GPUKV: An Integrated Framework with KVSSD and GPU Through P2P Communication Support

Min-Gyo Jung[1], Chang-Gyu Lee[1], Donggyu Park[1], Sungyong Park[1], Jungki Noh[2]
Woosuk Chung[2], Kyoung Park[2], Youngjae Kim[1,*]
[1]Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
[2]SK hyinx
{mingyoj, changgyu, dgpark, parksy, youkim}@sogang.ac.kr
{jungki.noh, woosuk.chung, kyoung.park}@sk.com

ABSTRACT

When data is loaded from a key-value store to the GPU in a conventional GPU-driven computing model, it entails the overhead of all the heavy I/O stacks of the key-value store and file system. This paper presents GPUKV, a GPU-driven computing framework that eliminates the aforementioned overhead with less host-side usage of resources such as CPU and memory. GPUKV has the following three features: (i) GPUKV provides a key-value store abstraction to the GPU; (ii) In GPUKV, when loading data from the key-value store to the GPU, it is performed through PCIe peer-to-peer (P2P) communication without copying to the user and kernel space memory; and (iii) GPUKV uses KVSSD, which implements a key-value store inside an SSD, completely eliminating the interaction with the key-value store and file system for P2P communication. We have developed GPUKV with a KVSSD implemented on the Cosmos+ OpenSSD platform in a Linux environment. Our extensive evaluations demonstrate that GPUKV improves execution time by up to 18.7 times and reduces host CPU cycle usage by up to 175 times compared to conventional CPU-based GPU computing models.

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**; • **Computing methodologies → Graphics processors**; • **Software and its engineering →** *Data flow architectures*; *Operating systems*;

KEYWORDS

Key-Value SSD, GPGPU, Peer-to-Peer Communication

ACM Reference Format:
Min-Gyo Jung[1], Chang-Gyu Lee[1], Donggyu Park[1], Sungyong Park[1], Jungki Noh[2] and Woosuk Chung[2], Kyoung Park[2], Youngjae Kim[1,]. 2021. GPUKV: An Integrated Framework with KVSSD and GPU Through P2P Communication Support. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3441990

*Y. Kim is the corresponding author.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

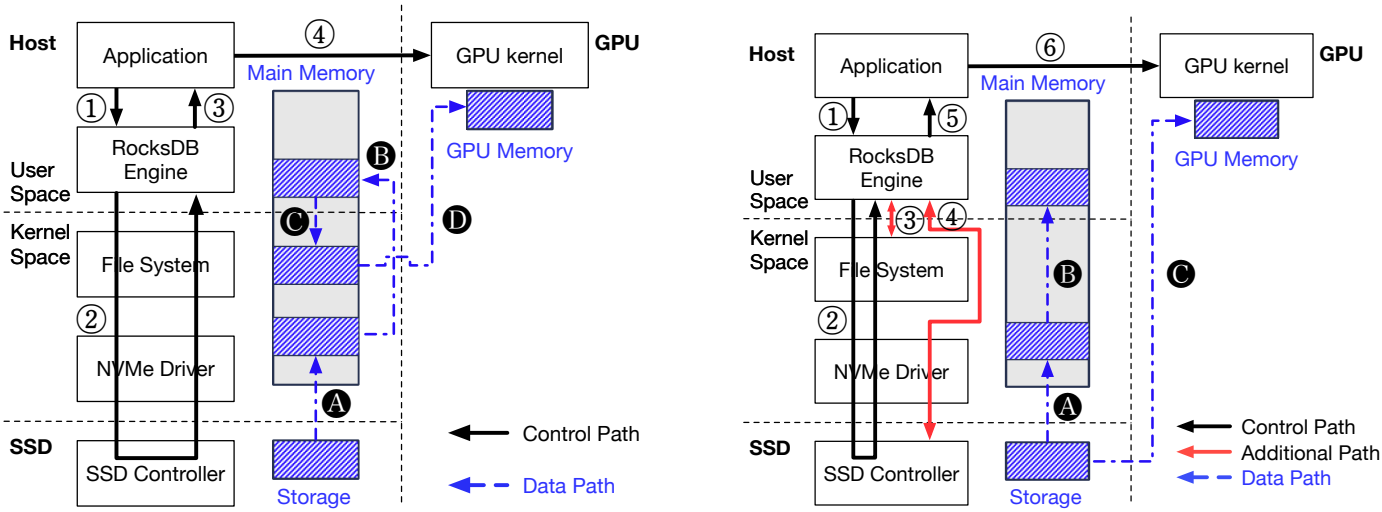
ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3441990

1 INTRODUCTION

The massively parallel processing of a GPU accelerates the processing speed of data-intensive applications in various areas such as deep learning, simulation and graph processing. Data processing using GPUs consists of the following steps [21]: First, the host application loads input data from storage devices to user space memory via a file system. Then it copies the loaded input data from user space memory to GPU memory. After that, when input data is prepared in GPU memory, the host application can start executing the GPU kernel. Thus, the host application controls the execution process of the GPU kernel in a conventional GPU programming model. This approach is called a *CPU-driven GPU computing model* [21]. However, CPU-driven GPU computing models have the following problems: First, in data-intensive applications, loading a large amount of input data from storage devices to GPU can create an I/O bottleneck due to excessive memory copy operations by the CPU. Second, a significant amount of host-side resources such as CPU and DRAM can be overwhelmingly used to load data from storage devices into GPU memory.

Several studies such as SPIN [1], NVMMU [30], Morpheus [27], and HippogriffDB [13] have used PCIe peer-to-peer communication (P2P) to minimize the overhead caused by data movement from storage devices to GPU. P2P allows two PCIe devices to transfer data directly to each other without using host-side resources. NVIDIA and AMD GPUs support API for PCIe P2P communication [17, 25]. Such studies using P2P [1, 27, 30] eliminated the memory copy overhead when transferring data to the GPU. However, in order to perform P2P, the block address of the file must be known beforehand. This file-to-block address translation involves interaction with the file system, consuming host-side CPU cycles. Even worse, when input data used by the GPU kernel is stored in a key-value store on the file system, P2P may not provide any performance gains (Refer to Section 2.2). This is because the overhead of heavy I/O stacks from the key-value store and file system can overwhelm the benefits of reduced data movement overhead by P2P.

For instance, consider a graph analysis application that uses a database to extract graph data and runs a kernel on the GPU. In order to feed graph data directly from storage devices to the GPU using P2P, the application first needs to find database files and file offsets for the graph data. Then the application must consult the file system to find the logical block addresses (LBAs) of the files. Then, using this LBA information, the application makes a request to



(a) CPU-driven GPU computing model w/o P2P

(b) CPU-driven GPU computing model w/ P2P

Figure 1: Conventional data transfer flow from key-value store (RocksDB) to GPU

storage devices and initiate direct data transfer to the GPU via P2P. In CPU-driven GPU computing models, CPU intervention for the database file search and file-to-block address translation described above is inevitable for P2P.

To solve the aforementioned problems, in this paper, we propose GPUKV, a *GPU-driven computing* framework where the GPU kernel can initiate I/O requests while executing the kernel, directly obtain input data from storage devices, and process them. GPUKV is implemented on object storage devices such as a Key-Value SSD (KVSSD) [3, 6–8, 11, 23, 24, 26, 28]. KVSSD implements a key-value store inside an SSD. Therefore, there is no need to interact with the key-value store and the file system for file-to-LBA address translation for P2P in the host. Therefore, it eliminates the use of host resources. P2P also allows us to directly transfer data from KVSSD to GPU without CPU intervention in the host.

This paper makes the following specific contributions:

- **Key-Value Store Abstraction to GPU:** In a conventional GPU-driven programming model, the CPU had to manually supply input data for the execution of the GPU kernel using cudaMemcpy. However, GPUKV provides key-value store abstraction and I/O programming APIs to GPU kernels to send key-value requests to KVSSD when executing GPU kernels.
- **Key-Value SSD:** GPUKV adopts KVSSD, which implements a key-value store inside an SSD, removing database and file system dependency from the host system. Since GPUKV does not rely on the host's database or file system, the overhead by I/O interactions for file-to-block address translation is eliminated.
- **P2P Communication between KVSSD and GPU:** In GPUKV, a GPU kernel can fetch its input data directly from KVSSD via P2P communication, eliminating excessive memory copy overhead during data transfer.

We implemented GPUKV, a GPU-driven computing model, by extending GPUfs [22] on KVSSD, which we have prototyped on the Cosmos+ OpenSSD platform [18] in a Linux environment. We also implemented several conventional CPU-driven GPU programming approaches such as KVSSD and RocksDB (C/NC) (Refer to Section 4). For evaluations, we used two kinds of server environments, a powerful high-end server and a less powerful low-end server and evaluated it for both synthetic and realistic GPU workloads. In our extensive evaluation, GPUKV had up to 9.1× and 18.7× performance improvement of execution time over KVSSD and RocksDB (C/NC) for both synthetic and real workloads. Also, GPUKV consumed 176× and 35× less CPU cycles than conventional CPU-driven GPU computing approaches with synthetic workloads on high-end and low-end servers, respectively.

2 BACKGROUND AND MOTIVATION

This section provides background on data transfer from key-value store to GPU and a description of the motivation for this study.

2.1 Data Transfer from Key-Value Store to GPU

Figure 1 shows two different approaches to transferring input data from key-value store to GPU. The conventional approach involves performing multiple memory copy operations from storage devices to GPU memory when sending input data to the GPU. On the other hand, the P2P data transfer approach is to directly transfer input data from storage devices to GPU memory without CPU intervention during data transfer.

2.1.1 Conventional Approach. To clearly articulate data transfer overhead in the conventional approach, we will use the RocksDB key-value store [19] as an example. RocksDB manages multiple SSTable files in a file system. A single SSTable stores a set of key-value pairs. In addition to data blocks containing values, an SSTable maintains a bloom filter to examine the existence of a key, and an index block to search the data block offset. To get the corresponding key-value, RocksDB first selects candidate SSTables and checks their

bloom filters. For each SSTable, if the bloom filter returns a positive, it checks the index block whether the key exists. If the key is present, it finally retrieves the corresponding values. Since RocksDB runs on top of the file system, SSTable files must be loaded into the main memory and then evaluated. Then, values corresponding to keys will be copied to GPU memory.

Figure 1(a) shows the aforementioned data transfer flow in terms of control and data path. The host application is responsible for fetching key-value pairs from RocksDB, copying them into GPU memory, and launching the GPU kernel. ① First, the application issues a GET request to RocksDB. ② To get values corresponding to keys, RocksDB reads candidate SSTable files including bloom filters through the file system. The file system figures out the LBAs of the files and sends a read request with LBAs of files to SSD through the NVMe Driver. Ⓐ After the read request with LBAs reaches the SSD, the SSD controller transfers the data of LBAs to kernel space buffers of the file system using direct memory access (DMA). Ⓑ Once again, the file system copies back the data to user space buffers of RocksDB, and then RocksDB tests bloom filters. Then, RocksDB searches index and data blocks. ③ After fetching values, the RocksDB GET request completes. ④ The input data is now ready to be sent to GPU memory, and the host application requests the GPU driver to copy the data. Ⓒ Since the GPU driver actually copies the data, the input data is temporarily stored on the kernel buffer of the system memory. Ⓓ Finally, the input data is copied to GPU memory via DMA.

2.1.2 Data Transfer using P2P Communication. Direct data transfer using PCIe P2P communication can reduce host-side CPU and memory overhead due to multiple memory copy operations compared to the conventional approach. However, in practice, applying P2P data transfer to a system using a key-value store is not that simple, and it still entails significant host-side CPU usage. Figure 1(b) shows a scenario of P2P data transfer with RocksDB. Issuing a GET request, examining bloom filters and searching index blocks work in the same manner in the conventional approach (①, ②, Ⓐ, Ⓑ). However, there are additional paths (③, ④). Refer to red solid arrows in Figure 1(b). ③ When the RocksDB engine finds a value corresponding to the key, it should consult with file system using tools such as `FIBMAP` *ioctl* to determine the LBAs of the value. In order to transfer the value using P2P, it has to specify the LBA information in the NVMe command to the SSD controller. ④ A P2P data transfer request with LBAs is sent to the SSD controller via the NVMe driver. Then Ⓒ the SSD controller directly transfers data corresponding to LBAs to GPU memory without host-side CPU intervention. Finally, ⑤ the completion of NVMe command is informed to the host application and ⑥ the application launches the GPU kernel. Even though P2P communication enables direct data transfer, the on-disk layout of files must be extracted from the file system. Moreover, data has to be aligned for P2P transfer in both RocksDB and file system.

2.2 Motivation

We conducted several experiments to identify performance problems when performing P2P from SSD to GPU in a key-value store. The detailed experimental setup is described in Section 4.1. We measured the latency of a 4 KB key-value data transfer from SSD to GPU for RocksDB [19] both with and without P2P. We compared three configurations – *RocksDB*, *RocksDB w/ P2P* and *GPUKV (Ideal)*. *RocksDB* and *RocksDB w/ P2P* follow data transfer flows presented in Figure 1(a) and (b) respectively. *GPUKV (ideal)* is an ideal situation that can only be achieved by removing all the I/O stack overhead of the key-value store and file system in GPUKV. For a fair evaluation, we disabled all caching features of RocksDB and used *directIO* for file system operations.

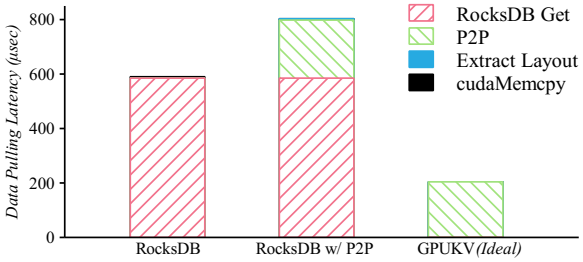
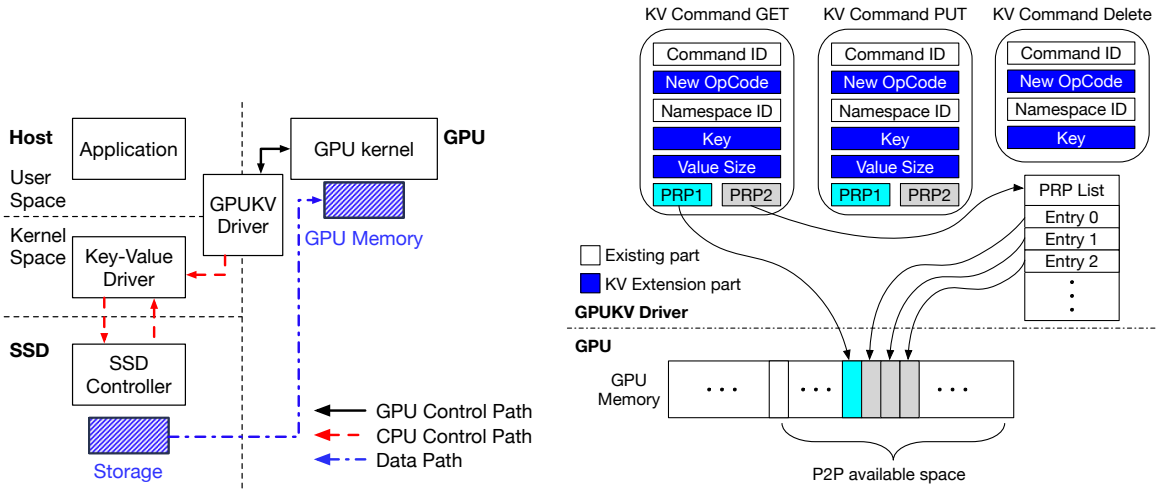


Figure 2: Comparison of latency breakdown of I/O calls for RocksDB with and without P2P, and NVMe SSD with P2P (Ideal)

Figure 2 shows latency breakdown results with RocksDB Get, P2P communication, data layout extraction (Extract Layout), and cudaMemcpy. We observed that for *RocksDB*, most of the latency is dedicated to I/O operations for RocksDB Get, which involves reading RocksDB files such as bloom filters, index blocks, and data blocks from an SSD. *RockDB* uses cudaMemcpy to copy data from host memory to GPU memory. The latency contribution of cudaMemcpy is negligible. On the other hand, surprisingly, *RocksDB w/ P2P* has higher latency than *RocksDB*.

RocksDB w/ P2P has almost the same contribution amount of I/Os for RocksDB Get as *RocksDB*. This is because RocksDB has to read metadata blocks of SSTable files such as bloom filter blocks to check if there exist values corresponding to keys, and it has to read entire SSTable files. *RocksDB w/ P2P* uses P2P for data transfer. P2P transfer adds a significant amount of latency compared to *RocksDB*. *RocksDB w/ P2P* should actually read data from an SSD's NAND flash. Note that NAND reads are much slower than DRAM reads. However, *RocksDB* moves data from the host's memory to GPU memory, which is much faster than the P2P communication from SSD to GPU. In addition, *RocksDB w/ P2P* has some overhead for extracting LBAs of the data to be transferred for P2P. This contributes a negligible amount of latency because the layout information is in the *inode* of the file and most of the *inodes* are already cached in the main memory by previous file system I/Os for SSTable files. Compared to *RocksDB* and *RocksDB w/P2P*, *GPUKV (Ideal)* has the lowest latency, less than half their latency.

In this paper, we propose GPUKV, which does not require interactions with the database and file system for P2P by using key-value SSDs. GPUKV aims to have a latency close to that of *GPUKV (Ideal)*. Because KVSSD manages key-value pairs internally inside an SSD, the overhead of extracting LBAs is much less and the host application does not need to care about data alignment for P2P. In addition, GPUKV is designed to directly issue key-value requests (*GET*, *PUT*)



(a) P2P data transfer to GPU with KVSSD

(b) Key-Value command extension of NVMe protocol

Figure 3: Data transfer with P2P from KVSSD to GPU in GPUKV

while executing the GPU kernel. Thus, it enables GPU-driven computing, excluding CPUs.

3 GPUKV FRAMEWORK

In this section, we present our design and implementation for the GPUKV framework.

3.1 System Architecture

GPUKV consists of the following three technologies: (i) Key-value abstraction for GPU that allows key-value requests while executing the GPU kernel, (ii) embedding key-value store in an SSD, and (iii) zero-copy kernel-bypass communication by P2P when loading data from KVSSD to GPU.

GPUKV provides a key-value store abstraction to the GPU, so the GPU kernel can issue key-value requests to KVSSD while it is running. Basic key-value operations are point queries, which are key-value commands such as *GET*, *PUT* and *DELETE*. Table 1 shows the prototype of the GPUKV API. All API operations take a `key` as a parameter in common, while `gpukv_get` and `gpukv_put` require `value_size` and `buffer`. Also, `buffer` is the address of the PCIe P2P-enabled GPU memory for the value. Key-value commands are sent to KVSSD via the GPUKV Driver.

Figure 3 shows data transfer with P2P from KVSSD to GPU and key-value command specification for KVSSD. Specifically, Figure 3(a) shows a diagram of the system architecture for GPUKV. The GPUKV Driver is the core module that conveys key-value commands from GPU to KVSSD and prepares P2P data transfer. The GPUKV Driver conveys key-value commands to the Key-Value Driver using remote procedure call (RPC) implemented using GPU shared memory. We extended a Linux NVMe Driver to support Key-Value NVMe commands and P2P data transfer. The Key-Value Driver issues NVMe commands to KVSSD on behalf of the GPUKV Driver. Destination addresses of the NVMe command are filled with P2P-capable GPU memory addresses. Therefore, the GPU kernel receives key-value data directly from KVSSD in P2P-capable GPU memory addresses.

Key-value requests are processed asynchronously in batches. The GPUKV Driver is implemented employing two CPU threads: The Issue thread and Completion thread. Issue thread batches at most 32 key-value requests and issues NVMe commands asynchronously. Batch size is determined based on the number of threads in a single warp of GPU. In the meantime, the Completion thread waits for the completion of NVMe commands. When the Completion thread notices the completion of any NVMe command issued, it immediately informs the GPUKV Driver so that RPC from GPU can be returned and resume computation with the request value.

The Key-Value Driver running in the host conveys key-value requests from the GPUKV Driver to KVSSD. To support key-value requests and P2P data transfers, we extended the NVMe protocol to support Key-Value operations and modified the Linux NVMe driver. Figure 3(b) shows the extended NVMe protocol for key-value commands. The opcode area specifies *GET*, *PUT* and *DELETE* operations using vendor-specific opcodes in the NVMe protocol. We used the starting LBA area in the NVMe commands to specify a key with a length of 8 bytes. *PUT* and *GET* commands use the page list for the value. However, because of its variable size, the length of the value may not be expressed with the block size and the number of blocks. To solve this problem, we used the reserved area of the NVMe command to specify the length of the value for *PUT*, and the size of the buffer for *GET*.

Table 1: GPUKV API for KV Command

Command	Function	Parameters
GET	`gpukv_get`	`key, value_size, buffer`
PUT	`gpukv_put`	`key, value_size, buffer`
DELETE	`gpukv_delete`	`key`

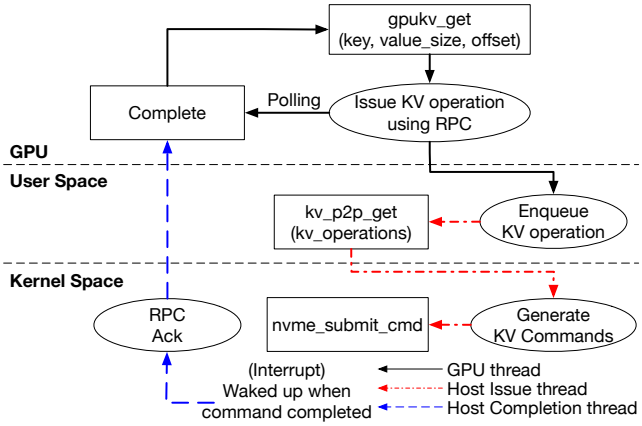


Figure 4: Functional flow of GPUKV

Meanwhile, the Physical Region Page (PRP) list in the NVMe command includes the page addresses of the data. For P2P with GPU, the PRP list needs to be filled with physical addresses of the P2P-available regions in the GPU memory. To this end, we used `nvidia_p2p_get_pages` to pin the pre-allocated GPU memory and map the pinned GPU memory to the DMA address space for P2P communication. Also, the NVIDIA API provides a page table of the pinned GPU memory. The Key-Value Driver stores this page table during initialization. Then, it translates the `buffer` passed from the GPU kernel to a physical address to fill the PRP list in the NVMe command.

3.2 Functional Flow of Get Operation

Figure 4 shows the functional flow of how GPUKV handles `gpukv_get`. When the GPU thread calls `gpukv_get`, the request information is pushed to the request queue in the GPU shared memory. The request queue is a shared memory that CPU and GPU threads can see. The GPU thread waits for the completion of the request by polling the status of the RPC request. In the meantime, the Issue thread in the host-side waits for new key-value requests by polling the request queue. When the Issue thread fetches a batch of key-value requests, it issues commands to the Key-Value Driver. Then the Key-Value Driver builds NVMe requests based on `buffer` and `value_size` in the request information. It fills the (PRP) lists using the page table of the pinned GPU memory. After that, NVMe commands are issued to KVSSD, and KVSSD transfers values to the GPU directly via P2P, which is a zero-copy kernel-bypass communication. On completion of an NVMe command, KVSSD wakes up the Completion thread. Then the Completion thread updates the status of the RPC requests. Finally, the GPU kernel notices completion and continues its computation. Meanwhile, the GPU kernel has waited for the RPC request completion by polling. Currently, GPUKV supports P2P only for Get operation. Put operation is performed by asynchronous buffered I/O through the CPU.

4 EVALUATION

In this section, we describe the experimental setup followed by the GPUKV evaluation.

4.1 Evaluation Setup

Implementation. We (i) extended GPUfs [22] to build a framework for GPUKV that issues key-value operations to KVSSD when executing the GPU kernel, and (ii) developed hash-map based KVSSD on the Cosmos+ OpenSSD Platform [18], the detailed specifications of which are shown in Table Table 2. Experiments were performed on an Intel 4 Core server equipped with NVIDIA Quadro P4000. Hardware details of the server and GPU are shown in Table 3 and 4, respectively.

In particular, to evaluate the performance/resource use efficiency of GPUKV according to the host's CPU performance, we modified the CPU settings of the server for two different configurations: *high-end* server (4 cores, 3.5GHz CPU) and *low-end* server (2 cores, 800MHz CPU). Note that the low-end server has lower CPU clock speeds and fewer CPUs than the high-end server.

Workload. We used both synthetic and realistic workloads to evaluate GPUKV. We developed an in-house benchmark for synthetic workload, which can simulate various data access patterns of GPU applications. The data access pattern of GPU applications is divided into *streaming* and *dynamic*. *Streaming* has predictable data access patterns. Therefore, the GPU kernel can expect to increase I/O performance by prefetching the next input dataset. By contrast, *Dynamic* has unpredictable data access patterns, thus prefetching does not help I/O performance.

For realistic workloads, we used GARDENIA [29], a graph processing benchmark. Among the various workloads of GARDENIA, we selected two representative workloads, Page Rank (PR) and Breadth-First Search (BFS). PR is a workload that scores relevance between web pages. PR updates the scores of pages over several rounds using their connectivity information and scores calculated in previous rounds. PR has a *streaming data access pattern*. In the workload, when updating the scores, vertex pages are updated collectively. By contrast, BFS is a *dynamic data access pattern* because

Table 2: Cosmos+ OpenSSD Platform Specification

SoC	Xilinx Zynq-7000
	ARM Cortex-A9 (up to 1000MHz)
	HYU Tiger 4 Controller (NVMe Controller in FPGA)
NAND Module	500 GB module, NVDDR2
	4-Channel, 8-Way
Interconnect	PCIe Gen2 8-lane
FTL	Page-level Mapping, On-demand GC

Table 3: Host Machine Specification

CPU	Intel Core i5-4690@3.5GHz 4C/4T
RAM	DDR3 4GB × 2
OS Kernel	Linux Kernel 4.1.52

Table 4: GPU Specification

GPU model	NVIDIA Quadro P4000 (CUDA ver. 10.1)
Interconnect	PCIe Gen3 8-lane
Memory	GDDR5 8GB
BAR memory	256MB (P2P enabled)

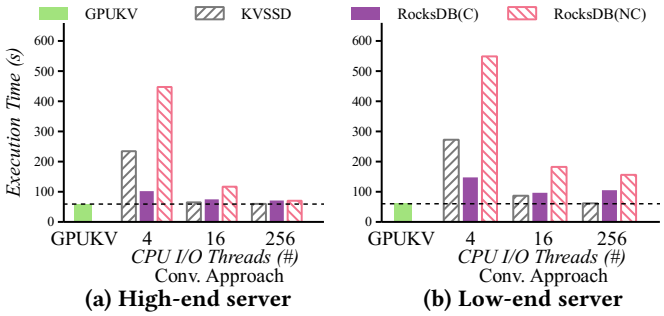


Figure 5: Streaming Workload ($W_{Streaming}$)

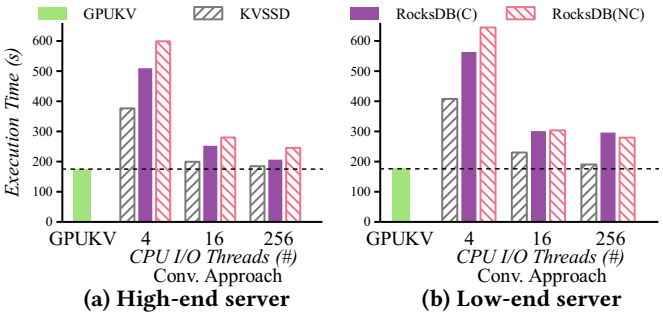


Figure 6: Dynamic Workload ($W_{Dynamic}$)

the next set of vertices is not predictable and can only be known when the current search step is complete. We stored graph datasets for PR and BFS in KVSSD using vertex ID as the key and the list of edge information (a pair of destination vertex ID and weight of edge) as its value. For the input dataset, we used roadNet-CA [12] from a graph repository [20]. RoadNet-CA is a road network graph of California with 1,971,281 vertices and 5,533,214 edges. The total data size of the graph in the key-value store is about 7.5 GB.

Comparison. We compared GPUKV with the following four systems:

- ***GPUKV***: It directly communicates with KVSSD via P2P communication. The GPU kernel is launched only once because key-value I/O operations are issued while executing the GPU kernel.
- ***KVSSD***: The host's application can directly communicate with KVSSD, allowing it to bypass the file system. However, unlike GPUKV, it has to use *cudaMemcpy* to feed data to the GPU's memory every time the GPU kernel needs it.
- ***RocksDB(C)***: RocksDB runs on the Ext4 file system. Input data for GPU kernels have to be copied using `cudaMemcpy` of the host's CPU. Similar to *KVSSD*, data has to be fed to the GPU every time the GPU kernel needs it. RockDB's cache is enabled and *buffered I/O* is used.
- ***RocksDB(NC)***: This is the same as RocksDB(C) except that RocksDB's cache is disabled and *directIO* is used.

Among the systems presented above, GPUKV is a representative GPU-based computing model, while other systems such as KVSSD, RocksDB (C), and RocksDB (NC) are representative CPU-based GPU computing models in which the host's application is responsible for moving data from storage devices to the GPU.

4.2 Performance Analysis

We first show the results for synthetic workloads and then show the results for realistic workloads.

4.2.1 Synthetic Workloads. We experimented with streaming patterns and dynamic patterns of key-values of 4 million 4KB sized values. Figure 5 and 6 show the measurements of total execution times for streaming and dynamic pattern workloads, respectively. In particular, for CPU-driven GPU computing models (KVSSD, RocksDB(C), and RocksDB(NC)), we experimented with increasing the number of CPU I/O threads. Note that in the figure, they are called the conventional approach (Conv. Approach). Increasing the number of I/O threads accelerates data movement to the GPU and reduces the execution time of the GPU kernel, but increases CPU use. Note that GPUKV does not use I/O threads for data movement on the CPU since GPU kernels and KVSSD can transfer data directly via P2P communication. GPUKV just uses two I/O command issue threads – *Issue thread* and *Completion thread*, which are not I/O threads that actually move data to GPU memory.

Streaming Pattern Workload Results. Figure 5(a) and (b) show the results for streaming workloads ($W_{Streaming}$) on high-end and low-end servers, respectively. In Figure 5(a), KVSSD and RocksDB(C/NC) have lower execution times as the CPU's I/O thread count increases. This is because parallel I/O can overlap memory copy operations from host memory to GPU memory, thus reducing the time caused by I/O. This pipeline effect between them maximizes the performance of each system. For example, when the I/O thread count is 256, KVSSD and RocksDB(C/NC) reach similar execution times as GPUKV. However, KVSSD and RocksDB(C/NC) require 256 threads, whereas GPUKV does not. On the other hand, even if GPUKV uses two I/O command issue threads, it outperforms KVSSD and RocksDB(C/NC). Note that these I/O command issues threads can be eliminated if the GPU kernel can operate the NVMe driver directly and issue NVMe commands to KVSSD. This observation explains why GPUKV needs less host-side resources such as CPU cycle and memory than KVSSD and RocksDB(C/NC).

Figure 5(b) depicts the results on the low-end server. Overall, we observed performance trends similar to Figure 5(a). However, compared to Figure 5(a), KVSSD and RocksDB(C/NC) had higher execution times. This is because of less powerful host CPUs. KVSSD shows a small performance drop of 19% on average, while RocksDB(C) and RocksDB(NC) show large performance drops of 41% and 50% on average, respectively. KVSSD and RocksDB(C/NC) use the host's CPU cycle and memory, and thus their performance is strongly affected by the host's CPU performance. The performance of KVSSD is less affected by host CPU performance than RocksDB(C/NC). This is because KVSSD bypasses the file system. However, in RocksDB(C/NC), it has to go through a heavy I/O software stack such as the file system and page cache, so it requires a high dependency on the host's CPU performance. On the other hand, GPUKV, which rarely uses the host's resources, has little performance degradation.

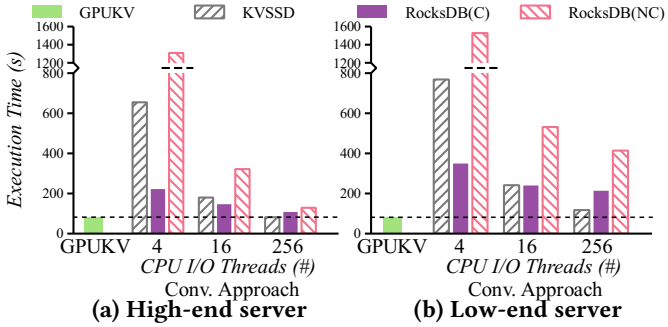


Figure 7: Page Rank (PR)

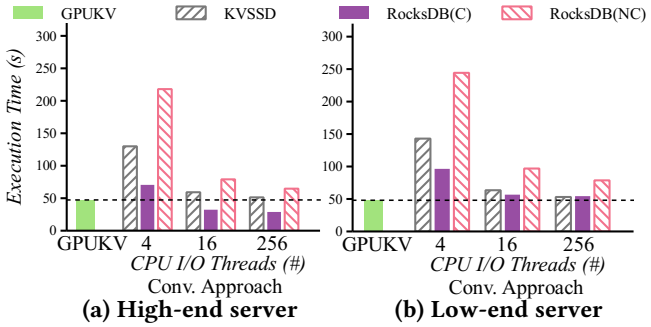


Figure 8: Breadth-First Search (BFS)

Dynamic Pattern Workload Results. Figure 6(a) shows the results for dynamic workloads in the high-end server. We observed that the performance gain of GPUKV is much higher in $W_{Dynamic}$ than $W_{Streaming}$. This is because CPU-driven GPU computing models (KVSSD and RocksDB(C/NC)) cannot overlap computation time and I/O time at all in dynamic data access pattern workloads. Unlike streaming workloads, the next dataset to be loaded in dynamic workloads can be known only when the current kernel execution is complete. Thus, prefetching does not help in the dynamic workload. In particular, we saw that KVSSD has lower execution times than RocksDB(C). Figure 6(b) depicts the results of the lower-end system. Overall, we observed that CPU-driven GPU computing models (KVSSD and RocksDB(C/NC)) had increased execution times due to less power CPUs.

In summary, in Figure 5 and 6, it can be seen that GPUKV had the lowest execution time in high-end and low-end servers. Other CPU-driven GPU computing models require a large number of I/O threads to achieve an execution time similar to GPUKV. In other words, in our prototype, GPUKV enables GPU-driven computing with very little dependence on the host's CPU.

4.2.2 Realistic Workloads. Figure 7 and 8 show the results for PR and BFS workloads. Note that PR is a streaming pattern workload, whereas BFS is a dynamic pattern workload. Figure 7 shows the results for PR. We have similar observations as Figure 5. Figure 7(a) depicts the results on the high-end server. It clearly shows the performance difference between GPUKV and CPU-driven GPU computing models (KVSSD and RocksDB (C/NC)). RocksDB has lower execution times when the number of I/O threads increases, but it still has a higher execution time than GPUKV. When 256 I/O threads are used, RocksDB(C) has 1.3 times and RocksDB(NC) has 1.6 times higher execution times than GPUKV. On the other hand, KVSSD, which has a simpler I/O software stack than RocksDB, has a similar execution time as GPUKV when the number of I/O threads is 256. Figure 7(b) shows the results on the low-end server. Overall, KVSSD and RocksDB have increased execution times compared to the high-end server. In particular, we observed that even if the number of I/O threads is 256, KVSSD has a higher execution time than GPUKV.

Figure 8 depicts the results for BFS. Interestingly, in Figure 8(a), we observed that RocksDB(C) has a lower execution time than GPUKV when more than 16 I/O threads are used. This is because in BFS, the data size is small and there is a big caching advantage in RocksDB(C) due to the small workload size. However, in Figure 8(b) where the low-end server is used, there is no benefit from caching, and GPUKV has the lowest execution time.

4.3 Resource Usage Analysis

We confirmed in previous experiments that GPUKV's performance is not affected by the host's CPU performance. On the other hand, it was confirmed that the performance of KVSSD and RocksDB is highly sensitive to the host's CPU performance. In this experiment, we evaluate how little GPUKV uses the host's CPU cycles compared to KVSSD and RocksDB. We used synthetic streaming workloads ($W_{Synthetic}$) for evaluation.

Figure 9 shows the result of the breakdown of CPU cycles consumed by each server configuration according to *I/O Request Handling, Thread Management, Polling, and Others*. IO Request Handling refers to CPU cycles consumed in processing IO Requests in each server configuration. Thread Management refers to CPU cycles consumed in thread creation, join, etc. when processing I/O with multiple threads in KVSSD and RocksDB. Polling refers to CPU cycles consumed when polling the shared memory for RPC communication. Others in the graphs show the CPU cycles consumed for context switch, thread synchronization, and other software overhead. In Figure 9, KVSSD and Rocks used 256 I/O threads on the high-end server, and they used 4 I/O threads on the low-end server. By contrast, GPUKV uses only two CPU I/O command issue threads. In Figure 9, all results are normalized to the CPU cycles used by GPUKV.

First, we analyzed the CPU cycles consumed by GPUKV. CPU cycles consumed by GPUKV in Figure 9(a) are about 4.3 times higher than that of Figure 9(b). This is because the high-end server has more CPU cores and faster CPU speed than the low-end server. GPUKV waits for key-value requests from the GPU through the GPUKV driver. Also, since there is no interrupt support between the host and GPU, the host has to poll the shared memory between the host and GPU and wait for the request. However, polling is unnecessary if (i) the request can be sent directly from the GPU to the SSD controller or (ii) an interrupt between the host and GPU is allowed. In Figure 9(a), polling occupies 96% of the consumed CPU cycles, and 70% in Figure 9(b). From these results, we can see that in our prototype, GPUKV is significantly consuming CPU cycles by polling. However, we note that the polling overhead can be eliminated if the aforementioned techniques are adopted.

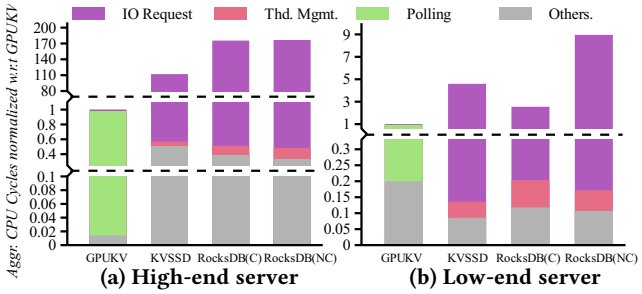


Figure 9: Normalized CPU utilization with respect to GPUKV for wynthetic streaming workloads ($W_{Streaming}$). The CPU cycles of GPUKV measured on high-end and low-end servers are 2.07×10^{11} and 4.78×10^{10} respectively.

Next, we analyzed the CPU cycles consumed by KVSSD and RocksDB. As expected, IO request handling accounts for more than half of the total CPU cycles in KVSSD and RocksDB. They consume many more CPU cycles than GPUKV. Note that KVSSD and RocksDB used 256 I/O threads. Referring to the results in Figure 5(a), the execution times of KVSSD and RocksDB when using 256 I/O threads are close to GPUKV. However, as observed in Figure 9, KVSSD and RocksDB consumed two to 117 times more CPU cycles than GPUKV.

5 RELATED WORK

P2P Communication. There have been studies on allowing direct access to the file system from GPUs [1, 22]. GPUfs [22] uses shared memory for communication between the host and GPU, which allows the kernel running on the GPU to perform I/O operations using a file system API. SPIN [1] also extends GPUfs to implement data transfer to the GPU using the CPU's page cache. In particular, SPIN proposed a hybrid algorithm that opportunistically uses both page cache and P2P communication when transmitting data from SSD to GPU. However, both GPUfs [22] and SPIN [1] rely on the host's file system to find the block layout of the file when performing P2P, so they use host resources considerably.

In-Storage-Processing (ISP) SSD. Summarizer [10] and GraphSSD [15] offload the host's CPU tasks to the SSD and execute them using its hardware resources. Summarizer [10] reduces communication overhead between the host processor and SSDs by analyzing TPC-H queries and offloading only data-intensive query tasks to the SSD processor. In addition, a method of properly setting the query offload ratio considering the computational efficiency of the SSD and communication overhead was proposed. GraphSSD [15] is an SSD that recognizes graph semantics and can perform graph processing inside the SSD, minimizing the communication overhead between the host and the SSD. When storing raw data, the SSD processor transforms data into a graph semantic aware format and stores it.

Key-Value SSD. Several works [6, 9, 11] have suggested KVSSDs that utilize ISPs to implement key-value storage inside SSDs. The host's key-value store is offloaded to an SSD and runs within the SSD. KVSSD extends the NVMe command to send a key-value request to the SSD. In particular, Transaction [9] proposed a compound command that can minimize I/O interface overhead with the SSD by merging multiple key-value pairs in one NVMe operation. iLSM [11] implements a log structure merge tree in firmware for key-value pair management. These KVSSDs completely move away from the file system overhead, simplifying the I/O stack. Pink [6] compared and analyzed various data structures such as hash and LSM for KVSSD design.

Heterogeneous Computing. Heterogeneous computing refers to a system composed of high-speed main host processors and slow but large-scale parallel co-processors (CPU-GPU). In this system, the CPU and GPU can execute the workload cooperatively, improving performance [2, 4, 5, 14, 16]. However, in these systems, when the GPU and CPU share data in the page cache, problems such as data inconsistency and false sharing occur. Qilin [14] automated computational mapping, distributing work within the heterogeneous systems. Solros [16] used the Xeon Phi co-processor, and proposed an OS that can use it efficiently by delegating a complex I/O stack to a high-speed host processor. GAIA [2] solved the problem of false sharing and data inconsistency by integrating the GPU memory and the OS page cache.

6 CONCLUSION

In a system environment where the key-value store is running on the host's file system, it has to go through a complex I/O stack to supply input data to the GPU kernel. Previously, P2P communication was proposed, allowing the GPU and SSD to communicate directly without memory copy operations. However, in order to copy data from the key-value store that stores unstructured data to GPU memory, CPU intervention for interaction with the database or file system cannot be avoided.

In this paper, we proposed GPUKV, a framework where KVSSDs and GPUs perform direct P2P communication to bypass complex I/O stacks for loading data into GPU memory. In GPUKV, the GPU kernel sends key-value requests directly to KVSSDs. Then, KVSSDs can transfer data directly to GPUs without the intervention of the CPU, enabling full GPU-driven computing. We used both synthetic and realistic workloads for evaluation. Our extensive evaluation results show that for most workloads, GPUKV results in the lowest execution time with minimal host resource usage such as CPU cycle and memory compared to conventional CPU-based GPU computing models.

ACKNOWLEDGMENTS

This work was supported by a research grant from SK hynix.

REFERENCES

[1] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. SPIN: Seamless Operating System Integration of Peer-to-peer DMA between SSDs and GPUs. In *In Proceedings of the USENIX Annual Technical Conference (USENIX ATC '17)*. 167–179.

[2] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. GAIA: An OS Page Cache for Heterogeneous Systems. In *In Proceedings of the USENIX Annual Technical Conference (USENIX ATC '19)*. 661–674.

[3] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. 2019. LightStore: Software-defined Network-attached Key-value SSD Drives. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 939–953.

[4] Prince Hamandawana, Awais Khan, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. 2020. Crocus: Enabling Computing Resource Orchestration for Inline

Cluster-wide Deduplication on Scalable Storage Systems. *IEEE Transactions on Parallel & Distributed Systems* 31, 08 (August 2020), 1740–1753.

[5] Anakhi Hazarika, Soumyajit Poddar, and Hafizur Rahaman. 2020. Survey on Memory Management Techniques in Heterogeneous Computing Systems. *IET Computers & Digital Techniques* 14, 2 (February 2020), 47–60.

[6] Junsu Im, Jinwook Bae, Changwoo Chung, Avind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *In Proceedings of the USENIX Conference on File and Storage Technologies (USENIX FAST '20)*. 173–187.

[7] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-performance Key-value SSD. In *In Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '17)*. 373–384.

[8] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG Lee. 2019. Towards Building A High-performance, Scale-in Key-value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage (Systor '19)*. 144–154.

[9] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. 2019. Transaction Support Using Compound Commands in Key-value SSDs. In *In Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '19)*.

[10] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Hung-Wei Li, Jing an Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*. 219–231.

[11] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. iLSM-SSD: An Intelligent LSM-Tree Based Key-value SSD for Data Analytics. In *In Proceedings of the IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '19)*. 384–395.

[12] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and The Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[13] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endow.* 9, 14 (2016), 1647–1658.

[14] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*. 45–55.

[15] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. 116–128.

[16] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. 2018. Solros: A Data-centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*. 1–15.

[17] NVidia. 2020. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[18] OpenSSD. 2017. Cosmos Plus OpenSSD Platform. http://openssd.io/.

[19] RocksDB. 2020. RocksDB. https://rocksdb.org/.

[20] Ryan A. Rossi and Nesreen K. Ahmed. 2013. Graph Repository. http://www.graphrepository.com.

[21] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6 (January 2018), 1–35.

[22] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating A File System with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 1–13.

[23] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. 2015. Analyzethis: An Analysis Workflow-aware Storage System. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[24] Hyogi Sim, Geoffroy Vallee, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, and Ali R Butt. 2018. An Analysis Workflow-aware Storage System for Multi-core Active Flash Arrays. *IEEE Transactions on Parallel and Distributed Systems* 30, 2 (2018), 271–285.

[25] Bruno Stefanizzi. 2014. DirectGMA on AMD's FIREPRO GPUs. http://developer.amd.com/wordpress/media/2014/09/DirectGMA_Web.pdf.

[26] Devesh Tiwari, Simona Bobila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solin. 2013. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *In Proceedings of the USENIX Conference on File and Storage Technologies (USENIX FAST '13)*. 119–132.

[27] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 53–65.

[28] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2018. KVSSD: Close Integration of LSM Trees and Flash Translation Layer for Write-efficient KV Store. In *In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '18)*. 563–568.

[29] Zhen Xu, Xuhao Chen, Jie Shen, Yang Zhang, Cheng Chen, and Canqun Yang. 2019. GARDENIA: A Graph Processing Benchmark Suite for Next-Generation Accelerators. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 15, 1 (2019), 1–13.

[30] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *In Proceedings of the International Conference on Parallel Architecture and Compilation (PACT '15)*. 13–24.