HPDR: High-Performance Portable Scientific Data Reduction Framework

Jieyang Chen*, Qian Gong◇, Yanliang Li*, Xin Liang†, Lipeng Wan‡, Qing Liu△, Norbert Podhorszki◇, Scott Klasky◇
*University of Oregon, OR, USA
◇Oak Ridge National Laboratory, TN, USA
†University of Kentucky, KY, USA
‡Georgia State University, GA, USA
△New Jersey Institute of Technology, NJ, USA
jieyang@uoregon.edu, gongq@ornl.gov, leonli@uoregon.edu, xliang@uky.edu
lwan@gsu.edu, qing.liu@njit.edu, pnorbert@ornl.gov, klasky@ornl.gov

2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3315-3237-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/IPDPS64566.2025.00101

***Abstract*—The rapid growth in scientific data generation is outpacing advancements in computing systems necessary for efficient storage, transfer, and analysis, particularly in the context of exascale computing. With the deployment of first-generation exascale computing systems and next-generation experimental facilities, this gap is widening and necessitates effective data reduction techniques to manage enormous data volumes. Over the past decade, various data reduction methods, including lossless compression, error-controlled lossy compression, and data refactoring, have been developed to accelerate I/O in scientific workflows. Despite significant reductions in data volume, these methods introduce considerable computational overhead, which can become the new bottleneck in data processing. To mitigate this, GPU-accelerated data reduction algorithms have been introduced. However, challenges remain in their integration into exascale workflows, including limited portability across different GPU architectures, substantial memory transfer overhead, and reduced scalability on dense multi-GPU systems. To address these challenges, we propose HPDR, a high-performance and portable data reduction framework. HPDR is designed to enable the execution of state-of-the-art reduction algorithms across diverse processor architectures while reducing memory transfer overhead to 2.3% of the original, resulting in up to 3.5× faster throughput compared to existing solutions. It also achieves up to 96% of the theoretical speedup in multi-GPU settings. In addition, evaluations on accelerating I/O operations at scale up to 1,024 nodes of the Frontier supercomputer demonstrate that HPDR can achieve up to 103 TB/s reduction throughput, providing up to 4× acceleration in parallel I/O performance compared to existing data reduction routines. This work highlights the potential of HPDR to significantly enhance data reduction efficiency in exascale computing environments.**

***Index Terms*—Data reduction; GPU; Portability; I/O**

I. Introduction

The generation of scientific data is outpacing the rate of improvement in the computing systems needed to save, transfer, and analyze data. With the deployment of exascale computing workflows that consist of the first generation of exascale computing systems [1], [2] and the next generation of experimental and observation facilities [3]–[5], there will be an even wider compute and I/O gap in exascale computing of the foreseeable future. For example, XGC simulation code [6], [7], which simulates a fusion reactor with magnetically confined plasma, can generate data at a rate of exabytes per day. In addition, Square Kilometer Array [8], the world's largest radio telescope, is anticipated to have a raw data rate of approximately 2 PB/s, or 600 PB/yr after hierarchical beam-forming reduction, all to be stored on the buffer file system [9]. These scientific workflows highlight the growing need to integrate data reduction in workflows to accelerate data storage, sharing, and analysis.

During the past decade, many data reduction techniques, such as lossless compression [10]–[14], error-controlled lossy compression [15]–[22], and data refactoring [23]–[25], have been developed to accelerate I/O in scientific workflows. While they provide significant data volume reduction, modern data reduction routines also bring a considerable amount of computational overhead, which tends to increase as more advanced reduction algorithms are built. Since such reduction overhead may become the new bottleneck of I/O, several Graphics Processing Unit (GPU) accelerated data reduction algorithms have recently been developed. For instance, nvCOMP [26] offers a collection of GPU lossless compression developed by NVIDIA. Lossy compression such as MGARD [17], SZ [16], and ZFP [27] also developed GPU accelerated parallel design: MGARD-GPU [17], cuSZ [28]–[30], and ZFP-CUDA [27].

Although GPU-accelerated data reduction offers performance acceleration compared with their serial implementation, scientists still face several significant challenges in using them to accelerate I/O in exascale computing workflows: (1) Current

1530-2075/25/$31.00 ©2025 IEEE
DOI 10.1109/IPDPS64566.2025.00101

data reduction has limited portability across architectures. Most existing reduction algorithms are accelerated for Nvidia GPUs with very limited support on other GPU architectures (such as AMD and Intel GPUs) that are used in current exascale computing systems. Without portability, data cannot be easily shared across computing facilities. (2) A large gap exists between the performance obtained on GPU and the perceived performance by user applications. This is because most reduction algorithms are memory-bound, but the overhead brought by memory operations such as management and transfer are often overlooked. Those operations can create a large overhead or dominate the reduction pipeline. (3) Modern large-scale computing systems often use dense GPU architecture that packs multiple GPUs into one computing node. While such types of architecture bring performance advantages, they also raise challenges for scalability for memory-bound operations, which is typically the case in data reduction. This is because all GPUs share the same runtime, where memory operations issued concurrently can easily cause contention that degrades the scalability of data reduction routines. To overcome those challenges and help modern scientific workflows benefit from data reduction, we design HPDR. HPDR is a high-performance and portable framework tailored for efficiently running reduction algorithms across different processor architectures. In addition, our framework is highly extensible to execute new reduction algorithms and support new processor architectures via programming models (e.g., CUDA [31], HIP [32], etc.) and general-propose portability libraries (i.e., SYCL [33], Kokkos [34], etc.).

Our paper makes the following contributions.

- We propose a portable data reduction framework, HPDR, featuring a series of runtime abstractions to optimize the performance and scalability of common data reduction algorithms across multi-GPUs and multi-core architecture. We demonstrate how three state-of-the-art data reduction tools—MGARD, ZFP, and Huffman encoding—can be implemented using HPDR, ensuring compatibility across five distinct CPU and GPU architectures.
- We identify CPU-GPU memory transfer as a significant performance bottleneck, which is often overlooked in existing state-of-the-art data reduction works. Furthermore, for applications that continuously generate data, reduction and data movement must be optimized in tandem to maximize overall performance. To address this, we propose a highly optimized pipeline. By adaptively overlapping reduction with data transfer and coordinating the execution of multiple asynchronous tasks, HPDR reduces the data transfer overhead to 2.3% of the original and accelerates end-to-end reduction throughput by up to $3.5\times$ for the three state-of-the-art reduction pipelines. When scaling on multiple GPUs, reduction optimized with HPDR achieves up to 96% of the theoretical speedup, compared to a mere 74% with non-optimized designs.
- To demonstrate the I/O acceleration using HPDR for reading and writing scientific data at scale, we integrate HPDR with ADIOS2 I/O library [35] on two leadership-class supercomputers, Summit and Frontier, at Oak Ridge Leadership Computing Facility (OLCF) with scale up to 1,024 nodes. Our evaluation shows that HPDR achieves up to 103 TB/s reduction throughput and provides up to $4\times$ parallel I/O acceleration compared with I/O with existing data reduction routines.

The rest of the paper is organized as follows. Section II introduce backgrounds and challenges of achieving high-performance data reduction on modern computing systems. Then, we introduce our novel high-performance portable data reduction framework in Section III and demonstrate the applicability of our reduction framework for three state-of-the-art reduction pipelines in Section IV. Moreover, to optimize reduction process on accelerators we further introduce an optimized reduction pipeline design in Section V. Finally, we provide a thorough experimental evaluation of our proposed works in Section VI.

II. Background and Problem Statement

A. Scientific data reduction

In the past decade, various data reduction techniques have been developed to reduce the volume of scientific data since they has long been considered as a powerful tool for alleviating I/O and storage costs. Among those, error-bounded lossy compression techniques have gained popularity. Methods like MGARD [17], SZ [16], ZFP [27], and ISABELA [36], offer remarkable compressibility while guaranteeing that the difference between original and the reconstructed data remains within user prescribed error bounds. Most error-bounded lossy compressors follows a three-step processes, involving decorrelation, quantization, and lossless encoding, supported by mathematical theories to ensure error-bound satisfaction. Specifically, the decorrelation and quantization first transform original data into a set of discrete coefficients with lower entropy, enabling efficient encoding. Then, lossless encoding compress the coefficients to achieve data reduction.

Although modern data reduction techniques can greatly reduce data volume, the computational costs of reduction can sometimes be non-trivial, leading to situations where the time spent on reduction and reconstruction outweighs the benefits gained from reduced data read and write time [23]. To minimize the data reduction overhead, many data reduction have been re-designed for parallel execution on CPU or GPU accelerators [24], [29].

B. Challenges of achieving high-performance data reduction

Despite many data reduction optimizations that have been done to improve their performance on various types of processors, there are two major overlooked challenges that hinder application users from obtaining low-cost data reduction.

Diverse processor architectures: Modern computing systems are equipped with diverse types of processors, such as CPUs and GPUs, designed with different micro-architectures.

Because of their fundamental difference in terms of architecture, a data reduction pipeline typically leverages different algorithm designs on different processor types. Although different algorithm designs may share the same foundational theory and method, their difference can cause data portability challenges: data reduced by one type of processor cannot be reconstructed by another type of processor with a guarantee. Because of such portability challenges, application users are forced to use the most compatible processor to make sure application data are (1) shareable across different components workflows that may reside on different systems; and (2) retrievable on future systems that use new architectures. However, the most compatible processors, such as single-core CPUs, cannot guarantee the best reduction performance.

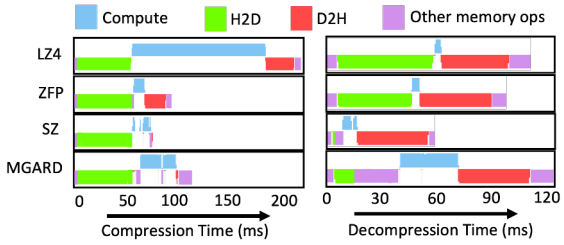

Fig. 1: Time breakdown of reducing a 500 MB NYX data [37] using four different reduction pipelines on a V100 GPU. $1e^{-2}$ error bound is used for lossy compression. Both application and I/O buffers are on the host.

Large gap between GPU kernel and end-to-end performance: Although many works [24], [27], [30], [38]–[40] have been done to optimize the throughput of data reduction on GPU, the improvement can hardly be transformed into the advancement of end-to-end performance perceived by application users. This is because the latency of data transfers is often overlooked. Those include memory copy between application buffer, reduction buffer, and I/O buffer. Such data transfers cannot be easily eliminated as host memory is typically used by applications to save output data before reduction and/or I/O libraries as internal serialization buffers before I/O operations. Unlike scientific applications where computation dominates costs, data reduction algorithms are typically memory bound [24], so the cost of memory operations brings considerable performance overhead. Figure 1 shows the profiling results of four state-of-the-art GPU compression pipelines. We can see a large amount of time is spent on memory operations. Specifically, 34 - 89% of the total time is spent on memory operations (e.g., H2D and D2H) during compression or decompression pipeline. This ratio will increase if reduction kernels are further accelerated.

III. HPDR Portable Data Reduction Framework

In this work, we propose a novel high-performance portable data reduction framework aiming to streamline data sharing across systems with different processor architectures. Figure 2 shows the overall framework structure of HPDR, including

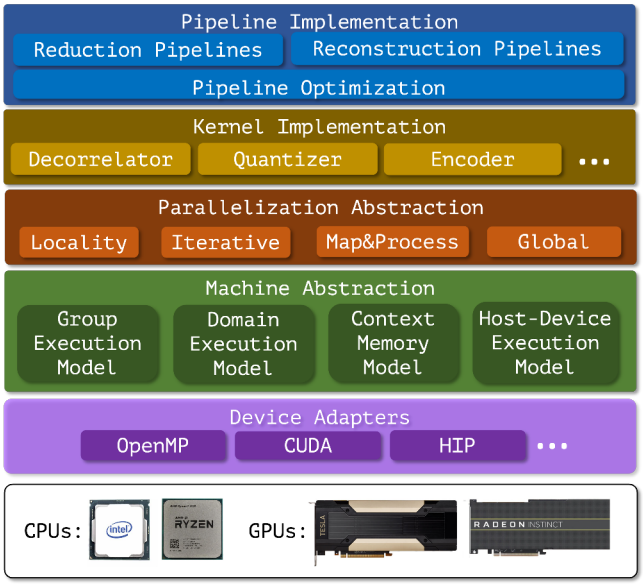

Fig. 2: High-perf. portable data reduction framework (HPDR)

five software architectural layers. All layers work together to enable portability and optimizations that allow data reduction algorithms to take advantage of the best processor on a system, while producing portable data that can be reconstructed on another system. In the rest of this section, we discuss three bottom layers: parallel abstraction, machine abstraction, and device adapter. We will discuss how to implement data reduction algorithms on top of the three layers in Section IV and pipeline optimization in Section V.

A. Parallelization Abstraction

The parallelization abstraction layer enables data reduction algorithms to express their fine-grain parallelisms effectively. Specifically, we introduce four example parallelization abstractions currently built in HPDR.

1) Locality Abstraction: Many reduction algorithms exploit the correlations in data across multiple dimensions. For instance, MGARD uses a multi-dimensional multi-level decomposition. ZFP applies a block-wise compression on every $4^{dim.}$ cubic of data. Figure 3(a) shows the locality abstraction designed for exploiting decorrelating the data using an algorithm-defined function f. In this abstraction, the input data is parallelized by decomposing into blocks with customizable sizes and halo regions. Then, a group of threads cooperatively executes f on each block. In addition, blocks can be loaded into a faster memory tier throughout the entire processing pipeline.

2) Iterative Abstraction: Some algorithms also need to process data iteratively along one dimension by repeatedly executing a function f. For example, solving tri-diagonal systems in MGARD. Figure 3(b) shows that iterative abstraction parallelizes the processing of each vector of data among different threads with every B vector organized into a group.

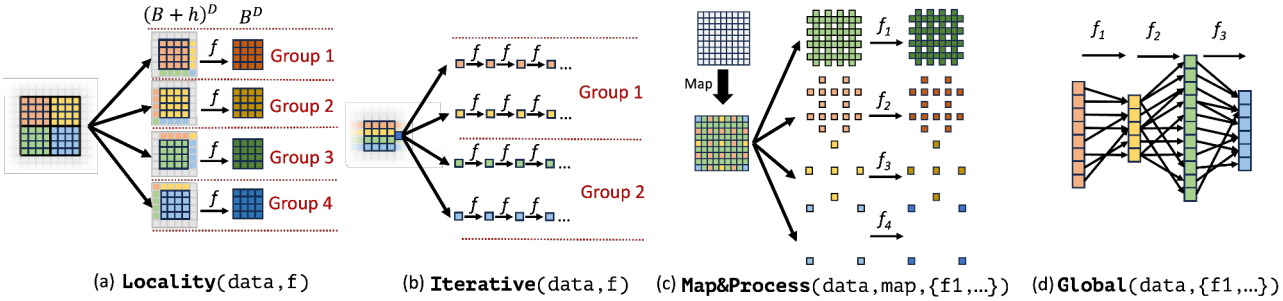

Fig. 3: Parallel Abstractions in HPDR

Similar to locality abstraction, each group of vectors is cached in a faster memory tier for better performance.

3) Map And Process Abstraction: For decomposed-based data reduction such as MGARD, data needs to be decomposed into a hierarchy, and each level of the hierarchy needs to be processed individually. Figure 3(c) shows the map and process abstraction. The input data is first mapped to the subsets, and each is processed using a different function.

4) Global Pipeline Abstraction: Many encoding algorithms, such as Huffman encoding, exploit the correlation and redundancy in the whole data domain. Also, parallel serialization operation commonly used in many compression algorithms also needs to collect global information to compact bit streams. This requires global-wide processing that allows all threads to collaborate globally. Figure 3(d) shows the global parallel abstraction. All data is processed at the same time with global synchronizations to enable communication and data exchange among all threads.

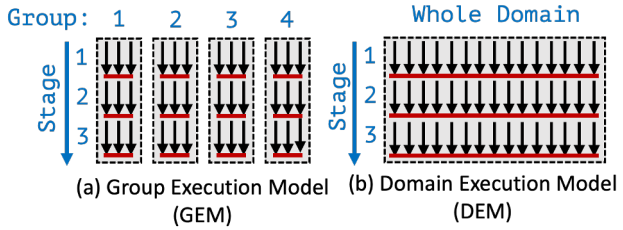

Fig. 4: Group and Domain Execution Models

TABLE I: Mapping Parallel Abstractions to Execution Models

Parallel Abstraction	GEM	DEM
Locality	Block→Group	-
Iterative	B*Vectors→Group	-
Map&Process	-	All Subsets→Whole Domain
Global	-	Domain→Whole Domain

B. Machine Abstraction

To execute the four parallel abstractions, we design three execution models and one memory management model to abstract the underlying hardware: the Group Execution Model (GEM), Domain Execution Model (DEM), Host-Device Execution Model (HDEM), Context Memory Model (CMM). We will introduce GEM, DEM, and CMM in this section and disucss HDEM in Section V. GEM partitions threads into groups and executes independently. DEM, put all threads in one domain and execute in a synchronized fashion. Both DEM and GEM support multi-stage execution such that multiple operations sharing the same execution model can be fused into one model for more efficient execution. Table I shows how the four parallel abstractions are mapped to execution models. For locality abstraction, blocks are 1:1 mapped to groups in GEM to exploit coarse-grain parallelism across blocks and fine-grain parallelism within a block. For iterative abstraction, vectors are B:1 mapped to groups in GEM so that we can exploit memory locality across vectors. Map&process and global pipeline are mapped to DEM such that the whole data domain can be processed at the same time.

In addition to GEM and DEM, we build a memory management model, CMM, to manage the context memory of the reduction process. One major performance bottleneck in data reduction that is often overlooked is the overhead of memory allocation management. Data reduction algorithms tend to have low arithmetic intensity, so the cost of memory-related operations such as allocations for the purpose of building reduction context can easily dominate the cost of a reduction pipeline (e.g., other memory operations as shown in Figure 1). High memory allocation overhead can negatively impact end-to-end I/O performance greatly when a reduction pipeline is repeatedly invoked by scientific applications (e.g., every write iteration). In addition, when a data reduction pipeline is deployed on a computing system with multiple GPUs that share the same runtime system (e.g., a multi-GPU compute node), all memory allocation management operations typically cannot be efficiently parallelized due to shared internal memory management, which can greatly impact the scalability of data reduction. To overcome such a challenge, we propose a context memory management optimization that uses a hash map to cache reduction contexts so they can be reused across repetitive reduction processes that share similar data characteristics. With this optimization, all memory allocations associated with a context will be persistent across calls to minimize the need for allocations.

C. Device Adapters

To enable portability support for data reduction, device adapters are designed to execute the two execution models on

TABLE II: Mapping Execution Models to Devices

Models	Res. Mapping		OMP	CUDA	HIP
GEM	Compute	Thread	Serial	Core	Core
		Group	Core	SM	CU
	Staging	Data	Cache	ShMem.	ShMem.
		Order	Serial	Blk. Sync.	Blk. Sync.
DEM	Compute	Domain	All Cores	All Cores	All SUs
	Staging	Data	DRAM	DRAM	DRAM
		Order	Serial	Grid Sync.	Grid Sync.

different architectures in a compatible way. Device adapters have two major design advantages: (1) Separately designed device adapters enable us to fully exploit the functionalities that is only available for certain hardware architecture or programming models. For example, whole domain synchronization can be more efficiently enabled using Cooperative Groups exclusively available in CUDA and HIP. (2) HPDR can be easily extended to support newer architectures or leveraging general-propose portability libraries such as Kokkos [41] and SYCL [42] by implementing new device adapters. Currently, we implement three device adapters: OpenMP, CUDA, and HIP for multi-core CPUs, Nvidia GPUs, and AMD GPUs. Table II shows how the two execution models are mapped to device adapters. For OpenMP, GEM is executed by parallelizing groups among CPU cores, and the workload of each group is executed sequentially. Such a parallelization strategy can improve memory efficiency by letting each CPU core exploit data locality within a group. When executing a multi-stage GEM model, the execution order is maintained through sequential execution, and working data can be staged in the cache for sharing between stages. For DEM, OpenMP parallelizes all threads in the domain along all CPU cores to maximize parallelism. When there is more than one stage, the execution order is maintained through sequential execution, and working data among stages are shared via DRAM as it might be too large to fit into the cache. For CUDA and HIP, GEM is executed by parallelizing groups among Streaming Multiprocessors (SMs) for CUDA or Compute Units (CUs) for HIP, and the workload of each group is parallelized on GPU cores. When executing a multi-stage GEM model, the execution order is maintained through threadblock-level synchronization, and working data can be staged in shared memory for sharing between stages. For DEM, all threads in the domain are parallelized along all GPU cores to maximize parallelism. When there is more than one stage, the execution order is maintained through Cooperative Groups, and working data among stages are shared via DRAM.

IV. Data Reduction Pipeline Case Study

To demonstrate the applicability of HPDR, we show how to build three data reduction pipelines: MGARD compression [17], ZFP fixed-rate data reduction [27], and Huffman lossless compression [40].

A. MGARD lossy compression

MGARD is designed to compress both uniform and non-uniform grids while preserving the error of Quantities of Interest. Figure 5 shows the MGARD compression pipeline. MGRAD decorrelates data through a specially designed multilevel decomposition process. Specifically, the data is treated as a piecewise linear function u that takes the same values as the original data for each node, and it is recursively decomposed level by level. The function approximation $Q_l u$ on level l is projected onto the next level $l-1$ by (1) calculating multi-level coefficients $mc_l = I - \Pi_{l-1})Q_l u$, which is the difference between the values of the fine grid nodes at level l and their corresponding piecewise linear approximations; and (2) calculating and applying the global correction $Q_{l-1}u = Q_l u + \Omega_l mc_l$. Then, the multi-level coefficients can be compressed and used to reconstruct the approximation of the original data.

Algorithm 1: MGARD Lossy Compression

```
1  Function MGARD():
       In  : Data u
       In  : Error bound e
2      hierarchy ← u
3      mc = {}
4      l ← 0
5      while l < hierarchy.total_levels do
6          mc_l ← Locality(u, lerp())
7          correction_l ← mc_l
8          Locality(correction_l, mass_trans())
9          Iterative(correction_l, tridiag())
10         Locality(u, add(correction_l))
11         mc = mc ∪ mc_l
12         l ← l + 1
13     end
14     mc_quantized ← MapAndProcess(mc,
        hierarchy, quantization(e))
15     u_compressed ← Huffman(mc_quantized)
16 return u_compressed
```

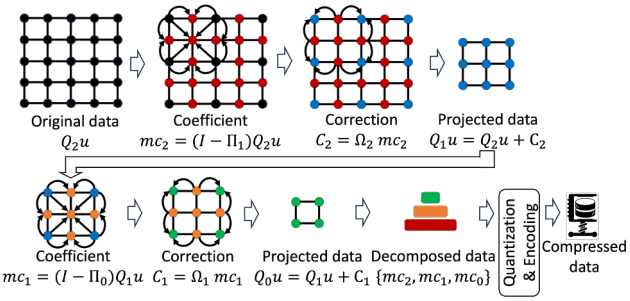

Fig. 5: MGARD compression pipeline

Algorithm 1 shows the implementation of MGARD using HPDR. At each level, mc_l is first computed using multi-linear interpolation (`lerp`) that can be done using our locality abstraction (line 6). Line 7-9 calculate the global correction that applies L^2 projection of mc_l to u. This process involves a transfer-mass matrix multiplication that can be done using the locality abstraction and a tri-diagonal linear system

solver, which requires iterative abstraction since calculations in solving each system are sequential. Line 10 applies the correction followed by multilevel coefficients collected in line 11. After all levels have been decomposed, linear quantization is applied to the coefficients with different quantization bin sizes applied to different levels to improve the compression ratio and capability to preserve the quantities of interest. We use map and process abstraction to first map multilevel coefficients to different levels and then apply different quantization functions. Finally, we use entropy encoding, such as Huffman compression, to compress quantized data.

Algorithm 2: Huffman Data Compression

```
1 Function Huffman():
      In     : Data u
2     freq ← Global(u, histogram())
3     SortByKey(freq)
4     Global(freq, filter_non_zeros())
5     codebook ← Global(freq, tree_build())
6     u_enc ← Locality(u, encode(codebook))
7     u_compressed ← Global(u_enc, serialize())
8 return u_compressed
```

B. Huffman lossless compression

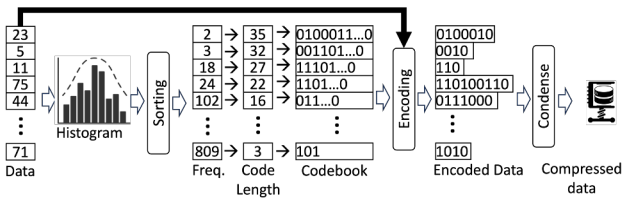

Fig. 6: Huffman compression pipeline

Huffman is a widely used lossless compression in many scientific data compressors. It works as an entropy encoder that can effectively compress decorrelated data. Figure 6 shows the Huffman compression pipeline. It works by first collecting the frequency distribution of input elements using a histogram, then using the frequencies to generate variable-length Huffman codes for different keys. The encoding process replaces the original values in data with Huffman codes such that more frequent keys are represented with fewer bits. Finally, encoded variable-length keys are condensed into a compact format that forms the compressed data.

Algorithm 2 shows the implementation of Huffman using HPDR. In line 2, we first collect the frequency of each key using the histogram operation proposed in [43]. As all threads need to collaboratively update the frequency counters, we use the global abstraction. After sorting the frequencies in line 2, we filter all the keys that have non-zero frequency in line 3. Next, at line 5, we implement the parallel two-phase Huffman treeless codebook generation algorithm [44] as it provides much higher parallelism. Limited by the page space, we refer readers to [40] for a detailed algorithm for codebook generation. At line 6, we encode the input key using the codebook. As each key can be encoded independently, we use the locality pipeline abstraction to exploit maximum parallelism. Finally, we serialize all encoded keys into a compact form. Since all threads need to write to the same buffer, global coordination is required to avoid conflicts.

Algorithm 3: ZFP Fix-Rate Compression

```
1 Function ZFP():
      In     : Data u
      In     : Bit rate r
2     u_aligned ← Locality(u, exp_align())
3     u' ← Locality(u,
       orthogonal_transform(u))
4     u_compressed ← Locality(u',
       bitplane_encode(r))
5 return u_compressed
```

C. ZFP fix-rate compression

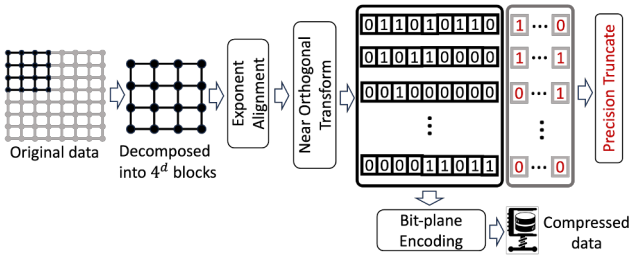

Fig. 7: ZFP compression pipeline

ZFP is designed to enable high throughput compression as it brings relatively lower computational costs compared with other compression pipelines. Figure 7 shows the ZFP compression pipeline. The input data is first decomposed into blocks of 4^d (d: number of dimensions). The exponents of elements of each block is first aligned according to the max components and converted to fixed-point values. Then those values are transformed using a customized near-orthogonal transformation to be more compressible. Then a number of less significant bits of transformed coefficients are truncated and the rest of the bits are serialized into bitplanes that forms compression data stream.

Algorithm 3 shows the implementation of ZFP fix-rate compression using HPDR. ZFP has three compression modes: fix-rate, fix-accuracy, fix-precision. To demonstrate the applicability of HPDR, we choose to only implement the fix-rate mode. The other two modes can be implemented in similarly. Since both exponent alignment and near-orthogonal transformation operations are applied blockwise, we can implement them using the locality abstraction (line 2-3). The transformed data needs to be truncated and serialized (line 4) to form the compression data. As all blocks output the same size bit streams, this can be done without global coordination, we also apply locality abstraction.

V. Pipeline Optimization

To minimize the data movement overhead, especially serious in GPU-based data reduction pipelines, we build a carefully tailored pipeline optimization in HPDR for data reduction. Although we mainly target GPU-based pipelines, our optimization can be portable across different processors.

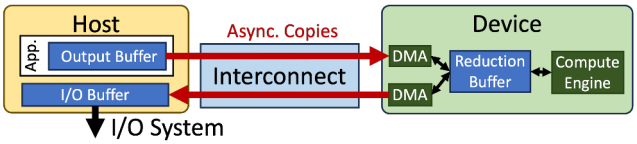

Fig. 8: Host-Device Execution Model

A. Host-Device Execution Model

First, to enable portability across different GPU-accelerated systems, we use a machine abstraction, Host-Device Execution Model (HDEM), to design our pipeline. Figure 8 shows our execution model that represents the common architecture in many modern GPU-based systems. One GPU device in our abstraction has two Direct Memory Access (DMA) engines, each of which can work independently for asynchronous memory copy. They can support data copies between the application buffer, I/O buffer, and reduction buffer. The device also has a compute engine to support the concurrent execution of reduction kernels during data copy operations.

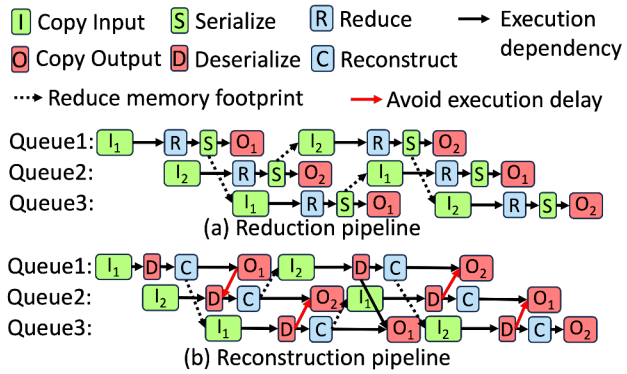

Fig. 9: Optimized reduction and reconstruction pipeline represented as DAGs that enable data transfer latency hiding. Green: host-to-device copy; Red: device-to-host copy; Blue: compute

B. Optimized Pipeline

Next, to ease the design of our pipeline, we consider two restrictions: (1) we assume reduction kernels are already highly optimized by developers so that they can achieve maximum occupancy with large input sizes. So, we restrict only one kernel running at the same time. (2) As moving both input and output data can take a considerable amount of time in a reduction pipeline, we dedicate one DMA for moving input data and another for moving output to exploit data movement overlap in opposite directions. We do not consider the data movement overlaps in the same direction (e.g., copying output and deserialization in the reconstruction pipeline) since the potential improvement is limited.

To fully use the hardware throughput, our pipeline needs to have enough depth for latency hiding. Depending on the time complexity of the reduction algorithm, data compressibility, and hardware performance, the overall performance of our pipeline could rely on the throughput of one of the DMAs or the compute engine. So, according to Little's law [45], three is the minimum depth for our pipeline to fully use the hardware. Figure 9 shows our optimized pipeline for both reduction and reconstruction. In the figure, the reduction process is pipelined among three queues (1-3). Green boxes represent host-to-device (H2D) DMA copy tasks. Red boxes represent device-to-host (D2H) DMA copy tasks. Blue boxes are compute tasks. According to our restrictions, no two tasks with the same color can be executed at the same time. We assume serialization and deserialization are needed for embedding and extracting metadata after and before computation, which also relies on D2H and H2D copies.

Theoretically, a pipeline with three execution queues requires three distinct input/output buffers for correctness. This imposes challenges for data reduction since a high memory footprint could limit the input data chunk size of each reduction process, which in turn limits the overall data compressibility and computing efficiency of reduction. To reduce the memory footprint, we insert additional dependencies in the pipeline (marked as dotted lines) to avoid data races between executions on queue X and $(X+2)\%3$. This optimization reduces the needed input/output buffers to two: I_1/O_1 and I_2/O_2. To explain how the extra dependencies eliminate the needs for the third set of buffers, we use the dependency between I_1 and S in the reduction pipeline as an example. This dependency enforces that the third reduction pipeline must not start until the serialization of the first reduction process finishes, which indicates the buffer I_1 is no longer needed. So, we can immediately reuse I_1 as the input buffer for the third reduction process; We add the same dependencies to the reconstruction pipeline. In addition, we also add dependencies (red arrows) to optimize launching orders for each operation. During the reconstruction process, both the deserialization operation of the next upcoming process and the output copy operation rely on the same D2H DMA, which will cause serialized execution. By default, copying output is initialized right after the previous reconstruction and before the deserialization of the next reconstruction. However, this can cause serious delays to the next reconstruction. So, we reverse the order of the two such that deserialization is done first, and the reconstruction overlaps with the output copy operation.

C. Optimizing chunk size

The chunk size can greatly impact the overall pipeline performance in terms of compute efficiency of reduction kernels on GPU and the effectiveness of compute and communication overlap. We define the overlap ratio as:

$$Overlap = \frac{\text{Total overlapped H2D and D2H time}}{\text{Total H2D and D2H time}}$$

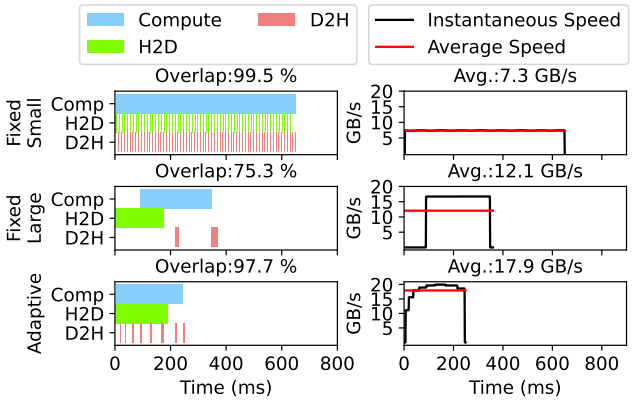

Fig. 10: Reduction pipeline of compressing a 4.3 GB NYX variable with $1e^{-2}$ error bound using MGARD. Fixed small and large use fixed chunks of size 100 MB and 2 GB. Adaptive represents our adaptive chunk size adjustment strategy.

On one hand, when the chunk size is small, the pipeline tends to have a high overlap ratio because reduction computation can start soon after the first small chunk is transferred to the GPU. However, due to limited input size, the computing efficiency may be limited. On the other hand, when the chunk size is large, each chunk is large enough to fully utilize the GPU to achieve high reduction throughput. However, the reduction process needs to wait for a longer time for the first chunk to be transferred. Figure 10 shows the effect of using different chunk sizes. With a small chunk size, the sustained throughput is low (7.3 GB/s) as chunks are too small to fully occupy the GPU. When the chunk size is large, only 75.3% of the data transfer latency can be hidden. To both improve the

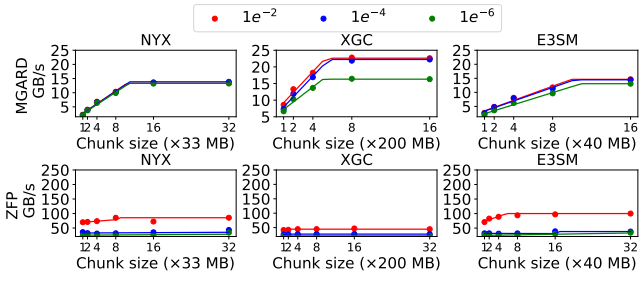

Fig. 11: Modeling MGARD and ZFP performance of different chunk sizes using the Roofline model

overlap ratio while maximizing sustained throughput, we build a reduction pipeline with an adaptive chunk size adjusting strategy as shown in Algorithm 4. Our algorithm first uses a small user-specified chunk size as the initial size (line 2) to reduce the leading time of the whole pipeline. To minimize the GPU computing inefficiency caused by small chunk sizes, we adaptively increase the chunk size as reduction make progresses. We determine the next chunk size by estimating how much data we can transfer to GPU while GPU is working

Algorithm 4: Adaptive Pipeline

1 **Function** `AdaptivePipeline()`:
  In : Data u
  In : Initial chunk size C_{init}
2 $C_{curr} \leftarrow C_{init}$;
3 $in\_offset, out\_offset \leftarrow 0$
4 $size_{rest} \leftarrow LargestDim(u)$
5 $Q[3] \leftarrow$ init. device queues
6 $I[2] \leftarrow$ init. input device buffers
7 $O[2] \leftarrow$ init. output device buffers
8 $i, j \leftarrow 0$ $C_{limit} \leftarrow$ maximum chunk size
9 $I[j] \xleftarrow[copy\ size=C_{curr}]{H2D\ on\ queue\ Q[i]} u[offset]$
10 $size_{rest} = size_{rest} - C_{curr}$
11 $in\_offset = in\_offset + C_{curr}$
12 **while** $size_{rest} \geq 0$ **do**
13 $i_{next} = (i+1)\%2$
14 $j_{next} = (j+1)\%3$
15 **if** $size_{rest} > 0$ **then**
16 $C_{next} \leftarrow min(\Theta(C_{curr}/\Phi(C_{curr})), C_{limit})$
17 $C_{next} \leftarrow min(size_{rest}, C_{next})$
18 $I[j_{next}] \xleftarrow[copy\ size=C_{next}]{H2D\ on\ queue\ Q[i_{next}]} u[offset]$
19 $size_{rest} = size_{rest} - C_{curr}$
20 $in\_offset = in\_offset + C_{curr}$
21 **end**
22 $O[j] \xleftarrow{Reduce\ on\ queue\ Q[i]} I[j]$
23 $u_{reduced}[out\_offset] \xleftarrow{D2H\ on\ queue\ Q[i]} O[j]$
24 $out\_offset = out\_offset + O[j].size$
25 $C_{curr} = C_{next}$
26 $i = i_{next}$
27 $j = j_{next}$
28 **end**
29 **return** $u_{reduced}$

on the current chunk. This can ensure the memory copy time is completely hidden by the compute time. This needs us to build two estimation functions: (1) $p = \Phi(C)$: estimating the reduction throughput p on GPU given a chunk size C (2) $C_{max} = \Theta(t)$: estimating the maximum chunk size C_{max} can be transferred from host to device given a time limit t. Then, we can use the current chunk size C_{curr} and the maximum chunk size limited by GPU memory C_{limit} to compute the next chunk size $C_{next} = min(\Theta(C_{curr}/\Phi(C_{curr})), C_{limit})$ (line 17). We build $\Phi(C)$ using our modified roofline model:

$$\Phi(C) = \begin{cases} \alpha \times C + \beta & \text{if } C < C_{threshold} \\ \gamma & \text{if } C \geq C_{threshold} \end{cases}$$

We use a linear model while chunk sizes are small and GPU is not saturated and use a constant function to estimate saturated throughput when chunk size is larger than a threshold $C_{threshold}$. The model can be obtained by profiling a given dataset and error bound on different chunk sizes until the chunk is large enough to saturate the GPU. We use the

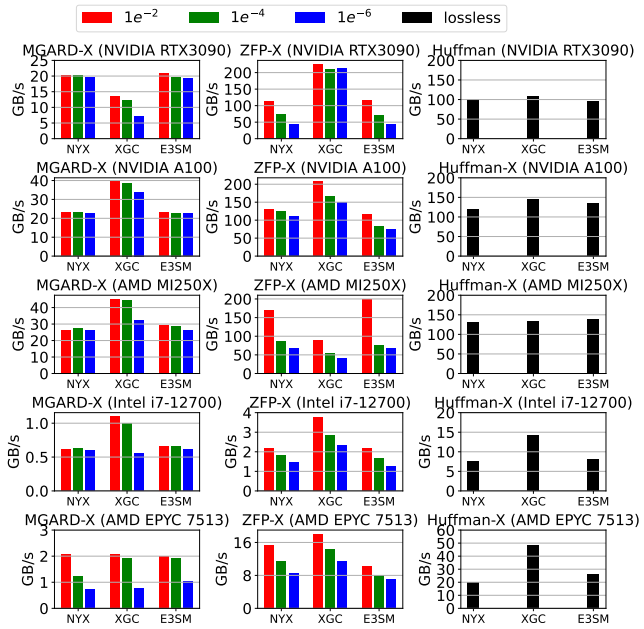

Fig. 12: Kernel throughput of portable MGARD-X, ZFP-X, and Huffman-X implementations on five processors. MGARD-X and ZFP-X compress with three different relative error bounds and Huffman-X provides lossless compression.

throughput of the largest profiled chunk to determine γ and orderly check on the throughput of smaller chunks until the throughput drops below $f \times \gamma$ (e.g., $f = 0.1$), then we use linear regression to fit the rest chunks sizes to obtain the linear model. Figure 11 shows the profiling results and fitted models on three datasets with three error bounds. As for Θ we use a linear model to estimate the maximum transferable chunk size: $\Theta(t) = \frac{t}{\beta}$. We treat host-to-device throughput β as constant since we do not consider small chunk sizes that do not saturate CPU-GPU interconnect, which would lead to an inefficient pipeline.

VI. Experimental Evaluation

A. Experimental Methodology

We implemented three data reduction pipelines, MGARD [17], ZFP [19], and Huffman [40], using HPDR based on their published algorithm designs. We name our portability implementation as MGARD-X, ZFP-X, and Huffman-X. As comparison reference, we use the current release version of GPU implementation of MGARD, ZFP, SZ, and LZ4: `MGARD-GPU v1.5` [24], `ZFP-CUDA v1.0` [27], `cuSZ v0.6` [29], `NVCOMP-LZ4 v2.2` [26]. At the time of evaluation, ZFP only supports fix-rate mode on GPU, so we only use fix-rate mode in our evaluation. Also, we restrict our evaluation to only compare with stable version of each software, so we exclude the evaluation of HIP version of SZ and ZFP on Frontier. For parallel I/O evaluations, we integrate each reduction routine into the I/O pipeline of the

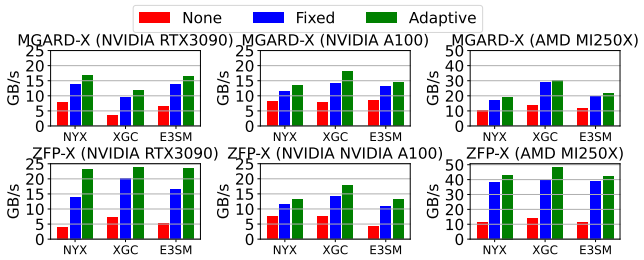

Fig. 13: Comparing end-to-end throughput of MGARD and ZFP using (1) None: no overlapping pipeline; (2) Fixed: fixed size pipeline; and (3) Adaptive: adaptive size pipeline

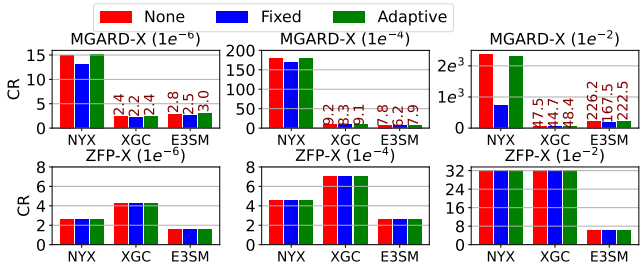

Fig. 14: Comparing compression ratio of MGARD and ZFP with three pipeline settings and errors: $1e^{-2}$, $1e^{-4}$, and $1e^{-6}$

ADIOS 2 library [35] and use the latest BP5 file format. Aggregation strategy is tuned for each system (i.e., one writer per node on Summit and one writer per GPU on Frontier) to achieve the best I/O performance. In addition, three scientific datasets are used in our data reduction evaluations:

TABLE III: Datasets used for evaluation

Dataset	Field	Dimensions	Data Type	Size
NYX	density	$512 \times 512 \times 512$	FP32	536.8MB
XGC	e_f	$8 \times 33 \times 1117528 \times 37$	FP64	87.3 GB
E3SM	PSL	$2880 \times 240 \times 960$	FP32	2.7 GB

B. Experimental Environment

All evaluates are done on four platforms: Summit, Frontier, Jetstream2, and a GPU workstation. Summit and Frontier are two leadership class supercomputers at Oak Ridge Leadership Computing Facility (OLCF). Summit is a 4,608-node pre-exascale supercomputer using a GPFS filesystem with peak I/O bandwidth of 2.5 TB/s. Each compute node is equipped with 6 NVIDIA V100 GPUs with 16 GB memory on each GPU and two 22-core IBM POWER9 CPUs with 512 GB memory. Frontier is a 9,408-node exascale supercomputer using a Luster filesystem with peak I/O bandwidth of 9.4 TB/s. Each compute node is equipped with 4 AMD Instinct MI250X GPUs with 128 GB memory on each GPU and one 64-core AMD EPYC CPU with 512 GB memory. The Jetstream2 is a system provided by the Indiana University via the NSF ACCESS program [46]. Jetstream2 includes 90 GPU-enabled nodes and each node is equipped with 4 NVIDIA

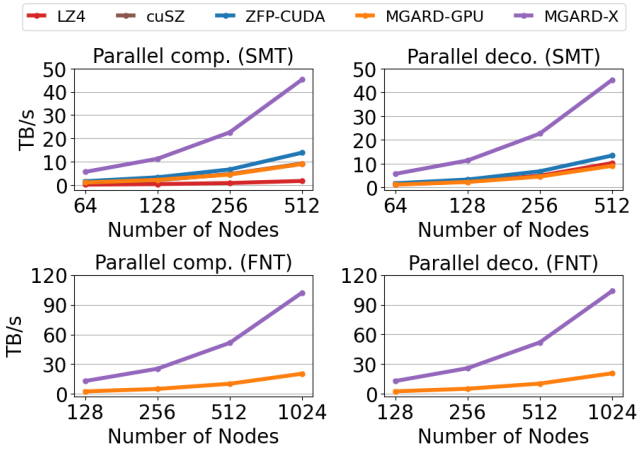

Fig. 15: Aggregated compression and decompression throughput on Summit (SMT) and Frontier (FNT)

A100 GPUs with 40 GB memory on each GPU and two 64-core AMD Milan 7713 CPUs with 512 GB memory. The GPU workstation is equipped with an NVIDIA RTX 3090 GPU with 24 GB memory and 20-core Intel i7 CPU with 32 GB memory.

C. Kernel-level portability and performance

We evaluate the portability and performance of HPDR on five different CPU and GPU processors for each reduction kernel we implemented. Figure 12 shows the performance of reduction kernel excluding the data transferring cost between host and device. On GPUs, we achieve up to 45 GB/s, 210 GB/s, and 150 GB/s throughput for MGARD-X, ZFP-X, and Huffman-X respectively. On CPUs, we obtain up to 2 GB/s, 18 GB/s and 48 GB/s throughput for the three reduction kernels.

D. Single-GPU end-to-end pipeline performance

Next, we evaluate the end-to-end pipeline performance on a single GPU with and without our proposed optimization techniques. We exclude applying pipeline optimization on CPUs because the data transfer time typically is not the dominating factor of the whole reduction pipeline such that our optimizations can only achieve limited improvement. Figure 13 shows the end-to-end pipeline performance including both data transfer and computation costs. Compared with not using an overlapping pipeline, our fixed size pipeline (100 MB chunk) achieved up to $2.1\times$ and $3.5\times$ speedups for MGARD-X and ZFP-X, respectively. In addition, compared with the fixed size pipeline, our adaptive size pipeline achieved up to $1.3\times$ and $1.6\times$ speedups for the three reduction techniques, respectively. Figure 14 compares the compression ratios when using three different pipeline settings. Compared with not using overlapping pipelines, the fixed size pipeline reduces the compression ratios by $5-67\%$ for MGARD due to degraded compressibility of the small chunk size. Our Adaptive pipeline brings a similar compression ratio ($<1\%$ difference) compared with the non-pipeline setting, as it leverages large chunk sizes that have better compressibility. The pipeline setting

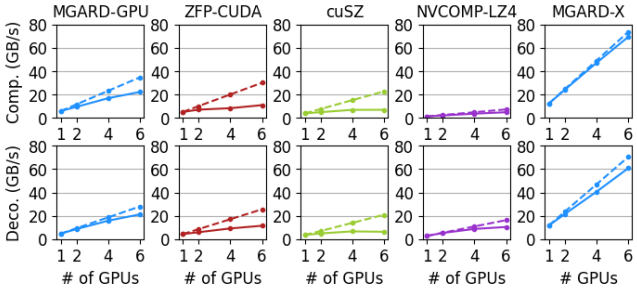

Fig. 16: Scalability on multiple V100 GPUs. Solid: real throughput; Dotted: throughput of ideal scalability

brings negligible impact to ZFP as it compresses data blocks by blocks with much smaller generality than our chunk size.

E. Multi-GPU end-to-end performance and scalability

As modern supercomputers commonly use dense multi-GPU architectures, we evaluate the performance and scalability of HPDR on a Summit node with 6 V100 GPUs. As shown in Figure 16, we use MGARD-X as an example to compare the performance and scalability of HPDR with four other data reduction not implemented using HPDR. To measure the scalability, we calculate the ideal speed at scale by multiplying the speed on one GPU with the number of GPUs. To quantify the scalability, we calculate the average real-to-ideal speed ratios across tests on different numbers of GPUs. Because of our context memory management optimization, HPDR minimizes most of the device memory operations during the execution of the pipelines, which could cause contention in the multi-device environment. For compression, MGARD-X achieved 96% avg. scalability while the non-optimized MGRAD, ZFP, SZ, and LZ4 achieved 72%, 48%, 46%, and 74% avg. scalability. Similarly for decompression, MGARD-X achieved 88% avg. scalability while existing designs achieved 76%, 55%, 48%, and 70% avg. scalability.

F. Multi-node end-to-end performance

Figure 15 shows the multi-node aggregated data reduction end-to-end throughput on Summit and Frontier. We perform a weak scaling test where each node compression and decompresses the NYX data. To fully saturate the data reduction pipelines on each GPUs, we let each GPU process 14 time steps of NYX data. For Summit, we scale the compression and decompression process up to 512 nodes using the computing power of 3,072 V100 GPUs and 23 TB of data. At this scale, MGARD-X achieved 45 TB/s throughput while NVCOMP-LZ4, cuSZ, ZFP-CUDA, and MGARD-GPU only achieved 10 TB/s, 9 TB/s, 13 TB/s, and 9 TB/s throughput respectively. For Frontier, we scaled our test using up to 1,024 nodes using the computing power of 4,096 MI250X GPUs and 62 TB of data. At this scale, MGARD-X achieved 103 TB/s throughput while MGARD-GPU only achieved 18 TB/s.

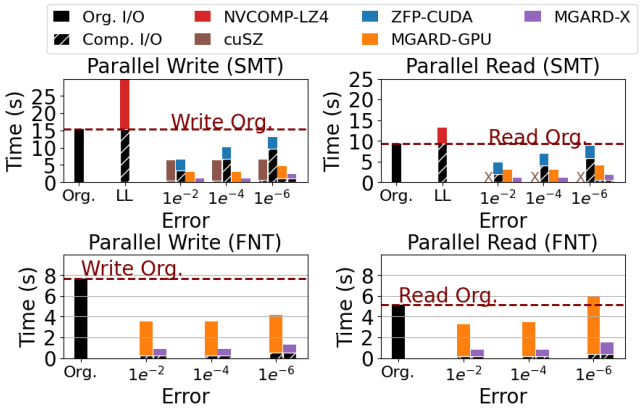

Fig. 17: Weak scaling parallel I/O using NYX data on Summit (SMT) and Frontier (FNT)

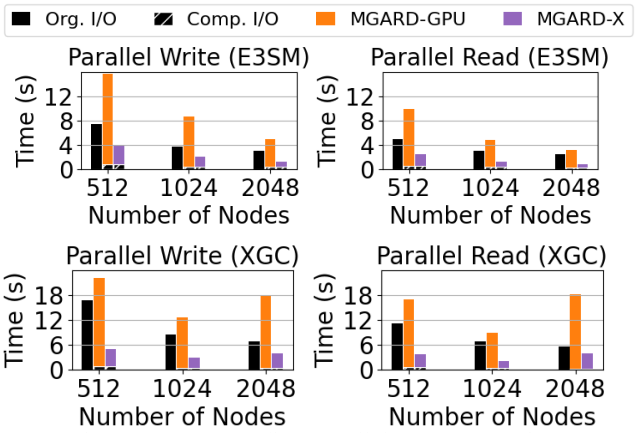

Fig. 18: Strong scaling parallel I/O on Frontier

G. Weak scaling I/O acceleration evaluation

To evaluate the effectiveness of using data reduction to accelerate parallel I/O at scale, we first perform weak scaling I/O using the NYX data on Summit with 512 nodes and Frontier with 1,024 nodes. Each GPU compresses 7.5 GB of data. Figure 17(a) shows the results on Summit. Compared with the original read and write cost, NVCOMP-LZ4 cannot bring I/O acceleration due to a limited compression ratio ($1.1\times$) with extra computational overhead, which leads to an extra 83.5% and 42.7% read/write overhead. cuSZ achieved $2.3-2.4\times$ write acceleration with compression ratio $20-31\times$. However, cuSZ crashes at scales larger than 64, so we could not measure its read acceleration. ZFP-CUDA achieved $1.2-2.3\times$ write acceleration and $1.1-1.9\times$ read acceleration with $2.4-32\times$ compression ratio. MGARD-GPU archived $3.3-5.1\times$ write acceleration and $2.3-3.1\times$ read acceleration with $14-2379\times$ compression ratio. MGARD-X archived $6.8-15.3\times$ write acceleration and $5.2-9.3\times$ read acceleration with the same compression ratio as MGARD-GPU. Figure 17(b) shows the results on Frontier. Compared with the original read and write cost, MGARD-GPU archived $1.8-2.1\times$ write acceleration and $0.8-1.5\times$ read acceleration with the same compression ratio as on Summit. MGARD-X archived $6.0-8.5\times$ write acceleration and $3.5-6.5\times$ read acceleration with the same compression ratio as MGARD-GPU.

H. Strong scaling I/O acceleration evaluation

In addition to the weak scaling test, we also perform strong scaling write and read I/O test with and without data reduction on Frontier. Figure 18 (a) shows the I/O cost of writing and reading 32 TB of E3SM data using 512, 1024, and 2048 nodes. The data is compressed with a relative error bound of 10^{-4} with a compression ratio of $7.9\times$ for both MGARD-GPU and MGARD-X. Compared with I/O without reduction, MGARD-GPU brings 28% - 134% extra overhead due to low reduction throughput. MGARD-X, on the other hand, can accelerate write by $2.4-1.8\times$ and read by $2.1-2.9\times$ across different scales. Figure 18 (a) shows the I/O cost of writing and reading 67 TB of XGC data using 512, 1024, and 2048 nodes. The data is compressed with a relative error bound of 10^{-4} with a compression ratio of $9.1\times$ for both MGARD-GPU and MGARD-X. Compared with I/O without reduction, MGARD-GPU brings 32% - 227% extra overhead due to low reduction throughput. MGARD-X, on the other hand, accelerates write by $1.7-3.4\times$ and read by $1.5-3.3\times$ across different scales.

VII. Conclusion

The increasing gap between scientific data generation and the capabilities of computing systems to process and analyze this data underscores the urgent need for effective data reduction strategies. While GPU-accelerated data reduction techniques have shown promise, several challenges, such as limited portability, memory-bound performance, and scalability issues, still hinder their broader adoption in exascale workflows. To address these obstacles, we developed HPDR, a portable, high-performance data reduction framework that optimizes memory transfer overhead and enhances scalability across multiple GPU and CPU architectures. Through rigorous testing and integration with large-scale systems, HPDR demonstrates significant improvements in data reduction throughput and I/O acceleration, highlighting its potential to transform scientific workflows in the exascale era.

Acknowledgment

This work used Jetstream2 system through allocation CIS230203 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by U.S. National Science Foundation (NSF) grants 2138259, 2138286, 2138307, 2137603, and 2138296. This work was partially supported by the NSF under Grant OAC-2311756, OAC-2313122, and OIA-2327266. This research was also supported by the SIRIUS-2 ASCR research project, the Scientific Discovery through Advanced Computing (SciDAC) program, specifically the RAPIDS-2 SciDAC institute.

REFERENCES

[1] "Frontier Supercomputer:." [Online]. Available: https://www.olcf.ornl.gov/frontier/

[2] "Aurora Supercomputer:." [Online]. Available: https://www.alcf.anl.gov/aurora

[3] "ITER Project:." [Online]. Available: https://www.iter.org

[4] "SKA Project:." [Online]. Available: https://www.skao.int/en

[5] J.-C. Golaz, P. M. Caldwell, L. P. Van Roekel, M. R. Petersen, Q. Tang, J. D. Wolfe, G. Abeshu, V. Anantharaj, X. S. Asay-Davis, D. C. Bader *et al.*, "The doe e3sm coupled model version 1: Overview and evaluation at standard resolution," *Journal of Advances in Modeling Earth Systems*, vol. 11, no. 7, pp. 2089–2129, 2019.

[6] S. Ku, C.-S. Chang, and P. Diamond, "Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry," *Nuclear Fusion*, vol. 49, no. 11, 2009.

[7] C.-S. Chang, S. Ku, and H. Weitzner, "Numerical study of neoclassical plasma pedestal in a tokamak geometry," *Physics of Plasmas*, vol. 11, no. 5, 2004.

[8] R. Taylor and R. Braun, "Science with the Square Kilometer Array: Motivation, key science projects, standards and assumptions," *arXiv preprint astro-ph/0409274*, 2004.

[9] A. Chrysostomou, C. Taljaard, R. Bolton, L. Ball, S. Breen, and A. van Zyl, "Operating the square kilometre array: the world's most data intensive telescope," in *Observatory Operations: Strategies, Processes, and Systems VIII*, vol. 11449. SPIE, 2020, pp. 156–170.

[10] F. Knorr, P. Thoman, and T. Fahringer, "ndzip: A high-throughput parallel lossless compressor for scientific data," in *2021 Data Compression Conference (DCC)*. IEEE, 2021, pp. 103–112.

[11] J. Liu, S. Li, S. Di, X. Liang, K. Zhao, D. Tao, Z. Chen, and F. Cappello, "Improving lossy compression for sz by exploring the best-fit lossless compression techniques," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 2986–2991.

[12] B. Zhang, J. Tian, S. Di, X. Yu, M. Swany, D. Tao, and F. Cappello, "Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 348–359.

[13] M. Shah, X. Yu, S. Di, M. Becchi, and F. Cappello, "Lightweight huffman coding for efficient gpu compression," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 99–110.

[14] Y. Mao, Y. Cui, T.-W. Kuo, and C. J. Xue, "Trace: A fast transformer-based general-purpose lossless compressor," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1829–1838.

[15] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, Z. Chen, and F. Cappello, "Significantly improving lossy compression for hpc datasets with second-order prediction and parameter optimization," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 89–100.

[16] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1643–1654.

[17] Q. Gong, J. Chen, B. Whitney, X. Liang, V. Reshniak, T. Banerjee, J. Lee, A. Rangarajan, L. Wan, N. Vidal *et al.*, "Mgard: A multigrid framework for high-performance, error-controlled data compression and refactoring," *SoftwareX*, vol. 24, p. 101590, 2023.

[18] X. Liang, B. Whitney, J. Chen, L. Wan, Q. Liu, D. Tao, J. Kress, D. Pugmire, M. Wolf, N. Podhorszki *et al.*, "Mgard+: Optimizing multilevel methods for error-bounded scientific data reduction," *IEEE Transactions on Computers*, vol. 71, no. 7, pp. 1522–1536, 2021.

[19] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[20] Q. Gong, Z. Wang, V. Reshniak, X. Liang, J. Chen, Q. Liu, T. M. Athawale, Y. Ju, A. Rangarajan, S. Ranka *et al.*, "A general framework for error-controlled unstructured scientific data compression," in *2024 IEEE 20th International Conference on e-Science (e-Science)*. IEEE, 2024, pp. 1–10.

[21] T. Banerjee, J. Lee, J. Choi, Q. Gong, J. Chen, S. Klasky, A. Rangarajan, and S. Ranka, "Fast algorithms for scientific data compression," in *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2023, pp. 143–152.

[22] T. Banerjee, J. Lee, J. Choi, Q. Gong, J. Chen, C. Chang, S. Klasky, A. Rangarajan, and S. Ranka, "Online and scalable data compression pipeline with guarantees on quantities of interest," in *2023 IEEE 19th International Conference on e-Science (e-Science)*. IEEE, 2023, pp. 1–10.

[23] X. Liang, Q. Gong, J. Chen, B. Whitney, L. Wan, Q. Liu, D. Pugmire, R. Archibald, N. Podhorszki, and S. Klasky, "Error-controlled, progressive, and adaptable retrieval of scientific data with multilevel decomposition," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[24] J. Chen, L. Wan, X. Liang, B. Whitney, Q. Liu, D. Pugmire, N. Thompson, J. Y. Choi, M. Wolf, T. Munson *et al.*, "Accelerating multigrid-based hierarchical scientific data refactoring on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 859–868.

[25] X. Wu, Q. Gong, J. Chen, Q. Liu, N. Podhorszki, X. Liang, and S. Klasky, "Error-controlled progressive retrieval of scientific data under derivable quantities of interest," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1–16.

[26] "NVCOMP:." [Online]. Available: https://developer.nvidia.com/nvcomp

[27] P. Lindstrom, J. Hittinger, J. Diffenderfer, A. Fox, D. Osei-Kuffuor, and J. Banks, "Zfp: A compressed array representation for numerical computations."

[28] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao *et al.*, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 3–15.

[29] J. Tian, S. Di, X. Yu, C. Rivera, K. Zhao, S. Jin, Y. Feng, X. Liang, D. Tao, and F. Cappello, "cusz (x): Optimizing error-bounded lossy compression for scientific data on gpus," *CoRR*, 2021.

[30] Y. Huang, S. Di, X. Yu, G. Li, and F. Cappello, "cuszp: An ultra-fast gpu error-bounded lossy compression framework with optimized end-to-end performance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[31] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[32] AMD, *HIP Programming Guide*, 2025, accessed: [2025]. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP

[33] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator programming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[34] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[35] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[36] S. Lakshminarasimhan, N. Shah, S. Ethier, S.-H. Ku, C.-S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Isabela for effective in situ compression of scientific data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 524–540, 2013.

[37] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[38] B. Zhang, J. Tian, S. Di, X. Yu, Y. Feng, X. Liang, D. Tao, and F. Cappello, "Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 129–142.

[39] C. Flint, A. Huppé, P. Helluy, B. Bramas, and S. Genaud, "Using the discrete wavelet transform for lossy on-the-fly compression of gpu fluid simulations," 2024.

[40] J. Tian, C. Rivera, S. Di, J. Chen, X. Liang, D. Tao, and F. Cappello, "Revisiting huffman coding: Toward extreme performance on modern gpu architectures," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 881–891.

[41] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.
[42] A. Alpay and V. Heuveline, "Sycl beyond opencl: The architecture, current state and future direction of hipsycl," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–1.
[43] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on gpu," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.
[44] B. Accelerated, S. A. Ostadzadeh, Z. Zeinalpour-tabrizi, M. A. Moulavi, and K. Bertels, "A two-phase practical parallel algorithm for construction of huffman codes." 2007.
[45] J. D. Little and S. C. Graves, "Little's law," *Building intuition: insights from basic operations management models and principles*, pp. 81–100, 2008.
[46] T. J. Boerner, S. Deems, T. R. Furlani, S. L. Knuth, and J. Towns, "Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support," in *Practice and Experience in Advanced Research Computing*, 2023, pp. 173–176.