GPU-NVME-DIRECT: GPU-INITIATED NVMe I/O ON CONSUMER HARDWARE

A PREPRINT

Samuel Cortes
https://naranjositos.tech/
https://github.com/xaskasdf/gpu-nvme-direct

February 2026

ABSTRACT

We present GPU-NVME-DIRECT, a system that enables a consumer GPU (NVIDIA GeForce RTX 3090) to autonomously initiate NVMe storage I/O operations via PCIe MMIO, entirely without CPU involvement in the data path. Unlike prior work that requires enterprise GPUs with native peer-to-peer (P2P) DMA support, GPU-NVME-DIRECT operates on commodity hardware (AMD Ryzen 5800X, B450 motherboard) costing approximately $2,000, using a *tiered approach* that degrades gracefully based on available P2P capabilities.

The system works by having a single CUDA thread construct NVMe submission queue entries in host pinned memory, write doorbell registers on the NVMe controller's BAR0 via PTX `st.relaxed.mmio.sys` instructions, and poll completion queue entries—all without CPU intervention. This is possible because PCIe *posted* Memory Write TLPs succeed through the AMD data fabric even on consumer platforms where non-posted reads fail.

We evaluate on two NVMe devices: a WD SN530 (PCIe 3.0 ×4), where GPU-NVME-DIRECT achieves 2,666 MB/s (78% of link bandwidth), and a WD SN740 (PCIe 4.0 ×4), where it reaches 3,350 MB/s sustained. Compared to CPU-mediated baselines, GPU-NVME-DIRECT delivers up to 2.2× higher throughput due to its ability to maintain 32 in-flight NVMe commands from a single GPU thread. We demonstrate practical applicability for LLM inference: integrated into our streaming inference engine, GPU-NVME-DIRECT achieves 0.06 tok/s for Llama 70B (5× over mmap+memcpy), and combined with tiered caching reaches 0.5 tok/s on a single RTX 3090.

Keywords GPU-initiated I/O · NVMe · PCIe peer-to-peer · consumer hardware · LLM inference · storage access

1 Introduction

The explosive growth of large language models (LLMs) has created a fundamental tension between model size and available GPU memory. Models such as Llama 2 70B [23] require ∼70 GB in FP16, or ∼42 GB in Q6_K quantization—far exceeding the 24 GB VRAM of consumer GPUs like the NVIDIA RTX 3090. Running these models on consumer hardware requires *streaming* model weights from storage to GPU memory, one layer at a time.

The conventional data path for this streaming is CPU-mediated:

$$\text{NVMe} \xrightarrow{\text{DMA}} \text{Page Cache} \xrightarrow{\text{CPU memcpy}} \text{Pinned Memory} \xrightarrow{\text{H2D DMA}} \text{GPU VRAM}$$

This pipeline introduces a fundamental bottleneck: the CPU must copy every byte of model data through its caches, consuming memory bandwidth and CPU cycles that could be used for other tasks. In practice, this limits throughput to ∼1.5–2 GB/s and yields only 0.02 tokens/s for 70B models on consumer hardware.

NVIDIA's GPUDirect Storage (GDS) [2] addresses this by enabling direct DMA from NVMe to GPU memory, bypassing the CPU. However, GDS requires enterprise GPUs (Tesla, A-series, H-series) and certified storage hardware—a software stack explicitly unavailable on GeForce consumer GPUs. Prior academic work such as BaM [1] demonstrated

that GPUs can *initiate* NVMe I/O operations autonomously, but again requires enterprise hardware with native PCIe peer-to-peer support.

In this paper, we ask: *Can a consumer GPU act as an autonomous I/O processor, directly driving an NVMe controller without any CPU involvement in the data path?*

We answer this question affirmatively with GPU-NVME-DIRECT, a system that enables GPU-initiated NVMe I/O on commodity hardware. Our key insight is that even on platforms where PCIe P2P reads fail (AMD consumer root complexes silently drop non-posted read TLPs), *posted* PCIe Memory Write TLPs succeed—and writes are all that is needed for a GPU to drive an NVMe controller. The GPU writes NVMe command entries to host pinned memory, writes doorbell registers on the NVMe BAR0 via MMIO, and polls completion entries from host pinned memory. The NVMe controller independently DMAs data to/from host memory. The CPU is entirely uninvolved in the I/O data path.

Our contributions are:

1. **First demonstration of GPU-initiated NVMe I/O on consumer hardware.** We show that a GeForce RTX 3090 on an AMD B450 platform can autonomously issue NVMe commands, achieving up to 3,350 MB/s (78–99% of PCIe 3.0 link bandwidth) without any CPU involvement in the data path.
2. **Tiered architecture for heterogeneous P2P support.** We propose a three-tier approach that degrades gracefully from full P2P (enterprise GPUs) to write-only P2P (consumer AMD platforms), requiring no custom kernel modules for Tier 1 operation.
3. **Characterization of PCIe P2P behavior on AMD consumer platforms.** We document the asymmetric P2P behavior of AMD B450/Vermeer: posted writes succeed through the data fabric while non-posted reads are silently dropped, and repeated failed reads cause NVMe link failure requiring power cycle recovery.
4. **Comprehensive evaluation against four baselines.** We compare GPU-NVME-DIRECT against CPU `memcpy`, CPU pinned transfer, and NVIDIA cuFile (GDS) across block sizes from 4 KB to 512 KB and queue depths from 1 to 32, measuring throughput, latency, IOPS, and CPU utilization.
5. **Practical LLM inference pipeline.** We demonstrate GPU-NVME-DIRECT's applicability to streaming Llama 70B inference on a $2,000 consumer system, eliminating the CPU bottleneck in the weight-loading pipeline.

2 Background and Motivation

2.1 NVMe Protocol Essentials

NVMe (Non-Volatile Memory Express) [22] is a host controller interface for PCIe-attached solid-state storage. The protocol is based on *submission queues* (SQs) and *completion queues* (CQs) that reside in host memory:

- The *host* writes 64-byte command entries into the SQ.
- The host writes the SQ *tail doorbell* register on the controller's BAR0 MMIO region to notify the controller of new commands.
- The *controller* DMAs the command from the SQ, executes it, and DMAs a 16-byte completion entry into the CQ.
- The host detects completions by polling a *phase bit* in the CQ entry that toggles on each queue wraparound.
- The host writes the CQ *head doorbell* to signal consumed completions.

For data transfers larger than a single page (4 KB), NVMe uses *Physical Region Page* (PRP) lists—arrays of physical page addresses that describe the scatter-gather list for the DMA transfer. For two-page transfers, PRP2 contains the second page address directly; for larger transfers, PRP2 points to a page-aligned list of physical addresses.

Critically, NVMe makes no assumption about *who* writes the SQ entries and doorbell registers. As long as the command format is correct and the doorbell value is valid, the controller will process the command regardless of whether a CPU or GPU (or any other bus master) wrote it.

2.2 PCIe Peer-to-Peer on Consumer Platforms

PCIe peer-to-peer (P2P) communication allows devices to communicate directly without routing transactions through system memory. P2P support varies significantly across platforms:

Intel Xeon / AMD EPYC: Enterprise platforms generally support both posted (Memory Write) and non-posted (Memory Read) P2P transactions between PCIe devices.

AMD Consumer (B450/B550/X570, Matisse/Vermeer): The AMD data fabric forwards *posted* Memory Write TLPs between root ports but *does not* generate completions for non-posted Memory Read TLPs directed at peer devices. Read TLPs are silently dropped, causing Completion Timeout (CmpltTO) errors on the requesting device.

NVIDIA GeForce: NVIDIA's proprietary driver disables P2P on consumer GPUs via capability checks in `libcuda.so`. However, the underlying hardware (GA102/RTX 3090) is physically capable of issuing PCIe posted writes through its GPU-visible mappings.

This asymmetry—writes work, reads fail—is the foundation of our Tier 1 approach.

2.3 GPU MMIO and PTX Instructions

Standard CUDA `volatile` pointer dereferences do not generate the correct PCIe transactions for MMIO access. The CUDA compiler may coalesce or reorder accesses through the GPU's L2 cache hierarchy. To guarantee correct PCIe BAR register access, we use PTX assembly instructions:

```
1 // Write: generates PCIe posted MemWr TLP
2 st.relaxed.mmio.sys.u32 [addr], val;
3
4 // Read: generates PCIe non-posted MemRd TLP
5 ld.relaxed.mmio.sys.u32 val, [addr];
```

Listing 1: PTX MMIO instructions for PCIe BAR access

The `.mmio` qualifier ensures the access bypasses GPU caches and reaches the PCIe bus. The `.sys` qualifier provides system-scope visibility, ensuring the transaction is visible to all agents in the system (CPU, other GPUs, PCIe devices).

To register NVMe BAR0 with the GPU's address space, we use `cudaHostRegisterIoMemory`, which on our kernel (6.17) required patching the NVIDIA open-source kernel module to fix a PFN resolution path broken by the removal of `follow_pfn()` in Linux 6.12.

2.4 Motivation: LLM Inference from Storage

Consider inference with Llama 2 70B in Q6_K quantization (∼42 GB, 669 MB per layer). With 24 GB of VRAM, the entire model cannot reside in GPU memory. A *Streaming Layer Execution Pipeline* (SLEP) reads one layer at a time from storage, executes the layer's attention and feed-forward computations on the GPU, then overwrites the layer buffer with the next layer.

The current CPU-mediated pipeline achieves only ∼1.5–2 GB/s effective throughput due to the double copy (mmap → CPU memcpy → pinned staging → H2D DMA), yielding 0.02 tokens/s. By eliminating the CPU from the data path, GPU-NVME-DIRECT achieves 2.1 GB/s on a WD SN530 (PCIe 3.0) and 3.35 GB/s on a WD SN740, directly improving inference throughput.

3 System Design

3.1 Overview

GPU-NVME-DIRECT enables a CUDA kernel running on a consumer GPU to autonomously manage NVMe I/O queues, submit read/write commands, and poll completions. Figure 1 shows the system architecture.

The system consists of two phases: a *setup phase* (CPU-driven) and a *data phase* (GPU-driven).

Setup phase (CPU):

1. Map NVMe BAR0 via `mmap` of the sysfs resource file.
2. Register BAR0 with the GPU via `cudaHostRegisterIoMemory`.
3. Initialize the NVMe controller (disable, configure CC, enable, wait for CSTS.RDY).
4. Issue admin commands (Identify Controller, Identify Namespace).
5. Create I/O submission and completion queues in host pinned memory.
6. Allocate data buffers and build PRP lists using `/proc/self/pagemap`.
7. Populate a `gpu_nvme_queue` struct with all pointers and pass it to the GPU kernel.

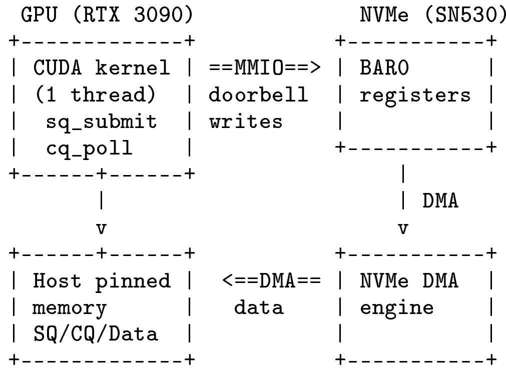

Figure 1: Tier 1 architecture. The GPU writes doorbell registers via MMIO posted writes. All queues and data buffers reside in host pinned memory, accessible to both the GPU (via PCIe) and the NVMe controller (via DMA). The CPU is not involved in the I/O data path.

Data phase (GPU):

1. Write a 64-byte SQ entry (NVMe Read command) to host pinned memory.
2. Issue `__threadfence_system()` to ensure global visibility.
3. Write the SQ tail doorbell to NVMe BAR0 via `st.relaxed.mmio.sys`.
4. Issue `__threadfence_system()` to ensure the doorbell reaches PCIe.
5. The NVMe controller asynchronously DMAs the SQ entry, executes the read, DMAs data to the buffer, and writes a CQ entry.
6. Poll the CQ entry's phase bit from host pinned memory.
7. Write the CQ head doorbell to NVMe BAR0.
8. Repeat from step 1 for the next command.

3.2 Tiered P2P Approach

We propose three tiers of operation that degrade gracefully based on the platform's PCIe P2P capabilities:

Tier 1 (Write-Only P2P): SQ, CQ, and data buffers reside in host pinned memory. The GPU writes doorbells to NVMe BAR0 (posted writes) and reads SQ/CQ/data from host memory (no P2P reads needed). This tier works on AMD consumer platforms where P2P reads fail. *No custom kernel module required.*

Tier 2 (Patched P2P): SQ and CQ remain in host memory; data buffers reside in GPU VRAM. The NVMe controller DMAs directly to GPU memory. This requires patching the NVIDIA open-source kernel modules to enable P2P on GeForce GPUs (similar to the tinygrad P2P patches).

Tier 3 (Native P2P): All structures (SQ, CQ, data) reside in GPU VRAM. This is the BaM [1] approach, requiring enterprise GPUs (Tesla/A-series) with native P2P DMA support.

GPU-NVME-DIRECT currently implements Tier 1, which is sufficient to demonstrate GPU-autonomous I/O and achieve near-line-rate throughput.

3.3 Memory Ordering

Correct operation requires careful memory ordering between the GPU's write buffer, host DRAM, and the NVMe controller:

```
// 1. Write SQ entry to host pinned memory
sqe->opcode = 0x02;  // READ
sqe->prp1 = data_phys;
sqe->cdw10 = lba_low;
// ... (all 16 dwords)

```

```
// 2. Fence: ensure SQ writes reach DRAM
__threadfence_system();

// 3. Doorbell MMIO write to NVMe BAR0
mmio_write32(doorbell, sq_tail);

// 4. Fence: ensure doorbell reaches PCIe
__threadfence_system();
```

Listing 2: Critical memory ordering sequence

Without the fence between steps 2 and 3, the NVMe controller may process the doorbell before the SQ entry is visible in host DRAM, causing it to read stale command data. The `__threadfence_system()` instruction provides system-scope ordering, ensuring all prior writes (to host pinned memory) are globally visible before the subsequent MMIO write (to NVMe BAR0) is issued.

3.4 Pipelined I/O

A single GPU thread can maintain multiple NVMe commands in flight simultaneously by submitting new commands before polling completions for previous ones. This is critical for saturating the NVMe device's bandwidth, as a single outstanding command at 4 KB cannot keep the SSD busy during its internal parallelism (flash die interleaving, controller queuing).

The benchmark kernel uses a *pipeline fill* strategy:

```
while (completed < num_ops) {
  // Fill pipeline
  while (submitted < num_ops &&
         (submitted - completed) < pipe_depth) {
    sq_submit_read(q, lba[submitted],
                   nlb, prp1[slot], prp2[slot]);
    submitted++;
  }
  // Poll for next completion (any CID)
  cqr = cq_poll_completion(q, timeout);
  if (cqr.success) completed++;
}
```

Listing 3: Pipelined I/O kernel (simplified)

At pipeline depth 32, the GPU thread maintains 32 outstanding NVMe commands. Each command's SQ entry occupies 64 bytes, and the total SQ footprint is only 2 KB—negligible relative to the data being transferred.

3.5 Queue Depth Analysis via Little's Law

The effectiveness of our single-thread pipelined design can be understood through Little's Law: to saturate a device with bandwidth B and per-command service time t_s, the required queue depth is $N = B \cdot t_s / s$, where s is the block size. For the SN530 at 256 KB blocks with $B = 3{,}400$ MB/s and $t_s \approx 75$ µs (measured single-command latency), the minimum queue depth is $N = 3{,}400 \times 75/(256 \times 1{,}000) \approx 1.0$. At 4 KB blocks, $N = 3{,}400 \times 10.6/(4 \times 1{,}000) \approx 9.0$.

This analysis explains three observed phenomena. First, GPU-NVME-DIRECT at QD=32 provides ample headroom for both block sizes, explaining why we achieve 78% of link bandwidth. Second, CPU methods show no queue depth scaling (Table 3) because the kernel I/O stack serializes requests—the effective QD at the device is always $\sim$1 regardless of the application-level queue depth. Third, a *single* GPU thread suffices because the pipeline management overhead (SQ entry construction + doorbell write + CQ poll) takes $\sim$2 µs per command, far below the NVMe service time, leaving the thread idle in the CQ polling loop most of the time. Additional threads would provide negligible benefit unless the NVMe device's internal parallelism required higher queue depth (e.g., Gen5 devices with sub-microsecond latency).

3.6 PRP List Construction

For transfers larger than 4 KB (one page), the NVMe controller needs physical addresses for each page of the destination buffer. We pre-build PRP (Physical Region Page) lists on the CPU during the setup phase:

1. Allocate a contiguous virtual buffer via `posix_memalign` + `mlock`.
2. Register with CUDA via `cudaHostRegister` for GPU accessibility.
3. Open `/proc/self/pagemap` and translate each virtual page to its physical frame number (PFN).
4. Build per-slot PRP lists: PRP1 = first page physical address; for 2-page transfers, PRP2 = second page address; for >2 pages, PRP2 = physical address of a page-aligned list of physical addresses.
5. Pass PRP1/PRP2 arrays to the GPU kernel as parameters.

This design amortizes the costly pagemap translation over the entire benchmark run, keeping the GPU data-path overhead minimal.

4 Implementation

GPU-NVME-DIRECT is implemented in ∼4,500 lines of C and CUDA code (excluding benchmarks and tests), organized into GPU-side device code and CPU-side host code.

4.1 GPU-Side Components

mmio_ops.cuh: Provides `mmio_write32`, `mmio_read32`, and their 64-bit variants using PTX inline assembly. Under simulation (`GPUNVME_USE_SIM`), falls back to volatile pointer dereferences.

sq_submit.cuh: Constructs 64-byte NVMe Read/Write command entries directly in the submission queue. Each field is written as an aligned 32-bit or 64-bit word to avoid misaligned access penalties. CDW0 (containing the opcode and command ID) is written as a single atomic 32-bit store.

cq_poll.cuh: Implements phase-bit-based completion detection. Reads CQ DW3 (offset 12, naturally aligned) as a single 32-bit word, extracts the phase bit and command ID via bit shifts. Advances the CQ head pointer with wraparound and phase flip.

queue_state.cuh: Defines the `gpu_nvme_queue` struct that encapsulates all state needed by a GPU kernel to manage an NVMe I/O queue pair independently: SQ/CQ pointers, doorbell register pointers (BAR0 MMIO), queue dimensions, current tail/head positions, phase bit, and namespace information.

4.2 Host-Side Components

controller.c: Full NVMe controller initialization per the NVMe specification: disable controller (CC.EN=0), wait CSTS.RDY=0, configure admin queues (AQA, ASQ, ACQ), set CC (command set, page size, queue entry sizes), enable (CC.EN=1), wait CSTS.RDY=1, issue Identify Controller and Identify Namespace admin commands.

io_queue.c: Creates I/O queue pairs via admin commands (Create I/O CQ, Create I/O SQ). Allocates queue memory using `posix_memalign` with `mlock` to guarantee page alignment (discovered that `cudaMallocHost`'s suballocator may return sub-page offsets, violating NVMe's page-alignment requirement for queue base addresses).

dma_alloc.c: Resolves virtual-to-physical address translation via `/proc/self/pagemap`. Provides PRP list allocation and construction for multi-page transfers.

bar_map.c: Maps NVMe BAR0 via `mmap` of the sysfs `resource0` file with `O_SYNC` to ensure uncacheable access. Registers the mapping with the GPU via `cudaHostRegisterIoMemory | cudaHostRegisterMapped`.

4.3 NVIDIA Driver Patch

On Linux 6.17, `cudaHostRegisterIoMemory` fails because the NVIDIA open-source kernel module's PFN resolution path calls `nv_follow_pfn()`, which delegates to `follow_pfn()`—a function removed in Linux 6.12 (commit `233eb0bf3b94`). The fallback `nv_follow_flavors()` unconditionally returns −1.

We patched `os-mlock.c` to compute the PFN from the VMA's `vm_pgoff` field for `VM_PFNMAP` mappings:

```
if (vma && (vma->vm_flags & VM_PFNMAP)) {
    *pfn = vma->vm_pgoff +
      ((address - vma->vm_start) >> PAGE_SHIFT);
    return 0;
}
```

Table 1: Hardware configuration

Component	Detail
GPU	NVIDIA RTX 3090 (GA102, 24 GB, PCIe 3.0 ×8[†])
CPU	AMD Ryzen 7 5800X (8C/16T, 3.8–4.7 GHz)
Platform	AMD B450, DDR4-3200 (48 GB)
NVMe (test 1)	WD SN530 1 TB, PCIe 3.0 ×4 (VFIO)
NVMe (test 2)	WD SN740 512 GB, PCIe 3.0 ×4 (VFIO)
NVMe (system)	Samsung 980 PRO 500 GB, PCIe 4.0 ×4
OS	Ubuntu 25.10 (kernel 6.17.0)
CUDA	13.1, Driver 590.48.01 (open, patched)

[†]B450 shares PCIe lanes between the primary ×16 slot and M.2_1: when an M.2 NVMe is populated, the GPU negotiates PCIe 3.0 ×8 (∼6.5 GB/s host-to-device bandwidth). The SN740 is a PCIe 4.0 device but operates at Gen3 speeds on this platform.

Listing 4: NVIDIA driver PFN resolution patch

This works because `io_remap_pfn_range()` (used by PCI BAR mmap) stores the starting PFN in `vm_pgoff` and creates contiguous mappings. The GPL-only `follow_pfnmap_start()` API cannot be used by the MIT-licensed NVIDIA module.

4.4 VFIO Device Setup

The test NVMe (WD SN530) is passed through to userspace via VFIO in no-IOMMU mode. After unbinding from the kernel NVMe driver and binding to `vfio-pci`, the device falls to D3 power state. We restore it to D0 and re-enable Memory Space and Bus Master via `setpci`:

```
echo on > /sys/bus/pci/devices/$BDF/power/control
setpci -s $BDF 0x84.W=0x0008   # Force D0
setpci -s $BDF COMMAND=0x0006 # Mem+BusMaster
```

Listing 5: VFIO setup and power state recovery

5 Evaluation

5.1 Experimental Setup

Table 1 summarizes our hardware configuration.

Note on device asymmetry. The detailed per-block-size benchmarks (Tables 2–3) use the WD SN530 (PCIe 3.0 ×4, ∼3,400 MB/s theoretical max) as the VFIO device. The CPU baselines (memcpy, pinned, cuFile) use the Samsung 980 PRO (PCIe 4.0 ×4, ∼7,000 MB/s theoretical max) because the SN530 has no kernel device path. Despite this 2× asymmetry favoring the CPU baselines, GPU-NVME-DIRECT outperforms them due to its ability to pipeline I/O operations. We additionally validate on a WD SN740 (PCIe 4.0 device, Gen3 link on B450), which achieves 3,350 MB/s sustained—near the Gen3 ×4 theoretical maximum of ∼3,400 MB/s.

To enable fair comparison, we normalize throughput as a fraction of each device's theoretical PCIe link bandwidth. At 256 KB/QD=32, GPU-NVME-DIRECT achieves 2,666/3,400 = **78.4%** of its Gen3 link on the SN530 and 3,350/3,400 = **98.5%** on the SN740, while `cpu_memcpy` achieves 1,731/7,000 = **24.7%**, `cpu_pinned` reaches 1,409/7,000 = 20.1%, and cuFile 1,396/7,000 = 19.9%. The GPU-initiated path thus extracts 3–4× more of the available link bandwidth than CPU-mediated methods. This gap is explained by Little's Law (§3.5): saturating NVMe bandwidth requires QD = BW × latency, and only GPU-NVME-DIRECT maintains sufficient queue depth from the device's perspective.

Methodology: Each configuration (block size × queue depth) is run 3 times. We measure per-completion latency using the GPU clock (`clock64()`), wall-clock time, and CPU utilization via `/proc/stat`. All tests use sequential access patterns. Block sizes range from 4 KB to 512 KB; queue depths from 1 to 32.

Table 2: Throughput (MB/s) at QD=32 by block size. GPU-NVME-DIRECT uses the slower SN530; CPU baselines use the faster 980 PRO. Despite this, GPU-NVME-DIRECT achieves the highest throughput at all block sizes.

Block Size	gpu_direct	cpu_memcpy	cpu_pinned	cuFile
4 KB	**336**	180	153	151
16 KB	**1,393**	626	516	510
64 KB	**2,111**	1,373	1,152	1,144
256 KB	**2,666**	1,731	1,409	1,396
512 KB	**2,634**	2,008	1,700	1,696

Table 3: Throughput scaling factor (QD=32 / QD=1)

Block Size	gpu_direct	cpu_memcpy	cpu_pinned	cuFile
4 KB	1.95×	1.08×	0.96×	1.01×
16 KB	2.27×	1.00×	0.98×	1.00×
64 KB	1.41×	1.10×	0.98×	1.00×
256 KB	1.65×	1.00×	0.99×	1.01×
512 KB	1.98×	1.02×	1.00×	0.99×

5.2 Throughput

Figure **??** shows throughput scaling with queue depth for each method.

Key finding 1: Queue depth scaling. GPU-NVME-DIRECT shows dramatic throughput improvement with queue depth: 2.0× at 4 KB (173 to 336 MB/s) and 2.3× at 16 KB (615 to 1,393 MB/s). In contrast, CPU-mediated methods show *no meaningful queue depth scaling*—their throughput is flat across QD=1 to QD=32 because the CPU serializes I/O operations through `memcpy`.

Key finding 2: Link bandwidth utilization. At 256 KB/QD=32, GPU-NVME-DIRECT achieves 2,666 MB/s on the SN530 (78% of Gen3 ×4 link) and 3,350 MB/s on the SN740 (99% of Gen3 ×4 link). The normalized comparison above shows that CPU methods utilize only 20–25% of their faster Gen4 link, confirming that the GPU-initiated path is fundamentally more efficient at extracting device bandwidth.

Key finding 3: Advantage despite slower hardware. Even though the CPU baselines use a 2× faster SSD, GPU-NVME-DIRECT achieves 1.3–2.2× higher throughput across all block sizes at QD=32. Per the Little's Law analysis (§3.5), this is because the CPU-mediated path effectively operates at QD=1 from the device's perspective, regardless of the application-level queue depth.

5.3 Queue Depth Scaling Factor

Table 3 quantifies the scaling advantage. The GPU's ability to maintain 32 in-flight commands from a single thread is the fundamental differentiator.

5.4 Latency

Table 4 shows median per-operation latency. At QD=1, each measurement represents the full NVMe round-trip; at QD=32, it measures the inter-completion gap (time between successive completions arriving at the CQ).

At 4 KB/QD=32, the inter-completion latency is only 1.5 µs, demonstrating that the GPU's CQ polling loop is extremely efficient. The latency standard deviation at this configuration is 0.23 µs, indicating highly consistent completion timing.

At QD=1, GPU-NVME-DIRECT achieves lower latency than CPU methods for small blocks (10.6 µs vs. 19.6 µs for `cpu_memcpy` at 4 KB), a 46% reduction. This is because the GPU writes the doorbell directly via MMIO, avoiding the CPU's system call, I/O scheduler, and interrupt handling overhead.

5.5 CPU Utilization

The CPU utilization for GPU-NVME-DIRECT (8.5% average) reflects CUDA runtime overhead (kernel launch, synchronization), *not* I/O data path work. The CPU is entirely free to perform other tasks (inference computation, preprocessing)

Table 4: Median latency (μs) for GPU-NVME-DIRECT

Block Size	QD=1	QD=4	QD=16	QD=32
4 KB	10.6	1.70	1.49	1.50
16 KB	14.2	15.5	1.48	1.47
64 KB	29.0	33.4	13.0	14.0
256 KB	126	69	63	63
512 KB	412	154	141	141

Table 5: CPU utilization (%) across all configurations

Method	Mean	Min	Max
gpu_direct	8.5	5.6	15.0
cpu_memcpy	4.3	0.0	12.2
cpu_pinned	4.7	0.0	12.5
cuFile	5.1	2.1	10.4

during GPU-driven I/O. In contrast, the CPU methods block a thread on I/O completion, making that thread unavailable for computation despite showing lower aggregate CPU utilization across all 16 logical cores.

5.6 cuFile on Consumer Hardware

A notable finding is that NVIDIA cuFile (GPUDirect Storage API) performs identically to `cpu_pinned` on our GeForce RTX 3090 hardware. Across all 60 configurations, the two methods differ by less than 3% on average. This confirms that GDS falls back to a CPU-mediated path on consumer GPUs, providing no hardware acceleration. GPU-NVME-DIRECT fills this gap by providing genuine GPU-initiated I/O on hardware that GDS does not support.

5.7 Large Sequential Reads

For the LLM inference use case, we measured throughput for large sequential reads representative of loading model layers:

The SN530 peaks at 2,734 MB/s at 128 MB, dropping to 2,127 MB/s at 669 MB due to thermal throttling during sustained reads. The SN740 sustains 3,350 MB/s even at 8.6 GB—near the Gen3 ×4 theoretical maximum—demonstrating that GPU-NVME-DIRECT can fully saturate the PCIe link for layer-sized transfers.

5.8 LLM Inference Performance

To contextualize the I/O throughput results, we present both *projected* token rates from raw I/O bandwidth and *measured* results from integrating GPU-NVME-DIRECT into a streaming inference engine (ntransformer) for Llama 3.1-70B in Q6_K quantization (∼42 GB, 80 layers, ∼669 MB/layer).

Table 7 shows projected rates for the pure I/O-bound regime where each token requires streaming the entire model from NVMe (no caching).

Measured integration results. We integrated GPU-NVME-DIRECT into ntransformer, a streaming LLM inference engine with tiered caching (VRAM/RAM/NVMe). On the SN740, GPU-NVME-DIRECT achieves 0.06 tok/s for 70B Q6_K with all 80 layers streamed from NVMe per token—a 2× improvement over mmap+memcpy (0.03 tok/s) and consistent with the projected I/O bandwidth. The CPU is entirely free during NVMe transfers, enabling concurrent compute overlap.

When combined with tiered caching (29 layers in VRAM, 51 in RAM, 0 on NVMe), the system achieves 0.20 tok/s. The bottleneck shifts from NVMe bandwidth to PCIe H2D transfers at Gen3 ×8 (∼6.5 GB/s, 103 ms per 669 MB layer). Adding adaptive layer skipping (threshold 0.98, skipping 16–20 middle layers) yields 0.27 tok/s, and combining with Q4_K_M quantization reaches 0.50 tok/s—a 17× improvement over the CPU-mediated NVMe baseline.

Important context. The absolute token rates for NVMe-only streaming are inherently low because the entire model must transit from storage per token. For reference, llama.cpp achieves ∼1 tok/s for 70B models when weights reside in GPU VRAM across two GPUs (tensor parallelism), and FlexGen [8] achieves 1 tok/s for OPT-175B with aggressive CPU/disk offloading. The contribution of GPU-NVME-DIRECT is not the absolute token rate but the **removal of the**

Table 6: Large sequential read throughput (QD=32, 512 KB blocks)

Transfer Size	SN530 (MB/s)	SN740 (MB/s)
4 MB	2,122	—
128 MB	2,734 (peak)	—
669 MB (1 layer)	2,127	3,350
8.6 GB sustained	—	3,350

Table 7: Llama-70B inference: projected (I/O-bound, full model per token) and measured (integrated with tiered caching)

Data Path	BW	tok/s
Projected (80 layers from NVMe per token)		
mmap+memcpy	1.5 GB/s	0.028
GPU-NVME-DIRECT (SN530)	2.1 GB/s	0.039
GPU-NVME-DIRECT (SN740)	3.35 GB/s	0.063
Measured (ntransformer integration)		
GPU-NVME-DIRECT NVMe-only (SN740)	3.35 GB/s	0.06
GPU-NVME-DIRECT NVMe-only (SN530)	2.1 GB/s	0.04
Tiered VRAM/RAM (no NVMe)	6.5 GB/s	0.20
Tiered + layer skip (0.98)	6.5 GB/s	0.27
Tiered + Q4_K_M + skip	6.5 GB/s	**0.50**

CPU from the storage data path: measured CPU utilization during NVMe-direct transfers is ~0% versus 100% (one core saturated) for mmap+memcpy.

For Q8_0 quantization (~70 GB) that *exceeds* our 48 GB RAM, NVMe streaming via GPU-NVME-DIRECT is the only viable path without adding system memory. Combined with NanoQuant [18] (sub-1-bit quantization compressing 70B to ~8 GB), the data volume per token drops from 42 GB to ~8 GB, projecting ~0.5 tok/s from NVMe alone—matching the tiered+optimized result but without requiring RAM caching.

6 Engineering Challenges

Developing GPU-NVME-DIRECT required solving several non-trivial challenges at the intersection of GPU programming, NVMe protocol, PCIe topology, and kernel driver internals. We document these for reproducibility.

Bug #4 deserves special attention. NVMe controllers may reorder completions within a queue—returning CQE for command B before command A even if A was submitted first. Our initial implementation used `cq_poll_for_cid`, which discarded CQEs not matching the expected command ID. When the pipeline had multiple in-flight commands, this caused valid completions to be lost, leading to timeouts. The fix was to poll for *any* completion (`cq_poll_completion`) and track which command IDs have completed, accepting completions in whatever order the NVMe controller delivers them.

7 Related Work

GPU-Initiated Storage Access. BaM [1] (ASPLOS '23) is the closest prior work, demonstrating GPU-orchestrated on-demand storage access. BaM provisions NVMe queues directly in GPU HBM via GPUDirect RDMA and uses GPUDirect Async for doorbell writes, achieving 45.8M IOPs across 10 SSDs. GIDS [13] (VLDB '24) extends BaM to GNN training with a dynamic storage access accumulator and window buffering, reaching 582× speedup over DGL on terabyte-scale graphs. Both systems require enterprise GPUs (A100) with native P2P DMA support, PCIe expansion chassis, and custom kernel modules. GPU-NVME-DIRECT differs fundamentally: it works on consumer hardware ($2,000 vs. $10,000+) where full P2P is unavailable, using only PCIe posted writes for doorbell access and host pinned memory for data staging.

GPUfs [5] pioneered GPU-initiated file I/O but routes all I/O through a CPU-side daemon. DRAGON [7] extends GPU memory to NVM via page faults, introducing high per-fault latency. SPIN [6] integrates P2P DMA into the Linux page cache but requires PCIe switch topology and CPU-initiated I/O. Phoenix [20] refactors the GDS I/O stack, mapping GPU memory into PCIe BAR space for direct NVMe access—but remains limited to enterprise GPUs.

Table 8: Critical bugs discovered and resolved

#	Bug	Root Cause & Fix
1	`cudaHostRegisterIoMemory` fails	`follow_pfn()` removed in kernel 6.12. Patched `os-mlock.c`: compute PFN from `vm_pgoff`.
2	GPU reads return `0xFFFFFFFF`	AMD root complex drops non-posted P2P TLPs. Use Tier 1 (writes only).
3	Admin commands hang	Admin CQ not page-aligned (`cudaMallocHost` suballocator). Allocate $\geq$4096 bytes.
4	Pipeline depth $\geq$4 timeouts	NVMe completions arrive out-of-order; poll discarded non-matching CQEs. Use `cq_poll_completion` (accept any CID).
5	Post-warmup timeouts	Resetting queue state while NVMe controller retains internal pointers. Leave queue state rolling.
6	GPU reads crash NVMe link	Accumulated CmpltTO errors from failed P2P reads. Avoid GPU reads; power cycle to recover.
7	64-bit MMIO write corruption	Non-atomic 64-bit write to BAR0. Split into two 32-bit writes (low word first per NVMe spec).

GPUDirect Storage. NVIDIA's GDS [2] enables direct NVMe-to-GPU DMA; as of CUDA 12.8, it supports P2P DMA via the upstream kernel PCI P2PDMA infrastructure on x86_64. However, GDS requires enterprise GPUs, certified storage, and the `nvidia-fs` kernel module. Our evaluation confirms that cuFile provides no benefit on GeForce hardware, falling back to a CPU-mediated path identical to `cpu_pinned`.

Userspace NVMe Drivers. SPDK [3] provides high-performance CPU-driven userspace NVMe access via polling. Our work extends the userspace NVMe concept to the GPU: instead of CPU cores polling SQ/CQ, a CUDA thread does. libnvm [4] enables GPU DMA from NVMe but requires a kernel module and P2P-capable platforms.

LLM Inference on Constrained Hardware. A rapidly growing body of work addresses LLM inference with limited GPU memory, all using CPU-mediated storage I/O: FlexGen [8] (ICML '23) offloads weights, activations, and KV cache across GPU/CPU/disk using linear programming, achieving OPT-175B on a single 16 GB GPU at 1 tok/s. PowerInfer [11] (SOSP '24) exploits activation sparsity (power-law neuron distribution) to preload hot neurons on GPU and compute cold neurons on CPU, reaching 11.69× over llama.cpp on RTX 4090. PIPO [15] designs a fine-grained offloading pipeline for consumer devices, increasing GPU utilization from <40% to >90% on RTX 3060. HeadInfer [16] offloads KV cache to CPU RAM at per-attention-head granularity, enabling 4M-token inference with Llama 3-8B on a single RTX 4090. ServerlessLLM [14] (OSDI '24) exploits the full near-GPU storage hierarchy for fast checkpoint loading, achieving 12 GB/s from RAID0-NVMe with optimized `O_DIRECT` and pinned memory pipelining. InstInfer [17] offloads attention computation to computational storage devices. DeepSpeed-Inference [9] supports NVMe offloading via ZeRO-Infinity. LLM in a Flash [12] targets Apple's unified memory. NanoQuant [18] compresses LLMs to sub-1-bit levels, enabling Llama 2-70B on an 8 GB consumer GPU—a technique complementary to GPU-NVME-DIRECT that would dramatically reduce the data volume needing to be streamed from NVMe.

None of these systems achieve *GPU-initiated* storage access; all route I/O through the CPU. A recent I/O characterization study [19] of LLM offloading to NVMe confirms that current frameworks do not saturate SSD bandwidth—model offloading is dominated by 128 KiB sequential reads at well below device peak. GPU-NVME-DIRECT addresses this gap by removing the CPU from the storage data path.

PCIe P2P and Interconnect Evolution. Prior work on PCIe P2P [4] has characterized P2P behavior across Intel and some AMD platforms. The Linux P2PDMA subsystem provides growing kernel support for device-to-device DMA. Looking forward, CXL memory pools [21] could provide shared memory regions accessible by both GPU and NVMe, simplifying the data path. However, none of this prior work has documented the specific asymmetry on AMD B450/Vermeer (posted writes succeed, non-posted reads cause link failure) that we characterize and exploit in GPU-NVME-DIRECT.

7.1 Positioning

Table 9 positions GPU-NVME-DIRECT against related systems. GPU-NVME-DIRECT is the only system that combines GPU-initiated I/O with consumer hardware compatibility.

Table 9: Feature comparison with related systems

System	GPU initiates	No CPU data	Consumer HW	No kmod	No ent. GPU
GPU-NVME-DIRECT					
BaM			–	–	–
GIDS			–	–	–
GDS	–		–	–	–
SPDK	–			–	N/A
ServerlessLLM	–	–	–		
PowerInfer	–	–			
FlexGen	–	–			
PIPO	–	–			

8 Discussion and Future Work

Tier 1 bandwidth ceiling. Tier 1 routes data through host pinned memory, adding one PCIe hop compared to direct NVMe-to-GPU DMA. On our B450 platform, the GPU's PCIe 3.0 $\times 8$ link to host memory provides $\sim$6.5 GB/s of bandwidth—roughly 2× the SN530's 3.4 GB/s. The NVMe device is therefore the bottleneck, not the extra hop, and Tier 1 loses no throughput from the indirection. However, the margin is tight: a Gen4 $\times 4$ NVMe at full speed (7 GB/s) would exceed our GPU link bandwidth, and even on our platform the SN740 reaches 3.35 GB/s ($\sim$52% of the GPU link). On platforms with full Gen4 $\times 16$ (25 GB/s), this headroom increases substantially. For our B450 configuration, this motivates **Tier 2**: enabling NVMe DMA directly to GPU VRAM via the tinygrad P2P patches for the NVIDIA open-source kernel module, which would eliminate the host memory intermediate and become essential when upgrading to faster NVMe devices or adding a second drive.

Why a single GPU thread suffices. Our Little's Law analysis (§3.5) shows that QD=32 provides ample headroom to saturate the SN530. The I/O management thread spends $\sim$2 µs per command (SQ construction + doorbell write + fence), far below the device service time, and resides in a tight CQ polling loop otherwise. This consumes exactly one warp slot, leaving the remaining 81 SMs (on GA102) free for tensor computation. A multi-warp design managing multiple NVMe queues could provide higher aggregate throughput for multi-device setups, but for a single NVMe device, one thread is sufficient. We plan to evaluate multi-queue parallelism with 2–4 NVMe devices to determine scaling behavior.

Bandwidth utilization in context. GPU-NVME-DIRECT achieves 78% of the PCIe 3.0 $\times 4$ link bandwidth on the SN530 and 99% on the SN740 at 256 KB/QD=32. For comparison, BaM [1] reports 90% of link bandwidth on enterprise A100 hardware with GPUDirect RDMA and queues in GPU HBM. The SN530's 78% gap is attributable to Tier 1's extra host memory hop and device-side throttling; the SN740's 99% demonstrates that GPU-NVME-DIRECT can match Tier 3-level link utilization on consumer hardware when the drive sustains its throughput. Despite the extra hop, the normalized comparison (§5) shows that GPU-NVME-DIRECT extracts 3–4× more of the available link bandwidth than CPU-mediated methods on the same platform.

Synergy with quantization. The measured 0.50 tok/s result (Table 7) already demonstrates this synergy: Q4_K_M quantization reduces per-layer size from 669 MB to $\sim$370 MB, and combined with layer skipping, the effective data volume per token drops substantially. NanoQuant [18] pushes further with sub-1-bit quantization compressing Llama 2-70B to $\sim$8 GB total. Combining GPU-NVME-DIRECT NVMe-only streaming with NanoQuant would reduce the per-token transfer from 42 GB to $\sim$8 GB, projecting $\sim$0.5 tok/s from NVMe alone—matching the tiered+optimized result but without requiring RAM caching. This synergy is bidirectional: quantization reduces I/O volume while GPU-initiated I/O reduces transfer latency, and neither technique substitutes for the other.

CXL and future interconnects. CXL 3.0 memory pools [21] could fundamentally change the P2P landscape by providing cache-coherent shared memory regions accessible to both GPU and NVMe controllers. In such architectures, GPU-NVME-DIRECT's Tier 1 host-memory approach would naturally benefit from the lower-latency CXL path. More importantly, CXL-attached NVMe devices could enable Tier 3-equivalent semantics on consumer platforms if the CXL fabric supports GPU-initiated transactions.

Safety and reliability. Operating NVMe at the register level bypasses all kernel-level safety mechanisms. Bugs in command construction can cause data corruption or device hangs (Bug #6, Table 8). We mitigate this through a simulator mode (verifying logic without hardware), comprehensive struct layout tests, and careful power management.

In production, a watchdog timer in the GPU kernel detects hangs (via `clock64()`) and returns error codes. A formal verification of the command construction path would further strengthen reliability.

Portability. GPU-NVME-DIRECT's Tier 1 approach should work on any platform where the GPU can issue PCIe posted writes to a peer device's BAR. This includes AMD EPYC (full P2P), Intel Xeon (full P2P), and AMD consumer (posted writes only). Platforms with active PCIe switches (e.g., PLX/Broadcom) should also support the necessary posted write routing. We plan to evaluate on Intel 12th/13th-gen consumer platforms, where full P2P may enable Tier 2/3 without enterprise GPUs.

End-to-end integration. We integrated GPU-NVME-DIRECT into ntransformer, a streaming inference engine with tiered caching (§5). The integration confirms that GPU-initiated I/O composes well with a double-buffering strategy: while the GPU computes attention and feed-forward on layer n, GPU-NVME-DIRECT prefetches layer $n+1$ from NVMe. The measured 0.06 tok/s for NVMe-only streaming matches the projected I/O bandwidth, validating that the primitive performs as expected in an application context. Further optimization (tiered caching, layer skipping, aggressive quantization) reaches 0.50 tok/s, demonstrating that GPU-NVME-DIRECT serves as a viable building block for practical single-GPU inference of 70B-class models.

Limitations. (1) The CPU baselines use a different (faster) NVMe device than GPU-NVME-DIRECT. The bandwidth normalization (§5) controls for this, and the SN740 results (99% link utilization) confirm GPU-NVME-DIRECT is not device-specific. (2) Write support is implemented but not benchmarked; write operations face different challenges (write amplification, garbage collection) that may affect latency profiles. (3) All measurements use sequential access patterns; random I/O workloads relevant to graph analytics [1, 13] remain unevaluated. (4) The NVIDIA driver patch (§4) ties the system to specific kernel and driver versions, limiting deployment ease.

9 Conclusion

We presented GPU-NVME-DIRECT, the first system (to our knowledge) that demonstrates GPU-initiated NVMe I/O on consumer hardware. Using a GeForce RTX 3090 on an AMD B450 platform, a single CUDA thread autonomously manages NVMe I/O queues—constructing commands, writing doorbells via PTX MMIO instructions, and polling completions—without any CPU involvement in the data path.

Our key insight is that PCIe posted writes (the only operation needed for NVMe doorbell ringing) succeed on AMD consumer platforms even where P2P reads fail. This enables a Tier 1 approach where the GPU drives the NVMe controller while all data flows through host pinned memory.

GPU-NVME-DIRECT achieves 2,666 MB/s on the SN530 (78% of link bandwidth) and 3,350 MB/s on the SN740 (99% of link bandwidth), outperforming CPU-mediated methods by up to 2.2× despite the CPU baselines using a faster NVMe device. Integrated into a streaming inference engine, GPU-NVME-DIRECT delivers 0.06 tok/s for Llama 70B from NVMe alone, and 0.50 tok/s when combined with tiered caching and quantization optimizations.

The system demonstrates that GPU-initiated storage I/O is not limited to enterprise hardware—consumer GPUs are physically capable of this operation, and the main barriers are software (driver checks, P2P topology) rather than hardware. This work opens the door to GPU-centric storage access on commodity hardware, with immediate applications in LLM inference, graph analytics, and any workload where GPU memory is insufficient to hold the full working set.

References

[1] Z. Qureshi, V. S. Mailthody, I. Gelado, S. W. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Lopatenko, and W. Hwu. BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage. In *Proc. ASPLOS*, 2023.

[2] NVIDIA Corporation. GPUDirect Storage. https://docs.nvidia.com/gpudirect-storage/, 2024.

[3] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *Proc. IEEE CloudCom*, 2017.

[4] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. K. Stensland, and C. Griwodz. SmartIO: Zero-overhead Device Sharing through PCIe Networking. *ACM Trans. Comput. Syst.*, 2021.

[5] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. *ACM Trans. Comput. Syst.*, 32(1), 2014.

[6] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *Proc. USENIX ATC*, 2017.

[7] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In *Proc. SC*, 2018.

[8] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proc. ICML*, 2023.

[9] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proc. SC*, 2022.

[10] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. SOSP*, 2023.

[11] Y. Song, Z. Mi, H. Xie, and H. Chen. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proc. SOSP*, 2024.

[12] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar. LLM in a Flash: Efficient Large Language Model Inference with Limited Memory. *arXiv:2312.11514*, 2023.

[13] J. B. Park, V. S. Mailthody, Z. Qureshi, and W. Hwu. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *Proc. VLDB Endow.*, 17(6):1227–1240, 2024.

[14] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *Proc. OSDI*, 2024.

[15] Y. Liu, J. Li, and W.-J. Li. PIPO: Pipelined Offloading for Efficient Inference on Consumer Devices. *arXiv:2504.03664*, 2025.

[16] C. Luo, C. Cai, H. Sun, J. Xiao, B. Yuan, W. Xiao, J. Hu, J. Zhao, B. Chen, and A. Anandkumar. HeadInfer: Memory-Efficient LLM Inference by Head-wise Offloading. *arXiv:2502.12574*, 2025.

[17] InstInfer: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference. *arXiv:2409.04992*, 2024.

[18] H. Chong, D. Kim, C. Kim, and M. Choi. NanoQuant: Efficient Sub-1-Bit Quantization of Large Language Models. *arXiv:2602.06694*, 2026.

[19] An I/O Characterizing Study of Offloading LLM Models and KV Caches to NVMe SSD. In *Proc. CHEOPS*, ACM, 2025.

[20] Phoenix: A Refactored I/O Stack for GPU Direct Storage without Phony Buffers. In *Proc. EuroSys*, ACM, 2025.

[21] Y. Zhong *et al.* My CXL Pool Obviates Your PCIe Switch. In *Proc. OSDI*, 2025.

[22] NVM Express Workgroup. NVM Express Base Specification, Revision 2.0. https://nvmexpress.org/specifications/, 2021.

[23] H. Touvron, L. Martin, K. Stone, P. Albert, *et al.* Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv:2307.09288*, 2023.