

Cross-vendor GPU-Initiated I/O with native Linux `io_uring`

Rodrigo Ceccato¹, Jociclelio Junior¹, Joseph Huber²,
Jose M. Monsalve Diaz², Hervé Yviquel¹

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

rodrigo.ceccato@ic.unicamp.br, hyviquel@unicamp.br

²Advanced Micro Devices (AMD)

joseph.huber@amd.com

Abstract. GPU applications increasingly require irregular file I/O whose offsets, buffer addresses, and sizes are determined dynamically during kernel execution. Usually, such requests are serviced by host software, placing the CPU in the I/O control path. We remove this per-request userspace mediation by allowing the accelerator to submit `io_uring` requests directly into GPU-visible pinned host memory. We provide a cross-vendor interface that preserves the standard Linux filesystem and storage stack. For 4 KiB random reads at queue depth 64, it sustains 90% of CPU-controlled `io_uring` throughput and 34% more IOPS than a host-service baseline modeling RPC-style mediation. For reads of 64 KiB or larger, our approach saturates the storage device.

1. Introduction

Multiple GPU workloads execute kernels that require file operations during execution, such as loading input data or storing checkpoints, with request parameters determined dynamically by GPU computation. However, GPU kernels cannot directly invoke conventional Linux file operations, since system calls are handled by the host CPU. Typically, the GPU sends a request to the host, which performs the corresponding operation and returns its result. Previous works expose operating-system services to GPU kernels through this model [Silberstein et al. 2013, Veselý et al. 2018], but the CPU remains in the control path of every request.

We provide a POSIX-like interface that allows GPU code to issue file operations. Asynchronous submission and completion are handled through the two circular queues exposed by `io_uring`, which form a shared-memory interface between userspace and the kernel. With `SQPOLL` enabled, a kernel thread monitors the submission queue and processes new requests without requiring a userspace syscall for every operation while the poller remains active. We expose this submission path to the GPU, allowing it to author submission entries and, in the GPU-visible-ring configuration, publish the submission tail directly. The evaluated prototype reaps completions on the host, while GPU-side completion reading is validated separately with a standalone ring probe. File data are transferred into pinned host memory and subsequently accessed by the GPU across Peripheral Component Interconnect Express (PCIe). Thus, the accelerator initiates the Input/Output (I/O) control path asynchronously, while the data path remains in host memory. This design is compatible with direct I/O using `O_DIRECT`, which bypasses the page cache when its alignment requirements are satisfied.

Unlike GPU I/O approaches that manage NVMe queues through specialized storage stacks, our design reuses the existing Linux `io_uring` interface, allowing GPU-generated requests to be processed through the filesystem and block layers. On both NVIDIA and AMD GPUs, our GPU-visible configuration allocates the ring in pinned host memory accessible to the GPU, allowing GPU threads to construct and publish requests directly. The submission protocol ensures that complete SQEs and their array entries are visible before the submission tail is advanced. We also evaluate an additional `mmap` configuration, in which the GPU writes SQEs to a pinned staging array, which the host copies into the ring before publishing the tail. Both configurations preserve ordinary file access through the existing Linux storage stack and remove per-request userspace SQE construction. The host remains responsible for ring creation, file registration, per-batch slot assignment, kernel launches, and SQPOLL wake-ups after idle. Overall, this work makes three contributions:

- a Linux-native AMD and NVIDIA design for GPU-authored `io_uring` submissions without a per-request userspace request-construction proxy;
- a queue protocol covering ring ownership, wrap-around, completion correlation, and system-scope publication across the GPU and the host CPU;
- an evaluation across AMD and NVIDIA GPUs demonstrating that GPU-authored `io_uring` submissions approach CPU-controlled throughput for small requests and preserve device-limited throughput for larger transfers, with no modification to the Linux storage stack.

The remainder of this paper is organized as follows. Section 2 reviews the background of current GPU I/O interfaces, considering host-mediated OS services, the RPC-based interface of LLVM GPU `libc`, and the `io_uring` interface. Section 3 details our Linux-native design for GPU-initiated I/O, covering the ring ownership protocol that preserves the `io_uring` queue ordering across GPU, CPU, and kernel agents. Section 4 reports the evaluation on NVIDIA and AMD platforms, separating userspace service cost, SQPOLL cost, and storage performance. Section 5 compares our approach with related works on GPU access to OS services, direct data paths, GPU-initiated storage access, and GPU-side filesystems. Finally, Section 6 presents the conclusions and outlines directions for future work.

2. Background of Current GPU I/O Interfaces

2.1. Host-mediated OS services

Conventional GPU interfaces rely on the CPU to manage device resources, data movement, and kernel execution. Consequently, GPU code cannot directly invoke operating-system services such as file I/O, networking, or memory management, and must communicate requests to the host [Vesely et al. 2018]. This model is suitable for applications organized into coarse phases of data movement and computation, but becomes less effective when I/O operations are determined during kernel execution. For example, a GPU thread may compute a file offset and then require the host to issue the corresponding operation. The GPU therefore determines the operation, but the CPU remains in its control path.

2.2. LLVM GPU `libc`

LLVM GPU `libc` aims to provide GPU implementations of a subset of the C standard library for AMD and NVIDIA GPUs [LLVM Project 2026a]. Functions that require

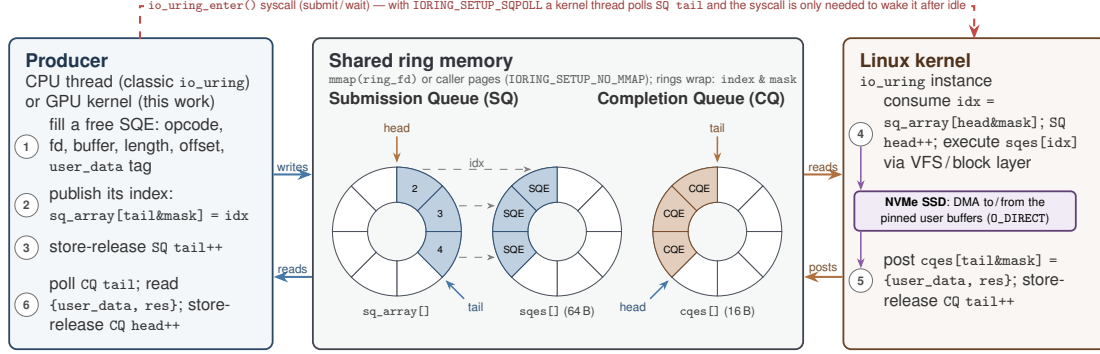


Figure 1. The `io_uring` shared-memory submission and completion queues.

operating-system services use a Remote Procedure Call (RPC) mechanism based on shared memory to communicate with a host-side server [LLVM Project 2026b]. This allows functions such as file operations to be called from GPU code, but each request must pass through a userspace CPU component before reaching the kernel. Our work targets this part of the execution path by investigating whether file operations can instead be submitted directly to the Linux kernel through `io_uring`.

2.3. Linux asynchronous I/O and `io_uring`

POSIX file operations are traditionally synchronous, requiring the caller to wait for completion. Linux Asynchronous I/O (AIO) introduced asynchronous submission and completion, but interacting with the interface requires system calls. `io_uring` provides a more efficient queue-based shared-memory interface between userspace and the kernel [Didona et al. 2022].

To use `io_uring`, the application creates a Submission Queue (SQ) and a Completion Queue (CQ), two circular buffers shared between userspace and the kernel, as Figure 1 illustrates. To request an operation, the user fills a Submission Queue Entry (SQE), a 64-byte record containing an opcode, a file descriptor, a buffer address, a length, an offset, and a `user_data` tag that identifies the request. The producer then publishes the entry index in the `sq_array` indirection table and advances the SQ tail (steps 1–3 in the figure). The kernel consumes entries from the SQ head, executes them through the regular Virtual File System (VFS) and block layers, and posts a Completion Queue Entry (CQE) containing the same `user_data` tag and a result code (steps 4–5). Since operations can complete in any order, this tag correlates each CQE with its originating request. The user reaps completions by reading CQEs and advancing the CQ head (step 6), which requires no system call.

By default, the application still calls `io_uring_enter` to notify the kernel about new submissions, although a single call can submit a batch of SQEs. The `IORING_SETUP_SQPOLL` flag removes this remaining syscall by creating a kernel polling thread that monitors the SQ tail, so publishing the tail is enough to start the operation. When idle, this thread sleeps and sets `IORING_SQ_NEED_WAKEUP`, and the application reactivates it with one `io_uring_enter` call. The application can also pre-register files and buffers, avoiding per-operation lookups. Finally, `IORING_SETUP_NO_MMAP`, available upstream since Linux 6.5, allows the application to provide the memory backing the rings. This feature enables our GPU-visible ring.

Table 1. Ring-pointer ownership and publication order in the queue protocol.

Pointer	Producer	Consumer	Must be visible first
SQ tail	GPU ^a	kernel (SQPOLL thread)	complete SQE; <code>sq_array[idx]</code>
SQ head	kernel	GPU/host (free slots)	—
CQ tail	kernel	host ^b	complete CQE
CQ head	host ^b	kernel (free slots)	CQE fully read

Every advance is published with release semantics by its producer and observed with an acquire load by its consumer. On the GPU, `__threadfence_system()` provides the system-scope ordering.

^aWith the GPU-visible ring (`IORING_SETUP_NO_MMAP`), the GPU publishes the tail directly; with the staged ring, the host copies the GPU-authored SQEs into the ring and publishes the tail on its behalf.

^bThe evaluated prototype consumes completions on the host. GPU-side CQ consumption is validated by the standalone ring probe.

2.4. GPU-visible host memory

Besides explicit copies between host and device memory, GPUs can directly access pinned host memory mapped into their address space. The Input-Output Memory Management Unit (IOMMU) can translate device-issued I/O addresses to the corresponding host physical pages. CUDA supports this through mapped pinned host memory on NVIDIA GPUs [NVIDIA Corporation 2026], while HIP provides equivalent zero-copy access on AMD GPUs [Advanced Micro Devices, Inc. 2026]. Since these allocations remain in host memory, GPU accesses must traverse the PCIe bus. This mechanism is particularly useful for small shared control structures, allowing GPU threads to read and modify host-resident data without explicit transfers.

For our design, GPU-visible host memory provides a shared region that allows the GPU and Linux kernel to access the same `io_uring` queues. However, sharing the memory is not sufficient, since queue updates must become visible in the correct order. An SQE must be completely written before its submission is published, and completion data must be visible before the CQ head is advanced. Our implementations therefore use platform-specific synchronization mechanisms to enforce the required memory ordering, as described in Section 3.

3. Linux-native GPU-initiated I/O

In this section, we detail how we implemented the GPU-visible support of `io_uring` on AMD and NVIDIA cards. We treat the `io_uring` queues as device-accessible interfaces. The rings live in the pinned host memory, where GPU threads write entries that the Linux kernel then consumes. We thus provide asynchronous GPU-initiated I/O with no new kernel code, no new file system and no per-request userspace service.

The host is responsible for ring creation at initialization, kernel launches, and SQPOLL wake-ups after idle. After the setup, our prototype removes per-request userspace mediation. A GPU thread fills a free SQE, publishes its index, and advances the SQ tail, with system-scope fences between these steps, represented by items 1–3 in Figure 1. At items 4–5, the SQPOLL kernel thread executes the operation, and at item 6 the host reaps the completion and advances the CQ head, after which the GPU can access the corresponding buffer. Algorithm 1 formalizes the submission steps.

Algorithm 1 GPU-visible ring: submission of one batch.

Require: batch of n requests; for each request p the host assigns an operation id g_p and a free pinned buffer slot b_p ; SQ mask m

Host, per batch

- 1: $t \leftarrow \text{SQ tail}$ ▷ GPU is producer of the SQ tail
- 2: launch $\text{FILLSQES}(t)$ with n GPU threads, then $\text{PUBLISHTAIL}(t + n)$ on the same stream and wait for both
- 3: **if** SQPOLL **then**
- 4: **if** $\text{ACQUIRELOAD}(\text{SQ flags}) \& \text{IORING_SQ_NEED_WAKEUP}$ **then**
- 5: $\text{io_uring_enter}(\text{IORING_ENTER_SQ_WAKEUP})$ ▷ after idle; counted
- 6: **else**
- 7: $\text{io_uring_enter}(to_submit = n)$ ▷ one syscall per batch
- 8: **procedure** $\text{FILLSQES}(t)$ ▷ GPU, one thread per request p
- 9: $i \leftarrow (t + p) \& m$
- 10: $sques[i] \leftarrow \{op, fd_p, addr = \text{hostVA}(b_p), len, off_p, user_data = (g_p, b_p)\}$ ▷ step 1
- 11: $\text{FENCESYSTEM}()$ ▷ complete SQE visible first
- 12: $sq_array[i] \leftarrow i$ ▷ step 2: publish the index
- 13: $\text{FENCESYSTEM}()$ ▷ $sq_array[i]$ visible first
- 14: **procedure** $\text{PUBLISHTAIL}(t')$ ▷ GPU, single thread, after FILLSQES
- 15: $\text{RELEASESTORE}(\text{SQ tail}, t')$ ▷ step 3: entries now owned by the kernel

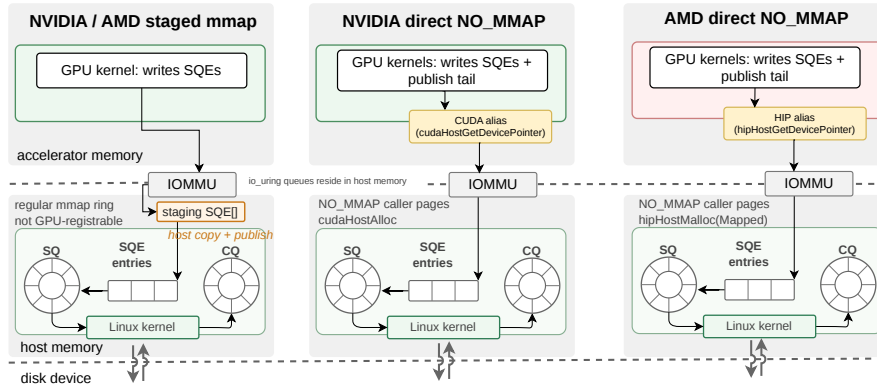


Figure 2. Cross-vendor exposure of the host-resident `io_uring` ring.

3.1. Ring ownership protocol

Algorithm 1 follows the `io_uring` queue protocol by assigning a single producer to each ring pointer. On the GPU-visible ring, the GPU advances the SQ tail, the Linux kernel advances the SQ head and CQ tail, and the host advances the CQ head. These pointers are updated with release semantics and read with acquire semantics to preserve queue ordering. Each GPU thread writes a complete SQE (line 10) and publishes its index (line 12), using system-scope fences to order these writes. Once all threads have finished, a second kernel on the same stream advances the SQ tail with release semantics (line 15). This ensures that the Linux kernel can observe the new submissions only after their SQEs and indices are visible. With the staged ring, the GPU writes SQEs to a pinned staging array instead, and the host copies them into the ring before publishing the indices and SQ tail. Table 1 summarizes these ownership and publication rules.

Figure 2 compares the mechanisms used to expose the `io_uring` queues to NVIDIA and AMD GPUs. Our NVIDIA implementation supports two ring placements. The first uses the regular `io_uring_setup` interface and maps the ring with `mmap`. Because the NVIDIA driver does not allow us to register this mapping with `cudaHostRegister`, the GPU writes complete SQEs to a pinned staging array allocated with `cudaHostAlloc`. The host then copies these entries into the `io_uring` ring and publishes the SQ tail. This approach preserves the regular ring allocation mechanism, but requires a host copy for each batch.

We remove this copy with a second implementation that lets the application

provide the memory used by the ring, instead of mapping kernel-allocated pages. We use `IORING_SETUP_NO_MMAP` and allocate the ring and SQE regions with `cudaHostAlloc`, pass their host addresses to `io_uring_setup`, and obtain the corresponding GPU pointers with `cudaHostGetDevicePointer`. This allows the GPU to write SQEs directly to the ring and publish the SQ tail without a host copy. The address fields stored in an SQE still contain host virtual addresses for the registered buffers, since the Linux kernel interprets these addresses in the process address space. The buffers therefore remain in pinned host memory, while the GPU accesses the same pages through a device-visible alias. Data transfers go through host memory, but our approach changes the control path, allowing the GPU to publish I/O requests without per-request userspace mediation. Direct SSD-to-GPU transfers are orthogonal to this approach and are discussed in Section 5.

On AMD, the same two placements apply through HIP, with `hipHostMalloc` and `hipHostGetDevicePointer` replacing their CUDA counterparts. As on NVIDIA, the driver rejects registering the `mmap` ring mapping, so the staged ring copies SQEs from a pinned staging array, while the GPU-visible ring relies on `IORING_SETUP_NO_MMAP`. Thus, both vendors expose the same `io_uring` protocol to GPU code through the same two ring placements, and Algorithm 1 applies unchanged to both.

3.2. Kernel-side polling and GPU completion reading

With `IORING_SETUP_SQPOLL`, the kernel creates a polling thread that monitors the submission queue. While this thread is active, publishing a new SQ tail is enough to make requests available to the kernel without a submission syscall. After the 2 s idle period configured through `sq_thread_idle=2000`, the thread goes to sleep and requests a wake-up through the SQ flags. The host checks this flag after each batch and, when it is set, reactivates the thread with a single system call (Algorithm 1, lines 3–5). We count these wake-ups to measure how often host intervention is required. Without `SQPOLL`, the host instead issues one `io_uring_enter` call per batch (line 7). Thus, `SQPOLL` allows syscall-free submission while the polling thread is active, while both paths avoid per-request submission syscalls through batching.

For completions, the host supports two waiting strategies. The spin mode busy-polls the CQ tail with acquire loads, while the block mode waits for completions through `io_uring_enter`. In both modes, the host reads the available CQEs, correlates each with its request through `user_data`, and advances the CQ head once per batch with release semantics. These strategies trade CPU usage for completion latency and behave differently under host CPU contention, as Section 4 shows.

4. Results and discussion

We ran our experiments on the systems described in Table 2. All file I/O targets regular files opened with `O_DIRECT`, and we snapshot `/proc/diskstats` around every run to confirm that the completed bytes reached the device. The P100 node carries the primary dataset and uses a distribution backport of `IORING_SETUP_NO_MMAP`. We use a GTX 980 workstation as a portability check, since its 4 KiB throughput is limited by the host submission rate rather than the device. The MI100 node provides the AMD side of the

Table 2. Evaluation platforms and measured read ceilings.

Platform	GPU	Host	Kernel	SSD	4 KiB (kIOPS)	64 KiB (MiB/s)
P100 node	Tesla P100	Xeon Silver 4108	5.14	NVMe 760p	116	1402
Workstation	GeForce GTX 980	Core i7-3770K	7.1.5	SATA SA400	27.4	176
M1100 node	Instinct MI100	2× EPYC 7763	6.17	SATA S4510	17.9	358

cross-vendor evaluation. On this platform, the measurements are device-bound in the best configuration, reaching 94% and 96% of the ceilings in Table 2 at 4 KiB and 64 KiB, respectively.

Across the platforms, we sweep request size, queue depth, access pattern, and submission path, with ten repetitions per point. The collected data and source code of our prototype are available at a public repository¹. Every figure reports the mean of these ten runs with a bootstrap 95% confidence interval, shown as a shaded band along each line, which is thinner than the line itself at very stable points. The AMD node repeats a reduced matrix using the same protocol, with sequential and random reads at 4 KiB and 64 KiB over five methods: CPU `io_uring` without SQPOLL, and the staged and GPU-visible rings, each with and without SQPOLL. Thus, this platform has no RPC or synchronous baseline and no 1 MiB point. These axes expose the two regimes that determine control-path performance. Small requests at deep queues are limited by submission rate, where the cost of GPU-authored submission becomes visible. Large requests saturate the device, where any correct path converges to the same throughput. Queue depth 1 isolates the control path, since there is no queueing to amortize submission.

We could not directly compare against BaM, AGILE, or AGIO (see Section 5) because we could not deploy their available implementations on our platforms. BaM requires a Volta-class or newer NVIDIA GPU, control of an NVMe device that can be rebound to its custom driver, and root privileges to load its kernel module. AGILE similarly requires a dedicated NVMe device, custom drivers loaded as root, and kernel and boot configuration changes, with its released drivers targeting Linux 6.8. Our P100 node uses a Pascal GPU and its only NVMe device is the cluster system disk. The workstation uses an older GPU and SATA storage, while the AMD node cannot run these CUDA-based stacks. We did not find any AGIO public artifact. We therefore validate our experimental harness against fio [Axboe 2024], a widely used storage workload generator, using its `io_uring` backend. For matched random-read request sizes and queue depths using `O_DIRECT`, fio reproduces the throughput of our CPU-controlled `io_uring` implementation with a maximum deviation of 6%. We compare BaM, AGILE, and AGIO architecturally in Section 5.

We compare systems that share the same workload generator, files, and measurement windows. CPU `io_uring` submits from the host, while GPU `io_uring` submits GPU-authored SQEs through the staged ring. Host-service RPC is a baseline in which GPU threads publish request descriptors and a host thread builds and submits the SQEs. This baseline models the per-request control path of RPC-based GPU services, but it is not a measurement of LLVM GPU libC itself. Each `io_uring` path is measured with and without SQPOLL, which, together with the RPC baseline, gives five systems evaluated across queue depths. In the GPU configurations, GPU threads directly author I/O requests, while the host handles their completions.

¹Code and raw data: <https://github.com/Jociclelio/sscad>.



Figure 3. P100 read throughput versus queue depth and request size.

Two synchronous references issue `pread/pwrite` directly. *POSIX syscalls, 4 threads* uses four host threads that block in parallel, while *GPU + CPU syscall* has the GPU produce or consume the data and one host thread perform the call. Their concurrency is fixed by construction, with four and one requests in flight, respectively. Therefore, they have no queue-depth sweep and appear in the figures as reference levels at their corresponding concurrency rather than as lines.

The two GPU `io_uring` systems are each evaluated with both ring placements. Together with the three other asynchronous configurations and two synchronous references, this gives nine configurations on the P100 node. We use request sizes of 4 KiB, 64 KiB, and 1 MiB, queue depths from 1 to 64, and both sequential and random access. In the throughput and latency plots, color and marker identify the system, while line style distinguishes the GPU ring placements. The staged ring uses dashed lines with hollow markers, while the GPU-visible ring uses solid lines with filled markers.

Figure 3 shows how throughput scales with queue depth. GPU submission follows the CPU path across the entire grid. At 64 KiB and 1 MiB, every asynchronous system saturates the NVMe device, so authoring SQEs on the GPU does not reduce bandwidth and the two ring placements become indistinguishable at all but one point. The difference appears at 4 KiB, where the request rate is highest. With SQPOLL and the staged ring, the GPU path reaches 99 kIOPS at queue depth 64, compared with 114 kIOPS for the CPU path. Moving the ring into GPU-visible memory removes the host copy from the submission path and increases this to 103 kIOPS. Without SQPOLL, the staged ring reaches 78 kIOPS, matching the host-service baseline of 77 kIOPS, while the GPU-visible ring still reaches 92 kIOPS. Thus, GPU-side SQE authorship benefits both from syscall-free submission and from writing the ring in place. The only case where ring placement matters at a large request size is 1 MiB sequential access, where the GPU-visible ring sustains 2.1 kIOPS, compared with 1.6 kIOPS for every staged and CPU path.

Figure 4 reports median and tail latency at 4 KiB, the only request size where the control path is visible. At 64 KiB and 1 MiB, the device dominates, and all systems remain within 20% of each other at the same queue depth. Queue depth 1 is the point of interest, since there is no queueing to amortize submission. With the staged ring, GPU

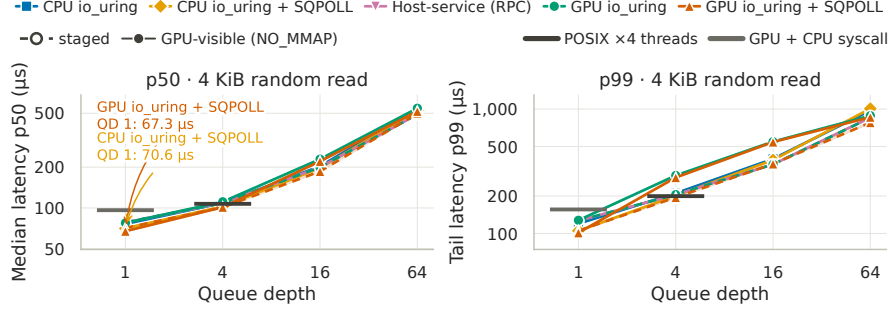


Figure 4. P100 median (p50) and tail (p99) latency for 4 KiB random reads.

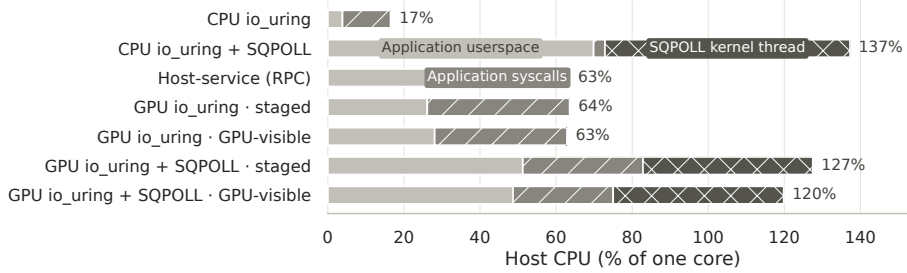


Figure 5. P100 host CPU cost at queue depth 64 for 64 KiB random reads.

submission adds at most 2 μs over the CPU path, while SQPOLL saves about 6 μs for both. The GPU-visible ring has a measured median of 67.3 μs , compared with 71.1 μs for the staged ring and 70.6 μs for CPU io_uring + SQPOLL. We do not interpret this difference as a gain, since the GPU-visible ring timestamps a request only after the kernel that publishes the tail completes. At deeper queues, the medians converge for all systems, since latency grows with the number of requests resident in the device. Tail latency is less uniform. At queue depths 4 and 16, the GPU-visible ring has a p99 34–52% higher than the other paths, while at queue depth 64 every GPU path has a lower p99 than either CPU path, but also lower throughput. Our instrumentation does not identify the source of this difference, so we report it as an open observation.

These results establish that moving request construction and submission publication to the GPU does not inherently compromise storage throughput. Although GPU submission introduces additional overhead at high request rates, the implementation approaches CPU-controlled io_uring performance for small requests and reaches the same device-limited throughput for larger transfers. The main contribution is not higher peak bandwidth, but the ability to preserve the existing Linux I/O path while allowing the GPU to directly construct and publish requests.

Figure 5 separates where CPU time is spent at device saturation, that is, 64 KiB, queue depth 64, where every asynchronous path delivers the same throughput and CPU cost can be compared fairly. We sample the CPU time of each host thread from `/proc` and normalize it to one core, so 100% corresponds to one fully busy core, and a bar above 100% indicates that the workload consumes more than one core worth of CPU time. Each bar stacks the time of all threads involved in the I/O path, including the SQPOLL kernel thread, which runs concurrently with the application.

For the GPU paths without SQPOLL, driving batches and reaping completions

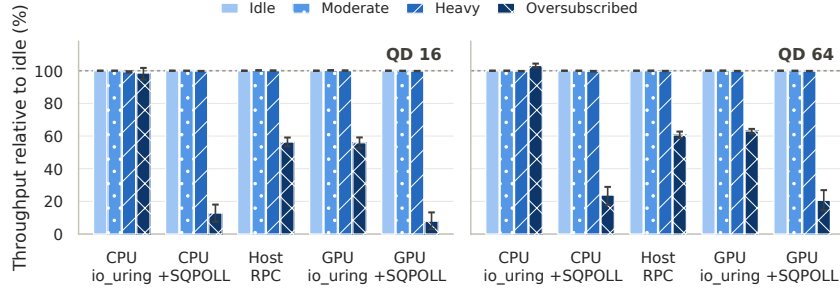


Figure 6. P100 throughput under host contention for 64 KiB random reads.

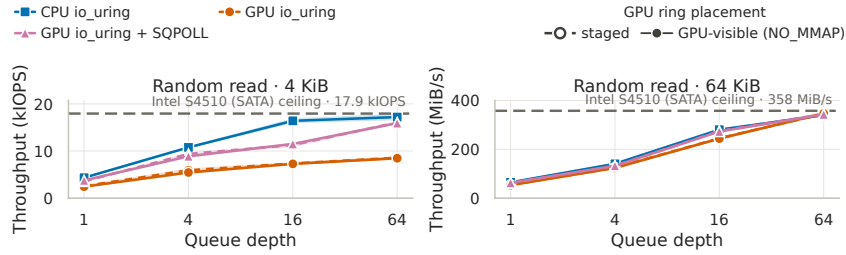


Figure 7. Cross-vendor throughput validation on the MI100 node.

costs about 26% of one core in userspace, with another 35–38% spent in system calls, including submission and the asynchronous driver copy used to transfer completed data into GPU memory. This copy models an accelerator that consumes the data and is also performed by the host-service baseline, while the CPU-only paths have no accelerator consumer. The GPU paths without SQPOLL therefore use 63–64% of one core, comparable to the 63% of the host-service baseline.

With SQPOLL, submission syscalls are avoided while the polling thread remains active, but the thread introduces additional CPU usage. The application userspace share increases to 49–51%, its syscall share falls to 26–32%, and the SQPOLL thread consumes about 45% of one core. Combined, the GPU paths with SQPOLL use 120–127% of one core, compared with 137% for CPU io_uring + SQPOLL. Blocking CPU io_uring, with one submission syscall per batch and no polling thread, uses only 17%, although it does not include the GPU data-transfer work. These results show that GPU-side request submission removes per-request userspace mediation from the storage submission path, while batch orchestration, completion handling, and data transfers still consume host CPU time. SQPOLL further removes submission syscalls, at the cost of a dedicated polling thread.

Figure 6 evaluates the effect of host CPU contention using hog processes at four load levels. Each system is reported relative to its own idle throughput, since this experiment measures sensitivity to contention rather than storage performance. Moderate and heavy load have no effect. Under oversubscription, however, every busy-polling configuration degrades. The GPU path with SQPOLL falls to 8% of its idle throughput at queue depth 16 and 21% at queue depth 64, while CPU io_uring + SQPOLL falls to 13% and 24%, respectively. Blocking CPU io_uring maintains its idle throughput at both depths. The block completion mode from Section 3 recovers the GPU path to about half of its idle throughput. Busy-polling should therefore be reserved for hosts with spare cores.

Finally, Figure 7 evaluates the cross-vendor behavior on the AMD node. The same

GPU-visible ring, compiled for MI100 through HIP, reproduces the behavior measured on NVIDIA. With SQPOLL, it reaches 93% of the CPU `io_uring` baseline, which runs without SQPOLL on this platform, at 4 KiB and queue depth 64, while at 64 KiB every path reaches 95–97% of the measured device ceiling. The staged and GPU-visible placements behave alike on this platform, differing by at most 8% at any point, with no consistent ordering. We also validated the GPU-visible ring on a Maxwell GTX 980 workstation, where at 4 KiB and queue depth 64 it reaches 5.6 kIOPS, 98% of CPU `io_uring` without SQPOLL (5.7 kIOPS), and 6.8 kIOPS with SQPOLL, confirming that the same submission protocol operates across GPU generations and vendors.

5. Related Works

GPU access to host OS services. Previous work has provided interfaces that free GPU code from manually orchestrating host round trips and explicit data-movement API calls. GPUfs [Silberstein et al. 2013] provides a POSIX-like file interface to GPU kernels and uses host-side components to execute file operations. GENESYS [Vesely et al. 2018] extends this approach to generic system calls, using the host CPU to fulfill requests from GPU threads. These systems make OS services available to the GPU during kernel execution, but retain a host-mediated control path. Our work instead uses the shared queues exposed by `io_uring` to submit file operations asynchronously and directly to the Linux kernel, removing the per-request userspace intermediary.

Direct data paths. Besides the CPU control path, data is also staged in host memory before reaching the accelerator. Some works remove this step when the CPU does not access the data. GPUDirect Storage, available on some NVIDIA platforms, allows direct copies from the storage device to the accelerator [Inupakutika et al. 2022]. SPIN [Bergman et al. 2019] incorporates the copy from the storage device to the GPU into the operating system file service, enabling features such as page cache and read-ahead. These works improve throughput and reduce host memory consumption by removing the intermediary copy, but the control path remains on the host CPU, which initiates every operation. Our work instead uses an in-memory queue that allows the GPU to initiate the operations while leaving the data path unchanged. Direct storage-to-GPU mechanisms are orthogonal to our approach and could be integrated in the future.

GPU-initiated storage access. Recent works remove per-request CPU mediation from GPU file access to reduce its overhead. BaM [Qureshi et al. 2023] places NVMe submission and completion queues in GPU memory and provides a software stack that allows GPU threads to insert requests into them. AGILE [Yang et al. 2025] follows the same approach and adds a configurable software cache and asynchronous operations. AGIO [Han et al. 2026] similarly provides explicit asynchronous read and write operations initiated by GPU threads, separating submission from completion. Our work also enables GPU-initiated I/O, but instead of managing NVMe queues through a specialized software stack, GPU threads submit requests through the existing Linux `io_uring` interface. The Linux kernel processes these requests through the existing filesystems and storage drivers, preserving ordinary file access without modifications to the storage stack. Our implementation supports both AMD and NVIDIA GPUs and currently exposes read and write operations on host-opened file descriptors. The same submission mechanism could be extended to other `io_uring` operations.

Table 3. GPU I/O approaches (✓ supported, × not supported, ~ partial)

System	GPU init.	Async.	Linux FS path	Unmodified Linux storage stack	Queue location	Completion handling	Supported GPUs
GPUfs	✓	~	✓	×	host	host	NVIDIA
GDS	×	✓	✓	~	host	host	NVIDIA
BaM	✓	×	×	×	GPU	GPU	NVIDIA (≥ Volta)
AGILE	✓	✓	×	×	GPU	GPU	NVIDIA
AGIO	✓	✓	~	×	GPU	GPU	NVIDIA
GoFS	✓	✓	×	×	GPU	GPU	NVIDIA
This work	✓	✓	✓	✓	host	host	NVIDIA and AMD

GPU-side filesystems. Other works, such as GoFS [Li et al. 2025], introduce GPU-side filesystem management and expose a POSIX-style interface. Filesystem metadata and submission and completion queues are kept in GPU memory and managed by GPU code. This approach removes the host from both the control and data paths, but introduces a new filesystem. Our work instead leaves filesystem management to the Linux kernel and requires only the userspace side of `io_uring` to be accessed by the accelerator. Thus, our approach requires no kernel modifications and preserves the existing storage stack. Table 3 summarizes these approaches and highlights the main difference of our work: GPU-initiated asynchronous I/O while preserving the standard Linux storage stack across AMD and NVIDIA GPUs.

6. Conclusion

We presented a Linux-native approach to GPU-initiated file I/O, allowing GPU threads to construct and publish requests directly through `io_uring` queues in GPU-visible host memory. Unlike approaches that introduce specialized GPU storage stacks, our implementation preserves the existing Linux filesystems and storage drivers, supports both AMD and NVIDIA GPUs, and removes per-request userspace mediation. For small random reads at deep queue depths, GPU-controlled `io_uring` reaches about 90% of CPU throughput on NVIDIA and 93% on AMD. Once the storage device is saturated, both paths reach comparable throughput. These results demonstrate that GPU-authored submissions can use the existing Linux I/O interface without substantially compromising storage performance.

Future work includes improvements to the I/O control path, since `SQPOLL` still consumes CPU resources and requires host intervention after becoming idle. Our implementation also relies on the host to assign submission slots, launch GPU kernels for each batch, and reap completions. Supporting independent GPU producer warps will require GPU-side slot arbitration, consideration of GPU forward-progress guarantees, and GPU-accessible mechanisms for waking the poller and managing driver resources. These extensions could enable more autonomous GPU-initiated I/O and extend the same model to other operating-system services.

7. Acknowledgments

Part of this work was developed as an LLVM project in Google Summer of Code 2025, mentored by Joseph Huber. This work is supported by the São Paulo Research Foundation (FAPESP), Grants No. 2025/09515-0, by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) - Finance Code 001. This work was also supported in part by Advanced Micro Devices, Inc. under the AMD University Program’s AI & HPC Cluster.

References

- Advanced Micro Devices, Inc. (2026). *HIP Programming Manual*. Advanced Micro Devices, Inc. Available at: https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/programming_manual.html. Accessed: Aug. 12, 2026.
- Axboe, J. (2024). fio – Flexible I/O Tester. https://fio.readthedocs.io/en/latest/fio_doc.html. Accessed: Aug. 15, 2026.
- Bergman, S., Brokhman, T., Cohen, T., and Silberstein, M. (2019). SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. *ACM Transactions on Computer Systems*, 36(2):5:1–5:26.
- Didona, D., Pfefferle, J., Ioannou, N., Metzler, B., and Trivedi, A. (2022). Understanding modern storage APIs: A systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR)*, pages 120–127. ACM.
- Han, J., Sivasubramaniam, A., Chang, C.-H., Mailthody, V. S., Qureshi, Z., and Hwu, W. (2026). Asynchrony and GPUs: Bridging this dichotomy for I/O with AGIO. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Volume 2, pages 208–222. ACM.
- Inupakutika, D., Davis, B., Yang, Q., Kim, D., and Akopian, D. (2022). Quantifying performance gains of GPUDirect storage. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–9. IEEE.
- Li, S., Zhou, Y. E., Xue, Y., Xu, Y., and Huang, J. (2025). Managing scalable direct storage accesses for GPUs with GoFS. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, pages 979–995. ACM.
- LLVM Project (2026a). libc for GPUs. <https://libc.llvm.org/gpu/>. Accessed: Aug. 13, 2026.
- LLVM Project (2026b). Remote Procedure Calls — The LLVM C Library. <https://libc.llvm.org/gpu/rpc.html>. Accessed: Aug. 13, 2026.
- NVIDIA Corporation (2026). *CUDA C++ Programming Guide*. NVIDIA Corporation. Available at: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Accessed: Aug. 12, 2026.
- Qureshi, Z., Mailthody, V. S., Gelado, I., Min, S., Masood, A., Park, J., Xiong, J., Newburn, C. J., Vainbrand, D., Chung, I.-H., Garland, M., Dally, W., and Hwu, W. (2023). GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 325–339. ACM.
- Silberstein, M., Ford, B., Keidar, I., and Witchel, E. (2013). GPUfs: Integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 485–498. ACM.

- Veselý, J., Basu, A., Bhattacharjee, A., Loh, G. H., Oskin, M., and Reinhardt, S. K. (2018). Generic system calls for GPUs. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 843–856. IEEE.
- Yang, Z., Zhuang, J., Chen, X., Jones, A. K., and Zhou, P. (2025). AGILE: Lightweight and efficient asynchronous GPU-SSD integration. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1028–1042. ACM.