorithm 1** Pseudo code for HLS-C/C++ accelerated KNN

```
 1: function LOAD_LOCAL_BUFFER
 2:     local_searchSpace[# buffered points][# features]
 3:     memcpy (local_searchSpace ⟵ a portion of searchSpace from off-chip memory) //pipeline II=1
 4: function COMPUTE_DISTANCES
 5:     inputPoint [# features]
 6:     local_searchSpace[# buffered points][# features]
 7:     distanceResult[# buffered points] //Initialized as zeros
 8:     for i in # of buffered points do //pipeline II=1
 9:         for j in # of features do //fully unrolled
10:             feature_delta = inputPoint[j] - local_searchSpace[i][j]
11:             distanceResult[i] += feature_delta * feature_delta
12: function SORT_TOP_K_NEIGHBORS
13:     distanceResult[(# buffered points) + K] //Extra K dummy distances initialized as MAX_DISTANCE
14:     top_K_distance[K+2] //Initialized as MAX_DISTANCE
15:     top_K_id[K+2] //Initialized as non-valid IDs
16:     for i in range 0 to (# buffered points) + K do //pipeline II=2
17:         top_K_distance[0] = distanceResult[i]
18:         top_K_id[0] = start_id + i
19:         //Parallel compare-and-swap with items ahead
20:         for j in 1 to K-1; j+=2 do //fully unrolled
21:             if top_K_distance[j] < top_K_distance[j+1] then
22:                 swap (top_K_distance[j], top_K_distance[j+1])
23:                 swap (top_K_id[j], top_K_id[j+1])
24:         //Parallel compare-and-swap with items behind
25:         for j in 1 to K; j+=2 do //fully unrolled
26:             if top_K_distance[j-1] < top_K_distance[j] then
27:                 swap (top_K_distance[j], top_K_distance[j-1])
28:                 swap (top_K_id[j], top_K_id[j-1])
```

25-28). As a result, both loops can be fully unrolled and parallelized on FPGA. After the first K iterations of the i loop, $top\_K\_distance[1:K]$ swaps in the first K distances. For any loop iteration $i > K$ (line 16), it compares $top\_K\_distance[0]$ (i.e., incoming $distanceResult[i]$) and $top\_K\_distance[1:K]$ (i.e., current top K distances) so that the biggest distance is swapped to $top\_K\_distance[0]$. That is, $top\_K\_distance[1:K]$ always keeps the K smallest distances. This is proved in our CHIP-KNN paper [23]. The final extra K iterations (line 16) make sure the final $top\_K\_distance[1:K]$ are sorted from biggest to smallest.

#### 7.1.2 *Baseline Accelerator Design.*

To provide a fair comparison, we first apply a series of common HLS optimization techniques [12, 13], which is briefly summarized as below. Please refer to our more generalized CHIP-KNN paper [23] for more details on the design optimizations.

1. *Buffer Tiling.* As shown in the function *Load_Local_Buffer* of Algorithm 1 (lines 1-3), to improve the memory access performance, a portion of the search space points are read from off-chip memory through burst access and buffered on chip before running the *Compute_Distances* and *Sort_Top_K_Neighbors*. Note the top_K results are global. This allows for an efficient DRAM access bandwidth and a faster memory access time during compute.
2. *Customized Pipeline.* First, the *memcpy* in *Load_Local_Buffer* is pipelined with initiation interval (II) of 1 (line 3). Second, the i loop for buffered points in *Compute_Distances* (line 8) is piplelined with II of 1. As a result, its inner j loop (line 9) is fully unrolled. Lastly, the i loop for buffered

points in *Sort_Top_K_Neighbors* (line 16) is piplelined with II of 2. As a result, its inner j loops (lines 20 and 25) are fully unrolled.

3. *Ping-Pong Buffer*. We use double buffers to allow the *Load_Local_Buffer*, *Compute_Distances*, *Sort_Top_K_Neighbors* functions to run in a coarse-grained pipeline. This doubles the on-chip memory usage. We call these three functions together as a single processing element (PE). Each PE uses one AXI read port. Note that with a single PE, the top_K results are global across all tiled buffers it processes.
4. *PE Duplication*. We also duplicate the above PE to enable coarse-grained parallelism. Note this also duplicates the number of PE ports, leading to multi-ports connected to a DRAM. To get the final top_K results, we add a global merger to merge #PE (number of PEs) local copies of sorted top_K results from each PE. To enable efficient data communication between the global merger and PEs, we use the accelerator-to-accelerator streaming connection. This global merger uses one AXI port to write the results to the DRAM and only needs to execute once after all PEs finish the processing of all tiled buffers. Its execution time is negligible. Therefore, it is included in the final measured execution time but not in our analysis.

## 7.2 Design Exploration with Our Insights

*7.2.1 KNN Setup.* In this paper, we use the same setup as the KNN setup used in the Rodinia benchmark suite [8]. Each data point uses a two-dimensional feature vector, and each feature uses 32-bit floating-point type (i.e., 8 bytes per data point). The distance we use is Euclidean distance, but the square root operation (*sqrt*) is skipped in the accelerator to save hardware resource; in fact, we also implemented a version with *sqrt*, which does not impact our performance as it is fully pipelined. The total number of data points in the search space we use is 4,194,304 (i.e., 4M points). We find the top 10 (i.e., $K = 10$) nearest neighbors. The design is evaluated on the Alveo U200 FPGA.

*7.2.2 Performance Model with Our Insights.* To guide our design space exploration, we build a performance model to calculate the latency of each function: the optimal design should balance the latencies between *Load_Local_Buffer*, *Compute_Distances*, and *Sort_Top_K_Neighbors* which execute in a coarse-grained pipeline. Since each function is pipelined, its latency is calculated as:

$$latency = (pipe\_iterations - 1) \times II + pipe\_depth \tag{6}$$

where $pipe\_iterations$ is the number of loop iterations or the number of array elements in the *memcpy* operation, $II$ and $pipe\_depth$ (pipeline depth) can be read from HLS report. However, when reporting the $II$ and $pipe\_depth$ for off-chip memory accesses, Vitis HLS always assumes a theoretical peak port bandwidth for a given port width and kernel clock frequency using Equation 2, which gives inaccurate results. To correct this, we scale the load latency of *Load_Local_Buffer* as the following, by considering our off-chip memory bandwidth insights :

$$load_{effective} = load_{HLS} \times theoretical\_port\_BW / effective\_BW \tag{7}$$

where the effective bandwidth is a function of $kernel\_freq$, $\#ports$, $port\_width$, $max\_burst\_length$, and $buf\_size$ (consecutive data access size in a tiled buffer), as we have characterized in Section 4. Note that for our case studies, we always begin our optimization by minimizing this load latency to achieve a high off-chip memory bandwidth. Then, we balance the latencies of other computation functions, by tuning their loop pipeline II and unrolling factor.

Finally, we also guide the design exploration based on resource usage, especially for how many PEs can be duplicated, based on the post place and routing resource utilization. We use Equation 5 to estimate the on-chip memory usage by the AXI ports.

Table 4. Four KNN design points with different memory configurations, using one copy of each function inside each PE. Performance speedup, frequency, and resource usage are measured using a single SLR and a single DRAM on Alveo U200 FPGA.

| design choices | buffer size | port width | max_burst _length | #PEs #ports | resource utilization in SLR 0 (%) | | | | | freq (MHz) | speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | LUT | FF | BRAM | URAM | DSP | | |
| baseline | 1KB | 32-bit | 16 | 14 | 36 | 27 | 46 | 6 | 9 | 300 | 1x |
| aggressive | 128KB | 512-bit | 32 | 11 | 35 | 30 | 98 | 55 | 7 | 300 | 2.6x |
| **optimal** | 2KB | 64-bit | 256 | 14 | 42 | 29 | 60 | 17.5 | 9 | 300 | 3.5x |
| suboptimal | 2KB | 64-bit | 16 | 14 | 37 | 27 | 47 | 17.5 | 9 | 300 | 1.2x |

*7.2.3 Illustration without Optimizing Function Ratios.* First, we start exploring the KNN design with one copy of each function (i.e., parallel ratio 1:1:1) inside each PE, as presented in Section 7.1. Note this is an under-optimized version of KNN, since *Sort_Top_K_Neighbors* takes roughly two times more latency than *Compute_Distances*, according to Equation 6. However, this mimics a practical design where a balanced computing is not always possible (e.g., due to data dependency) and illustrates the usefulness of our insights. As summarized in Table 4, we compare four memory access configurations of this KNN design, all running at 300MHz. In this study, we constrain our designs with a single SLR (i.e., one FPGA die) and a single DRAM of the Alveo U200 board.

1. *Baseline version.* The baseline version uses a buffer size of 1KB (i.e., consecutive data access size), 32-bit port width, $max\_burst\_length = 16$, and 14 ports (14 PEs), which leads to a significantly underutilized bandwidth according to our characterization. Only 14 PEs can be duplicated because a single DRAM only allows up to 15 AXI ports and the global merger already uses 1 AXI port. It uses the least amount of resource and is the slowest. Its performance is limited by the *Load_Local_Buffer* function that underutilizes the bandwidth.
2. *Aggressive version.* For the aggressive, *suboptimal* and *optimal* versions, we apply the memory coalescing optimization in [12, 13] to widen the AXI port width. The aggressive design uses the best bandwidth configuration even for a single PE: a buffer size of 128KB, 512-bit port width, and $max\_burst\_length = 32$ (i.e., $max\_burst\_size = 16Kb$). This shifts its performance bottleneck to the *Sort_Top_K_Neighbors* function. However, it also uses a lot more on-chip memory due to the large size of partitioned buffers. As a result, it can only duplicate 11 PEs, which limits its overall performance speedup over the baseline to be 2.6x.
3. *Optimal version.* The *optimal* design uses a more balanced configuration: a buffer size of 2KB (to save BRAM and URAM usage), 64-bit port width, and $max\_burst\_length = 256$ (i.e., $max\_burst\_size = 16Kb$). This allows us to duplicate 14 PEs and also shifts the performance bottleneck to the *Sort_Top_K_Neighbors* function. Compared to the baseline and aggressive designs, it achieves 3.5x and 35% speedups.
4. *Suboptimal version.* Finally, to demonstrate the impact of $max\_burst\_length$, we also include the *suboptimal* design, where the only difference to the *optimal* design is that it uses the default $max\_burst\_length$ of 16, and has a 2.9x slowdown compared to the *optimal* design.

*7.2.4 Final Design with Balanced Function Ratios.* Finally, according to our performance model in Section 7.2.2, we choose the optimal bandwidth configurations and adjust the parallel ratio between the three functions to balance their latencies. We have built multiple optimal design points to use all three SLRs and four DRAMs of the Alveo U200 board, but only two of them passed the timing. As presented in Table 5, the first is the *4-PE-512-bit design* at 229MHz: each PE uses 512-bit port width with a parallel ratio of 1:8:24, and each PE connects to one DRAM. The second is the *8-PE-256-bit design* at 262MHz: each PE uses 256-bit port width with a parallel ratio of 1:4:12, and every two PEs connect to one DRAM. In both designs, each PE uses an optimal buffer size

Table 5. Time and resource usage of final KNN design

| design choices | parallel ratio | device resource utilization (%) | | | | | freq (MHz) | speedup |
|---|---|---|---|---|---|---|---|---|
| | | LUT | FF | BRAM | URAM | DSP | | |
| 4-PE-512-bit | 1:8:24 | 46 | 31 | 66 | 7 | 7 | 229 | 5.2x |
| 8-PE-256-bit | 1:4:12 | 46 | 32 | 48 | 7 | 7 | 262 | 5.6x |
| 24-core CPU | dual-socket Intel Xeon Silver 4214 CPU | | | | | | | 1x |

**Algorithm 2** Pseudo code for HLS-C/C++ accelerated SpMV

```
1:  function LOAD_LOCAL_BUFFERS
2:      local_sparseMatrix [# buffered rows][# compressed columns]
3:      local_columnIndex [# buffered rows][# compressed columns]
4:      memcpy (local_sparseMatrix ⟵ a portion of sparseMatrix from off-chip memory) //pipeline II=1
5:      memcpy (local_columnIndex ⟵ a portion of columnIndex from off-chip memory) //pipeline II=1
6:  function COMPUTE_SPARSE_PRODUCT
7:      local_sparseMatrix [# buffered rows][# compressed columns]
8:      local_columnIndex [# buffered rows][# compressed columns]
9:      inputVector [# uncompressed matrix columns]
10:     local_outputResult [# buffered rows] //Initialized as zeros
11:     for i in # of buffered rows do //fully unrolled
12:         for j in # of compressed columns do //pipeline II=8
13:             idx = local_columnIndex[i][j]
14:             this_product = local_sparseMatrix[i][j] * inputVector[idx]
15:             local_outputResult[i] += this_product
```

of 128KB and $max\_burst\_size = 16Kb$. Moreover, in both designs, the pipeline II of the loop in *Sort_Top_K_Neighbors* increases to three due to the increased complexity for HLS scheduling. Inside each PE, since there are 24 or 12 parallel copies of *Sort_Top_K_Neighbors*, we also add a local merger to merge the top_K results. This merger leads to a frequency drop in both designs. Table 5 summarizes their execution time, frequency and resource usage.

We use the *8-PE-256-bit design* as our final KNN design. It balances all three functions' latencies and fully utilizes the effective bandwidths of all DRAMs of the Alveo U200 board, which achieves the theoretical peak performance on the Alveo U200 board. Compared to a 24-thread software implementation running on dual-socket Intel Xeon CPU server, it achieves about 5.6x speedup.

## 8 CASE STUDY 2: SPMV

To further demonstrate the usefulness of our insights in designing an efficient HLS-based hardware accelerator on datacenter FPGAs, we conduct another case study on a sparse matrix-vector multiplication (SpMV) accelerator in HLS-C/C++. The SpMV computational kernel [6, 29] is widely used in graph algorithms and machine learning [5, 27].

### 8.1 SpMV Algorithm and Accelerator Design

*8.1.1 SpMV Algorithm.* We use the software code of SpMV from the widely used accelerator benchmark suite MachSuite [29]. The sparse matrix is stored in the Ellpack compression format [18], which allows for a more regular sequential access pattern. The pseudo code of SpMV is shown in Algorithm 2. In function *Compute_Sparse_Product* (lines 6-15), for each row of the sparse matrix, its dot-product is calculated with the input vector. The column indices of the sparse matrix elements are stored in the same location in the corresponding index matrix *local_columnIndex*. The dot-product calculation for each row is independent.

*8.1.2 Baseline Accelerator Design.* Similar to the accelerator designing process described in Section 7.1.2 for the KNN FPGA kernel, we apply a series of common HLS optimization techniques [12, 13] similar to that in Section 7.1.2.

1. *Buffer Tiling.* As shown in the function *Load_Local_Buffers* of Algorithm 2 (lines 1-5), to improve the memory access performance, a number of rows in the sparse matrix and the column index matrix are read from the off-chip memory through burst access and buffered in the on-chip memory. Then *Compute_Sparse_Product* works on these local buffers. Note the input vector is small and can always be buffered on chip.
2. *Customized Pipeline and Parallelism.* First, the *memcpy* calls in *Load_Local_Buffers* are pipelined with initiation interval (II) of 1 (lines 4-5). Second, the j loop to process buffered points inside one matrix row in *Compute_Sparse_Product* (line 12) is pipelined with II of 8 due to the indirect array access. Lastly, the i loop (line 11) is fully unrolled to parallelize the computation of all rows in the buffered matrix.
3. *Ping-Pong Buffer.* We use Ping-Pong buffers to allow functions *Load_Local_Buffers* and *Compute_Sparse_Product* to run in a coarse-grained pipeline. We call these two functions together as a single processing element (PE). Each PE uses two AXI read ports, leading to multi-ports connected to a single DRAM.
4. *PE Duplication.* We duplicate the above PE to enable coarse-grained parallelism and utilize multiple memory banks.

## 8.2 Design Exploration with Our Insights

*8.2.1 SpMV Setup.* In this paper, we scale up the workload size used in MachSuite [29] for datacenter FPGAs. Our sparse matrix dimension is N by L, where N is 8,192 and L is 1,024. It has a compression ratio of 8 in its rows. Each data element of the sparse matrix is of 32-bit float type. The corresponding column index matrix has the same dimension as the sparse matrix; its data are stored in 32-bit unsigned int type. The total number of elements in each matrix is 8,388,608 (i.e., 32MB in size). We find 8,192 number of dot-products between each row of the matrix and the input vector, which has 8,192 of 32-bit float values.

*8.2.2 Performance Model with Our Insights.* For design space exploration, we formulate a performance model in the same approach used for the KNN accelerator in Section 7.2.2. We use Equation 6 to calculate the latencies of the two functions in the SpMV accelerator. We adjust the latency calculation for memory access using Equation 7 based on the affecting factors: $kernel\_freq$, $\#ports$, $port\_width$, $max\_burst\_length$, and $buf\_size$ (consecutive data access size in a tiled buffer), characterized in Section 4. Finally, to determine how many PEs can be mapped on a datacenter FPGA, we use the post place and routing resource utilization report and Equation 5 to estimate the on-chip memory usage by the AXI ports.

*8.2.3 Design Evaluation.* As summarized in Table 6, we compare four memory access configurations of our SpMV design, as well as their performance speedup, frequency, and resource usage. These designs have balanced latencies between the *Load_Local_Buffers* and *Compute_Sparse_Product* stages while utilizing all three SLRs and four DRAMs of the Alvero U200 board.

1. *Baseline version.* The baseline version uses a buffer size of 32KB, 32-bit port width, $max\_burst\_length = 16$, and 30 PEs with 60 AXI ports. Although it already utilizes the maximum 15 AXI ports for each DRAM, only 480-bit port width is utilized, while the ideal aggregated port width is 547 bits according to Equation 3. It uses the most amount of logic resource and runs the slowest. The significantly low off-chip memory bandwidth limits the design performance,

Table 6. Resource usage and performance speedup of four SpMV designs with different memory configurations

| design choices | buffer size | port width | max_burst _length | #PEs : #ports | device resource utilization (%) | | | | | freq (MHz) | speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | LUT | FF | BRAM | URAM | DSP | | |
| baseline | 32KB | 32-bit | 16 | 30 : 60 | 44 | 33 | 84 | 31 | 5 | 281 | 1x |
| aggressive | 256KB | 512-bit | 32 | 4 : 8 | 32 | 24 | 65 | 53 | 3 | 250 | 7.7x |
| **optimal** | 256KB | 256-bit | 64 | 4 : 8 | 36 | 26 | 53 | 27 | 5 | 291 | 8.5x |
| suboptimal | 256KB | 256-bit | 16 | 4 : 8 | 36 | 26 | 50 | 27 | 5 | 283 | 5.7x |

hindered together by the relatively low port width, under-optimized *max_burst_length*, and the constant access switching in the DRAM row buffer from multiple AXI ports.

2. *Aggressive version.* For the aggressive, suboptimal, and optimal versions, we apply the memory coalescing optimization in [12, 13] to widen the AXI port width. These designs all have four PEs with eight AXI ports, each PE dedicating to one DRAM on the U200 FPGA. The aggressive design uses the best bandwidth configuration for each AXI port: a buffer size of 256KB (to allow more unroll in the i loop in line 11 of Algorithm 2 to balance the two stages), 512-bit port width, and $max\_burst\_length = 32$ (i.e., $max\_burst\_size = 16Kb$). Compared to the *optimal* design, it utilizes 23% and 96% more BRAM and URAM due to the large size of partitioned buffers. This degrades the accelerator frequency to 250MHz, causing the performance of the aggressive design 10% lower than the *optimal* design, while other versions can achieve above 280MHz.
3. *Optimal version.* The *optimal* design uses a more balanced configuration: a buffer size of 256KB, 256-bit port width, and $max\_burst\_length = 64$ (i.e., $max\_burst\_size = 16Kb$). This brings down resource usage for BRAM and URAM, and improves the design frequency to 291MHz. Compared to the baseline and aggressive designs, it achieves 8.5x and 1.1x speedups.
4. *Suboptimal version.* To further demonstrate the impact of *max_burst_length*, we also include the *suboptimal* design, where the only difference to the *optimal* design is that it uses the HLS default $max\_burst\_length = 16$. As a result, the *suboptimal* design is 1.5x slower than the *optimal* design, demonstrating again the importance of our insight of optimal *max_burst_length* selection.

In summary, we use the *optimal* design as our final SpMV design. It balances the latencies of compute and off-chip memory access while fully utilizing the effective off-chip memory bandwidth, which achieves the theoretical peak performance on the U200 board. Compared to a 24-thread software implementation running on an Intel Xeon Silver 4214 CPU server, it is 3.4x faster.

## 9 RELATED WORK

In this section, we summarize previous studies that have evaluated the off-chip DRAM and HBM performance on datacenter and embedded FPGAs and state the novelties in our work. We also discuss other studies that characterize the CPU-FPGA communication, and the memory system of CPUs and GPUs, which are orthogonal to our work.

**Characterization of FPGA DRAM.** In [42], Zhang et al. characterized the off-chip DDR3 bandwidth in an HLS-based FPGA design for a single AXI port, considering both the port width and consecutive data access size. However, they did not consider the accelerator kernel clock frequency, the number of concurrent memory access ports and the maximum burst access length for each AXI port. In [14], Cong et al. developed an analytical model to optimize HLS-based accelerator designs by balancing the on-chip resource usage and the off-chip DRAM bandwidth. In their model, they only considered the impact of port widths on the off-chip DRAM bandwidth and did not consider other factors that we summarized in Section 3.1.

To the best of our knowledge, we are the first to characterize the off-chip memory bandwidth and accelerator-to-accelerator streaming bandwidth of HLS-based designs under a comprehensive set of factors. We are also the first to reveal the unstable bandwidth degradation behavior for multiple concurrent AXI ports access in datacenter FPGAs that use a soft memory system and provide the guideline of setting *max_burst_size* to 16Kb to achieve the optimal off-chip bandwidth. We also derive the optimal AXI port width based on the DRAM/HBM configuration and the accelerator kernel clock frequency to achieve the optimal off-chip bandwidth.

**Characterization of FPGA HBM.** Recently, in [33], Wang et al. evaluated how address mapping policy, bank group, and memory access locality impact the bandwidth and latency of the HBM for FPGAs. However, they did not evaluate the impact of the accelerator kernel clock frequency, port widths, multiple concurrent ports, and *max_burst_length* as we did. Moreover, the memory access pattern evaluated in their design traverses a memory region with a stride of bytes, which is not the most efficient or commonly used memory access pattern on FPGAs. Finally, their microbenchmark is developed in RTL, which still leaves a gap for software programmers who use HLS. More recently, Choi et al. further proposed HBM Connect [9], a fully customized HBM crossbar to better utilize HBM bandwidth when multiple PEs access multiple HBM banks, which is orthogonal to our work.

**Characterization of CPU-FPGA Communication.** In [10, 11], Choi et al. evaluated the communication latency and bandwidth between the host CPU and FPGA for a variety of modern CPU-FPGA platforms, which is orthogonal to our work.

**Characterization of CPU and GPU Memory.** Microbenchmarking memory system performance on CPUs and GPUs has been well studied and characterized. For example, In [16], Fang et al. developed a set of microbenchmarks to measure the memory system microarchitectures of commodity multicore and many-core CPUs. In [34], Wong et. al used microbenchmarks to disclose the characteristics of commodity GPU memory hierarchies. These are orthogonal to our work.

## 10 CONCLUSION

In this paper, we have developed a suite of open source HLS-C/C++ microbenchmarks to quantitatively characterize the effective memory system performance of modern datacenter FPGAs such as Xilinx Alveo U200 (DDR4-based) and U280 (HBM-based) FPGAs, and embedded FPGAs such as the Xilinx ZCU104 (DDR4-based) FPGA, including off-chip memory access performance and accelerator-to-accelerator streaming performance. We have identified a comprehensive set of affecting factors in HLS-based FPGA accelerators, including the clock frequency of the accelerator kernel, the number of concurrent memory access ports, the port width, the maximum burst access length for each port, the size of consecutive data accesses. By analyzing our microbenchmarking results for both datacenter and embedded FPGAs, we also provided insights for software programmers into efficient memory access in HLS-based accelerator designs, and discuss the trade-offs between the soft and hardened memory systems. Moreover, we conducted two case studies on the HLS-based KNN and SpMV accelerator designs and demonstrated that leveraging our insights can provide up to 3.5x and 8.5x speedups over the baseline designs. Our final designs for KNN and SpMV on the U200 FPGA achieved about 5.6x and 3.4x speedups over the 24-core CPU implementation. Finally, our microbenchmark suite and case study benchmarks are open sourced at: https://github.com/SFU-HiAccel/uBench.

## ACKNOWLEDGEMENTS

We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020, COHESA (NETGP485577-15), CWSE PDF (470957), and RGPIN341516; Canada Foundation for Innovation John R. Evans Leaders Fund and British Columbia Knowledge Dev. Fund; Simon Fraser University New Faculty Start-up Grant; Huawei Canada, and Xilinx.

## REFERENCES

[1] Alibaba. 2020. Alibaba compute optimized instance families with FPGAs. https://www.alibabacloud.com/help/doc-detail/108504.htm. Last accessed September 12, 2020.

[2] N. S. Altman. 1992. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician* 46, 3 (1992), 175–185.

[3] Amazon. 2020. Amazon EC2 F1 Instances, Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/. Last accessed September 12, 2020.

[4] AnandTech. 2018. Intel Shows Xeon Scalable Gold 6138P with Integrated FPGA, Shipping to Vendors. https://www.anandtech.com/show/12773/intel-shows-xeon-scalable-gold-6138p-with-integrated-fpga-shipping-to-vendors Last accessed September 12, 2020.

[5] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan. 2014. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, New Orleans, Louisana, 781–792.

[6] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–11.

[7] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. *SIGMETRICS Perform. Eval. Rev.* 44, 1 (June 2016), 323–336.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. http://rodinia.cs.virginia.edu/doku.php?id=start

[9] 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* (Virtual Conference) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 116–126.

[10] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 109, 6 pages.

[11] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2019. In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.* 12, 1, Article 4 (Feb. 2019), 20 pages.

[12] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. 2018. Best-Effort FPGA Programming: A Few Steps Can Go a Long Way. *CoRR* abs/1807.01340 (2018).

[13] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. Association for Computing Machinery, New York, NY, USA, 93–96.

[14] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2017. Bandwidth Optimization Through On-Chip Memory Restructuring for HLS. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 43, 6 pages.

[15] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 365–376.

[16] Zhenman Fang, Sanyam Mehta, Pen-Chung Yew, Antonia Zhai, James Greensky, Gautham Beeraka, and Binyu Zang. 2015. Measuring Microarchitectural Details of Multi- and Many-Core Memory Systems through Microbenchmarking. *ACM Trans. Archit. Code Optim.* 11, 4, Article 55 (Jan. 2015), 26 pages.

[17] Guo Gongde, Wang Hui, Bell David, Bi Yaxin, and Greer Kieran. 2003. KNN Model-Based Approach in Classification. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE 2003)*. Springer, Switzerland, 986–996.

[18] Weerawarana Houstis, S. Weerawarana, E. N. Houstis, and J. R. Rice. 1990. An Interactive Symbolic–Numeric Interface to Parallel ELLPACK for Building General PDE Solvers. In *Symbolic and Numerical Computation for Artificial Intelligence*. 303–322.

[19] Intel. 2020. Intel Stratix 10 FPGAs. https://www.intel.ca/content/www/ca/en/products/programmable/fpga/stratix-10.html Last accessed September 12, 2020.

[20] Intel. 2021. Intel oneAPI: A Unified X-Architecture Programming Model. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html#gs.9wo7rg Last accessed Aug 26, 2021.

[21] Jin Hee Kim, Brett Grady, Ruolong Lian, John Brothers, and Jason H. Anderson. 2017. FPGA-based CNN inference accelerator synthesized from multi-threaded C software. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*. 268–273.

[22] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. 2019. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proc. IEEE* 107, 8 (2019), 1697–1716.

[23] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. 2020. CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs. In *2020 International Conference on Field-Programmable Technology* (Virtual Conference) *(FPT '20)*. 139–147.

[24] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. In *2021 International Symposium on Field-Programmable Gate Arrays* (Virtual Conference) *(FPGA '21)*. 105–115.

[25] Microsoft. 2020. Azure SmartNIC. https://www.microsoft.com/en-us/research/project/azure-smartnic/. Last accessed September 12, 2020.

[26] Nimbix. 2020. Xilinx Alveo Accelerator Cards. https://www.nimbix.net/alveo. Last accessed September 12, 2020.

[27] E. Nurvitadhi, A. Mishra, and D. Marr. 2015. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 109–116.

[28] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 13–24.

[29] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*. Raleigh, North Carolina, 110–119.

[30] Thomas Seidl and Hans-Peter Kriegel. 1998. Optimal Multi-Step k-Nearest Neighbor Search. *SIGMOD Rec.* 27, 2 (June 1998), 154–165.

[31] Masayuki Shimoda, Youki Sada, Ryosuke Kuramochi, and Hiroki Nakahara. 2019. An FPGA Implementation of Real-Time Object Detection with a Thermal Camera. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 413–414.

[32] J Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59, 1 (2015), 7:1–7:7.

[33] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Shuhai: Benchmarking High Bandwidth Memory on FPGAs. In *The 28th IEEE International Symposium On Field-Programmable Custom Computing Machines* (Fayetteville, AR) *(FCCM '20)*. 111–119.

[34] Henry Wong, Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 235–246.

[35] Xilinx. 2017. Vivado Design Suite Vivado AXI Reference. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf. Last accessed September 12, 2020.

[36] Xilinx. 2018. ZCU104 Evaluation Board - User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf Last accessed Aug 17, 2021.

[37] Xilinx. 2020. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf Last accessed September 12, 2020.

[38] Xilinx. 2020. Alveo U280 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf Last accessed September 12, 2020.

[39] Xilinx. 2020. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development Last accessed September 12, 2020.

[40] Xilinx. 2021. Vitis High-Level Synthesis User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf. Last accessed Aug 27, 2021.

[41] Bin Yao, Feifei Li, and Piyush Kumar. 2010. K nearest neighbor queries and kNN-Joins in large relational databases (almost) for free. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 4–15.

[42] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019), 2072–2085.