GPU Direct I/O with HDF5

John Ravi
North Carolina State University
jjravi@ncsu.edu

Suren Byna
Lawrence Berkeley National Laboratory
sbyna@lbl.gov

Quincey Koziol
Lawrence Berkeley National Laboratory
koziol@lbl.gov

***Abstract*—Exascale HPC systems are being designed with accelerators, such as GPUs, to accelerate parts of applications. In machine learning workloads as well as large-scale simulations that use GPUs as accelerators, the CPU (or host) memory is currently used as a buffer for data transfers between GPU (or device) memory and the file system. If the CPU does not need to operate on the data, then this is sub-optimal because it wastes host memory by reserving space for duplicated data. Furthermore, this "bounce buffer" approach wastes CPU cycles spent on transferring data. A new technique, NVIDIA GPUDirect Storage (GDS), can eliminate the need to use the host memory as a bounce buffer. Thereby, it becomes possible to transfer data directly between the device memory and the file system. This direct data path shortens latency by omitting the extra copy and enables higher-bandwidth. To take full advantage of GDS in existing applications, it is necessary to provide support with existing I/O libraries, such as HDF5 and MPI-IO, which are heavily used in applications.**

In this paper, we describe our effort of integrating GDS with HDF5, the top I/O library at NERSC and at DOE leadership computing facilities. We design and implement this integration using a HDF5 Virtual File Driver (VFD). The GDS VFD provides a file system abstraction to the application that allows HDF5 applications to perform I/O without the need to move data between CPUs and GPUs explicitly. We compare performance of the HDF5 GDS VFD with explicit data movement approaches and demonstrate superior performance with the GDS method.

***Index Terms*—GPU I/O, NVIDIA GPUDirect Storage (GDS), HDF5 GDS Virtual File Driver (VFD)**

I. Introduction

Complexity of next-generation systems is increasing rapidly with heterogeneous processing units and deep memory and storage hierarchies. Recent and upcoming high performance computing (HPC) systems are relying on accelerators to speed up parts of an large-scale simulations, visualizations and machine learning applications. For instance, the main computation for the National Energy Research Scientific Computing Center's (NERSC) next supercomputer, Perlmutter, will be done on GPUs [1]. Similarly, exascale systems such as Aurora and Frontier at the Department of Energy (DOE) leadership computing facilities are also designed with GPUs. In addition to heterogeneity in processing, memory and storage hierarchies are also deepening with multiples levels of memory and storage. This leads to various challenges in utilizing on-device memory and data movement. As systems become more heterogeneous and complex, it becomes increasingly challenging to tune I/O performance for an application on different platforms. I/O libraries, such as HDF5 [2] and MPI-IO [3], provide interfaces to implement optimizations for different file systems and memory hierarchies [4].

By providing an I/O abstraction to applications, high-level I/O libraries, such as HDF5 and netCDF [5], enable easier performance tuning for each system [6], [7]. In particular, system specific I/O optimizations can be reused across different applications with minimal effort. On top of this, I/O libraries can support complex data structures with an easy-to-use interface. HDF5 is one of most widely used middleware libraries in HPC applications [8]. HDF5 provides storage abstraction layers to enable per system I/O performance tuning. Applications that use HDF5 can benefit from significant I/O performance increase over the default I/O calls. In this paper, we describe moving data between NVIDIA GPUs and storage transparently using HDF5 and the latest GPUDirect Storage (GDS) technology [9]. I/O between GPUs and storage traditionally include using CPU's memory as an intermediate buffer and transfers between the GPU and CPU memories. These not only uses the extra buffer on CPUs, but also can be slow. NVIDIA recently introduced GPUDirect Storage technology that allows avoiding CPU memory as an intermediate buffer and uses a direct path between GPU memory and storage. Our work enables utilizing GDS directly from HDF5. In this effort, we have developed a prototype HDF5 Virtual File Driver (VFD) for enabling applications using HDF5 to utilize GDS without the need to using the GDS API. In summary, the contributions of this effort are:

- Design and implementation of HDF5 GDS VFD that integrates GDS functionality in HDF5 for transparently moving the data between GPUs and storage. We note that the GDS library is still in early user testing phase, which we use in this effort. Hence, this is one of the earliest efforts to use the cutting-edge GDS technology.
- Performance characterization of write and read functionalities for GPU compute applications
- Performance tuning of the HDF5 GDS VFD using multiple threads

The remainder of the paper is organized as follows. In Section II, we provide brief background to NVIDIA's GDS technology, HDF5 Virtual File Driver, and data movement involved in GPU I/O. We present our design and implementation of HDF5 GDS VFD in Section III and evaluate this prototype implementation in IV. We present future efforts in Section V and conclude our description of this effort in VI.

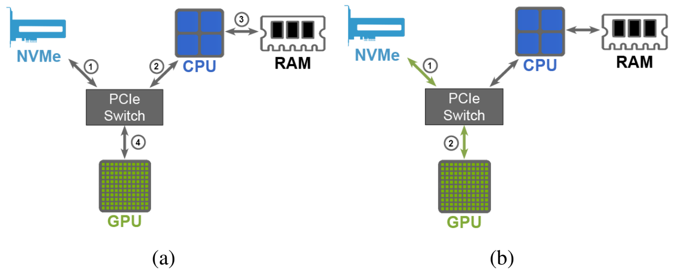

Fig. 1. Read operation data flow from local storage to GPU (a) without GPUDirect Storage and (b) with GPUDirect Storage

II. Background and Motivation

A. GPU Compute: GPUDirect Storage

Data parallel applications can achieve an immense speed up by utilizing GPUs. Its parallel architecture is optimized to perform many operations at once; however, GPU-accelerated applications still rely on CPUs to supply it with data and to launch compute tasks as kernels on GPUs. This can be inefficient due to multiple copies of the same data, in which case the CPU, or host, memory is referred to as a "bounce buffer". In Fig. 1a., we show the dataflow of current GPU accelerated applications. In a read operation (e.g., in machine learning applications), data is first fetched from the file system to CPU memory and then transferred to the GPU memory. Applications using NVIDIA's CUDA programming perform these explicit data movements between CPU (host) and GPU (device) using `cudaMemcpy`, where destination and source can be specified with host and device. In some applications, the CPU does perform some pre-processing of the data, but in other cases staging the data in the host memory is unnecessary.

NVIDIA's GPUDirect Storage enables a direct path between local or remote file systems and GPU memory. This new technology is similar to GPUDirect RDMA, which used the GPU's remote direct memory access feature to move data directly between a network interface card and GPU memory. In Fig. 1b., we show how the data can avoid being copied to the CPU memory and sent directly between the GPU memory and the file system.

GPUDirect Storage (GDS) currently supports Quadro and Tesla GPUs newer than the Volta architecture. Moreover, a GPUDirect Storage-enabled distributed file system or block system needs to be present. In this paper, we also include results experimental results on a local file system using `ext4` and a distributed file system using `lustre`. If those technologies are not available, `cuFile` (the kernel module driver of the GDS technology) provides a compatibility mode which uses an internal CPU bounce buffer to provide GPU I/O. In some scenarios, the compatibility mode can perform better than the `cudaMemcpy` method.

B. HDF5: Virtual File Driver

HDF5 is a widely used I/O library that provides portability, reliability, and performance. It used by many different types of applications ranging from high performance computing

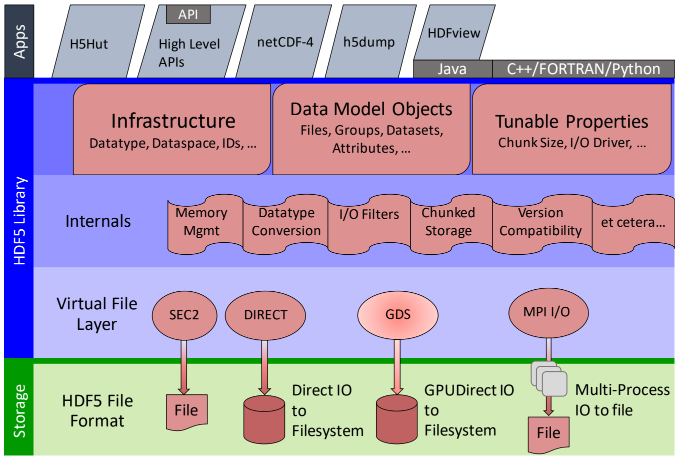

Fig. 2. HDF5 Virtual File Driver Abstraction

to mobile devices. As shown in Fig. 2, the HDF5 library provides a rich set APIs for describing various data models and organizing the data objects along with their descriptions as metadata. HDF5 contains various components to manage memory, converting data types, storing data as chunks, I/O filters such as compression and de-compression, etc. The virtual file driver (VFD) layer of HDF5 allows the implementation of mappings between the HDF5 address space and storage [10]. An application can specify which VFD to use through an HDF5 API call or by setting an environment variable, *HDF5_DRIVER*. By default, the *SEC2* VFD is used to provide POSIX file-system functions calls, such as read and write to perform I/O to a single file. A commonly used VFD in HPC applications is the *MPI-IO* VFD to allow parallel I/O using MPI and MPI-IO [11]. There also exists the DIRECT VFD which forces data to be written to the file system directly without being copied into system kernel buffer by making use of the `O_DIRECT` flag. To create a mapping for HDF5 applications to utilize GPUDirect Storage, we create a *GDS* VFD that utilizes `cuFile` to provide direct GPU I/O through POSIX-like system calls (read and write to a single file).

III. Design and Implementation

A. Differences from SEC2 VFD

We implement HDF5 GDS VFD to interface with GPUDirect Storage POSIX-like API function calls.

The self-describing format of HDF5 requires two types of I/O requests: metadata and data. The metadata describes the data stored in HDF5 dataset objects. The metadata have to be added separately by the user from CPU memory. Even though I/O calls to the GDS VFD will enable direct access to GPU memory, the HDF5 metadata still needs to reside in CPU memory. To enable a direct path between the storage system and GPU memory, it is necessary to disable OS buffering. This is done by opening the file descriptor with `O_DIRECT` enabled. Furthermore, in the HDF5 GDS VFD, we differentiate between a CPU memory pointer and a GPU memory pointer using the CUDA API. CPU memory operations are serviced using native POSIX reads/write calls using `pread` and `pwrite`. GPU memory operations are serviced using the GDS `cuFile` calls for reading and writing the data, i.e., `cuFileRead` and `cuFileWrite`, respectively.

B. GDS VFD Performance Tuning Options

The GDS VFD, based on HDF5's DIRECT VFD, supports sequential applications with a single process. The VFD can be used by MPI applications, where each GPU can write its own file. To improve concurrency of each process in performing I/O operations, we have implemented support for multi-threading and for selecting the size of I/O blocks. We provide these two options as configurable options that an application can tune, i.e., the number of CPU Threads participating in a single I/O call and the size of each I/O call. Each of these knobs can be dynamically changed during the runtime of the application. By default, HDF5 GDS VFD will not spawn any threads and will perform the full I/O request without chunking them into smaller sizes. When the multi-threading option is selected, threads are spawned to achieve concurrent I/O operations.

1) Multiple I/O Threads: On parallel file systems, such as Lustre, multiple storage servers and devices (Lustre Object Storage Servers and Object Storage Targets) are available for providing concurrent I/O operations. In order to saturate the I/O bandwidth of these parallel file systems, it is necessary to use multiple CPU I/O threads accessing these storage targets concurrently. In order to study the effects of using multiple I/O threads, we implemented support for serving each I/O call to HDF5 with multiple POSIX threads (pthreads) when using the GDS VFD. A user can provide the number of threads to be used by HDF5.

2) Block Size: Parallel file systems also provide an option for writing a block of data to be stored on a storage target. In Lustre, each block is called a stripe and the size of it as 'stripe size'. Depending on the number of available storage targets, users can select a stripe count to set the number of stripes on Lustre. Making I/O calls with a block size similar to the stripe size, applications can achieve higher throughput [12]. In the GDS VFD, we provide a configurable option to set the stripe size and stripe count for writing to the Lustre file system.

IV. Experimental Evaluation and Results

To evaluate the performance of our HDF5 GDS VFD, we designed a few micro-benchmarks, which use different application techniques aimed at observing the cost of I/O at different sizes. In these experiments, we compare the performance of using default HDF5 SEC2 VFD, DIRECT VFD, and GDS VFD. In the default SEC2 option, data I/O includes explicit data transfers between CPU and GPU using the `cudaMemcpy` calls. We ran these tests on an NVIDIA DGX-2 test-bed equipped with an NVMe-based local storage and a Lustre File System. The local storage was configured in RAID 0 with two NVMe drives with 1.8 GB/s theoretical max sequential write each. The configured Lustre on this system is using a progressive file layout. Note that these performance results are preliminary, as the version of GPUDirect Storage (GDS) library is still in beta testing stages, and some of the performance metrics shown in this paper might vary in the full release.

A. Performance on the Local File System

We characterized the performance of GPU-accelerated applications, which copy data between the file system and GPU memory using `cudaMemcpy` (labeled 'SEC2' and 'DIRECT' in the figures), `cudaMemcpyAsync` (labeled 'SEC2+PINNED' and 'DIRECT+PINNED'). As mentioned earlier, without the GDS VFD, applications have to transfer data explicitly from the GPU memory buffer to CPU memory and write to file system using `H5Dwrite`, which writes using a POSIX write call. With pageable memory allocations, the data transfers between the GPU and CPU can be slow due to Operating System overhead of verifying that memory pages have not been swapped. By pinning the CPU memory allocations, the extra overhead can be avoided at the cost of reducing the overall memory available for other applications on CPUs. Each I/O request to HDF5 is internally performed without spawning extra threads or splitting I/O calls to smaller block size; this is the default behavior for both the SEC2 VFD, DIRECT VFD, and GPUDirect Storage VFD.

In Fig. 3, we demonstrate the performance benefit of using GPUDirect Storage VFD. For write sizes greater than 256 MB using the GDS VFD achieves higher observed write rate (the ratio of the size of data written to the wall-clock time) when compared to the total cost of writing data from the GPU buffer to the local file system. In this experiment, we perform a sequential write operation varying from 64 MB to 2 GB. The observed write rate using pinned memory with DIRECT VFD is higher than the other cases for lower write sizes. Since this is a sequential write benchmark, the extra buffering the OS does when not enabling `O_DIRECT` adds some extra latency for the SEC2 VFD case. Also for lower I/O sizes, the extra cost of querying CUDA for the memory buffer location inside the GDS VFD reduces the observed write rate. Nonetheless, for the larger write sizes by utilizing GPUDirect Storage we were able to achieve almost the theoretical maximum write speeds of our local file system.

In Fig. 4, we compare the performance of using the GDS VFD when performing read operations. With a smaller read size, the GDS VFD does not outperform using the SEC2 VFD with Read Ahead enabled; our GDS VFD outperforms the SEC2 VFD for read sizes greater than 128 MB. If `O_DIRECT` is not enabled, the OS will try to 'Read Ahead' by prefetching and caching data; this offers a noticeable performance boost for sequential reads. The read rates of SEC2 with Read Ahead disabled, using the `fadvise` system call, performs similarly to DIRECT VFD. Although, it is possible to support Read Ahead with GDS, it is not yet supported by the `cuFile` driver. Thus, using SEC2 VFD with Read Ahead left enabled would yield higher write rates than our HDF5 GDS VFD for read sizes less than 128 MB.

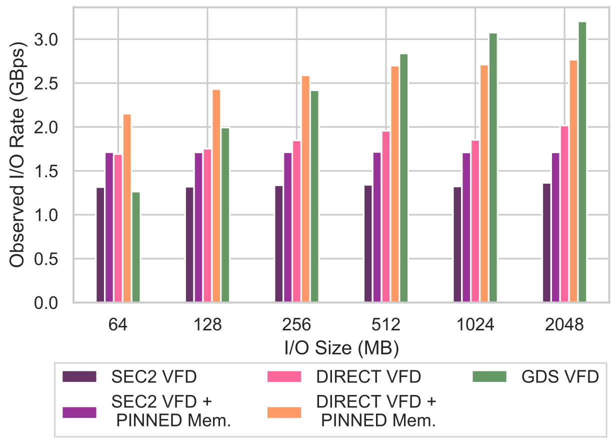

Fig. 3. Write performance from a GPU to a node-local NVMe storage with varying write sizes

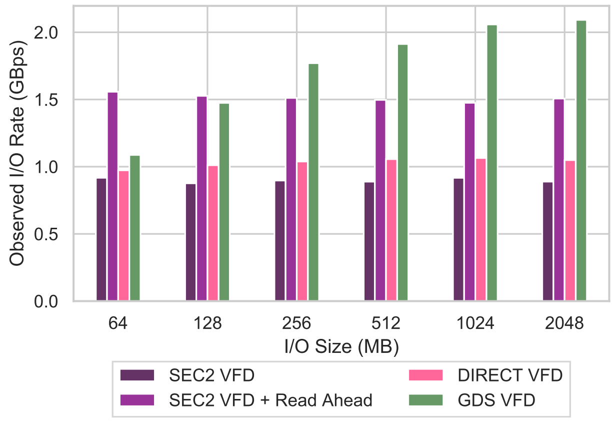

Fig. 4. Read performance from a node-local NVMe storage to a GPU with varying read sizes

B. Performance on Lustre with multiple threads

When using a network file system, such as Lustre [13] or WekaFS [14], better performance can be achieved by using multiple CPU I/O threads and splitting the I/O requests in smaller block sizes. We offer these parameters to be tuned with the GDS VFD. The following results compare the observed read and write rates with the GDS VFD and the default HDF5 SEC2 VFD which does not spawn any threads or split the I/O requests into smaller blocks. This experiment uses a single GPU and writes to a Lustre file system.

In Fig. 5, we show the write rate to Lustre with varying number of CPU threads and varying block size. The write rate of the SEC2 VFD is overlaid on each plot for easy comparison; however, the SEC2 VFD does not split the I/O requests as noted earlier. We show that with more I/O threads, the overall throughput increases substantially for larger write sizes. We only tested with block sizes from 1 MB to 8 MB; we observe that varying the block size does not affect the write rate. In Fig. 6, we compare the read rates between the SEC2 VFD (with Read Ahead enabled) and the GDS VFD. In both cases, we noticed far smaller throughput than the write rates.

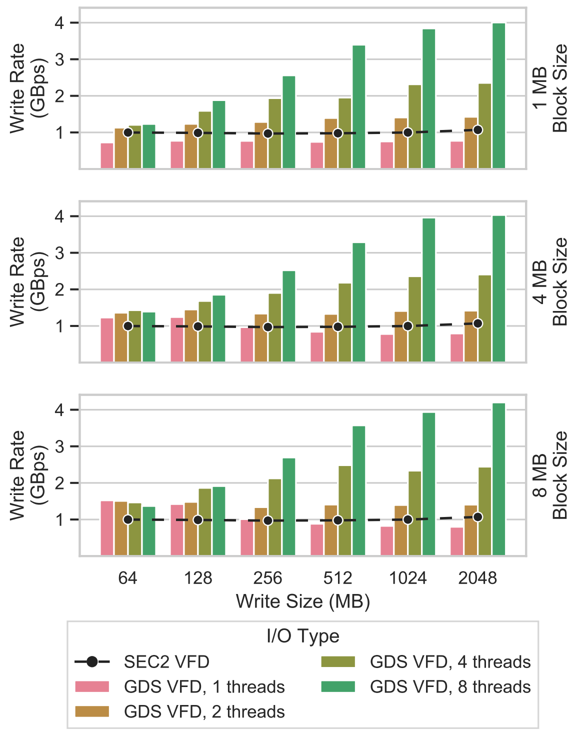

Fig. 5. Write rates with multi-threaded GDS VFD with varying I/O sizes

C. Performance with Multiple GPUs

The benefit of the GDS VFD with a distributed file system is much more apparent when using multiple GPUs. We demonstrate that by increasing the number of GPUs used, we can achieve a much higher throughput to the file system. One can use multiple GPUs with our HDF5 GDS VFD by utilizing MPI and letting each MPI rank write to a separate file. In these two experiments, we make each MPI rank operate on 2 GB data size with 4 MB block size. The GDS VFD is configured to spawn 4 CPU threads to service each I/O request.

In Fig. 7, we show more than 2$\times$ speed up when using the GDS VFD versus the SEC2 VFD. Moreover, the aggregate write rate scales linearly as we add more GPUs to participate in the I/O requests. In Fig. 8, we also show an increase in the aggregate read rate as we add more GPUs with our GDS VFD; however, the SEC2 VFD (with Read Ahead enabled) performs better in all cases. We suspect that Lustre caching to be the primary reason for this discrepancy, which needs further profiling to confirm.

V. Future Work

Per System Tuning Although we offer tuning parameters for the GDS VFD, such as the number of I/O threads and I/O block size, we plan on including the support to automatically query the system for optimal configurations. For example, the I/O block size can be set based on the stripe size for a

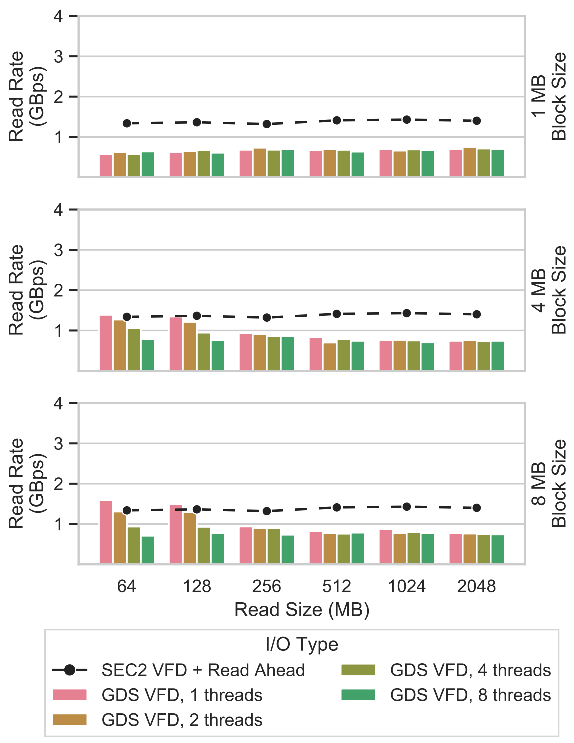

Fig. 6. Read rates with multi-threaded GDS VFD with varying I/O sizes

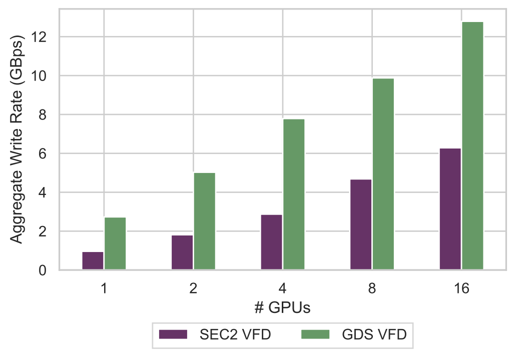

Fig. 7. Write performance of GDS using multiple GPUs

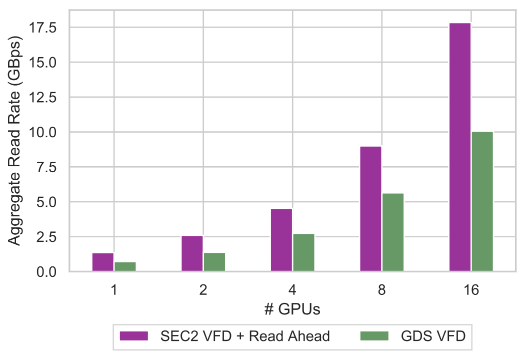

Fig. 8. Read performance of GDS using multiple GPUs

distributed file system. Similarly, the number of I/O threads can avoid oversubscribing if there are not enough CPU cores.

Asynchronous I/O: One way to achieve higher throughput after reaching the bandwidth limitations of a system is to overlap data transfers with computation. It is possible to achieve asynchronous I/O through the current HDF5 asynchronous Virtual Object Layer (VOL) connector [15]. By utilizing CUDA Streams, it becomes possible to overlap I/O calls made with GPUDirect Storage with computation. We could extend the functionality of HDF5 ASYNC VOL to support GDS asynchronous operations.

Multi-threaded POSIX I/O: To make a better comparison of using multiple threads to service a GDS I/O request, HDF5's default POSIX I/O could also use multiple threads. Multi-threaded I/O offer the possibility of achieving higher throughput on distributed file system. We are currently designing multi-threaded SEC2 VFD, which we will compare in future.

Parallel I/O: Design work is in progress to extend HDF5 to perform parallel I/O directly from GPUs. We are profiling performance with multiple prototypes. These include using background threads for transferring the data between CPU and GPU and performing I/O asynchronously using MPI-IO enabled with NVIDIA's GDS, etc.

VI. Conclusions

In this paper, we described and demonstrated how applications can benefit from utilizing GPUDirect Storage VFD with HDF5. We compared the performance of GDS VFD with the default HDF5 SEC2 VFD for GPU accelerated applications. We demonstrated about 2× speed up for write rate and read rates with GPU I/O to and from local storage. For distributed file systems, we demonstrated around 2× speed up for the write rate. The HDF5 GDS VFD development is based on a cutting-edge technology and various follow up R&D efforts are in progress building on this initial successful implementation and performance improvements.

Acknowledgment

This research was supported by the Exascale Computing Project under Grant No. 17-SC- 20-SC, a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration. This work is also supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract Nos. DE-AC02- 05CH11231. We thank NVIDIA for providing a test system to experiment with GPUDirect Storage.

References

[1] *Perlmutter the supercomputer at NERSC*. [Online]. Available: https://www.nersc.gov/systems/perlmutter/.
[2] The HDF Group, *HDF5*, https://www.hdfgroup.org/solutions/hdf5/.
[3] ROMIO team at ANL, *ROMIO: A High-Performance, Portable MPI-IO Implementation*, https://www.mcs.anl.gov/projects/romio/.

[4] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, “An Overview of the HDF5 Technology Suite and Its Applications,” in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD ’11, Uppsala, Sweden: Association for Computing Machinery, 2011, pp. 36–47, ISBN: 9781450306140. DOI: 10.1145/1966895.1966900. [Online]. Available: https://doi.org/10.1145/1966895.1966900.

[5] R. Rew and G. Davis, “NetCDF: an interface for scientific data access,” *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, 1990.

[6] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, “Parallel netCDF: A High-Performance Scientific I/O Interface,” in *SC ’03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ID: 1, 2003, p. 39. DOI: 10.1109/SC.2003.10053.

[7] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, “Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks,” *Concurr.Comput.: Pract.Exper.*, vol. 26, no. 7, pp. 1453–1473, May 2014. DOI: 10.1002/cpe.3125. [Online]. Available: https://doi.org/10.1002/cpe.3125.

[8] Z. Zhao, *Automatic Library Tracking Database at NERSC*. [Online]. Available: https://sdm.lbl.gov/exahdf5/papers/201810-HDF5-Usage.pdf.

[9] *GPUDirect Storage: A Direct Path Between Storage and GPU Memory*, https://developer.nvidia.com/blog/gpudirect-storage/.

[10] *The HDF5 library and file format*. [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/.

[11] S. R. Walli, “The POSIX Family of Standards,” *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[12] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, “Taming Parallel I/O Complexity with Auto-Tuning,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13, Denver, Colorado: Association for Computing Machinery, 2013, ISBN: 9781450323789. DOI: 10.1145/2503210.2503278. [Online]. Available: https://doi.org/10.1145/2503210.2503278.

[13] P. Braam, “The Lustre Storage Architecture,” *CoRR*, vol. abs/1903.01955, 2019, 1903.01955. [Online]. Available: http://arxiv.org/abs/1903.01955.

[14] *WekaFS Architecture White Paper*, https://www.weka.io/wp-content/uploads/files/2017/12/Architectural_WhitePaper-W02R6WP201812-1.pdf.

[15] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, “Enabling Transparent Asynchronous I/O using Background Threads,” in *PDSW 2019, in conjunction with SC19*, 2019.