cuSZp2: A GPU Lossy Compressor with Extreme Throughput and Optimized Compression Ratio

Yafan Huang[†], Sheng Di[‡*], Guanpeng Li[†], Franck Cappello[‡]
[†] Computer Science Department, University of Iowa, Iowa City, IA, USA
[‡] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
yafan-huang@uiowa.edu, sdi1@anl.gov, guanpeng-li@uiowa.edu, cappello@mcs.anl.gov

***Abstract*—Existing GPU lossy compressors suffer from expensive data movement overheads, inefficient memory access patterns, and high synchronization latency, resulting in limited throughput. This work proposes cuSZp2, a generic single-kernel error-bounded lossy compressor purely on GPUs designed for applications that require high speed, such as large-scale GPU simulation and large language model training. In particular, cuSZp2 proposes a novel lossless encoding method, optimizes memory access patterns, and hides synchronization latency, achieving extreme end-to-end throughput and optimized compression ratio. Experiments on NVIDIA A100 GPU with 9 real-world HPC datasets demonstrate that, even with higher compression ratios and data quality, cuSZp2 can deliver on average 332.42 and 513.04 GB/s end-to-end throughput for compression and decompression, respectively, which is around 2× of existing pure-GPU compressors and 200× of CPU-GPU hybrid compressors.**

***Keywords*—Data Compression, Parallel Computing, GPU**

I. Introduction

Modern scientific simulations and Large Language Model (LLM) training generate enormous volumes of data, creating a bottleneck for High-Performance Computing (HPC) systems. This big data issue motivates domain scientists to explore more efficient data reduction techniques. While lossless compressors are limited by their modest compression ratios [1] (around 2:1), error-bounded lossy compression [2]–[4] offers significantly higher compression ratios by introducing user-controllable errors, thus turns out to be a promising solution in HPC simulations, such as cosmology simulation [5], quantum circuit simulation [6], and seismic imaging [7], [8].

A. Motivation for Ultra-Fast GPU Lossy Compression

Recently, there have been increasingly more HPC scenarios requiring GPU compression and rapid processing speeds [9]–[14]. One example is *Reducing Data Stream Intensity* [10]. In the Linear Coherent Light Source (LCLS) [11], a leading free-electron laser facility at the Stanford Linear Accelerator Center, the raw acquisition rate of high-brilliance X-ray beams reaches approximately 250 GB/s. This rate demands a compression throughput that exceeds the capabilities of CPU-based compressors, underscoring the need for high-speed GPU solutions. Another case is *Benefiting LLM Training*. LLaMA [15], for example, takes 2,048 NVIDIA A100 GPUs to store its parameters and 21 days to complete model training [9]. To use lossy compression to reduce such GPU memory footprint, any expensive CPU computations or CPU-GPU data movement overhead can downgrade performance drastically. Specifically, while theoretical computation throughput for GPU can reach thousands of GB/s [16], PCIe [17], transferring data between CPUs and GPUs, has only a limited throughput of around 10∼20 GB/s. CPU-GPU hybrid designs can result in much longer training periods, thus leading to huge financial losses. These practical scenarios drive researchers to explore ultra-fast GPU lossy compression techniques.

*Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA

B. Limitations of Existing Works and Goal

However, existing GPU lossy compressors suffer from limited throughput, with the underlying reasons detailed in Table I. For cuSZ [18], cuSZx [19], and MGARD-GPU [20], although the core compression algorithm executes within GPU, they require expensive CPU computations to perform global synchronization, build Huffman tree, or conduct GPU kernel communications. In the meanwhile, cuZFP [21], FZ-GPU [22], and cuSZp [23] have pure-GPU designs, but they either underutilize memory bandwidth or are bounded by latency, which critically impacts GPU kernel throughput [24].

Existing GPU Lossy Compressor	Pure GPU Design?	Single Kernel?	High MB Utilization?	Latency Control?
cuSZ	✗	✗	✗	—
MGARD-GPU	✗	✗	✗	—
cuSZx	✗	✓	✗	—
cuZFP	✓	✓	✗	—
FZ-GPU	✓	✗	✗	✗
cuSZp	✓	✓	✗	✗
cuSZp2 (our work)	✓	✓	✓	✓

TABLE I: Key designs related to throughput in existing GPU lossy compressors. "MB" denotes memory bandwidth.

Ideally, a promising GPU lossy compressor should satisfy:

- Pure-GPU design/implementation without any CPU computations and data movement overheads.
- Extreme throughput with high memory bandwidth utilization and high-speed latency control.
- High compression ratio and user-satisfied data quality – intrinsic requirements for designing a lossy compressor.

C. Our Solution: cuSZp2

In this work, we propose cuSZp2, an error-bounded lossy compressor purely executed in one GPU kernel, achieving extreme throughput, optimized compression ratios, and high reconstructed data quality. cuSZp2 compresses data at block granularity and combines each compressed block into a single, unified byte array. Specifically, there are three key designs in cuSZp2[1]. (1) *Outlier Fixed-Length Encoding*, which is a novel lossless encoding algorithm and cooperates with a fine-tuning selection strategy, allows cuSZp2 to achieve higher compression ratios than existing error-bounded GPU compressors. (2) *Vectorized Memory Accesses*. This design enables cuSZp2 to reduce the number of memory instructions and access global memory in a coalescing manner, highly utilizing memory bandwidth in modern GPU architectures. (3) *Global Prefix-sum via Decoupled Lookback*. Inspired by [25], we propose a fine-tuned compression-aware decoupled lookback strategy to minimize synchronization (i.e. concatenating compressed blocks within GPU) latency with TB-level throughput. Some key results of cuSZp2 evaluated from several representative HPC datasets are listed below:

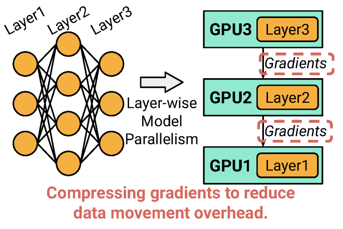
- On average, cuSZp2 offers 332.42 GB/s and 513.04 GB/s end-to-end throughput for compression and decompression on NVIDIA A100 GPU. This throughput is around 2× of existing pure-GPU lossy compressors and 200× of CPU-GPU hybrid lossy compressors.
- Compared with state-of-the-art error-bounded GPU lossy compressors, cuSZp2 has higher compression ratios in 24/27 cases, with high isosurface visualization quality.
- cuSZp2 delivers from 612.83 GB/s to 809.71 GB/s throughput for processing double-precision HPC datasets and supports random access with TB-level throughput.
- Such high throughput is also observed in other lower-end NVIDIA GPUs, such as RTX 3090 and RTX 3080, demonstrating the compatibility of cuSZp2 designs.

II. Rethinking Throughput in GPU Lossy Compressors: Why End-to-End?

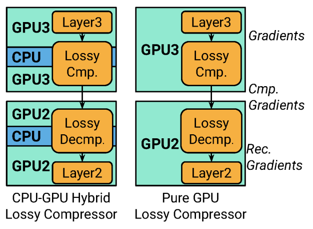
Due to the massive parallelism designs, GPU lossy compressors [21]–[23], [26] achieve much higher throughput (10×∼100×) compared with CPU solutions [2], [3], [20], [27], [28] and hence are widely used in inline compression tasks [10] that require rapid processing speed, such as reducing data stream intensity [11], [29] and memory footprint [6], [8]. In these scenarios, throughput, defined as the volume of data processed per unit of time (e.g. GB/s), is a primary concern in the design of such compressors.

To evaluate GPU lossy compressor throughput, current research typically focuses on either **kernel throughput**, which measures only GPU-executed compression functions (i.e. kernels), or **end-to-end throughput**, encompassing all computational processes to generate the output. This distinction is analyzed using a machine learning distributed training example, as shown in Figure 1. For example, in a three-layer neural network using layer-wise model parallelism [30]–[32] (Figure 1(a)), each layer is distributed to different GPUs. During

[1]cuSZp2 code: https://github.com/szcompressor/cuSZp. Compared with cuSZp, cuSZp2 has different algorithm designs and implementations. The reasons we maintain cuSZp2 in the same code repository are two-fold: (1) cuSZp2 is a better alternative to cuSZp in any perspective. (2) In principle, cuSZp2 follows a similar 4-stage compression pipeline with cuSZp.


Layer1
Layer2
Layer3
Layer-wise Model Parallelism
GPU3
Layer3
Gradients
GPU2
Layer2
Gradients
GPU1
Layer1
Compressing gradients to reduce data movement overhead.

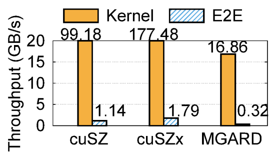
(a) Explaining how lossy compression can be used in distributed training.



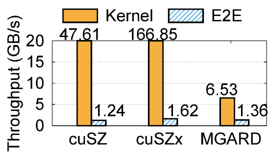
(b) CPU-GPU hybrid vs pure GPU lossy compressors.

Fig. 1: Demonstrating why end-to-end throughput is more important than kernel throughput using a straightforward example from machine learning distributed training.

backward propagation, gradients from each layer are transmitted to preceding layers for model training. In this case, as seen in Figure 1(b), both CPU-GPU hybrid compressors and pure GPU compressors can compress such gradients to minimize data movement overheads between different GPUs/nodes. It is possible to observe high kernel throughput for both types of compressors, however, the GPU kernels are only part of hybrid compressors – they also include CPU computations and data movement between CPU and GPU, which are routinely expensive but crucial in overall training phase. We show such differences in three CPU-GPU hybrid lossy compressors, including cuSZ [18], cuSZx [19], and MGARD-GPU [26]. As seen in Figure 2, while the kernel throughput can be as high as 177.48 GB/s, the end-to-end throughput only limits from 0.32 (MGARD compression) to 1.79 GB/s (cuSZx compression). *This gap shows that kernel throughput is an overly optimistic measurement and can drastically downgrade the performance of entire program executions, making end-to-end throughput a more appropriate evaluation metric.* This conclusion applies not just to ML training but also to other applications that collaborate on GPU clusters, including large-scale HPC simulations [33], [34] and MPI communications [35]–[37].



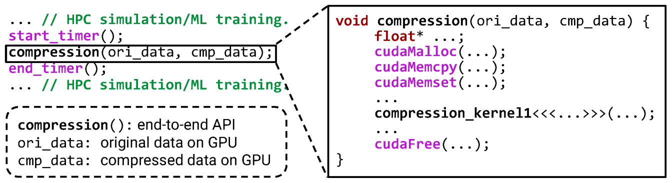
(a) Compression



(b) Decompression

Fig. 2: Kernel throughput vs end-to-end throughput in several CPU-GPU hybrid error-bounded lossy compressors.

In this study, we measure the end-to-end throughput of a GPU lossy compressor in a rigorous and practical way for users, as illustrated in Figure 3 (using compression as an example, which also applies to decompression). Given the original data on GPU, we treat compression API as a black box and include all code before a compressor generates the compressed data on GPU. This measurement method is rigorous even for some pure GPU compressors since it also includes all GPU intrinsic APIs, such as `cudaMemcpy()`, which are routinely not considered in past end-to-end measurements [22], [23].



```
... // HPC simulation/ML training.
start_timer();
compression(ori_data, cmp_data);
end_timer();
... // HPC simulation/ML training.
```

```
void compression(ori_data, cmp_data) {
     float* ...;
     cudaMalloc(...);
     cudaMemcpy(...);
     cudaMemset(...);
     ...
     compression_kernel1<<<...>>>(...);
     ...
     cudaFree(...);
}
```

```
compression(): end-to-end API
ori_data:  original data on GPU
cmp_data:  compressed data on GPU
```

Fig. 3: A code example for GPU compression (also for decompression) throughput measurement in this work.

Definition: For brevity, *Throughput* of a GPU compressor in this work is defined as *end-to-end throughput*, which encompasses entire computations between original (or compressed) data on GPU and compressed (or reconstructed) data on GPU, with the output being concatenated into a single, unified array.

III. cuSZp2: High-level Overview

We propose cuSZp2, a GPU error-bounded lossy compressor with ultra-fast throughput and optimized compression ratio. cuSZp2 executes holistic compression or decompression on a single GPU kernel, to avoid expensive data movement overhead and extra global memory accesses. In this section, we explain its high-level designs with a running example.

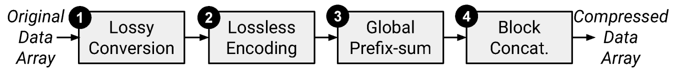


Fig. 4: Workflow in cuSZp2 compression kernel.

High-level overview. Figure 4 presents a high-level overview of the workflow in cuSZp2 compression kernel. Given an original HPC dataset, cuSZp2 segments it into uniformly-sized data blocks of consecutive floating-point numbers, then compresses them in parallel through four major steps. Within each block, cuSZp2 initiates the compression workflow with **Lossy Conversion (❶)**, transforming each floating-point number into a quantization integer within the user-defined error bound. Then, cuSZp2 compresses each processed data block by **Lossless Encoding (❷)**. Specifically, two encoding modes are supported in cuSZp2: *plain fixed-length encoding* and *outlier fixed-length encoding*. While the former preserves a fixed number of bits for each integer, the latter reduces this fixed number by adaptively storing the outlier. Since the length of each compressed block is different, cuSZp2 conducts **Global Prefix-sum (❸)** (i.e. device-level parallel prefix scan) to generate the index of each compressed block in the final compressed array. This step is achieved by the decoupled lookback strategy in cuSZp2, significantly reducing device-level synchronization latency compared with state-of-the-art GPU compressors [22], [23]. Finally, cuSZp2 combines all compressed blocks based on the indexes by **Block Concatenation (❹)**. For decompression, after generating the compressed block indexes via ❸, cuSZp2 reconstructs each data block in reverse order of the rest three steps (❹→❷→❶). Note that ❶ and ❹ in both compression and decompression kernels require reading input and writing output to global variables, hence they involve extensive access to GPU memory. In cuSZp2, these two steps are designed in a fully vectorized manner, highly utilizing the GPU memory bandwidth compared with existing solutions [21]–[23].

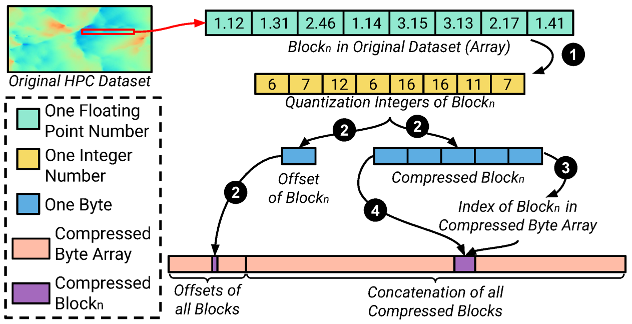


Fig. 5: A running example for compressing one data block in cuSZp2. Here block size is 8 and error bound is 0.1. The step number (e.g. ❶) refers to workflow mentioned in Figure 4.

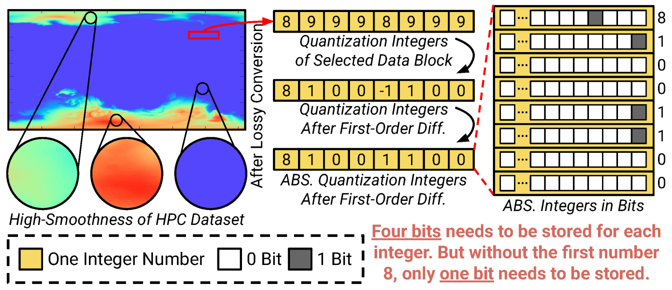
A running example. Figure 5 further explains the cuSZp2 compression algorithm with a running example on one data block. In this case, block size L is 8 and error bound eb is 0.1. First, ❶ converts each floating-point data into its corresponding quantization integer by a rounding (or ceiling) operation. For example, 1.12 is transformed into 6. In decompression, this number is reconstructed by $6 \times 2eb = 1.2$ to satisfy the error bound (i.e. $|1.12 - 1.2| < eb$). This is the only lossy step in cuSZp2. Then, ❷ compresses this integer block into two parts: the first part requires 1 byte to record the offset information and the second part (5 bytes in this block) saves the compressed data. We will explain more details for this encoding algorithm in Section IV-A. Since different compressed blocks exhibit different lengths, ❸ calculates indexes for these blocks by formulating this process as a prefix-sum problem. Based on these indexes, ❹ concatenates all information into a single, unified compressed byte array. Like individual data blocks, the final compressed byte array also consists of two parts. We store offset information because each data block's offset requires only 1 byte, ensuring predictable locations. cuSZp2 leverages this offset data to guide decompression to access a specific compressed data block.

IV. cuSZp2: Key Designs

In this section, we detail three key designs in cuSZp2. The step number (e.g. ❶) in this section still refers to Figure 4.

A. Outlier Fixed-length Encoding (in ❷)

Motivation for processing outliers. High smoothness, a notable feature of many HPC datasets [13], [38], implies that data values tend to be similar when they are spatially near each other. In Figure 6, we visualize one slice from CESM-ATM [38], a standard climate simulation dataset, to explain this motivation. For an arbitrary data block, lossy conversion transforms it into a set of integers with close values. If we use the first-order difference (i.e. $n_i' = n_i - n_{i-1}$) to eliminate redundant bit patterns, *the first element in this block possibly* *turns out to be an outlier.* In the context of fixed-length encoding, the presence of an outlier necessitates storing four bits for each integer, compared to just one bit in its absence. This defect not only exits for the fixed-length encoding mentioned above but naturally for all GPU compressors, since GPU parallelism requires processing datasets in a blockwise manner, breaking the spatial smoothness inevitably.



Quantization Integers of Selected Data Block
Quantization Integers After First-Order Diff.
ABS. Quantization Integers After First-Order Diff.
After Lossy Conversion
ABS. Integers in Bits
High-Smoothness of HPC Dataset
One Integer Number
0 Bit
1 Bit
Four bits needs to be stored for each integer. But without the first number 8, only one bit needs to be stored.

Fig. 6: Explaining a drawback of GPU blockwise designs and *plain fixed-length encoding* while compressing a data block with smooth values. This block is also used in Figure 7 and 8.

Outlier fixed-length encoding (Outlier-FLE). To manage outliers efficiently, this work proposes Outlier-FLE. In contrast, existing fixed-length encoding is denoted as Plain-FLE. The main idea and benefits of Outlier-FLE are illustrated in Figure 7. Given a set of quantization integers after first-order difference, Outlier-FLE first gets their absolute values and stores signs separately, allocating one bit per integer. A bit value of 0 or 1 indicates a positive or negative integer. In this example, as the block size is 8, the length of aggregated signs is 1 byte. For preserving the outlier, value 8 is within 0$\sim$255, indicating Outlier-FLE can save it with only 1 byte without any losses. For preserving the remaining integers, Outlier-FLE stores 1 bit (i.e. length of effective bits) for each integer. In all, Outlier-FLE compresses this data block ($8 \times 4 = 32$ bytes) into 3 bytes, achieving a compression ratio of $32/3 = 10.7$, whereas this number in Plain-FLE is only $32/5 = 6.4$.

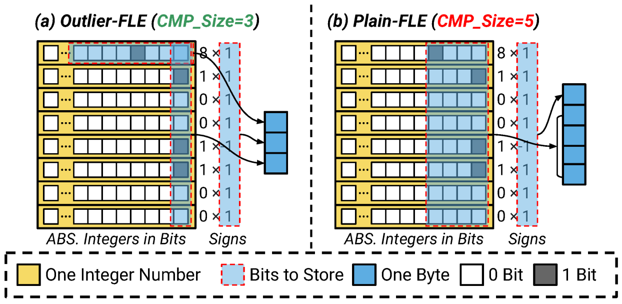


Fig. 7: Illustrating the main idea of *outlier fixed-length encoding* and its benefits over *plain fixed-length encoding*.

Recall that two modes are supported in the Lossless Encoding (❷) of CUSZP2 workflow (see descriptions for Figure 4). While using Outlier-FLE as the target mode, CUSZP2 adopts a fine-tuned selection strategy, that is, *"for each data block, selecting Outlier-FLE only when it offers a higher compression ratio"*. CUSZP2 is capable of this selective approach because the compression ratios for both Outlier- and Plain-FLE can be determined by simply iterating the absolute values of all integers within a block, avoiding any costly re-computations. The selection information, as shown in Figure 8, is recorded in the block offset (definition see Figure 5). Since the absolute value of a signed int32 data ranges from 0 to $2^{31} - 1$, the number of fixed-lengths can only range from 0 to 31, fitting within a 5-bit representation. Unlike Plain-FLE, which does not utilize the three most significant bits, Outlier-FLE employs these for outlier data encoding. The foremost bit serves as a mode flag: a value of 1 signals Outlier-FLE compression; otherwise, Plain-FLE is indicated. The next two bits are used to encode the outlier's size adaptively – 00, 01, 10, or 11 denote outlier sizes of 1, 2, 3, or 4 bytes, respectively. This adaptive strategy enhances compression ratios and outlier storage efficiency without adding overhead.

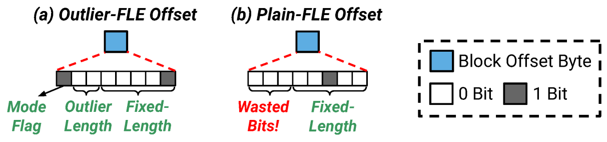


Fig. 8: The block offset design in *outlier fixed-length encoding*.

The other Lossless Encoding mode in CUSZP2 strictly employs Plain-FLE for all data blocks, and there are two reasons for maintaining both encoding modes. (1) *Variability in HPC dataset characteristics*: Not all HPC datasets, such as the VX field from HACC [13] (a premier cosmology simulation) and QMCPACK [39] (a quantum Monte Carlo simulation), display significant smoothness. In these instances, Plain and Outlier modes achieve almost identical compression ratios. (2) *Trade-offs between modes*: Each mode offers distinct advantages. The Outlier mode can deliver optimized compression ratios and enhanced throughput, while the Plain mode, without the fine-tuned selection strategy, targets extreme throughput. Note that even the Outlier mode in CUSZP2 offers much higher throughput than any other existing GPU lossy compressors, and we will evaluate them thoroughly in Section V and VI.

B. *Vectorized Memory Accesses (in ❶ and ❹)*

Existing GPU lossy compressors are highly memory-bounded. Compression algorithms [18], [20], [21], [23], routinely characterized by modular structures, exhibit inherent irregularity compared with computation-intensive tasks such as matrix-matrix multiplication on GPUs [40]–[42]. This modular composition leads to diverse memory access patterns, making these algorithms more susceptible to being memory-bounded. We demonstrate this statement in Figure 9. Since CPU-GPU data movement overheads already highly limit the throughput of hybrid compressors [18]–[20], we only focus on compressors with pure GPU designs, including cuZFP [21], FZ-GPU [22],

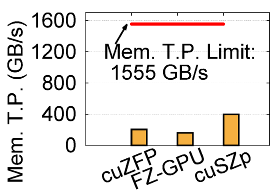


Fig. 9: Memory throughput for 3 lossy compressors with pure GPU designs (on NVIDIA A100 (40 GB) GPU).

and cuSZp [23]. We conducted tests using field *P3000* from RTM [43] dataset (from seismic imaging) on an NVIDIA A100 GPU (40 GB) and assessed the memory throughput with Nsight Compute (CUDA V11.2). The memory throughput observed for these compressors ranged between 159.95 GB/s (FZ-GPU) and 397.26 GB/s (cuSZp) – substantially below the A100 GPU's memory bandwidth capacity of 1555 GB/s. This significant underutilization of the memory capabilities motivates us to optimize memory access behaviors in cuSZp2.

Vectorized memory accesses. We propose vectorized memory accesses to exploit GPU global memory bandwidth when performing read (❶) and write (❹) operations, thereby achieving higher throughput in cuSZp2. The key idea is two-fold. (1) *Within each data block, use vector variables to reduce the number of memory instructions.* (2) *At data block-level, enable memory access in a coalescing manner.*

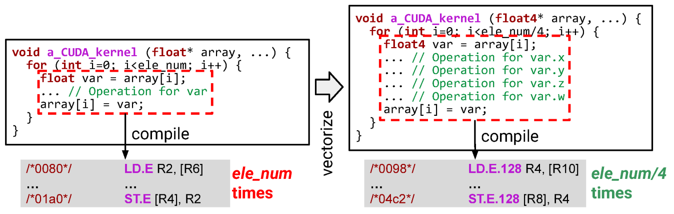
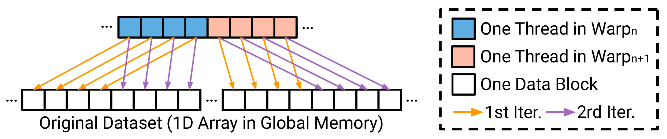

```
void a_CUDA_kernel (float* array, ...) {
  for (int i=0; i<ele_num; i++) {
    float var = array[i];
    ... // Operation for var
    array[i] = var;
  }
}
```
```
/*0080*/          LD.E R2, [R6]
...               ...
/*01a0*/          ST.E [R4], R2
```
```
void a_CUDA_kernel (float4* array, ...) {
  for (int i=0; i<ele_num/4; i++) {
    float4 var = array[i];
    ... // Operation for var.x
    ... // Operation for var.y
    ... // Operation for var.z
    ... // Operation for var.w
    array[i] = var;
  }
}
```
```
/*0098*/          LD.E.128 R4, [R10]
...               ...
/*04c2*/          ST.E.128 [R8], R4
```

Fig. 10: Reducing the number of memory instructions by vectorizing a CUDA kernel function (below is SASS code).

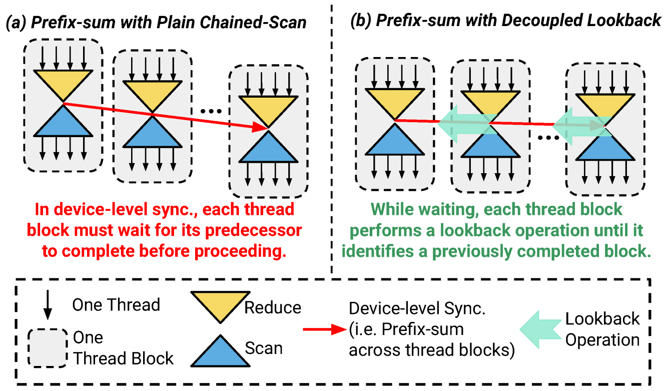
Figure 10 illustrates the benefits of using vector variables with a demo CUDA code and its corresponding SASS instructions (i.e. low-level assembly language for NVIDIA GPU). In the original program (left), a loop iterates over each `float` element, conducting memory instructions `LD.E` and `ST.E` to load and store 32-bit width for `ele_num` times. After vectorization (right), each iteration groups four consecutive elements into a `float4` variable, and performs memory operations together. The resulting compiled code shows that vectorized SASS instructions `LD.E.128` and `ST.E.128` execute only `ele_num/4` times, significantly reducing the number of memory instructions and maximizing L1 cache utilization, and hence better utilizing the memory bandwidth. This loop vectorization design also reduces control-flow penalties [44], [45], which GPUs generally handle less efficiently. Note that vectorizing the entire compression workflow can be extremely challenging. cuSZp2 *realizes this due to the inherently regular nature of fixed-length encoding, which treats each element uniformly (i.e. preserving the same number of bits).* In contrast, vectorizing Huffman [46], [47] or run-length encoding [48] can drastically complicate program control-flow, negating the benefits of vectorization and potentially reducing GPU compression throughput in return.

Figure 11 shows how we enable coalescing memory access patterns in cuSZp2 at the data block level. In cuSZp2, each thread compresses multiple data blocks iteratively – one data block for one iteration. In each iteration, the key design is to *let all threads in one warp (i.e. consecutive 32 threads) access adjacent data blocks in global memory*. By doing so, memory transactions are consolidated, as the GPU's memory controller


Original Dataset (1D Array in Global Memory)
One Thread in Warp$_n$
One Thread in Warp$_{n+1}$
One Data Block
1st Iter.
2rd Iter.
Fig. 11: Illustration of enabling coalescing global memory access at data block-level in cuSZp2. Note that a real warp in CUDA contains 32 threads.

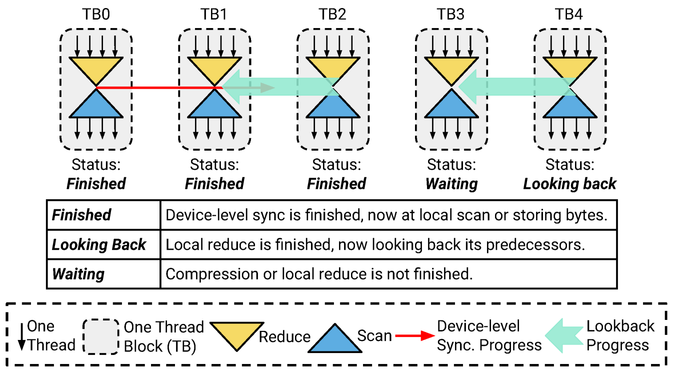
can merge accesses from the warp into fewer transactions, thus optimizing memory bandwidth usage. After the above two memory optimizations, with the same setting in Figure 9, cuSZp2 can achieve 1330.24 GB/s memory throughput in the compression stage. We will present more details in Section V.

C. Global Prefix-sum via Decoupled Lookback (in ❸)


(a) Prefix-sum with Plain Chained-Scan
In device-level sync., each thread block must wait for its predecessor to complete before proceeding.
(b) Prefix-sum with Decoupled Lookback
While waiting, each thread block performs a lookback operation until it identifies a previously completed block.
One Thread
One Thread Block
Reduce
Scan
Device-level Sync. (i.e. Prefix-sum across thread blocks)
Lookback Operation
Fig. 12: Illustrating the ideas of two device-level prefix-sum designs: plain chained-scan (left, without latency control) and decoupled lookback in cuSZp2 (right, with latency control).

Limitation for existing synchronization techniques. Synchronization in blockwise parallel compressors [19], [22], [23], [49] is crucial for determining the location of each compressed block before concatenating them into a single, unified byte array. This task can be formulated as an exclusive prefix-sum problem, where each block must know the total length of all its preceding blocks. While this task on CPUs can be realized by straightforwardly constructing a loop [19], the same thing on GPUs poses substantial challenges. The inherent linear recurrences disrupt GPU parallelism [50], leading to inevitable latency. To tackle this, existing GPU compressors adopt a "*Reduce-then-Scan*" strategy [25] comprising three steps. (1) *Reduce*: Computing the total compressed block length for all threads within a thread block. (2) *Global Synchronization*: Performing prefix-sum across different thread blocks, retrieving synchronized device-wide total lengths. (3) *Scan*: Still within each thread block, distributing the synchronized lengths to each thread, allowing each data block to know its location in the final compressed data. Since atomic operation for global memory is relatively slow [22], [51], the state-of-the-art approach for global synchronization is plain chained-scan [23], [52], [53], as explained in Figure 12 (left). As seen, the global synchronization is performed via a serial implementation, where *each thread block must wait for its predecessors to* *complete before proceeding*. This design unavoidably leads to high latency, especially for large HPC datasets, which is intolerable for a compressor that targets extreme throughput.

Hiding latency with decoupled lookback. In cuSZp2, we hide such synchronization latency via a decoupled lookback strategy. Inspired by [25] and [23], we propose a compression-aware adaptive lookback design and incorporate it into the single-kernel compression workflow (❸). The main idea can be explained in Figure 12 (right). *While the serial chained-scan is performing (see the red arrow), each thread block performs a lookback operation for its predecessors and aggregates the local reduction lengths until it identifies a previously completed thread block.* In this process, cuSZp2 decouples the serial chained-scan and exploits computation resources in those "waiting" blocks, thus controlling latency successfully.



Finished	Device-level sync is finished, now at local scan or storing bytes.
Looking Back	Local reduce is finished, now looking back its predecessors.
Waiting	Compression or local reduce is not finished.

Fig. 13: Explaining the implementation of decoupled lookback in cuSZp2 using an example captured at a particular moment.

In Figure 13, we capture a moment to explain the implementation of this strategy in cuSZp2. Each thread block (`TB0` to `TB5`) can be in one of three possible statuses. (1) *Finished*: It indicates device-level synchronization finishes, and this thread block is in later steps such as local reduce and storing bytes (or complete). (2) *Looking Back*: In this status, although the local scan is complete, device-level synchronization has not yet propagated to this thread block. So it is looking back at its predecessors and aggregating their local reduction values. (3) *Waiting*: This status means the compression or local scan within this thread block has not finished. When a *Looking Back* thread block aggregates a *Waiting* one, it will also wait until it is completed. In this example, `TB2` achieves *Finished* status by looking back `TB1` rather than device-level synchronization. In the next moment, the device-level synchronization will bypass computations in `TB2` (i.e. decouples the original "chain") and look for the next unfinished thread block, resulting in much lower latency than plain chained-scan. Our fine-tuned implementation allows device-level synchronization on several representative HPC datasets to achieve 846.85 GB/s throughput, 2.41 times of the state-of-the-art approaches. We will show more results in Section V.

V. Evaluation

In this section, we evaluate cuSZp2 with several state-of-the-art GPU lossy compressors from three perspectives: throughput, compression ratio, and reconstructed data quality.

A. Experimental Setups

1) Platforms: We evaluate cuSZp2 and other GPU compressors on one NVIDIA A100 (108 SMs, 40 GB) GPU, provided by Swing Cluster [54] from Argonne National Laboratory. Each node in Swing Cluster has two AMD EPYC 7742 CPUs and 1 TB of DDR4 memory. The operating system is Ubuntu 20.04 and the CUDA toolkit version is 11.2 (including NVCC compiler and Nsight Compute).

2) Dataset: We selected 9 real-world HPC datasets (details see Table II) from 2 suites: Scientific Data Reduction Benchmarks (SDRBench) [55] and Open Scientific Visualization Datasets (Open-SciVis) [56]. These datasets are from various HPC domains and have been extensively studied in recent data compression works [19], [20], [22], [23], [57]–[59].

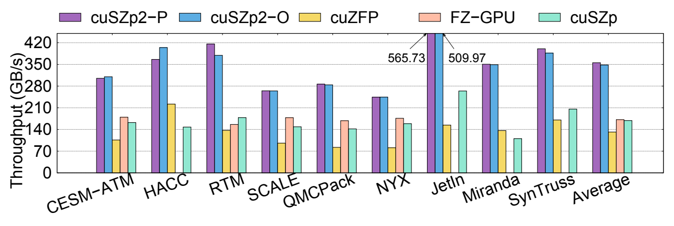
Datasets	Suite	Dims per Field	# Fields	Total Size
CESM-ATM [38]	SDRBench	$3600\times1800\times26$	33	20.71 GB
HACC [13]	SDRBench	1,073,726,487	6	23.99 GB
RTM [43]	SDRBench	$1008\times1008\times352$	3	3.99 GB
SCALE [60]	SDRBench	$1200\times1200\times98$	12	6.31 GB
QMCPack [39]	SDRBench	$69\times69\times33120$	2	1.17 GB
NYX [61]	SDRBench	$512\times512\times512$	6	3.00 GB
JetIn [62]	Open-SciVis	$1408\times1080\times1100$	1	6.23 GB
Miranda [63]	Open-SciVis	$1024\times1024\times1024$	1	4.00 GB
SynTruss [64]	Open-SciVis	$1200\times1200\times1200$	1	6.42 GB

TABLE II: Real-world HPC datasets used in this work.

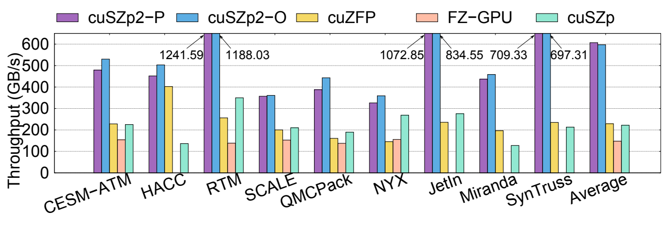
3) Compressor Settings: For cuSZp2, we evaluate both supported lossless encoding modes and denote them as cuSZp2-P (with plain-FLE) and cuSZp2-O (with outlier-FLE). The block size for cuSZp2 is 32 since we find this is the overall best choice in balancing high throughput and high compression ratio. For baseline compressors, we select FZ-GPU [22], cuSZp [23], and cuZFP [21], [28], which are three state-of-the-art pure-GPU lossy compressors. The reasons for such selections are twofold. (1) The throughput of CPU-GPU hybrid compressors (such as cuSZ [18], cuSZx [19], and MGARD-GPU [26]) is highly limited by expensive CPU computations and CPU-GPU data movement overheads, making them impractical for inline compression tasks [11], [12], [31]. (2) Existing studies [22], [23] prove that pure-GPU compressors can outperform hybrid ones for both throughput and compression ratio. For error-bounded compressors (cuSZp2, FZ-GPU, cuSZp), we use value-range-based relative error bound (REL). For example, an error bound REL λ ($\lambda \in (0, 1)$) indicates the difference between each original data point and its corresponding reconstructed one should be smaller than λr, where r denotes the value range of this dataset. Note that cuZFP only supports fixed-rate mode (i.e. preserving a fixed number of bits per data point).

B. Throughput

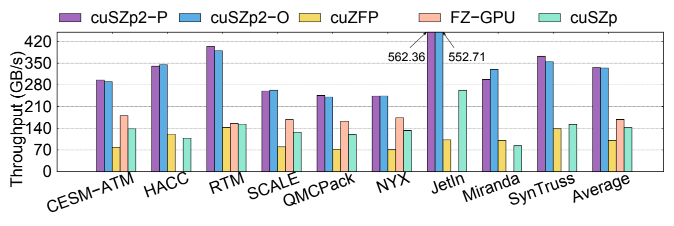
Recall that throughput is the primary concern for a GPU compressor (Section II), we comprehensively evaluate the throughput for cuSZp2 and other baseline compressors in this section. For error-bounded compressors, we evaluate throughput for both compression and decompression with REL 1E-



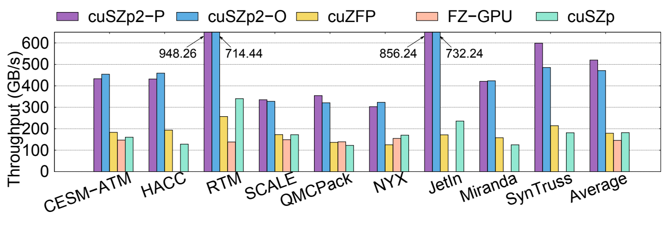
(a) Compression throughput with REL 1E-2 (Fixed-Rate 4 for cuZFP).



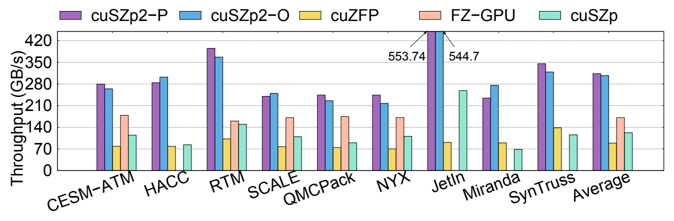
(b) Decompression throughput with REL 1E-2 (Fixed-Rate 4 for cuZFP).



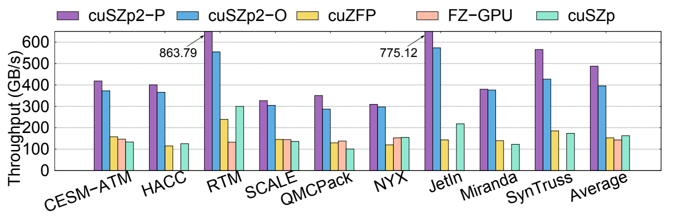
(c) Compression throughput with REL 1E-3 (Fixed-Rate 8 for cuZFP).



(d) Decompression throughput with REL 1E-3 (Fixed-Rate 8 for cuZFP).



(e) Compression throughput with REL 1E-4 (Fixed-Rate 16 for cuZFP).

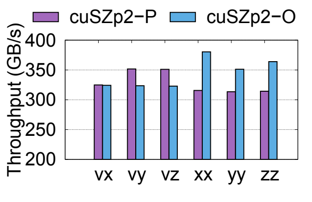


(f) Decompression throughput with REL 1E-4 (Fixed-Rate 16 for cuZFP).

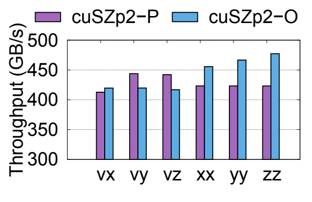
Fig. 14: Compression and decompression throughput evaluation of cuSZp2 and other baseline compressors.

2, REL 1E-3, and REL 1E-4 error bounds. For cuZFP, we measure throughput under fixed rates 4, 8, and 16.

Main Results of Throughput. Figure 14 presents throughput for compression and decompression. *We observe that both* cuSZp2*-P and* cuSZp2*-O consistently outperform all existing pure-GPU compressors in throughput. On average,* cuSZp2*-P achieves compression and decompression throughput of 334.91 and 538.27 GB/s, respectively, while* cuSZp2*-O reaches 329.94 GB/s for compression and 597.29 GB/s for decompression.* These numbers only range from 107.10 (cuZFP compression) to 188.74 GB/s (cuSZp decompression) for other GPU compressors. In JetIn dataset, cuSZp2-P can even reach 1072.85 GB/s for decompression at REL 1E-2. The reason is that JetIn is highly sparse and consists of lots of zero data blocks (i.e. only containing zero values). While processing such blocks, both cuSZp2-P and cuSZp2-O directly flush zero values with `cudaMemset()` instead of performing decompression computations, resulting in exceptionally high throughput. Similarly, since larger error bounds create more zero data blocks, increasing error bounds (e.g. from REL 1E-4 to REL 1E-2) in cuSZp2 leads to higher throughput. We also observe that decompression in cuSZp2 usually has higher throughput than compression. The reason is that cuSZp2 compression requires an extra loop to obtain the lossless encoding information (e.g. fixed-length for each block), whereas decompression can obtain this information by directly reading block offsets from GPU global memory.



(a) Compression



(b) Decompression

Fig. 15: cuSZp2 throughput analysis on 6 fields from HACC.

cuSZp2-O vs cuSZp2-P. As mentioned in Section IV-A, we preserve both lossless encoding methods within cuSZp2 framework due to their pros and cons. cuSZp2-P is optimized for extreme throughput, while cuSZp2-O achieves higher compression ratios with marginally reduced throughput, due to the fine-tuned encoding selection. However, in HACC dataset, we found that cuSZp2-O exhibits higher throughput than cuSZp2-P in both compression and decompression, even with more computations. We evaluate the throughput of all 6 fields of HACC dataset, and the results can be seen in Figure 15. For example, in xx field, cuSZp2-O and cuSZp2-P has 315.64 and 380.36 GB/s compression throughput, respectively. The reason is that HACC is a highly smooth dataset, making the first element very likely to be an outlier, allowing cuSZp2-O to have much higher compression ratios ($\sim 2\times$) than cuSZp2-P. A higher compression ratio indicates fewer amounts of data to be processed, for example, storing fewer compressed bytes

	REL	CESM-ATM	HACC	RTM	SCALE	QMCPack	NYX	JetIn	Miranda	SynTruss
cuSZp2-O	1E-2	18.44∼82.41 **(avg: 42.98)**	11.49∼20.09 **(avg: 15.50)**	30.12∼104.18 **(avg: 61.48)**	16.80∼109.55 **(avg: 46.19)**	12.44∼23.57 **(avg: 18.01)**	14.36∼127.80 (avg: 69.14)	126.28∼126.28 **(avg: 126.28)**	11.10∼11.10 **(avg: 11.10)**	12.96∼12.96 **(avg: 12.96)**
	1E-3	12.99∼57.45 **(avg: 24.53)**	5.85∼12.47 **(avg: 8.82)**	12.00∼84.96 **(avg: 40.24)**	11.10∼79.69 **(avg: 29.52)**	6.07∼13.29 **(avg: 9.68)**	10.50∼125.56 (avg: 41.75)	120.04∼120.06 **(avg: 120.06)**	5.98∼5.98 **(avg: 5.98)**	6.47∼6.47 **(avg: 6.57)**
	1E-4	7.85∼39.01 **(avg: 14.97)**	3.67∼6.27 **(avg: 4.84)**	6.51∼67.81 **(avg: 29.36)**	6.31∼49.95 **(avg: 17.92)**	3.79∼7.25 (avg: 5.52)	5.43∼98.37 **(avg: 24.12)**	106.50∼106.50 **(avg: 106.50)**	3.80∼3.80 **(avg: 3.80)**	4.25∼4.25 **(avg: 4.25)**
FZ-GPU	1E-2	17.62∼100.02 (avg: 40.52)	N.A. (due to bugs)	12.25∼70.09 (avg: 34.60)	16.39∼124.25 (avg: 45.21)	7.53∼19.04 (avg: 13.28)	13.38∼222.62 **(avg: 86.15)**	N.A. (due to bugs)	N.A. (due to bugs)	N.A. (due to bugs)
	1E-3	12.03∼58.03 (avg: 21.57)	N.A. (due to bugs)	6.37∼43.76 (avg: 20.42)	10.89∼69.61 (avg: 25.39)	4.33∼12.08 (avg: 8.20)	9.81∼183.98 **(avg: 42.34)**	N.A. (due to bugs)	N.A. (due to bugs)	N.A. (due to bugs)
	1E-4	7.10∼36.03 (avg: 12.98)	N.A. (due to bugs)	4.02∼30.7 (avg: 13.92)	7.26∼39.22 (avg: 16.16)	2.99∼8.26 **(avg: 5.62)**	5.98∼59.98 (avg: 16.15)	N.A. (due to bugs)	N.A. (due to bugs)	N.A. (due to bugs)
cuSZp	1E-2	3.88∼69.43 (avg: 32.56)	5.28∼10.6 (avg: 7.92)	29.08∼102.73 (avg: 60.10)	3.88∼105.89 (avg: 37.76)	12.44∼22.21 (avg: 17.33)	9.6∼127.8 (avg: 66.73)	126.27∼126.27 (avg: 126.27)	4.46∼4.46 (avg: 4.46)	12.67∼12.67 (avg: 12.67)
	1E-3	2.78∼39.01 (avg: 14.53)	3.45∼5.37 (avg: 4.41)	11.06∼81.90 (avg: 38.43)	2.75∼72.60 (avg: 21.11)	6.08∼10.08 (avg: 8.08)	5.09∼125.55 (avg: 38.44)	119.86∼119.86 (avg: 119.86)	3.04∼3.04 (avg: 3.04)	6.37∼6.37 (avg: 6.37)
	1E-4	2.11∼24.55 (avg: 8.26)	2.53∼3.47 (avg: 3.00)	6.07∼65.04 (avg: 28.04)	2.14∼42.06 (avg: 12.34)	3.79∼5.56 (avg: 4.68)	3.35∼98.23 (avg: 22.14)	105.59∼105.59 (avg: 105.59)	2.32∼2.32 (avg: 2.32)	4.21∼4.21 (avg: 4.21)

TABLE III: Compression ratio of 3 GPU error-bounded lossy compressors. We exclude cuSZp2-P here because it has very close compression ratios with cuSZp due to the same lossless encoding method (i.e. plain fixed-length encoding). Each cell follows a format "min∼max (avg: XX)", and N.A. means "not applicable". The highest average values are **highlighted**.

into global memory in the compression kernel. This reduces the overhead for accessing global memory, increasing runtime throughput in return. We will report more detailed results about compression ratios in Section V-C.

Memory Bandwidth Utilization. In Figure 16, we inspect the GPU memory bandwidth utilization of cuSZp2 and other baseline compressors. Same as the settings in Figure 9, we profile the memory throughput of compression kernels for all compressors on NVIDIA A100 GPU by Nsight Compute. Similar observations can be obtained for decompression as well. As seen, on average, cuSZp2-P and cuSZp2-O achieve global memory throughput of 1175.34 and 1103.45 GB/s, respectively, approaching the hardware limit of 1555 GB/s. In the meantime, this number only ranges from 134.10 (FZ-GPU, due to atomic operations in global synchronization) to 410.90 GB/s (cuSZp, due to strided and scalar-manner memory access patterns). Such results highlight the efficiency of our proposed vectorized memory accesses in cuSZp2 framework.

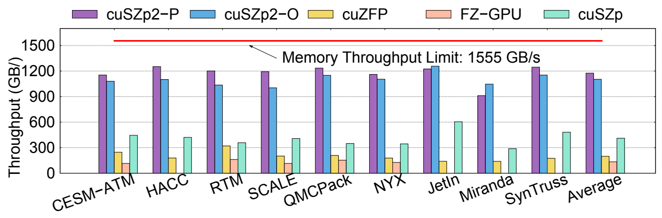


Fig. 16: Memory throughput profiled on NVIDIA A100 GPU.

Examining Latency Control. We also examine the performance of latency control in cuSZp2. Recall that the latency in blockwise compressors is caused by global synchronization (i.e. device-level prefix-sum while concatenating compressed blocks), we measure the throughput of cuSZp2 synchronization with the state-of-the-art synchronization method – single-pass plain chained-scan [23], [52], [53]. As seen in Figure 17, on average, our proposed fine-tuned decoupled lookback in cuSZp2 can achieve 846.85 GB/s in synchronization (2.41× of baseline), hiding latency successfully.

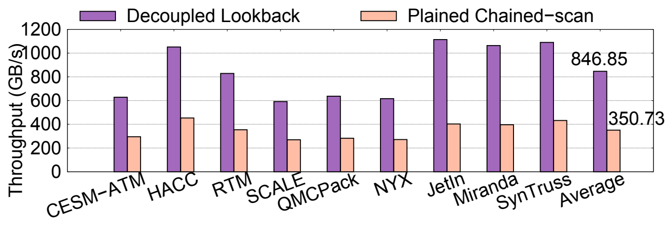


Fig. 17: Evaluating proposed fine-tuned decoupled lookback in cuSZp2 with state-of-the-art synchronization techniques.

Observation I: On average, cuSZp2 delivers 332.42 GB/s and 513.04 GB/s throughput for compression and decompression, which is 2.85× of cuZFP, 2.11× of FZ-GPU, 2.03× of cuSZp, and approximately 200× of existing CPU-GPU hybrid compressors.

C. Compression Ratio

We evaluate compression ratios in this section. cuZFP is excluded because it only supports fixed-rate mode, making the compression ratio on one dataset a fixed number. Each cell in Table III provides details for a specific compressor on a given dataset at an error bound, formatted as "min∼max, (avg)" to display the range and average values. We do not show the compression ratios of cuSZp2-P, because it has similar compression ratios (e.g. less than 0.01% differences) with cuSZp due to the same lossless encoding method (i.e. Plain-

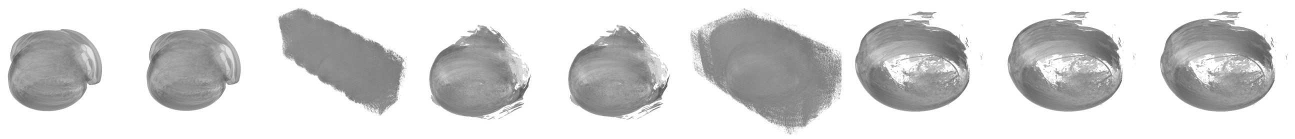

(a) P1000 Ori. (b) P1000 *Ours* (c) P1000 cuZFP (d) P2000 Ori. (e) P2000 *Ours* (f) P2000 cuZFP (g) P3000 Ori. (h) P3000 *Ours* (i) P3000 cuZFP

Fig. 18: Isosurface visualization of all three fields in RTM dataset reconstructed by cuSZp2 (i.e. *Ours* mentioned in (b), (e), and (h) figure captions) and cuZFP under the same compression ratio. For field P1000, P2000, and P3000, the compression ratios of cuSZp2 and cuZFP are configured as ~64, ~30, and ~3, respectively.

FLE). In FZ-GPU, we met bugs in launching `3D-Lorenzo` kernel while compressing several datasets, and we recorded those cells as "N.A." (i.e. not applicable).

Main results of compression ratio. Table III presents compression ratio results for three error-bounded GPU lossy compressors. *As we can see,* cuSZp2*-O outperforms baseline compressors and exhibits the highest compression ratios in 24/27 cases.* In RTM, cuSZp2 delivers around 2× compression ratios of FZ-GPU, since cuSZp2 only preserves one byte for a zero data block, making it exceptionally efficient for compressing sparse datasets. Such observations are also demonstrated in JetIn dataset. While FZ-GPU achieves the best compression ratios in the NYX dataset at REL 1E-2 and REL 1E-3 error bounds, this number in cuSZp2-O is 49.34% higher than that of FZ-GPU under REL 1E-4.

Outlier-FLE vs Plain-FLE. Due to the fine-tuning selection of Outlier-FLE, cuSZp2-O can always have higher compression ratios than cuSZp2-P (see results in cuSZp). In RTM, QMCPack, JetIn, and SynTruss datasets, the benefit of outlier design is relatively modest – under 10%. In those scenarios, cuSZp2-P is a better choice with higher throughput. However, in CESM-ATM, HACC, and Miranda datasets, where data exhibits global smoothness, Outlier-FLE proves to be significantly more effective. In one field of CESM-ATM, cuSZp2-O even achieves a ~6× compression ratio than cuSZp2-P. The drastically optimized compression ratios also reduce memory access overhead when writing compressed data to global memory, explaining why cuSZp2-O sometimes has better throughput than cuSZp2-P.

Observation II: Despite higher throughput, cuSZp2 achieves the best compression ratios in 24/27 cases compared with state-of-the-art error-bounded GPU compressors. While compressing datasets with global smoothness, cuSZp2-O is better than cuSZp2-P.

D. Data Quality

We evaluate reconstructed data quality in this section. In general, there are two metrics to evaluate data quality for a lossy compressor: rate-distortion curve and visualization. Rate-distortion curves quantitatively measure the quality of reconstructed datasets (using metrics like PSNR [65] or SSIM [66]) at the same compression ratios. Higher values of PSNR and SSIM indicate better data quality provided by the compressor.

cuSZp2 vs Error-bounded GPU Compressors. FZ-GPU, cuSZp, and cuSZp2 (including cuSZ) share the same lossy step, which rounds or ceils floating point data into quantization integers. In other words, given the same error bound, these compressors will generate the same reconstructed data – the only difference is the compressed data size, which is decided by different lossless encoding strategies. Since cuSZp2 has proved higher compression ratios under the same error bound (see Table III), we know it should have the best rate-distortion curves among all error-bounded GPU lossy compressors.

cuSZp2 vs cuZFP. Based on orthogonal transformation and embedded coding, cuZFP has been proved to deliver notably high reconstructed data quality in both rate-distortion curves [22], [23] and visualization [28]. However, cuZFP possibly corrupts original data patterns when compression ratios are aggressive. In Figure 18, we reconstruct each field in RTM dataset by cuZFP and cuSZp2 with the same compression and visualize their isosurfaces. In P1000 and P2000, when the compression ratios are ~64 and ~30, cuZFP corrupts the original images (see Figure 18(c) and 18(f)), whereas cuSZp2 almost preserves identical features due to error control (see Figure 18(b) and 18(e)). In P3000, the compression ratio is reduced to ~3, and both cuZFP and cuSZp2 achieve high visualization quality (see Figure 18(h) and 18(i)). This demonstrates cuSZp2's ability in preserving data quality.

Observation III: cuSZp2 preserves high-quality isosurfaces in reconstructed data (even with high compression ratios) and exhibits the best rate-distortion curves among GPU error-bounded lossy compressors.

VI. Discussion of cuSZp2

A. Double-Precision Support

Datasets	Suite	Dims per Field	# Fields	Total Size
S3D [67]	SDRBench	11×500×500×500	5	51.22 GB
NWChem [68]	SDRBench	801,098,891	1	5.96 GB

TABLE IV: Real-world double-precision HPC datasets.

Besides single-precision HPC datasets in Table II, cuSZp2 also supports compression for double-precision datasets. As seen in Table IV, we select NWChem (computational chemistry) and S3D (combustion simulation) from SDRBench [55] for evaluation, and the main results can be found in Figure 19 and Table V. *On average,* cuSZp2*-P achieves a compression* *throughput of 612.83 GB/s and a decompression throughput of 780.33 GB/s, while* cuSZp2*-O records slightly higher rates of 628.54 GB/s for compression and 809.71 GB/s for decompression. This is around 2× of processing single-precision datasets.* The efficiency of handling double-precision formats in cuSZp2 is attributed to its lossy conversion process, which translates both single- and double-precision data points into quantization integers. Since the subsequent lossless encoding computations remain unchanged, this uniform treatment enhances the processing efficiency of double-precision data within cuSZp2. In Table V, we observe cuSZp2-O achieves around 3× compression ratios of cuSZp2-P in S3D at REL 1E-4, and this is because global smoothness exists in S3D dataset, as explained in Section V-C.

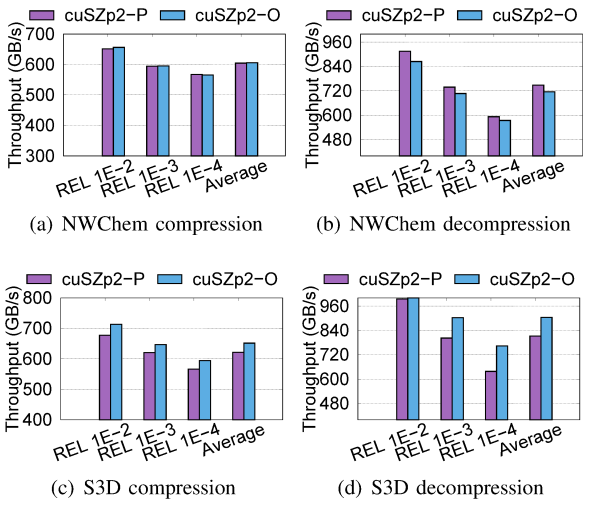


Fig. 19: cuSZp2 throughput for double-precision datasets.

REL		NWChem	S3D		NWChem	S3D
1E-2	cuSZp2-P	82.51	44.28	cuSZp2-O	82.51	89.85
1E-3		26.85	21.71		26.87	56.51
1E-4		13.73	12.64		13.74	37.48

TABLE V: Compression ratios of double-precision datasets.

B. Random Access Support

Since cuSZp2 compresses HPC datasets at block granularity, it supports random accesses to the compressed data. This process can be conducted by reading block offsets and performing global synchronization in the cuSZp2 decompression kernel. In Figure 20, we evaluate the throughput of accessing one arbitrary data block for all datasets in Table II, whereas accessing multiple blocks and random access write have similar results. As seen, cuSZp2 has 1010.07 GB/s throughput on average, varying from 793.14 GB/s in Scale to 1305.32 GB/s in JetIn. This high throughput of random access in cuSZp2 will benefit multiple HPC scenarios, such as cosmic ray propagation [69].

C. Compatibility with Other NVIDIA GPUs

Besides NVIDIA A100 (40 GB) GPU, we also check the compatibility of cuSZp2 and baseline compressors by

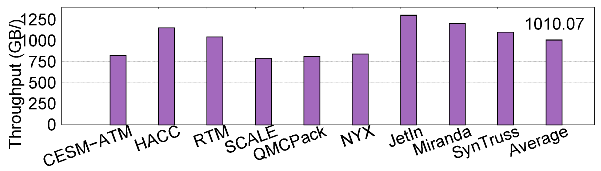


Fig. 20: The throughput of random accessing one arbitrary data block in cuSZp2, measured with REL 1E-4 error bound.

evaluating throughput for other NVIDIA GPUs, including RTX 3090 and RTX 3080 (10 GB) that are commonly used in HPC community [70]–[74]. We use P3000 field in RTM as an example, and similar observations can be obtained in all other datasets. Note that the throughput of all compressors is averaged across three error settings mentioned in Section V-B. As shown in Figure 21, cuSZp2 achieves 232.45 and 405.09 GB/s throughput for compression and decompression on 3090 GPU, and these two numbers are 180.94 and 329.62 GB/s on 3080 GPU. In all, cuSZp2 consistently achieves around 2× throughput than all baseline compressors, demonstrating our optimization is generic across different platforms.

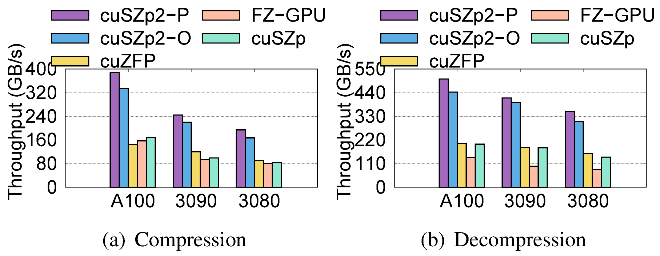


Fig. 21: Evaluating throughput on other NVIDIA GPUs.

D. Rationale for 1D Data Processing

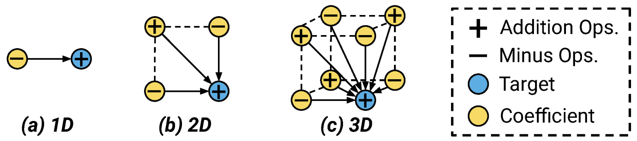


Fig. 22: Explaining 1D, 2D, and 3D first-order difference.

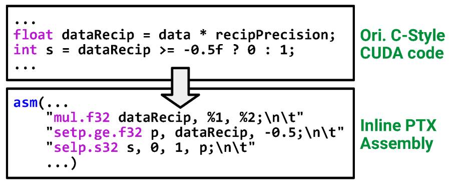
As mentioned in Section III, cuSZp2 processes data in a 1D manner with a first-order difference. However, some other compressors [3], [18], [20], [22] use higher dimensional (e.g. 2D and 3D) differences, also called Lorenzo Prediction, to remove redundant bit patterns in this literature. We illustrate multi-dimensional first-order differences in Figure 22. While 1D processing necessitates computations only with preceding values, 2D and 3D processing involve calculations with 3 and 7 adjacent data points, respectively. Multi-dimension operations not only introduce extra computations and complex partial-sum in decompression but also significantly complicate memory access patterns, downgrading throughput drastically ($>$ 50%) in return. In Table VI, we also implement cuSZp2 with multi-dimensional designs and report their compression ratio. It is true cuSZp2-2D and cuSZp2-3D exhibit higher compression ratios in some cases, however, this benefit becomes limited for non-sparse dataset (e.g. P3000) with conservative error bounds (e.g. REL 1E-3 and 1E-4). This demonstrates the rationale of 1D processing in cuSZp2. The 1D design is also observed in an industry-level closed-source compressor [75].

	REL	P1000	P2000	P3000
cuSZp2-1D	1E-2	158.12	49.61	27.53
	1E-3	110.05	22.38	11.19
	1E-4	78.97	12.72	6.11
cuSZp2-2D	1E-2	176.65	65.72	34.26
	1E-3	124.80	25.21	11.29
	1E-4	92.04	14.31	6.22
cuSZp2-3D	1E-2	176.11	66.26	33.65
	1E-3	125.14	24.87	10.96
	1E-4	93.03	14.38	6.19

TABLE VI: Explaining the rationale of 1D data processing by compressing 3 RTM fields with multi-dimensional cuSZp2 (with outlier encoding). To be fair, the block size for 1D, 2D, and 3D cuSZp2 are 64, 8×8 (=64), and 4×4×4 (=64).

E. Breakdown Throughput Gain Analysis

To quantify the throughput gains, we conduct an ablation study by individually disabling each throughput-related factor in cuSZp2 and assessing the impact. On average, memory optimization and latency hiding contribute to the throughput gains by 56.23% and 41.29%, respectively. In the meanwhile, cuSZp2 is equipped with other optimizations, such as inline parallel thread execution (PTX) assembly and loop unrolling. While the former is a CUDA feature that allows assembly-like instructions to be inlined directly within CUDA code (one example see Figure 23), the latter reduces the number of loop iterations by multiple copies within the loop body. Both target better utilization of GPU hardware resources, to improve cuSZp2 runtime throughput. However, based on our testing, their benefits are trivial (<3%) compared to the primary designs and will not be discussed in details here.



```
...
float dataRecip = data * recipPrecision;
int s = dataRecip >= -0.5f ? 0 : 1;
...
```

```
asm(...
    "mul.f32 dataRecip, %1, %2;\n\t"
    "setp.ge.f32 p, dataRecip, -0.5;\n\t"
    "selp.s32 s, 0, 1, p;\n\t"
    ...)
```

Fig. 23: Illustration of inline PTX assembly. This code segment is from the Lossy Conversion stage (i.e. ❶ in Figure 4).

VII. Related Works

A. Error-bounded Lossy Compression

Error-bounded lossy compression, with much higher compression ratios than lossless ones, is proposed to benefit HPC by introducing user-controllable errors [2]–[4], [28], [57], [76]–[79]. Among such, two notable lossy compressors are SZ [2]–[4] and ZFP [28]. SZ is a prediction-based compressor composed of four major steps: data prediction (e.g. Lorenzo [3], Regression [4], [80], and Interpolation [81] predictions), linear-scale quantization, variable-length encoding, and dictionary encoding. ZFP is a transform-based compressor that processes at block granularity, involving various alignments, orthogonal transforms, and embedded encoding.

B. Data Compression on NVIDIA GPU

Data compression within NVIDIA GPU can achieve much higher throughput and is hence widely explored in HPC community in the past decade [18]–[20], [22], [23], [82], [83]. O'Neil et al. [82] proposed GFC, a lossless compressor for double-precision datasets, achieving 75 GB/s throughput at NVIDIA FX 5800 GPU. Lindstrom [28] implemented ZFP into GPU [21], preserving high throughput and promising visualization quality. Tian et al. [18] proposed cuSZ, the first prediction-based GPU error-bounded lossy compressor. Zhang et al. [22] improved cuSZ with pure-GPU designs and a novel lossless encoding method, balancing between throughput and compression ratio. While existing works suffer from either limited throughput or compatibility, cuSZp2 achieves both with optimized compression ratios and high data quality.

C. Compression on Other Heterogeneous Processors

Other than NVIDIA GPUs, data compression algorithms are also implemented in other heterogeneous processors, such as AMD GPU, FPGA, and DPU, to assist multiple HPC scenarios [84]–[90]. Tian et al. [84] explored a hardware and algorithm co-design and proposed waveSZ to manage data within FPGA. Tavana et al. [88] adopts data compression on a multi-AMD-GPU system to improve both energy and performance efficiency. There are some other compressors [85], [86] that are based on emerging AI chips – Cerebras [91]. Song et al. [85] enabled and scaled an error-bounded lossy compression algorithm on the Cerebras CS-2 system. Although it achieves promising throughput (around 500 GB/s), its compatibility with existing HPC simulations and machine learning frameworks remains an open question.

VIII. Conclusion

In this work, we propose cuSZp2, a GPU error-bounded lossy compressor targeting extreme throughput and optimized compression ratios. Specifically, cuSZp2 compressor features outlier fixed-length encoding, vectorized memory access patterns, and latency control with decoupled lookback. Based on evaluations from 9 real-world HPC datasets, on average, cuSZp2 exhibits 332.42 and 513.04 GB/s end-to-end throughput on NVIDIA A100 GPU for compression and decompression, respectively. Compared with state-of-the-art pure-GPU compressors such as cuZFP, FZ-GPU, and cuSZp, cuSZp2 achieves ∼2× throughput and the highest compression ratios.

Acknowledgment

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357 and DE-SC0024559. The material was also supported by the National Science Foundation under Grant OAC-2003709, OAC-2104023, OAC-2311875, and OAC-2211538. The experimental resource for this paper was provided by the Laboratory Computing Resource Center on the Swing cluster at Argonne National Laboratory.

References

[1] S. W. Son, Z. Chen, W. Hendrix, A. Agrawal, W.-k. Liao, and A. Choudhary, “Data compression for the exascale computing era-survey,” *Supercomputing frontiers and innovations*, vol. 1, no. 2, pp. 76–88, 2014.

[2] S. Di and F. Cappello, “Fast error-bounded lossy HPC data compression with SZ,” in *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 730–739.

[3] D. Tao, S. Di, Z. Chen, and F. Cappello, “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, 2017, pp. 1129–1139.

[4] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, “Error-controlled lossy compression optimized for high compression ratios of scientific datasets,” in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 438–447.

[5] S. Jin, J. Pulido, P. Grosset, J. Tian, D. Tao, and J. Ahrens, “Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling,” in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 45–56.

[6] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, “Full-state quantum circuit simulation by using data compression,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–24.

[7] M. Dmitriev, T. Tonellot, H. AlSalem, and S. Di, “Error-bounded lossy compression in reverse time migration,” in *Sixth EAGE High Performance Computing Workshop*, vol. 2022, no. 1. European Association of Geoscientists & Engineers, 2022, pp. 1–5.

[8] Y. Huang, K. Zhao, S. Di, G. Li, M. Dmitriev, T.-L. D. Tonellot, and F. Cappello, “Towards improving reverse time migration performance by high-speed lossy compression,” in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2023, pp. 651–661.

[9] Large language models - the hardware connection. [Online]. Available: https://community.juniper.net/blogs/sharada-yeluri/2023/10/03/large-language-models-the-hardware-connection

[10] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, “Use cases of lossy compression for floating-point data in scientific data sets,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019.

[11] G. Marcus, Y. Ding, P. Emma, Z. Huang, J. Qiang, T. Raubenheimer, M. Venturini, and L. Wang, “High fidelity start-to-end numerical particle simulations and performance studies for lcls-ii,” in *Proceedings, 37th International Free Electron Laser Conference (FEL 2015): Daejeon, Korea, August 23-28*, 2015.

[12] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su *et al.*, “General atomic and molecular electronic structure system,” *Journal of computational chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[13] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, “Hacc: Extreme scaling and performance across diverse architectures,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.

[14] Ncar: Community earth system model. [Online]. Available: https://www.cesm.ucar.edu/

[15] Meta: Llama. [Online]. Available: https://llama.meta.com/

[16] Nvidia a100 gpu white paper. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

[17] Pcie 3.0 white paper. [Online]. Available: https://www.intel.com.ec/content/dam/doc/white-paper/pci-express3-accelerator-white-paper.pdf

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

[19] X. Yu, S. Di, K. Zhao, J. Tian, D. Tao, X. Liang, and F. Cappello, “Ultrafast error-bounded lossy compression for scientific datasets,” in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 159–171.

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

[21] P. Lindstrom, “cuzfp,” https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp.

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

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

[24] V. Volkov, *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

[25] D. Merrill and M. Garland, “Single-pass parallel prefix scan with decoupled look-back,” *NVIDIA, Tech. Rep. NVR-2016-002*, 2016.

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

[27] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom, “Error analysis of zfp compression for floating-point data,” *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. A1867–A1898, 2019.

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

[29] J. Thayer, D. Damiani, C. Ford, M. Dubrovin, I. Gaponenko, C. O’Grady, W. Kroeger, J. Pines, T. Lane, A. Salnikov *et al.*, “Data systems for the linac coherent light source,” *Advanced structural and chemical imaging*, vol. 3, pp. 1–13, 2017.

[30] W. Xu, Y. Zhang, and X. Tang, “Parallelizing dnn training on gpus: Challenges and opportunities,” in *Companion Proceedings of the Web Conference 2021*, 2021, pp. 174–178.

[31] Y. Ko, K. Choi, J. Seo, and S.-W. Kim, “An in-depth analysis of distributed training of deep neural networks,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 994–1003.

[32] C. Dun, C. R. Wolfe, C. M. Jermaine, and A. Kyrillidis, “Resist: Layer-wise decomposition of resnets for distributed training,” in *Uncertainty in Artificial Intelligence*. PMLR, 2022, pp. 610–620.

[33] S. C. Gundabolu, T. Vijaykumar, and M. Thottethodi, “Fastz: accelerating gapped whole genome alignment on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[34] M. G. Awan, S. Hofmeyr, R. Egan, N. Ding, A. Buluc, J. Deslippe, L. Oliker, and K. Yelick, “Accelerating large scale de novo metagenome assembly using gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–11.

[35] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Y. Huang, K. Raffenetti, H. Zhou, K. Zhao, X. Lu *et al.*, “gzccl: Compression-accelerated collective communication framework for gpu clusters,” in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 437–448.

[36] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Y. Huang, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen *et al.*, “Poster: Optimizing collective communications with error-bounded lossy compression for gpu clusters,” in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 454–456.

[37] Q. Zhou, C. Chu, N. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, “Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 444–453.

[38] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. Bates, G. Danabasoglu, J. Edwards *et al.*, “The community

earth system model (cesm) large ensemble project: A community resource for studying climate change in the presence of internal climate variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333–1349, 2015.

[39] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley *et al.*, "Qmcpack: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed Matter*, vol. 30, no. 19, p. 195901, 2018.

[40] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 133–137.

[41] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus," in *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2020, pp. 376–388.

[42] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[43] E. Baysal, D. D. Kosloff, and J. W. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.

[44] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 132–143, 2006.

[45] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu, "Flexvec: Auto-vectorization for irregular loops," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 697–710.

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

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

[48] A. Balevic, "Parallel variable-length encoding on gpgpus," in *Euro-Par 2009–Parallel Processing Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25-28, 2009, Revised Selected Papers 15*. Springer, 2010, pp. 26–35.

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

[50] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[51] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 96–107.

[52] S. Yan, G. Long, and Y. Zhang, "Streamscan: fast scan algorithms for gpus without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 229–238.

[53] S. Maleki and M. Burtscher, "Automatic hierarchical parallelization of linear recurrences," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 128–138, 2018.

[54] Argonne swing cluster. [Online]. Available: https://www.lcrc.anl.gov/systems/swing

[55] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "Sdrbench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE international conference on big data (Big Data)*. IEEE, 2020, pp. 2716–2724.

[56] P. Klacansky, "Open scivis datasets," December 2017, `https://klacansky.com/open-scivis-datasets/`. [Online]. Available: https://klacansky.com/open-scivis-datasets/

[57] S. Li, P. Lindstrom, and J. Clyne, "Lossy scientific data compression with sperr," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 1007–1017.

[58] T. Lu, Y. Zhong, Z. Sun, X. Chen, Y. Zhou, F. Wu, Y. Yang, Y. Huang, and Y. Yang, "Adt-fse: A new encoder for sz," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[59] A. Rodriguez, N. Azami, and M. Burtscher, "Adaptive per-file lossless compression of floating-point data."

[60] G.-Y. Lien, T. Miyoshi, S. Nishizawa, R. Yoshida, H. Yashiro, S. A. Adachi, T. Yamaura, and H. Tomita, "The near-real-time scale-letkf system: A case of the september 2015 kanto-tohoku heavy rainfall," *Sola*, vol. 13, pp. 1–6, 2017.

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

[62] R. W. Grout, A. Gruber, H. Kolla, P.-T. Bremer, J. Bennett, A. Gyulassy, and J. Chen, "A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates," *Journal of Fluid Mechanics*, vol. 706, pp. 351–383, 2012.

[63] A. W. Cook, W. Cabot, and P. L. Miller, "The mixing transition in rayleigh–taylor instability," *Journal of Fluid Mechanics*, vol. 511, pp. 333–362, 2004.

[64] P. Klacansky, H. Miao, A. Gyulassy, A. Townsend, K. Champley, J. Tringe, V. Pascucci, and P.-T. Bremer, "Virtual inspection of additively manufactured parts," in *2022 IEEE 15th Pacific Visualization Symposium (PacificVis)*. IEEE, 2022, pp. 81–90.

[65] A. Hore and D. Ziou, "Image quality metrics: Psnr vs. ssim," in *2010 20th international conference on pattern recognition*. IEEE, 2010, pp. 2366–2369.

[66] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[67] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki *et al.*, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001, 2009.

[68] E. Apra, E. Bylaska, W. de Jong, N. Govind, K. Kowalski, T. Straatsma, M. Valiev, H. van Dam, Y. Alexeev, J. Anchell *et al.*, "Nwchem," 2020.

[69] G. Müller, "Static multiresolution grids with inline hierarchy information for cosmic ray propagation," *Journal of Cosmology and Astroparticle Physics*, vol. 2016, no. 08, p. 025, 2016.

[70] F. Knorr, P. Thoman, and T. Fahringer, "ndzip-gpu: efficient lossless compression of scientific floating-point data on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[71] F. Zhang, Y. Hu, H. Ding, Z. Yao, Z. Wei, X. Zhang, and X. Du, "Optimizing random access to hierarchically-compressed data on gpu," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[72] S. Song and P. Jiang, "Rethinking graph data placement for graph neural network training on multiple gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–10.

[73] S. Li, F. Tu, L. Liu, J. Lin, Z. Wang, Y. Kang, Y. Ding, and Y. Xie, "Ecssd: Hardware/data layout co-designed in-storage-computing architecture for extreme classification," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[74] W. Chen, Z. Mo, H. Xu, K. Ye, and C. Xu, "Interference-aware multiplexing for deep learning in gpu clusters: A middleware approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[75] Nvcomp. [Online]. Available: https://github.com/NVIDIA/nvcomp

[76] X. Liang, S. Di, S. Li, D. Tao, B. Nicolae, Z. Chen, and F. Cappello, "Significantly improving lossy compression quality based on an optimized hybrid prediction model," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–26.

[77] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—the univariate case," *Computing and Visualization in Science*, vol. 19, no. 5, pp. 65–76, 2018.

[78] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao *et al.*, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, vol. 9, no. 2, pp. 485–498, 2022.

[79] S. Di, J. Liu, K. Zhao, X. Liang, R. Underwood, Z. Zhang, M. Shah, Y. Huang, J. Huang, X. Yu *et al.*, "A survey on error-bounded lossy compression for scientific datasets," *arXiv preprint arXiv:2404.02840*, 2024.

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

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

[82] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a gpu," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–7.

[83] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "Mpc: a massively parallel compression algorithm for scientific data," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 381–389.

[84] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "Wavesz: A hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 74–88.

[85] S. Song, Y. Huang, P. Jiang, W. Zheng, S. Di, Q. Cao, Y. Feng, Z. Xie, and F. Cappello, "Ceresz: Enabling and scaling error-bounded lossy compression on cerebras cs-2," in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, 2024.

[86] H. Ltaief, Y. Hong, L. Wilson, M. Jacquelin, M. Ravasi, and D. E. Keyes, "Scaling the "memory wall" for multi-dimensional seismic processing with algebraic compression on cerebras cs-2 systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–12.

[87] Y. Zhou, F. Zhang, T. Lin, Y. Huang, S. Long, J. Zhai, and X. Du, "F-tadoc: Fpga-based text analytics directly on compression with hls," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 3739–3752.

[88] M. K. Tavana, Y. Sun, N. B. Agostini, and D. Kaeli, "Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-gpu systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 664–674.

[89] Y. Huang, S. Guo, S. Di, G. Li, and F. Cappello, "Mitigating silent data corruptions in hpc applications across multiple program inputs," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–14.

[90] Y. Li, A. Kashyap, W. Chen, Y. Guo, and X. Lu, "Accelerating lossy and lossless compression on emerging bluefield dpu architectures," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 373–385.

[91] Cerebras ai chips. [Online]. Available: https://www.cerebras.net/

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. *Paper's Main Contributions*

C_1 We summarize the drawbacks of existing GPU lossy compressors and redefine the appropriate metrics for evaluating lossy compressors on GPU.

C_2 We propose and implement a new GPU error-bounded lossy compressor called cuSZp2[1], with extreme throughput and optimized compression ratios, by integrating a novel lossless encoding method, optimized memory access patterns, and latency control.

C_3 We evaluate cuSZp2 with three state-of-the-art pure GPU compressors from three perspectives, including throughput, compression ratios, and data quality.

C_4 We propose four important use cases and evaluate cuSZp2, including double-precision support, random access support, lower-end GPU support, and a multi-dimensional version of cuSZp2.

B. *Computational Artifacts*

The AD/AE version of the artifact can be downloaded through the following Links.

A_1 https://github.com/hyfshishen/SC24-cuSZp2[2] and its corresponding generated persistent Zenodo DOI https://zenodo.org/doi/10.5281/zenodo.13315525.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Sec. II
A_1	C_2	Sec. IV
A_1	C_3	Sec. V
A_1	C_4	Sec. VI

II. Artifact Identification

A. *Computational Artifact* A_1

<u>*Relation To Contributions*</u>

The artifact includes the source code of the proposed ultra-fast error-bounded GPU compressor – cuSZp2. All four major contributions identified in this work are based on the proposed cuSZp2, including (1) strict throughput measurement, (2) optimized implementation, (3) comprehensive measurements, and (4) proposed use cases (e.g. random access support).

[1]cuSZp2 follows a similar 4-stage compression pipeline compared with cuSZp – so that we decided to name it as cuSZp2. However, the implementation and algorithm of each stage in cuSZp2 are significantly different.

[2]This repository is only for AD/AE purposes (e.g. including the execution script for each discussion subsection). The publicly available version for cuSZp2 can be found in GitHub Link: https://github.com/szcompressor/cuSZp.

<u>*Expected Results*</u>

After compiling the cuSZp2 code and executing the generated binary, the results will be printed in the command line environment. Note that we integrate the time measurement inside the code. All results should be consistent with what is reported in the paper. Specifically:

1) The best throughput of all existing GPU compressors.
2) Higher compression ratios than FZ-GPU and cuSZp.
3) Better isosurface visualization compared with cuZFP.

Note that all baseline compressors, including FZ-GPU, cuSZp, and cuZFP, along with adopted HPC datasets (SDRBench and Open-SciVis), are publicly available.

<u>*Expected Reproduction Time*</u>

The major goal of GPU compression is throughput, so a single pass for compression and decompression in cuSZp2 on all datasets can be executed within several seconds. However, compilation, measurement, and visualization may take different efforts. These efforts are explained below:

- Compilation: *Several minutes* for cuSZp2 repository.
- Throughput: *Several seconds for one compressor on one field of one dataset* (including passing data to GPU, kernel execution, and printing all required information).
- Compression Ratio: *Several seconds for one compressor on one field of one dataset*. The same as throughput.
- Data Quality: We visualize the isosurface to evaluate reconstructed data quality using Mayavi, a Python-based tool. One RTM field may take around *10 minutes*.

Artifact Evaluation (AE)

In this section, we will provide the detailed steps to reproduce paper results. The required code is updated in a GitHub repository mentioned before. For simplicity, we will use the name GSZ (the name of cuSZp2 in the paper submission version) in this section – they here refer to the same thing. For the two encoding mode, GSZ-P and GSZ-O denote, GSZ with plain and outlier fixed-length encoding, respectively.

<u>*Artifact Setup (incl. Inputs)*</u>

■ *Hardware:* Most evaluations are conducted on NVIDIA Ampere A100 GPU (40 GB). The compatibility for lower-end GPU (in the Discussion Section) requires NVIDIA RTM 3090 and NVIDIA RTM 3080 (10 GB) GPUs.

■ *Software:*

- Git 2.15 or newer
- CMake 3.21 or newer
- CUDA Toolkit 11.0 or newer
- No requirement for GCC, better with 7.0 and newer
- Python3 and Mayavi Package.

■ *Datasets:* The metadata (including dimension) of each dataset can be found in Table I. Downloading information for each dataset can be found in the bullet point below:

- SDRBench datasets can be found in LINK.
  - CESM-ATM: Download-Link
  - HACC: Download-Link
  - RTM: We exclude RTM due to confidential issues.
  - SCALE: Download-Link
  - QMCPack: Download-Link
  - NYX: Download-Link
- Open-SciVis datasets can be found in LINK.
  - JetIn: Download-Link
  - Miranda: Download-Link
  - SynTruss: Download-Link

The dataset can be downloaded using `wget` command.

Datasets	**Suite**	**Dims per Field**	**# Fields**	**Total Size**
CESM-ATM	SDRBench	3600×1800×26	33	20.71 GB
HACC	SDRBench	1,073,726,487	6	23.99 GB
RTM	SDRBench	1008×1008×352	3	3.99 GB
SCALE	SDRBench	1200×1200×98	12	6.31 GB
QMCPack	SDRBench	69×69×33120	2	1.17 GB
NYX	SDRBench	512×512×512	6	3.00 GB
JetIn	Open-SciVis	1408×1080×1100	1	6.23 GB
Miranda	Open-SciVis	1024×1024×1024	1	4.00 GB
SynTruss	Open-SciVis	1200×1200×1200	1	6.42 GB

TABLE I
METADATA FOR REAL-WORLD HPC DATASETS USED IN THIS WORK.

■ *Installation and Deployment of cuSZp2:*

```
# Step 1: Download cuSZp2 source code
git clone https://github.com/hyfshishen/SC24-cuSZp2.git

# Step 2: Go to target building directory
cd SC24-cuSZp2/main-results && mkdir build && cd build

# Step 3: Prepare makefile using CMake.
cmake -DCMAKE_BUILD_TYPE=Release \
      -DCMAKE_INSTALL_PREFIX=../install ..

# Step 4: Make and Install
make -j && make install
```

You can see two executable binary `gsz_p` and `gsz_o` generated in folder `main-results/install/bin/`. These two executable binary represent GSZ-P and GSZ-O mentioned in paper (for Figure 14, Table III, and Figure 21).

Artifact Execution

We use HACC dataset and GSZ-P as an example. GSZ-O will be executed in the exactly same way. Besides, since all fields in one dataset will exhibit similar throughput and consistent compression ratios, so executing one field to showcase the results and compressibility of GSZ compressor.

Given an error bound `REL 1E-3` and field `vx.f32`, GSZ-P can compress it by command:

```
cd main-results/install/bin/

./gsz_p vx.f32 1e-3
# 1e-3 here denotes the relative error bound;
# you can also set it as 0.001.
```

After that, you can see output as below:

```
GSZ finished!
GSZ compression   end-to-end speed: 359.554510 GB/s
GSZ decompression end-to-end speed: 437.775719 GB/s
GSZ compression ratio: 5.365436

Pass error check!
```

- The compression end-to-end speed (i.e. throughput) reflects to the HACC bar mentioned in Figure-14-(c).
- The decompression end-to-end speed (i.e. throughput) reflects to the HACC bar mentioned in Figure-14-(d).
- The compression ratio is reported in Table III.
- If you are executing those scripts in other GPUs, such as 3080 and 3080. The throughput that reported denote Figure-21.
- The Pass error check! is the interal error bound checking.

Other datasets and GSZ-O will work in the same way.

Artifact Oneline Execution

##### Reproducing MAIN Results #######

In this part, we can reproduce all experiments about Figure 14 and Table III with several wrap-up python (version 3+) command lines, including:

- Dataset prepartion.
- GSZ compilation.
- Execution and results observation.

The three procedures are described in the code block below.

```
# Step 1: Dataset preparation
cd SC24-cuSZp2/
python dataset-preparation.py
# After that, all datasets will be prepared in the folder
# SC24-cuSZp2/dataset, and we can go to the next step.

# Step 2: GSZ compilation
cd SC24-cuSZp2/main-results
python 0-compilation.py
# After that, the compilation of GSZ will be finished,
and we can go to the next step (execution).

# Step 3: GSZ execution
cd SC24-cuSZp2/main-results # The same folder as Step 2.
python 1-execution.py ERROR-BOUND-YOU-WANT-TO-EXECUTE
#   python 1-execution.py 1E-2
#   python 1-execution.py 1E-3
#   python 1-execution.py 1E-4
```

After the execution, you can observe an output. We use `python 1-execution.py 1E-3` to understand such output. After that, you can see a generated output as shown in the following code block.

```
=======================================================
Done with Execution GSZ-P and GSZ-O on cesm_atm under 1e-3
GSZ-P   compression throughput: 267.28176896969694 GB/s
GSZ-P decompression throughput: 395.9575038787878 GB/s
GSZ-P    max compression ratio: 39.039537
GSZ-P    min compression ratio: 2.776141
GSZ-P    avg compression ratio: 14.53542281818182

GSZ-O   compression throughput: 256.82364506060605 GB/s
GSZ-O decompression throughput: 409.47252312121213 GB/s
GSZ-O    max compression ratio: 57.453092
```

```
GSZ-O    min compression ratio: 12.995819
GSZ-O    avg compression ratio: 24.53496509090909
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on hacc under 1e-3
GSZ-P   compression throughput: 339.03042400000004 GB/s
GSZ-P decompression throughput: 431.46155999999996 GB/s
GSZ-P    max compression ratio: 5.365436
GSZ-P    min compression ratio: 3.451861
GSZ-P    avg compression ratio: 4.405594000000001

GSZ-O   compression throughput: 344.9251053333334 GB/s
GSZ-O decompression throughput: 459.2428156666667 GB/s
GSZ-O    max compression ratio: 12.470066
GSZ-O    min compression ratio: 5.851711
GSZ-O    avg compression ratio: 8.823446833333334
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on scale under 1e-3
GSZ-P   compression throughput: 240.40884108333333 GB/s
GSZ-P decompression throughput: 335.2613445 GB/s
GSZ-P    max compression ratio: 72.598979
GSZ-P    min compression ratio: 2.750328
GSZ-P    avg compression ratio: 21.11330458333333

GSZ-O   compression throughput: 250.15284741666665 GB/s
GSZ-O decompression throughput: 316.6550965833334 GB/s
GSZ-O    max compression ratio: 79.695224
GSZ-O    min compression ratio: 11.102816
GSZ-O    avg compression ratio: 29.52363491666667
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on qmcpack under 1e-3
GSZ-P   compression throughput: 236.19716549999998 GB/s
GSZ-P decompression throughput: 315.8376475 GB/s
GSZ-P    max compression ratio: 10.075567
GSZ-P    min compression ratio: 6.076028
GSZ-P    avg compression ratio: 8.0757975

GSZ-O   compression throughput: 183.3123765 GB/s
GSZ-O decompression throughput: 319.8198355 GB/s
GSZ-O    max compression ratio: 13.296692
GSZ-O    min compression ratio: 6.077027
GSZ-O    avg compression ratio: 9.6868595
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on nyx under 1e-3
GSZ-P   compression throughput: 244.1977426666667 GB/s
GSZ-P decompression throughput: 305.4707613333333 GB/s
GSZ-P    max compression ratio: 125.551299
GSZ-P    min compression ratio: 5.090097
GSZ-P    avg compression ratio: 38.44212666666666

GSZ-O   compression throughput: 244.53299016666665 GB/s
GSZ-O decompression throughput: 326.64762433333334 GB/s
GSZ-O    max compression ratio: 125.560284
GSZ-O    min compression ratio: 10.501972
GSZ-O    avg compression ratio: 41.756694333333336
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on jetin under 1e-3
GSZ-P   compression throughput: 559.848637 GB/s
GSZ-P decompression throughput: 2626.161979 GB/s
GSZ-P    max compression ratio: 119.858277
GSZ-P    min compression ratio: 119.858277
GSZ-P    avg compression ratio: 119.858277

GSZ-O   compression throughput: 554.342144 GB/s
GSZ-O decompression throughput: 2658.088986 GB/s
GSZ-O    max compression ratio: 120.064674
GSZ-O    min compression ratio: 120.064674
GSZ-O    avg compression ratio: 120.064674
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on miranda under 1e-3
GSZ-P   compression throughput: 297.81917 GB/s
GSZ-P decompression throughput: 420.187394 GB/s
GSZ-P    max compression ratio: 3.038741
GSZ-P    min compression ratio: 3.038741
GSZ-P    avg compression ratio: 3.038741

GSZ-O   compression throughput: 330.657079 GB/s
GSZ-O decompression throughput: 423.426028 GB/s
GSZ-O    max compression ratio: 5.981446
GSZ-O    min compression ratio: 5.981446
GSZ-O    avg compression ratio: 5.981446
====================================================

====================================================
Done with Execution GSZ-P and GSZ-O on syntruss under 1e-3
GSZ-P   compression throughput: 319.446766 GB/s
GSZ-P decompression throughput: 317.919891 GB/s
GSZ-P    max compression ratio: 6.371377
GSZ-P    min compression ratio: 6.371377
GSZ-P    avg compression ratio: 6.371377

GSZ-O   compression throughput: 354.674088 GB/s
GSZ-O decompression throughput: 485.769935 GB/s
GSZ-O    max compression ratio: 6.470316
GSZ-O    min compression ratio: 6.470316
GSZ-O    avg compression ratio: 6.470316
====================================================
```

To understand such results.

- ”1E-3” denotes throughput in Fig.14 (c) and (d) (whereas ”1E-2” denote (a) and (b), ”1E-4” denote (e) and (f)).
- For the throughput, you may observe a similar number as the bar shown in Figure 14.
- For the compression ratio, you may observe the number as reported in Table III.
- For the baseline compressors, our evaluations are consistent with existing works (e.g. cuSZp and FZ-GPU). So that they can be directly found in Figure itself.

##### Reproducing RTM Results ########

In this part, we can reproduce all experiments related to RTM dataset within just several scripts. Note that the link to this dataset is not directly provided in this repository due to confidential issues – it can only be accessed internally in the AD-AE discussion. Assuming we already have our datasets, the execution step of this phase includes:

- GSZ compilation.
- Execution and results observation. (this step can reproduce the results about throughput and compression ratio in Figure.14 and Table.III)

Since RTM dataset only has three fields: pressure_1000, pressure_2000, and pressure_3000, the dataset preparation steps are described in the text below. In all, the execution to reproduce all experiments is shown as the code block below.

```
# Step 0: Dataset preparation
cd SC24-cuSZp2/rtm-evaluation-results/
# Download pressure_1000, pressure_2000,
# and pressure_3000 manually from Google Drive.
# After that, when you list all files in this folder,
# all files should be arranged as below.
ls
1-compilation.py  2-execution.py  3-visualization.py  cmake
CMakeLists.txt  Config.cmake.in  examples  include
pressure_1000  pressure_2000  pressure_3000  src

# Step 1: GSZ compilation
python 1-compilation.py
# After that, the compilation of GSZ will be finished,
# and we can go to the next step (execution).

# Step 3: GSZ execution
python 2-execution.py
# After that, both GSZ-P and GSZ-O compression will be
# conducted under the error bound 1E-2, 1E-3, and 1E-4.
```

After the execution, results similar to the code block shown below can be seen (such results are still measured on A100).

```
=====================================
GSZ-O 1E-3 Execution on Pressure_1000
=====================================
GSZ finished!
GSZ compression   end-to-end speed: 469.758409 GB/s
GSZ decompression end-to-end speed: 1146.214499 GB/s
GSZ compression ratio: 84.968878

Pass error check!

=====================================
GSZ-O 1E-3 Execution on Pressure_2000
=====================================
GSZ finished!
GSZ compression   end-to-end speed: 399.663872 GB/s
GSZ decompression end-to-end speed: 625.772303 GB/s
GSZ compression ratio: 23.767280

Pass error check!

=====================================
GSZ-O 1E-3 Execution on Pressure_3000
=====================================
GSZ finished!
GSZ compression   end-to-end speed: 336.690098 GB/s
GSZ decompression end-to-end speed: 464.315184 GB/s
GSZ compression ratio: 12.002271

Pass error check!
```

##### Reproducing Double-Precision #####

In this part, we can reproduce all experiments related to double-precision datasets within just several scripts. Note that the data preparation step may take some time since S3D dataset is more than 50 GB, and downloading it may take some time. More information about the evaluated two double-precision datasets (both are from SDRBench) can be found in the table below.

It is worth mentioning that, similar to previous sections, all provided scripts executed in Python are still under Python 3.0+. Specifically, the scripts include:

Datasets	Suite	Dims per Field	# Fields	Total Size
S3D	SDRBench	$11\times500\times500\times500$	5	51.22 GB
NWChem	SDRBench	801,098,891	1	5.96 GB

TABLE II
REAL-WORLD DOUBLE-PRECISION HPC DATASETS.

- Dataset prepartion.
- GSZ compilation.
- Execution and results observation. (this step can reproduce the results)

Specifically, the first three procedures are explained in the code block below.

```
# Step 1: Dataset preparation
cd SC24-cuSZp2/double-precision-results
python 0-dataset-preparation.py
# This step may take some time,
# since s3d dataset is more than 50 GB.

# Step 2: GSZ compilation
python 1-compilation.py
# After that, the compilation of GSZ will be finished,
# and we can go to the next step (execution).

# Step 3: GSZ execution
python 2-execution.py ERROR-BOUND-YOU-WANT-TO-EXECUTE
# There are 3 error-bounds, so the demo input includes:
#   python 2-execution.py 1E-2
#   python 2-execution.py 1E-3
#   python 2-execution.py 1E-4
```

After the execution, you can observe an output. We will then use `python 2-execution.py 1E-2` to understand such output. After that, you can see a generated output as shown in the following code block.

```
====================================================
Done with Execution GSZ-P and GSZ-O on nwchem under 1e-2
GSZ-P   compression throughput: 652.95619 GB/s
GSZ-P decompression throughput: 2350.036331 GB/s
GSZ-P    max compression ratio: 82.506696
GSZ-P    min compression ratio: 82.506696
GSZ-P    avg compression ratio: 82.506696

GSZ-O   compression throughput: 656.172996 GB/s
GSZ-O decompression throughput: 2326.732979 GB/s
GSZ-O    max compression ratio: 82.51842
GSZ-O    min compression ratio: 82.51842
GSZ-O    avg compression ratio: 82.51842
====================================================


====================================================
Done with Execution GSZ-P and GSZ-O on s3d under 1e-2
GSZ-P   compression throughput: 678.3954736 GB/s
GSZ-P decompression throughput: 1221.6985906 GB/s
GSZ-P    max compression ratio: 44.289605
GSZ-P    min compression ratio: 44.273392
GSZ-P    avg compression ratio: 44.2824034

GSZ-O   compression throughput: 712.9922364 GB/s
GSZ-O decompression throughput: 1318.336099 GB/s
GSZ-O    max compression ratio: 90.287397
GSZ-O    min compression ratio: 89.566029
GSZ-O    avg compression ratio: 89.8573926
====================================================
```