Shuhai: Benchmarking High Bandwidth Memory on FPGAs

Zeke Wang   Hongjing Huang   Jie Zhang   Gustavo Alonso
Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China   *Systems Group, ETH Zurich, Switzerland*

***Abstract*—FPGAs are starting to be enhanced with High Bandwidth Memory (HBM) as a way to reduce the memory bandwidth bottleneck encountered in some applications and to give the FPGA more capacity to deal with application state. However, the performance characteristics of HBM are still not well specified, especially in the context of FPGAs. In this paper, we bridge the gap between nominal specifications and actual performance by benchmarking HBM on a state-of-the-art FPGA, i.e., a Xilinx Alveo U280 featuring a two-stack HBM subsystem. To this end, we propose Shuhai, a benchmarking tool that allows us to demystify all the underlying details of HBM on an FPGA. FPGA-based benchmarking should also provide a more accurate picture of HBM than doing so on CPUs/GPUs, since CPUs/GPUs are noisier systems due to their complex control logic and cache hierarchy. Since the memory itself is complex, leveraging custom hardware logic to benchmark inside an FPGA provides more details as well as accurate and deterministic measurements. We observe that 1) HBM is able to provide up to 425 GB/s memory bandwidth, and 2) how HBM is used has a significant impact on performance, which in turn demonstrates the importance of unveiling the performance characteristics of HBM so as to select the best approach. As a yardstick, we also apply Shuhai to DDR4 to show the differences between HBM and DDR4. Shuhai can be easily generalized to other FPGA boards or other generations of memory, e.g., HBM3, and DDR3. We will make Shuhai open-source, benefiting the community.**

I. Introduction

The computational capacity of modern computing system continues increasing due to the constant improvements on CMOS technology, typically by instantiating more cores within the same area and/or by adding extra functionality to the cores (AVX, SGX, etc.). In contrast, the bandwidth capability of DRAM memory has only slowly improved over many generations. As a result, the gap between memory and processor speed keeps growing and is being exacerbated by multicore designs due to the concurrent access. To bridge the memory bandwidth gap, semiconductor memory companies such as Samsung[1] have released a few memory variants, e.g., Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM), as a way to provide significantly higher memory bandwidth. For example, the state-of-the-art Nvidia GPU V100 features 32 GB HBM2 (the second generation HBM) to provide up to 900 GB/s memory bandwidth for its thousands of computing cores.[2]

Compared with a GPU of the same generation, FPGAs used to have an order of magnitude lower memory bandwidth since FPGAs typically feature up to 2 DRAM memory channels, each of which has up to 19.2 GB/s memory bandwidth on our tested FPGA board Alevo U280 [47].[3] As a result, an FPGA-based solution using DRAM could not compete with a GPU for bandwidth-critical applications. Consequently, FPGA vendors like Xilinx [47] have started to introduce HBM[4] in their FPGA boards as a way to remain competitive on those same applications. HBM has the potential to be a game-changing feature by allowing FPGAs to provide significantly higher performance for memory- and compute-bound applications like database engines [37] or deep learning inference [19]. It can also support applications in keeping more state within the FPGA without the significant performance penalties seen today as soon as DRAM is involved.

Despite the potential of HBM to bridge the bandwidth gap, there are still obstacles to leveraging HBM on the FPGA. First, the performance characteristics of HBM are often unknown to developers, especially to FPGA programmers. Even though an HBM stack consists of a few traditional DRAM dies and a logic die, the performance characteristics of HBM are significantly different than those of, e.g., DDR4. Second, Xilinx's HBM subsystem [48] introduces new features like a *switch* inside its HBM memory controller. The performance characteristics of the switch are also unclear to the FPGA programmer due to the limited details exposed by Xilinx. These two issues can hamper the ability of FPGA developers to fully exploit the advantages of HBM on FPGAs.

To this end, we present Shuhai,[5] a benchmarking tool that allows us to demystify all the underlying details of HBM. Shuhai adopts a software/hardware co-design approach to provide *high-level insights* and *ease of use* to developers or researchers interested in leveraging HBM. The high-level insights come from the first end-to-end analysis of the performance characteristic of typical *memory access patterns*. The ease of use arises from the fact that Shuhai performs the majority of the benchmarking task without having to reconfigure the FPGA between parts of the benchmark. To our knowledge, Shuhai is the first platform to systematically benchmark HBM on an FPGA. We demonstrate the usefulness of Shuhai by identifying four important aspects on the usage of HBM-enhanced FPGAs:

F1: HBMs Provide Massive Memory Bandwidth. On the tested FPGA board Alveo U280, HBM provides up to 425 GB/s memory bandwidth, an order of magnitude more than using two traditional DDR4 channels on the same board. This is still half of what state-of-the-art GPUs obtain but it represents a significant leap forward for FPGAs.

F2: The Address Mapping Policy is Critical to High Bandwidth. Different address mapping policies lead to an order of magnitude throughput differences when running a typical memory access pattern (i.e., sequential traversal) on HBM, indicating the importance of matching the address mapping policy to a particular application.

F3: Latency of HBM is Much Higher than DDR4. The connection between HBM chips and the associated FPGA is done via serial I/O connection, introducing extra processing for parallel-to-serial-to-parallel conversion. For example, Shuhai identifies that the latency of HBM is 106.7 ns while the latency of DDR4 is 73.3 ns, when the memory transaction hits an open page (or row), indicating that we need more on-the-fly memory transactions, which are allowed on modern FPGAs/GPUs, to saturate HBM.

[1]https://www.samsung.com/semiconductor/dram/hbm2/

[2]https://www.nvidia.com/en-us/data-center/v100/

[3]https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

[4]In the following, we use HBM which refers to HBM2 in the context of Xilinx FPGAs, as Xilinx FPGAs feature two HBM2 stacks.

[5]Shuhai is a pioneer of Chinese measurement standards, with which he measured the territory of China in the Xia dynasty.

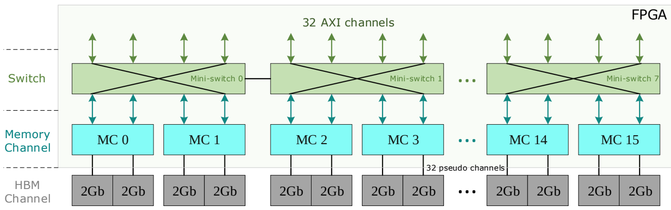


Fig. 1: Architecture of Xilinx HBM subsystem

F4: FPGA Enables Accurate Benchmarking Numbers. We have implemented `Shuhai` on an FPGA with the benchmarking engine directly attaching to HBM modules, making it easier to reason about the performance numbers from HBM. In contrast, benchmarking memory performance on CPUs/GPUs makes it difficult to distinguish effects as, e.g., the cache introduces significant interference in the measurements. Therefore, we argue that our FPGA-based benchmarking approach is a better option when benchmarking memory, whether HBM or DDR.

II. Background

An HBM chip employs the latest development of IC packaging technologies, such as Through Silicon Via (TSV), stacked-DRAM, and 2.5D package [7], [15], [20], [27]. The basic structure of HBM consists of a base logic die at the bottom and 4 or 8 core DRAM dies stacked on top. All the dies are interconnected by TSVs.

Xilinx integrates two *HBM stacks* and an HBM controller inside the FPGA. Each HBM stack is divided into eight independent *memory channels*, where each memory channel is further divided into two 64-bit *pseudo channels*. A pseudo channel is only allowed to access its associated *HBM channel* that has its own address region of memory, as shown in Figure 1. The Xilinx HBM subsystem has 16 memory channels, 32 pseudo channels, and 32 HBM channels.

On the top of 16 memory channels, there are 32 *AXI channels* that interact with the user logic. Each AXI channel adheres to the standard AXI3 protocol [48] to provide a proven standardized interface to the FPGA programmer. Each AXI channel is associated with a HBM channel (or pseudo channel), so each AXI channel is only allowed to access its own memory region. To make each AXI channel able to access the full HBM space, Xilinx introduces a switch between 32 AXI channels and 32 pseudo channels [45], [48].[6] However, the switch is not fully implemented due to its huge resource consumption. Instead, Xilinx presents eight *mini-switches*, where each mini-switch serves four AXI channels and their associated pseudo channels and the mini-switch is fully implemented in a sense that each AXI channel accesses any pseudo channel in the same mini-switch with the same latency and throughput. Besides, there are two bidirectional connections between two adjacent mini-switches for global addressing.

III. General Benchmarking Framework `Shuhai`

In this section, we present the design methodology followed by the software and hardware components of `Shuhai`.

A. Design Methodology

In this subsection, we summarize two concrete challenges **C1** and **C2**, and then present `Shuhai` to tackle the two challenges.

[6]By default, we disable the switch in the HBM memory controller when we measure latency numbers of HBM, since the switch that enables global addressing among HBM channels is not necessary. The switch is on when we measure throughput numbers.

C1: High-level Insight. It is critical to make our benchmarking framework meaningful to FPGA programmers in a sense that we should provide high-level insights to FPGA programmers for ease of understanding. In particular, we should give the programmer an end-to-end explanation, rather than just incomprehensible memory timing parameters like row precharge time T_{RP}, so that the insights can be used to improve the use of HBM memory on FPGAs.

C2: Easy to Use. It is difficult to achieve ease of use when benchmarking on FPGAs when a small modification might need to reconfigure the FPGA. Therefore, we intend to minimize the reconfiguration effort so that the FPGA does not need to be reconfigured between benchmarking tasks. In other words, our benchmarking framework should allow us to use a single FPGA image for a large number of benchmarking tasks, not just for one benchmarking task.

Our Approach. We propose `Shuhai` to tackle the above two challenges. In order to tackle the first challenge, **C1**, `Shuhai` allows to directly analyze the performance characteristics of typical memory access patterns used by FPGA programmers, providing an end-to-end explanation for the overall performance. To tackle the second challenge, **C2**, `Shuhai` uses runtime parameterization of the benchmarking circuit so as to cover a wide range of benchmarking tasks without reconfiguring the FPGA. Through the access patterns implemented in the benchmark, we are able to unveil the underlying characteristics of HBM and DDR4 on FPGAs.

`Shuhai` adopts a software-hardware co-design approach based on two components: a software component (Subsection III-B) and a hardware component (Subsection III-C). The main role of the software component is to provide flexibility to the FPGA programmer in terms of runtime parameters. With these runtime parameters, we do not need to frequently reconfigure the FPGA when benchmarking HBM and DDR4. The main role of the hardware component is to guarantee performance. More precisely, `Shuhai` should be able to expose the performance potential, in terms of maximum achievable memory bandwidth and minimum achievable latency, of HBM memory on the FPGA. To do so, the benchmarking circuit itself cannot be the bottleneck at any time.

B. Software Component

`Shuhai`'s software component aims to provide a user-friendly interface such that an FPGA developer can easily use `Shuhai` to benchmark HBM memory and obtain relevant performance characteristics. To this end, we introduce a memory access pattern widely used in FPGA programming: *Repetitive Sequential Traversal (RST)*, as shown in Figure 2.

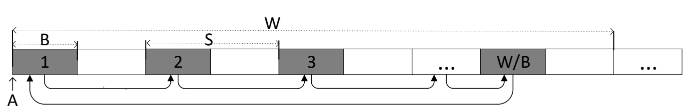


Fig. 2: Memory access pattern used in `Shuhai`.

The RST pattern traverses a *memory region*, a data array storing data elements in a sequence. The RST repetitively sweeps over the memory region of size W with the starting address A, and each time reads B bytes with a stride of S bytes, where B and S are a power of 2. On our tested FPGA, the burst size B should be not smaller than 32 (or 64) for HBM (or DDR4) due to the constraint of HBM/DDR4 memory application data width. The stride S should be not larger than the working set size W. The parameters are summarized in Table I. We calculate the address

Parameter	Definition
N	Number of memory read/write transactions
B	Burst size (in bytes) of a memory read/write transaction
W	Working set size (in bytes). W (>16) is a power of 2.
S	Stride (in bytes)
A	Initial address (in bytes)

TABLE I: Summary of runtime parameters

$T[i]$ of the i-th memory read/write transaction issued by the RST, as illustrated in Equation 1. The calculation can be implemented with simple arithmetic, which in turn leads to fewer FPGA resources and potentially higher frequency. Even though the supported memory access pattern is quite simple, it can still unveil the performance characteristics of the memory, e.g., HBM and DDR4, on FPGAs.

$$T[i] = A + (i \times S)\%W \tag{1}$$

C. Hardware Component

The hardware component of Shuhai consists of a *PCIe module*, M *latency modules*, a *parameter module* and M *engine modules*, as illustrated in Figure 3. In the following, we discuss the implementation details for each module.

1) Engine Module: We directly attach an instantiated engine module to an AXI channel such that the engine module directly serves the AXI interface, e.g., AXI3 and AXI4 [2], [46], provided by the underlying memory IP core, e.g., HBM and DDR4. The AXI interface consists of five different channels: read address (RA), read data (RD), write address (WA), write data (WD) and write response (WR) [46]. Besides, the input clock of the engine module is exactly the clock from the associated AXI channel. For example, the engine module is clocked with 450 MHz when benchmarking HBM as it allows at most 450 MHz for its AXI channels. There are two benefits to use the same clock. First, no extra noise, such as longer latency, is introduced by FIFOs needed to cross different clock regions. Second, the engine module is able to saturate its associated AXI channel, not leading to underestimates of the memory bandwidth capacity.

The engine module, written in Verilog, consists of two independent modules: a *write module* and a *read module*. The write module serves three write-related channels WA, WD, and WR, while the read module serves two read-related channels RA and RD.

Write Module. The write module contains a state machine to serve a memory-writing task at a time from the CPU. The task has the initial address A, number of write transactions N, burst size B, stride S, and working set size W. Once the writing task is received, this module always tries to saturate the memory write channels WR and WD by asserting the associated valid signals before the writing task completes, aiming to maximize the achievable throughput. The address of each memory write transaction is specified in Equation 1. This module also probes the WR channel to validate that the on-the-fly memory write transactions are successfully finished.

Read Module. The read module contains a state machine to serve a memory-reading task at a time from the CPU. The task has the initial address A, number of read transactions N, burst size B, stride S, and working set size W. Unlike the write module, that only measures the achievable throughput, the read module measures as well the latency of each of serial memory read transactions: we immediately issue the second memory read transaction only after the read data of the first read transaction is returned.[7] When measuring throughput, this module always tries to saturate the memory read channels RA and RD by always asserting the RA valid signal before the reading task completes.

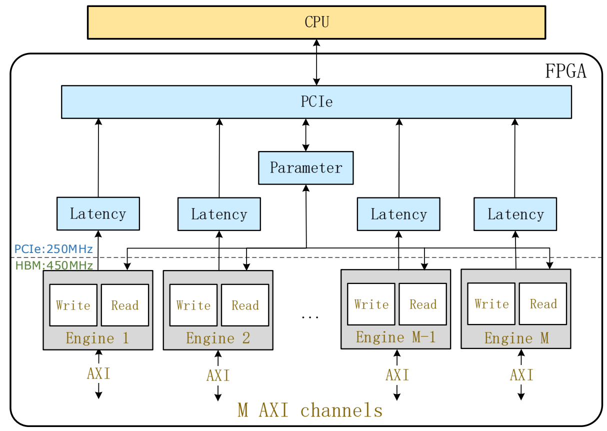


Fig. 3: Overall hardware architecture of our benchmarking framework. It can support M hardware engines running simultaneously, with each engine for one AXI channel. In our experiment, M is 32 for HBM, while M is 2 for DDR4.

2) PCIe Module: We directly deploy the Xilinx DMA/Bridge Subsystem for PCI Express (PCIe) IP core in our *PCIe module*, which is clocked at 250 MHz. Our PCIe kernel driver exposes a PCIe bar mapping the runtime parameters on the FPGA to the user such that the user is able to directly interact with the FPGA using software code. These runtime parameters determine the control and status registers stored in the parameter module.

3) Parameter Module: The parameter module maintains the runtime parameters and communicates with the host CPU via the PCIe module, receiving the runtime parameters, e.g., S, from the CPU and returning the throughput numbers to the CPU.

Upon receiving runtime parameters, we use them to configure M engine modules, each of which needs two 256-bit control registers to store its runtime parameters: one register for the read module and the other register for the write module in each engine module. Inside a 256-bit register, W takes 32 bits, S takes 32 bits, N takes 64 bits, B takes 32 bits, and A takes 64 bits. The remaining 32 bits are reserved for future use. After setting all the engines, the user can trigger the start signal to begin the throughput/latency testing.

The parameter module is also responsible for returning the throughput numbers (64-bit status registers) to the CPU. One status register is dedicated to each engine module.

4) Latency Module: We instantiate a *latency module* for each engine module dedicated to an AXI channel. The latency module stores a *latency list* of size 1024, where the latency list is written by the associated engine module and read by the CPU. Its size is a *synthesis parameter*. Each latency number containing an 8-bit register refers to the latency for a memory read operation, from the issue of the read operation to the data having arrived from the memory controller.

[7]We are able to unveil many performance characteristics of HBM and DDR4 by analyzing the latency difference among serial memory read transactions. The fundamental reason of the immediate issue is that a refresh command that occurs periodically will close all the banks in our HBM/DDR4 memory, and then there will be no latency difference if the time interval of two serial read transactions is larger than the time (e.g., 7.8 μs) between two refresh commands.

IV. Experiment Setup

In this section, we present the tested hardware platform (Subsection IV-A) and the address mapping policies explored (Subsection IV-B), followed by the hardware resource consumption (Subsection IV-C) and our benchmarking methodology (Subsection IV-D).

A. Hardware Platform

We run our experiments on a Xilinx's Alevo U280 [47] featuring two HBM stacks of a total size of 8GB and two DDR4 memory channels of a total size of 32 GB. The theoretical HBM memory bandwidth can reach 450 GB/s (450 MHz * 32 * 32 B/s), while the theoretical DDR4 memory bandwidth can reach 38.4 GB/s (300 MHz * 2 * 64 B/s).

B. Address Mapping Policies

The application address can be mapped to memory address using multiple policies, where different address bits map to bank, row, or column addresses. Choosing the right mapping policy is critical to maximize the overall memory throughput. The policies enabled for HBM and DDR4 are summarized in Table II, where "xR" means that x bits are for row address, "xBG" means that x bits are for bank group address, "xB" means that x bits are for bank address, and "xC" means that x bits are for column address. The default policies of HBM and DDR4 are "RGBCG" and "RCB", respectively. "-" stands for address concatenation. We always use the default memory address mapping policy for both HBM and DDR4 if not particularly specified. For example, the default policy for HBM is RGBCG.

Policies	HBM (app_addr[27:5])	DDR4 (app_addr[33:6])
RBC	14R-2BG-2B-5C	17R-2BG-2B-7C
RCB	14R-5C-2BG-2B	17R-7C-2B-2BG
BRC	2BG-2B-14R-5C	2BG-2B-17R-7C
RGBCG	14R-1BG-2B-5C-1BG	
BRGCG	2B-14R-1BG-5C-1BG	
RCBI		17R-6C-2B-1C-2BG

TABLE II: Address mapping policies for HBM and DDR4. The default policies of HBM and DDR4 are marked blue.

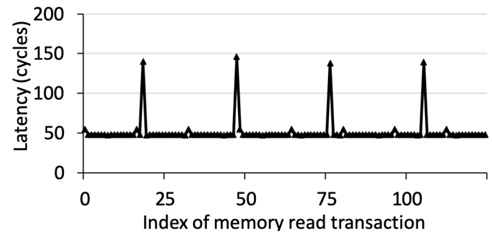
C. Resource Consumption Breakdown

In this subsection, we breakdown the resource consumption of the hardware design of Shuhai when benchmarking HBM.[8] Table III shows the exact FPGA resource consumption of each instantiated module. We observe that Shuhai requires a reasonably small amount of resources to instantiate 32 engine modules, as well as additional components such as the PCIe module, with the total resource utilization being less than 8%.

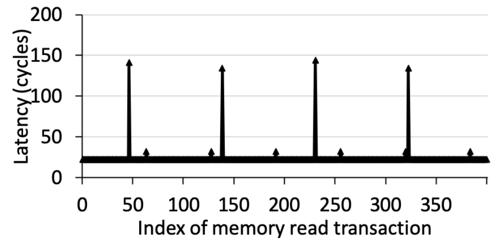
Hardware modules	LUTs	Registers	BRAMs	Freq.
Engine	25824	34048	0	450MHz
PCIe	70181	66689	4.36Mb	250MHz
Parameter	1607	2429	0	250MHz
Latency	672	1760	1.17Mb	250MHz
Total resources used	104K	122K	5.53Mb	
Total utilization	8%	5%	8%	

TABLE III: Resource consumption breakdown of the hardware design for benchmarking HBM

[8]Due to space constraints, we omit the resource consumption for benchmarking DDR4 memory on the FPGA.



(a) HBM



(b) DDR4

Fig. 4: Higher access latency of memory refresh commands that occur periodically on HBM and DDR4.

D. Benchmarking Methodology

We aim to unveil the underlying details of HBM stacks on Xilinx FPGAs under Shuhai. As a yardstick, we also analyze the performance characteristics of DDR4 on the same FPGA board U280 [47] when necessary. When we benchmark a HBM channel, we compare the performance characteristics of HBM with that of DDR4 (in Section V). We believe that the numbers obtained for a HBM channel can be generalized to other computing devices such as CPUs or GPUs featuring HBMs. When benchmarking the switch inside the HBM memory controller, we do not do the comparison with DDR, since the DDR4 memory controller does not contain such a switch (Section VI).

V. Benchmarking an HBM Channel

In this section, we aim to unveil the underlying performance details of a HBM channel on Xilinx FPGAs via using Shuhai.

A. Effect of Refresh Interval

When a memory channel is operating, memory cells should be refreshed repetitively such that the information in each memory cell is not lost. During a refresh cycle, normal memory read and write transactions are not allowed to access the memory. We observe that a memory transaction that experiences a memory refresh cycle exhibits a significantly longer latency than a normal memory read/write transaction that is allowed to directly access the memory chips. Thus, we are able to roughly determine the refresh interval by leveraging memory latency differences between normal and in-a-refresh memory transactions. In particular, we leverage Shuhai to measure the latency of serial memory read operations. Figure 4 illustrates the case with B = 32, S = 64, W = 0x1000000, and N = 1024. We have two observations. First, for both HBM and DDR4, a memory read transaction that coincides with an active refresh command has significantly longer latency, indicating the need to issue enough on-the-fly memory transactions to amortize the negative effect of refresh commands. Second, for both HBM and DDR4, refresh commands are scheduled periodically, the interval between any two consecutive refresh commands being roughly the same.

B. Memory Access Latency

We leverage Shuhai to accurately measure the latency of consecutive memory read transactions when the memory controller is in an "idle" state, i.e., where no other pending memory transactions exist in the memory controller such that the memory controller is able to return the requested data to the read transaction with minimum latency. We aim to identify latency cycles of three categories: *page hit*, *page closed*, and *page miss*.[9]

Page Hit. The "page hit" state occurs when a memory transaction accesses a row that is open in its bank, so no Precharge and Activate commands are required before the column access, resulting in minimum latency.

Page Closed. The "page closed" state occurs when a memory transaction accesses a row whose corresponding bank is closed, so the row Activate command is required before the column access.

Page Miss. The "page miss" state occurs when a memory transaction accesses a row that does not match the active row in the bank, so one Precharge command and one Activate command are issued before the column access, resulting in maximum latency.

We employ the read module to accurately measure the latency numbers for the cases B = 32, W = 0x1000000, N = 1024, and varying S. Intuitively, the small S leads to high probability to hit the same page while a large S potentially leads to a page miss. Besides, a refresh command closes all the active banks. In this experiment, we use two values of S: 128 and 128K.

We use the case S=128 to determine the latency of page hit and page closed transactions. S=128 is smaller than the page size, so the majority of read transactions will hit an open page, as illustrated in Figure 5. The remaining points illustrate the latency of page closed transactions, since the small S leads to a large amount of read transactions in a certain memory region and then a refresh will close the bank before the access to another page in the same bank.[10]

We use the case S=128K to determine the latency of a page miss transaction. S=128K leads to a page miss for each memory transaction for both HBM and DDR4, since two consecutive memory transaction will access the same bank but different pages.

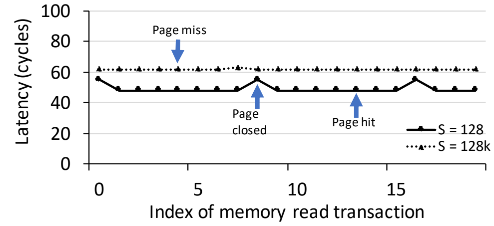
Put it All Together. We summarize the latency on HBM and DDR in Table IV. We have two observations. First, the memory access latency on HBM is higher than that on DDR4 by about 30 nano seconds under the same category like page hit. It means that HBM could have disadvantages when running latency-sensitive applications on FPGAs. Second, the latency number is accurate, demonstrating the efficiency of Shuhai.

Idle Latency	HBM		DDR4	
	Cycles	Time	Cycles	Time
Page hit	48	106.7 ns	22	73.3 ns
Page closed	55	122.2 ns	27	89.9 ns
Page miss	62	137.8 ns	32	106.6 ns

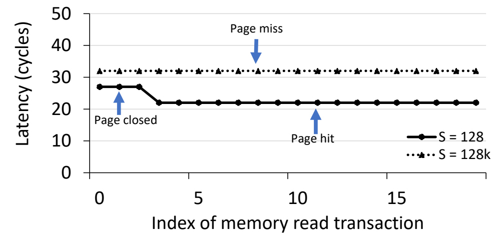
TABLE IV: Idle memory access latency on HBM and DDR4. Intuitively, the HBM latency is much higher than DDR4.

[9]The latency numbers are identified when the switch is disabled. The latency numbers will be seven cycles higher when the switch is enabled, as the AXI channel accesses its associated HBM channel through the switch. The switching of bank groups does not affect memory access latency, since at most one memory read transaction is active at any time in this experiment.

[10]The latency trend of HBM is different of that of DDR4 due to the different default address mapping policy. The default address mapping policy of HBM is RGBCG, indicating that only one bank needs to be active at a time, while the default policy of DDR4 is RCB, indicating that four banks are active at a time.



(a) HBM



(b) DDR4

Fig. 5: Snapshots of page miss, page closed and page hit, in terms of latency cycles, on HBM and DDR4.

C. Effect of Address Mapping Policy

In this subsection, we examine the effect of different memory address mapping policies on the achievable throughput. In particular, under different mapping policies, we measure the memory throughput with varying stride S and burst size B, while keeping the working set size W (= 0x10000000) large enough. Figure 6 illustrates the throughput trend for different address mapping policies for both HBM and DDR4. We have five observations.

First, different address mapping policies lead to significant performance difference. For example, Figure 6a illustrates that the default policy (RGBCG) of HBM is almost 10X faster than the policy (BRC) when S is 1024 and B is 32, demonstrating the importance of choosing the right address mapping policy for a memory-bound application running on the FPGA.

Second, the throughput trends of HBM and DDR4 are quite different even though they employ the same address mapping policy, demonstrating the importance of a benchmark platform such as Shuhai to evaluate different FPGA boards or different memory generations.

Third, the default policy always leads to the best performance for any combination of S and B on HBM and DDR4, demonstrating that the default setting is reasonable.

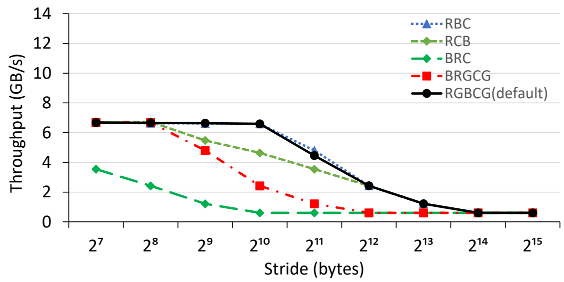
Fourth, small burst sizes lead to low memory throughput, as shown in Figures 6a, 6e, meaning that FPGA programmers should increase spatial locality to achieve higher memory throughput out of HBM or DDR4.

Fifth, large S (>8K) always leads to an extremely low memory bandwidth utilization, indicating the extreme importance of keeping spatial locality. In other words, the random memory access that does not keep spatial locality will experience low memory throughput.

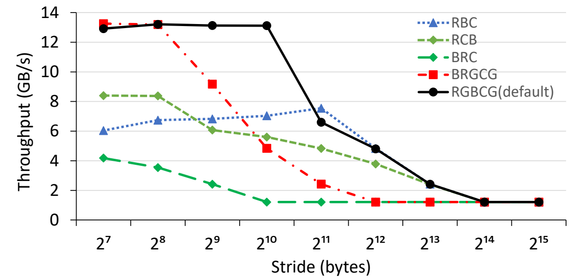
We conclude that choosing the right address mapping policy is critical to optimize memory performance on FPGAs.

D. Effect of Bank Group

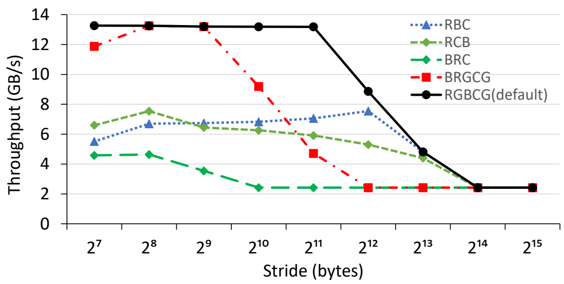
In this subsection, we examine the effect of bank group, which is a new feature of DDR4, compared to DDR3. Accessing multiple bank groups simultaneously helps us relieve the negative effect of DRAM timing restrictions that have not improved over generations



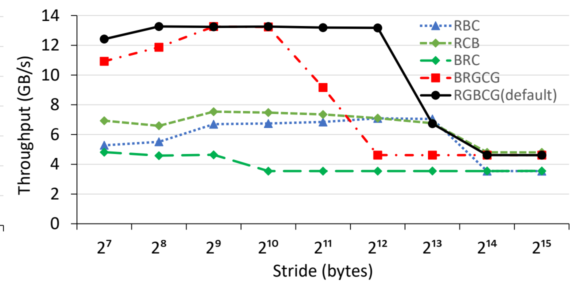
(a) B=32 (HBM)



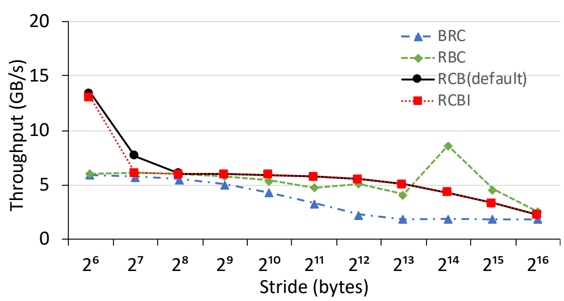
(b) B=64 (HBM)



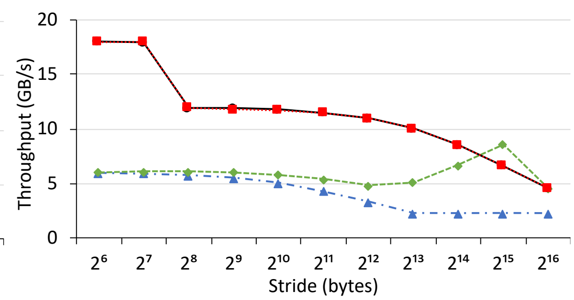
(c) B=128 (HBM)



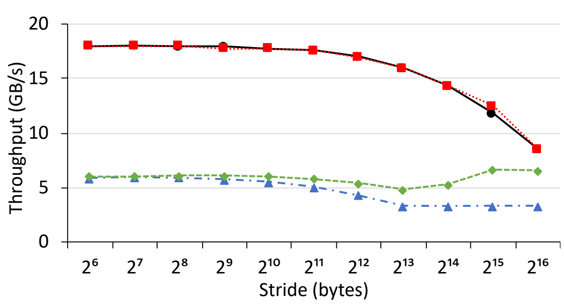
(d) B=256 (HBM)



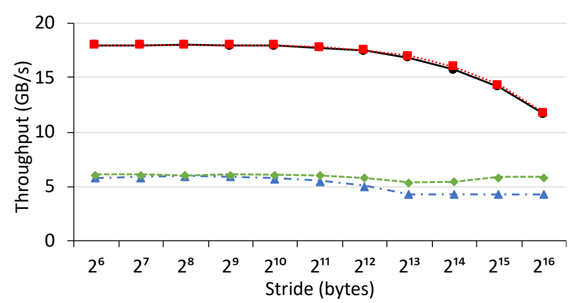
(e) B=64 (DDR4)



(f) B=128 (DDR4)



(g) B=256 (DDR4)



(h) B=512 (DDR4)

Fig. 6: Memory throughput comparison between an HBM channel and a DDR4 channel, with different burst sizes and stride under all the address mapping policies. In this experiment, we use the AXI channel 0 to access its associated HBM channel 0 for the best performance from a single HBM channel. We use the DDR4 channel 0 to obtain the DDR4 throughput numbers. We have two main observations. First, different address mapping policy leads to up to an order of magnitude different performance, indicating the extreme importance of choosing the right address mapping policy for the particular application. Second, the performance characteristics, in terms of throughput trend, of HBM are different from that of DDR4, indicating that not all the experience from DDR4 can directly apply to HBM.

of DRAM. A higher memory throughput can be potentially obtained by accessing multiple bank groups. Therefore, we use the engine module to validate the effect of a bank group (Figure 6). We have two observations.

First, with the default address mapping policy, HBM allows to use large stride size while still keeping high throughput, as shown in Figures 6a, 6b, 6c, 6d. The underlying reason is that even though each row buffer is not fully utilized due to large S, bank-group-level parallelism is able to allow us to saturate the available memory bandwidth.

Second, a pure sequential read does not always lead to the highest throughput under a certain mapping policy. Figures 6b, 6c illustrate that when S increases from 128 to 2048, a bigger S can achieve higher memory throughput under the policy "RBC", since a bigger S allows more active bank groups to be accessed concurrently, while a smaller S potentially leads to only one active bank group that serves user's memory requests.

We conclude that it is critical to leverage bank-group-level parallelism to achieve high memory throughput under HBM.

E. *Effect of Memory Access Locality*

In this subsection, we examine the effect of memory access locality on memory throughput. We vary the burst size B and the stride S, and we set the working set size W to two values: 256M and 8K. The case W=256M refers to the baseline that does not benefit from any memory access locality, while the case W=8K refers to the case that benefits from locality. Figure 7 illustrates the throughput for varying parameter settings on both HBM and DDR4. We have two observations.

First, memory access locality indeed increases the memory throughput for each case with high stride S. For example, the memory bandwidth of the case (B=32, W=8K, and S=4K) is 6.7 GB/s on HBM, while 2.4 GB/s of the case (B=32, W=256M, and S=4K), indicating that memory access locality is able to eliminate the negative effect of a large stride. Second, memory access locality cannot increase the memory throughput when S is small. In contrast, memory access locality can significantly increase the total throughput on modern CPUs/GPUs due to the on-chip caches which have dramatically higher bandwidth than off-chip memory [21].

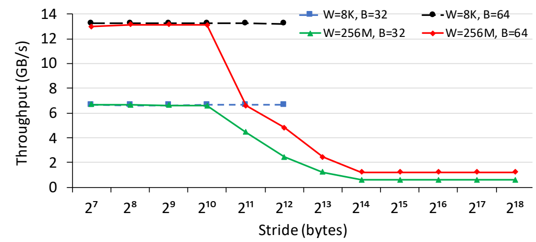
F. *Total Memory Throughput*

In this subsection, we explore the total achievable memory throughput of HBM and DDR4 (Table V). The HBM system on the tested FPGA card, U280, is able to provide up to 425 GB/s (13.27 GB/s * 32) memory throughput when we use all the 32 AXI channels to simultaneously access their associated HBM channels.[11] The DDR4 memory is able to provide up to 36 GB/s (18 GB/s * 2) memory throughput when we simultaneously access both DDR4 channels on our tested FPGA card. We observe that the HBM system has 10 times more memory throughput than DDR4 memory, indicating that the HBM-enhanced FPGA enables us to accelerate memory-intensive applications, which are typically accelerated on GPUs.

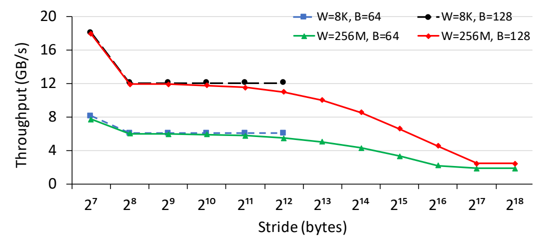
VI. Benchmarking the Switch in the HBM Controller

Each HBM stack segments memory address space into 16 independent pseudo channels, each of which is associated with an AXI port mapped to a particular range of address [45], [48]. Therefore,

[11]Each AXI channel accesses its local HBM channel, there is no inference among the 32 AXI channels. Since each AXI channel approximately has the same throughput, we estimate the total throughput by simply scaling up the throughput of the channel 0 by 32.



(a) HBM



(b) DDR4

Fig. 7: Effect of memory access locality. Intuitively, memory access locality can relieve the negative effect of large stride S.

	HBM	DDR4
Throughput of a channel	13.27 GB/s	18 GB/s
Number of channels	32	2
Total memory throughput	425 GB/s	36 GB/s

TABLE V: Total memory throughput comparison between HBM and DDR4. HBM has an order of magnitude higher memory throughput than DDR4.

the 32 × 32 switch is required to make sure each AXI port is able to reach the whole address. The 32 × 32 switch fully implemented in a HBM memory controller requires a massive amount of logic resources. Thus, the switch is only partially implemented, thereby consuming significantly fewer resources but achieving lower performance for particular accessing patterns. Our goal in this section is to unveil the performance characteristics of the switch.

A. *Performance between AXI Channel and HBM Channel*

In a fully implemented switch, the performance characteristics of the access from any AXI channel to any HBM channel should be roughly the same. However, in the current implementation, the relative distance could play an important role. In the following, we examine the performance characteristics between any AXI channel and any HBM channel, in terms of latency and throughput.

1) Memory Latency: Due to space constraints, we only demonstrate the memory access latency using the memory read transaction issued in any AXI channel (from 0 to 31) to the HBM channel 0.[12] Access to other HBM channels has similar performance characteristics. Similar to the experimental setup in Subsection V-B, we also employ the engine module to determine the accurate latency for the case B = 32, W = 0x1000000, N = 1024, and varying S. Table VI illustrates the latency difference among 32 AXI channels. We have two observations.

First, the latency difference can be up to 22 cycles. For example, for a page hit transaction, an access from the AXI channel 31 needs 77 cycles, while an access from the AXI channel 0 only needs 55 cycles. Second, the access latency from any AXI channel in the same mini-switch is identical, demonstrating that the mini-switch

[12]The switch is enabled to allow global addressing, when comparing the latency difference among AXI channels.

Channels	Page hit		Page closed		Page miss	
	Cycles	Time	Cycles	Time	Cycles	Time
0-3	55	122.2 ns	62	137.8 ns	69	153.3 ns
4-7	56	124.4 ns	63	140.0 ns	70	155.6 ns
8-11	58	128.9 ns	65	144.4 ns	72	160.0 ns
12-15	60	133.3 ns	67	148.9 ns	74	164.4 ns
16-19	71	157.8 ns	78	173.3 ns	85	188.9 ns
20-23	73	162.2 ns	80	177.7 ns	87	193.3 ns
24-27	75	166.7 ns	82	182.2 ns	89	197.8 ns
28-31	77	171.1 ns	84	186.7 ns	91	202.2 ns

TABLE VI: Memory access latency from any of 32 AXI channels to the HBM channel 0. The switch is on. Intuitively, longer distance yields longer latency. The latency difference reaches up to 22 cycles.

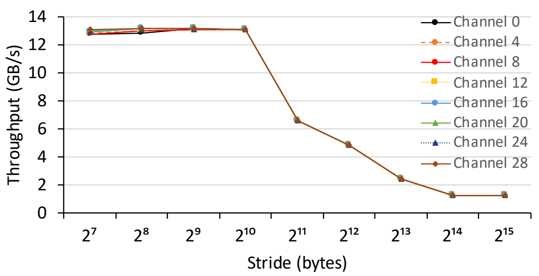


Fig. 8: Memory throughput from eight AXI channels to the HBM channel 1, where each AXI channel is from a mini-switch. Each AXI channel has roughly the same throughput when accessing the HBM channel 1, even though their latency could be obviously different.

is fully-implemented. For example, the AXI channels 4-7 in the same mini-switch have the same access latency to the HBM channel 0. We conclude that an AXI channel should access its associated HBM channel or the HBM channels close to it to minimize latency.

2) Memory Throughput: We employ the engine module to measure memory throughput from any AXI channel (from 0 to 31) to HBM channel 0, with the setting B = 64, W = 0x1000000, N = 200000, and varying S. Figure 8 illustrates the memory throughput from an AXI channel in each mini-switch to the HBM channel 0. We observe that AXI channels are able to achieve roughly the same memory throughput, regardless of their locations.

B. Interference among AXI Channels

We examine the effect of interference among AXI channels by using a varying number (e.g., 2, 4, and 6) of remote AXI channels to simultaneously access the same HBM channel 1. We also vary the size of B. Table VII shows the throughput with different values of B and a different number of remote AXI channels. The empty slot indicates that this remote AXI channel is not involved in the throughput testing. We have two observations. First, the total throughput slightly decreases when the number of remote AXI channels increases, indicating that the switch is able to serve memory transactions from multiple AXI channels in a reasonably efficient way. Second, two lateral connections and four masters within a mini-switch are scheduled in a round-robin manner. Take the case (AXI channels 4, 5, 8 and 9 concurrently access and B=32) as an example, the total throughput of the remote channels 8 and 9 are roughly equal to that of channels 4 or 5.

Channel	4	5	8	9	12	13	In total (GB/s)
B=32	3.15	3.15					6.30
B=32	2.10	2.10	1.05	1.05			6.30
B=32	2.10	2.10	0.70	0.70	0.35	0.35	6.30
B=64	5.64	5.64					11.28
B=64	3.42	3.42	1.73	1.73			10.30
B=64	2.81	2.81	0.95	0.95	0.48	0.48	8.48
B=128	6.09	6.09					12.18
B=128	3.52	3.52	1.78	1.78			10.60
B=128	3.31	3.31	1.11	1.11	0.56	0.56	9.96

TABLE VII: Effect of inference among remote AXI channels. We measure the throughput (GB/s) with a varying number (2, 4, or 6) of remote AXI channels to access the HBM channel 1. When the number of remote AXI channels is 2, the AXI channels 4 and 5 are active. When only using the local AXI channel 1 to access the HBM channel 1, the throughput is 6.67, 12.9 or 13.3 GB/s for the case with B = 32, B = 64 or B = 128. The empty slot indicates the corresponding AXI channel is not involved in the particular benchmarking.

VII. Related Work

To our knowledge, Shuhai is the first platform to benchmark HBM on FPGAs in a systematic and comprehensive manner. We contrast closely related work with Shuhai on 1) benchmarking traditional memory on FPGAs; 2) data processing with HBM; and 3) accelerating application with FPGAs.

Benchmarking Traditional Memory on FPGAs. Previous work [22], [24], [25], [51] tries to benchmark traditional memory, e.g., DDR3, on the FPGA by using high-level languages, e.g., OpenCL. In contrast, we benchmark HBM on the state-of-the-art FPGA.

Data Processing with HBM/HMC. Previous work [4], [6], [17], [18], [23], [30], [31], [50] employs HBM to accelerate their applications, e.g., hash table deep learning and streaming, by leveraging the high memory bandwidth provided by Intel Knights Landing (KNL)'s HBM [14]. In contrast, we benchmark the performance of HBM on the Xilinx FPGA board.

Accelerating Applications with FPGAs. Previous work [1], [3], [5], [8], [9], [10], [11], [12], [13], [16], [19], [26], [28], [29], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [49] accelerates a broad range of applications, e.g., database and deep learning inference, using FPGAs. In contrast, we systematically benchmark HBM on the state-of-the-art FPGA regardless of the application.

VIII. Conclusion

FPGAs are being enhanced with High Bandwidth Memory (HBM) to tackle the memory bandwidth bottleneck that dominates memory-bound applications. However, the performance characteristics of HBM are still not quantitatively and systematically analyzed on FPGAs. We bridge the gap by benchmarking HBM stack on a state-of-the-art FPGA featuring a two-stack HBM2 subsystem. Accordingly, we propose Shuhai to demystify the underlying details of HBM such that the user is able to obtain a more accurate picture of the behavior of HBM than what can be obtained by doing so on CPUs/GPUs as they introduce noise from the caches. From the benchmarking numbers obtained, we observe that 1) HBM provides up to 425 GB/s memory bandwidth, which is roughly half of memory bandwidth on a state-of-the-art GPU, and 2) how HBM is used has a significant impact on performance, which in turn demonstrates the importance of unveiling the performance characteristics of HBM. Shuhai can be easily generalized to other FPGA boards or other generations of memory modules. We will make the related benchmarking code open-source such that new FPGA boards can be explored and the results across boards are compared. The code is available: https://github.com/RC4ML/Shuhai.

Acknowledgements. We thank the valuable feedback from Xilinx University Program to improve the quality of this paper. This work is supported by the National Natural Science Foundation of China (U19B2043, 61976185), and the Fundamental Research Funds for the Central Universities.

References

[1] Altera. Guidance for Accurately Benchmarking FPGAs, 2007.

[2] Arm. AMBA AXI and ACE Protocol Specification, 2017.

[3] M. Asiatici and P. Ienne. Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas. In *FPGA*, 2019.

[4] B. Bramas. Fast Sorting Algorithms using AVX-512 on Intel Knights Landing. *CoRR*, 2017.

[5] E. Brossard, D. Richmond, J. Green, C. Ebeling, L. Ruzzo, C. Olson, and S. Hauck. A model for programming data-intensive applications on fpgas: A genomics case study. In *SAAHPC*, 2012.

[6] X. Cheng, B. He, E. Lo, W. Wang, S. Lu, and X. Chen. Deploying hash tables on die-stacked high bandwidth memory. In *CIKM*, 2019.

[7] K. Cho, H. Lee, H. Kim, S. Choi, Y. Kim, J. Lim, J. Kim, H. Kim, Y. Kim, and Y. Kim. Design optimization of high bandwidth memory (hbm) interposer considering signal integrity. In *EDAPS*, 2015.

[8] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *DAC*, 2016.

[9] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.*, 2019.

[10] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *FCCM*, 2014.

[11] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. Spector: An OpenCL FPGA benchmark suite. In *FPT*, 2016.

[12] Z. He, Z. Wang, and G. Alonso. Bis-km: Enabling any-precision k-means on fpgas. In *FPGA*, 2020.

[13] Z. István, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *FCCM*, 2016.

[14] Jim Jeffers and James Reinders and Avinash Sodani. Intel Xeon Phi Processor High Performance Programming Knights Landing Edition, 2016.

[15] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim. Hbm (high bandwidth memory) dram technology and architecture. In *IMW*, 2017.

[16] S. Jun, M. Liu, S. Xu, and Arvind. A transport-layer network for distributed fpga platforms. In *FPL*, 2015.

[17] S. Khoram, J. Zhang, M. Strange, and J. Li. Accelerating graph analytics by co-optimizing storage and access on an fpga-hmc platform. In *FPGA*, 2018.

[18] A. Li, W. Liu, M. R. B. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song. Exploring and Analyzing the Real Impact of Modern On-Package Memory on HPC Scientific Kernels. In *SC*, 2017.

[19] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow. An fpga-based processor for training convolutional neural networks. In *FPT*, 2017.

[20] J. Macri. Amd's next generation gpu and high bandwidth memory architecture: Fury. In *Hot Chips*, 2015.

[21] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *PVLDB*, 2002.

[22] K. Manev, A. Vaishnav, and D. Koch. Unexpected diversity: Quantitative memory analysis for zynq ultrascale+ systems. In *FPT*, 2019.

[23] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. In *ASPLOS*, 2019.

[24] S. W. Nabi and W. Vanderbauwhede. Mp-stream: A memory performance benchmark for design space exploration on heterogeneous hpc devices. In *IPDPSW*, 2018.

[25] S. W. Nabi and W. Vanderbauwhede. Smart-cache: Optimising memory accesses for arbitrary boundaries and stencils on fpgas. In *IPDPSW*, 2019.

[26] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet. Lowering the latency of data processing pipelines through fpga based hardware acceleration. In *PVLDB*, 2019.

[27] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally. Fine-grained dram: Energy-efficient dram for extreme bandwidth systems. In *MICRO*, 2017.

[28] M. J. H. Pantho, J. Mandebi Mbongue, C. Bobda, and D. Andrews. Transparent Acceleration of Image Processing Kernels on FPGA-Attached Hybrid Memory Cube Computers. In *FPT*, 2018.

[29] H. Parandeh-Afshar, P. Brisk, and P. Ienne. Efficient synthesis of compressor trees on fpgas. In *ASP-DAC*, 2008.

[30] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. Exploring the performance benefit of hybrid memory system on hpc environments. In *IPDPSW*, 2017.

[31] C. Pohl and K.-U. Sattler. Joins in a heterogeneous memory hierarchy: Exploiting high-bandwidth memory. In *DAMON*, 2018.

[32] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides. A case for work-stealing on fpgas with opencl atomics. In *FPGA*, 2016.

[33] S. Taheri, P. Behnam, E. Bozorgzadeh, A. Veidenbaum, and A. Nicolau. Affix: Automatic acceleration framework for fpga implementation of openvx vision algorithms. In *FPGA*, 2019.

[34] D. B. Thomas, L. Howes, and W. Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *FPGA*, 2009.

[35] S. I. Venieris and C. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *FCCM*, 2016.

[36] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen. Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga. In *FPL*, 2018.

[37] Z. Wang, B. He, and W. Zhang. A study of data partitioning on OpenCL-based FPGAs. In *FPL*, 2015.

[38] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*, 2016.

[39] Z. Wang, K. Kara, H. Zhang, et al. Accelerating Generalized Linear Models with MLWeaving: A One-size-fits-all System for Any-precision Learning. *VLDB*, 2019.

[40] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on OpenCL-based FPGAs. In *FPL*, 2016.

[41] Z. Wang, J. Paul, B. He, and W. Zhang. Multikernel data partitioning with channel on OpenCL-based FPGAs. *TVLSI*, 2017.

[42] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A MapReduce framework on OpenCL-based FPGAs. *TPDS*, 2016.

[43] J. Weberruss, L. Kleeman, D. Boland, and T. Drummond. Fpga acceleration of multilevel orb feature extraction for computer vision. In *FPL*, 2017.

[44] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe. A study of pointer-chasing performance on shared-memory processor-fpga systems. In *FPGA*, 2016.

[45] M. Wissolik, D. Zacher, A. Torza, and B. Day. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance, 2019.

[46] Xilinx. AXI Reference Guide, 2011.

[47] Xilinx. Alveo U280 Data Center Accelerator Card Data Sheet, 2019.

[48] Xilinx. AXI High Bandwidth Memory Controller v1.0, 2019.

[49] Q. Xiong, R. Patel, C. Yang, T. Geng, A. Skjellum, and M. C. Herbordt. Ghostsz: A transparent fpga-accelerated lossy compression framework. In *FCCM*, 2019.

[50] Y. You, A. Buluç, and J. Demmel. Scaling deep learning on gpu and knights landing clusters. In *SC*, 2017.

[51] H. R. Zohouri and S. Matsuoka. The memory controller wall: Benchmarking the intel fpga sdk for opencl memory interface. In *H2RC*, 2019.