HP-MDR: High-performance and Portable Data Refactoring and Progressive Retrieval with Advanced GPUs

Yanliang Li*
University of Oregon
Eugene, Oregon, USA
leonli@uoregon.edu

Wenbo Li*
University of Kentucky
Lexington, Kentucky, USA
Wenbo.Li@uky.edu

Qian Gong
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
gongq@ornl.gov

Qing Liu
New Jersey Institute of Technology
Newark, New Jersey, USA
qing.liu@njit.edu

Norbert Podhorszki
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
pnorbert@ornl.gov

Scott Klasky
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
klasky@ornl.gov

Xin Liang
University of Kentucky
Lexington, Kentucky, USA
xliang@uky.edu

Jieyang Chen
University of Oregon
Eugene, Oregon, USA
jieyang@uoregon.edu

Abstract

Scientific applications produce vast amounts of data, posing grand challenges in the underlying data management and analytic tasks. Progressive compression is a promising way to address this problem, as it allows for on-demand data retrieval with significantly reduced data movement cost. However, most existing progressive methods are designed for CPUs, leaving a gap for them to unleash the power of today's heterogeneous computing systems with GPUs.In this work, we propose HP-MDR, a high-performance and portable data refactoring and progressive retrieval framework for GPUs. Our contributions are four-fold: (1) We carefully optimize the bitplane encoding and lossless encoding, two key stages in progressive methods, to achieve high performance on GPUs; (2) We propose pipeline optimization and incorporate it with data refactoring and progressive retrieval workflows to further enhance the performance for large data process; (3) We leverage our framework to enable high-performance data retrieval with guaranteed error control for common Quantities of Interest; (4) We evaluate HP-MDR and compare it with state of the arts using five real-world datasets. Experimental results demonstrate that HP-MDR delivers an average 13.68× and 6.31× throughput in data refactoring and progressive retrieval tasks, respectively. It also leads to 11.22× throughput for recomposing required data representations under Quantity-of-Interest error control and 6.04× performance for the corresponding end-to-end data retrieval, when compared with state-of-the-art solutions.

CCS Concepts

• **Information systems** → **Data management systems**; • **Computing methodologies** → **Parallel algorithms**.

*Both authors contributed equally to this research.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SC '25, St Louis, MO, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759845

Keywords

High-performance computing, scientific data, progressive compression, advanced GPUs

ACM Reference Format:
Yanliang Li, Wenbo Li, Qian Gong, Qing Liu, Norbert Podhorszki, Scott Klasky, Xin Liang, and Jieyang Chen. 2025. HP-MDR: High-performance and Portable Data Refactoring and Progressive Retrieval with Advanced GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3712285.3759845

1 Introduction

With the recent deliveries of exascale computing systems [1–3], scientific applications are producing an unprecedented amount of data that overwhelms the storage and data transfer systems. This poses grand challenges in the design and development of exascale data management systems, necessitating the need for efficient and effective data reduction.

Error-controlled lossy compression is a direct way to address the scientific data challenge, as it can significantly reduce the size of scientific data while enforcing user-specified error controls. It has developed rapidly in the last decade, and has been widely deployed in a broad range of application domains, including climatology [4], cosmology [5], fusion [6], and artificial intelligence [7]. Nonetheless, error-controlled lossy compression has a severe limitation that prevents its broader adoption in science: it provides only a single precision, although the data may be used for different scientific analytics with diverse precision requirements. This leads to a dilemma for scientists when choosing the proper error control. On the one hand, strict error control may ensure data fidelity for most downstream tasks, but it yields limited benefits in data transfer and storage. On the other hand, loose error control can significantly mitigate the pressure on data movement, but it may produce wrong results for downstream tasks that require high precision.

Data refactoring with error-controlled progressive retrieval [8–11] has been recently proposed and regarded as an alternative way

to manage scientific data. Similar to progressive compression with JPEG/JPEG2000 [12, 13] in the image processing community, these approaches refactor data into different precision/resolution segments. During retrieval, these segments are used to reconstruct the data toward user-specified error control in a progressive and incremental fashion. While this does not reduce the data size for storage, it significantly improves the efficiency of data retrieval by providing just enough precision on demand. Meanwhile, it eliminates the risk of inaccurate data in classic error-controlled lossy compression, as it can provide near-lossless representations that satisfy the requirement for most downstream tasks.

Although several progressive methods [8–10] have been proposed in literature, most of them are designed and optimized for CPU architectures. Nonetheless, almost all recent leadership computing facilities are highly heterogeneous with modern GPUs, leaving a significant gap for progressive methods to fully unleash the computational power of these systems. This situation is further exacerbated by the diverse GPU architectures on those systems (e.g., NVIDIA GPUs on Summit [14], AMD GPUs on Frontier [2], and Intel GPUs on Aurora [1]), as a tailored implementation on one system may not work for the others. Because of their fundamental differences in architecture, a data refactoring pipeline may use a different algorithm variant on different processor types for optimized performance. Such differences can cause data portability challenges: data refactored by one type of processor cannot be reconstructed by another type of processor with a guarantee. Due to portability challenges, users are forced to use the most compatible processor to ensure data are still retrievable on future systems that use different architectures. However, the most compatible processors, such as a single-core CPU, cannot guarantee the best performance.

In this work, we propose a high-performance and portable framework – HP-MDR, implementing progressive methods on exascale systems with advanced GPUs. In particular, we propose end-to-end portable refactoring and reconstruction pipelines with highly optimized register block-based bitplane encoding and hybrid lossless encoding strategies. To this end, we construct a pipeline to compose PMGARD [9], a state-of-the-art progress method, and further design an execution workflow to enable progressive retrieval with guaranteed error control on Quantities of Interest (QoIs), which represent derived information of most interest to application scientists. In summary, our contributions are as follows.

- Based on the algorithmic characteristics of bitplane encoding and many-core architectures, we thoroughly study several optimized parallel bitplane encoding strategies, and we design a highly optimized bitplane encoder that accelerates encoding by 2.1× and decoding by 8.3× on modern GPUs while providing portability across architecture types.
- To adapt to the diverse compressibility of bitplanes, we build a hybrid lossless compression that accelerates bitplane compression throughput by 3.1× with only 8% data retrieval overhead on average.
- To improve GPU utilization, we design a highly optimized pipeline that accelerates refactoring and reconstruction by 1.43× and 1.83× on average. This yields over 6.6× end-to-end throughput when compared with existing approaches.
- We leverage our optimized encoding algorithm and pipeline to enable progressive retrieval with guaranteed error control under common Quantities of Interest (QoIs), leading to 11.22× throughput for data recomposition and 6.04× performance for the underlying end-to-end data retrieval.

The rest of the paper is organized as follows. In Section 2, we discuss the related work on scientific data compression and progressive methods. In Section 3, we provide an overview of the proposed framework. We then detail our optimization on bitplane encoding and lossless compression in Section 4 and Section 5, respectively. In Section 6, we describe how to compose PMGARD in our framework and enable error control on downstream QoIs with high performance. In Section 7, we present our evaluation results with real-world datasets. In Section 8, we conclude the paper with a vision for future works.

2 Related Works

In this section, we review the related works on scientific data compression, which is broadly categorized into error-controlled lossy compression and progressive compression.

2.1 Error-controlled lossy compression

The ever-increasing amount of scientific data imposes grand challenges on the underlying data management and analytic tasks, which cannot be addressed by generic lossless compressors such as GZIP [15] and ZSTD [16] due to their limited compression ratios. Meanwhile, error-controlled lossy compression [17–26] has been evolving as a promising solution because it significantly reduces data size while enforcing user-specified error controls that are essential for scientific applications.

Error-controlled lossy compressors can be broadly categorized as prediction-based and transform-based ones. SZ [20–22, 26] is one of the most widely used prediction-based lossy compressors. It relies on various predictors, including Lorenzo [27] and splines [22], to decorrelate the data, followed by a linear-quantization stage to reduce the entropy while ensuring error control. The quantized data is then fed to lossless encoders such as Huffman [28] and ZSTD for further size reduction. ZFP [19] is a typical transform-based lossy compressor that leverages a block transform for decorrelation. In particular, it divides the original data into independent blocks, and performs a near-orthogonal transform after fixed-point alignment in each block. The transform coefficients are then encoded with an efficient embedded encoding algorithm and concatenated for storage. MGARD [23–25, 29] is another popular lossy compressor lying in the middle, which features rigorous error control theories on raw data and downstream Quantities of Interest (QoIs). It establishes a novel decomposition algorithm based on finite element analysis and wavelet theories, followed by linear quantization and lossless encoding stages similar to those of SZ. In addition to these mainstream compressors, scientific data compression has also been advanced by many other methods, such as wavelet transforms (SPERR [30]), singular value decomposition (TTHRESH [31]), and deep learning (AE-SZ [32]).

There has been a growing trend in implementing and optimizing error-controlled lossy compressors on GPUs to facilitate their use on leadership computing facilities. Nonetheless, adaptations of

compression algorithms are usually required to better unleash the parallel processing power of advanced GPUs due to either inherently sequential operations or underoptimized designs. For instance, cuSZ [33] adopts a dual-quantization design to eliminate the dependency in Lorenzo prediction, achieving decent compression ratios with high throughput. GPU-MGARD [34] optimizes three critical kernels for efficient grid, linear, and iterative processing, leading to significant speedup over a naive porting version. Recently, the throughput of the scientific data compression kernel has been pushed to hundreds of GB/s on NVIDIA GPUs [35].

One critical problem of error-controlled lossy compressors, along with their GPU variations, is that they only provide a single error bound and assume this accuracy could be sufficient for all subsequent data analytics. However, this could hardly be true due to the diverse requirements in scientific analytics, and scientists have to choose a conservative error bound during compression to ensure sufficient accuracy in the data. This usually leads to limited compression ratios, which limit the use of error-controlled lossy compressors in practice.

2.2 Progressive compression

Unlike error-controlled lossy compression, progressive compressors [8–11] store the data in a near-lossless fashion and provide on-demand access during retrieval. Although this may not improve the writing performance, it significantly reduces the data movement time during retrieval. This aligns well with the characteristics of scientific data, which is usually written once and retrieved multiple times for diverse analytics.

Progressive compression was first adopted in the image processing community, where JPEG/JPEG2000 [12, 13] divides the image into multiple scans to provide different quality levels. This allows for progressive rendering that starts with low quality but gradually refines with additional data, leading to a better experience for displaying images in webpages. PMGARD [9] borrowed this concept and took it to the scientific data domain by coupling MGARD decomposition theories and bitplane encoding algorithms to provide near-lossless data refactoring and error-controlled retrieval. It was recently improved to provide error control on a set of derived quantities of interest (QoIs) [11], significantly expanding its usage in practice. Another family of progressive methods relies on existing error-control lossy compressors to achieve progressiveness with error control [10]. In particular, they iteratively compress the original data and the corresponding residues with off-the-shelf compressors using a set of progressively decayed error bounds.

Despite the promising usage of progressive compression in scientific data management, little effort has been made to implement the entire procedure on advanced GPUs. While the iterative procedure [10] could be easily ported to GPUs using GPU-based error-controlled lossy compressors, it suffers from low efficiency because GPU-based lossy compressors are not adept at dealing with residue compression, especially when the error bound is relatively low. This leaves a significant gap for deploying progressive compression methods in the leadership computing facilities.

In this work, we propose HP-MDR, a high-performance portable data refactoring and progressive retrieval framework for advanced GPUs. In particular, we propose a set of tailored optimizations to

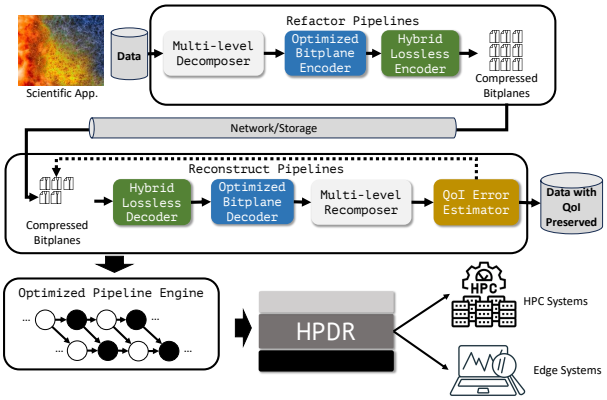


Figure 1: Overview of the HP-MDR data refactoring and reconstruction pipelines, including multi-level decomposition, optimized bit-plane encoding, hybrid lossless compression, and QoI-aware reconstruction. HP-MDR is built on top of the HPDR framework to enable GPU portability.

significantly improve the performance of bitplane encoding and lossless compression of the encoded bitplanes, which are identified as the primary performance bottlenecks. To this end, we couple our methods with GPU-MGARD to form end-to-end data refactoring and progressive retrieval pipelines, and we further enable error controls on downstream QoIs for practical usage.

3 System Overview

Figure 1 shows the overview of the system of the framework HP-MDR. The input data is first decomposed using the multi-level decomposer, then encoded into bitplanes, and finally compressed into reduced form. Part of the compressed bitplanes can be used to reconstruct data with error control via the reverse operations. In the traditional CPU-based pipeline [9], GPU-based acceleration is only available for the multilevel (re)decomposer [34], leading to limited throughput for the end-to-end procedures. In this work, we design the first end-to-end GPU-accelerated refactoring and reconstruction pipelines. We first study the main challenges of accelerating and parallelizing the bitplane encoding and propose optimized encoding kernels that maximize the GPU utilization for various encoding workloads. Then, we design a hybrid lossless compressor that adaptively leverages Huffman and Run-Length Encoding (RLE) to compress each bitplane efficiently. To further optimize end-to-end performance, we design pipeline optimizations that overlap CPU-GPU memory copy with computation for both refactoring and reconstruction pipelines. Furthermore, we leverage the pipeline optimization to build the first high-performance progressive retrieval pipeline with QoI error control on GPUs. Finally, the end-to-end computations are implemented on top of the High Performance Portable Data Reduction framework [36], which enables portability across various types of GPU and CPU architectures.

4 Bitplane Encoding

Bitplane encoding is a core component in many progressive compression frameworks, as it can provide very fine-grained precision decomposition to enable progressiveness. The encoding process is illustrated in Algorithm 1. Given a decomposed input array Q,

Algorithm 1: Bitplane Encoding Overview

Input: Decomposed data array Q of length N, target bitplane count B
Output: Bitplane-encoded stream S

```
/* 1. Shift all values to align MSBs                    */
aligned_Q ← AlignExponent(Q)
/* 2. Loop over B bitplanes                             */
for b ← B − 1 to 0 do
    encoded_bitplane ← empty array of length N ;
     // Initialize empty bitplane
    for i ← 0 to N − 1 do in parallel
        bit ← (aligned_Q[i] ≫ b) & 1 ; // Extract bit-b
        encoded_bitplane[i] ← bit ;      // Store bit-b
    write_bitplane(S, encoded_bitplane) ; // Flush one
     bitplane
```

the algorithm first performs exponent alignment so that bits in the mantissa are aligned for encoding. This alignment is performed by adjusting the exponent of each element with respect to the maximum exponent across all elements in a level, which preserves the most significant B bits after converted to integer form *align_Q*. It then iterates through B bitplanes from the most to the least significant bits. For each bitplane, it extracts the corresponding bit from every element, stores the results as 1 bit of the encoded bitplane, and finally writes the entire encoded bitplane to the output stream.

Although it has low algorithmic complexity and arithmetic intensity, designing and optimizing parallel encoding algorithms for many-core architectures such as modern GPUs is non-trivial. This is because (1) it is hard to choose a proper parallelization strategy that maximizes both GPU occupancy and memory access efficiency, and (2) fine-grain parallelization can incur large inter-thread communication overhead. Previously, several GPU parallel bitwise processing algorithms have been proposed [37, 38] that potentially can be used to build bitplane encoding. However, they have not been thoroughly optimized and compared in the context of bitplane encoding. In this work, we explore and compare three optimized bitplane encoding designs.

4.1 Bitplane encoding with locality block

Our first design takes inspiration from the ZFP lossy compressor [38], where bitplane coding is one key step in its compression pipeline. In this design, a relatively coarse parallelization strategy is used where each thread encodes a 4^D block consisting of neighboring elements. In our design, we group every contiguous b input element into a locality block and encode their bits into the same bitplane data block. Similar to ZFP, each thread is assigned to encode one locality block when parallelized on GPUs. Figure 2 (a) illustrates a toy example of encoding 4 bitplanes with each locality block containing 4 elements. The figure shows four threads (i.e., $T_0 ... T_3$), with each thread encoding 4 input data and storing the encoded bitplanes independently. This design's main advantage is that it preserves the locality of the input elements in the encoded bitplanes, which can help preserve the bitplane's compressibility. For example, neighboring coefficients tend to have a closer value

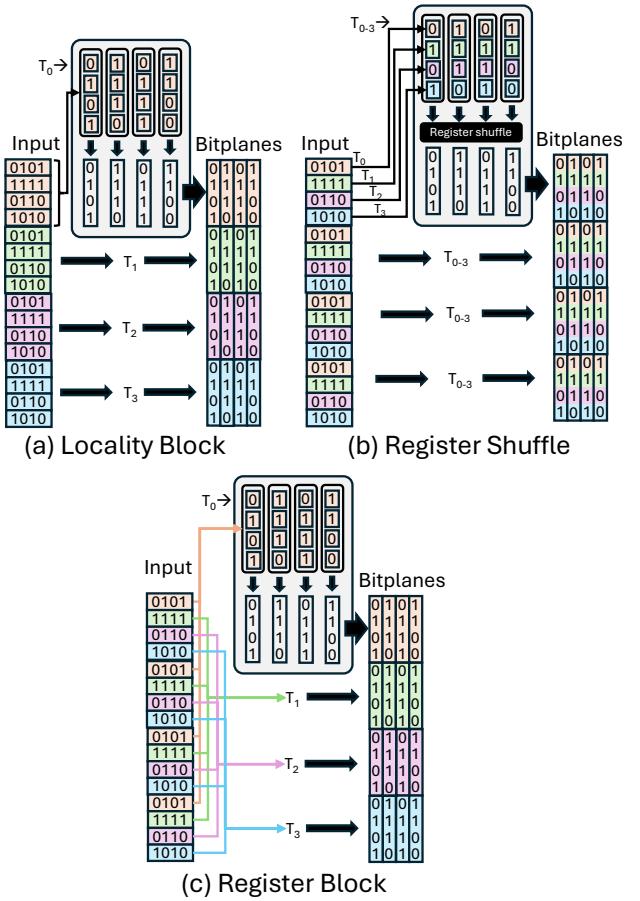


Figure 2: Toy examples to illustrate our three GPU bit-plane encoding strategies for encoding four bitplanes: four threads ($T_0 ... T_3$), each processing four elements, shown as (a) Locality Block, (b) Register Shuffle, and (c) Register Block.

range, and their higher bits tend to be similar, resulting in more contiguous 0 or 1 bits in the encoded bitplanes. This design also provides a fair amount of parallelism when the input size is large, and it does not involve any inter-thread communications. Moreover, the data access pattern for storing encoded bitplanes can be fully coalesced. The main drawback of this design is that the memory load pattern is not coalesced. For smaller block sizes, this issue can be mitigated through the L2 cache. However, smaller blocks tend to reduce the amount of work per thread, which impacts inter-instruction parallelism.

4.2 Bitplane encoding with register shuffling

For smaller input sizes, the locality block design suffers from low parallelism (e.g., parallelism = n/b, where n is the input size and b is the number of elements in a batch that participate in a common shuffling operation). An alternative approach to improving concurrency is having each thread load one element from the input. However, this creates a problem for encoding as threads loading neighboring elements must exchange bits to encode bitplanes. In this design, we extend the register shuffling-based bit-matrix transpose algorithm proposed in [37] to enable bit exchange. After each thread loads

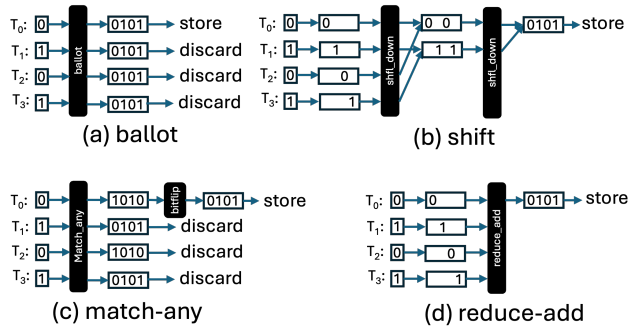


Figure 3: Four encoding designs using register shuffling.

its element, each bit is extracted and shared with others via GPU register shuffling. Our design differs from existing work [37] in that we explore the design with four different register-shuffling instructions in addition to the warp ballot operation used in [37].

As shown in Figure 3, in the ballot approach, each thread sends its bit as a predicate value, and although not needed, all threads get the voting results in the end. In this case, only the thread responsible for storing this particular bitplane keeps the results, and the rest of the threads discard the results. The ballot approach uses the fewest instructions, but it does introduce unnecessary communication as results are broadcast to all threads. The shift approach leverages a classic tree reduction, and only the storing thread obtains the final encoded bitplane. Although it incurs less communication, it requires multiple rounds of shift operations. Match-any behaves similarly to ballot. The only difference is that the result may need an additional bit flip operation if the storing thread holds bit 0. Reduce-add behaves similarly to the shift approach, except the reduction process is repeated with the reduce instruction, which may exploit dedicated hardware optimizations.

4.3 Bitplane encoding with GPU register block

Both the locality block and register shuffling approaches contain certain performance degradation factors. The locality block brings non-fully coalesced memory access patterns, and the register shuffling approach requires extensive inter-thread communication. Our third design explores a fully memory-coalesced and communication-free bitplane encoding approach. As shown in Figure 2 (c), we propose a register block-based approach. Specifically, to avoid any inter-thread communication, we let each thread encode b elements. To achieve fully coalesced memory access, instead of loading contiguous elements, we let each thread load interleaved elements, such as neighboring threads always loading consecutive elements to achieve a coalesced pattern. After loading all data, each thread caches the intermediate data in its own register block and performs encoding independently. Moreover, as all data are in the register blocks, GPUs can fully exploit instruction-level parallelism for better overall throughput. The main drawback of this approach is that bit correlation is not preserved through the encoding process as data are used in an interleaved manner. However, such an impact is limited to each $warp\_size \times b$ region where an interleaved access pattern occurs. So, the compressibility degradation is limited.

5 Lossless Encoding

Lossless encoding is applied to the encoded bitplanes for further size reduction without information loss. It is a crucial step in data refactoring and progressive retrieval framework as its efficiency directly influences the data size and the underlying data movement cost. In HP-MDR, we consider the adaptive use of three core lossless methods —Huffman coding, Run-Length Encoding, and Direct Copy— to achieve the best efficiency. In the following texts, we first introduce the three lossless encoding techniques, followed by our hybrid lossless compression algorithm.

5.1 Lossless encoding technique

Huffman coding (Huffman) [39] is an entropy-based method that assigns shorter binary codes to more frequent symbols. In our framework, it is particularly effective for higher-order bitplanes, where the frequent distribution of symbols is heavily concentrated on a few values, e.g., zeros. We adopt a parallel, GPU-optimized implementation to efficiently compress large-scale bitplane blocks.

Run-Length Encoding (RLE) [40] compresses sequences of repeated values by encoding them as `(value,count)` pairs. This method performs well on lower-order bitplanes, where long runs of zeros frequently occur due to quantization and truncation. Compared to Huffman, RLE achieves lower computational overhead and excels in capturing structured sparsity.

Direct Copy (DC) bypasses compression entirely and stores the bitplane data as is. This strategy is applied when the data size is small or the bitplane is not sufficiently compressible, thus avoiding unnecessary encoding overhead while maintaining high throughput during progressive retrieval.

Huffman and RLE are the primary choices due to their complementary strengths in exploiting different types of redundancy, while DC serves as a lightweight fallback when neither method is effective. These three techniques collectively form the foundation of our hybrid lossless encoding strategy.

5.2 Hybrid lossless compression

To further optimize storage and retrieval performance, we propose an *hybrid lossless compression* that dynamically selects the most appropriate method for each group of bitplanes.

Every m (a configurable parameter) consecutive bitplanes are merged and evaluated for compressibility. We estimate the potential compression ratio of both Huffman and RLE using light-weight predictors, and then choose the most suitable encoding method according to size and compression ratio thresholds. The complete decision logic is presented in Algorithm 2.

The key to the efficiency of our hybrid lossless compression is the accurate yet inexpensive estimate of the compression ratio (CR) for both Huffman encoding and RLE. In the following paragraphs, we dive into the details of each CR estimation method.

Huffman Compression Ratio Estimation. For Huffman encoding, we first compute a frequency histogram of the symbols in the merged bitplane. Based on this histogram, an optimal Huffman tree is generated, assigning shorter code lengths to more frequent symbols. The estimated bit cost is then calculated as the sum of the products of each symbol's frequency and its corresponding code

Algorithm 2: Hybrid Lossless Compression Strategy

Input: Bitplane array B, group size m, size threshold T_s, CR threshold T_{cr}
Output: Compressed bitplane array C

```
N ← total number of bitplanes in B
for i ← 0 → N −1 do   // loop over bitplane groups
    G ← merge B[i..i+m-1];   // merge m bitplanes
    S ← size of G
    if S > T_s then
        r_H ← estimate CR by Huffman on G
        r_R ← estimate CR by RLE on G
        if r_H > T_cr then
            C[i] ← HuffmanEncode(G); // Use Huffman
        else
            if r_R > T_cr then
                C[i] ← RLEEncode(G);      // Use RLE
            else
                C[i] ← DirectCopy(G);      // Use DC
    else
        C[i] ← DirectCopy(G);             // Use DC
    for j ← 1 to m −1 do
        C[i + j] ← EmptyPlaceholder()
```

length. The CR is determined by comparing the original data size with this estimated cost (adjusted for any constant overhead).

RLE Compression Ratio Estimation. For RLE, the estimation is based on an efficient scan of the data to mark the beginnings of runs of consecutive identical symbols. We then compute the total run length by summing these markers. The encoding cost for each run is approximated by considering both the fixed cost to store the symbol and the variable cost to encode the run length. The CR is derived by taking the ratio of the original data size to the estimated total bit cost for encoding all the runs.

If either estimation exceeds the thresholds T_{cr}, the corresponding encoder is selected. Otherwise, Direct Copy is applied. This logic ensures that the encoding effort is applied only when it is beneficial.

CR estimation is performed before actual encoding and incurs minimal overhead. Furthermore, to preserve stream alignment and decoding compatibility, placeholder slots are reserved for nonleading bitplanes in each group.

6 Pipeline Optimization and Implementation

In this section, we introduce our pipeline optimization, which significantly improves GPU utilization to achieve high end-to-end refactoring and reconstruction performance for large-scale data. We then leverage it to construct the first end-to-end data refactoring and progressive retrieval framework on GPUs and enable guaranteed error control for derivable Quantities of Interest (QoIs).

6.1 Pipeline optimization

When refactoring or reconstructing a large-scale dataset, the entire data may not fit entirely on GPU memory. In this case, data must be decomposed into subdomains and processed sequentially. In addition, when processing multiple variables, each variable also needs to be refactored and reconstructed sequentially. As noted in [36], frequent data copy in and out of GPU devices can incur large overhead for data reduction pipelines. In this work, we extend the optimization of the pipeline in [36] to the refactoring and reconstruction pipeline.

To make pipeline optimization portable across GPU architectures, we use the Host-Device Execution Model (HDEM) [36], to aid design. In this machine model, one GPU device is equipped with two Direct Memory Access (DMA) engines, each of which can work independently for asynchronous memory copy. They are used to copy data between the application buffer, I/O buffer, and refactoring/reconstruction buffer. The device also has a compute engine to support the concurrent execution of refactoring/reconstruction kernels during data copy operations.

6.1.1 Data Refactoring. Figure 4 (a) shows our optimized refactoring pipeline. The refactoring process is pipelined among three queues (1-3). Green boxes represent CPU-to-GPU copy tasks. Red boxes represent GPU-to-CPU copy tasks. Blue boxes are pure computing tasks. Yellow represents mixed memory copy and computing tasks. According to our restrictions, no two tasks with the same color can be executed simultaneously, and a yellow task cannot be executed concurrently with any other tasks. Three input/output buffers are: I_1/O_1, I_2/O_2, and I_3/O_3. We assume serialization and deserialization are needed for embedding and extracting metadata before and after computation, which also relies on memory copies. Also, lossless compression and decompression contain computation and data copies between CPU and GPU due to its internal serialization and deserialization process, which are color-coded in yellow. To ensure refactoring correctness, we enforce execution ordering with solid arrows. Additionally, to hide the memory copy latency, we prefetch the next input while refactoring the current subdomain. To avoid delaying the current execution, prefetch needs to be done during multi-level decompositions and bitplane encoding and finished before lossless compression, so we enforce additional dependencies between $I \rightarrow Z$. Also, the prefetching should be done as soon as its DMA becomes available (after serialization), so we add another dependency between $S \rightarrow I$. Finally, to hide the latency of copying refactoring data back to the CPU, we let it overlap with multi-level decompositions, bitplane encoding, and prefetch.

6.1.2 Progressive Retrieval. Figure 4 (b) shows our optimized reconstruction pipeline. Similar to the refactoring pipeline, we add additional dependencies to maximize the latency hiding while minimizing potential delay to the original reconstruction pipeline. To perfect refactored input data, we delay its initiation until we are done with deserialization and lossless decompression ($X \rightarrow I$). This is because having an early data prefetch can delay the current pipeline due to the conflict used with CPU-to-GPU DMA. Also, similarly, the GPU-to-CPU memory copy for storing reconstructed data of the last iteration can also delay the current process, so we delay it until we are about to start bitplane decoding and multi-level recomposition ($X \rightarrow O$).

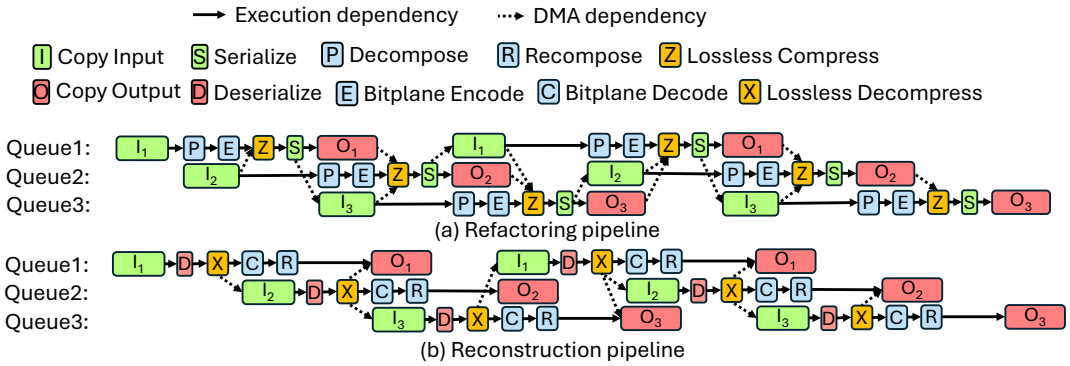


Figure 4: Optimized refactoring (a) and reconstruction (b) pipelines expressed as DAGs over three queues under the Host–Device Execution Model with dual DMA engines. Tasks are color-coded: host-to-device copy (green), device-to-host copy (red), pure compute (blue), and mixed copy-compute (yellow). In our settings, no two tasks of the same color may execute concurrently, and yellow tasks cannot overlap with any other. Additional DMA and execution dependencies overlap CPU–GPU transfers with computation to hide data-transfer latency.

6.2 Progressive retrieval with QoI error control

We further leverage pipeline optimization to efficiently enable multivariate QoI error control during progressive retrieval in HP-MDR based on prior work [11]. The algorithm is presented in Algorithm 3, with orange statements representing memory operations and blue statements indicating computing operations. In particular, the estimated QoI error τ' is initialized as infinity (line 1), and it will be updated iteratively until its value is less than the target QoI error tolerance τ (lines 2-10). In each iteration, we first fetch the necessary bitplanes and use them to recompose data to their estimated data error bounds in lines 3-7 (initialized as the relative value of τ over its maximal value multiplied by the value range of data according to [11]). We then implement the GPU kernels to estimate the supremum of QoI errors and use it to update τ' based on [11], which consists of the error estimation for several families of base QoIs and some specific operations (line 8). Since the target QoIs are computed point-wise with a constant number of operations, this step could be very fast with GPUs. If τ' is greater than τ, we use dedicated methods (to be detailed below) to estimate the next data error bounds to guide the retrieval procedure based on the information we have (lines 9-10). Data transfer and recomposition (lines 3-7) are the most time-consuming parts in a single iteration, and our pipeline optimization could effectively overlap them with the proposed pipeline optimization to achieve high throughput.

We then introduce the methods for estimating the next data error bounds, which are essential to both efficiency and throughput of progressive retrieval with QoI error control. Generally speaking, a small number of retrieved bitplanes represents high efficiency (since less data is retrieved), and fewer iterations indicate high throughput (due to less computation). We explore three methods to perform the estimation in HP-MDR, as detailed below.

CPU Porting (CP). We directly port error estimation method from the CPU implementation in [11]. In particular, the algorithm first identifies the data point with the maximal estimated QoI error (on GPU), and then iteratively decays the data error bounds and re-evaluates the QoI error for that single data point until the target QoI error tolerance is met (on CPU after transferring necessary information back). This algorithm usually converges to a set of sufficient data error bounds quickly, but it may suffer from over-preservation because the estimation is not accurate due to the use of stale data. This generally leads to redundancy in data retrieval and, thus, suboptimal efficiency.

Algorithm 3: Progressive retrieval with QoI error control

Input: QoI error tolerance τ, initial data error bounds $\{\epsilon_i\}$, encoded bitplanes for all variables $\{\{S_j\}^{(i)}\}$, QoI Q
Output: Decompressed data with QoI error less than τ

```
// Colors encode memory and computing operations
// Initialize current QoI error
1  τ' ← ∞
2  while τ' > τ do
       // Fetch the first variable
3      copy_to_device({S_j}^(0), ε_0)
4      for i ← 0 to n_v − 2 do
           // Fetch the next variable
5          copy_to_device({S_j}^(i+1), ε_{i+1})
           // recompose current variable
6          v_i ← recompose(v_i, {S_j}^(i))
       // recompose the last variables
7      v_{n_v−1} ← recompose(v_{n_v−1}, {S_j}^(n_v−1))
       // Estimate QoI errors
8      τ' ← estimate_QoI_error({v_j}, {ε_j}, Q)
9      if τ' > τ then
           // Estimate error bounds for all variables
10         {ε_j} ←estimate_next_eb({v_j}, {ε_j}, τ, τ', Q)
11 return {v_j}
```

Minimal Augmentation (MA). To address the over-preservation issue in CP, we propose minimal augmentation to obtain a near-optimal retrieval efficiency by fetching data with fine granularity. In particular, we directly fetch one more merged bitplane for each variable and update the corresponding data error bounds accordingly.

Since this method explores the possible combinations of data error bounds at very fine granularity, it could terminate the procedure promptly when sufficient bitplanes are retrieved, leading to high retrieval efficiency. Nonetheless, it may cost a number of iterations to complete, which negatively impacts the throughput.

Minimal Augmentation with Proportional Estimation (MAPE). We further propose to couple minimal augmentation with proportional estimation to reduce the number of iterations needed. Given maximal estimated QoI error τ' and target QoI error tolerance τ, we first check their proportion $p = \tau'/\tau$ to see if they are close enough. If p is larger than a threshold c, we assume the same proportional relationship on data error bounds and estimate the next data error bound ϵ_{i+1} as ϵ_i/p, where ϵ_i is the current data error bound; otherwise, we switch to minimal augmentation as the current data representations are very close to the target ones. As such, MAPE reduces the number of iterations for convergence while enjoying the benefits of MA, leading to a good tradeoff between retrieval efficiency and throughput. Note that we use proportional estimation instead of CP in MAPE because CP easily leads to over-preservation, which requires a relatively large c to make the switch. This, in turn, will result in a high number of iterations in some instances.

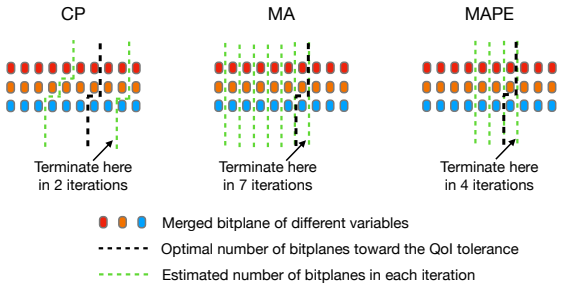


Figure 5: Illustration of the three error bound estimation methods: CPU Porting (CP), Minimal Augmentation (MA), and Minimal Augmentation with Proportional Estimation (MAPE). A lower number of bitplanes indicates a smaller retrieval size, and fewer iterations indicate higher throughput.

We illustrate the three methods using an example in Figure 5, with a black dashed line indicating the optimal number of bitplanes needed for each variable. As shown in the figure, CP can quickly identify the feasible solution, but it may retrieve more bitplanes than needed. MA yields a near-optimal solution but takes a long time to converge. MAPE has medium efficiency and throughput, which usually provides the best trade-off.

7 Evaluation

7.1 Experimental setup

7.1.1 Experimental environment and datasets. We conduct experimental evaluations on two systems: Frontier and Talapas. Frontier is a leadership-class exascale supercomputer at Oak Ridge Leadership Computing Facility (OLCF) [2]. It consists of a total of 9,408 computing nodes. Each compute node has 8 AMD Instinct MI250X GPUs with 64 GB of memory on each GPU and one 64-core AMD EPYC CPU with 512 GB of memory. Talapas is a heterogeneous cluster system, where each of its GPU computing nodes is equipped with 4 NVIDIA H100 GPUs with 80 GB memory and two 24-core Intel Xeon CPUs with 1,024 GB memory. We evaluate on five real-world scientific datasets from [41][42][43], summarized in Table 1. For the scalability test, we broadcast the data to different GPUs for evaluation with diverse scales, which yields variable data sizes.

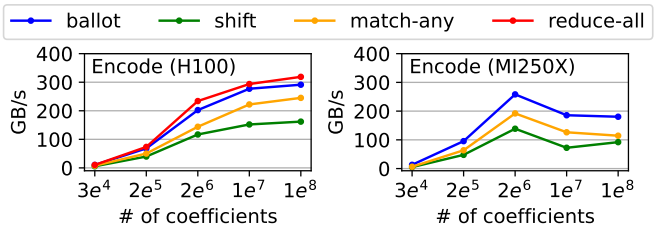


Figure 6: Bitplane encoding throughput with four types of register shuffle instruction. Evaluated with NYX dataset.

Table 1: Datasets used for evaluation

Dataset	Variable	Dimensions	Type	Size
NYX	velocity_x/y/z, temperature	512 × 512 × 512	FP32	2 GB
LETKF	PRES	98 × 1200 × 1200	FP32	538 MB
Miranda	velocityz	256 × 384 × 384	FP64	288 MB
Hurricane ISABEL	Pf48	100 × 500 × 500	FP32	95 MB
JHTDB	velocityx/y/z	1024 × 2048 × 2048	FP32	48 GB

7.1.2 Baselines. We compare HP-MDR with two baselines: **MDR Baseline:** As HP-MDR builds on MDR [9], we include it as a direct baseline. MDR performs multilevel hierarchical decomposition for progressive reconstruction. **Multi-Component Baselines:** We also evaluate the progressive framework [10], which progressively compresses residual components using different lossy compressors. Selected backends include: ZFP-GPU [38] (fixed-rate), MGARD-GPU [34], SZ3-OMP [22, 26], ZFP-OMP [19] (fixed-accuracy).

7.2 Data refactoring and retrieval

7.2.1 Bitplane Encoding. We first compare bitplane encoding with different register shuffling approaches. We show the performance of encoding 32-bit NYX data into 32 bitplanes and decoding all 32 bitplanes back to 32-bit data with various input sizes. Please note the performance of encoder does not depends on the data content, so it will perform similarly on other datasets. As shown in Figure 6, we evaluate all four register shuffling instructions on H100 and three register shuffling approaches on MI250X since the reduce-all instruction is not supported on AMD GPUs. On H100, reduce-all instruction provides the best encoding throughput. Specifically, it improves the encoding performance by up to 15% compared with start-of-the-art design [37]. This could be due to the existence of dedicated hardware that supports fast reduction. On MI250X, the ballot outperforms other approaches since it requires the least amount of instructions. However, we do observe performance degradation as input size increases, which does not exist on H100. This could be due to the architectural difference that causes communication contention to have a more negative impact on AMD GPUs.

Figure 7 shows the throughput comparison of three bitplane encoding parallelization designs on both H100 and MI250X. We show the performance of encoding/decoding 32-bit data with 32 bitplanes of different input sizes. For the register shuffling encoder, we use the best-performing instruction throughout the rest of the evaluations. The evaluation results show that the locality block

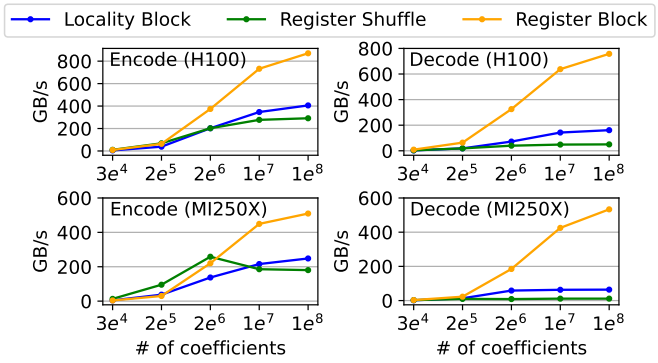
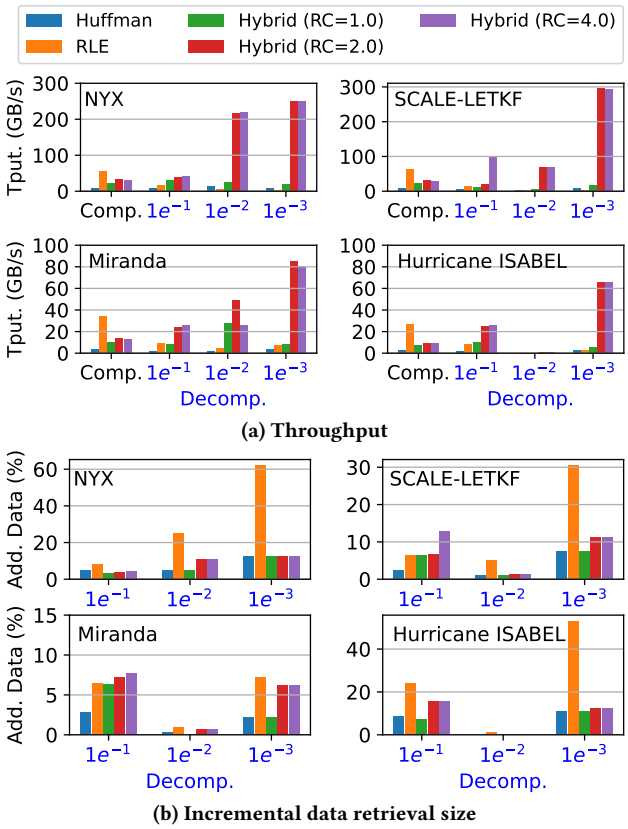


Figure 7: Bitplane encoding throughput of three parallelization designs. Evaluated with NYX dataset.

outperforms register shuffling by 1.4× for encoding and 3.2× for decoding on H100 and 1.4× for encoding and 6.6× for decoding on MI250X. Additionally, the register block approach outperforms the locality block by 2.1× for encoding and 4.7× for decoding on H100 and 2.1× for encoding and 8.3× for decoding on MI250X. The register block provides the highest throughput for both encoding and decoding on both GPUs due to its fully coalesced and communication-free computation for both encoding and decoding. We use this encoding approach for the rest of the evaluations.

7.2.2 Lossless Encoding. Figure 8 compares the performance and compressibility of different lossless compression strategies. Specifically, we compare (1) applying Huffman to all bitplanes (Huffman); (2) applying RLE to all bitplanes (RLE); (3) applying a hybrid approach of Huffman, RLE, and direct copy with different compression ratio thresholds (Hybrid-rc). We compare the throughput of the end-to-end lossless (de)compression stage by showing throughput relative to the original data (for compression) and decompressed bitplane size (for decompression). We also show the incremental data retrieval size when progressively reconstructing to a certain error tolerance. A lower retrieval size indicates less I/O cost for progressive data reconstruction. For the same variable with the same error tolerance, the difference in the retrieval size is only due to the lossless compression's compressibility, since the number of bitplanes needed is the same. As shown in the result, comparing Huffman with others, Huffman brings the smallest retrieval sizes. However, Huffman has the lower throughput: 5.7 GB/s for compression and 4.8 GB/s for decompression on average. RLE, on the other hand, brings on average 44.4 GB/s for compression and 6.4 GB/s for decompression with 270% additional data needed for retrieval on average compared with Huffman. The hybrid approach brings 15.5 GB/s, 20.8 GB/s, and 22.4 GB/s average compression throughput and 14.1 GB/s, 94.9 GB/s, and 99.8 GB/s average decompression throughput with 8%, 70%, and 93% additional data need compared with Huffman respectively for rc = 1.0, 2.0, and 4.0.

We also show how tunable parameters in our framework affect the performance and compression in Figure 8 and Figure 9. The group size (m) directly impacts bitplane granularity. When we increase m to 8, throughput reaches 230 GB/s but brings higher data overhead. Smaller values like m=2 enable finer progressive access at reduced performance. The size threshold (Ts) determines compression candidates: setting Ts=1e9 can aggressively skip less compressible data for maximum throughput, while setting Ts=1e3 can compress more bitplanes to reduce storage overhead. The compression ratio threshold (RC) plays a similar role to Ts. Setting RC=1.0 provides conservative compression with moderate performance gains, setting RC=2.0 gains a balance between compression efficiency and throughput, while setting RC=4.0 delivers the highest performance with slightly more storage overhead. These parameters work together to address different application needs. Based on our analysis, we choose default settings of m=4, Ts=1e6, and RC=2.0 for subsequent experiments, as they provide a balanced trade-off between compression performance and storage efficiency.



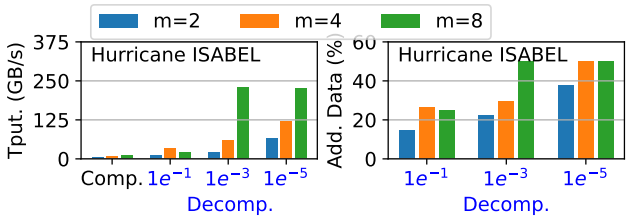
(a) Throughput

(b) Incremental data retrieval size

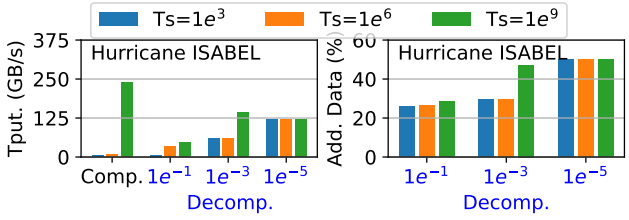
Figure 8: Comparing performance and compressibility of different lossless compression approaches.

7.2.3 End-to-end Data Refactoring and Retrieval. Next, we conduct end-to-end refactoring and reconstruction evaluation. Figure 10 compares the end-to-end throughput with and without pipeline optimization using four datasets. On H100, the pipeline optimization accelerates refactoring by 1.43× and reconstruction by 1.83× on average. On MI250X, the pipeline optimization accelerates refactoring by 1.41× and reconstruction by 1.43× on average.

Furthermore, we conduct multi-node and multi-GPU scalability evaluations. We evaluated the end-to-end performance of refactoring and reconstruction in weak-scaling settings. For H100, we scale up to 4 GPUs and up to 4 nodes. For MI250X, we scale up to 8 GPUs. Figure 11 shows that our design achieves near-linear scalability, reaching 90% and 89% on average of the ideal speed-up on H100 and MI250X, respectively.



(a) Group size(m) parameter analysis



(b) Size threshold(Ts) parameter analysis

Figure 9: Impact of parameter configuration on compression performance and data overhead.

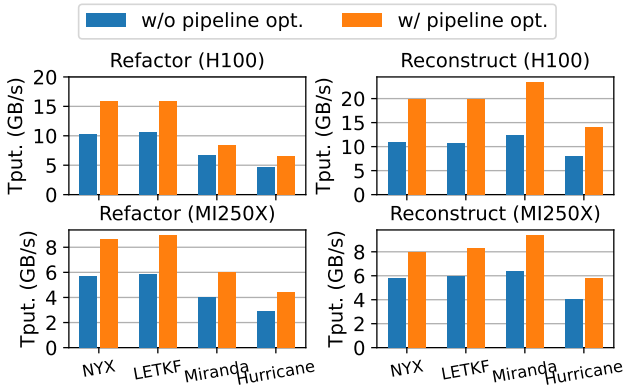


Figure 10: End-to-End throughput comparison with and without pipeline optimization.

Finally, we compare the end-to-end throughput and retrieval efficiency of our proposed HP-MDR with all five baselines. CPU tests are parallelized using OpenMP with 32 threads, while GPU tests are executed on NVIDIA H100. As shown in Figure 12, we consistently outperform all baselines across four datasets and a wide range of error tolerances from 10^{-1} to 10^{-6} on throughput. For example, we achieve an average throughput of 11.9 GB/s and 12.5 GB/s for refactoring and reconstruction, respectively, while the best baseline (M-MGARD-GPU) only obtains throughput of 0.87 GB/s and 1.98 GB/s, meaning that HP-MDR delivers average 13.68× and 6.31× speedups for refactoring and reconstruction.

While HP-MDR does not yield the smallest retrieval size, it remains competitive. For instance, when reconstructing data for the Miranda dataset, we achieve an average additional retrieval ratio of 4.36%, which is higher than the best-performing framework (2.19%) but still better than the overall average across all evaluated baselines (5.55%). This demonstrates a strong trade-off between retrieval cost and overall performance, with HP-MDR outperforming most existing approaches.

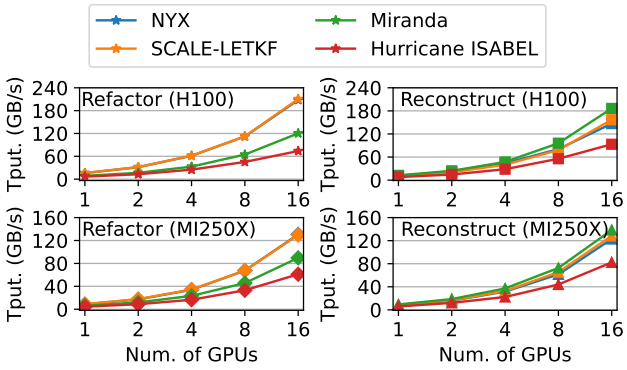


Figure 11: Scalability on multi-node multi-GPU systems.

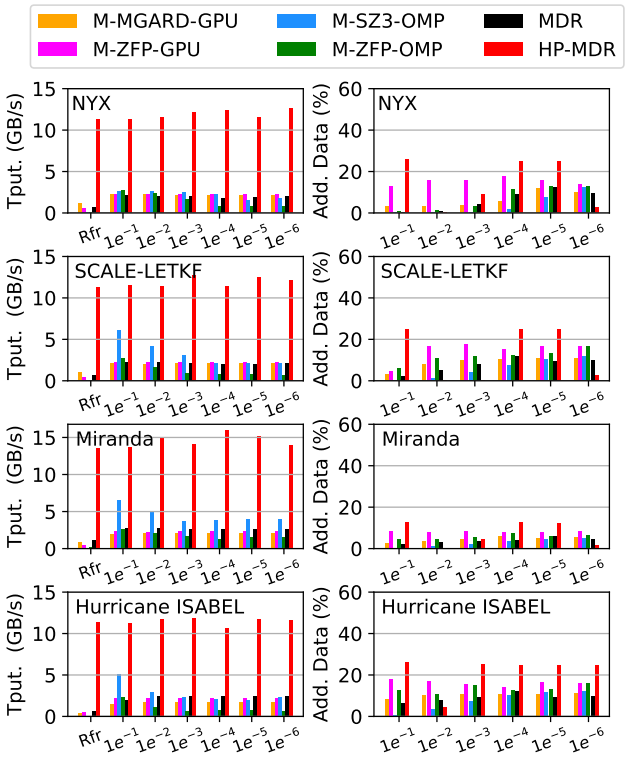


Figure 12: Comparing HP-MDR with state-of-the-art progressive data retrieve frameworks.

7.3 Progressive retrieval with QoI error control

All the evaluations in this subsection are performed on the Frontier supercomputer with MI250X GPUs. Without loss of generality, we use $V_{total} = \sqrt{V_x^2 + V_y^2 + V_z^2}$ as the target QoI.

7.3.1 Single-GPU Evaluation. We perform single-GPU evaluation using velocity fields from NYX (1.5 GB) and mini-JHTDB (6 GB, a cropped region from JHTDB to fit in a single GPU). In particular, we compare the efficiency and throughput of all three error bound (EB) estimation methods.

Retreival efficiency and throughput. We present the retrieval efficiency on the two datasets in Table 2 and 3, respectively, and we report the corresponding throughput in Figure 13. Overall, it

Table 2: Bitrate of EB estimation methods on NYX

Method	1E-1	5E-1	1E-2	5E-2	1E-3	5E-3	1E-4	5E-4	1E-5	5E-5
CP	6.89	6.89	**6.89**	7.49	12.57	14.90	14.90	15.20	22.90	22.90
MA	**4.23**	**5.99**	6.90	**6.90**	7.86	14.90	14.90	**14.90**	22.90	22.90
MAPE(c=2)	6.03	6.03	**6.89**	7.20	**7.82**	**12.57**	14.90	15.49	22.90	22.90
MAPE(c=10)	**4.23**	6.89	**6.89**	**6.90**	**7.82**	14.90	14.90	**14.90**	22.90	22.90

Table 3: Bitrate of EB estimation methods on mini-JHTDB

Method	1E-1	5E-1	1E-2	5E-2	1E-3	5E-3	1E-4	5E-4	1E-5	5E-5
CP	10.42	10.42	10.43	10.43	**11.31**	18.43	18.43	18.43	26.43	26.43
MA	**5.76**	**5.76**	10.43	10.43	**11.31**	18.43	18.43	18.43	26.43	26.43
MAPE(c=2)	6.82	10.42	**10.42**	10.43	11.38	18.76	18.43	18.43	26.43	26.43
MAPE(c=10)	6.82	8.42	**10.42**	10.43	11.38	**16.09**	18.43	18.43	26.43	26.43

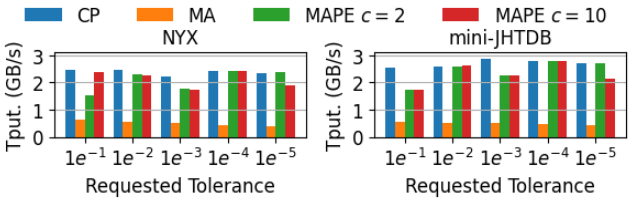


Figure 13: Overall kernel throughput on the NYX and mini-JHTDB dataset.

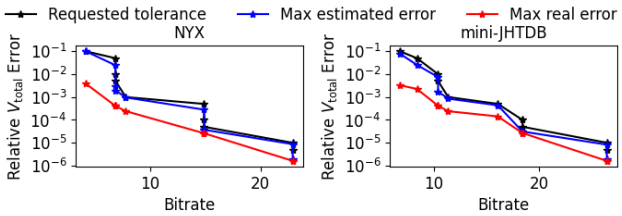


Figure 14: Requested V_{total} tolerance, max estimated V_{total} errors, and max actual V_{total} errors during progressive retrieval towards V_{total} in the NYX and mini-JHTDB dataset.

has been observed that Minimal Augmentation (MA) achieves the best bitrates under the majority of requested tolerances with the lowest throughput, while the CPU Porting (CP) achieves the highest throughput under most of the requested tolerances with the worst bitrates. Minimal Augmentation with Proportional Estimation (MAPE) with threshold $c = 10$ makes a good tradeoff between ensuring a suboptimal bitrate and maintaining a relatively high throughput, so we use it for the following validation of QoI error control and multi-GPU evaluations.

Guaranteed QoI error control. We validate the QoI error control by presenting and comparing three values: (1) requested tolerance τ; (2) max estimated error computed by HP-MDR; and (3) max real error of the provided data. As illustrated in Figure 14, the max real error is always smaller than the max estimated error, which is close to but strictly smaller than the requested tolerance on both datasets. This shows that HP-MDR can faithfully enforce the QoI error control during progressive retrieval.

7.3.2 Multi-GPU Evaluation. We further evaluate the throughput and end-to-end data retrieval performance of HP-MDR on an entire Frontier node (8 GPUs), and compare it with multicore CPUs in the same node (64 cores) using the JHTDB dataset (48 GB). Under this setting, each CPU processes 0.75 GB of data, while each GPU handles 6 GB. We report both the overall kernel throughput (which only includes computational time) and the end-to-end data retrieval time (which measures the time from data reading to the completion of data reconstruction) in Figure 15.

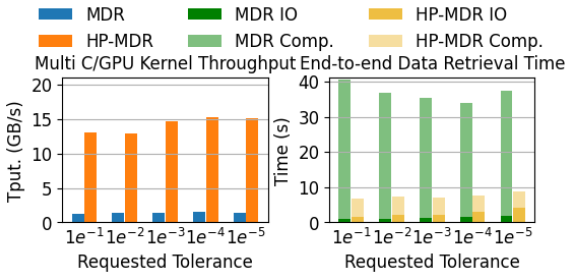


Figure 15: Multi CPU/GPU kernel throughput and end-to-end data retrieval time on the JHTDB dataset.

According to the figure, HP-MDR exhibits over 11.22× speedup in kernel throughput, although the end-to-end performance gain is reduced to 6.04×. This is caused by two reasons: (1) I/O overhead is more significant in HP-MDR because it has a higher bit rate than MDR; and (2) there is some particular overhead in GPUs (e.g., memory allocation) for end-to-end evaluation.

8 Conclusion

In this paper, we presented HP-MDR, a high-performance and portable framework designed to accelerate data refactoring and progressive retrieval on heterogeneous systems with advanced GPUs. By thoroughly optimizing the bitplane encoding and lossless compression stages, we addressed key performance bottlenecks in current progressive methods. Our register block-based encoding and hybrid lossless compression techniques significantly improve throughput while maintaining data fidelity and portability. We further enhanced end-to-end efficiency through a pipeline optimization strategy that overlaps computation and memory operations. By integrating a GPU-accelerated multilevel decomposition algorithm and extending it with GPU-optimized QoI error control, HP-MDR enables precise and efficient data retrieval tailored to the needs of scientific analytics. Extensive evaluations on real-world datasets across multiple GPU architectures demonstrated that HP-MDR delivers substantial speedups over existing frameworks, achieving up to 6.6× improvement in throughput and competitive retrieval efficiency. In the context of progressive retrieval under error constraints in derived Quantities of Interest, HP-MDR leads to 11.22× throughput for recomposing required data representations and 6.04× performance for end-to-end retrieval, when compared with state-of-the-art solutions. In the future, we will optimize HP-MDR with large-scale I/O systems and adapt it to various data analysis tasks using different data organization strategies.

Acknowledgments

The research is supported in part by the U.S. Department of Energy (DOE) RAPIDS-2 SciDAC and Sirius2 projects under contract number DE-AC05-00OR22725, and National Science Foundation (NSF) under Grant OAC-2311756, OAC-2313122, OAC-2442627, OAC-2311757, and OAC-2144403. This research used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility.

References

[1] [n. d.]. Aurora exscale system. https://www.alcf.anl.gov/support-center/aurora.
[2] [n. d.]. Frontier exscale supercomputer. https://www.olcf.ornl.gov/frontier.
[3] [n. d.]. El Captain exscale system. https://asc.llnl.gov/exascale/el-capitan.
[4] Allison H Baker, Dorit M Hammerling, Sheri A Mickelson, Haiying Xu, Martin B Stolpe, Phillipe Naveau, Ben Sanderson, Imme Ebert-Uphoff, Savini Samarasinghe, Francesco De Simone, et al. 2016. Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development* 9, 12 (2016), 4381–4403.
[5] Jesus Pulido, Zarija Lukic, Paul Thorman, Caixia Zheng, James Ahrens, and Bernd Hamann. 2019. Data reduction using lossy compression for cosmology and astrophysics workflows. In *Journal of Physics: Conference Series*, Vol. 1290. IOP Publishing, 012008.
[6] Franck Cappello, Sheng Di, and Ali Murat Gok. 2020. Fulfilling the promises of lossy compression for scientific applications. In *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference, SMC 2020, Oak Ridge, TN, USA, August 26-28, 2020, Revised Selected Papers 17*. Springer, 99–116.
[7] Robert Underwood, Jon C Calhoun, Sheng Di, and Franck Cappello. 2024. Understanding The Effectiveness of Lossy Compression in Machine Learning Training Sets. *arXiv preprint arXiv:2403.15953* (2024).
[8] Shaomeng Li, Stanislaw Jaroszynski, Scott Pearse, Leigh Orf, and John Clyne. 2019. Vapor: A visualization package tailored to analyze simulation data in earth system science. *Atmosphere* 10, 9 (2019), 488.
[9] Xin Liang, Qian Gong, Jieyang Chen, Ben Whitney, Lipeng Wan, Qing Liu, David Pugmire, Rick Archibald, Norbert Podhorszki, and Scott Klasky. 2021. 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*. 1–13.
[10] Victor AP Magri and Peter Lindstrom. 2023. A general framework for progressive data compression and retrieval. *IEEE Transactions on Visualization and Computer Graphics* 30, 1 (2023), 1358–1368.
[11] Xuan Wu, Qian Gong, Jieyang Chen, Qing Liu, Norbert Podhorszki, Xin Liang, and Scott Klasky. 2024. 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, 1–16.
[12] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE transactions on consumer electronics* 38, 1 (1992), xviii–xxxiv.
[13] Majid Rabbani. 2002. JPEG2000: Image compression fundamentals, standards and practice. *Journal of Electronic Imaging* 11, 2 (2002), 286.
[14] [n. d.]. Summit exscale system. https://www.olcf.ornl.gov/summit.
[15] Peter Deutsch. 1996. GZIP file format specification version 4.3. (1996).
[16] Yann Collet. [n. d.]. Zstandard - Real-time data compression algorithm. http://facebook.github.io/zstd/. Online.
[17] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics* 12, 5 (2006), 1245–1250.
[18] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Seung-Hoe Ku, Choong-Seock Chang, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. 2013. ISABELA for effective in situ compression of scientific data. *Concurrency and Computation: Practice and Experience* 25, 4 (2013), 524–540.
[19] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.
[20] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1129–1139.
[21] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 438–447.
[22] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1643–1654.
[23] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2019. Multilevel techniques for compression and reduction of scientific data—the multivariate case. *SIAM Journal on Scientific Computing* 41, 2 (2019), A1278–A1303.
[24] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2020. Multilevel techniques for compression and reduction of scientific data—The unstructured case. *SIAM Journal on Scientific Computing* 42, 2 (2020), A1402–A1427.
[25] Xin Liang, Ben Whitney, Jieyang Chen, Lipeng Wan, Qing Liu, Dingwen Tao, James Kress, David Pugmire, Matthew Wolf, Norbert Podhorszki, et al. 2021. Mgard+: Optimizing multilevel methods for error-bounded scientific data reduction. *IEEE Trans. Comput.* 71, 7 (2021), 1522–1536.
[26] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* 9, 2 (2022), 485–498.
[27] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, Vol. 22. Wiley Online Library, 343–348.
[28] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.
[29] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2018. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science* 19, 5 (2018), 65–76.
[30] Shaomeng Li, Peter Lindstrom, and John Clyne. 2023. Lossy Scientific Data Compression With SPERR. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1007–1017. doi:10.1109/IPDPS54959.2023.00104
[31] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2019. TTHRESH: Tensor compression for multidimensional visual data. *IEEE transactions on visualization and computer graphics* 26, 9 (2019), 2891–2903.
[32] Jinyang Liu, Sheng Di, Kai Zhao, Sian Jin, Dingwen Tao, Xin Liang, Zizhong Chen, and Franck Cappello. 2021. Exploring Autoencoder-Based Error-Bounded Compression for Scientific Data. *arXiv preprint arXiv:2105.11730* (2021).
[33] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. 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*. 3–15.
[34] Jieyang Chen, Lipeng Wan, Xin Liang, Ben Whitney, Qing Liu, David Pugmire, Nicholas Thompson, Jong Youl Choi, Matthew Wolf, Todd Munson, et al. 2021. Accelerating multigrid-based hierarchical scientific data refactoring on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 859–868.
[35] Yafan Huang, Sheng Di, Guanpeng Li, and Franck Cappello. 2024. cuSZp2: A GPU Lossy Compressor with Extreme Throughput and Optimized Compression Ratio. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.
[36] Jieyang Chen, Qian Gong, Yanliang Li, Xin Liang, Lipeng Wan, Qing Liu, Norbert Podhorszki, and Scott Klasky. 2025. HPDR: High-Performance Portable Scientific Data Reduction Framework. *arXiv preprint arXiv:2503.06322* (2025).
[37] Ang Li, Tong Geng, Tianqi Wang, Martin Herbordt, Shuaiwen Leon Song, and Kevin Barker. 2019. BSTC: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–30.
[38] Peter Lindstrom, Jeffrey Hittinger, James Diffenderfer, Alyson Fox, Daniel Osei-Kuffuor, and Jeffrey Banks. 2025. ZFP: A compressed array representation for numerical computations. *The International Journal of High Performance Computing Applications* 39, 1 (2025), 104–122.
[39] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. Revisiting huffman coding: Toward extreme performance on modern gpu architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 881–891.
[40] Ana Balevic. 2009. Fine-Grain Parallelization of Entropy Coding on GPGPUs.
[41] [n. d.]. SDRBench. https://sdrbench.github.io.
[42] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific data reduction benchmark for lossy compressors. In *2020 IEEE international conference on big data (Big Data)*. IEEE, 2716–2724.
[43] PK Yeung, DA Donzis, and KR Sreenivasan. 2012. Dissipation, enstrophy and pressure statistics in turbulence simulations at high Reynolds numbers. *Journal of Fluid Mechanics* 700 (2012), 5–15.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

In this work, we propose a high-performance and portable framework – HP-MDR, implementing progressive data reconstruction methods on exascale systems with advanced GPUs. In particular, our contributions are as follows.

- **C_1** Highly optimized bitplane encoder that accelerates encoding for modern GPUs with portability across architecture types.
- **C_2** Hybrid lossless compression that greatly accelerates bitplane compression throughput with comparable compression ratio with the state-of-the-art lossless compressor on GPUs.
- **C_3** End-to-end data refactoring and reconstruction on GPU with pipeline optimization that combines optimized data decomposition, encoding, and compression techniques.
- **C_4** Progressive retrieval pipeline with guaranteed error control under common Quantities of Interest (QoIs) on GPUs that greatly accelerates end-to-end data retrieval.

A.2 Computational Artifacts

A_1 https://github.com/JieyangChen7/MGARD/tree/hp-mdr (DOI: https://doi.org/10.5281/zenodo.16938476)
https://github.com/MasterVChicken/Multi-Component-Compression-Framework.git (DOI: https://doi.org/10.5281/zenodo.16938521)
https://github.com/MasterVChicken/OpenMP-Optimization-for-MDR_CPU.git (DOI: https://doi.org/10.5281/zenodo.16938519)
https://github.com/Linus-Li-1037/qoi-control (DOI: https://doi.org/10.5281/zenodo.16938512)

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3, C_4	Figure 6-14 Table 2-3

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

Our HP-MDR is in artifact A_1. It contains optimized bitplane encoder, hybrid lossless compressor, and end-to-end data refactoring and reconstruction pipelines and enhanced data reconstruction pipeline with QoI error control.

Expected Results

- For C_1, the supporting experiment is evaluating the bitplane encoding/decoding performance on various types of GPUs as shown in Figure 6-7 of the submitted paper draft. The expected speedups various on different GPU models. On NVIDIA H100 we observed average 2.1× speedup for encoding and average 4.7× speedup for decoding. On AMD MI250X, we observed average 2.1× speedup for encoding and average 8.3× speedup for decoding.
- For C_2, the supporting experiment is evaluating the throughput and data retrieval size (compression ratio) of our hybrid lossless compressor as shown on in Figure 8 of the submitted paper draft. The expected throughput various on different GPU models and retrieval size various for different datasets. On NVIDIA H100, testing four datasets we observed 8× throughput improvement against Huffman with comparable retrieval size to the state-of-the-art GPU lossless compression baseline.
- For C_3, the supporting experiments is evaluating the end-to-end performance and scalability of the data refactoring and reconstruction pipelines as shown in Figure 9-11 of the submitted paper draft. The expected speedups various on different GPU models and datasets. On NVIDIA H100, testing four dataset, we observed 6.6× speedup compared with the state-of-the-art works. Furthermore, on the same system and settings, we measured 1.4× and 1.8× speedups using our pipeline optimization. On multi-GPU setting, we measured aggregated throughput achieving 90% of theoretical throughput with idea speedups.
- For C_4, the supporting experiment is evaluating the performance of progressive retrieval pipeline with guaranteed error control under common QoIs on GPUs as shown in Table 2-3 and Figure 12-14 of the submitted paper draft. On an entire Frontier node (64 cores, and 8 AMD MI250X GPUs), we observed 11.22× and 6.04× speedups for recomposing required data representations and end-to-end retrieval (The performance results may vary when using fewer GPUs).

Expected Reproduction Time (in Minutes)

The total expected time needed is 140 min, and they are distributed as follows:

- Building artifact with dependencies: 20 min
- Testing data preparation (download): 20 min
- Evaluation for C_1: 10 min
- Evaluation for C_2: 10 min
- Evaluation for C_3: 30 min
- Evaluation for C_4: 50 min

Artifact Setup (incl. Inputs)

Hardware.

- NVIDIA GPUs compatible with CUDA 12 or AMD GPUs compatible with ROCm 6
- At least 100 GB of available disk space for datasets, software, and experimental results
- GPU with at least 40 GB memory

Software.

- CMake 3.9+

- NVCOMP v2.2.0 (only if on NVIDIA GPUs): https://github.com/NVIDIA/nvcomp.git
- ZSTD 1.5.0 https://github.com/facebook/zstd.git
- Protobuf 3.19.4 https://github.com/protocolbuffers/protobuf.git
- CUDA 12+ (only if on NVIDIA GPUs) https://developer.nvidia.com/cuda-downloads
- ROCm 6+ (only if on AMD GPUs) https://github.com/ROCm/ROCm
- GCC 8.5 - 11.x
- MPICH 4+ or Open MPI 5+
- OpenMP 2.0+
- Artifact A_1

Datasets / Inputs. Refer to Table 4 for download link and Table 5 for variables used.

Table 4: Input Datasets with Downloadable Links

Dataset	Links	Size
NYX	Click to Download	8 GB
LETKF	Click to Download	8.4 GB
Miranda	Click to Download	4.5 GB
Hurricane ISABEL	Click to Download	1.5 GB
JHTDB	Click to take a overview Click to Apply a token Click to Download through Scis-erver slice [1536:2560, 1024:3072, 1024:3072] from $4096 \times 4096 \times 4096$	48 GB
mini-JHTDB	slice [512:1024, 0:1024, 0:1024] from the $1024 \times 2048 \times 2048$ JHTDB **OR** slice [2048:2560, 1024:2048, 1024:2048] from original $4096 \times 4096 \times 4096$	6 GB

Table 5: Input Datasets with Variables

Dataset	Variables
NYX	(For QoI) velocity_x, velocity_y, velocity_z (For others) temperature
LETKF	PRES
Miranda	velocityz
Hurricane ISABEL	Pf48
JHTDB	velocityx/y/z
mini-JHTDB	velocityx/y/z

Installation and Deployment.

Artifact Execution

The experiments need to be done following the task order:

Task 1 Download datasets and un-tar each dataset

Task 2 Build and install artifact A_1

Task 3 For $C_1 - C_3$, configure the data path in `HP_MDR_Perf.sh` and run the `HP_MDR_Perf.sh <cuda | hip>` to refactor and reconstruct data on NVIDIA or AMD GPUs. For reconstruction error bounds, we use $1e^{-1}$, $1e^{-2}$, $1e^{-3}$, $1e^{-4}$, $1e^{-5}$, and $1e^{-6}$ as the relative error bound.

Task 4 For C_4, scripts are provided to run the `mdr-x-qoi` executable to refactor and reconstruct data on a single GPU, then run the `pmdr-x-qoi` executable to refactor and reconstruct data using multi-GPUs. For single GPU reconstruction error bounds, we use $1e^{-1}$, $5e^{-2}$, $1e^{-2}$, $5e^{-3}$, $1e^{-3}$, $5e^{-4}$, $1e^{-4}$, $5e^{-5}$, and $1e^{-5}$. For multi-GPUs reconstruction error bounds, we use $1e^{-1}$, $1e^{-2}$, $1e^{-3}$, $1e^{-4}$, and $1e^{-5}$.

Artifact Analysis (incl. Outputs)

- For $C_1 - C_3$, when running each test, the bitplane encoding/decoding throughput, lossless compression throughput, data retrieval sizes, and end-to-end refactoring and reconstruction throughput will be printed or written to files.
- For C_4, when running each given test script, the throughput of progressive retrieval pipeline with guaranteed error control under a common QoI V_{total} on GPUs will be printed, so do the bitrate, corresponding requested tolerance, max estimated error, and max real error.

  Get the results according to the following instructions:

  (1) For results of NYX in Table 2, Figure 13, and Figure 14: run the `QoI_NYX.sh` script which includes downloading NYX dataset, concatenating velocity files, refactoring, and retrieval.

  (2) For results of mini-JHTDB in Table 3, Figure 13, and Figure 14: Suppose the data downloading and slicing are finished, run the `QoI_mini-JHTDB.sh` script which includes refactoring and retrieval.

  (3) For results of JHTDB in Figure 15: Suppose the data downloading and slicing are finished, run the `QoI_JHTDB.sh` script which includes refactoring and retrieval.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

Step 1: installation and compilation. A_1 and its dependency software can be built and installed using the follow two scripts with slight modification for the testing system, The build_scripts folder is located at MGARD/build_scripts:

- System with NVIDIA GPUs:

```
./build_scripts/build_mgard_cuda_hopper.sh NCORES
```

- System with AMD GPUs:

```
./build_scripts/build_mgard_hip_frontier.sh NCORES
```

For GPUs with other micro-architecture generations (e.g., Ampere), CMAKE_CUDA(HIP)_ARCHITECTURES (e.g., in line 86 for build_scripts/build_mgard_cuda_hopper.sh, with similar locations in other scripts) needs to be modified accordingly , or you can directly use different building scripts provided by us. For systems with RAM less than 16GB, we strongly suggest setting *NCORES* to 8 to avoid insufficient memory problems.

In addition, we provide two comparison baseline MDR and progressive compression. They can be compiled as follows.

For MDR baseline, clone the repository from https://github.com/MasterVChicken/OpenMP-Optimization-for-MDR_CPU.git and using the following script with slight modification for the testing system:

```
./build_script.sh
```

Before building, make sure to update the include path for EVA (a dependency of the MDR baseline) in test/CMakeLists.txt as described in README.md (Cloning SZ repository is a legacy issue which does not affect the functionality of MDR baseline).

For progressive compression baseline, clone the repository from https://github.com/MasterVChicken/Multi-Component-Compression-Framework.git and using the following script with slight modification for the testing

```
./build_script.sh 8
```

Now you can install all base compressors with our run script. For MGARD compressor, if you are with different micro-architecture generations, please modify line 41, 69 for Multi-Component-Compression-Framework /build_script.sh and line 20 for Multi-Component-Compression-Framework /CMakeLists.txt.

For MDR with QoI error control, clone the repository from https://github.com/Linus-Li-1037/qoi-control and using the following script:

```
./build_script.sh
```

Step 2: preparing input datasets. Please use the links in Table 4 to download all testing datasets. Some datasets come with multiple variables/fields, which needs to be un-tared to extra separated binary files.

Artifact Execution

C_1: Bitplane Encoder. Evaluating C_1 involves profiling three types of bitplane encoder: Locality Block, Register Shuffle, and Register Block. For Register Shuffle, there are four variants: shift, ballot, reduce-all, and match-any. Totally, there are 6 different encoder implementations.

To profile encoding and decoding: the ComposedRefactor and ComposedReconstructor code need to be modified.

ComposedRefactor:

```
../MGARD/include/mgard-x/MDR-X/Refactor/
ComposedRefactor.hpp
```

ComposedReconstructor:

```
../MGARD/include/mgard-x/MDR-X/Reconstructor/
ComposedReconstructor.hpp
```

Step 1: To enable bitplane encoder profiling, ProfileBPEncoder toggle needs to be set to true for both ComposedRefactor and ComposedReconstructor.

Step 2: Un-comment only *one* bitplane encoder (Encoder): BPEncoderLocalityBlock, BPEncoderRegisterBlock, BPEncoderRegisterShift, BPEncoderRegisterBallot, BPEncoderRegisterReduceAll, BPEncoderRegisterMatchAny.

Step 3: Recompile the updated project by running the build script again.

Step 4: Modify the DATA_DIR of the BP_Encoder_Perf.sh to point to the NYX dataset. Then, run this script:

```
BP_Encoder_Perf.sh cuda or hip
```

Step 5: To try out a different bitplane encoder. Repeat Step 2 to 4.

Step 6: When finished, restore the ProfileBPEncoder toggle to be *false* for both ComposedRefactor and ComposedReconstructor and recompile the project.

C_2: Lossless compressor. Evaluating C_2 involves profiling three types of lossless compressor: Huffman, run-length encoding (RLE), and a hybrid compressor with a configurable parameter (cr_threshold). The cr_threshold variable controls the trade-off between compression ratio and speed. Lower value (minimum is 1.0) leads to maximum compression.

To profile compression speed and data retrieval size: the ComposedRefactor and ComposedReconstructor code need to be modified.

ComposedRefactor:

```
../MGARD/include/mgard-x/MDR-X/Refactor/
ComposedRefactor.hpp
```

ComposedReconstructor:

```
../MGARD/include/mgard-x/MDR-X/Reconstructor/
ComposedReconstructor.hpp
```

Step 1: Un-comment only *one* lossless compressor (Compressor):

- DefaultLevelCompressor<T_bitplane, HUFFMAN, ...>
- DefaultLevelCompressor<T_bitplane, RLE, ...>
- HybridLevelCompressor<T_bitplane, ...>

Step 2: Recompile the updated project by running the build script again.

Step 3: Modify the DATA_DIR of the Lossless_Perf.sh to point to the NYX, LETKF, Miranda, Hurricane ISABEL datasets. Then, run this script:

```
Lossless_Perf.sh cuda or hip
```

Step 4: To try out a different lossless compressor. Repeat Step 1 to 3.

Step 5: To configure cr_threshold, modify the cr_threshold member of HybridLevelCompressor. These three values were used in our

evaluation: 1.0, 2.0, or 4.0. After modified the value of cr_threshold, repeat 2 and 3 to perform test.

`../MGARD/include/mgard-x/MDR-X/LosslessCompressor/HybridLevelCompressor.hpp`

Step 6: When finished, restore the Compressor to HybridLevelCompressor with cr_threshold=2.0 and recompile the project.

C_3: End-to-End Refactoring and Reconstruction. Evaluating C_3 involves profiling the end-to-end refactoring and reconstruction performance. To evaluate the pipeline optimization, we used two baseline works to compare (1) the performance of refactoring and reconstruction; (2) data retrival size for total six target errors $1e^{-1}$, $1e^{-2}$, $1e^{-3}$, $1e^{-4}$, $1e^{-5}$, and $1e^{-6}$.

To evaluate our HP-MDR refactoring and reconstruction pipeline, we have provided an executable mdr-x. To refactor data:

```
mdr-x --refactor --input <input data> --output          \
  <refactored data> -dt <s:float|d:double> -dim <ndim>  \
  <dim1> <dim2> <dim3> -d <cuda|hip>
```

To reconstruct data:

```
mdr-x --reconstruct --input <refactored data> --output  \
  <reconstructed data> --original <original data>       \
  -dt <s:float|d:double> -dim <ndim> <dim1> <dim2>      \
  <dim3> -d <cuda|hip>
```

Step 1: We provided a run script to run mdr-x on the four datasets and six error settings. To run the script, modify the DATA_DIR of the HP_MDR_Perf.sh to point to the NYX, LETKF, Miranda, Hurricane ISABEL datasets. Then, run this script:

```
HP_MDR_Perf.sh cuda or hip
```

To evaluate the MDR baseline, we have provided 2 executables in OpenMP-Optimization-for-MDR_CPU repository:

test_refactor_omp and test_reconstructor_omp. To refactor data:

```
test_refactor_omp <input data> <num of levels> \
  <num of bitplanes> <ndim> <dim1> <dim2> <dim3>
```

To reconstruct data:

```
test_reconstructor_omp <original data> <error mode> \
  <nerr> <err1> <err2> <err3> <smoothness parameter>
```

Step 1: We provided a run script to run MDR baseline on the four datasets and six error settings. To run the script, modify the DATA_NYX, DATA_ISABELLA, DATA_SCALE_LETKF, DATA_MIRANDA to point to the NYX, Hurricane ISABEL, LETKF, Miranda datasets. Then, run this script:

```
run_script.sh
```

To evaluate the progressive compression baseline, we have provided 3 executables ProgressiveCompressionCPU, ProgressiveCompressionGPUMGARD, ProgressiveCompressionGPUZFP in Multi-Component-Compression-Framework repository. To refactor and reconstruct data:

```
ProgressiveCompressionCPU -i <input data> -n <ndim>    \
  <dim1> <dim2> <dim3> -ECount <nerr> -E <nerr> <err1> \
  <err2> <err3> -p <data precision>

ProgressiveCompressionGPUMGARD -i <input data> -n <ndim> \
  <dim1> <dim2> <dim3> -ECount <nerr> -E <nerr> <err1>   \
  <err2> <err3> -p <data precision>

ProgressiveCompressionGPUZFP -i <input data> -n <ndim> \
  <dim1> <dim2> <dim3> -ECount <nerr> -E <nerr> <err1> \
  <err2> <err3> -p <data precision>
```

Step 1: We provided a run script to run progressive baseline on the four datasets and six error settings. To run the script, modify the each IN_DATA to point to the NYX, LETKF, Miranda, Hurricane ISABEL datasets. Then, run the following scripts:

```
run_script_cpu.sh

run_script_gpu_mgard.sh

run_script_gpu_zfp.sh
```

C_4: Progressive retrieval with QoI error control. Evaluating C_4 involves profiling the efficiency and performance of proposed QoI error bound estimation methods on single-GPU, validating the guaranteed QoI error control on single-GPU, and profiling the multi-GPU kernel throughput, IO time, and end-to-end data retrieval time.

To evaluate the efficiency and performance of proposed QoI error bound estimation methods (CP, MA, MAPE(c=2), and MAPE(c=10)) and validate the guaranteed QoI error control on single-GPU, we have provided an executable mdr-x-qoi. To refactor data:

```
mdr-x-qoi --refactor --input <input data> --output \
  <refactored data> -dt <s:float|d:double> -dim    \
  <ndim> <dim1> <dim2> <dim3> -dd max-dim -d <cuda|hip>
```

To reconstruct data:

```
mdr-x-qoi --reconstruct --input <refactored data> --output \
  <reconstruct data> --original <original data> -dt        \
   <s:float|d:double> -dim <ndim> <dim1> <dim2> <dim3>     \
   -m abs -e <relative error bound> -s inf -ar 0          \
   -d <cuda|hip> -dm <0:CP|1:MA|2:MAPE(c=2)|3:MAPE(c=10)>
```

To evaluate the multi-GPU kernel throughput and end-to-end data retrieval time, we have provided an executable `pmdr-x-qoi`. To refactor data:

```
pmdr-x-qoi --refactor --input <input data> --output \
  <refactored data> -dt <s:float|d:double>          \
  -dim <ndim><dim1><dim2><dim3> -dd max-dim -d <cuda|hip>
```

To reconstruct data:

```
pmdr-x-qoi --reconstruct --input <refactored data>       \
  --output <reconstruct data> --original <original data> \
  -dt <s:float|d:double> -dim <ndim> <dim1> <dim2> <dim3>\
  -m abs -e <relative error bound> -s inf -ar 0          \
  -d <cuda|hip> -dm <0:CP|1:MA|2:MAPE(c=2)|3:MAPE(c=10)>
```

Also, we provided an repository for MDR with QoI error control: https://github.com/Linus-Li-1037/qoi-control as our baseline. There are two executables `para_refactor` and `para_VTOT` in `qoi-control/parallel_src`. The data directory should be arranged like the followings:

```
JHTDB{Path_prefix}
+-- JHTDB0
|   +-- data
|   |   +-- VelocityX.dat
|   |   +-- VelocityY.dat
|   |   +-- VelocityZ.dat
|   +-- refactor
|       +-- VelocityX_refactored
|       +-- VelocityY_refactored
|       +-- VelocityZ_refactored
...
+-- JHTDB63 # same structure like JHTDB0
```

To refactor data:

```
mpirun -n 64 para_refactor 2 JHTDB \
  <path to JHTDB dataset, e.g. /Path/to/JHTDB/JHTDB>
```

To reconstruct data:

```
mpirun -n 64 para_VTOT <relative error bound>  \
  <refactored data, e.g. /Path/to/JHTDB/JHTDB> \
  <output directory>
```

Step 1: We provided two run scripts to run mdr-x-qoi on two datasets (NYX and mini-JHTDB) and five error settings. The script for NYX can be directly executed as long as you are under the directory:

```
MGARD
```

Then, run the script:

```
QoI_NYX.sh
```

To run the script for mini-JHTDB, download the JHTDB dataset listed in Table 4. Each downloaded chunk will be stored as a 4D array of shape $Dim1 \times Dim2 \times Dim3 \times 3$, where the last dimension contains the velocity components (V_x, V_y, V_z). You will need to split this last dimension into three separate 3D arrays (one for each component), and then concatenate them sequentially (Vx first, then Vy, then Vz) into a single file as required by the script.

Additionally, if you download the dataset in multiple spatial blocks (e.g., by parallel tile-based cutout requests), you must first assemble all Vx sub-blocks into a single global Vx volume, do the same for Vy and Vz, and finally concatenate these three full-volume components in order.

Before run the script, modify each data location in line 47 and line 54 of `QoI_mini-JHTDB.sh`.

Then, run the script:

```
QoI_mini-JHTDB.sh
```

Step 2: We provide a run script to evaluate pmdr-x-qoi on the JHTDB dataset with five error settings. The required data preparation and input format are identical to those described in **Step 1**. Please ensure the dataset has been preprocessed accordingly before running the script. Then, modify each data location in line 55 and line 61 of `QoI_JHTDB.sh`, and run:

```
QoI_JHTDB.sh
```

As for the Multi-CPU Kernel throughput, IO time, and end-to-end MDR baseline with QoI error control, clone https://github.com/Linus-Li-1037/qoi-control:

```
git clone https://github.com/Linus-Li-1037/qoi-control.git
cd qoi-control
sh build_script.sh
```

Then, run the script:

```
CPU_JHTDB.sh
```

Then, use MPI to run the `para_refactor` and `para_VTOT` on JHTDB with five error settings.

Artifact Analysis (incl. Outputs)

C_1: Bitplane Encoder. **Output:** The output should show the encoding and decoding throughput (GB/s) per level.

```
[time] (De)Encoding level: XXX s (YYY GB/s)
```

Analysis: When comparing different implementations: Register Block should largely outperform all other bitplane encoder designs. Locality Block should be the second best for most case. Register Shuffle approach leads to the worse performance with some variability between the four variants. Figure 6&7 shows the performance of the last five levels.

C_2: Lossless Compressor. **Output:** The output should show the per step time cost and throughput for refactoring and reconstructing the four datasets to three error bounds. For evaluation, we focus only on evaluating the lossless compressor in terms of it (de)compress performance and how it impact the data retrieval size (how well it can compress bitplanes). The throughput is shown as the output line (compress when refactoring and decompress when reconstructing)

```
[time] Lossless: XXX s (YYY GB/s)
```

The data retrieval size is shown as the output line (only when reconstructing)

```
[info] Additional ZZZ bytes (WWW%) read for reconstruction
```

Analysis: Huffman should brings the least average retrieval sizes and speed. RLE, on the other hand, should brings high average retrieval sizes and speed. Hybrid approach enables the trade-off between the two. Figure 8 show the results of our evaluation.

C_3: Refactoring and Reconstruction. **Output:** For refactoring in HP-MDR, the output should show the refactoring time and throughput (GB/s).

```
[time] Refactor pipeline: XXX s (YYY GB/s)
```

For reconstruction, HP-MDR will print the incremental reconstruction time to the error bound and corresponding throughput (GB/s) and additional data size (bytes) retrieved.

```
[time] Reconstruct pipeline: XXX s (YYY GB/s)
[info] Additional XXX bytes read for reconstruction
```

For MDR baseline, the output should show the total refactor time (s) in file:`\build\refactor_time.txt` and additional data size (retrieved data size) (bytes) and incremental retrieval time (s) in file: `retrieved_size.txt` taken to reconstruct to dedicated tolerance.

```
Parallel refactor time: XXX s
tolerance XXX -> retrieved size = YYY bytes
              -> retrieved time = ZZZ
```

For progressive baseline, the output should write the index of component, current tolerance, time(s) , additional data size (compressed size) (bytes) as well as total refactor time to file:
result_<compressor name>.csv

```
Index,Tolerance,Time(s),CompressedSize(Bytes)
XXX, YYY, TTT, ZZZ
Refactor Time AAA
```

It will also write index of component, target tolerance and decompression time to file:
result_reconstruct_bound_<compressor name>.csv.

```
Component,TargetTolerance,DecompressionTime(s)
XXX, YYY, ZZZ
```

Analysis: For refactoring time, HP-MDR consumes least time among all baselines, thus produces highest throughput. As for progressive reconstruct, HP-MDR steadily outperforms all baselines with six error settings. When comparing the additional size required during reconstruction, HP-MDR does not retrieve the smallest data size but still remains competitive. This means HP-MDR enables the trade-off between the least retrieval size and speed. Figure 12 show

the result of our evaluation.

C_4: Progressive retrieval with QoI error control. **Output:** The output of `QoI_NYX.sh` and `QoI_mini-JHTDB.sh` should show the compression efficiency (bitrate, Bitrate in Table 2&3), performance (Reconstruct pipeline, overall kernel throughput in Figure 13), and QoI error control (Requested_Tau, Est_max_error, and Real_max_error with respect to Requested tolerance, Max estimated error, and Max real error in Figure 14) on a single GPU in their output files `NYX_output.txt` and `mini-JHTDB_output.txt` like the following format:

```
<QoI error bound estimation methods>, Request eb = AAA,
  Bitrate = BBB, Reconstruct pipeline: XXX s (YYY GB/s),
  Requested_Tau = CCC, Est_max_error = DDD,
  Real_max_error = EEE
```

For the results of HP-MDR in Figure 15, the output of `QoI_JHTDB.sh` should show the end-to-end data retrieval time on multi-GPU in `JHTDB_output.txt` like the following format:

```
Request eb = AAA, Bitrate = BBB, IO_time = CCC,
  max_kernel_time = DDD, max_elapsed_time = XXX,
  Requested_Tau = EEE, Est_max_error = FFF,
  Real_max_error = GGG
```

The Multi GPU Kernel Throughput can be computed as

$$\frac{sizeof(JHTDB)}{max\_kernel\_time}$$

where $sizeof(JHTDB)$ is 48 GB.

For the results of MDR in Figure 14, the output of `CPU_JHTDB.sh` should show the end-to-end data retrieval time as `elapsed_time` and IO time as `IO_time` :

```
Request eb = AAA, Aggregated bitrate = BBB,
  retrieved_size = CCC, total_num_elements = DDD,
  IO_time = EEE, elapsed_time = XXX,
  Target V_TOT error = FFF,
  Max aggregated V_TOT est_error = GGG,
  Max aggregated V_TOT act_error = HHH
```

Then the multi CPU kernel throughput can be computed as

$$\frac{sizeof(JHDTB)}{elapsed\_time - IO\_time}$$

where $sizeof(JHTDB)$ is 48 GB.

Analysis: For compression efficiency and performance evaluation, MAPE c=10 should achieve a good tradeoff between ensuring a suboptimal bitrate and maintaining a relatively high throughput.

For the QoI error control validation, the QoI errors are always bounded in all proposed error bound estimation methods: CP, MA, MAPE c=2, MAPE c=10.

HP-MDR consumes less end-to-end data retrieval time and produces higher kernel throughput compared with MDR baseline, although the I/O overhead is more significant in HP-MDR.