FlashGPU: Placing New Flash Next to GPU Cores

Jie Zhang^{1,3}, Miryeong Kwon^{1,3}, Hyojong Kim², Hyesoon Kim², Myoungsoo Jung³

Computer Architecture and Memory Systems Laboratory³

Yonsei University¹, Georgia Institute of Technology², Korea Advanced Institute of Science and Technology³

ABSTRACT

We propose FlashGPU, a new GPU architecture that tightly blends new flash (Z-NAND) with massive GPU cores. Specifically, we replace global memory with Z-NAND that exhibits ultra-low latency. We also architect a flash core to manage request dispatches and address translations underneath L2 cache banks of GPU cores. While Z-NAND is a hundred times faster than conventional 3D-stacked flash, its latency is still longer than DRAM. To address this shortcoming, we propose a dynamic page-placement and buffer manager in Z-NAND subsystems by being aware of bulk and parallel memory access characteristics of GPU applications, thereby offering high-throughput and low-energy consumption behaviors.

1 INTRODUCTION

Modern GPUs approach a terabyte per second bandwidth with massive high-throughput cores, which can make GPUs ideal for diverse data-intensive applications. Even though there is an engineering effort to increase embedded DRAM to accommodate the large data, the memory capacity is limited due to memory scaling problem, high cost/budget and high power consumption issues. To address this, most GPU technologies provide a method to address their limits by directly accessing its host memory [3] or putting data onto SSDs close to GPUs [2]. However, such unified memory techniques or SSD-linked GPU approaches yet introduce data movements and often need a host-side software assistance, such as file systems. For example, to access a large amount of data in storage, both unified memory and SSD-linked GPU require copying the data from the underlying SSD to the CPU's memory and moving them back again to GPU internal DRAM through a PCIe interface. This computing model varies data bandwidth and degrades the execution performance of target GPU applications.

While GPUs have innovated their architecture and infrastructure, flash also has a significant technology shift. One of the challenges in introducing flash into a GPU to address the aforementioned memory limit is millisecond-scale latency for memory accesses. Specifically, new flash memory such as Z-NAND offers 17× and 6× faster access speeds for reads and writes of a conventional 3D-stacked flash with hundred times longer life cycles [10]. Specifically, Z-NAND redesigns DDR interface and peripherals of I/O circuitry by reorganizing single-level cell (SLC) based vertically-stacked flash

[8], which in turn can exhibit 3 us and 100 us memory access time for 2KB ~ 4KB-sized page reads and writes, respectively.

Even though this *u*-scale latency of Z-NAND is longer than DRAM, we observe that bulk and parallel memory access behaviors of GPU applications can make Z-NAND feasible to be blended with massive GPU cores, referred streaming multiprocessors (SMs). In this work, we propose *FlashGPU*, a new GPU architecture that places new flash next to GPU cores and tightly couple them within a single GPU device. We replace GPU global memory with Z-NAND and architect a flash core to manage request dispatches and address translations underneath L2 cache banks of SMs. Unlike traditional storage, our flash core communicates with GPU cores through DRAM in/out queues and GPU's Miss Status/Information Holding Registers (MSHR). This design allows all SMs to access many Z-NANDs without any source-level change of GPU applications as well as to directly execute GPU applications on new flash, which can remove the overheads of data movement between a GPU and an SSD. Compared to a traditional approach that puts an SSD near a GPU, FlashGPU improves IPC by 263% while consuming energy 71% less than the SSD+GPU approach, on average. The main contributions of this paper can be summarized as follows:

- *Architecting GPU with new flash*: In our architecture, L2 cache banks communicate with our flash core over a ring-based interconnection network. The flash core employs a small cache, referred to as flash cache, and it contains multiple DMA and flash controllers per memory channel, which handles data movement between the flash cache and the underlying flash, and manages all processes of flash asynchronous protocol, respectively. Another challenge in integrating flash into GPU cores is reliability. We observe that L2 cache misses are frequently written to a same target page of Z-NAND. Thus, the flash cache also places a small write buffer in front of the underlying Z-NAND dies, which can reduce write amplification by 28K times, compared to our baseline.
- *GPU-aware dynamic page placement*: We analyze multiple real GPU workloads and observe that, while there exists significant page-level temporal locality, the 128-byte granule memory requests of L2 caches seriously degrade performance as well as underutilize Z-NAND resources. A way to improve the Z-NAND utilization is to implement a super-page based I/O processing [4], but its benefits vary based on the access patterns of GPU active warps. Thus, we propose GPU-aware dynamic page placement, which places either a normal page or super-page into a read buffer. To this end, we introduce an access pattern tracker that identifies different warp and tracks the request pattern of each warp to decide the granularity of page placement at runtime. The results show that our placement scheme improves the Z-NAND utilization by 70% thereby improving performance by 37%, compared to the baseline.
- *Flash-integrated GPU model*: While simulation-based studies for GPU are widely performed, there is no model that integrates flash into a GPU. We expand MacSim by fusing SimpleSSD, which are a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6725-7/19/06...\$15.00

<https://doi.org/10.1145/3316781.3317827>

Figure 1: GPU architecture, Z-NAND internal and SSD firmware.

3D NAND	BiCS	V-NAND	Z-NAND
# layer	48	64	48
t_R	45 μ s	60 μ s	3 μ s
t_{PROG}	660 μ s	700 μ s	100 μ s
Capacity	256Gb	512Gb	64Gb
Page Size	16KB	16KB	2KB
P/E cycles	3,000	7,000	100,000

Figure 2: Comparisons of 3D flash.

Figure 3: Data movement and execution latency analysis.

high fidelity, hardware-validated GPU and SSD simulation models. In this extension, the L2 cache generates a memory packet and send it to the flash core through ring-network rather than using conventional memory interface. MSHR handles miss-under-miss by keeping information of source and destination blocks and address stack, including target's core instruction ID. In addition, each router of ring-network contains multiple queues (e.g., fill, in/out queues, etc.) thereby tolerating the variation of latency of flash core.

2 BACKGROUND

2.1 Baseline GPU Architecture

Figure 1a illustrates the baseline GPU that we consider integrating Z-NAND into. In this architecture, massive GPU cores are grouped as multiple *streaming multiprocessors* (SMs). SMs carries out a task by running multiple threads blocks in highly-parallel manner; each block consists of multiple *warps*, which contains 32 kernel threads. Each SM employs its private L1 cache, and all miss requests are forwarded to shared level-level (L2) caches via an interconnection network. If the target data do not reside in L2, it is served from memory controllers over the network again [7]. Each endpoint of the network has a router that maintains read (fill), write (input), output queues in front of the underlying storage media.

The L1 and L2 caches in this design contain *miss status/information holding registers* (MSHR) structure, which tracks outstanding memory requests. In the case of L1 cache misses, the load (LD) and store (ST) units of each SM coalesce the memory requests, and each coalesced requests books an entry of MSHR. MSHR also maintains memory address offsets and thread identifiers. Thus, if a response of an issued request is received from the lower level memory, the corresponding MSHR identifier and requested data are fetched from the response. The LD/ST units then fill the data into private registers of the target thread. The MSHR management is also applied to the L2 caches, which in turn makes the role of L2 MSHR similar to a pending request table structure in our baseline architecture.

2.2 New Flash

While flash can offer a larger memory capacity than DRAM, it is mostly impossible to directly replace DRAM due to its long access latency. However, new flash, referred to as Z-NAND, provides a comparably promising latency, which reconsider many architects to bring it close to computational resources. Z-NAND leverages vertically-stacked NAND flash (V-NAND) (as shown in Figure 1b), but Z-NAND tailors the access latency by reorganizing its low-level cell technology and micro-architecture. In addition, Z-NAND leverages the single-level cell (SLC) technologies, which not only shorten the latency (than conventional multi-level or triple-level cells) but also significantly increase program/erase (P/E) cycles. Z-NAND further reduces the page size and number of stacked layers with an optimization of I/O circuitries like DDR interface [8].

The aforementioned engineering efforts, in turn, make the read and write operations of Z-NAND $17 \times$ and $6 \times$ faster than those of V-NAND and bit-cost scaling type 3D NAND (BiCS, another type of 3D NAND), respectively. As shown in Figure 2, Z-NAND adopts 48 layers, which are 25% lesser than V-NAND's 64 stacked layers; the default size of Z-NAND is 2KB, which is a half of V-NAND page size. Thanks to these optimizations and design strategy, the latency of reads (t_R) and writes (t_{PROG}) of 64Gb Z-NAND is 3 us and 100 us, respectively. Considering massive parallelism of a modern multi-channel and way SSD architecture [4], Z-NAND is capable of bringing out a large amount of the data within a very short period. As shown in the table, the P/E cycles of Z-NAND are also longer than BiCS and V-NAND by $33 \times$ and $14 \times$, respectively.

2.3 Motivation

Figure 3a illustrates an overview of a heterogeneous system, which places an SSD close to a GPU into a single computing system. In this architecture, the data should be loaded or stored from or to the underlying SSD. The host side storage stack and GPU software framework are involved in moving the target data between the SSD and GPU, which significantly waste host CPU cycles. In addition, it requires copying data multiple times among the physical memories of the SSD, GPU and host system, as well as redundant data copies in the host side due to user/privilege mode switches [14, 15].

To be precise, we evaluate such system by putting a GPU and Z-SSD prototype that employs 16 Z-NAND packages [8] together into a real system, and the results are shown in Figure 3b. One can observe from this execution time breakdown analysis that the latency of Z-SSD accounts for only 14% of the total execution time, on average, while the data movement between the host and GPU

Figure 4: Locality on reads, write amplification, and flash resource utilization analyses.

(host $\leftrightarrow$ GPU) and the Z-SSD and host (Z-SSD $\leftrightarrow$ host) takes 20% and 41% of total execution. These data movement overheads take latency longer than GPU execution times by 144%, on average.

3 HIGH-LEVEL VIEW

To address the overheads described in Section 2.3, FlashGPU allows warp executions to LD and ST within the target GPU platform by placing new flash next to SMs. This section will describe our observations and technical challenges to tightly couple the SMs with Z-NAND. Then, we will explain the proposed GPU architecture.

3.1 Observations and Challenges

Locality on reads. We first characterize the performance degradation imposed by directly replacing the global memory with Z-NAND. In this simulation-based study, we configure the target GPU similar to GTX 580 by using MacSim [7]; important configurations are listed in Table 1. The results are shown in Figure 4a. One can observe that the performance can degrade by 291 $\times$ compared to the baseline architecture. The main reason for this degradation is that the memory requests missed from L2 cache are coalesced in the form of 128 bytes packet, which are discrepant distance to the I/O access unit of Z-NAND. Even though there is an issue with such asymmetric I/O access granularity, as shown in the figure, the tested GPU workloads exhibit many read requests that repeatedly access the same pages. Motivated by this, FlashGPU employs a small-size read buffer, which considers the high flash page-level locality on reads thereby removing the performance degradation.

Write amplification. Another challenge that we should address is reliability of Z-NAND. While Z-NAND exhibits 33 $\times$ higher P/E cycles than conventional flash, we observed that average write-amplification values (WAV) exhibited by the aforementioned GPU workloads are 28,219; in this work, WAV is calculated by dividing the total write data in flash over the volume of all data modified by GPU applications. This high WAV can make Z-NAND’s lifetime and reliability shorter and lower, respectively. Fortunately, we observed (from our workload analysis) that most writes generated by GPU applications are related to update the results and frequently stored to the same page of the underlying memory. Thus, we introduce a small-size write buffer in front of the Z-NAND arrays, which can significantly reduce WAV.

Resource underutilization. While the write and read buffers can shorten the latency of Z-NAND, we observed the bandwidth of the underlying flash subsystems runs far behind that of the conventional memory. Even though the advantage of massive internal parallelism is essential to achieve high bandwidth [6], as shown in Figure 4c, only 1~4 flash dies (among 128 dies) are active under

Figure 5: Overview of FlashGPU.

Figure 6: Integrated architecture of FlashGPU.

executions of all the workloads that we examined. This low flash parallelism and utilization may increase by fetching data with a large size of a group page, called *super-page*. However, performing I/O with the granularity of super-page can waste our buffer space (i.e., cache pollution). Thus, we introduce an access pattern tracker, which is aware of GPU application behaviors and selectively scheduling requests based on the super-page.

3.2 FlashGPU Overview

Figures 5 and 6 show an overview of FlashGPU internal component and integration architecture, respectively. FlashGPU employs a *flash cache* that handles a write buffer and a read buffer in front of Z-NAND arrays. This cache communicates with our access pattern tracker being aware of each warp execution behavior based on its program counter (PC) so that it can decide whether FlashGPU needs to fetch target data in a super-page scheduling manner. In addition, we allocate a few ALU logic, referred to as *flash core*, which installs most flash firmware components, including host interface layer (HIL), flash translation layer (FTL) and flash interface layer (FIL). HIL handles the requests by communicating with a router, which is assigned to the flash core, while FTL and FIL manage the address translation and flash protocol/timing, respectively. The flash core also implements our flash cache; in this work, we configure the write and read buffers by allocating 1 MB and 32 MB memory.

As shown in Figure 6, in the case of L1 cache misses, the memory requests are forwarded to shared L2 cache via the interconnect network (①). The memory request is firstly staged in the input queue of L2 cache and processed by the L2 cache afterwards (②). For L2 cache misses, the memory requests are inserted in the output queue (③), and an entry in MSHR is registered by updating cache line address, memory address and thread ID (④). The requests in the output queue are dispatched, converted to a network packet with the information of source endpoint ID and destination endpoint ID, and forwarded to the flash core over the network (⑤). When the packet arrives, the flash core’s interface controller dispatches such packet from the flash-side router. Afterwards, HIL re-constructs the memory requests by extracting the information of memory address,

Figure 7: Our flash cache and buffers.

access type and length from the target packet (⑨). The flash cache then checks if the request hits in its read buffer or write buffer. If there is a miss, the request should be served by Z-NAND. FTL translates the target memory address to a physical block address while FIL manages flash protocol and memory transaction (⑦). The response of the memory request (with data) then is forwarded to L2 cache through the network again (⑧). In L2 cache, the response is inserted in the fill queue (⑨). L2 cache extracts the target data and memory address from the response residing on the fill queue. The memory address is used for identifying and releasing the entry of MSHR (⑩), and then the fetched data is stored in L2 cache (⑪). This process is also applied to L1 cache so that stalled LD/ST instructions can be resumed from the target SM.

4 IMPLEMENTATION

4.1 Write and Read Buffer Details

Figure 7 shows the details of read and write buffers in FlashGPU. Both read and write buffers are configured as set-associative, which keep the hottest pages and evicts the coldest pages in an effective manner. However, accessing both the tag array (TA) and data from the buffers can increase the traffic of accessing the same buffer by up to three times. As the TA sizes of 32MB read buffer and 1MB write buffer are only 64KB and 2KB, we place the TAs in SRAM space for fast accesses. To reduce iterations of tag lookup, we compact each tag entry into 8B, of which the details are shown in Figure 7. For the write buffer, each TA entry consists of 29-bit tag, 3-bit LRU and 32-bit valid bits, one bit for the valid status of each 128B segment. When evicting a dirty page, only the segments that are valid will be written to Z-NAND. For the read buffer, each TA entry is comprised of 29-bit tag, 3-bit LRU and 1-bit valid bit. We also set the associativity of the read/write buffer as 8 ways, which makes the eight tag entries per cache set align with a single cache line, and it only costs flash core single SRAM access for every request.

Figure 8 shows the execution flow to coordinate the read and write requests in FlashGPU. For a read request, the flash core firstly looks up the TA of the write buffer. If the tag of the request matches with such TA, the write buffer directly serves data as the response, since the write buffer always keeps the latest version of data, which are evicted from shared L2. Otherwise, the flash core checks if the target data is cached in the read buffer. If there is a miss, the read buffer invalidates a stale page, fetches the new page from Z-NAND and responds the requests. On the other hand, as shown in Figure 8b, for writes, the write buffer immediately responds an incoming request, but if the write buffer is full, a stale page is evicted to Z-NAND. The flash core also checks the read buffer to invalidate the target to manage the consistency of buffers.

Figure 8: Execution flow.

Figure 9: Access pattern tracker.

Figure 10: Memory request analysis of workload bfs1.

4.2 Dynamic Page Placement

It is desired to fetch multiple sequential pages across all different Z-NAND in parallel, called super-page scheduling (in order to improve flash resource utilization). Since a kernel in practice unrolls its loop and assigns a thread to execute a single iteration, one can expect that there exist many sequential memory requests. However, we observed that most workloads access our flash core in a full random manner. Figure 10a shows logical page numbers (LPNs) of the memory requests of a representative workload, *bfs1*. The most memory requests randomly access different LPNs from the viewpoint of our flash core. This is because SMs usually co-run thousands of threads in parallel; the threads generate memory requests simultaneously, which make the access patterns at the flash core out of order. However, if we see the access patterns exhibited by warp executions at an SM-level, like what we expected, there are multiple warps that generate memory requests in contiguous. We further analyze the memory requests based on the PC address and illustrate LPNs, belonging to two different PCs of *bfs1* execution, in Figure 10b. One can observe from the figure that, while the executions of PC=8032 exhibit random access pattern, those of PC=8408 access the pages in a sequential manner. Since the super-page scheduling for random accesses results in wasting the bandwidth of Z-NAND and polluting our read buffer, we distinguish different warp executions based on PCs and treat them in a different way.

The key insight being aware of GPU execution context is that a warp (or threads), executing LD/ST instructions from a same PC, generate memory requests that exhibit the same access pattern. Motivated by this, we propose an *access pattern tracker*, which tracks the accesses of read requests and predicts the memory access patterns based on each thread's PC address. Figure 9 depicts the hardware design of our access pattern tracker. We add the field of PC address and warp ID into each memory request. For reads, our flash core inquires the access pattern tracker for such information, which is used for speculating the request pattern. Specifically, our access pattern tracker maintains a prediction table, which is indexed by PC addresses. In its each row, the access pattern tracker stores a 2-bit global counter and information of four sampled warps, including a

Figure 11: The simulation model of FlashGPU.

logical super-page number (LSN) of the latest request (came from the sampled warps) and a n -bit local counter, respectively, denoted by *Last Addr* and *Cout* in the figure. The n -bit is configured based on the number of flash pages grouped by a super-page; in this work, FlashGPU contains 16 Z-NAND packages, thus we use 4-bit counter. If target warp ID of the incoming request matches with the sampled warp ID, our flash core uses the PC address and warp ID as index in accessing the prediction table. The request’s LSN is compared to *Last Addr*, and the local counter increases by 1 if it is matched. Otherwise, the local counter decreases by 1. Since we predict the access pattern based a group of sampled warps, the prediction table also includes a k -bit global counter (*G-Cout* in each table entry, where k means the exponent of two to cover the number of sampled warp; in our case, it is 2. When the corresponding *Cout* is saturated, *G-Cout* increases, indicates if one observes a high number in *G-Cout*, it detects a sequential pattern for the target warp. Thus, the target *G-Sout* is saturated, the flash cache schedules super-page; otherwise, it fetches data for individual flash page.

4.3 Open-Source Simulation Model

Since there is no research framework that support both flash and SM executions, we implement our model by revising MacSim [7] and SimpleSSD [5].¹ As shown in Figure 11, the key GPU parameters such as the clock frequency of CPU, L1, L2 and network, cache line size, and associativity. The GPU cores (SM) inputs the trace files, which will be provided as a part of our open source framework as well as can be extracted by Ocelot. The memory requests missed from L2 is forwarded to HIL (in SimpleSSD) over network-on-chip (NoC) model as a form of packet. During this process, the input/output queues are also simulated for all endpoints, including caches and Z-SSD in the NoC model side. HIL parses the network packet and communicates the buffer manager that implement. The request is passed through other flash related software, such as FTL and FIL, and then returned back to the NoC model by calculating end-to-end latency of each I/O accesses. In our model, caching all memory requests are performed in MacSim side, while memory requests (missed from the caches) are managed by SimpleSSD.

5 EVALUATION

5.1 Methodologies

In this evaluation, we first implement a baseline GPU-SSD heterogeneous architecture by attaching discrete GPU and Z-SSD devices to the host (*HETERO*). We then build up several Z-NAND integrated FlashGPU architectures, including *FIGPU* (without read/write buffers), *FIGPU-RD* (with a read buffer), *FIGPU-RW* (with both a read buffer

GPU		flash core		DRAM	
SM/freq.	16/1.2 GHz	Processor	ARM, 4-core	tRCD/tRP/tCL	8.9/8.9/8.9ns
Max warps	80 per core	flash cache	512KB	Interface	LPDDR4
L1 cache	Z-NAND array			Write buffer	
	1-cycle, 64-set	Ch/Package	16/1	Capacity	1MB
	6-way, 48KB	Die/Plane	8/8	Set/Way	32/8
	LRU, private				
L2 cache	10-cycle, 6 banks	Block/Page	1024/384	Read buffer	
	128-set, 8-way	Interface	800MT/s	Capacity	32MB
	768KB, shared	Cell type	SLC	Set/Way	1024/8

Table 1: The system configuration of FlashGPU.

Figure 12: Performance analysis of FlashGPU.

Figure 13: Energy consumption of FlashGPU.

and a write buffer), and *FIGPU-DP* (*FIGPU-RW* with dynamic data placement). We use our proposed simulator (cf. Section 4.3) to model a GPU similar to NVIDIA GTX580. In the simulator, we also create a flash core, a flash cache and DRAM-based buffers. As the Z-NAND specification is unfortunately not in a public domain yet, we characterize Z-NAND with a blackbox test of our Z-SSD 800GB engineering samples. The important parameters of our FlashGPU are listed in Table 1. We evaluate the performance of different GPU-SSD architectures, in terms of IPC and energy consumption, with 15 different workloads from GraphBIB benchmark [9].

5.2 Performance and Energy Efficiency

Performance. Figure 12 shows the IPC values of five different GPU-SSD architectures, normalized to those with *HETERO*. While *HETERO* deploys a large GPU internal DRAM to accommodate all the working set for fast accesses, it performs worse than *FIGPU-RW* and *FIGPU-DP*, which only employ small read and write buffers. This is because *HETERO* needs to manipulate the data copy from the SSD to GPU before the kernel starts its execution, which imposes long latency. In contrast, FlashGPU fully eliminates the data movement overheads between the GPU and SSD by tightly integrating Z-NAND in the GPU. However, *FIGPU* performs worst among all four FlashGPU configurations, as it needs to access Z-NAND for every memory request. *FIGPU-RD* outperforms *HETERO* and *FIGPU* by $2.5 \times$ and $310 \times$ in read-intensive workloads *bfs2*, *bfs5*, *bfs6* and *degree*, since *FIGPU-RD* can benefit from the temporal locality on reads by caching read-intensive pages in our read buffer. As the write buffer can significantly reduce the latency of Z-NAND writes, *FIGPU-RW* achieves $14 \times$ higher performance than *FIGPU-RD* in write-intensive workloads *bfs1*, *conn*, *gc1* and *sssp1*. Compared to *FIGPU-RW*, *FIGPU-DP* improves the performance by 32%, on

¹All source codes of the simulator we revised, including GPU and flash models, will be available to download from <http://flashgpu.camelab.org>.

Figure 14: Detailed analysis of Z-NAND performance.

average, in the workloads *bfs3*, *kcore* and *sssp1*. This is because *FIGPU-DP* can load multiple sequential pages from Z-NAND to the read buffer before SMs start to read the data, which can avoid the relatively long delay of Z-NAND reads (compared to DRAM) for the outstanding misses in the read buffer.

Energy efficiency. Figure 13 plots the energy decomposition for all the hardware components of GPU-SSD heterogeneous architectures we evaluated. As shown in the figure, the energy consumption of software stack accounts for 56% of total energy usage in *HETERO*, making *HETERO* more energy-hungry than *FIGPU-DP*. *FIGPU* consumes highest energy among all the evaluated GPU-SSD architectures, due to the intensive NAND flash read and write operations. *FIGPU-RD* and *FIGPU-RW* reduce Z-NAND access energy by 83% and 97%, respectively, compared to *FIGPU*, as most read and write requests can be served from low-power read and write buffers rather than accessing the underlying NAND flash. This, in turn, makes *FIGPU-RD* and *FIGPU-RW* reduce the total energy consumption by 78% and 94%, compared to *FIGPU*. Lastly, as dynamic data placement technique can reduce the time spent for the NAND flash read operations by paralleling multiple page accesses, it can reduce the static power by 24%, on average, in workloads *bfs3*, *kcore* and *sssp1*.

5.3 Effectiveness of Read/Write Buffers

Z-NAND access overhead. Figure 14a shows the ratio of Z-NAND access time over the total execution time for different GPU-SSD architectures. The Z-NAND latency of *FIGPU* accounts for 99% of its total execution time. This is because, for each outstanding L2 cache misses, the SMs in *FIGPU* are stalled by the Z-NAND accesses. *FIGPU-RD* reduces the Z-NAND access overhead by 22%, compared to *FIGPU*, as it places hot pages in the read buffer for fast response. *FIGPU-RW* reduces the flash access overhead further, as it can accumulate the 128B write requests in the write buffer to reduce the unnecessary write amplification.

Average parallel die accesses. Figure 14b shows the average utilization of NAND flash internal parallelism in terms of the active dies in parallel. As shown in the figure, in workloads *bfs3*, *kcore* and *sssp1*, which have high ratio of sequential accesses, *FIGPU-DP* fetches multiple pages in parallel, which increases the average utilization by 70%, compared to *FIGPU-RW*.

6 RELATED WORK

Several studies have been proposed in addressing the data movement between GPU and SSD [1, 14]. NVMMU [14] puts a runtime library or kernel-level component that directly forward the input or output data between those two devices, which can reduce the number of data copies among different levels of DRAM. While NVMMU is a software-based approach, DCS [1] implements such

model in a hardware with an FPGA acceleration. Even though these studies address the overheads brought by the data migration between different devices, the GPU memory size is still limited by the scalability of current DRAM technologies, which require moving the data between the GPU and SSD frequently. In addition, they all need a file system or operating system to maintain the system consistency and coherency. There are more aggressive approaches that in part process data in a discrete accelerator [15] or SSD [11]. These methods can remove the needs of OS assistances at some extent, but they do not target to integrating storage with massive computing of GPU and the data-intensive applications therein, which is one of the main goals that FlashGPU is designed forwards. On the other hand, several studies to replace DRAM with emerging NVM [12] or to co-place computing logic with NVMs at very-low level [13]. Unlike FlashGPU, these low device-level approaches consider mainly byte-addressable NVM rather than flash, and do not explore the direction of massive kernel thread executions.

7 CONCLUSION

We proposed FlashGPU, a new architecture that co-place multiple GPU cores and Z-NAND in a single device platform. We analyzed multiple challenges and revealed observations in such tight integration approach. To address the observed challenges, the proposed FlashGPU integrated Z-NAND on the network-on-chip of GPU cores and replaced the global memory by being aware of the memory locality of GPU applications.

8 ACKNOWLEDGEMENTS

This research is mainly supported by NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF 2015M3C4A7065645, NRF-2017R1A4A1015498 and MemRay grant. The authors thank Samsung for its Z-SSD sample donations.

REFERENCES

- [1] Jaehyung Ahn et al. 2015. DCS: a fast and scalable device-centric server architecture. In *MICRO*. ACM.
- [2] AMD. 2017. Radeon Pro SSG Graphics. <https://www.amd.com/en/products/professional-graphics/radeon-pro-ssg>. (2017).
- [3] Mark Harris. 2013. Unified Memory in CUDA 6. <https://devblogs.nvidia.com/unified-memory-in-cuda-6/>. (2013).
- [4] Myoungsoo Jung et al. 2012. Physically addressed queueing (PAQ): improving parallelism in solid state disks. In *SIGARCH Computer Architecture News*. IEEE.
- [5] Myoungsoo Jung et al. 2018. SimpleSSD: modeling solid state drives for holistic system simulation. *Computer Architecture Letters* (2018).
- [6] Myoungsoo Jung and Mahmut T Kandemir. 2014. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 524–535.
- [7] Hyesoon Kim et al. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology* (2012).
- [8] Sungjoon Koh et al. 2018. Exploring system challenges of ultra-low latency solid state drives. In *HotStorage 18*.
- [9] Lifeng Nai et al. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC*. IEEE.
- [10] Samsung. 2017. Ultra-Low Latency with Samsung Z-NAND SSD. *Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf*. (2017).
- [11] Sudharsan Seshadri et al. 2014. Willow: A User-Programmable SSD. In *OSDI*.
- [12] Mimi Xie, et al. 2018. AIM: Fast and energy-efficient AES in-memory implementation for emerging non-volatile main memory. In *DATE*. IEEE.
- [13] Yuan Xue et al. 2017. Age-aware logic and memory co-placement for RRAM-FPGAs. In *DAC*. ACM, 1.
- [14] Jie Zhang et al. 2015. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *PACT*. IEEE.
- [15] Jie Zhang and Myoungsoo Jung. 2018. Flashabacus: a self-governing flash-based accelerator for low-power systems. In *EuroSys*. ACM, 15.