An Architecture for Direct File System Access on the GPU

Dvir Hilu

***Abstract*— As Graphics Processing Unit (GPU) applications become more general purpose, there is an increasing need for them to operate more independently. A major component of this is allowing GPUs to directly perform file operations, thus eliminating their reliance on the Central Processing Unit (CPU) for data accesses. This paper discusses the GPU-side implementation of Virtio-Drive, a project aiming to enable direct file-system access on a GPU. An interface layer with a POSIX-like Application Programming Interface (API) is provided for use by GPU applications. Below it is a FUSE layer implementing the FUSE protocol for file system commands, and a driver layer used to interact directly with the file system over unified memory. While an efficiently running Virtio-Drive requires a file system implementation built into the Solid State Drive (SSD) firmware, a file system emulator running on a CPU is currently used to demonstrate the system as a proof-of-concept.**

***Index Terms*— GPUs, File System, Hardware Accelerators**

I. Introduction

With increasing demand for large parallelization in modern compute workloads, GPUs are being used more and more frequently as general purpose processors as opposed to only for graphics applications [1], [2], [3]. As pointed out by Silberstein [4], while many modern GPU applications operate on large amounts of data and therefore require data to be transferred from the file system, GPU applications still require data to be explicitly copied in at kernel launch. A mechanism that allows the GPU direct access to the file system would greatly improve the efficiency of these applications by eliminating kernel launch overhead, shortening the datapath, and freeing the CPU from having to perform file I/O tasks.

Nider and Fedorova [5] further note that many modern high performance compute workloads are mainly performed on hardware accelerators while the CPU simply acts as a centralized coordinator. They proposed instead a decentralized, CPU-less hardware architecture in which devices operate independently and communicate with each other to coordinate task completion.

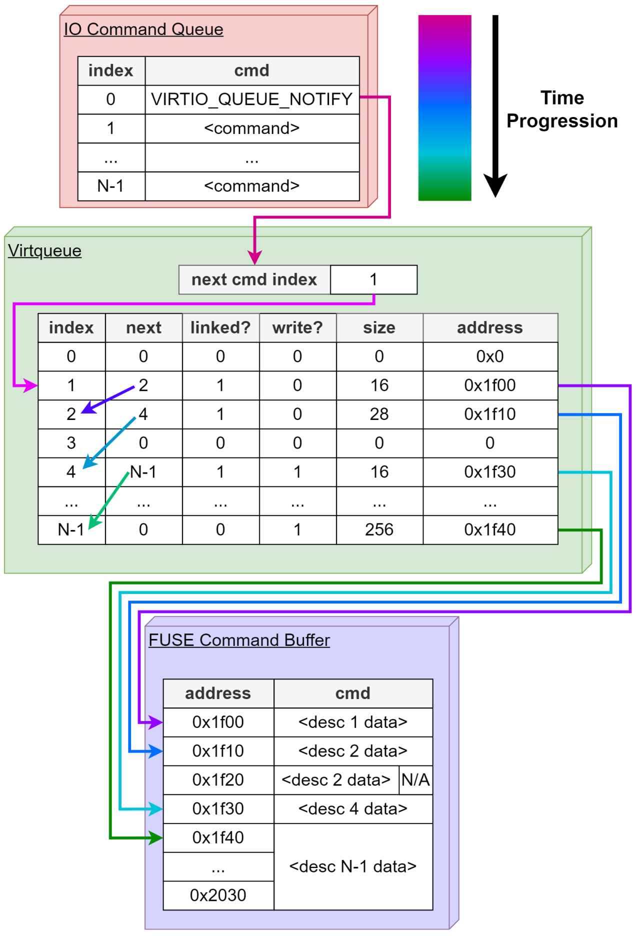
The first step in the realization of such a hardware architecture is Virtio-Drive: a system that allows communication between the GPU and SSD and provides the GPU direct storage access without the participation of the CPU. Virtio-Drive is planned to include a file system implementation directly embedded in the SSD firmware, as well as a GPU implementation that can interact with such a file system. In this paper we describe the GPU-side architecture, implementation, and design considerations for Virtio-Drive.

II. Previous Works

There are a couple of existing solutions that already allow GPU programmers to access storage while the GPU application is running (as opposed to pre-fetching the data and copying it in at kernel launch). NVIDIA provides a method for GPUs to communicate directly with 3rd party devices over PCIe with GPUDirect RDMA [6]. However, the provided API is very low level and does not allow to access files directly, instead requiring explicit memory addresses for the transfer.

Silberstein introduced GPUfs [4], which does provide an easy to use file API for accessing storage directly while the GPU application is running, but relies on a CPU daemon to process file I/O requests and copy them into the GPU. Many of the design decisions made in Virtio-Drive were inspired by decisions made in GPUfs, such as the file system API and some read/write optimizations which will be discussed later in Section IV.

Virtio-Driver aims to reconcile the benefits of GPUDirect and GPUfs by providing a system with an easy to use file API for GPU programmers and an implementation that bypasses the CPU and directly accesses files from the SSD. In addition to the GPU implementation described in this paper, achieving this goal requires hardware modifications and a file system that runs directly on the SSD firmware. As such a system is not built yet, the feasibility of the GPU implementation is first explored by interacting with a file system emulator running on the CPU.



index	cmd
0	VIRTIO_QUEUE_NOTIFY
1	<command>
...	...
N-1	<command>

next cmd index	1

index	next	linked?	write?	size	address
0	0	0	0	0	0x0
1	2	1	0	16	0x1f00
2	4	1	0	28	0x1f10
3	0	0	0	0	0
4	N-1	1	1	16	0x1f30
...	...	...	...	...	...
N-1	0	0	1	256	0x1f40

address	cmd
0x1f00	<desc 1 data>
0x1f10	<desc 2 data>
0x1f20	<desc 2 data> N/A
0x1f30	<desc 4 data>
0x1f40	
...	<desc N-1 data>
0x2030	

Fig. 1. Communication model for Virtio devices. Time progression of structure traversal is presented with arrow colours: starting with the purple arrow stemming from the IO Command Queue and ending with the green arrow pointing to the last descriptor data in the FUSE Command Buffer.

This paper was submitted June, 2022.

D. Hilu is an Engineering Physics undergraduate student at the University of British Columbia, Vancouver, BC, Canada V6T 1Z1 (e-mail: dvirhilu16@gmail.com).

Previous work on Virtio-Driver [7] focused mostly on establishing communication between the GPU and the file system emulator over unified memory. Virtio, a standardized interface for I/O communication and virtualization used largely in virtual machines [8], was chosen as the standard for communicating with the file system emulator, and will eventually be implemented as part of the SSD file system solution (hence the name Virtio-Drive). The actual file system emulator was chosen to be Virtiofs [9].

The Virtio model for device communication is shown in Figure 1. The structures shown in the figure (IO Command Queue, Virtqueue, FUSE Command Buffer) are shared between the device (either the emulator or the SSD) and the GPU. After updating the structures with the file system command, the GPU sends a notification signal to the device via the IO Command Queue. Upon receiving the notification, the device grabs a descriptor from the virtqueue. If the file system command is comprised of multiple descriptors (as is the case with virtiofs), these can be chained together as a linked list. The virtqueue descriptors each contain an address pointing to the data location in the FUSE Command Buffer and the size of the data.

III. Architecture Overview

Figure 2 presents an overview of the Virtio-Drive architecture with file system emulation. The GPU implementation of Virtio-Drive consists of three layers: the Interface Layer, the FUSE Layer, and the Driver Layer. The interface layer is directly called by the GPU application, and defines the file API. The FUSE layer manages the per-block file table and configures the FUSE protocol commands used to interact with Virtiofs. The driver layer implements all operations relating to the Virtio queues (Virtqueues) and sends IO commands to the file system device. The IO commands, FUSE commands, and Virtqueues are stored in a unified address space which can be accessed by both the GPU and the file system device. Lastly, a CPU thread works to interface between Virtiofs and the GPU.

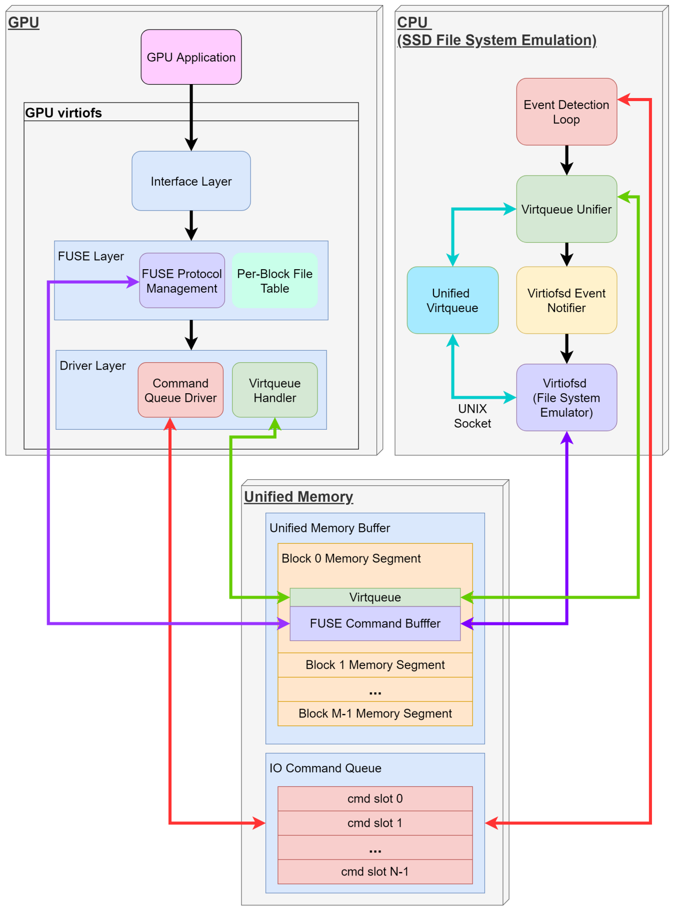


Fig. 2. Virtio-drive system architecture with Virtiofs emulator

IV. Implementation

A. Interface Layer

The Virtio-Drive API, summarized in Table I, presents a POSIX-like interface for GPU applications to use. Unlike CPU file APIs which allow CPU threads to open and operate on files independently, the Interface Layer requires different semantics in consideration with the hardware architecture of a GPU.

An NVIDIA GPU is comprised of groups of Streaming Processors (SMs), each containing 32 cores running separate threads. The CUDA programming model groups threads in blocks, with all threads in a single block sharing the same SM throughout the lifetime of the application. If blocks are larger than 32 threads, they are further divided into warps, which are groups of 32 threads that will run simultaneously on the same SM [10]. To achieve such a high level of parallelism, SMs follow a Single-Instruction, Multiple-Thread (SIMT) execution model. Under an SIMT model, each thread in an SM executes *the same instruction*, though they can operate on separate data [11].

A major implication of the SIMT model is that diverging threads do not execute in parallel: instead, the SM must execute through *all execution paths* of the threads it is running sequentially, turning threads active/inactive as needed [11]. Allowing threads to operate on and access files directly would result in diverging paths and significantly reduce the performance of the GPU application. As such, in line with design decisions made in GPUfs [4], the Virtio-Drive file API supports file calls at the block granularity rather than the thread granularity. Note that block granularity was chosen over warp granularity due to ease of implementation for this proof of concept. Further work on the system could reduce the granularity down to warp level while still avoiding thread divergence.

The APIs of Virtio-Drive and GPUfs differ in file offset semantics. Anticipating applications where a large amount of blocks concurrently open the same file, GPUfs opted to use a global file table, where entries correspond to files, as opposed to descriptors. This would avoid the overhead associated with multiple open calls of the same file which get largely serialized. To allow for file descriptors to be shared across all blocks and avoid tracking per-open states, GPUfs does not manage seek pointers for its files, and instead file offsets must be explicitly specified to its file API functions [4].

Unlike GPUfs, Virtio-Drive is built to be optimized for read-modify-write applications, in which blocks open different files, operate on their data, and write them back to a file (for example, preprocessing machine learning datasets, image resizing, or event stream processing) [12]. As a result, sharing file descriptors brings little benefit and does not reduce access overhead. File tables are instead managed on a per-block basis, allowing Virtio-Drive to track seek pointers internally and provide an API more similar to the POSIX file API which does not require explicit offsets.

B. FUSE Layer

The FUSE Layer manages the per-block file table and packages commands according to the FUSE protocol. As the file tables are managed on a per-block basis, we can exploit the GPU's memory hierarchy by storing the file table in shared memory. CUDA shared memory constitutes a portion of the L1 cache, and is accessible by all threads of the block running on the current SM. Since the memory

Function	Description
g_readdir	open current directory and read its contents
g_open	open file and add its file descriptor to the current block's file table
g_close	close file and remove its file descriptor from the current block's file table
g_fstat	query the size of the file
g_read	read file and increment seek pointer by amount read
g_write	write to file and increment seek pointer by amount written
g_lseek	Change the position of the file's seek pointer
g_dup2	Duplicate the file descriptor, allowing to track separate seek pointers in the file

TABLE I
VIRTIO-DRIVER FILE OPERATIONS API. ALL THREADS IN A BLOCK MUST CALL THESE FUNCTIONS WITH IDENTICAL PARAMETERS.

is placed on chip, it provides significantly lower access latency than global memory. Furthermore, the memory is not accessible by threads of other blocks (as they run on other SMs and thus access a different shared memory), which allows for isolation of file tables between blocks [13].

The FUSE protocol [14] follows a clients-server model, where a program (cilent) sends requests to the file system (server) and receives a reply based on the performed operation. FUSE packages commands into multiple descriptors (also known as FUSE headers). All request start with a FUSE in header followed by optional headers or data, and each reply starts with a FUSE out header followed by optional headers or data. The management of FUSE headers is largely done serially since it requires strictly ordered operations and control flows, though impacts to performance should be minimized since these are short code segments that operate on small amounts of data.

Virtio-Drive can exploit the parallelism of the GPU for operations like read or write, where data transfers could potentially be large. While the general construction of the FUSE headers is still done serially, data copying in/out of the buffer is done in multi-threaded fashion using coalesced memory operations. NVIDIA GPUs can coalesce requests of contiguous, aligned addresses into a single memory transaction of 32, 64, or 128 bytes in size. Thus, if each thread in a warp makes a 4 byte request amalgamating to a single, aligned chunk of 128 bytes, the requests are grouped into a single memory transaction [13]. The throughput of memory operations can be significantly increased this way, both by parallelizing the load/store commands across all threads in a warp and by sending only a single memory transaction for these requests, thereby significantly reducing the memory IO bottleneck. An example of how coalesced memory operations are used in the FUSE layer is presented in Figure 3. Memory coalescing is also utilized by GPUfs, though to copy data to/from its buffer cache [4], a structure that was deemed unnecessary overhead by Virtio-Drive due to the fact that the read-modify-write access pattern it is optimized for does not access a file more than once.

C. Driver Layer and the Unified Address Space

The low-level device communication over the unified address space is handled by the Driver Layer. The unified address space is split in the current implementation into two sections: The IO Command Queue and the Unified Memory Buffer. The IO Command Queue is used to read/write to specific Virtio addresses indicating specific command types. During device initialization, the Driver Layer writes to these addresses to negotiate device parameters and link virtqueue addresses to the device. Once the device is initialized, the IO Command Queue is used to notify the device that a new FUSE request has been made. Once a hardware solution is available, the IO Command Queue section of unified memory will likely be replaced with read/writes to bus addresses to communicate with the SSD.

To fully utilize the parallelism of the GPU, each block is allocated its own segment within the Unified Memory Buffer, with each segment containing its own Virtqueue and its own FUSE Command Buffer. The size of each block's segment is determined by the total allocated size of the Unified Memory Buffer at startup as well as the negotiated maximum request size. Normally, each block is given a FUSE Command Buffer as big as its negotiated maximum request size. Should the total size of all segments exceed the size of the Unified Memory Buffer, they are instead divided by creating equally divided, aligned segments in the available memory.

The virtqueues and FUSE Command Segments are unique to each block for the same reason as the file table: accesses to the Virtqueue must be synchronized. Allowing blocks to modify the same shared memory segment would require either expensive atomic operations or creating a sequential bottleneck which would significantly decrease throughput. Allocating each block their own memory segment allows for a lock-free system that could operate completely asynchronously.

After the FUSE layer updates the FUSE Command Buffer, it passes the relevant addresses and data sizes to the Driver Layer to update the Virtqueues. Upon updating the Virtqueues, the driver layer sends a notification IO Command to the device for processing. The notification command passes the block ID as a value, thereby pointing the device to the correct memory segment. The GPU thread block then proceeds to poll the output header of the FUSE command for the reply from the device.

D. File System Emulator

Virtiofs itself operates on a single Virtqueue, and as a result, an interface is required between the GPU and Virtiofs that combines the GPU's Virtqueues into a single, unified Virtqueue that Virtiofs can query. In this proof of concept implementation, a CPU thread performs this task serially (adding commands to the unified Virtqueue in the order in which it receives IO Commands). Thus, the system in its current form does not utilize the full extent of the parallelism of the GPU and performance data beyond a reduction in CPU workload as measured in [12] cannot be reliably collected without the hardware solution.

V. RECOMMENDATIONS

As mentioned in Section IV-D, the current implementation provides a high potential to bottleneck the application by processing multiple parallel I/O channels in the GPU (the per-block virtqueues and buffers) serially. As such, exploring a file system implementation and bus system that can handle these parallel channels is important for maintaining a high performance.

One potential way in which these bottlenecks can be bypassed on the GPU side is by exploring a file system command execution model similar to out-of-order execution in CPUs. Many file system operations do not require an immediate response, and the GPU should be able to continue executing commands until data dependencies are detected (such as in the case of read operations).

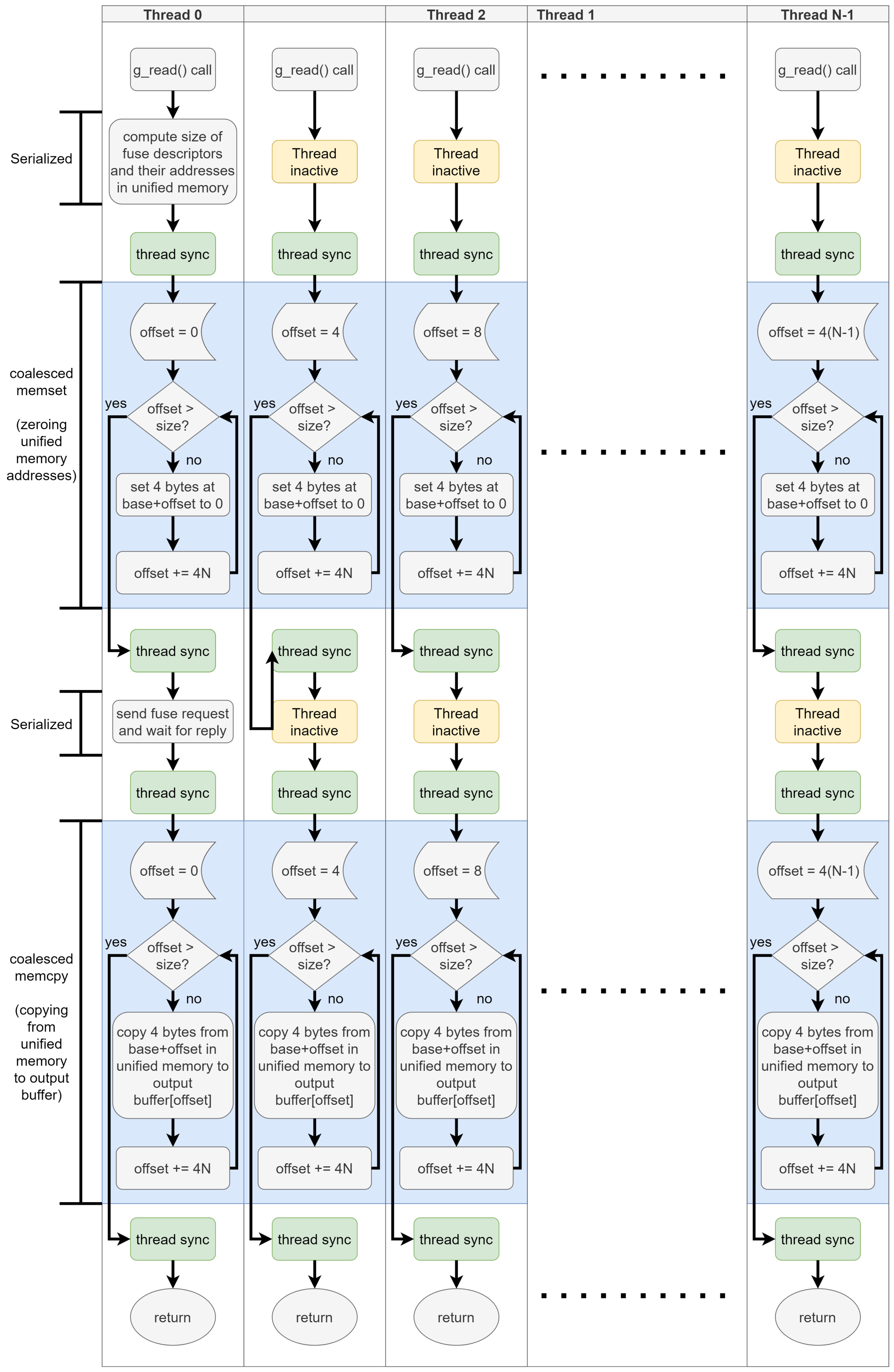

Thread 0
Thread 2
Thread 1
Thread N-1
Serialized
coalesced memset (zeroing unified memory addresses)
Serialized
coalesced memcpy (copying from unified memory to output buffer)
g_read() call
compute size of fuse descriptors and their addresses in unified memory
Thread inactive
thread sync
offset = 0
offset = 4
offset = 8
offset = 4(N-1)
offset > size?
yes
no
set 4 bytes at base+offset to 0
offset += 4N
send fuse request and wait for reply
copy 4 bytes from base+offset in unified memory to output buffer[offset]
return

Fig. 3. Logic flow for the g_read function in the FUSE layer. Serialized operations are abstracted away and emphasis is on how control logic is performed for coalesced memory operations.

Additionally, the current architecture presents an inherent trade-off between the maximum size of a read/write request and the amount of memory utilization in the GPU (as each block's FUSE Command Buffer is sized to the maximum request size). Depending on hardware capabilities, increasing the maximum request size could present higher performance as less total requests need to be made to the file system, though coming at the cost of higher overhead. The relationship between these two parameters should be explored once a hardware implementation is available,

VI. Conclusion

The GPU-side architecture and implementation described in this report provides the first half in creating an efficient mechanism for GPUs to access storage independently. Such a step is crucial in supporting the hardware architecture porposed by Nider and Fedorova [5], though even more generally, its need is becoming ever more apparent in an age where GPUs are used for more general purpose tasks [1], [2], [3]. To further enhance performance, future work on Virtio-Drive should explore an SSD file system implementation that can efficiently handle the parallel I/O channels in the GPU, a potential out-of-order execution-like model for file system commands, and optimization of memory utilization vs. request size.

Acknowledgment

This work was supported by Professor Alexandra (Sasha) Fedorova and Joel Nider, Department of Electrical and Computer Engineering, University of British Columbia.

References

[1] R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography," in *Cryptographic Hardware and Embedded Systems – CHES 2008* (E. Oswald and P. Rohatgi, eds.), vol. 5154, pp. 79–99, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
[2] R. D. Evans, L. Liu, and T. M. Aamodt, "JPEG-ACT: Accelerating Deep Learning via Transform-based Lossy Compression," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020.
[3] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data," *BMC Bioinformatics*, vol. 20, p. 520, Oct. 2019.
[4] Mark Silberstein et al., "Gpufs: Integrating a file system with gpus," in *ACM Transactions on Computer Systems*, vol. 32, 2014.
[5] Joel Nider and Alexandra (Sasha) Fedorova, "The last cpu," in *Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021.
[6] NVIDIA, *Developing a Linux Kernel Module using GPUDirect RDMA*, May 2022.
[7] Joel Nider, "Virtio-drive." https://wiki.ubc.ca/Virtio-drive.
[8] "Virtual i/o device (virtio) version 1.1." https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf, Apr 2019.
[9] "Virtiofs." https://virtio-fs.gitlab.io/index.html#overview.
[10] Peter N. Glaskowsky, "Nvidia's fermi: The first complete gpu computing architecture," tech. rep., NVIDIA Corporation, September 2009.
[11] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.
[12] Zi Yu Xue, "Performance analysis of file system implementation on nvidia gpus." Undergraduate Thesis, 2022.
[13] NVIDIA, *CUDA C++ Programming Guide*, May 2022.
[14] Michael Kerrisk, *fuse(4) — Linux manual page*. Linux, Aug 2021.