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

Shai Bergman
Technion

Tanya Brokhman
Technion

Tzachi Cohen

Mark Silberstein
Technion

Abstract

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.

SPIN 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, and enables 3.3× higher throughput for a GPU-accelerated log server.

1 Introduction

GPU-accelerated applications often require fast data transfers between the GPU and storage devices. 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.

In order 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 1 shows one such example. For short sequential reads P2P is dramatically slower than CPU-mediated I/O. It performs faster only for reads larger than 512KB. In this scenario, 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. 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 [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 ap-

plication 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.

SPIN is 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, 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 that achieves about 98% of the optimal performance (§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 (§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 (§5.1).

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 2.65 GB/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.

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.

System architecture. We consider a system where the CPU, discrete GPUs, and NVMe SSD are connected via Peripheral Component Interconnect Express (PCIe) bus. The PCIe switch enables fast *peer-to-peer direct memory access* (P2P) between the GPU and the SSD. P2P allows the SSD to transfer data directly to/from GPU memory, bypassing the CPU.

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.

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.

3 Motivation

Prior works [4–8] show that P2P between SSDs and GPUs substantially boosts system performance for popular GPU benchmarks. These applications exhibit stream-

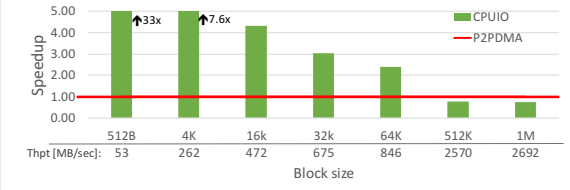


Figure 1: The speedup of CPU-mediated I/O over P2P for sequential reads.

ing 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 5.1. For the hardware setup see Section 6.

Figure 1 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 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, [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.

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.

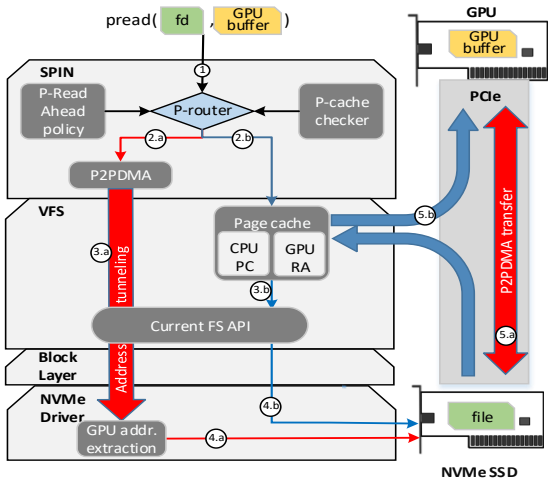


Figure 2: SPIN high-level design and control flow of `pread ()`, as explained in Section 4.2

- **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.
- **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 § 3), but the best way to integrate it with P2P is not obvious. Technically, the prefetcher never runs because P2P bypasses the heuristic 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 2 shows the main design components. SPIN is positioned on top of the Virtual File System (VFS) layer. 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 (§ 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 the Figure) 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 the Figure) of the *CPU* page cache, as described in § 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 § 4.3.1. Finally, it generates VFS I/O requests that are served by a combination of the page

cache (2.b) and P2P (2.a). To invoke P2P via direct disk I/O interface, P-router employs an *address tunneling* mechanism (3.a) described in § 5.1.

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. If the entire requested region is cached, the request is served from the page cache. However, if the cache contains only part of the requested data, 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 (5 pages) with its second, and fourth pages in the page cache. Then, there are 3 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 4KB 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 which can be served from the page cache. For every such a 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 Eq 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 $T_{pc}(s) = \frac{s}{BW_{pc}}$.

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

The greedy heuristic works as follows. For each three consecutive data ranges a, b, c, where b is in the page cache, if $|a| + |b| < S_{cutoff}$, 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 fit the parameters of the transfer time function in Eq 1 using regression. The function fits very well, with the coefficient of determination of over 0.99. We find $S_{cutoff} = 512$KB and $C_{p2p} = 584\mu sec$, 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 which quickly computes the transfer cost of an I/O request, given transfer schedule, using the transfer times measured on real hardware. We validate the simulator experimentally on 5,000 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. We evaluate the schedules on 200,000 random vectors, each representing an 8MB data transfer having different page cache residency patterns. We find that the transfer time of the greedy schedules is within 98.9% of the optimal schedule on average.

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 re-

quests. 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, 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.

libSPIN. is a shim that interposes on standard file I/O calls. The library is loaded via an `LD_PRELOAD` environment variable. Applications that do not load LIBSPIN may share files with those that do.

Interaction with GPUs. SPIN leverages existing tools for mapping GPU memory into the CPU address space. In particular, we use OpenCL's CL_MEM_USE_PERSISTENT_MEM_AMD extension from AMD, and `gdrcopy` module from NVIDIA. Using CPU-mapped GPU memory for I/O enables portability across GPU vendors, interaction with GPUs from kernel space, and independence from GPU software interfaces.

SPINdrv. The driver implements the SPIN design including the page cache and read-ahead as described in § 4. In addition, it introduces a new *address tunneling* mechanism to enable P2P via direct disk I/O which we discuss next.

5.1 P2P via direct disk I/O

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. 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.

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. This problem has no easy solution, as we discuss below (§5.2).

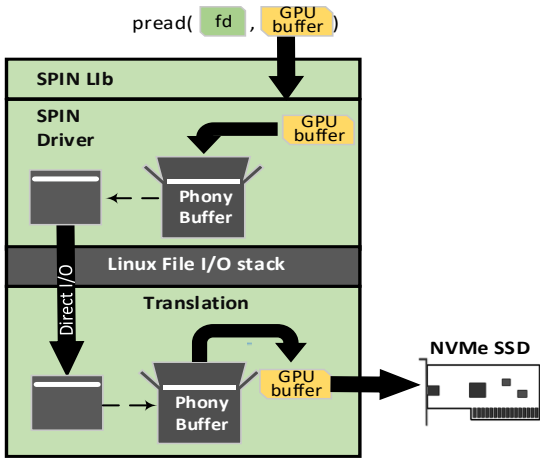


Figure 3: Address tunneling for direct disk I/O with GPU buffers.

Address tunneling. 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 3 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 the generic NVMe driver, the driver retrieves the address of the GPU buffer and uses this address to perform P2P.

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 a temporary pseudo-random token associated with the current request, and uses the token as the key to the kernel-space translation table with the actual GPU addresses.

Implementation details. The phony buffer is a user space CPU memory buffer allocated once during the process invocation when LIBSPIN is loaded. The buffer is pinned in memory and registered with the SPINDRV. Its size remains constant (currently 4MB) throughout the execution. Since an I/O request must fit in the phony buffer, the I/O requests larger than 4MB are split into multiple requests. Each memory page of the phony buffer is used to store the address of one page in the GPU buffer, since the block layer may reorder the requests and split them into smaller chunks.

Multiple threads in the same process may use the same phony buffer simultaneously in a lockless manner. One thread consumes only 16 bytes per a GPU address in a 4K page, therefore a single phony buffer may accommodate up to 256 concurrent requests from different threads.

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.

Interaction with generic NVMe driver. The phony buffer's pages are marked by setting an unused (for user mode) `arch_1` flag in their `page_struct`. This flag is used by the driver to differentiate P2P requests from regular pages and extract the GPU addresses.

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.

5.2 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 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

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

Table 1: Evaluation platforms. Both use one or two Intel P3700 800GB NVMe SSD

ClWrite	Regular `read` into the CPU, followed by a blocking `clEnqueueWriteBuffer / cudaMemcopy` 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.

Table 2: Transfer mechanisms used for evaluation.

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

6.1 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.

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

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. Figure 4a shows the results. Note that the drops in the relative throughput on the graph do not imply lower absolute throughput, rather they mean slowdown compared to SPIN in the respective configuration. The results for a single CPU thread are similar and omitted due space limitations.

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.

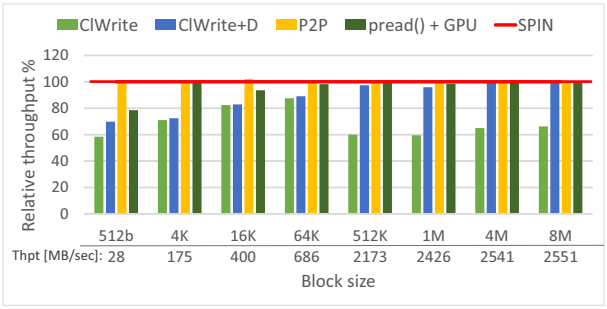
Sequential reads. For sequential reads, each worker thread in TIOtest reads an entire file of 100MB. Figure 4b shows that SPIN tracks the best performing method for the specific block size, switching from page cache to P2P at 512KB as explained in Section 4.3.2. We observe that for blocks smaller than 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. The `pwrite` +GPU mechanism is dramatically slower than P2P, as we explain in Section 5.2, therefore SPIN always performs aligned writes via P2P. Random writes perform similarly. Due to the lack of space, the figure is omitted.

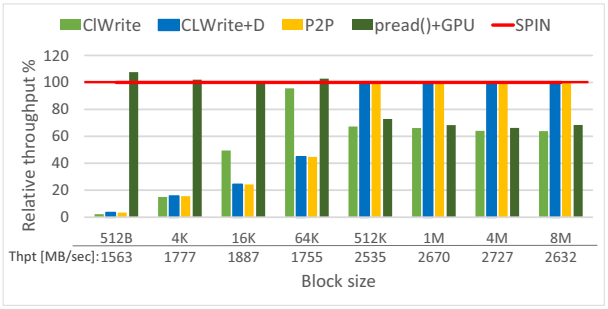
Performance on NVIDIA and AMD GPUs. SPIN achieves 5-10% higher throughput on AMD R9 GPU than on NVIDIA K40C GPU, while the overall behavior is similar. We find that cudaMemcopy might be slower then AMD ClWrite, and the GPU BAR writes for NVIDIA GPUs are slower for some block sizes. These results indicate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

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.

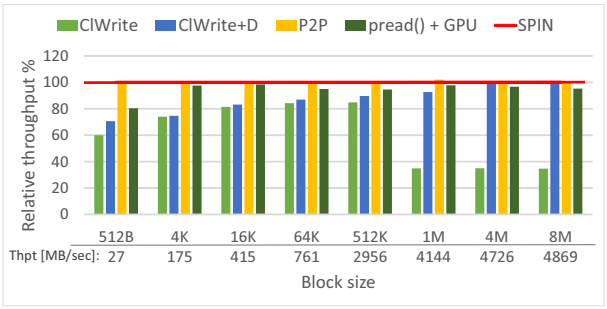
Figure 4c shows the relative throughput of random accesses for which SPIN always uses P2P. RAID-0 outperforms a single SSD only for large reads (above 512KB). This is due to extra overheads of additional processing in the RAID layer which get amortized for larger blocks. For large sequential reads, SPIN achieves a throughput of 5.2GB/s. The higher bandwidth is due to the SSDs performance characteristics.



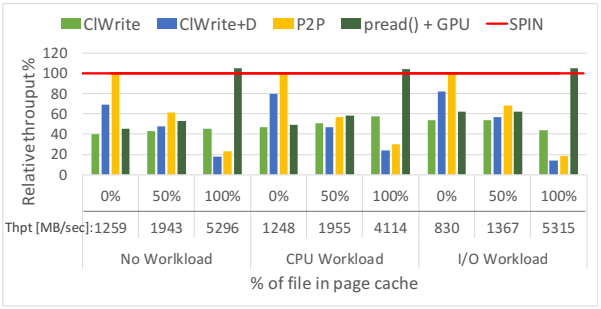
(a) Random reads, 4 threads



(b) Sequential reads, 4 threads



(c) RAID: Random reads, 4 threads



(d) Random 512KB reads, in parallel with CPU / I/O workloads

Figure 4: Threaded IO benchmarks for AMD GPUs.

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

Table 3: Max read throughput (GB/s). File in page cache.

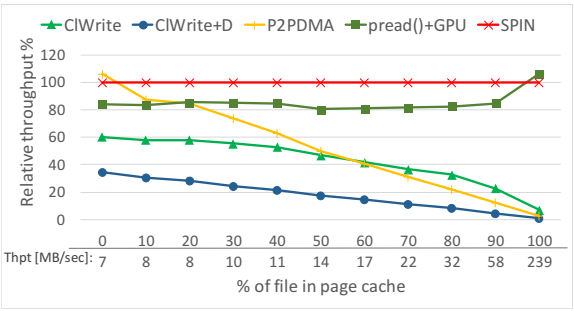


Figure 5: Random access performance for different page cache occupancy. Reading blocks of 512B.

Maximum sequential read throughput. We compare the maximum achievable throughput over different transfer mechanisms. The test performs sequential reads from 4 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%.

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 run TIOtest for 512B random reads.

Figure 5 shows the relative throughput, highlighting the differences between transfer methods. Not only does SPIN track the best alternative, 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 (§ 4.3.1)). SPIN is slightly slower on the extremes due to the 5% overhead it introduces in this scenario. ClWrite results in low performance due to its constant invocation overhead, whose relative weight grows when most requests are served from the page cache, as we also see in Figure 4b.

SPIN performance under CPU and I/O load. We execute the same experiment as in Figure 5, 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.

[2] https://hoytech.com/vmtouch/

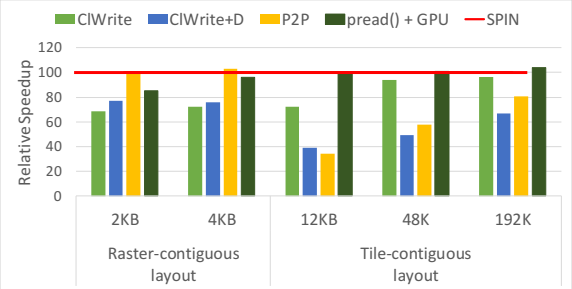


Figure 6: Aerial imagery benchmark throughput relative to SPIN for different file layouts. Higher is better.

We use `stress-ng` [3] benchmarking tool. Figure 4d 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

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 in order to shorten system response time. The rendering engine reads the tiles from a file depending on the view point, and stitches them together.

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 and (2) tile-contiguous layout, where each *tile* is a 1D vector and the raster is composed of many 1D tiles. The first layout results in mostly random accesses 2-4KB each, whereas the second involves mostly sequential accesses each from 12KB 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 64x64 pixels up to 1024x1024 pixels. In every trace we emulate rendering of 1000 different regions in full HD.

We generate the traces for different input layouts and compare the throughput of different transfer mechanisms. As Figure 6 shows, *the choice of the transfer mechanism depends on the layout in 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.

[3] https://openbenchmarking.org/test/pts/stress-ng

Configuration		CPU	GPU		
			P2P	`ClWrite()`	SPIN
R-time	Thput	771	594 (0.8×)	1921 (2.5×)	1950 (2.5×)
	CPU util	79.5%	3%	11.8%	10.7%
Offline	Thput	634	2549 (4×)	1822 (2.9×)	2550 (4×)
	CPU util	70.3%	8.5%	12.3%	8.5%

Table 4: Log server throughput (in MB/s), CPU utilization and speedup over the CPU-only version

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 to easily extend the analysis using several independent backends. Our implementation of the scanner offloads the string matching to a GPU.

We measure the maximum system 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 6 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 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 net-

work 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 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.

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 [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 512B each.

The original version of GPUfs first reads the file contents into the host staging area, and then copies the data into GPU memory via `cudaMemcopy`. We remove the staging area in the host, and allocate the staging area in the GPU memory, changing in total 30 LOC.

We measure the SPIN speedup over the unmodified version. 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.

7 Related work

System support for P2P. There have been several works which 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.

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 shares many conceptual similarities with NVMMU, and hence many of its limitations. Morpheus [7] enables P2P to GPUs from SSDs, but 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 Conclusions

SPIN 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 which outperforms the state-of-the-art in a range of realistic scenarios.

Current hardware trends are toward systems with multiple accelerators [30, 31], which will dramatically increase system heterogeneity and complicate software development. OS support for such increasingly heterogeneous systems must extend beyond low-level APIs, and provide the convenience of high level OS abstractions to achieve their performance potential. SPIN is a step in this direction.

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

Acknowledgements

Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14), and the Israeli Ministry of Economics via HiPer consortium.

References

[1] "AMD Radeon Pro SSG Set to Transform Workstation PC Architecture, and to Shatter Real-Time Visual Computing Barriers."

http://www.amd.com/en-us/press-releases/Pages/amd-radeon-pro-2016jul25.aspx, 2016.

[2] "GPUDirect RDMA." http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2015.

[3] "Tech Brief: AMD FirePro™ SDI - Link and AMD DirectGMA Technology." https://www.amd.com/Documents/SDI-tech-brief.pdf.

[4] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures," in *PACT*, pp. 13–24, IEEE, 2015.

[5] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson, "Gullfoss: Accelerating and Simplifying Data Movement Among Heterogeneous Computing and Storage Resources," *Tech. Rep. CS2015-1015, Department of Computer Science and Engineering, University of California, San Diego technical report*, 2015.

[6] M. Shihab, K. Taht, and M. Jung, "GPUDrive: Reconsidering Storage Accesses for GPU Acceleration," in *Workshop on Architectures and Systems for Big Data*, 2014.

[7] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: creating application objects efficiently for heterogeneous computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 53–65, IEEE, 2016.

[8] "Project Donard." https://github.com/sbates130272/donard, 2015.

[9] "NVM Express 1.0e." http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf, 2013.

[10] S. Kim, S. Huh, X. Z. Yige Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *OSDI 14*, pp. 6–8, USENIX, 2014.

[11] "Fail2Ban." www.fail2ban.org/.

[12] "mdadm - manage MD devices aka Linux Software RAID." https://www.kernel.org/pub/linux/utils/raid/mdadm/.

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

[14] "ArcGIS for Desktop." http://desktop.arcgis.com/en/arcmap.

[15] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A Case For Software Translation on GPUs," ISCA, IEEE, ACM, 2016.

[16] "Threaded I/O Tester." https://sourceforge.net/p/tiobench.

[17] "GPU Support in Apache Spark and GPU/CPU Mixed Resource Scheduling at Production Scale." http://www.spark.tc/gpu-support-in-spark-and-gpu-cpu-mixed-resource-scheduling-at-production-scale/, 2016.

[18] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: integrating file systems with GPUs," in *ASPLOS'13*, ACM, 2013.

[19] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha, "Vssim: Virtual machine based ssd simulator," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pp. 1–14, IEEE, 2013.

[20] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 266–277, IEEE, 2011.

[21] "A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology." https://github.com/NVIDIA/gdrcopy, 2015.

[22] "Evacuate struct_page from the block layer." https://lwn.net/Articles/636968/, 2015.

[23] "FOSS4G Benchmark." https://wiki.osgeo.org/wiki/FOSS4G_Benchmark.

[24] "True Marble." http://www.unearthedoutdoors.net/global_data/true_marble/.

[25] VMWare, "vRealize Log Insight." http://www.vmware.com/products/vrealize-log-insight.html.

[26] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 265–283, Springer, 2009.

[27] Antonio Torralba, Robert Fergus and William T Freeman, "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 11, pp. 1958–1970, 2008.

[28] "Benchmarking GPUDirect RDMA on Modern Server Platforms." https://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/, 2014.

[29] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," *TOCS*, vol. 32, no. 1, p. 1, 2014.

[30] *OpenCAPI*. http://opencapi.org/.

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