Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs

Hongjing Huang, Zeke Wang*, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, Gustavo Alonso

Abstract—FPGAs are starting to incorporate High Bandwidth Memory (HBM) to both reduce the memory bandwidth bottleneck encountered in some applications and to provide more capacity to store application state. However, the overall performance characteristics of HBMs are still not well understood, especially in the context of FPGAs, making it difficult to optimize designs relying on HBM. In this paper, we bridge the gap between nominal specifications and actual performance by characterizing HBM on a state-of-the-art FPGA, i.e., a Xilinx Alveo U280 featuring a two-stack HBM subsystem. To this end, we have developed Shuhai, a benchmarking tool that throws light on all the subtle details of the performance and usage of HBMs on an FPGA. FPGA-based benchmarking should also provide a more accurate picture of HBM than measuring performance 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 it directly from an FPGA provides more details as well as more 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 the achievable throughput, which in turn demonstrates the importance of unveiling the performance characteristics of HBM so as to use HBM in the right manner. To demonstrate the generality of Shuhai, we also show results for other types of memory, e.g., DDR4, and DDR3, and quantitatively compare the performance characteristics of HBM with those of DDR4 and DDR3.

Index Terms—High Bandwidth Memory, benchmarking, FPGA, DDR, latency, throughput.

1 Introduction

The computational capacity of modern computing systems continues to increase due to the constant improvements of CMOS technology. These improvements are typically achieved by either instantiating more cores within the same area and/or by adding extra functionality to the cores (e.g., AVX, SGX, GPGPU, etc.). In contrast, the DRAM bandwidth has improved only slowly over many generations. As a result, the gap between memory and processor speeds 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 new memory variants, e.g., Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM), as a way to provide significantly higher memory bandwidth. For example, 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 often 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 Alveo U280 [1].[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 [1] 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 [2] or deep learning inference [3]. It can also support applications by 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 significantly differ from those of, e.g., DDR4. Second, Xilinx's HBM subsystem [4] 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 issues can hamper the ability of FPGA developers to fully exploit the advantages of HBM on FPGAs.

- *Hongjing Huang, Zeke Wang, Jie Zhang, and Jun Xiao are with the Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China. E-mail: {Hu, wangzeke, carlzhang4}@zju.edu.cn, junx@cs.zju.edu.cn.*
- *Zhenhao He, and Gustavo Alonso are with Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland. E-mail: {zhe, alonso}@inf.ethz.ch.*
- *Chao Wu is with School of Public Affairs, Zhejiang University, China. E-mail: chao.wu@zju.edu.cn.*
- **Zeke Wang is the corresponding author.*

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.

To address the issue, in this paper we present Shuhai,[5] a benchmarking tool that allows us to demystify all the underlying details of HBMs. 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 tasks. To our knowledge, Shuhai is the first platform to systematically benchmark HBM on an FPGA. We demonstrate the usefulness of Shuhai by identifying three important aspects on the usage of HBM-enhanced FPGAs.

(1) 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.

(2) 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.

(3) The latency of HBM is Much Higher than DDR4. Shuhai identifies that the latency of HBM is 106.7 ns while the latency of DDR4 is 73.3 ns (Section 6), when the memory transaction hits an open page (or row). As a result, to saturate the HBM bandwidth, we need more on-the-fly memory transactions, something that it is possible on modern FPGAs/GPUs.

The paper makes the two key contributions:

- We develop Shuhai, a benchmarking tool that makes it easier to reason about the performance characteristics of various memory types, e.g., HBM and DDR4. On an FPGA, Shuhai allows us to get accurate numbers when benchmarking memory, whether HBM or DDR, thus providing invaluable help to developers interested in maximizing performance for memory bandwidth bound designs.
- Shuhai utilizes runtime parameters in the benchmarking circuit so as to cover a broad range of benchmarking tasks without requiring to reconfigure the FPGA.

2 Background

An HBM chip employs the latest development of IC packaging technologies, such as Through Silicon Via (TSV), stacked-DRAM, and 2.5D package [5], [6], [7], [8]. The basic structure of HBM consists of a base logic die at the bottom with 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/HBM channels.

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

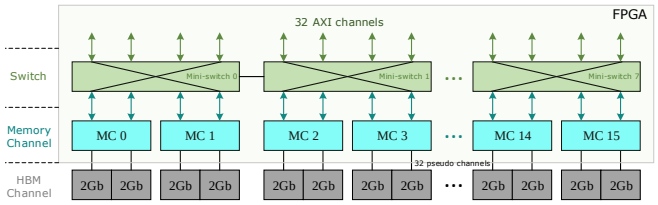

Fig. 1. Architecture of Xilinx HBM subsystem

On top of the 16 memory channels, there are 32 *AXI channels* enabling interaction with the user logic. Each AXI channel adheres to the AXI3 protocol [4] to provide a 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 introduced an optional switch between the 32 AXI channels and the 32 pseudo channels [4], [9].[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. The mini-switch is fully implemented in the sense that each AXI channel has accesses to 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.

3 Motivation: Benchmarking Memory on CPUs/GPUs

Benchmarking memory on FPGAs, rather than relying on figures from CPUs or GPUs, is motivated by the difficulty of eliminating the effects of the cache hierarchy and the TLB when benchmarking memory on CPUs/GPUs. In this section, we illustrate the issues associated to benchmarking memory latency on CPUs/GPUs.

3.1 Tested Hardware Platform

We run the CPU benchmarking experiments on a 10-core Intel CPU i9-10900X. Each core has a dedicated 32KB L1 cache and a dedicated 1MB L2 cache. A 19.25MB L3 cache is shared between 10 cores. The CPU has four 2133MHz 32GB DDR4 channels, each of which has 72-bit data width.

We run the GPU benchmarking experiments on an NVIDIA GPU RTX2080Ti. This GPU has a two-level caching system and 68 stream multiprocessors (SMs), each of which contains a combined 96 KB L1 data cache. The 5632 KB L2 cache is shared among all the SMs [10], [11]. The GPU features 11 GB GDDR6 which has 14 Gbps memory speed and a 352-bit aggregated memory bus, with its theoretical memory bandwidth reaching 616 GB/s.

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

Both the CPU and the GPU feature multiple memory channels, to which the CPU/GPU interleaves access, and the exact interleaving policy is transparent to the user. Therefore, our benchmarking code has to target all the channels, which is slightly different from benchmarking only one channel on the FPGA.

3.2 Measuring Memory Latency

Since CPUs/GPUs have both a cache and a TLB, which have the negative effect on external memory performance, the goal of this subsection is to measure latency of each memory access that hits the L1 TLB but misses in last level cache. To achieve this, we employ the fine-grained P-chase method [12], [13] to measure every single memory access latency on GPUs/CPUs, as illustrated in Listing 1. This method consists of two steps.

In this first step, we initialize the targeted array $array$ at the first step by setting the i-th element to $(i + S)\%size$ (Lines 4-6). There are two constraints on $size$. First, $size$ is significantly larger than the last-level cache size such that each memory access will reference external memory. Second, $size$ is smaller than the L1 TLB size multiplied by the number of L1 TLB entries, such that any further memory access to $array$ in the second step causes a TLB hit.

In the second step, we traverse the $array$, beginning with the first element ($array[0]$) of value S. Therefore, the second access jumps to the S-th element ($array[S]$) of value $2 * S$, and so on. By doing so, this method enables strided memory access, whose stride S is parameterized. We use the N-element array $index$ to store each access index j (Line 15), such that our compiler does not optimize away the P-chase part (Line 14). We use the N-element array $latency$ to store the latency number for each memory access (Line 17).

```
1  Input S, N, size; //S: stride, N: number of memory transactions
2                    //size: traversed array size
3  //Step 1: initializing the P-chase array
4  for(i=0;i<size;i++){ //last-level cache size < size < L1 TLB size
5      array[i]    = (i + S) % size; //array stays at external memory
6  }
7
8  //Step 2: traversing the P-chase array and storing each latency
9  j               = 0;    //set the first index to begin
10 int index[N];           //declare memory for array index
11 int latency[N];         //declare memory for latency
12 for( i=0; i < N; i++){
13     start       = clock();
14     j           = array[j];    //P-chase: cache miss & TLB hit
15     index[i]    = j;           //store the array index
16     end         = clock();
17     latency[i]  = end - start; //store the latency
18 }
```

Listing 1. Fine-grained P-chase method

3.2.1 Benchmarking on GPUs

We employ the fine-grained P-chase method in Listing 1 in an SM with a single thread to measure memory access latency. In the context of GPU, we use CUDA-specific *clock()* function to measure latency cycles. Besides, we allocate shared memory for $index$ and $latency$ such that any related memory write instruction will commit faster, minimizing the negative effect on the P-chase part.

We leverage the work [13], [14], to determine the cache and TLB parameters of the GPU respectively, e.g., last level (L2) cache size is 5632 KB, L1 TLB has 16 entries and L1 TLB size is 32 MB. Therefore, S is 2048 and $size$ is 16 MB in our experiment. Figure 2 illustrates the latency of each memory read transaction. We observe that the latency of *page hit*, *page closed*, and *page miss* is indistinguishable on the GPU due to the complex design of cache and pipeline.

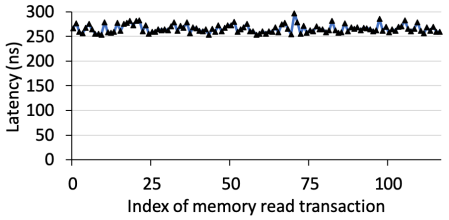

Fig. 2. Global memory access latency on the GPU (S=2KB, W=16MB)

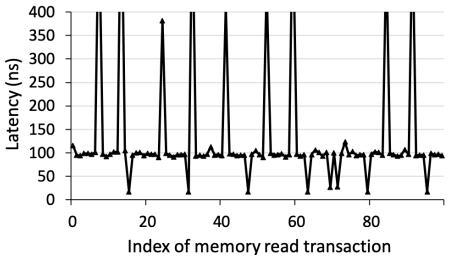

Fig. 3. Memory access latency on the CPU when S = 256 bytes

3.2.2 Benchmarking on CPUs

In the context of CPU, we implement the fine-grained P-chase method in Listing 1 using embedded assembly code to eliminate unnecessary noise from compilation. The L3 cache size is 19.25 MB on the tested CPU. With the goal of trying our best to make each memory access miss in the L3 cache and hit in the L1 data TLB, we use 2 MB huge pages, instead of 4 KB pages, since TLB entries are always scarce resources. The number of L1 TLB entries for 2 MB huge pages is 32 in our tested CPU. Therefore, S is set to 512 and $size$ is set to 32 MB in our experiment. Figure 3 illustrates a snapshot of latency measured at an idle core. The memory transaction issued from the fine-grained P-chase method cannot always simultaneously miss in the cache and hit in the L1 data TLB. This is because the replacement policies of the cache and TLB are not pure LRU (Intel does not unveil the exact replacement policies). As a result, we still see a small fraction of memory transactions that encounter TLB misses (latency $>$ 200 ns) and that hit in cache (latency $<$ 50 ns).

3.2.3 Putting Everything Together

As illustrated in Figures 2 and 3, we cannot determine accurate latency numbers on the CPU/GPU due to the many interposed layers and the severe interference introduced by factors such as TLB, multi-level caches, or instruction scheduling. This motivates us to explicitly benchmark memory on FPGAs, which allow us to easily reason about simple benchmarking logic (§4). Accordingly, we propose Shuhai that allows to easily obtain accurate memory latency measurements (Table 4).

4 General Benchmarking Tool Shuhai

4.1 Design Methodology

In designing Shuhai, we have two ambitious goals.

First, we aim to provide high-level insights on the performance characteristics of HBM (**C1**).This is critical to make the benchmarking results meaningful and understandable to FPGA programmers. In particular, Shuhai should give the programmer an end-to-end explanation, rather than just memory timing parameters such as row precharge time T_{RP}, so that the insights can be used to improve designs using HBM on FPGAs.

Second, Shuhai should be easy to use (**C2**). This is difficult to achieve as benchmarking on FPGAs involves many subtle designs aspects: a small modification might need to reconfigure the FPGA, making the process of benchmarking and comparing alternative design cumbersome and time consuming. Therefore, Shuhai minimizes the reconfiguration effort between tasks: it should use a single FPGA image for a large number of benchmarking tasks, instead of using one image for each benchmarking task.

4.1.1 Our Approach

To achieve the first goal, **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 accomplish the second goal, **C2**, Shuhai uses runtime parameters of the benchmarking circuit so as to cover a wide range of benchmarking tasks without requiring to reconfigure 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 4.2) and a hardware component (Subsection 4.3). The main role of the software component is to provide flexibility to the FPGA programmer in terms of runtime parameters. Through the use of 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 should not be the bottleneck.

4.2 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)* (Figure 4).

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 1. We calculate the address $T[i]$ 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.

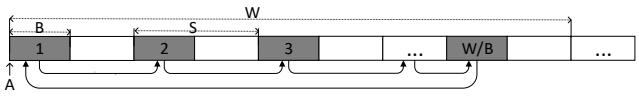

Fig. 4. Memory access pattern RST used in Shuhai.

TABLE 1
Summary of runtime parameters

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)

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

4.3 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 5. In the following, we discuss the implementation details for each module.

4.3.1 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 [15], [16], 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) [15]. Moreover, 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. Using the same clock brings in two benefits. First, no extra noise, e.g., 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.

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. The write module is able to measure both latency and throughput. When measuring throughput, this module always tries to saturate the memory write channels WR and WD by asserting the associated valid signals before the writing task completes,

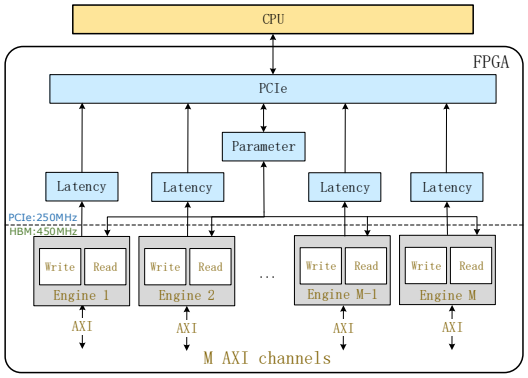

Fig. 5. Overall hardware architecture of our benchmarking framework. It can support M hardware engines running simultaneously, with each engine for one AXI channel. In our experiment, M is 32 for HBM, while M is 2 for DDR4.

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. When measuring latency, we immediately issue the second memory write transaction after the finishing signal of the first write transaction is received via the WR channel.

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. The read module is able to measure latency and throughput. When measuring the latency of memory read transactions, we immediately issue the second memory read transaction 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.

4.3.2 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.

4.3.3 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.

4.3.4 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 of a memory read/write.

7. We are able to unveil many performance characteristics of HBM 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, 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.

TABLE 2
Address mapping policies for HBM, DDR4 and DDR3. The default policies of HBM and DDR4 are marked blue.

Policies	HBM (addr[27:5])	DDR4 (addr[33:6])	DDR3 (addr[28:0])
RBC	14R-2BG-2B-5C	17R-2BG-2B-7C	16R-3B-10C
RCB	14R-5C-2BG-2B	17R-7C-2B-2BG	
BRC	2BG-2B-14R-5C	2BG-2B-17R-7C	3B-16R-10C
RGBCG	14R-1BG-2B-5C-1BG		
BRGCG	2B-14R-1BG-5C-1BG		
RCBI		17R-6C-2B-1C-2BG	

5 Experiment Setup

5.1 Hardware Platform

We run most experiments on a Xilinx's Alevo U280 [1] featuring two HBM stacks of a total size of 8GB and two DDR4 memory channels with a total size of 32 GB. The theoretical HBM memory bandwidth is 450 GB/s (450 MHz * 32 * 32 B/s), while the theoretical DDR4 memory bandwidth is 38.4 GB/s (300 MHz * 2 * 64 B/s). We also employ an ADM-PCIE-7V3 FPGA board featuring two DDR3 channels with 8GB in total, with a theoretical memory bandwidth of 21.3GB/s (166.7MHz * 2 * 64 B/s).

5.2 Address Mapping Policies (AMPs)

An 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 AMP is critical to maximize the overall memory throughput. The policies enabled for HBM, DDR4 and DDR3 are summarized in Table 2, 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, DDR4 and DDR3 are "RGBCG", "RCB" and "BRC", respectively. "-" stands for address concatenation. We always use the default AMP for any memory if not explicitly specified.

5.3 Resource Consumption Breakdown

We break down the resource consumption of the hardware design of Shuhai when benchmarking HBM.[8] Table 3 shows

8. Due to space constraints, we omit the resource consumption for benchmarking DDR4 and DDR3 memory on the FPGA.

TABLE 3
Resource consumption breakdown of the hardware design for benchmarking HBM

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%	

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. The total resource utilization is less than 8%.

5.4 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 [1] and DDR3 when necessary (in Section 6). 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 DDR memory controller does not contain such a switch (Section 7).

6 Benchmarking an HBM Channel

6.1 Effect of Refresh Interval

When a memory channel is operating, memory cells should be regularly refreshed 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 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 sequential memory read operations. Figure 6 illustrates the case with B = 32, S = 64, W = 0x1000000, and N = 1024. We have two observations. First, for HBM, DDR4, and DDR3, 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 HBM, DDR4 and DDR3, refresh commands are scheduled periodically, the interval between any two consecutive refresh commands being roughly the same.

6.2 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]

TABLE 4
Idle memory access latency on HBM and DDR4

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

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.

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.

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 of Shuhai to accurately measure the latency numbers for the cases with 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. A refresh command also 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 7. 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.

We summarize the latency on HBM and DDR4 in Table 4.[11] We observe that 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.

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 AMP. The default AMP 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.

11. We omit the latency on DDR3 since we cannot obtain the exact latency number for each category, as shown in Figure 6c.

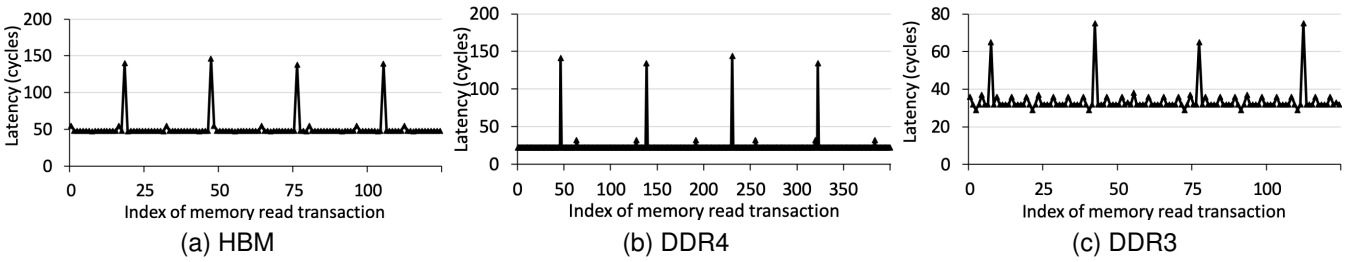

Fig. 6. Higher access latency of memory refresh commands that occur periodically on HBM, DDR4, and DDR3.

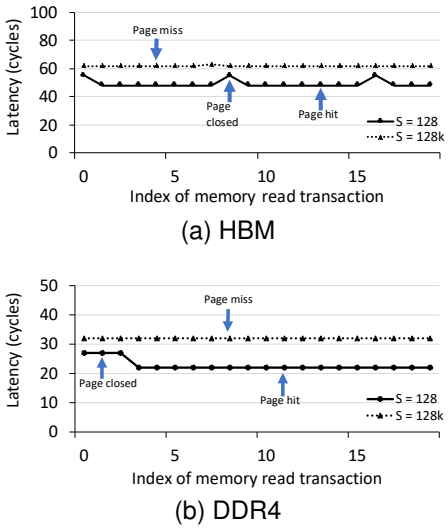

Fig. 7. Snapshots of a page miss, page closed and page hit, in terms of latency cycles, on HBM and DDR4.

6.3 Effect of Address Mapping Policy (AMP)

AMP plays a critical role in achieving high memory throughput. Intuitively, the input memory access pattern is able to yield the highest possible throughput with a right AMP. In this subsection, we examine the effect of AMP, in terms of achievable throughput, on the memory access pattern RST in Figure 4. In particular, under different AMPs, we measure the memory throughput with varying stride S and burst size B, while keeping the working set size W (= 0x10000000) large enough. Figure 8 illustrates the throughput trend for different AMPs for both HBM, DDR4 and DDR3. We have five observations.

First, different AMPs lead to significant performance difference. For example, Figure 8a 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 AMP for a memory-bound application running on the FPGA. Second, the throughput trends of HBM, DDR4 and DDR3 are quite different even though they employ the same AMP, demonstrating the importance of a benchmark platform such as Shuhai to evaluate different FPGA boards or different memory generations. Third, the default AMP always leads to the best performance for any combination of S and B on HBM, DDR4, and DDR3, demonstrating that the current default AMP is reasonable. Fourth, small burst sizes lead to low memory throughput, as shown in Figures 8a, 8b, meaning that FPGA programmers should increase spatial locality to achieve higher memory throughput out of HBM or DDR. Fifth, large S (>8K) always leads to an extremely low memory bandwidth utilization, indicating the extreme importance of keeping spatial locality. In other words, random memory access that does not keep spatial locality will experience low memory throughput. We conclude that choosing the right AMP is critical to optimizing memory performance for a particular memory access pattern on FPGAs.

6.4 Effect of Bank Group

We examine the effect of bank group, which is a new feature of DDR4, compared to DDR3. Accessing multiple bank groups simultaneously relieves the negative effect of DRAM timing restrictions that have not improved over DRAM generations. A higher memory throughput can be potentially obtained by accessing multiple bank groups. Figure 8 also illustrates the effect of bank group. We have two observations.

First, with the default AMP, DDR4 and HBM allows using a large stride size (up to 4K) while still keeping high throughput, as shown in Figures 8h, 8g. However, DDR3 only allows to use small stride (up to 256) in Figure 8i. The underlying reason is that even though each row buffer in DDR4 and HBM is not fully utilized due to large S, bank-group-level parallelism allows 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 8d, 8g illustrate that when S increases from 128 to 2048, a bigger S can achieve higher memory throughput under "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 and DDR4.

6.5 Effect of Memory Access Locality

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 9 illustrates the throughput for varying parameter settings on both HBM, DDR4 and DDR3. We have two observations.

First, memory access locality indeed increases the memory throughput for each case with large S. For example, the memory bandwidth of the case (B=32, W=8K, and S=4K)

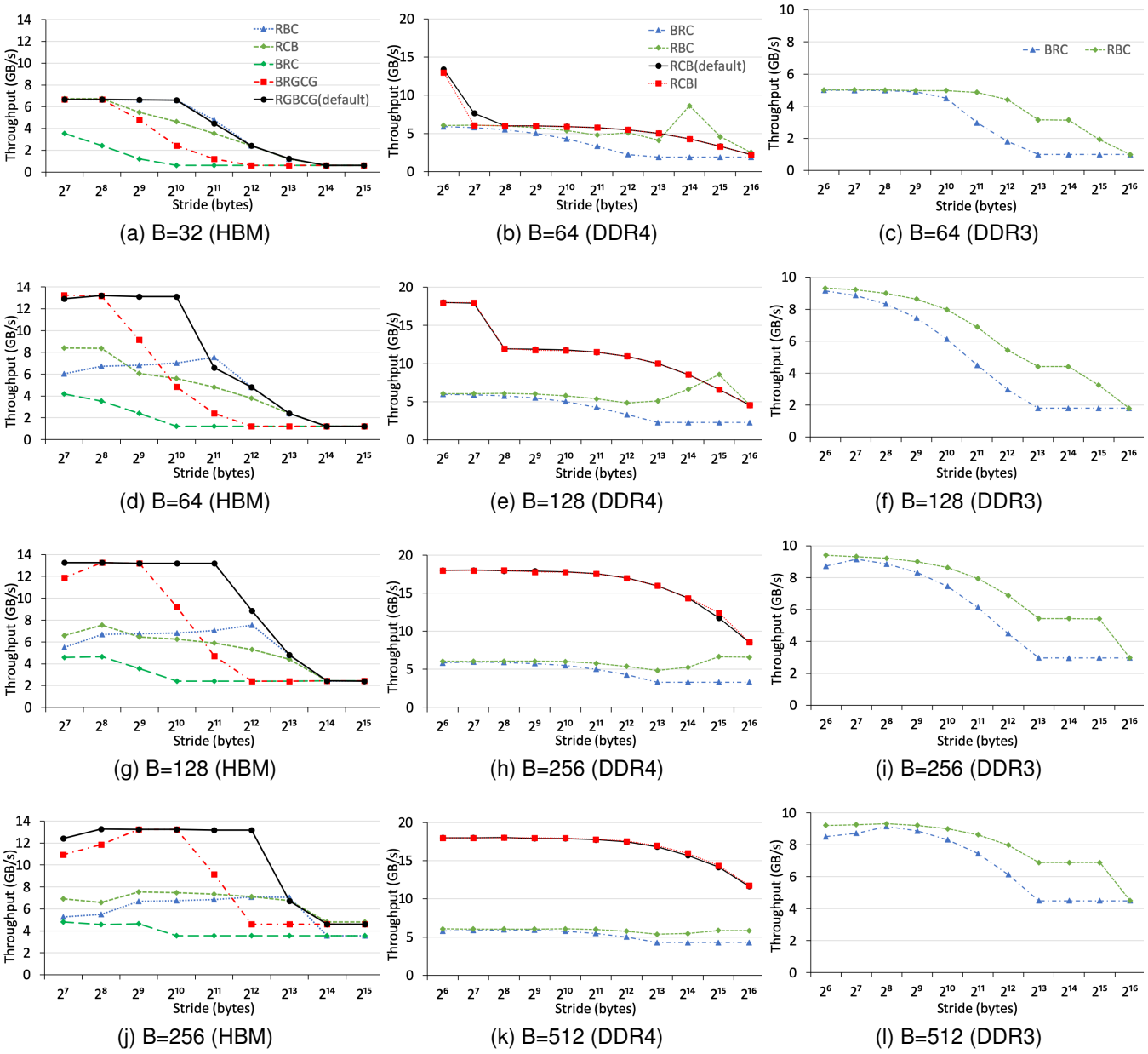

Fig. 8. Memory throughput comparison between an HBM channel, a DDR4 channel and a DDR3 channel, with different burst sizes and stride under all the address mapping policies. In this experiment, we use AXI channel 0 to access its associated HBM channel 0 for the best performance of a single HBM channel. We use the DDR4 channel 0 to obtain the DDR4 throughput numbers. We use the DDR3 channel 0 to obtain the DDR3 throughput numbers.

is 6.7 GB/s on HBM, while 2.4 GB/s of the case (B=32, W=256M, and S=4K), indicating that access locality is able to eliminate the negative effect of large S. Second, access locality cannot increase the memory throughput when S is small. In contrast, 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 [17].

6.6 Total Memory Throughput

We measure the total achievable memory throughput of HBM, DDR4, and DDR3 (Table 5). The HBM system 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.[12] 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. The DDR3 memory is able to provide up to 19 GB/s (9.5 GB/s * 2) memory throughput. We observe that the HBM system provides 10X higher memory throughput than DDR4, 22X more memory throughput than DDR3, indicating that the HBM-enhanced FPGA allows to accelerate memory-intensive applications, which are typically accelerated on GPUs.

12. 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.

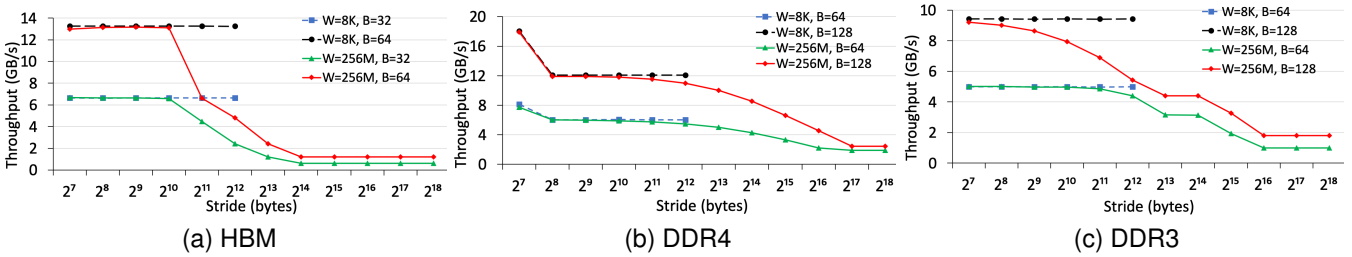

Fig. 9. Effect of memory access locality on HBM, DDR4, and DDR3.

TABLE 5
Total memory throughput comparison between HBM, DDR4 and DDR3.

	HBM	DDR4	DDR3
Throughput of a channel	13.27 GB/s	18 GB/s	9.5 GB/s
Number of channels	32	2	2
Total memory throughput	425 GB/s	36 GB/s	19 GB/s

6.7 Shuhai-Guided Optimization Process

We demonstrate the usefulness of Shuhai by illustrating the optimization process applied to matrix multiplication (MM). Suppose we have two operand matrices X and Y, whose size is $M \times M$. The result matrix Z is XY, whose size is also $M \times M$. All three matrices are stored row-wise. Each matrix occupies an HBM channel, such that there is no interference between each other. Each element is 32-bit. Next we illustrate how to leverage Shuhai to guide the optimization process.

We begin with the original implementation of MM. Each time, we calculate an element of matrix Z to be a dot product of a row of the matrix X and a column of the matrix Y. Since each matrix is stored row-wise, the memory access pattern (MAP) of matrices X and Z is sequential, leading to high throughput in their corresponding HBM channels. However, the achievable throughput of matrix Y is low, as its MAP is strided with $B = 4$, $S = 4 * M$, $W = 4 * M * M$. According to the previous benchmarking result in Figure 8, we can easily observe that its MAP is not memory-friendly. In our experiment, we always instantiate sufficient amount of compute units, which make the MM implementation memory bandwidth bound. As such, the overall performance is determined by the achievable memory bandwidth of the matrix Y.

Figure 10 compares the predicted and actual throughput trends of matrix Y with varying K and M, where the actual throughput is calculated to be the total memory access size ($4 * M * M * M/K$) divided by the elapsed time T, and the predicted throughput is measured via Shuhai on an HBM channel with the default AMP, as shown in Figures 8a, 8d, 8g, 8j. When B is smaller than 32, each time we still need to read out a memory transaction whose size 32 bytes. The predicted throughput trend roughly matches the actual throughput trend, indicating that we are able to leverage Shuhai to maximize the memory throughput of the matrix B that is bottleneck of the implementation of MM before implementing the entire MM code on the FPGA. Moreover, the actual throughput is slightly lower than the predicted throughput, because the MM calculation cannot fully overlap with external memory access.

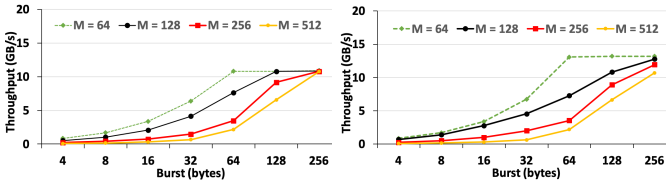

Fig. 10. Comparison of actual and predicted throughput trends of the matrix Y when performing matrix multiplication

7 Benchmarking the Switch

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 [4], [9]. Therefore, the 32 × 32 switch is required to make sure each AXI port is able to reach the whole address. However, the fully implemented 32 × 32 switch requires a massive amount of logic resources. Thus, the switch is only partially implemented, thereby consuming significantly fewer resources at the cost of lower performance for particular accessing patterns. Our goal here is to unveil the performance characteristics of the switch.

7.1 Between an AXI Channel and a HBM Channel

In the following, we examine the performance characteristics between any AXI channel and any HBM channel, in terms of latency and throughput.

7.1.1 Memory Latency

Read Latency. We measure the memory read latency from any AXI channel (from 0 to 31) to the HBM channel 0.[13] Access to other HBM channels has similar performance characteristics. We also employ the latency module to determine the accurate latency. Table 6 illustrates the latency difference among 32 AXI channels to the HBM channel 0. We have two observations. First, the latency difference is up to 22 cycles. For example, for a page hit transaction, an access from the AXI channel 31 needs 77 cycles, while the access from the AXI channel 0 only needs 55 cycles. Second, any AXI channel in the same mini-switch has identical access latency, demonstrating that the mini-switch is fully-implemented. For example, the AXI channels 4-7 have the same access latency. We conclude that an AXI channel should access its associated HBM channel or the HBM channels close to it to minimize latency.

13. The switch is enabled to allow global addressing, when comparing the latency difference among AXI channels, as illustrated in Table 7.

TABLE 6
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.

Chann.	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 7
Memory write latency from any of 32 AXI channels to the HBM channel 0. The switch is on.

Channels	Cycles	Time
0-3	15	33.3ns
4-7	17	37.8ns
8-11	19	42.2ns
12-15	21	46.7ns
16-19	32	71.1ns
20-23	34	75.6ns
24-27	36	80ns
28-31	38	84.4ns

Write Latency. We measure the HBM write latency from any AXI channel to the HBM channel 0, as shown in Figure 7. Particularly, we measure the number of clock cycles from issuing an AXI write operation using WA and WD channels to receiving a notification from the WR channel. We observe that the write latency is always stable from the same AXI channel, regardless of page hits, pages closed, and page misses. We can conclude that the write response is forwarded to the user logic after the write transaction reaches the HBM controller but before goes into HBM chips.

7.1.2 Memory Throughput

We employ the throughput module to measure memory throughput when any AXI channel (from 0 to 31) accesses the HBM channel 0, with the setting B = 64, W = 0x1000000, N = 200000, and varying S. Figure 11 illustrates the memory throughput from an AXI channel in each mini-switch. We observe that AXI channels are able to achieve roughly the same memory throughput, regardless of their locations.

7.2 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 8 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 three 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, a larger B leads to higher total throughput, under the same amount of active AXI channels, because a larger B reduces the number of switching times between AXI channels [4]. For example, two active channels 4 and 5 only have 6.3 GB/s total throughput when B=32, while

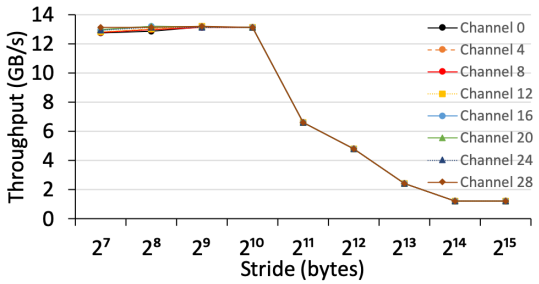

Fig. 11. Throughput from eight AXI channels to the HBM channel 1, where each AXI channel is from a mini-switch.

12.18 GB/s for B=128. Third, a lateral connection and four masters within a mini-switch are scheduled in a round-robin manner. Take the case with four active AXI channels 4, 5, 8, and 9 as an example: the remote channels 8 and 9 occupy the lateral connection of the mini-switch that consists of AXI channels 4-7. When B is 64, the throughput of an AXI channels 8 or 9 is 1.73 GB/s, so their total throughput is 3.46 GB/s, roughly the same as that (3.42 GB/s) of an AXI channel 4 or 5.

TABLE 8
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. The empty slot indicates the corresponding AXI channel is not involved.

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

8 Related Work

A preliminary version of this topic has been published in FCCM [18]. Compared with the preliminary version, this paper makes significant contributions in 1) benchmarking DDR3 on FPGAs and 2) benchmarking memory on CPUs/GPUs.

Benchmarking HBM on FPGAs. Recent work [19], [20] has analyzed the latency and throughput of HBMs on FPGAs when programmed using High Level Synthesis (HLS), which provides a higher level of abstraction and thus is widely adopted by software programmers. Similarly, [21] has measured HBM's throughput under various memory access pattern under Vitis to guide the optimization process of related applications. In contrast, Shuhai is implemented in Verilog, allowing more fine-grained and more precise measurements.

Benchmarking traditional memory on FPGAs. In addition to exploring HBM, there has been work [22], [23], [24] exploring traditional memory, e.g., DDR3, on the FPGA when using high-level languages such as OpenCL. In contrast, we benchmark HBM on an FPGA although Shuhai can also be used to benchmark conventional DDR.

Data processing with HBM/HMC. There is a growing number of research exploiting HBM/HMC for data pro-

cessing, as data processing is often memory bound. For instance, [25] accelerates a hash table by leveraging the HBM available in Intel Knights Landing [26], which is used to provide enough memory to a many-core architecture. Similarly, [27] does stream analytics on high bandwidth hybrid memory; [28] designs a heterogeneous memory hierarchy to exploit HBM bandwidth advantages; [29] accelerates a weather prediction modeling with a near high-bandwidth memory stencil; [30] designs a high-performance adaptive merge tree sorting; [31], [32] employ HBM to accelerate a number of data processing applications. All these efforts would benefit from how to benchmark the performance of HBM on an FPGA to optimize their designs. The need to better understand HBM has been shown in recent work [33] analyzing the impact of HBMs on tasks such as join and stochastic gradient descent as part of a study exploring how to make best use of FPGAs for data processing. [34] uses HBM to enable efficient embedding table lookups in memory-bound recommendation inference.

Accelerating applications with FPGAs. Previous work [35], [36], [37], [38], [39] accelerates a broad range of applications, e.g., database and deep learning inference, using FPGAs. We expect that many of these applications will improve their performance by using HBM as, often, these applications are memory bandwidth bound. The systematic benchmarking of HBM on Xilinx FPGAs provides the insights necessary to port such applications to boards with HBM optimizing the overall performance.

9 Conclusion

FPGAs are being equipped with High Bandwidth Memory (HBM) to tackle the memory bandwidth bottleneck in memory-bound applications. However, the performance characteristics of HBMs have been neither quantitatively nor systematically analyzed on FPGAs. We bridge this gap by benchmarking two HBM stacks on a state-of-the-art FPGA. 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 on CPUs/GPUs due the interference from the caches and TLBs. Shuhai can be easily generalized to other FPGA boards or other generations of memory modules, e.g., DDR3 and DDR4. Shuhai is open-source and will allow new FPGA boards and new memory types to be explored and quantitatively analyzed. The code is available: https://github.com/RC4ML/Shuhai.

Acknowledgements

We thank the Xilinx University Program for the generous donation of FPGA boards for our research and access to the XACC cluster at ETH Zurich. 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] Xilinx, *Alveo U280 Data Center Accelerator Card Data Sheet*, DS963, 2020, v1.3.

[2] R. Mueller, J. Teubner, and G. Alonso, "Data processing on fpgas," *VLDB*, p. 910–921, 2009.

[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[4] Xilinx, "AXI High Bandwidth Memory Controller v1.0," PG276, 2021, v1.0.

[5] 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.

[6] 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.

[7] 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.

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

[9] M. Wissolik, D. Zacher, A. Torza, and B. Day, "Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance," WP485, 2019.

[10] NVIDIA Corporation, "NVIDIA Turing Architecture Whitepaper," WP-09183-001, 2018, v01.

[11] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS*, 2010.

[12] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *TPDS*, 2017.

[13] T. Karnagel, T. Ben-Nun, M. Werner, D. Habich, and W. Lehner, "Big data causing big (tlb) problems: Taming random memory accesses on the gpu," in *DaMoN*, 2017.

[14] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 GPU via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[15] Xilinx, "AXI Reference Guide," UG761, 2012, v14.3.

[16] Arm, *AMBA AXI and ACE Protocol Specification*, IHI0022H, 2017, rev. F.

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

[18] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking high bandwidth memory on fpgas," in *FCCM*, 2020.

[19] Y.-k. Choi, Y. Chi, J. Wang, L. Guo, and J. Cong, "When hls meets fpga hbm: Benchmarking and bandwidth optimization," *arXiv preprint arXiv:2010.06075*, 2020.

[20] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking," in *FPGA*, 2021.

[21] R. Li, H. Huang, Z. Wang, Z. Shao, X. Liao, and H. Jin, "Optimizing memory performance of xilinx fpgas under vitis," *arXiv preprint arXiv:2010.08916*, 2020.

[22] H. R. Zohouri and S. Matsuoka, "The memory controller wall: Benchmarking the intel fpga sdk for opencl memory interface," in *H2RC*, 2019, pp. 11–18.

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

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

[25] 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.

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

[27] 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.

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

[29] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal, "Nero: A near high-bandwidth memory stencil accelerator for weather prediction modeling," in *FPL*, 2020.

[30] N. Samardzic, W. Qiao, V. Aggarwal, M. F. Chang, and J. Cong, "Bonsai: High-performance adaptive merge tree sorting," in *ISCA*, 2020.

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

[32] 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.

[33] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on fpgas: A data analytics perspective," in *FPL*, 2020.

[34] W. Jiang, Z. He, S. Zhang, T. B. Preußer, K. Zeng, L. Feng, J. Zhang, T. Liu, Y. Li, J. Zhou, C. Zhang, and G. Alonso, "Microrec: Efficient recommendation inference by hardware and data structure solutions," *MLSys*, vol. 3, 2021.

[35] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang, "Accelerating Generalized Linear Models with MLWeaving: A One-size-fits-all System for Any-precision Learning," *VLDB*, vol. 12, pp. 807–821, 2019.

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

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

[38] 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.

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