Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking

Alec Lu, Zhenman Fang, Weihua Liu, Lesley Shannon
{alec_lu,zhenman,weihua_liu,lesley_shannon}@sfu.ca
Simon Fraser University, Canada

ABSTRACT

With the public availability of FPGAs from major cloud service providers like AWS, Alibaba, and Nimbix, 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 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 number of concurrent memory access ports, 2) the data width of each port, 3) the maximum burst access length for each port, and 4) the size of consecutive data accesses. Then we carefully design a set of HLS-based microbenchmarks to quantitatively evaluate the performance of the Xilinx Alveo U200 and U280 FPGA memory systems when changing those affecting factors, and provide insights into efficient memory access in HLS-based accelerator designs. 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. Compared to the baseline designs, optimized designs leveraging our insights achieve about 3.5x and 8.5x speedups for the KNN and SpMV accelerators.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**.

KEYWORDS

Datacenter FPGAs; Memory System; HLS; Benchmarking

ACM Reference Format:
Alec Lu, Zhenman Fang, Weihua Liu, Lesley Shannon. 2021. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. In *2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28–March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431920.3439284

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

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439284

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 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 [21], Alibaba Cloud [1], and Nimbix [22]—have deployed FPGAs in their datacenters. As a result, hardware and software developers can easily access FPGA computing platforms as a cloud service. 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 [36] 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 3(a) and explained in Section 4.3.1. Unfortunately, neither the HLS report [34] nor the hardware emulation (i.e., register-transfer level simulation) [33] accurately models the effective off-chip memory bandwidth under all those affecting factors.

In this paper, our goal is to demystify the effective memory system performance of modern datacenter FPGA boards—which usually have multiple FPGA dies and multiple DRAM or HBM banks in a single board [31, 32]—under a comprehensive set of affecting factors. With a quantitative evaluation of the recent Xilinx Alveo U200 and U280 datacenter FPGA boards [31], 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 number of concurrent memory access ports, 2)

the data width of each port, 3) the maximum burst access length for each port, and 4) 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 FPGAs, while systematically changing those affecting factors.
3. We provide the following insights for software programmers: 1) The total effective off-chip memory bandwidth scales almost linearly with the aggregated port width of all concurrent memory access ports. For a single DDR4 bank whose theoretical peak bandwidth is 19.2GB/s, it flattens at 512-bit: the effective peak read and write bandwidths are about 18.01GB/s and 16.56GB/s. For a single HBM2 bank whose theoretical peak bandwidth is 14.4GB/s, it also flattens at 512-bit: the effective peak read and write bandwidths are about 13.18GB/s and 13.17GB/s. 2) For multiple memory ports to access a single DRAM bank, in order to achieve the best stable peak bandwidth, the maximum burst access size for each port—which is the product of each port's data width and maximum burst access length—should be set to 16Kb (i.e., 2KB). However, the single port access does not have this requirement. For HBM, typically, each memory access port connects to a separate HBM bank. 3) The effective off-chip memory bandwidth increases as the size of consecutive data accesses increases, and flattens when this size is around 128KB. 4) For accelerator-to-accelerator streaming ports, the total (on-chip) communication bandwidth scales linearly with both the data width per port (up to 1024-bit per port) and the number of ports (flattens at 16 ports) independently. 5) The optimal configuration should be chosen by jointly considering the computing-memory balance and resource utilization in the design, instead of aggressively picking the peak bandwidth configuration.
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, 25] algorithms to demonstrate how to leverage our results and insights during design space exploration for HLS-based accelerators. Optimized accelerator designs for KNN and SpMV with our insights can achieve 3.5x and 8.5x speedups over the baseline designs on an Alveo U200 FPGA.

2 DATACENTER FPGAS AND OUR FOCUS

Recently, a variety of FPGA platforms have been deployed in datacenters, such as Microsoft Catapult [24], AWS F1 instance [3], Alibaba F1 and F3 instances [1], IBM OpenCAPI [27], Intel HARP and HARPv2 platforms [4], Xilinx Alveo boards in Nimbix [22, 31, 32], 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 FPGA board consists of multiple FPGA dies and multiple DRAMs and even HBMs (high-bandwidth memory).

In this paper, we mainly focus on measuring the performance of off-chip memory access by accelerators using the memory-mapped AXI ports [30, 33] and on-chip accelerator-to-accelerator streaming using the AXIS streaming ports [30, 33]. 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 [31, 32]. 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. The U280 FPGA features not only three separate FPGA dies and two 16GB off-chip DDR4-2400MT DRAMs, but also two 4GB HBM2 stacks that have a total of 32 256MB HBM banks.

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. *Number of concurrent ports*, i.e., the number of concurrent AXI (memory-mapped) or AXIS (streaming) ports.
2. *Port width*, i.e., the bit-width of each AXI or AXIS port.
3. *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*.
4. *Consecutive data access size*, i.e., the size of consecutive data accesses for each AXI or AXIS port.

In addition to the above four 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 Vivado HLS [34] uses already achieves the best memory performance (results omitted).

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 [34] always assumes that an accelerator can achieve the theoretical peak bandwidth at a given port width and ignores other factors. For register-transfer level (RTL) hardware emulation [33], 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 Vivado HLS-C/C++ [34] and the CPU host code using OpenCL in the Xilinx Vitis tool [33]. To ensure accuracy, we limit our microbenchmarks to only use the most primitive operations, i.e., off-chip memory read or write, and streaming read or write. However, several design challenges arise.

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.
2. The highly repetitive accesses of simple memory operations in the microbenchmark kernel are treated as dead code and can be optimized away by Vivado HLS. We fix this by adding the *volatile* qualifier for those involved variables.

```
1  void krnl_ubench_RD (volatile ap_int<W>* in0, volatile ap_int<W>* in1,
2                       const int data_length) {
3  #pragma HLS INTERFACE m_axi port=in0 bundle=gmem0 max_read_burst_length=16...
4  #pragma HLS INTERFACE m_axi port=in1 bundle=gmem1 max_read_burst_length=16...
5  ...
6      volatile ap_int<W> temp_data_0, temp_data_1;
7
8  #pragma HLS DATAFLOW
9      //Repeat NUM_ITERATIONS times: read data_length number of consecutive data
10     RD_in_0: for (int i = 0; i < NUM_ITERATIONS: ++i)
11         for (int j = 0; j < data_length: ++j)
12             #pragama HLS PIPELINE II=1
13             temp_data_0 = in0[j];
14     RD_in_1: for (int i = 0; i < NUM_ITERATIONS: ++i)
15         for (int j = 0; j < data_length: ++j)
16             #pragama HLS PIPELINE II=1
17             temp_data_1 = in1[j];
18 }
```

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

3. By default, Vivado 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 [30]. The other is the on-chip accelerator-to-accelerator streaming bandwidth using the AXIS streaming ports [30]; this is a relatively new feature added in Xilinx Vitis 2019.2 [33].

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 aforementioned factors.

1. Line 1: the input arrays *in0* and *in1* use the *ap_int<W>* type in HLS [34], 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 Vivado 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 $data\_length * W/8$ bytes.
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 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.

```
1  #pragma HLS INTERFACE m_axi port=in0 bundle=gmem0...
2  #pragma HLS INTERFACE m_axi port=in1 bundle=gmem0...
3  ...
4      // Option 1 - array-wise access switching
5      for (int j = 0; j < data_length: ++j)
6          #pragama HLS PIPELINE II=1
7          temp_data_0 = in0[j];
8      for (int j = 0; j < data_length: ++j)
9          #pragama HLS PIPELINE II=1
10         temp_data_1 = in1[j];
11
12     // Option 2 - element-wise access switching
13     for (int j = 0; j < data_length: ++j)
14         #pragama HLS PIPELINE II=1
15         temp_data_0 = in0[j];
16         temp_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

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 [34], 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 $num\_outstanding\_access = 1$.

4 RESULTS AND INSIGHTS

4.1 Experimental Setup

As discussed in Section 2, the platforms we evaluate are the Xilinx Alveo U200 and U280 datacenter FPGA boards [31, 32]. We build our HLS C/C++ based microbenchmarks using Xilinx Vitis 2019.2 [33]. The FPGA kernels of all these microbenchmarks run at 300MHz. 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.

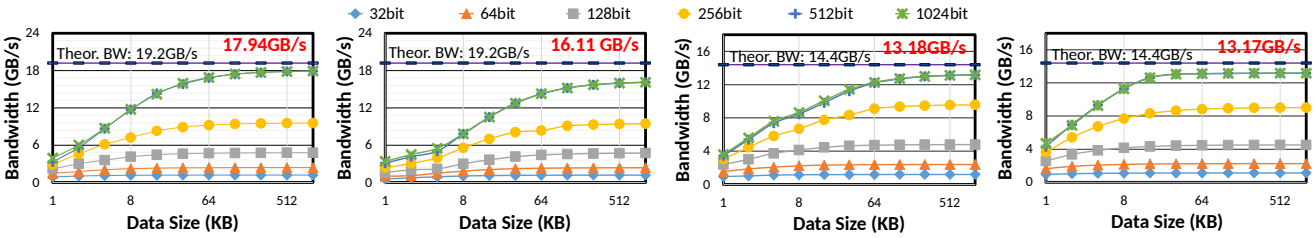


Figure 1: Bandwidth for accessing a single DDR4 and HBM bank with a single AXI port. Note the x-axis is plotted in log_2 scale.

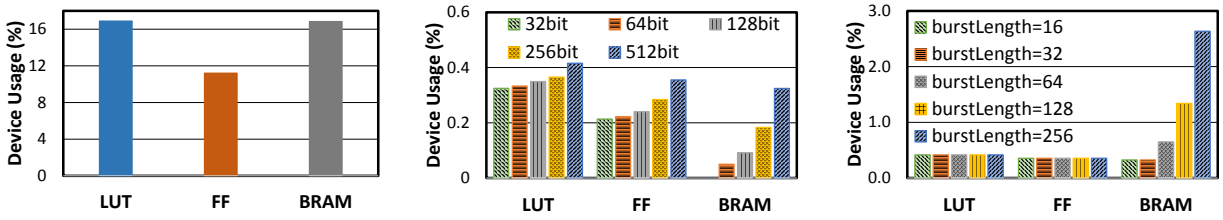


Figure 2: Resource usage of a single AXI port design with different port widths and *max_burst_lengths* (port width = 512 bits).

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

4.2.1 Bandwidth Results. For a single AXI port (and single argument) 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 1. 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. For DRAM, this is because the physical bus width of a DDR4 bank used in this paper is 512 bits. For HBM, the physical bus width of an HBM2 bank used in this paper is 256 bits, and its memory controller runs at 450MHz. Although the linear bandwidth scaling of an HBM bank stops at 256-bit, the 512-bit width still achieves the highest effective bandwidth because our microbenchmark runs at 300MHz.
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 a single AXI port. This is slightly lower than the reported 13.27GB/s HBM bandwidth in [28] 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 nor HBM2. We omit the results due to space constraints.

4.2.2 Resource Usage. Figure 2 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. Throughout this paper, we report the post place and routing resource utilization.

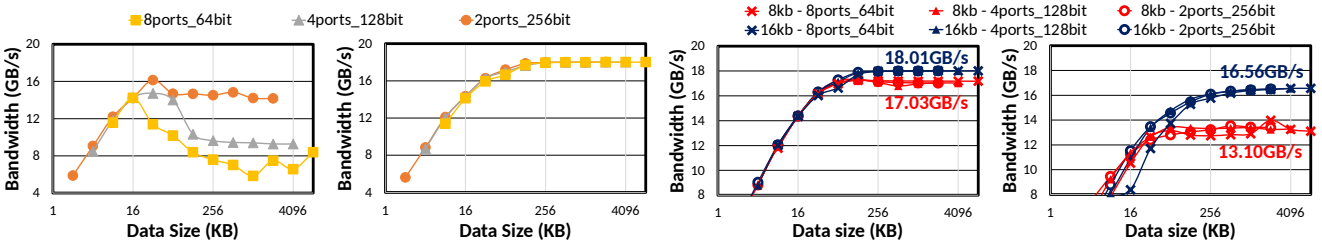
1. Shown in Figure 2(a), the platform shell resource usage remains unchanged as there is no additional AXI port connection.
2. Shown in Figure 2(b), as the AXI port width increases, the LUT and FF resources have a steady small percentage increase while the BRAM usage grows linearly. The total resource increase is less than 0.3% of the entire device across all three resources.
3. Shown in Figure 2(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 [30]. This buffer size can be calculated using the following equation [34]:

$$AXI\_buf\_size = port\_width * max\_burst\_length * num\_outstanding\_access \quad (1)$$

This equation explains the linear increase of BRAM resource in Figure 2(b) and (c). For the 32-bit case in Figure 2(b), it uses FIFO instead of BRAM due to its small buffer size. For the *max_burst_length* = 16 case in Figure 2(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.

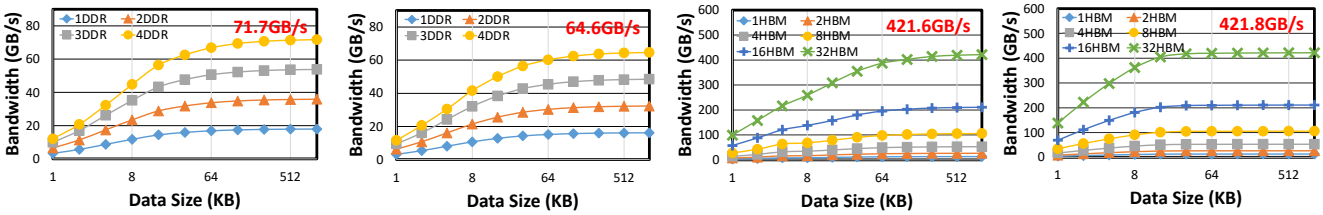
4.3 Bandwidth and Resource Utilization for Multiple AXI Ports Off-Chip Access

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



(a) READ using default *max_burst_length* = 16 (b) READ using maximum *max_burst_length* = 256 (c) READ using max burst sizes: 8kb and 16kb (d) WRITE using max burst sizes: 8kb and 16kb

Figure 3: Bandwidth for accessing a single DDR4 DRAM with multiple AXI ports and a total port width of 512-bit.



(a) DRAM READ (b) DRAM WRITE (c) HBM READ (d) HBM WRITE

Figure 4: Bandwidth for accessing multiple DDR4 and HBM banks using AXI ports. Note the x-axis is plotted in log_2 scale.

port width). Figure 3 presents the results for 2 ports each 256-bit wide, 4 ports each 128-bit wide, and 8-ports each 64-bit wide.

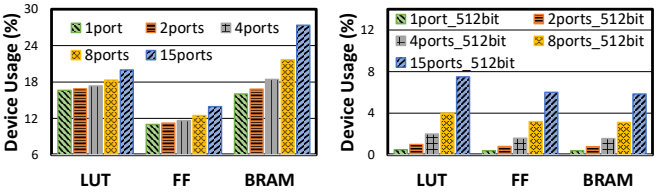
1. Figure 3(a) presents the bandwidth with *max_burst_length* = 16 (HLS default). As the consecutive data access size increases, the bandwidths for 2 ports, 4 ports, and 8 ports get an unstable degradation. This is mainly because the DRAM row buffer for one AXI port access will be replaced by another 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 3(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 1, a design with *max_burst_length* = 256 can significantly overuse 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* (= *max_burst_length* * *port width*).
   Figure 3(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 3(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. Due to space constraints, we omit the detailed results. Here is a summary of our findings.

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.

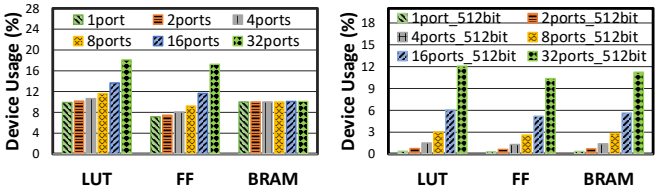
4.3.2 Bandwidths for Multiple Off-Chip Memory Banks. Figure 4 a) and b) present the effective off-chip bandwidths on the Alveo U200 FPGA with multiple DRAMs, each of which has a single AXI port with 512-bit width. The effective DRAM bandwidth scales linearly with the number of memory banks. The bandwidths for the four DDR4s on U200 peak at 71.7GB/s and 64.6GB/s for read and write.

Figure 4 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 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).



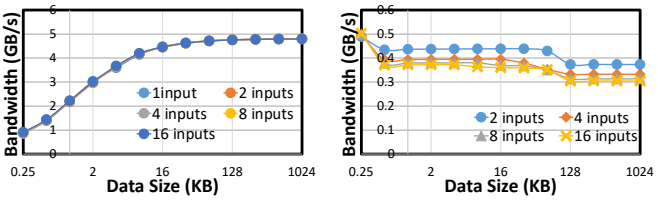
(a) FPGA platform shell (b) Kernel with different # of ports

Figure 5: U200 resource usage for multiple DRAM AXI ports.



(a) FPGA platform shell (b) Kernel with different # of ports

Figure 6: U280 resource usage for multiple HBM AXI ports.



(a) Input-wise Access Switching (b) Element-wise Access Switching

Figure 7: Read bandwidth for accessing multiple input data arrays via a single shared 128-bit AXI port.

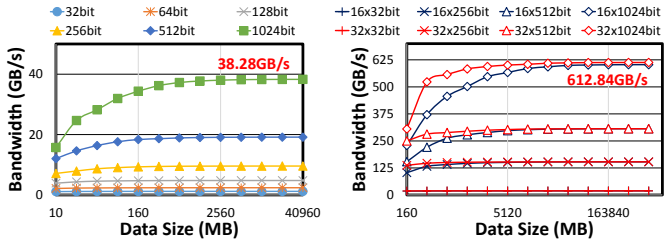
4.3.3 Resource Usage for Multiple AXI Ports in Single DRAM. As shown in Figure 5, when the number of AXI ports connected to the DRAM increases (the maximum is 15 for a single DRAM), the resource usage increases in both the U200 FPGA platform shell and the kernel design. Between a single-port design and 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.

4.3.4 Resource Usage for Multiple AXI Ports Accessing Multiple HBM Banks. Figure 6 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 5(b), which increases linearly with the number of ports.

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

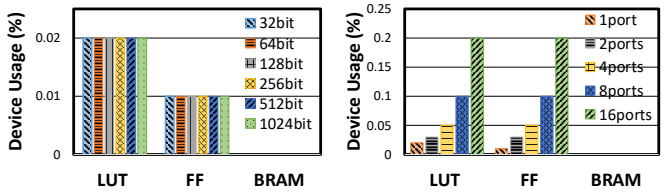
Figure 7 presents the read bandwidth when accessing multiple input data arrays on the DDR4 DRAM through a shared 128-bit AXI port.

1. Shown in Figure 7(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 7(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.



(a) Kernel with different port width (b) Kernel with different # of ports

Figure 8: Bandwidth for accessing multiple AXIS ports.



(a) Single port kernel with different port widths (b) Multiple port kernel

Figure 9: Resource usage of a multiple AXIS ports design.

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

4.5.1 Bandwidth Results. Figure 8 presents the accelerator-to-accelerator streaming bandwidths.

1. Shown in Figure 8(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 8(a), the effective inter-kernel streaming bandwidth increases as the streaming data size increases, and flattens at around 2GB size (38.28GB/s for a single AXIS port).
3. Shown in Figure 8(b), the effective inter-kernel streaming bandwidth scales linearly as the number of AXIS ports increases. It flattens until 16 AXIS ports are used, no matter which port width is used. The effective peak streaming bandwidth is about 612.84GB/s for 16 of 1024-bit wide AXIS ports.

4.5.2 Resource Usage. Shown in Figure 9(a), a single AXIS port has a fixed tiny resource usage regardless of its port width: it uses 0.02% of LUT and 0.01% of FF. The resource usage scales linearly as the number of AXIS ports increases, as shown in Figure 9(b). Therefore, accelerator-to-accelerator streaming is very efficient.

Table 1: Off-chip memory access latency at 300MHz

DDR4 latency	110ns	HBM2 latency	108ns

4.6 Latency for Off-Chip Access

Table 1 summarizes the latencies for randomly accessing 4 bytes of data from the off-chip DDR4 and HBM2 banks. 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 [28], 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 [28] because of the efficient hardened HBM memory interconnect. However, our measured DRAM latency is higher than that measured in [28] (73.3ns) due to the overhead from the HLS-generated memory interconnect IP.

5 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, 26, 35].

5.1 KNN Algorithm and Accelerator Design

5.1.1 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.

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 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 [20]. The final extra K iterations (line 16) make sure the final $top\_K\_distance[1:K]$ are sorted from biggest to smallest.

5.1.2 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 [20] for more details.

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])
```

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*.
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. 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 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

Table 2: 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

all tiled buffers. Its execution time is negligible. Therefore, it is included in the final execution time but not in our analysis.

5.2 Design Exploration with Our Insights

5.2.1 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. 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.

5.2.2 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 can be calculated as:

$$latency = (pipe\_iterations - 1) * II + pipe\_depth \quad (2)$$

where $pipe\_iterations$ is the number of loop iterations or the number of array elements in the *memcpy* operation, II and $pipe\_depth$ can be read from HLS report. However, when reporting the II and $pipe\_depth$ for off-chip memory accesses, Vivado HLS always assumes a theoretical peak bandwidth for a given port width (i.e., $19.2GB/s * port\_width/512$), which gives inaccurate results. To correct this, we scale the load latency of *Load_Local_Buffer* as:

$$load_{effective} = load_{HLS} * theoretical\_BW/effective\_BW \quad (3)$$

where the effective bandwidth is a function of $\#ports$, $port\_width$, max_burst_length, and buf_size (consecutive data access size in a tiled buffer), as we have characterized in Section 4.

Finally, we also guide the design exploration based on resource usage, especially for how many PEs can be duplicated. We use Equation 1 to estimate the on-chip memory usage by the AXI ports.

5.2.3 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 5.1. 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 2. 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 2, we compare four memory access configurations of this KNN design. In this study, we constrain our designs with a single SLR (Super Long Region, 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 $max\_burst\_length = 16$ (HLS default). As a result, it gets a 2.9x slowdown compared to the *optimal* design.

5.2.4 Final Design with Balanced Function Ratios. Finally, according to our performance model in Section 5.2.2, 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 3, 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 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

Table 3: 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

a local merger to merge the top_K results. This merger leads to a frequency drop in both designs. Table 3 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.

6 CASE STUDY 2: SPMV

To further demonstrate our insights' usefulness, we conduct another case study on a sparse matrix-vector multiplication (SpMV) accelerator in HLS-C/C++. The SpMV computational kernel [6, 25] is widely used in graph algorithms and machine learning [5, 23].

6.1 SpMV Algorithm and Accelerator Design

6.1.1 SpMV Algorithm. We use the software code of SpMV from the widely used accelerator benchmark suite MachSuite [25]. 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.

6.1.2 Baseline Accelerator Design. We apply a series of common HLS optimization techniques [12, 13] similar to that in Section 5.1.2.

1. *Buffer Tiling.* As shown in the function *Load_Local_Buffers* of Algorithm 2 (lines 1-5), 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 also duplicate the above PE to enable coarse-grained parallelism and utilize multiple memory banks.

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
```

6.2 Design Exploration with Our Insights

6.2.1 SpMV Setup. In this paper, we scale up the workload size used in MachSuite [25] 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.

6.2.2 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 5.2.2. This is because loop iterations in our SpMV functions *Load_Local_Buffers* and *Compute_Sparse_Product* are pipelined and executed in coarse-grained pipeline. We use Equation 2 to calculate the latencies of the two functions in the SpMV accelerator. We adjust the latency calculation for memory access based on the affecting factors: $\#ports$, $port\_width$, 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 1 to estimate the on-chip memory usage by the AXI ports.

6.2.3 Design Evaluation. As summarized in Table 4, we compare four memory access configurations of our SpMV design. 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. It uses the most amount of logic resource and runs the slowest. The significantly low off-chip memory bandwidth limits the design performance, hindered together by the relatively low port width (480-bit out of 512-bit bus width used per DRAM), 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

Table 4: 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

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.

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.

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. It is about 3.4x faster than the 24-thread CPU implementation.

7 RELATED WORK

Characterization of FPGA DRAM. In [36], 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 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 and provide the guideline of setting max_burst_size to 16Kb to achieve the optimal off-chip bandwidth.

Characterization of FPGA HBM. Recently, in [28], 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 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. 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 [29], Wong et. al used microbenchmarks to disclose the characteristics of commodity GPU memory hierarchies. These are orthogonal to our work.

8 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, 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 number of concurrent memory access ports, the port width, the maximum burst access length for each port, and the size of consecutive data accesses. By analyzing our microbenchmarking results, we also provided insights for software programmers into efficient memory access in HLS-based accelerator designs. 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 that 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 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 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.

[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] 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)*.

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] Xilinx. 2020. Vivado Design Suite User Guide, High-Level Synthesis. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf. Last accessed September 12, 2020.

[35] 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.

[36] 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.