SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs

SHAI BERGMAN, TANYA BROKHMAN, TZACHI COHEN, and MARK SILBERSTEIN,
Technion-Israel Institute of Technology

Recent GPUs enable Peer-to-Peer Direct Memory Access (p2p) from fast peripheral devices like NVMe SSDs to exclude the CPU from the data path between them for efficiency. Unfortunately, using p2p to access *files* is challenging because of the subtleties of low-level non-standard interfaces, which bypass the OS file I/O layers and may hurt system performance. Developers must possess intimate knowledge of low-level interfaces to manually handle the subtleties of data consistency and misaligned accesses.

We present *SPIN*, which integrates p2p into the standard OS file I/O stack, dynamically activating p2p where appropriate, transparently to the user. It combines p2p with page cache accesses, re-enables read-ahead for sequential reads, all while maintaining standard POSIX FS consistency, portability across GPUs and SSDs, and compatibility with virtual block devices such as software RAID.

We evaluate SPIN on NVIDIA and AMD GPUs using standard file I/O benchmarks, application traces, and end-to-end experiments. SPIN achieves significant performance speedups across a wide range of workloads, exceeding p2p throughput by up to an order of magnitude. It also boosts the performance of an aerial imagery rendering application by 2.6× by dynamically adapting to its input-dependent file access pattern, enables 3.3× higher throughput for a GPU-accelerated log server, and enables 29% faster execution for the highly optimized GPU-accelerated image collage with only 30 changed lines of code.

CCS Concepts: • **Computing methodologies → Graphics processors**; • **Software and its engineering → Operating systems**; *Software performance*; *Designing software*;

Additional Key Words and Phrases: Accelerators, operating systems, GPU, file systems, I/O subsystem

ACM Reference format:
Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2019. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. *ACM Trans. Comput. Syst.* 36, 2, Article 5 (April 2019), 26 pages.
https://doi.org/10.1145/3309987

1 INTRODUCTION

GPU-accelerated applications often require fast data transfers between the GPU and storage devices. For example, a video editor may offload its compute-intensive frame processing to the GPU,

M. Silberstein was supported by the Israel Science Foundation (Grant No. 1138/14) and the Israeli Ministry of Science. The authors also gratefully acknowledge funding from National Science Foundation Grants No. CNS-1017785 and No. CCF-1333594.

Authors' addresses: S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, Technion-Israel Institute of Technology, Electrical Engineering Department, Technion City, Haifa 3200003, Israel; emails: {shaiberg1, tanyabr}@tx.technion.ac.il; tzachi_cohen@hotmail.com; mark@ee.technion.ac.il.

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.
© 2019 Association for Computing Machinery.

0734-2071/2019/04-ART5 $15.00
https://doi.org/10.1145/3309987

while transferring tens of gigabytes of raw video contents from files. They combine high I/O demands with heavy computations amenable to GPU acceleration. Thus, application performance is bounded by the throughput of transfers between the disk and the GPU. As high-speed *NVMe SSDs* with multi-GB/s I/O rates are becoming commodity, we expect an increasing number of I/O-intensive applications to benefit from GPU acceleration. In fact, recent AMD Solid State GPUs (SSG) [1] target such I/O intensive workloads by hosting NVMe SSDs on a GPU card.

To realize the potential of high speed I/O devices in GPU workloads, all recent discrete GPUs enable *peer-to-peer direct memory access (*P2P*)* to GPU memory from PCIe-attached peripherals [2, 3]. P2P eliminates redundant copies in CPU memory when transferring data between the devices. Without P2P, copying file contents into a GPU buffer requires reading it first into an intermediate CPU buffer, which is then transferred to the GPU. P2P allows direct transfers into GPU memory, improving performance and power efficiency, as has been shown in several prior works [4–8].

Unfortunately, P2P poses significant programming challenges. First, the usage of P2P requires intimate knowledge of low-level hardware constraints. For example, P2P cannot access files at misaligned file offsets [9] and may be slow or unusable across devices in different NUMA nodes [10].

More crucially, P2P actually hurts system performance for a range of popular file access patterns. Figure 2 demonstrates that for short sequential reads, a popular file access pattern generated by programs like the grep utility, P2P is dramatically slower than CPU-mediated I/O. P2P outperforms CPU-mediated I/O only for reads larger than 512KB. For smaller reads though, CPU-mediated I/O reaps the benefits of the OS read-ahead mechanism, which P2P bypasses.

Finally, the use of P2P in hybrid CPU-GPU producer-consumer workloads is prone to subtle consistency bugs. Consider, for example, a log processing application like fail2Ban [11], accelerated by leveraging GPUs. fail2Ban is an open source Linux framework used to detect and prevent remote intrusion attacks. Specifically, it uses regular expressions to detect suspicious activities logged in a constantly updated file, and is able to trigger actions, such as inserting an IP into an iptables jail. Using P2P to read recently updated files might result in an inconsistent read if the contents have not yet reached the disk. Furthermore, because P2P is not integrated with the page cache, users would not benefit from the extensive OS efforts to cache file contents.

We conclude that P2P *between SSDs and GPUs is too low-level a mechanism to be exposed directly to developers*. Existing frameworks and academic research [4–8] provide non-standard, custom APIs for performing P2P, but rely on the programmer to work around its limitations and to choose the best-performing transfer mechanism for a given application scenario. Instead, the OS should hide the subtleties of direct access to storage, exploit existing file I/O optimization mechanisms like read-ahead and page cache, while *dynamically* and *transparently* steering the data path to P2P.

We introduce **SPIN**, a system that achieves these goals by integrating P2P into the file I/O layer in the OS. The programmer uses *standard* `pread` and `pwrite` calls to transfer the file contents to and from the GPU memory, while SPIN seamlessly activates P2P when necessary. Unlike previous works on P2P [4–8], which target GPU-only workloads with large sequential reads, SPIN addresses a broad range of application scenarios with diverse file access patterns and cooperative CPU-GPU processing.

SPIN addresses three key challenges: integration of P2P with the page cache, read-ahead for GPU reads, and invocation of P2P via a direct disk I/O interface.

Combining page cache and P2P. If a GPU read request can be partially served from the CPU page cache, then naively reading all the cached data from memory and the rest via P2P might be slower by up to 16× vs. serving the whole request via P2P. We construct a greedy heuristic that solves the underlying scheduling problem for every access, and produces the interleaving schedule of the two data transfer mechanisms that achieves about 98% of the optimal performance (Section 4.3.1).

GPU read-ahead. Our read-ahead mechanism uses CPU page cache pages to store the contents of prefetched data for GPU reads. However, SPIN prevents page cache pollution by maintaining a separate GPU read-ahead eviction policy that restricts the space used for prefetched contents (Section 4.3.2).

Direct disk I/O for p2p. Using direct disk I/O interface to invoke p2p SSD-GPU transfers is advantageous because of its tight integration with the file I/O stack, including page cache consistency handling and file offset-to-logical block address mapping. However, direct I/O calls cannot be used with GPU resident pages. We devise a lightweight *address tunneling* mechanism to overcome this problem (Section 5.3).

We implement and systematically evaluate SPIN in Linux by running standard file system benchmarks, application traces and full applications. We use NVIDIA K40 and AMD R9 Fury GPUs with two Intel P3700 SSDs, both separately and in a software RAID. SPIN tracks or exceeds the performance of the best transfer mechanism for the respective access pattern, with pronounced benefits over p2p for sequential accesses and accesses to cached files. For example, it achieves 10.1GB/s when reading a file from the page cache—3.8× higher than 2.65GB/s of p2p in the same configuration (within 5% of the maximum SSD bandwidth). For partially cached files, SPIN is faster than *either* CPU-mediated I/O or p2p in isolation, e.g., by 2× and 20%, respectively, for 50% cache hits. This is because it optimizes each I/O request by transferring data from both the page cache and via p2p, while dynamically minimizing the overheads of switching between the two.

SPIN is compatible with virtual block devices such as software RAID, in contrast to the published p2p implementations. SPIN achieves up to 5.2GB/s of file streaming performance from two SSDs in RAID-0 managed by Linux software RAID [12]—the fastest p2p result reported to date, to the best of our knowledge. For comparison, AMD SSG [1] GPUs with the SSD drives on a GPU card [13] reportedly achieve 4GB/s and require custom API and special-purpose hardware.

In real application scenarios, we evaluate a GPU-accelerated log server, an aerial imagery viewer [14], and an image collage creator [15]. SPIN achieves significant speedups for all applications, e.g., 3.3× for the log server. A highly optimized implementation of the collage creator is improved by 29% while requiring modification of only 10 LOC.

Our main contributions are as follows:

- Analysis of programmability and performance limitations of p2p.
- Integration of p2p into the OS file I/O stack, including standard file I/O API, page cache with a transfer interleaving scheduler, read-ahead and enabling p2p via direct I/O.
- Thorough evaluation on synthetic and real workloads for both NVIDIA and AMD GPUs, showing significant performance benefits of SPIN over alternatives.

2 BACKGROUND

This section provides a brief overview of the system architecture we target in our work.

2.1 System Architecture

We consider a system where the CPU, discrete GPUs, and NVMe SSD are connected via Peripheral Component Interconnect Express (PCIe) bus. An illustration of this setup can be seen in Figure 1. The PCIe switch enables fast *peer-to-peer direct memory access* (p2p) between the GPU and the SSD. The data between peripheral devices is usually transferred using an intermediate CPU memory buffer. For example, when copying data from the SSD to the GPU, the CPU requests the SSD to perform transfer into its local buffer, and then requests the GPU to copy it into the GPU memory buffer. p2p allows the SSD to transfer data directly to/from GPU memory, bypassing the CPU.

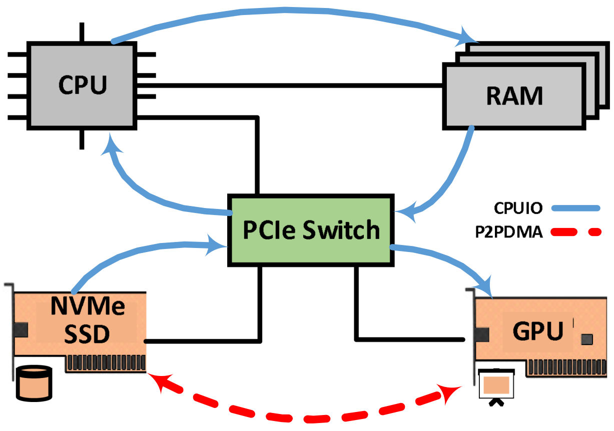

Fig. 1. System architecture and interconnect.

NVMe SSDs. NVM Express is a scalable host controller interface specification for accessing non-volatile storage over the PCIe interconnect.

PCIe BAR. PCIe devices expose their control registers and/or blocks of internal memory on the bus to be accessed by other PCIe devices. PCIe base address registers (BARs) hold the *bus* addresses of the regions where the device's registers and memory can be accessed, also commonly referred to as BARs.

Exposing Internal Memory via a BAR. Devices may expose their internal memory on their BAR, allowing direct access to that memory from the CPU or other PCIe devices. Both NVIDIA and AMD GPUs support this functionality, via GPUDirectRDMA and DirectGMA technologies, respectively.

To perform P2P, the GPU exposes a segment of its local memory on the BAR. Then the SSD DMA controller transfers data from or to the GPU BAR addresses. The DMA controller is oblivious to the physical location of its source or destination buffer, therefore P2P requires no special SSD support.

Mapping GPU Memory into Process Address Space. GPUs expose a portion of GPU memory on the PCIe bus (device's BAR) accessible to the CPU. To allow access to this memory from a user mode application NVIDIA's gdrcopy and AMD's OpenCL extensions provide the tools to map it into the process address space.

2.2 OS Components

Block Device Software Stack. Here, we provide a much simplified overview of the file I/O request control flow in Linux. File I/O requests are forwarded to the Virtual File System (VFS) by the respective system call. The VFS checks whether the requests can be served from the page cache, otherwise creating requests to the File System (FS) layer. The FS translates file's offsets into Logical Block Addresses (LBAs), which reflects their location inside the SSD. Further requests are then issued by the FS and forwarded to the block layer, which schedules and issues block requests to the relevant storage device.

Direct Disk I/O. Direct disk I/O (O_DIRECT) allows file system operations to bypass kernel caches and interact directly with the storage device.

POSIX File Semantics. POSIX File Semantics specifies the open, close, read, and write interfaces. Writes conforming to POSIX must be performed in a sequentially consistent manner. Writes must

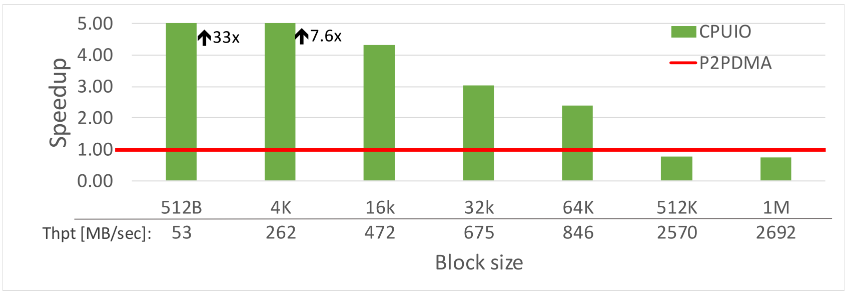

Fig. 2. The speedup of CPU-mediated I/O over P2P for sequential reads. Absolute P2P bandwidth (MB/s) below X axis.

appear as atomic operations. Reads occurring simultaneously to writes will retrieve in either all or none of any write.

3 MOTIVATION

Prior works [4–8] show that P2P between SSDs and GPUs substantially boosts system performance for popular GPU benchmarks. These applications exhibit streaming access patterns, sequentially reading files in large chunks. Our measurements in this section, however, show that P2P is actually *slower* than CPU-mediated I/O for access patterns and application scenarios that have not been considered previously. We then highlight the key challenges that P2P poses to programmers, motivating its integration into the OS file I/O stack.

3.1 P2P Inefficiencies

Short Sequential Reads. We compare the performance of P2P and CPU-mediated I/O for reading file contents into NVIDIA GPU (AMD GPUs are similar). We run the standard *TIOtest* [16] benchmark only modifying it to transfer data to GPU buffers. The CPU-mediated I/O version issues `pread()` into a CPU buffer followed by `cudaMemcpy()` to transfer the buffer to the GPU. For P2P, we use our own implementation described in detail in Section 5.3. For the hardware setup, see Section 6.

Figure 2 shows the relative throughput of sequential accesses to a 100MB file. P2P is more than an order of magnitude slower that CPU-mediated I/O for very short reads, and about 3× slower for larger 32KB reads. This is a common access pattern, found, e.g., in grep utility. P2P attains speedups only for reads of 512KB and above.

This performance gap is due to the read-ahead mechanism, which transparently optimizes CPU-mediated I/O, and which P2P bypasses entirely. The OS asynchronously prefetches the file into the page cache, overlapping the reads from the disk with CPU-GPU data transfers. The prefetcher gradually increases the size of the prefetch data requests up to 512KB (by default), achieving much higher effective bandwidth to SSD than P2P, which performs short reads.

Complex Workloads. P2P is efficient only for accessing files in persistent storage, but is significantly slower than CPU-mediated I/O if the file contents are cached in the page cache, as is often the case for complex software systems with multiple cooperating applications. However, since the page cache contents change dynamically depending on the workload, a programmer is left without a single best choice of file transfer mechanism. For example, consider a central log server that receives logs from other machines over the network and stores them locally. A log scanner invoked as another application might analyze the logs later to detect suspicious events. Using P2P for such a streaming workload might seem as a viable choice. However, if the scanner is invoked immediately after the files are updated, the contents might still be in the page cache, thus using P2P would reduce system throughput, as we also show in our experiments in Section 6.

3.2 P2P Programming Challenges

P2P is a low-level mechanism, exposed directly to the programmer. Besides the performance issues discussed earlier, it introduces a number of challenges to programmers.

Non-standard API. There is no standard OS API for accessing files via P2P. All the existing frameworks deviate from the standard file API, e.g., `send()/recv()` streaming-like calls in Gullfoss [5] and NVMMU's `move()` [4]. Custom APIs require programmers to explicitly select the file transfer mechanism, a choice that is not trivial in many cases, as we explain earlier.

Data Inconsistency. Updates written to a file via regular FS API will be stored in the page cache first, and might remain invisible to the P2P unless the file contents are written back to the disk.

Unsupported Misaligned Accesses. P2P requires both the source and destination to be aligned according to device-specific rules (p. 91, Reference [9]). Specifically, an SSD data offset and destination address must be aligned on the minimum transfer size supported by the device (512 bytes on Intel SSDs), otherwise the I/O request fails.

Incompatibility with Block Layer Extensions. Existing P2P frameworks are incompatible with block layer extensions, such as Logical Volume Manager and software RAID [12]. The P2P implementations must know the translation of the file offsets to SSD logical block addresses (LBA) to enable user programs to access files rather than disk blocks. The translation can be retrieved via `filemap` call, but it should not be used while the disk is mounted, and will not give correct mapping for virtual block devices such as LVM and software RAID.

In summary, as GPUs find their ways to accelerating complex data-processing systems, such as Apache Spark [17], the simplicity, portability, and transparent optimizations offered by OS file I/O interfaces make such interfaces essential for building efficient and maintainable GPU-accelerated systems.

These observations guide us in our goal to integrate P2P mechanism into the OS file system layer as we discuss next.

4 DESIGN

Design Goals. SPIN aims to integrate P2P into the OS file I/O layer. It uses P2P as a low-level mechanism for optimizing file I/O where applicable. We focus on the following design goals:

- **CPU-GPU workloads:** efficiently handle complex scenarios with opportunistic data reuse, where applications may share files, e.g., in producer-consumer interaction. SPIN should provide standard POSIX file consistency guarantees regardless of the transfer mechanism used.
- **Various access patterns:** enable high performance across random/sequential access patterns and an unrestricted range of request sizes, from as little as a few bytes.
- **Standard File API:** support standard I/O calls like `pwrite` () and `pread` () for portability, and ease of use.
- **Compatibility:** be compatible with virtual block devices such as LVM and software RAIDs, as well as with different GPUs and SSDs.

4.1 Design Considerations

Page cache is the cornerstone of file I/O in CPU systems, but its integration with P2P raises a number of questions.

Page Cache in GPU Memory? One way to combine caching with P2P is to partition the page cache between the CPU and GPU memories, and use each to cache file accesses from the respective device. In fact, GPUfs demonstrated the benefits of hosting a page cache for GPU tasks in GPU memory [15, 18]. Unfortunately, modern GPUs still lack critical features to enable OS-controlled GPU-resident page cache. In particular, they do not support anonymous memory that does not belong to any CPU process, neither do they provide the means for the OS to manage GPU memory mappings. As a result, GPUfs, for example, maintains a *per-application* page cache, which disappears when an application terminates. Workarounds, such as running a daemon process in user space that *owns* the GPU page cache, are insecure, because they expose the whole page cache to all running GPU tasks. We conclude that maintaining page cache in GPU memory is currently not practical.

Reusing File Contents from the CPU Page Cache. P2P transfers bypass the CPU page cache. But if the content is already in the cache, using P2P would be slower than reading the data from the page cache. However, if only part of the request can be served from the cache, the best way to combine P2P and cache accesses depends on the distribution of the pages in the cache. For example, if only every second page in a 8MB-large read request is cached, reading from the page cache is 16× slower than a single P2P of the whole requested buffer. We address the problem of optimal interleaving in Section 4.3.1.

Read-ahead Integration. A read-ahead mechanism is essential for fast sequential accesses (see Section 3), but the best way to integrate it with P2P is not obvious. Technically, the prefetcher never runs, because P2P bypasses the standard file I/O control path, which identifies a sequential access pattern and triggers the read-ahead mechanism. However, if we re-enable the prefetcher, where will it store the prefetched contents? One of the benefits of P2P is that it does not pollute the CPU page cache with the data used only by the GPU. But without the page cache on the GPU, the read-ahead mechanism would have to store the prefetched data in the CPU page cache, losing this advantage. We discuss the prefetcher in Section 4.3.2.

Portability Across GPU Software. GPU vendors expose different APIs for GPU management and data transfers to and from GPU memory, none of which are available for use from kernel space. As a result, providing a generic OS service, which is agnostic to the GPU type and its software stack, is challenging.

4.2 Overview

Figure 3 shows the main design components. SPIN is positioned on top of the Virtual File System (VFS) layer in the Linux kernel. It augments standard file I/O system calls to recognize the requests that may benefit from P2P, and it selects the transfer mechanism according to its policies. We illustrate the interaction of the SPIN components on the example of `pread` (). The user allocates the destination buffer in GPU memory and passes the pointer to the buffer to `pread`. To make GPU memory buffers accessible to I/O calls, the user *maps the buffers into the CPU process address space* using existing GPU vendor-specific tools (Section 5). We note that using CPU-mapped GPU buffers in I/O calls is possible without SPIN, however P2P is not invoked.

The SPIN core is implemented in *P-router*. P-router inspects every I/O request (① in Figure 3) and detects the requests that operate on GPU memory buffers and are amenable to P2P. P-router invokes the *P-readahead* mechanism, which identifies sequential access pattern and prefetches file contents into a *GPU read-ahead partition* (GPU RA in Figure 3) of the *CPU* page cache, as described in Section 4.3.2. It also checks with *P-cache* whether the request can be served from the page cache, and creates an I/O schedule to interleave P2P and page cache accesses, as discussed in Section 4.3.1.

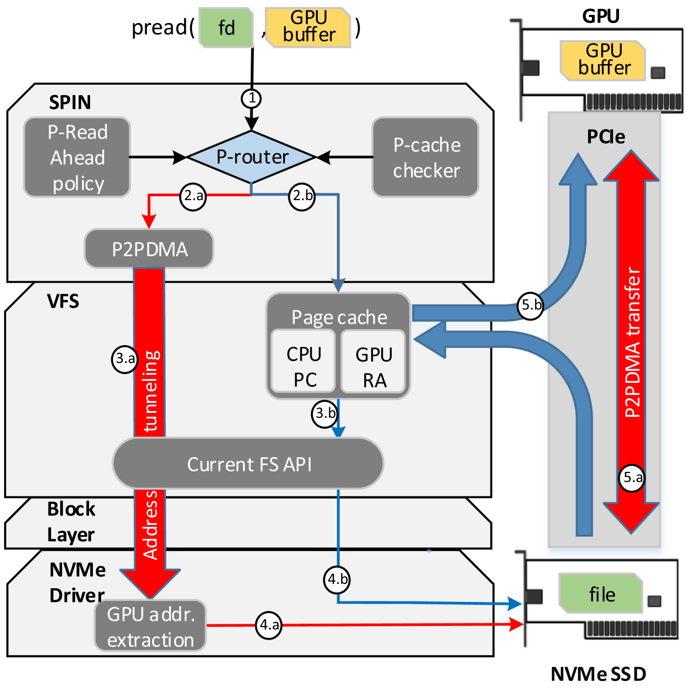

Fig. 3. SPIN high-level design and control flow of pread (), as explained in Section 4.2.

Finally, it generates VFS I/O requests that are served by a combination of the page cache (2.b) and P2P (2.a). In either case, P-router passes the control to the standard VFS I/O calls, but uses two different modes depending on the transfer mechanism. It invokes the regular CPU file call (4.b), (5.b) for accessing the file through the page cache, and uses the direct I/O access mode for P2P transfers (5.a). To invoke P2P via direct disk I/O interface, P-router employs an *address tunneling* mechanism (3.a) described in Section 5.3.

4.3 Integration with Page Cache

We deal with three aspects: interleaving page cache reads with P2P, integration with read-ahead, and data consistency.

4.3.1 Combining Page Cache with P2P.

Optimal Scheduling of Page Cache Transfers. P-cache retrieves the CPU page cache residence map for a given read access by consulting the cache's radix tree. If the entire requested region is cached, then the request is served from the page cache. However, if the cache contains only part of the requested data (due to eviction, for example), then the system combines both P2P and page cache transfers, by breaking the original request into sub-requests each served via its own method.

Finding the best interleaving of P2P and page cache accesses is a challenge. On the one hand, reading from the page cache is faster than reading from the SSD. On the other hand, interleaving P2P and page cache reads at a fine granularity results in poor performance, because short I/O requests to the SSD are less efficient than larger ones, and because of the P2P invocation overhead. Thus, SPIN needs to determine the best interleaving schedule for each I/O request.

The following example illustrates the problem. Consider a request of 20KB (five pages), with its second and fourth pages in the page cache. Then, there are three possible schedules: three

P2P transfers of 4KB and two 4KB transfers from page cache, a single P2P of 20KB of the whole range, and a combination of P2P and page cache transfers for the second and the fourth page, resulting in two P2P transfers of 4 and 12KB. The choice of the best schedule depends on the actual P2P throughput for each transfer size, as well as on the throughput of the page cache reads. The scheduling decision for different pages are *not* independent, however, because SSD transfer time is a non-linear function of the request size for smaller reads [19].

To summarize, the scheduling problem at hand is as follows: For a given I/O request, find all the constituent continuous ranges of pages that can be served from the page cache. For every such range, decide whether to transfer it from the page cache or via P2P, effectively merging it with the two flanking segments into a single P2P transfer, such that the total transfer time of the whole request is minimized.

Greedy Heuristic. This problem can be solved exactly in polynomial time via dynamic programming; however, this is too slow, since the solution has to be found for every I/O request. Instead, we simplify the problem to apply a simple greedy heuristic as follows.

We start by assuming that the P2P transfer time, $T_{p2p}(s)$, is a piece-wise linear function of the transfer size s of the form given in Equation (1). Intuitively, for requests smaller than S_{cutoff}, the device bandwidth is not saturated, thus the transfer time is constant and capped by the device's invocation overhead C_{p2p}. For requests larger than S_{cutoff}, the device operates at maximum bandwidth BW_{p2p}. These assumptions are consistent with the architectural model of modern SSDs [19]. Page cache transfers, in turn, always achieve maximum bandwidth, thus the transfer time for size s is $T_{pc}(s) = \dfrac{s}{BW_{pc}}$:

$$T_{p2p}(s) = \begin{cases} C_{p2p} & \text{if s} < S_{cutoff}, \\ C_{p2p} + \dfrac{s - S_{cutoff}}{BW_{p2p}} & \text{if s} \geq S_{cutoff}. \end{cases} \quad (1)$$

The greedy heuristic works as follows. For each three consecutive data ranges a, b, c, where b is in the page cache, if $|a| + |b| < S_{cutoff}$, then always choose P2P for b (where $|x|$ is the size of x). Otherwise, choose P2P for b if $T_{p2p}(|a| + |b| + |c|) < T_{p2p}(|a|) + T_{pc}(|b|) + T_{p2p}(|c|)$. In other words, P2P for b is preferable if the benefits of reading b from the page cache are smaller than the overhead of transferring c in a separate P2P transaction.

Parameter Fitting. We experimentally measure the transfer times for different request sizes for Intel P3700 SSD, and we fit the parameters of the transfer time function in Equation (1) using regression. The function fits very well, with the coefficient of determination of over 0.99. We find $S_{cutoff} = 512$KB and $C_{p2p} = 584\mu$s, which corresponds to the time for transferring 249 pages from the page cache. Thus, for two consecutive data ranges b, c, where b is in the page cache and c is not, b will be always transferred via P2P if $|b| < 249$ pages.

Evaluation. We build a simulator that quickly computes the transfer cost of an I/O request, given transfer schedule, using the transfer times measured on real hardware (AMD configuration in Table 1). Transfer times of data chunks ranging from 4KB to 8MB in increments of 4KB are measured on real hardware, several times for each transfer mechanism: P2P, and CPUIO. The simulator uses this dataset to approximate the transfer time of an interleaved data transfer, which utilizes both CPUIO and P2P. We validate the simulator experimentally on 5,000 interleaved I/O requests, and find that its error is 1.6% on average.

We use the simulator to evaluate the quality of the greedy heuristic, by comparing its results with the optimal transfer schedules obtained by the exact algorithm. Specifically, we compute the transfer time for the schedule obtained by the greedy heuristic, and compare it with the one of the

Table 1. Evaluation Platforms

Nvidia Tesla K40c	2 × Intel Xeon E5-2620v2, Intel C602 Chipset, 64GB DDR4, 1 NVMe SSD
AMD Radeon R9 Fury	Intel Core i7-5930K, Intel X99 Chipset, 24GB DDR4, 2 NVMe SSDs

Both use one or two Intel P3700 800GB NVMe SSD.

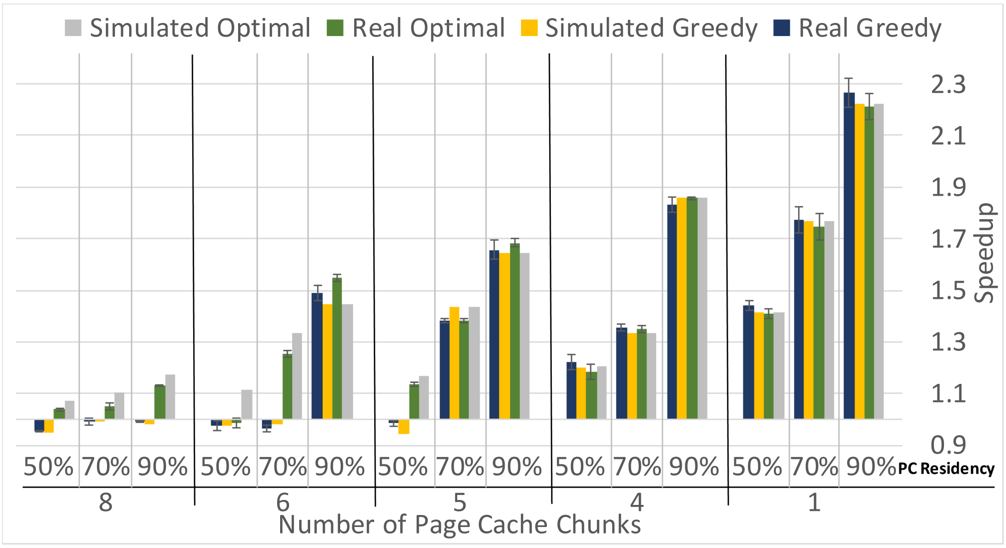

Fig. 4. Speedup of the simulator and real system transfer rates relative to real hardware P2P transfer rates (simulated transfer rates are identical) vs. number of page cache chunks (sparsity) and the transfer's data residency in the page cache.

optimal schedule obtained via exhaustive search. We evaluate the schedules on 200,000 random vectors, each representing an 8MB data transfer having different page cache residency patterns. We simulate all of the possible compositions of the data transfer for each vector to determine the optimal theoretical speedup that could be gained as well as the greedy algorithm's speedup. We find that the transfer time of the greedy schedules is within 98.9% of the optimal schedule on average. Figure 4 shows the speedups resulting from the simulator and the real system gained over real hardware P2P from a select subset of 8MB vectors, that vary in their sparsity and residency in the page cache.

Generalization to other SSDs. We believe that our heuristic reflects the general SSD performance trends and can be used with other SSDs. Specifically, architectural properties of SSDs, such as multi-channel/multi-way, enable a high degree of parallelism for relatively large requests. These requests are often striped across domains and exploit the internal parallelism SSDs offer [19, 20]. Therefore, our model, which predicts higher performance for larger requests, is consistent with these properties. We provide a calibration tool to perform the measurements and regression to automatically adjust S_{cutoff} and C_{p2p}.

4.3.2 *Read-ahead for GPU Accesses.* The OS read-ahead is not activated for accesses via P2P, therefore we introduce *P-readahead*. It stores the prefetched data in a special partition in the CPU page cache as we explain next.

GPU Read-ahead Cache. To avoid cache pollution by the contents prefetched as part of the read-ahead, we add a lightweight management mechanism, *GPU read-ahead cache, RA cache.* A page is assigned to the RA cache when it is first used by P-readahead to store the prefetched data. The pages in the RA cache belong to the OS page cache, and are subject to OS page cache management policies. In addition, the RA cache forces eviction of its pages once its total size exceeds a predefined threshold. If a page is later accessed by a CPU program, then the page is removed from the RA cache, but remains in the OS page cache. As a result, the pages used exclusively to store the data prefetched for GPU I/O do not pollute the OS page cache.

Read-ahead Mechanism. P-readahead watches for sequential access pattern by monitoring the last accessed offset in each file, similarly to the CPU read-ahead heuristic. For sequential accesses, the data is read into the GPU RA cache via CPU VFS calls, effectively engaging the original OS read-ahead mechanism redirected to store data in the GPU RA page cache. As a result, P-readahead respects the standard `fadvise` calls and does not require new management interfaces. We also modify the default behavior of P-readahead in response to `fadvise` policies, e.g., disabling it for `POSIX_FADV_RANDOM`.

For sequential requests that cannot be served from the page cache and exceed a certain threshold, P-router deactivates P-readahead and switches to p2p. The threshold equals to the maximum size of the OS-configured read-ahead window (512KB by default), which determines the maximum size of SSD requests generated by the read-ahead. Using p2p for requests exceeding the threshold results in larger SSD requests and higher throughput.

4.3.3 Data Consistency. Combining file accesses from the page cache with direct accesses to a storage device raises an obvious data consistency problem, since the data in the page cache might not be synchronized with the content on the SSD. Therefore, SPIN detects dirty pages in the range of the p2p transfer and explicitly performs a write back from the page cache to the SSD.

5 IMPLEMENTATION

Our implementation leverages existing kernel mechanisms to achieve SPIN's design goals. We encapsulate all new functionality in a kernel module SPINdrv, a slightly modified generic NVMe driver, and a lightweight user space library libSPIN. Thus, SPIN requires no modifications to the kernel and is readily deployable on existing systems. We explain the implementation of each component in detail.

5.1 User-space libSPIN

libSPIN is a shim that interposes on standard file I/O calls. The library is loaded via an `LD_PRELOAD` environment variable and overrides the original file I/O functions in `glibc`. Applications that do not load libSPIN may share files with those that do, retaining POSIX file semantics. We leave inter-process file sharing optimizations and coordination for future work.

When initialized, the library allocates a phony buffer for tunneling GPU addresses to the NVMe driver, as we describe in Section 5.3. Each I/O call intercepted in the library is translated into the respective `ioctl()` to the SPINdrv, which we discuss next.

5.2 SPINdrv

The driver implements the SPIN design including the page cache and read-ahead as described in Section 4. In addition, it introduces a new *address tunneling* mechanism to enable p2p via direct disk I/O, which we discuss in Section 5.3.

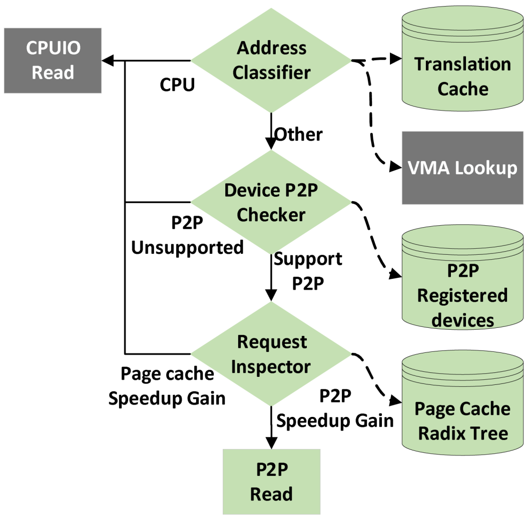

Fig. 5. The P-router data path section algorithm.

P-router. Figure 5 depicts the data path selection algorithm invoked for every I/O call to inspect the viability of a P2P transfer. For P2P to be performed, all of the following conditions must be satisfied: (a) The provided memory buffer is in GPU memory. (b) The buffer and the file offsets are properly aligned. (c) The storage device that stores the file supports P2P. (d) Read-ahead is not required and the request cannot be served from the page cache in full. If any of the conditions fails, then the request is served via CPU-mediated I/O. These conditions are checked by the respective components whose implementation we describe next.

Address Classifier. The classifier checks within the Virtual Memory Area (VMA) of the calling process to determine whether a virtual address is mapped to a BAR of an I/O device. The classifier keeps track of all the virtual address ranges for different GPU memory regions. For each range the classifier generates a random token used in the address tunneling mechanism, which we discuss below.

File Offset and Buffer Alignment. The file offset and the address of the memory buffer provided in the I/O request must be aligned on 512 bytes boundary to allow P2P, because NVMe requests must be aligned on the SSD sector boundaries [9]. While there are several possible ways to work around these constraints, our current implementation falls back to CPU-mediated I/O for the whole buffer. We leave optimizing misaligned transfers to future work.

Device Checker. SPIN requires all P2P-capable storage devices to register with the SPIN driver via a kernel-exported registration function. The driver uses this information to determine whether the I/O request can be served via P2P.

I/O Request Execution. Both CPU-mediated I/O and P2P are performed using standard `vfs_read` or `vfs_write` calls. CPU-mediated I/O simply re-invokes the original unmodified I/O request on VFS. For P2P, we use O_DIRECT access mode while taking advantage of the GPU address tunneling mechanism (transition 3.a in Figure 3).

GPU Read-ahead. The GPU read-ahead cache is implemented as a linked list that references 512 pages (tunable), located in the OS page cache. The eviction is policy is LRU. Pages accessed by a CPU program are simply removed from the list, and are not evicted from the OS page cache itself.

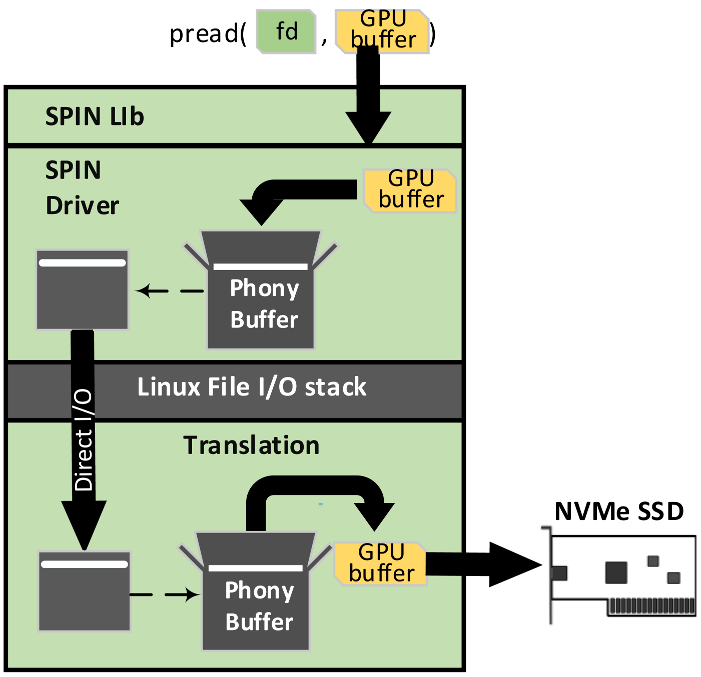

Fig. 6. Address tunneling for direct disk I/O with GPU buffers.

5.3 Address Tunneling

Our implementation of P2P takes advantage of the direct disk I/O file interface, adding a special mechanism to enable its use with GPU memory buffers.

Direct disk I/O and P2P pursue the same goals: They allow direct access to storage devices while bypassing the OS page cache. P2P requires direct access to the storage device that transfers data to/from GPU memory while bypassing the CPU page cache. Using direct disk I/O mechanisms for P2P has a number of advantages. First, the file I/O stack performs the standard file offset-to-LBA mapping, which is compatible with virtual block layers, e.g., software RAID. Second, the mechanism already implements various optimizations, e.g., uses multiple submission queues and merges/splits block I/O requests. Last, it already handles the data consistency by writing back dirty page cache pages in the range of its I/O request. Specifically, when a file is opened with an `O_DIRECT` flag, the I/O request bypasses the page cache, and directly transfers data between the storage device and the user buffer.

Unfortunately, direct disk I/O requires the user buffers to reside in CPU physical memory, and cannot accommodate CPU-mapped GPU buffers. This is because it pins user buffers in memory to perform DMA to/from the storage device, and fails to pin GPU buffers.

To overcome this limitation without major modifications to the Linux kernel, we design a simple mechanism that we call *address tunneling*, which delivers the GPU address through unmodified VFS stack and block layers down to the generic NVMe driver.

Figure 6 explains the basic idea. We allocate a special user-space *phony buffer* in the CPU, which is used as an envelope for the GPU buffer address. The phony buffer is then passed to a VFS file I/O call, instead of the original GPU buffer. Therefore, it successfully undergoes all the translation and pinning process while passing through intermediate I/O layers. When the envelope reaches

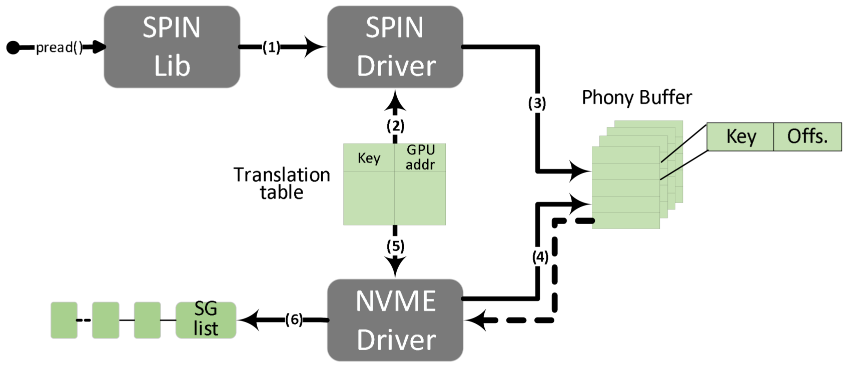

Fig. 7. Address tunneling mechanism. (1) Virtual address of the GPU buffer received from the SPIN library. (2) The SPIN driver locates the key in the translation table according to the virtual address (3) and stores it in an entry in the phony buffer. (4) The NVME driver fetches the key from the phony buffer (5) and locates the GPU address in the translation table according to the key. (6) The NVME driver invokes DMA with the GPU address.

the generic NVMe driver, the driver retrieves the address of the GPU buffer and uses this address to perform P2P.

When SPIN needs to invoke P2P, the system stores real GPU addresses in the phony buffer, in an encoded form. The pointer to the phony buffer is passed to the respective VFS call instead of the GPU address of the real buffer. The phony buffer must be at least as large as the I/O request. Therefore, I/O requests that exceed the size of the phony buffer are split into multiple smaller requests. Each sub-request serves 4MB of data, which is large enough to amortize transfer invocation overheads.

Security of Address Tunneling. One potential problem with the tunneling mechanism is that phony buffers are allocated in the user-space memory (otherwise they cannot be passed to VFS calls), hence they are accessible to user-space programs and can be overwritten by an adversary to potentially hold any physical CPU address, thereby enabling DMA attacks.

SPIN, therefore, does not store the actual GPU addresses in phony buffers. Instead, it first creates an 8-byte temporary pseudo-random token (③ in Figure 7), and stores it within the phony buffers. The token serves as a key in the translation table, which holds all the VMAs mapped to GPU BARs. Thus, the NVMe driver that receives the phony buffer uses this token to fetch the appropriate VMA from the translation table ⑤, and it retrieves the relevant GPU memory addresses to be used in P2P.

Using a phony buffer for Tunneling GPU Addresses. Each memory page of the phony buffer (4KB) is used to store the respective address of the GPU buffer. For example, if there is a read request of size 8K, the first phony buffer page will hold the base address of the destination buffer, whereas the second one will hold the address of the base with 4K offset. Using only the first page of the phony buffer for the entire request does not work, because the block layer may re-order the requests and split them into smaller chunks, and the LBAs embedded in the request structure will get lost.

Multithreading. Multiple threads in the same process may use the same phony buffer simultaneously in a lockless manner. One thread consumes only 16 bytes per GPU address in a 4K page; therefore, a single phony buffer may accommodate up to 250 concurrent requests from different threads. To avoid contention, SPINDRV allocates a separate phony buffer slot for each new thread calling in LIBSPIN. The slot offset is stored in a thread-to-slot table (not shown) and can be retrieved by using the thread PID as the key.

Interaction with NVMe Driver. The phony buffer's pages are marked by setting an unused (for user mode) `arch_1` flag in their `page_struct` to differentiate them from regular pages. Block requests that are determined to be P2P transfers go through a translation mechanism, which translates the contents of the phony buffer to GPU addresses used in P2P.

Cleanup. When a thread terminates, the relevant entries from the driver's internal data structures are removed using the `mmu_notifier` function call hook. For the process, its phony buffer gets unpinned from memory, its `arch_1` flag unset, and the allocated phony buffer freed.

Implementation Complexity. SPINDRV is implemented in 700 LOC and LIBSPIN just 30 LOC. We modified 10 LOC in the Linux generic NVMe driver to detect phony buffers and extract respective GPU addresses.

5.4 Limitations

Supporting `pwrite().` Mapping GPU memory into the process's address space is a recent capability that is not yet well supported in current systems. Specifically, CPU reads from that memory mapping are about two-three orders of magnitude slower than CPU writes [21], i.e., about 30MB/s and 70MB/s for NVIDIA and AMD GPUs, respectively. Therefore, while reading data from the page cache into the GPU is fast, writing files from the GPU into the page cache—which might be beneficial e.g., for buffering writes in CPU memory—results in severe performance degradation. Therefore, we currently configure SPIN to perform writes from GPU memory only via P2P, while taking care of data consistency.

Changing Linux to Natively Support GPU Buffers. The address tunneling mechanism sidesteps the problem of passing GPU buffers to direct disk I/O, but why not changing the kernel in the first place? Technically, the problem originates in the use of `struct_page`, which is not available for I/O re-mapped addresses such as GPU memory buffers. However, this struct is required by the block layer. Attempts have been made to solve the problem in a systematic way [22], yet they require major changes to the kernel, touching over 100 files of kernel code. We therefore choose a more conservative solution.

6 EVALUATION

We evaluate SPIN on two hardware systems (Table 1). We disable HyperThreading and configure the frequency governor to high performance to reduce overall system noise. Both machines run Ubuntu 15.04 with and untainted Linux kernel 3.19.0-47 and ext4 on SSD. We use CUDA 7.5 for NVIDIA and OpenCL 2.0 for AMD.

Methodology. We run each experiment 11 times, omit the first result as a warmup, and report the average of the last 10 runs. We explicitly flush the contents of the page cache before each run (unless stated otherwise). We observe the standard deviation below 1% across all the experiments and do not report it in the figures.

Alternative Transfer Methods. We compare SPIN with several different implementations described in Table 2. We note that the implementation where `pread` () is invoked with the CPU-mapped GPU buffer (last row) has not been evaluated in prior works.

Alternative Implementations of P2P. Although several prior works reportedly implement P2P between SSDs and GPUs [4–8], we found only the early prototype of Project Donard [8] to be

Table 2. Transfer Mechanisms Used for Evaluation

ClWrite	Regular `read` into the CPU, followed by a blocking `clEnqueueWriteBuffer/cudaMemcpy` call to the GPU.
ClWrite+D	Same as ClWrite but with bypassing the CPU page cache via O_DIRECT flag.
P2P	SPIN's implementation of P2P that bypasses the page cache.
pread+GPU	`pread` into the GPU memory that is mapped to the process's address space. Unlike SPIN, `pread` ()+GPU always uses the page cache. Not evaluated in prior works.

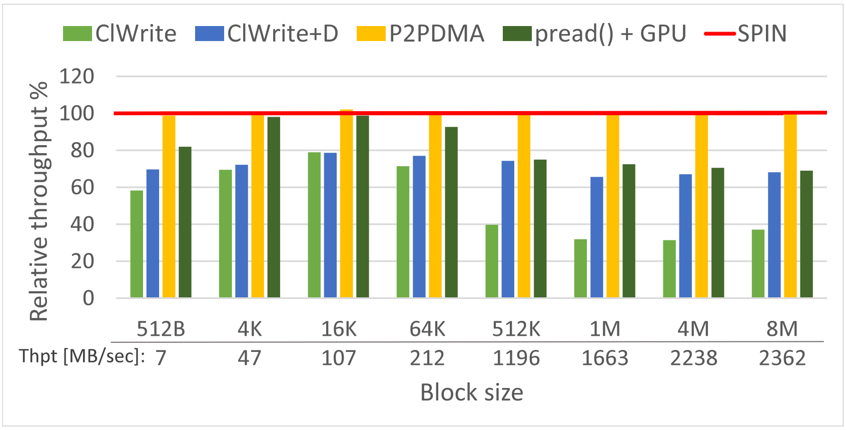

Fig. 8. Random reads, one thread.

publicly available. However, this prototype is limited and is slower for all request sizes, particularly for shorter requests; therefore, we do not include it in the experiments.

6.1 Microbenchmarks

6.1.1 Threaded IO Benchmarks. We use TIOtest [16] for our benchmarks. TIOtest is a standard tool for evaluating file I/O performance in CPU-only systems. It supports multi-threading (each thread accesses its own file), sequential/random access patterns, and different I/O request sizes. We modify the original code[1] to read data into GPU buffers using all the five evaluated implementations. For SPIN our changes required modifying 10 LOC for buffer allocation, the rest of the code is unchanged.

We report the results for the AMD GPU and discuss the performance of the NVIDIA GPU in the text.

Random Reads. In this experiment, each worker thread reads 500 blocks at random offsets from a 50GB thread-private file. Figures 8 and 9 show the relative throughput of the different transfer methods compared to SPIN performance for one and four threads, respectively. Note that the drops in the relative throughput on the graphs do not imply lower absolute throughput, rather they mean slowdown compared to SPIN in the respective configuration.

SPIN performance matches the one of P2P, adding only 1% overhead. For blocks above 1MB the overhead of additional memory copy in CPU memory gets amortized for all the implementations but ClWrite, because of its second extra copy in the temporary CPU buffer.

[1]https://wiki.codeaurora.org/xwiki/bin/Linux+Filesystems/Tiobench.

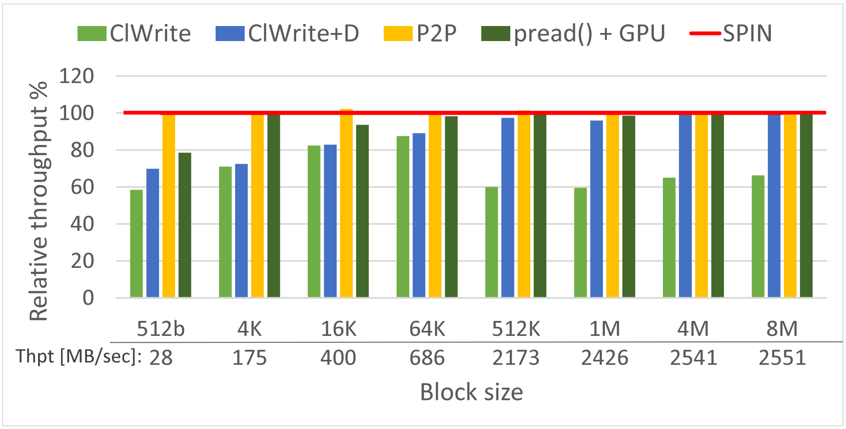

Fig. 9. Random reads, four threads.

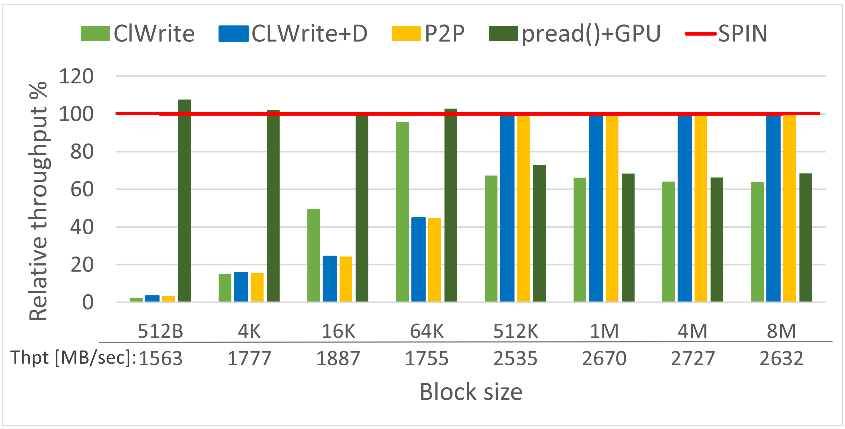

Fig. 10. Sequential reads, four threads.

Sequential Reads. For sequential reads, each worker thread in TIOtest reads an entire file of 100MB. Figure 10 shows that SPIN tracks the best performing method for the specific block size, switching from page cache to P2P at 512KB as explained in Section 4.3.2. We observe that for blocks smaller than 4K SPIN experiences higher relative overhead of up to 10%, because it serves them from the page cache. The overhead is amortized for larger reads, however.

Sequential/random Writes. For the sequential writes, each worker thread writes a 100MB file. Figure 11 shows that the `pwrite` +GPU mechanism is dramatically slower than P2P, as we explain in Section 5.4, therefore SPIN always performs aligned writes via P2P. Random writes perform similarly.

Performance on NVIDIA and AMD GPUs. SPIN achieves 5–10% higher throughput on AMD R9 GPU than on NVIDIA K40C GPU, while the overall behavior is similar. We find that cudaMemcopy might be slower than AMD ClWrite, and the GPU BAR writes for NVIDIA GPUs are slower for some block sizes. These results indicate that SPIN works well with GPUs from different vendors; however, the small performance gap we observe requires further investigation.

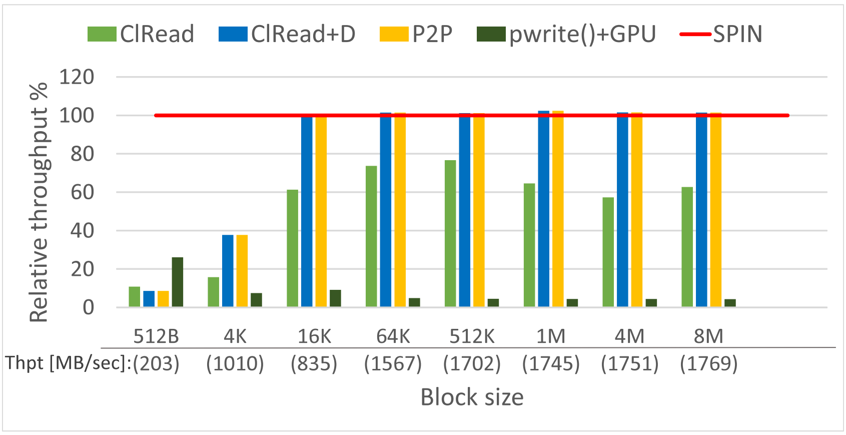

Fig. 11. Threaded IO write benchmark using one thread sequential write.

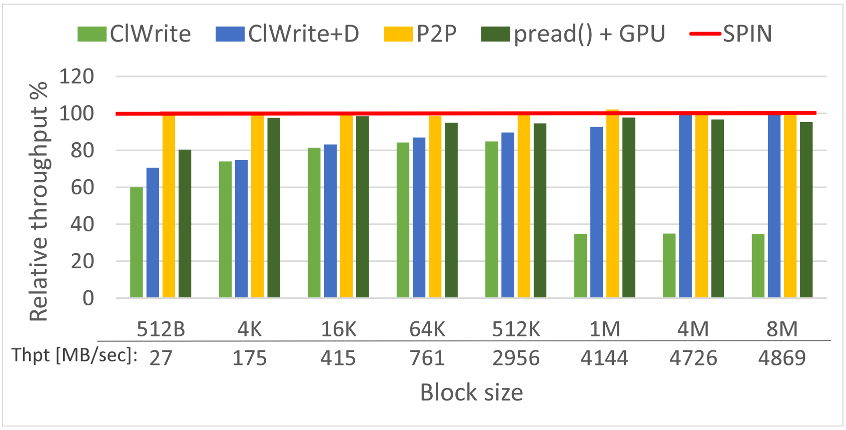

Fig. 12. RAID: Random reads, four threads.

Software RAID-0. We use the standard mdadm Linux utility to create a RAID-0 (striping) volume over two NVMe SSDs. In this configuration, the stored data is split between two SSDs according to the configured stripe size (512KB in our configuration), thus performing larger file accesses in parallel.

Figure 12 shows the relative throughput of random accesses for which SPIN always uses P2P. The main performance benefits are due to bypassing the page cache—pread ()+GPU performance eventually reaches the maximum throughput as well. RAID-0 outperforms a single SSD only for large reads (above 512KB). This is due to extra overheads of additional processing in the RAID layer, which get amortized for larger blocks. For large sequential reads, SPIN achieves a throughput of 5.2GB/s. The higher bandwidth is due to the SSDs performance characteristics.

Maximum Sequential Read Throughput. We compare the maximum achievable throughput over different transfer mechanisms. The test performs sequential reads from four threads, 8MB per read from a 4GB file, when a file is *prefetched into the page cache*. Table 3 shows the results. SPIN is faster than all the transfer methods that do not use page cache, and faster than ClWrite that does. SPIN's overhead in this scenario is 1.5%.

Table 3. Max Read Throughput (GB/s)

SPIN	pread + GPU	P2P	P2P + RAID	ClWrite	CLWrite+ D+RAID
10.13	10.28	2.65	5.29	5.72	4.69

File in page cache.

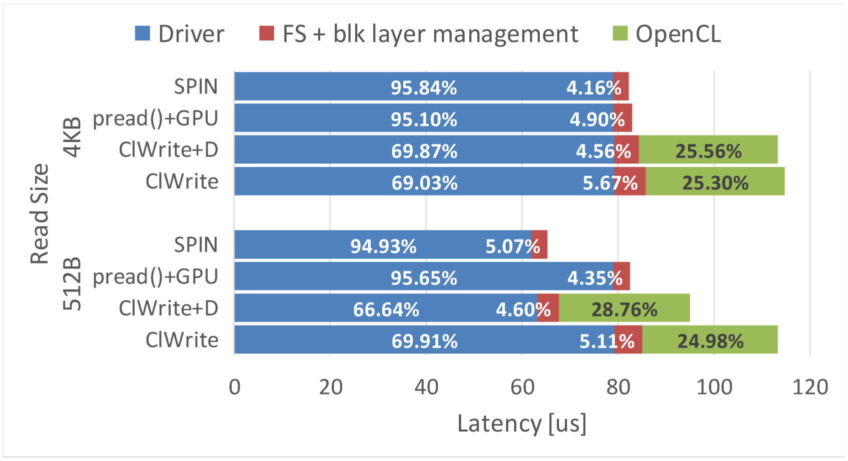

Fig. 13. Latency breakdown of pread via SPIN.

Latency Breakdown. Figure 13 shows the breakdowns of different transfer methods for a single read of 512B and 4K to SSD to measure the absolute time added by SPIN to each I/O request. We use `blktrace` to measure times in the file system layers. Specifically, we observe the D2C (driver and device time) reported by `btt`, which is the average time from when the actual I/O was issued to the driver, until it is completed and passed back to the block layer. This is denoted by "driver" in the graph. To deduce the time spent in the file system and the block layer management, we subtract the D2C time from the overall transfer time. This is denoted by "FS+ block layer management" in the graph. We measure the time for OpenCL and the total transfer time by making use of a timing function inside our microbenchmark, which generates the reads.

The requests originate from a random offset in a file and destined to a GPU buffer. In this setup, SPIN will utilize P2P. The data does not reside in the page cache.

The latency of retrieving the data from the SSD is depicted by the time it takes to process a block request by the driver. `pread` ()+GPU and ClWrite always read chunks of 4K, since they do not use direct I/O. The reason being that a non Direct I/O request always passes via the page cache, which imposes a granularity of 4K. The penalty for transferring a page instead of the request's original size can be seen in the Driver time. We can also derive that while performing small requests, the majority of the time is spent on accessing the device itself rather than transferring data, as the ratio between a Direct I/O 512B read is nearly 0.75× the time it takes to read 4K. We can also see from the breakdown that the additional logic added inside the NVMe driver for SPIN does not incur in any noticeable overhead, when comparing the Driver time between SPIN and ClWrite+D for both 512B and 4K reads. The FS+blk layer management segment includes the additional logic's execution time added by SPINDRV and LIBSPIN. Compared to the other data transfer methods, which utilize Direct I/O, the additional logic's overhead stands at most 0.5% of the total FS+blk layer management time. The additional data copy overhead from page cache to CPU/GPU buffer however, is noticeable, resulting in a 0.8× slowdown.

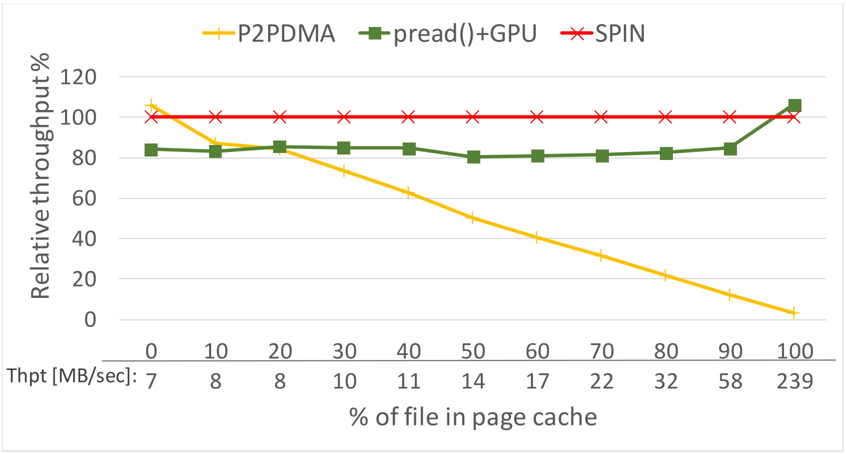

Fig. 14. Random access performance for different page cache occupancy. Reading blocks of 512B.

Transfer methods utilizing ClWrite incur a significant time penalty over the other methods. The graph shows that the time it takes to transfer 512B with ClWrite is similar to that of 4K, which suggests that the minimum DMA transfer size of OpenCL is at least 4K.

6.1.2 Effect of the Page Cache on Read Throughput. The goal of this experiment is to show potential performance gains for producer-consumer workloads, which may utilize both the CPU and GPU while they access a shared file. We prefetch different portions of a 40GB file into the page cache using `vmtouch`,[2] and we run TIOtest for 512B random reads.

Figure 14 shows the relative throughput of SPIN compared to P2PDMA and pread(). Not only does SPIN track the best alternative for the majority of page cache residence values, it is *faster* than the fastest among them by up to 20%. That is because it combines both page cache and P2P, dynamically choosing between them per request depending on the residence in the page cache (discussed in (Section 4.3.1)). SPIN is slightly slower on the extremes due to the 5% overhead it introduces in these scenarios.

6.1.3 SPIN Performance Under CPU and I/O Load. We execute the same experiment as in Figure 14 but now impose heavy load on all the CPUs or SSD in parallel with the benchmark. The benchmark performs 512KB random reads (cutoff size for reading from the page cache), to show the worst-case scenario for SPIN under CPU load. We use `stress-ng`[3] benchmarking tool. Figure 15 shows the relative throughput for 0%, 50%, and 100% file residency in the page cache, with and without CPU or SSD load. We observe that SPIN retains its performance advantages regardless of the system load.

6.2 Application Benchmarks

6.2.1 Aerial Imagery Rendering. GPUs are commonly used for rendering aerial imagery in *geographic information systems (GIS)*. The datasets used in such systems may grow to hundreds of GBs.

Large rasters are split into tiles to shorten system response time. The rendering engine reads the tiles from a file depending on the view point and stitches them together.

[2]https://hoytech.com/vmtouch/.
[3]https://openbenchmarking.org/test/pts/stress-ng.

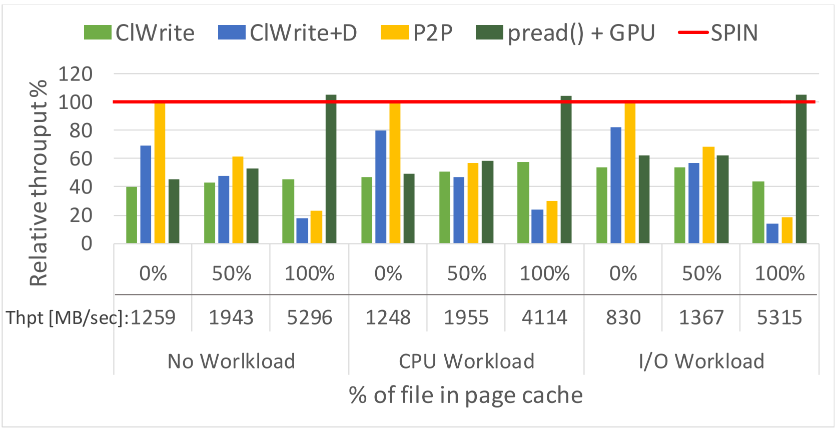

Fig. 15. Random 512KB reads, in parallel with CPU / I/O workloads.

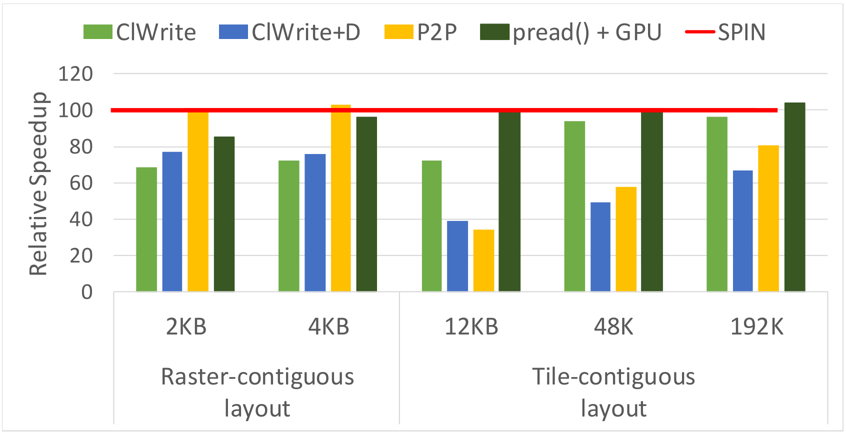

Fig. 16. Aerial imagery benchmark throughput relative to SPIN for different file layouts. Higher is better.

In our evaluation, we generate I/O traces via a benchmarking tool for web-based rendering engines [23]. We use TrueMarble dataset [24] from standard benchmarks [23], which is a 190GB multi-raster of the Earth; each raster corresponds to a different image resolution.

The actual file access pattern in this application depends on the underlying file layout. There are two layouts: (1) raster-contiguous layout, where the whole raster is stored as a 1D vector in the file (illustrated in Figure 17, file A), and (2) tile-contiguous layout, where each *tile* is a 1D vector and the raster is composed of many 1D tiles (illustrated in Figure 17, file B). The first layout results in mostly random accesses 2-4KB each, whereas the second involves mostly sequential accesses each from 12 to 192KB. We emphasize that the rendering applications must be able to accommodate files with both layouts.

To generate the trace, we randomly choose the target image resolution and the view region, derive the tiles to render that region and record their respective offsets in the dataset file.

We use tiles of sizes ranging from 64×64 pixels up to $1{,}024 \times 1{,}024$ pixels. In every trace, we emulate rendering of 1,000 different regions in full HD.

We generate the traces for different input layouts and compare the throughput of different transfer mechanisms. As Figure 16 shows, *the choice of the transfer mechanism depends on the layout in*

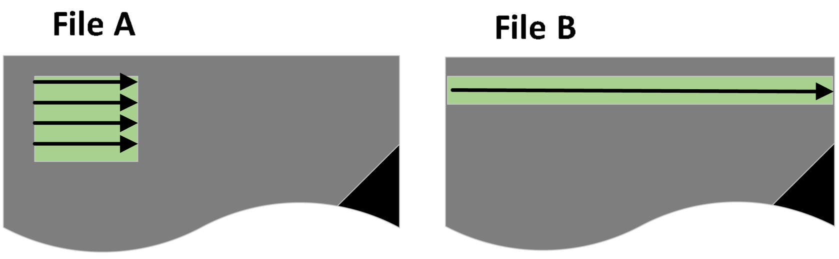

Fig. 17. Map file layouts. File A is raster-contiguous, and File B is tile-contiguous.

Table 4. Log Server Throughput (in MB/s), CPU Utilization, and Speedup Over the CPU-only Version

Configuration		CPU	GPU		
			P2P	`ClWrite()`	SPIN
R-time	Thput	771	594 (0.8×)	1,921 (2.5×)	1,950 (2.5×)
	CPU util	79.5%	3%	11.8%	10.7%
Offline	Thput	634	2,549 (4×)	1,822 (2.9×)	2,550 (4×)
	CPU util	70.3%	8.5%	12.3%	8.5%

use. For the native layout with mostly random access pattern, P2P and SPIN achieve the highest throughput. However, for tiled layout the reads are mostly sequential, and SPIN benefits from the read-ahead achieving up to 2.5× higher throughput than P2P for 12K reads. SPIN eliminates the need to manually perform such low level optimizations, reducing code complexity and development efforts.

6.2.2 *GPU-accelerated Log Server.*

Log servers, such as VMWare VRealize [25], are commonly used in distributed systems for centralized storage and processing of logs from multiple servers.

Log processing usually involves string and regular expression matching, which may benefit from acceleration on GPUs [26].

We implement a simple log server, which receives log files over the network, stores them locally in files, and scans them for suspicious IPs from the list provided by the user. As is common in log processing systems, e.g., Fail2Ban [11], log analysis is performed in a separate scanner process that reads the specified log file and processes it. Such a modular design is convenient, because it enables us to easily extend the analysis using several independent backends. Our implementation of the scanner offloads the string matching to a GPU. The application spends about 30% of its runtime in string matching procedure and the rest in data transfers.

We measure the maximum system throughput of the incoming log data our system can handle, which depicts the overall application performance. We measure the throughput in two scenarios: (1) real time, in which the scanner is invoked each time the files get updated (using `inotify` interface), (2) offline, in which the scanner is invoked on a specific log file to be processed as a whole. In both configurations, a total of 80GB of data is processed.

We evaluate our GPU implementation with different I/O mechanisms: (1) traditional pread() followed by `ClWrite()` to GPU memory, (2) P2P (3) SPIN. We also implement a CPU-only version that uses Intel's Threading Building Blocks and runs on six cores.

Table 4 shows that in the *real time* scenario SPIN achieves the highest throughput among all other I/O methods. Since the system triggers log processing right after it receives log file updates from the network, the new contents have not yet been written back to the disk and reside entirely

Table 5. Speedups of Pread() + GPU and SPIN Over the Original Version of GPUfs

Data transfer to GPU mechanism	Input image size		
	3MB	12MB	48MB
pread() + GPU	0.66×	0.66×	0.71×
SPIN	1.29×	1.25×	1.26×

in the page cache. SPIN, therefore, reads the data from the page cache, relieving I/O contention on the SSD, which do occur in P2P configuration. In the steady state, the system throughput is limited by the maximum SSD write throughput, because the network server keeps writing the updates to storage, eventually exhausting the page cache space.

In the *offline* scenario, the data is not in the page cache; therefore, SPIN switches to use P2P.

In this application, complex interactions between multiple processes dynamically create file data reuse opportunities that cannot be known in advance, hence they are hard to leverage without the OS support. SPIN re-enables the standard OS ability to handle such opportunistic reuse automatically for file transfers to the GPU.

6.2.3 Image Collage. The image collage application [15] creates an image collage by replacing blocks in the input image with "similar" tiny images from a data base (we use Reference [27]). Pre-processed tiny images are stored in a file of size 38GB. We use an open-source implementation that uses GPUfs [18] GPU-side library for accessing files from GPU kernels. GPUfs uses a dedicated worker thread running on the CPU to handle the file transfers into the GPU memory. This application performs mostly random reads of 512B each.

The original version of GPUfs first reads the file contents into the host staging area, and then copies the data into another staging area in GPU memory via `cudaMemcopy`. GPUfs has been optimized to perform multiple transfers concurrently with the GPU kernel execution. This application spends about 40% in processing the image, and the rest in data transfers. We remove the staging area in the host, and allocate the staging area in the GPU memory, using the NVIDIA GDRCopy module to map that memory to the CPU, changing in total 30 LOC.

We measure the SPIN speedup over the unmodified version for a single image collage. For three different input images of 3MB, 12MB, and 48MB SPIN is $\times 1.27 \pm 0.02$ faster on average thanks to the use of P2P for short random reads. Table 5 shows the individual results.

7 RELATED WORK

Our work relates to a variety of topics in computer architecture, operating system design, compilers, concurrent algorithms, and GPU computing.

System Support for P2P. There have been several works that enable P2P between NVMe SSDs and GPUs, but SPIN is the first to integrate P2P with the OS file I/O, dealing with page cache, read-ahead, data consistency, and compatibility with virtual block devices.

GPUDrive [6] is a system for processing streaming I/O-intensive GPU workloads based on an all-flash storage array connected to the GPU.

The authors observe that P2P is necessary to bridge the gap in the file I/O performance between the GPU and the CPU. However, it targets streaming workloads in which bypassing the CPU page cache is desirable.

NVMMU [4] introduces a special programming model and runtime for P2P with GPUs. NVMUU shows that P2P achieves high performance with standard GPU compute benchmarks modified to read input data from files. Unlike SPIN, however, it requires a custom interface for P2P, does not address the page cache integration issues, and focuses only on GPU-only applications with large sequential reads. In fact, it shows that P2P is slow for small I/O requests but does not address this problem.

Project Donard [8] was among the first to support P2P via a low level driver interface. Among its many limitations, it runs only with root privileges due to direct access to NVMe DMA and suffers from performance issues. Gullfoss [5] software framework for P2P offers a non-standard API for P2P-based file accesses. Therefore, it differs from SPIN in similar aspects as NVMMU.

Morpheus [7] enables P2P to GPUs from SSDs, but focuses on data processing on embedded CPUs in NVMe SSD. Similar to Gullfoss, Morpheus does not address the challenges of integrating P2P into standard file I/O, focusing primarily on low level P2P functionality.

GDRcopy [21] uses CPU-mapped regions of GPU memory for efficient data transfers to GPUs. SPIN leverages the same functionality.

P2P *Technologies.* Recent GPUs offer support for P2P, including GPUDirectRDMA [28] from NVIDIA and DirectGMA [3] from AMD. These technologies provide generic support for direct access to GPU memory from PCIe devices, but they do not integrate it into higher level services like file I/O.

System Abstractions for GPUs. GPUfs and GPUnet [10, 18, 29] provide file access and networking directly to GPU programs. The current work is complementary as it simplifies the use of P2P for *CPU* programs.

8 CONCLUSION AND FUTURE WORK

SPIN [4] focuses on the fundamental problem of providing generic OS abstractions in heterogeneous systems, extending the traditional I/O mechanisms to systematically deal with direct I/O into the GPU. We show the importance of tighter integration of P2P with the file I/O stack, expose the challenges associated with the use of P2P together with the page cache and read-ahead, and devise a practical solution that outperforms the state-of-the-art in a range of realistic scenarios.

One aspect that SPIN does not currently address is how to leverage data reuse in workloads with complex, non-sequential data access pattern from the GPU. Specifically, SPIN switches to P2P once it detects such a non-sequential access pattern, thereby bypassing the OS page cache. As a result, recurrent accesses to previously used pages will require accessing SSD again. In contrast, in CPU-only workloads, any access to files is cached in the page cache, regardless of the access pattern. A temporary solution might be to *broadcast* the data read from the SSD over PCIe to both the CPU page cache and the GPU memory, thereby retaining P2P's advantages while enabling data reuse. However, a more systematic way to address this issue requires extending the page cache into the GPU memory, which is not currently practical (Section 4.1).

We believe, however, that the future hardware trends toward systems with multiple accelerators and I/O devices [30, 31] will dictate the OS architecture with distributed page cache introduced in GPUfs [18], whereby each accelerator will host a page cache in its private local memory. In such a system, each computing and I/O accelerator might have different performance characteristics when it comes to data transfers (for example, SSDs are more efficient for larger data requests and have a relatively high access latency to data). Therefore, the ideas for scheduling data transfers introduced in SPIN will be applicable to the implementation of P2P transfers across different devices while serving a single I/O request.

[4]SPIN is available at https://github.com/acsl-technion/spin.

More generally, SPIN serves as an example of a system showing that OS support for increasingly heterogeneous systems must extend beyond low-level APIs and provide the convenience of high-level OS abstractions to achieve their performance potential.

REFERENCES

[1] AMD Radeon Pro SSG Set to Transform Workstation PC Architecture, and to Shatter Real-Time Visual Computing Barriers. Retrieved on February 7, 2017 from http://www.amd.com/en-us/press-releases/Pages/amd-radeon-pro-2016jul25.aspx.

[2] GPUDirect RDMA. Retrieved on February 7, 2017 from http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[3] Tech Brief: AMD FireProTM SDI—Link and AMD DirectGMA Technology. [n.d.] Retrieved from https://www.amd.com/Documents/SDI-tech-brief.pdf.

[4] 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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*. IEEE, 13–24.

[5] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2015. Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources. *Technical Report CS2015-1015, Department of Computer Science and Engineering, University of California, San Diego.*

[6] Mustafa Shihab, Karl Taht, and Myoungsoo Jung. 2014. GPUDrive: Reconsidering storage accesses for GPU acceleration. In *Proceedings of the Workshop on Architectures and Systems for Big Data.*

[7] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 53–65.

[8] Project Donard. Retrieved on February 7, 2017 from https://github.com/sbates130272/donard.

[9] NVM Express 1.0e. Retrieved on February 7, 2017 from http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf.

[10] Sangman Kim, Seonggu Huh, Xinya Zhang Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX, 6–8.

[11] Fail2Ban. [n.d.] Retrieved from www.fail2ban.org/.

[12] MDADM—Manage MD Devices AKA Linux Software RAID. [n.d.] Retrieved from https://www.kernel.org/pub/linux/utils/raid/mdadm/.

[13] Anandech. 2016. AMD Announces Radeon-Pro SSG. Retrieved from http://www.anandtech.com/show/10518/amd-announces-radeon-pro-ssg-fiji-with-m2-ssds-onboard.

[14] ArcGIS for Desktop. [n.d.] Retrieved from http://desktop.arcgis.com/en/arcmap.

[15] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: A case for software translation on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*. IEEE, ACM.

[16] Threaded I/O Tester. [n.d.] Retrieved from https://sourceforge.net/p/tiobench.

[17] GPU Support in Apache Spark and GPU/CPU Mixed Resource Scheduling at Production Scale. Retrieved on February 7, 2017 from http://www.spark.tc/gpu-support-in-spark-and-gpu-cpu-mixed-resource-scheduling-at-production-scale/.

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

[19] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choil, Sungroh Yoon, and Jaehyuk Cha. 2013. Vssim: Virtual machine-based SSD simulator. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 1–14.

[20] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory-based solid-state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE, 266–277.

[21] A Fast GPU Memory Copy Library Based on NVIDIA GPUDirect RDMA Technology. Retrieved on February 7, 2017 from https://github.com/NVIDIA/gdrcopy.

[22] Evacuate Struct_page from the Block Layer. Retrieved on February 7, 2017 from https://lwn.net/Articles/636968/.

[23] FOSS4G Benchmark. [n.d.] Retrieved from https://wiki.osgeo.org/wiki/FOSS4G_Benchmark.

[24] True Marble. [n.d.] Retrieved from http://www.unearthedoutdoors.net/global_data/true_marble/.

[25] VMWare. [n.d.] vRealize Log Insight. Retrieved from http://www.vmware.com/products/vrealize-log-insight.html.

[26] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. 2009. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 265–283.

[27] Antonio Torralba, Robert Fergus, and William T. Freeman. 2008. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 30, 11 (2008), 1958–1970.

[28] Benchmarking GPUDirect RDMA on Modern Server Platforms. Retrieved on February 7, 2017 from https://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/.

[29] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2014. GPUfs: Integrating a file system with GPUs. *TOCS* 32, 1 (2014), 1.

[30] OpenCAPI. Retrieved from http://opencapi.org/.

[31] Cache Coherent Interconnect for Accelerators (CCIX). Retrieved from http://www.ccixconsortium.com/.

Received January 2018; revised October 2018; accepted January 2019