Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds

Alex Fallin*, Noushin Azami*, Sheng Di†‡, Franck Cappello†‡, Martin Burtscher*
*Texas State University, USA
†Argonne National Laboratory, USA
‡University of Chicago, USA
Emails: {waf13,noushin.azami,burtscher}@txstate.edu, {sdi1,cappello}@anl.gov

***Abstract*—High-throughput data compression is increasingly important for large scientific projects. This paper presents PFPL, a lossy floating-point data compressor with guaranteed error bounds that is fully compatible between CPUs and GPUs. Despite this compatibility, PFPL delivers some of the highest compression and decompression speeds and compression ratios on both single- and double-precision data. For example, using an absolute error bound of 1E-3, it yields a single-precision compression throughput on the SDRBench inputs of 5 GB/s on a Ryzen 2950X CPU and 423 GB/s on an RTX 4090 GPU. This is at least 4.6 times higher than the throughput of seven leading compressors on both devices. Moreover, PFPL's compression ratio is higher than that of all tested GPU codes.**

***Index Terms*—lossy data compression, guaranteed error bounds, CPU/GPU compatibility**

I. Introduction

Many scientific instruments and simulations generate more data than can reasonably be handled, both in terms of throughput and in terms of total size [20]. There are two types of data compression, lossy and lossless, to alleviate this problem. Lossless compressors exactly reproduce the original data bit-for-bit. However, they are often not able to deliver the desired compression ratios. For example, ZFP [11] can losslessly compress the NYX file baryon_density [30] by only a factor of 1.34. In contrast, lossy compression can yield much higher compression ratios, depending on the selected error bound. Using the same NYX input with an absolute error bound of 0.01, ZFP obtains a compression ratio of 4.92. This increase in compression ratio comes at a price. As the name suggests, lossy compressors "lose" some information and are unable to perfectly reconstruct the data.

There are three critical issues in state-of-the-art error-bounded lossy compressors: (1) unguaranteed error bounds, (2) no support across heterogeneous devices, and (3) low compression ratio or throughput. Our PFPL compressor incorporates novel solutions to address these issues.

(1) *Unguaranteed error bounds*. The three most widely used error-bound types are point-wise absolute error (ABS), point-wise relative error (REL), and point-wise normalized absolute error (NOA). ABS with a bound of ε means each individual decompressed floating-point value will not vary by more than a difference of ε from its original value. REL bounds the error relative to the original value such that each reconstructed value will not vary by more than a factor of $1 + \varepsilon$. NOA ensures that an individual value will not differ by more than ε times the value range of the input (i.e., the maximum minus the minimum value). Which error-bound type is most appropriate depends on the data and for what it will be used. Guaranteeing the specified error bound is important to domain scientists who already distrust lossy compression [4].

Guaranteeing the error bound for any of the 3 types is quite difficult. In fact, our experiments show that most existing error-bounded compressors, including MGARD-X [6], SPERR [21], SZ2 [23], and ZFP, violate the bound in some cases, mainly due to the finite precision of floating-point operations. For example, the quantization used in SZ2 performs a floating-point division by the error bound during compression and a corresponding multiplication by the same error bound during decompression. Due to rounding, this does not always yield the expected value, sometimes leading to error-bound violations. Other compressors have even more serious issues. For instance, cuSZp [15] performs a pre-quantization of the floating-point data that may cause integer overflow.

(2) *No support across heterogeneous devices*. In today's heterogeneous HPC environments, scientific data is often generated and compressed on one device but decompressed on a different device. On the one hand, GPU-based compression may be critical for applications that produce data at a very high throughput, whereas CPU-based compression may be sufficient in other environments. On the other hand, the resulting data may be decompressed and analyzed by various users who may or may not have a GPU. Hence, cross-device compression and decompression is important but rarely supported by today's state-of-the-art lossy compressors.

(3) *Low compression ratio or throughput*. Existing error-bounded lossy compressors typically either deliver high compression ratios with limited throughput or high throughputs with limited compression ratios. For example, the CPU-based SZ3 [24, 26, 36] relies on Huffman coding and ZSTD to greatly reduce the data size, but these coders suffer from low throughput. In contrast, the GPU-based cuSZp yields high throughputs at the cost of low compression ratios. Improving compression and decompression throughput while delivering a high compression ratio is challenging because transformations that compress well tend to be slow and not GPU friendly, and transformations that are fast tend to not compress well.

The current state of the art is disconcerting. Of the 7 leading lossy compressors from the literature that we evaluate in this paper, only one (SZ2) supports all three major error-bound types but does not guarantee the error bound. Only one (SZ3) guarantees the error bound but does not support the REL error-bound type. Only one (MGARD-X) provides CPU/GPU compatibility but does not guarantee the error bound nor does it support REL, and only one (SZ3 OpenMP) combines a high compression ratio and a reasonably high throughput but does not support GPUs. Table III provides more detail on the features that each compressor supports.

To remedy these shortcomings, we developed the error-bounded PFPL (Portable Floating-Point Lossy) compression algorithm that yields high compression ratios for all three main error-bound types while also being efficiently implementable on both CPUs and GPUs and guaranteeing the error bound in all cases. On the SDRBench inputs [30], it delivers higher compression ratios than the other tested compressors that reach the same throughput, and it delivers higher throughputs than all other tested compressors that reach the same compression ratio. PFPL handles all single- and double-precision values, including infinities, NaNs (not a number), and denormals.

This paper makes the following main contributions.

- It presents PFPL, a new lossy compression algorithm, and its CPU and GPU implementations that guarantee bit-for-bit identical deterministic compressed and decompressed output on both types of devices.
- It describes how PFPL guarantees the error bound for all three major point-wise error-bound types by losslessly encoding the single- and double-precision values that would otherwise violate the bound.
- It explains how PFPL, whose compression pipeline comprises a novel combination of parallelism-friendly transformations, is optimized to yield the highest compression ratios on the tested GPUs and the highest throughputs on the tested CPUs.

Our PFPL C++/OpenMP and CUDA implementations are freely available through GitHub [12].

The rest of this paper is organized as follows. Section II provides background. Section III explains the PFPL algorithm. Section IV describes the evaluation methodology. Section V presents and discusses the results. Section VI summarizes related work. Section VII concludes the paper with a summary.

II. Background

This section describes the three point-wise error-control metrics, point-wise absolute (ABS), point-wise relative (REL), and point-wise normalized absolute (NOA), that are most commonly used in the literature [15, 22, 25, 27, 34, 37].

A. Point-Wise Absolute Error (ABS)

The point-wise absolute error of a data value is the *difference* between the original value of the data point and its reconstructed value [32]. The absolute error of a data value x is defined as $e_{abs} = |x_{original} - x_{reconstructed}|$. Therefore, to guarantee an absolute error bound of ε, each value in the reconstructed file must satisfy $e_{abs} \leq \varepsilon$. In other words, each reconstructed value must be in the following range: $x_{original} - \varepsilon \leq x_{reconstructed} \leq x_{original} + \varepsilon$.

ABS error bounds are useful when the data is quite homogeneous in terms of magnitude or when the user does not have a particular interest in areas where values may be small relative to the error bound, that is, when the user cares mainly about the "big picture".

B. Point-Wise Relative Error (REL)

The point-wise relative error of a data value is the *ratio* between the difference of the original and its reconstructed value and the original value [22]. The relative error of a value x is expressed as $e_{rel} = |\frac{x_{original} - x_{reconstructed}}{x_{original}}| = |1 - \frac{x_{reconstructed}}{x_{original}}|$. To guarantee a relative error bound of ε, each value in the reconstructed file must satisfy $e_{rel} \leq \varepsilon$. In other words, each reconstructed value must have the same sign as the original value and be in the following range: $|x_{original}|/(1+\varepsilon) \leq |x_{reconstructed}| \leq |x_{original}| \times (1+\varepsilon)$. The point-wise relative error is often referred to as PW_REL or PWR in the literature [1, 22]. We omit the "PW" as it is not used for any of the other point-wise error-bound names.

REL error bounds are employed when the user wants to preserve a high level of detail in areas where the values are close to zero but does not mind a higher absolute error in areas where the absolute values are larger.

C. Point-Wise Normalized Absolute Error (NOA)

The point-wise normalized absolute error is the ABS error normalized by the value range $R = x_{max} - x_{min}$, that is, the range between the largest and the smallest value in the input. The normalized absolute error of a data value x is defined as $e_{noa} = |\frac{e_{abs}}{R}|$. To guarantee an error bound of ε, each value in the reconstructed file must satisfy $e_{noa} \leq \varepsilon$. Hence, each reconstructed value must be in the range: $x_{original} - \varepsilon R \leq x_{reconstructed} \leq x_{original} + \varepsilon R$. Unfortunately, the literature often refers to NOA as "relative error" or "REL" [1, 5, 16]. We find this nomenclature misleading because, aside from multiplying the error bound by a constant, the quantization is identical to that of ABS (and not REL). Hence, we call it the normalized absolute error bound.

NOA error bounds are convenient when the user has multiple datasets at different scales but only wants to specify one absolute error bound for all of them.

III. PFPL Algorithm and Implementation

Supporting compatible compression and decompression between CPUs and GPUs while maintaining efficiency led us to use a modular approach in PFPL. Like other algorithms (see Section VI), PFPL employs a pipeline of data transformations, all of which are shared between the ABS, REL, and NOA error-bounded versions, as shown in Figure 1.

A. Enhanced Lossy Quantizers

Irrespective of the error-bound type, PFPL always starts with a lossy quantizer (Step 1 in Figure 1). ABS and REL are distinct quantizers whereas NOA is a special case of ABS.

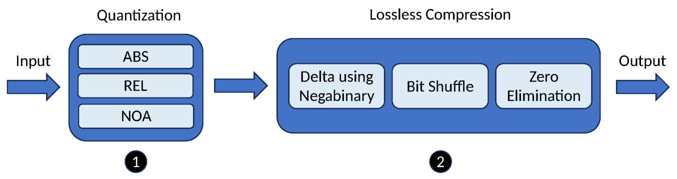


Fig. 1: Visual representation of the PFPL compression algorithm. Step 1 quantizes the input values using one of three error-bound types. Step 2 losslessly compresses the quantized values. Decompression employs the inverse of these transformations in the opposite order.

The ABS quantizer uses the supplied point-wise absolute error bound ε to quantize each floating-point value into a bin. This is accomplished by multiplying the value by $0.5/\varepsilon$ (i.e., the inverse of twice the error bound) and rounding the result to the nearest integer, which yields the bin number. All values within $\pm\varepsilon$ of the center of a bin are thus mapped to the same bin and will be reconstructed to the center value. For any value v, this quantizer guarantees that v will be decoded to a value v' such that $v - \varepsilon \leq v' \leq v + \varepsilon$.

The REL quantizer operates similarly but in logarithmic space. It quantizes each floating-point value into a bin by multiplying the logarithm of its absolute value by $0.5/log(\varepsilon)$, rounding the result to the nearest integer, and applying the sign of the original value. It reconstructs all values from a given bin into the same value. Operating in log space makes the error bound relative. For a value v and error bound ε, this quantizer guarantees v will be decoded to a value v' with the same sign as v such that $|v|/(1+\varepsilon) \leq |v'| \leq |v| \times (1+\varepsilon)$.

The NOA quantizer is identical to ABS except it first determines the minimum and maximum values in the input and multiplies the error bound ε by the resulting range, yielding $abs = \varepsilon \times (max - min)$. This abs error bound is then used for quantization as if it were passed directly to ABS.

B. Guaranteeing Error Bounds

The three PFPL quantizers guarantee the user-provided error bound by *losslessly* encoding all values that would otherwise violate the bound. In the case of denormals, infinities, and NaNs, the quantizers simply check for these special values. Regular values are more complex to guarantee. Due to the finite precision of floating-point values, there are cases where operations that ought to yield results within the error bound do not as outlined in Section I. This is why the encoder in all PFPL quantizers immediately decodes each value to check whether it meets the error bound. If it does, the bin number is emitted. Otherwise, the unmodified floating-point value is emitted. The cost of this error-bound-guarantee approach is relatively low. The throughput is unaffected and the compression ratio is, on average, lower by about 5% [13]. The loss in compression ratio varies based on the error bound, type, and number of unquantizable values. The ABS error-bound type is most affected, and smaller error bounds are more likely to yield unquantizable values. At an ABS error bound of 1E-3, on average 0.7% of the values in all our inputs are unquantizable with a maximum of 11.2% on a single input.

Other lossy compressors record such unquantizable values in a separate list and use a reserved bin number to tell the decoder when the next value from that list must be used [9, 21, 32]. This approach increases the amount of emitted data and complicates parallelization. Our quantizers avoid both issues by emitting a single data stream in which each value is either a bin number or the original floating-point value. We use the following technique to correctly decode the data.

For ABS and NOA, the error bound cannot be less than the smallest positive non-denormal floating-point value. This means that denormals are always quantized to zero. Hence, we can use the 8-million-value-wide denormal range in the floating-point representation for recording the bin numbers (in magnitude-sign format). Bin numbers that would be too large are not used and the corresponding value is encoded losslessly.

For REL, this approach does not work as it requires particularly high precision in the denormal range. Instead, we use the negative 8-million-value-wide NaN range for storing the bin numbers. To free it up, we make all negative NaNs positive. Since negative NaNs (i.e., bin numbers) have a large number of leading '1' bits, we invert those bits of all emitted values (bin numbers and losslessly encoded floating-point values) to make the data easier to compress in the later stages.

Double-precision values are handled in the same way. However, their denormal and NaN ranges are much larger, allowing for a wider range of bin numbers to be encoded.

C. Guaranteeing CPU/GPU Compatibility

Since we want the same bit-for-bit compression and decompression result on CPUs and GPUs, we only use operations that are guaranteed to yield the same result on both kinds of devices. In particular, we exclusively use floating-point operations that are fully IEEE 754 compliant [2] and, importantly, employ flags to prevent the compiler from introducing non-compliant operations. Hence, we implemented our three quantizers with only floating-point addition, subtraction, multiplication, and division (no fused multiply-adds) as well as integer operations. This was relatively straightforward for ABS and NOA. However, REL includes $log()$ and $pow()$ function calls, which often do not produce the same result on the GPU as they do on the CPU. To address this issue, we wrote approximations for both functions that use only IEEE-compliant operations. These approximations introduce small inaccuracies, which are fine in most cases because the result is still within the error bound. In cases where it is not, the aforementioned immediate verification catches the problem and losslessly encodes the affected values. On the tested inputs, our approximations for guaranteeing CPU/GPU compatibility cause a 5% loss in compression ratio, on average, and cause no change in throughput.

D. Lossless Compression

Quantization by itself does not shrink the data but only transforms the 32- or 64-bit floating-point values into more

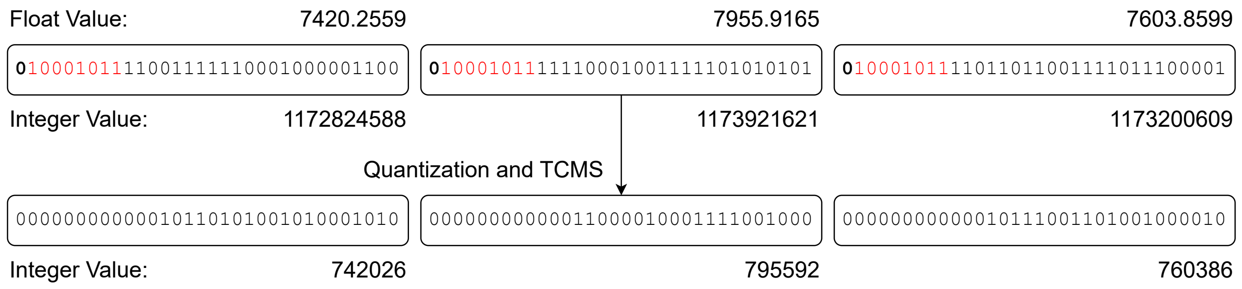


Fig. 2: ABS quantization with an error bound of 0.01. Successfully quantized values are stored as bin numbers in magnitude-sign format, values that cannot be quantized within the error bound are stored losslessly without changing any bits (not shown)

compressible integer bin numbers as outlined in Figure 2. To find a good lossless compression algorithm for the output of our quantizers, we tested a large number of combinations of data transformations (see below). To ensure a high throughput, we only considered transformations that can be implemented efficiently on CPUs and GPUs. The 3 lossless pipeline stages of PFPL (Step 2 in Figure 1) are the result of this search.

We designed these stages with the LC framework [3], which can automatically synthesize parallelized data compressors for CPUs and GPUs. In particular, we used LC to generate many algorithms and then optimized the best. This led to the creation of the transformations described below, which boost the compression ratio while maintaining a high throughput. Note that PFPL employs the same lossless compression pipeline for all three quantizers. Moreover, the double-precision code uses the same pipeline as the single-precision code but with the word size of all but the last stage increased to 64 bits.

The first lossless stage, an example of whose operation is shown in Figure 3, computes the difference sequence of the quantizer output, that is, it performs delta modulation [19]. This means each value (e.g., 3, 4, 4, 3) is replaced by itself minus the previous value (e.g., 3, 1, 0, -1). If the bin numbers are close to each other, which they are for many scientific datasets, this transformation yields residuals that cluster around zero. Importantly, this stage stores the residuals in negabinary format. Negabinary is a representation of values in base -2. Unlike in twos-complement representation, small positive and small negative negabinary values both have many leading zero bits, as shown in the third row of Figure 3. This is exploited in the later compression stages.

The second lossless stage performs bit shuffling (aka bit transposition) [28] as outlined in Figure 4. It outputs the most significant bit of all residuals, followed by the second-most significant bit of all residuals, and so on. If consecutive residuals have '0' bits in the same position, which they often do due to the negabinary values from the prior stage, this transformation yields long runs of '0' bits.

Unlike all other stages, the final stage operates at byte granularity. As illustrated in Figure 5, it generates a bitmap in which each bit corresponds to a byte of the input. A cleared bit indicates that the corresponding byte is zero. Otherwise, the bit is set. All zero bytes are then removed from the input. Hence, the compressed output consists of the bitmap and the non-zero bytes from the input. The size of the bitmap is always the size of the input divided by 8, but the number of non-zero bytes depends on the data. Since the bitmap represents considerable overhead, we compress it using a similar algorithm that creates a second, smaller bitmap in which a cleared bit means the byte repeats and a set bit means it does not. Only the non-repeating bytes of the first bitmap are emitted along with the second 8-times-smaller bitmap. This process is iteratively applied 4 times, generating a shorter bitmap in each iteration (plus the non-repeating bytes), until the bitmap is only a few bytes long.

By themselves, none of these transformations produce high compression ratios. In fact, only the last stage compresses at all. The overall performance stems from the specific sequence described. Removing any one of these transformations decreases the compression ratio by a substantial factor.

PFPL is designed to target a wide range of scientific data since this domain is one of the largest producers of floating-point data. For numeric stability, such data tends to be smooth, which our compressors exploit. As described above, data in which the consecutive values differ by relatively small amounts are transformed into long sequences of '0' bits. These '0' bits are then eliminated in the last stage. PFPL should, therefore, compress relatively smooth data from various domains well. We do not expect our lossy compressors to work as well on data that is not smooth. However, the wide range of scientific inputs we use for evaluation (see Section IV) tend to be quite smooth, are centered around zero, and contain no denormals, NaNs, or infinities [37].

E. Parallelization and Optimization

We implemented PFPL in OpenMP for parallel CPU execution and in CUDA for parallel GPU execution. Since our quantizers do not separately record outliers, ABS and REL are embarrassingly parallel and trivial to parallelize, i.e., every floating-point value in the input can be independently quantized. The same is true for NOA, except it first needs to perform a parallel minimum and maximum reduction. The resulting range is recorded in the compressed file so the decoder has access to it, making it embarrassingly parallel.

The lossless pipeline stages are parallelized for the CPU by breaking the data into 16 kB chunks that are independently

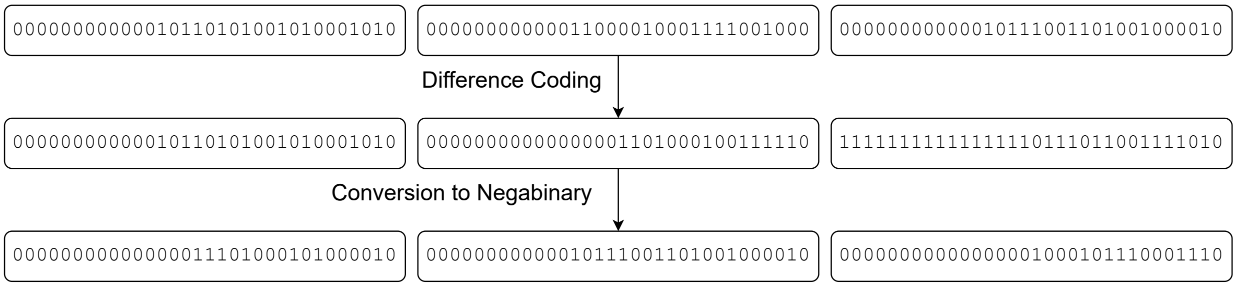


Fig. 3: Difference coding and negabinary conversion in the first lossless stage

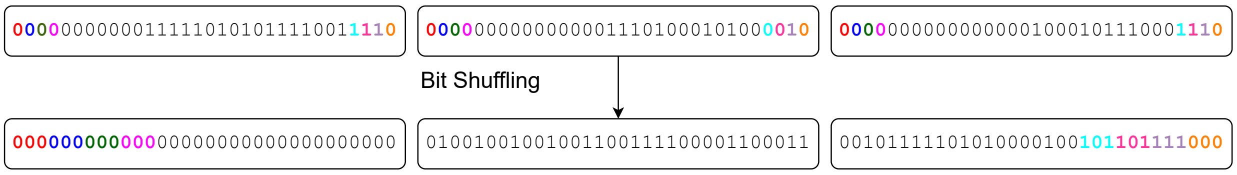


Fig. 4: Bit shuffling in the second lossless stage; for larger inputs, the sequences of bits with the same color will be longer

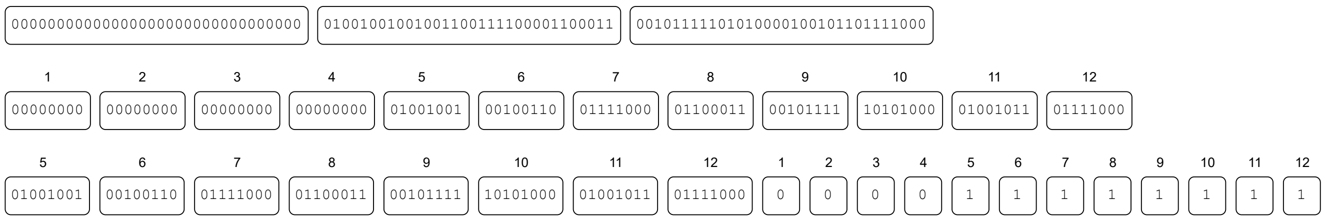


Fig. 5: Zero-byte elimination in the final lossless stage; further compression of the bitmap is not shown

processed by different threads. A similar approach is used for the GPU, where each chunk is assigned to a separate thread block. Since not all chunks are equally compressible, we dynamically assign the chunks to the threads or thread blocks to improve the load balance. If a chunk cannot be compressed, the original chunk data is emitted and the chunk is flagged as uncompressed to cap the worst-case expansion.

On both the CPU and the GPU, the resulting compressed chunks are concatenated and their sizes are separately stored in the compressed file. The concatenation is implemented by propagating the cumulative size of all prior compressed chunks to the next thread or thread block so it knows where to start writing its output. On the CPU, we use a shared carry array for this purpose that is accessed with atomic reads and writes. On the GPU, we use Merrill and Garland's decoupled look-back technique [29]. On both devices, the decoder computes a prefix sum over the stored chunk sizes to determine where to start reading each chunk.

In the CUDA code, each lossless stage is further parallelized across the threads within a thread block. This is trivial for the delta encoder, which is embarrassingly parallel. The decoder requires a block-wide prefix sum. The bit-shuffle encoder and decoder operate at warp granularity, where each warp is independently responsible for a chunk of 32 or 64 values. They employ $log_2(wordsize)$ shuffling steps, which are implemented using warp shuffle instructions that exchange data without accessing memory. Creating the bitmaps in the zero-elimination encoder is embarrassingly parallel and can be done without atomic operations as we assign 8 consecutive bytes to each thread. Outputting the non-zero (or non-repeating) bytes to the correct location requires a block-wide prefix sum. The decoder similarly performs a block-wide prefix sum over the bits in the bitmap to compute the location where each thread can find the non-zero (or non-repeating) bytes it needs to recreate the uncompressed data.

To maximize performance, we optimized our PFPL implementation as much as we could. In the GPU code, we use coalesced memory accesses wherever possible (even when reading and writing "unaligned" compressed data) to improve the memory throughput. We minimize the size of the relatively expensive prefix sums by allocating multiple values to each thread and computing a thread-local result before invoking the block-wide prefix sum. We maximize the use of shuffle instructions to exchange data between threads without accessing memory. In the CPU code, we avoid almost all prefix sums by having each thread process an entire chunk at a time. The most important optimization is fusing all four stages in both the CPU and the GPU code, including the quantizer. This way, the data is only read from main memory once, then all transformations are performed, and finally the data is written back to main memory. The GPU code keeps almost all intermediate data in shared memory (a software-controlled L1 data cache). The CPU code keeps most of the intermediate data in two 16 kB buffers that are alternately used and, therefore, likely resident in the L1 data cache.

IV. Experimental Methodology

We compare PFPL to 7 state-of-the-art lossy compressors, described in Section VI, on the two systems listed in Table I.

We compiled the CPU codes using the build processes supplied by their respective authors. When not specified, we used the "-O3 -march=native" flags. Unless automatically determined, the thread count was set to the number of CPU cores as hyperthreading usually does not help. We compiled the GPU codes using the "-O3 -arch=sm_89" flags for the RTX 4090 and the "-O3 -arch=sm_80" flags for the A100.

For all compressors, we measured the execution time of the compression and decompression functions, excluding reading the input file, verifying the results, and transferring data to and from the device. We ran each experiment 9 times and collected the compression ratio, median compression throughput, and median decompression throughput. The plots report the geometric mean of the geometric mean of each suite so as not to overemphasize suites with more files. Additionally, the use of the geometric rather than arithmetic mean helps dampen any inputs that significantly outperform the general case [14].

TABLE I: Systems used for experiments

	System 1	System 2
CPU	Threadripper 2950X	Xeon Gold 6226R
Base Clock	3.5 GHz	2.9 GHz
Sockets	1	2
Cores Per Socket	16	16
Threads Per Core	2	2
Main memory	64 GB	64 GB
GPU	RTX 4090	A100
Compute Capability	8.9	8.0
Base Clock	2.2 GHz	0.8 GHz
Boost Clock	2.5 GHz	1.4 GHz
SMs	128	108
CUDA Cores per SM	128	64
Main memory	24 GB HBM2e	40 GB GDDR6x
Operating System	Fedora 37	Fedora 36
g++ Version	12.2.1	12.2.1
nvcc Version	12.0	12.0
GPU Driver	525.85	535.113

TABLE II: Information about the used input suites

Name	**Description**	**Format**	**Files**	**Dimensions**	**Size (MB)**
CESM-ATM	Climate	Single	33	$26 \times 1800 \times 3600$	674
EXAALT Copper	Molecular Dyn.	Single	6	Various 2D	68 to 358
Hurricane Isabel	Weather Sim.	Single	13	$100 \times 500 \times 500$	100
HACC	Cosmology	Single	6	280,953,867	1124
NYX	Cosmology	Single	6	$512 \times 512 \times 512$	537
SCALE	Climate	Single	12	$98 \times 1200 \times 1200$	564
QMCPACK	Quantum MC	Single	2	$33{,}120 \times 69 \times 69$	631
NWChem	Molecular Dyn.	Double	1	102,953,248	824
Miranda	Hydrodynamics	Double	7	$256 \times 384 \times 384$	302
Brown Samples	Synthetic	Double	3	33,554,433	268

We used the 7 single-precision and 3 double-precision suites shown in Table II as inputs for the compressors (if supported), a total of 89 files. These inputs are sourced from the SDRBench repository [30, 37], which hosts real-world scientific datasets from various domains for compression evaluation. The table lists the suite's name, short description, floating-point format, number of files, input dimensions, and file size.

To keep the number of inputs reasonable and make the comparison as fair as possible, we excluded some SDRBench datasets. We only use the 3D CESM-ATM inputs as they are similar to the other CESM-ATM inputs and 3D is a commonly used dimension. We use only the EXAALT Copper dataset as it is in the middle in terms of size for the EXAALT sets. We use the raw (i.e., not cleared) data from the Hurricane ISABEL set. Additionally, we exclude SDRBench datasets that are incompatible with the tested compressors because they are either too large or in a proprietary format.

We present the results in x/y-scatter plots. The two dimensions are compression ratio and either compression or decompression throughput. Note that, for space reasons, the plots include serial and parallel CPU results as well as GPU results. One or both axes are logarithmic to capture the wide range of compression ratios and/or throughputs. For all compressors, the circular data point is for an error bound of 1E-1, the triangle for 1E-2, the square for 1E-3, and the pentagon for an error bound of 1E-4. The Pareto fronts, sets of empirical optima, are marked with light blue lines. For a compressor to be on the Pareto front, it must outperform every other compressor in at least one dimension for the given error bound. For the third-party compressors that support serial and parallel execution or CPU and GPU execution, we only show the fastest version if the compression ratio is the same between the versions. Otherwise, we show all versions. For ZFP, which supports serial and parallel compression but only serial decompression, we show serial results only. We always show all versions of PFPL.

We only compare to SPERR-3D because SPERR-2D does not run in parallel and SPERR-3D supports more of the input suites. However, SPERR-3D does not run in parallel on double-precision inputs. We only compare to SZ2 in Section V-C. In the other subsections, we instead compare to SZ3 because it performs better in terms of compression ratio and is comparable in terms of throughput. However, SZ3 does not support the REL error bound whereas SZ2 does. For the ABS and NOA error bounds, we show results for two versions of SZ3: $SZ3_{Serial}$ and $SZ3_{OMP}$. The OpenMP version of SZ3 produces different compression ratios, and therefore different files, than the serial version, but both versions can be used to compress and decompress interchangeably and still yield correct (but different) output. Since not all compressors perform a warm-up before timing, where present, we disabled the warm-up code to make the performance comparisons fairer.

ZFP bounds the relative error by truncating a requested number of least significant bits in the floating-point representation. Bounding the REL error by a specific value (e.g., 1E-3) is not always possible with ZFP. We report results for bit-truncations that yield similar errors to the bounds used when running the other two REL compressors.

The presented compression ratios are the uncompressed file size divided by the compressed file size. Rather than listing the measured runtimes, we show throughputs (i.e., the uncompressed file size divided by the runtime) because throughput, like compression ratio, is a higher-is-better metric.

V. Performance Evaluation

In this section, we evaluate 7 leading lossy compressors from the literature as well as PFPL. We first compare the error-bound guarantees and other features of each compressor. Then, we analyze the throughputs and compression ratios on different error bounds and data types for ABS, REL, and NOA. Next, we present results on the reconstruction quality of the lossily decompressed data. Finally, we discuss the performance of PFPL on additional GPU generations and our findings when profiling the CUDA version of PFPL.

A. Supported Features

This subsection compares the 8 lossy compressors in terms of which error-bound types they support, whether they guarantee the error bound, whether they support CPU and GPU execution, and other properties. Table III shows the results. The top row lists the features; each remaining row corresponds to one compressor.

TABLE III: All tested compressors (ordered by initial release date) and the features they support. '✓' indicates the compressor supports the feature, '×' indicates that it does not, and '○' indicates that the compressor supports an error bound type but does not always adhere to the requested error bound.

Compressor	ABS	REL	NOA	Float	Double	CPU	GPU
ZFP	○	✓	×	✓	✓	✓	×
SZ2	✓	○	✓	✓	✓	✓	×
SZ3	✓	×	✓	✓	✓	✓	×
MGARD-X	○	×	○	✓	✓	✓	✓
SPERR	○	×	×	✓	✓	✓	×
FZ-GPU	×	×	○	✓	×	×	✓
cuSZp	○	×	✓	✓	✓	×	✓
PFPL	✓	✓	✓	✓	✓	✓	✓

'ABS', 'REL', and 'NOA' denote the error-bound type. Note that SZ2, the only other compressor to support all three error bound types, fails to guarantee the error bound when using REL. In contrast, the related SZ3 guarantees the error bound because it does not support REL. Like SZ2, ZFP and SPERR also violate the error bounds in a few cases due to rounding issues. PFPL avoids such problems by double-checking each value after compression and storing it losslessly whenever the error bound would be violated. 'Float' and 'Double' indicate support for 32-bit single-precision and 64-bit double-precision data. 'CPU' and 'GPU' display support for compression and decompression on the respective device.

Even though lossy compression is quite mature, PFPL advances the state of the art in that, to our knowledge, it is the only compressor that supports all listed features. Moreover, it compresses on par with many other tested compressors and is at least as fast as all of them, including compressors that exploit input dimensionality or only run on and are optimized for a specific type of device, as illustrated below.

B. ABS Error Bounds

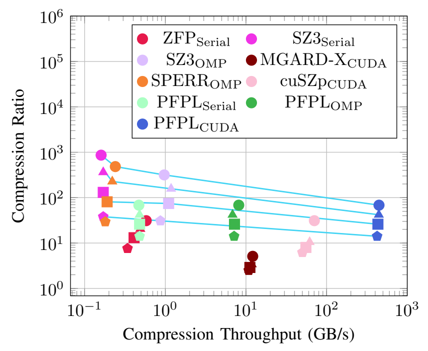
Compression: Figures 6a and 6b show scatter plots of the compression ratio versus the compression throughput for the ABS error bounds on System 1 for the single- and double-precision inputs, respectively. To produce these results, we excluded the EXAALT and HACC inputs because they are not 3D, which causes issues with SPERR, and because HACC makes MGARD-X run out of memory. Note that cuSZp and ZFP have major ($\geq 1.5\times$) error-bound violations for all tested error bounds and that SPERR has minor ($< 1.5\times$) violations for the 1E-2 error bound. MGARD-X has major error bound violations on all tested error bounds, but only for the double-precision inputs. SPERR is not listed for the double results as it does not support the majority of those inputs. FZ-GPU is not listed as it does not support ABS.

PFPL$_{\text{CUDA}}$ delivers the highest throughput for all tested error bounds, and PFPL$_{\text{OMP}}$ yields the highest throughputs on the CPU. PFPL$_{\text{OMP}}$ performs particularly well; it is 7.1 times faster than the next fastest CPU code (SZ3$_{\text{OMP}}$) on average and almost on par with the slowest GPU code. This good performance is primarily because we built our algorithm out of only fast transformations, fused all stages, and only read the input data once from main memory and write the output data once to main memory. In general, the throughput of the various compressors decreases with smaller error bounds, but by less than the decrease in compression ratio. PFPL$_{\text{CUDA}}$ achieves a compression throughput of 446 GB/s at the coarsest and 423 GB/s at the finest tested bound on the single-precision inputs.

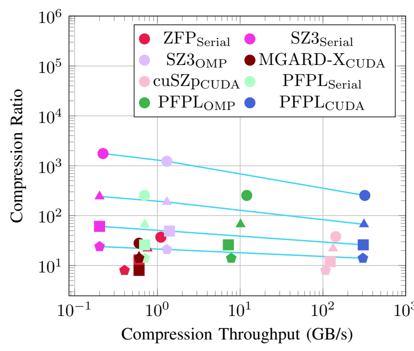
SZ3$_{\text{Serial}}$ delivers the highest compression ratio for all tested error bounds. The two versions of SZ3 compress more than PFPL on both data types, and SPERR compresses more on single-precision data. They are all CPU-only compressors that use GPU-unfriendly transformations to boost their compression ratio such as Huffman coding. For the same reason, the OpenMP version of SZ3 compresses significantly less than serial SZ3, that is, the serial version includes well-compressing transformations that are not parallelism friendly. We note that ZFP's compression ratios are not on par with the other CPU-only compressors. One reason is that ZFP often over-preserves the compression errors, meaning it tends to deliver a lower maximum error than allowed by the error bound. All tested GPU compressors (and ZFP) are both slower and compress less than PFPL$_{\text{CUDA}}$.

Since the three versions of PFPL are compatible and produce bit-for-bit identical output, their compression ratios are identical. For all tested compressors, the compression ratio decreases with a tighter error bound, as one would expect. However, for the compressors that deliver particularly high compression ratios for an error bound of 1E-1, the drop in compression ratio is more pronounced than for the other compressors. For example, at the largest error bound, PFPL produces a compression ratio of 68 whereas SZ3$_{\text{Serial}}$ yields a ratio of 863, a factor of almost 13. At the smallest error bound, the difference is only a factor of 3 with SZ3$_{\text{Serial}}$ yielding a compression ratio of 38 versus PFPL's ratio of 14. The reason SZ3 delivers higher compression ratios than PFPL is that SZ3 employs a more sophisticated data decorrelation method and lossless encoder at a cost of significantly lower throughput.

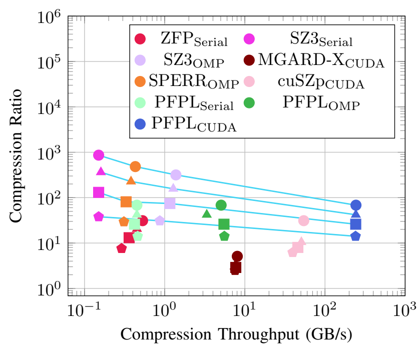
Decompression: Figures 7a and 7b show scatter plots of



(a) System 1 geo-mean compression ratio and compression throughput on single-precision data with 4 ABS error bounds

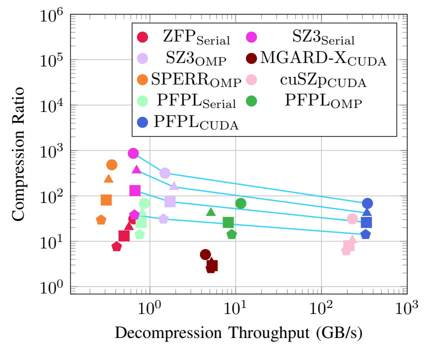


(b) System 1 geo-mean compression ratio and compression throughput on double-precision data with 4 ABS error bounds

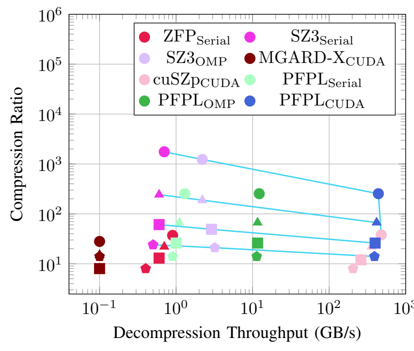


(c) System 2 geo-mean compression ratio and compression throughput on single-precision data with 4 ABS error bounds

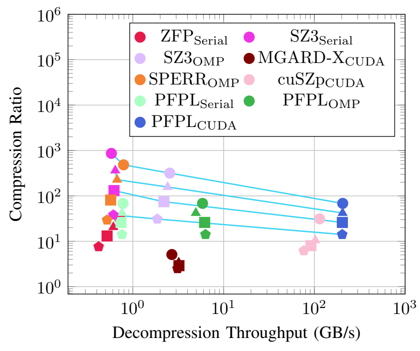
Fig. 6: Compression results for the ABS error-bound type, including Pareto fronts for the 4 error bounds



(a) System 1 geo-mean compression ratio and decompression throughput on single-precision data with 4 ABS error bounds



(b) System 1 geo-mean compression ratio and decompression throughput on double-precision data with 4 ABS error bounds



(c) System 2 geo-mean compression ratio and decompression throughput on single-precision data with 4 ABS error bounds

Fig. 7: Decompression results for the ABS error-bound type, including Pareto fronts for the 4 error bounds

the compression ratio vs. the decompression throughput for the ABS error bounds on System 1 for the single- and double-precision inputs, respectively. The compression ratios are the same as in Figures 6a and 6b. Only the throughputs differ.

At a decompression throughput of between 327 GB/s and 344 GB/s, PFPL$_{\text{CUDA}}$ is still the fastest on the single-precision data, but cuSZp$_{\text{CUDA}}$ is faster on the two coarsest error bounds on the double-precision data. This is because cuSZp$_{\text{CUDA}}$ decompresses much faster than it compresses due to its lightweight fixed-length decoding step. In contrast, our three PFPL versions compress faster than they decompress due to prefix-sum computations in the decoder. MGARD-X compresses noticeably faster than it decompresses, especially on double-precision values, where it is the slowest decompressor even though it runs on the GPU. Otherwise, the decompression performance trends are similar to the compression trends.

System 2: Figures 6c and 7c show the single-precision results for System 2. The compression ratios between the two systems are, of course, the same, but the throughputs differ because System 2 has a more powerful CPU and a less powerful GPU. Otherwise, the trends between the systems are very similar, including for the double-precision results and the other error-bound types. Hence, we only show results from System 1 in the following subsections.

Takeaway 1. In environments where not only compression ratio but also throughput is important, PFPL provides the currently best solution. Despite having features that other compressors do not, in particular full CPU/GPU compatibility, PFPL$_{\text{OMP}}$ is faster than the CPU-only compressors and PFPL$_{\text{CUDA}}$ is generally faster and compresses more than the GPU-only compressors. MGARD-X, the only other compressor that is CPU/GPU compatible, is 37 times slower at compression and 63 times slower at decompression and compresses between 6 and 13 times less than PFPL.

C. REL Error Bounds

Compression: Figures 8 and 9 show scatter plots of the compression ratio versus the compression throughput for the REL error bounds on System 1 for the single- and double-precision inputs, respectively. We used all inputs to produce the results shown in these charts. Note that SZ2 has large error-bound violations on CESM for all tested error bounds and ZFP does not conform to the error bound due to its different bounding technique. Only PFPL, SZ2, and ZFP are shown as they are the only tested compressors that support REL.

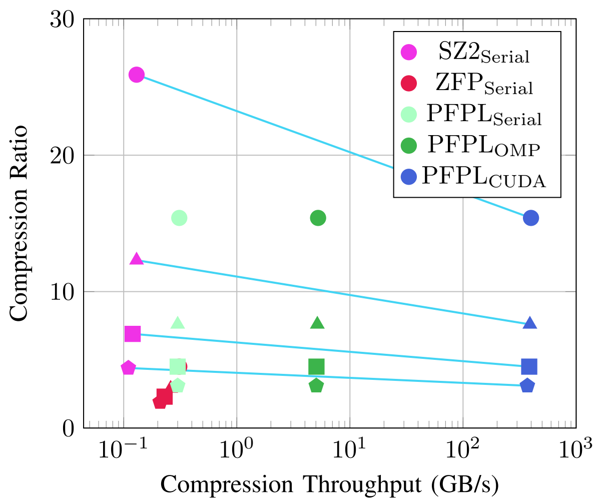


Fig. 8: System 1 geometric-mean compression ratio and compression throughput on single-precision data with REL error bounds, including Pareto fronts for the 4 error bounds

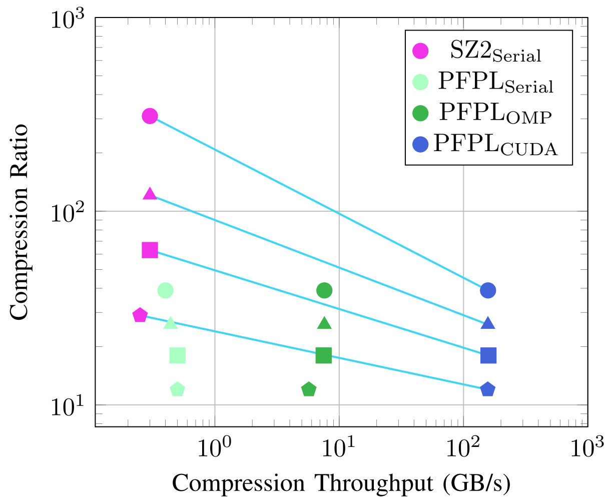


Fig. 9: System 1 geometric-mean compression ratio and compression throughput on double-precision data with REL error bounds, including Pareto fronts for the 4 error bounds

SZ2 yields higher compression ratios than PFPL. It is, however, unclear how much of that is due to the error-bound violations by SZ2. ZFP delivers lower compression ratios than the other compressors mainly due to its truncation-based REL implementation. Like with ABS, the advantage of SZ2 over PFPL shrinks as the error bound gets smaller. SZ2 outcompresses PFPL by a factor of 1.7 at an error bound of 1E-1 but only by a factor of 1.4 at an error bound of 1E-4. This is because, when the error bound is large, the bin values tend to be relatively smooth and easy to compress. Hence, compressors like SZ2 can get very high compression ratios from their sophisticated yet expensive designs.

PFPL$_{\text{CUDA}}$ delivers the highest throughput, followed by the two CPU versions of PFPL, all of which outperform serial SZ2 in terms of throughput due to our lightweight design and performance optimization (see Section III-D and III-E). At the highest error bound, ZFP reaches the compression throughput of PFPL$_{\text{Serial}}$. At all tested error bounds, PFPL$_{\text{CUDA}}$ compresses the single-precision inputs on the order of 3000 times faster than SZ2 and the double-precision inputs roughly 500 times faster. On the CPU, PFPL$_{\text{OMP}}$ compresses 41.4 times faster on average than serial SZ2.

Decompression: Figures 10 and 11 show scatter plots of the compression ratio versus the decompression throughput for the REL error bounds on System 1 for the single- and double-precision inputs, respectively.

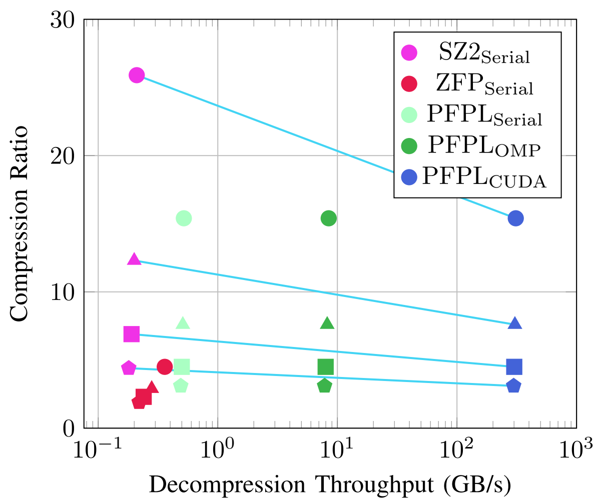


Fig. 10: System 1 geo-mean compression ratio and decompression throughput on single-precision data with REL error bounds, including Pareto fronts for the 4 error bounds

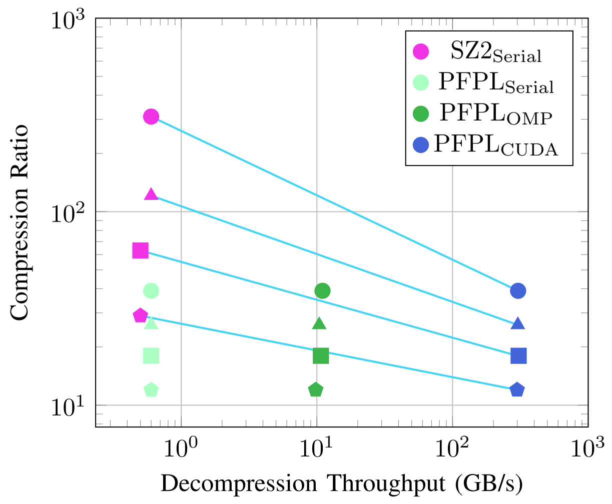


Fig. 11: System 1 geo-mean compression ratio and decompression throughput on double-precision data with REL error bounds, including Pareto fronts for the 4 error bounds

Decompression exhibits the same trends as compression. The three versions of PFPL outperform SZ2 and ZFP in terms of throughput. $\text{PFPL}_{\text{CUDA}}$ is hundreds to thousands of times faster than SZ2. Interestingly, the CPU-based codes decompress faster than they compress whereas $\text{PFPL}_{\text{CUDA}}$ compresses faster than it decompresses. The reason is that our first lossless stage requires a prefix sum for parallelizing the decoder on the GPU but not the encoder, making decompression slower.

Takeaway 2. PFPL greatly outperforms SZ2 in throughput and guarantees the error bound. SZ2 yields a higher compression ratio but violates the error bound on some inputs. ZFP has similar compression throughput to $\text{PFPL}_{\text{Serial}}$ but significantly lower compression ratios. Additionally, PFPL supports parallel CPU and GPU execution, a unique feature for the REL error-bound type, whereas SZ2 and ZFP are only available as serial CPU code.

D. NOA Error Bounds

Compression: Figures 12 and 13 show scatter plots of the compression ratio versus the compression throughput for the NOA error bounds on System 1 for the single- and double-precision inputs, respectively. These results again exclude the EXAALT and HACC inputs as they are not 3D and, therefore, unsupported by FZ-GPU. ZFP and SPERR do not support NOA error bounds and are not shown. FZ-GPU does not support double-precision data and crashes for the 1E-3 and 1E-4 bounds on some of the single-precision inputs. It has minor error bound violations for the other two bounds. MGARD-X and cuSZp have major error-bound violations on all tested double-precision inputs.

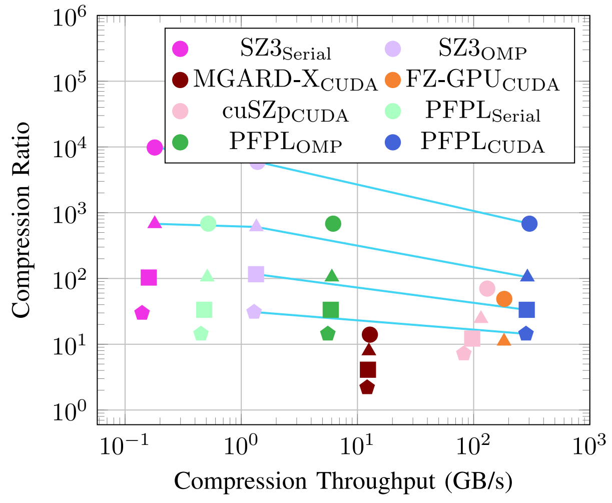


Fig. 12: System 1 geometric-mean compression ratio and compression throughput on single-precision data with NOA error bounds, including Pareto fronts for the 4 error bounds

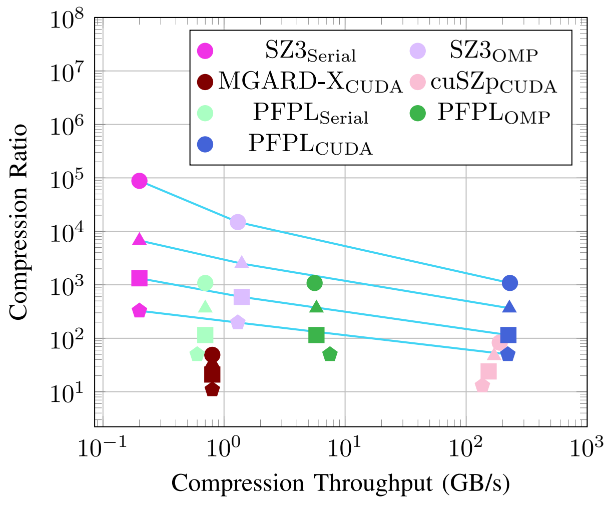


Fig. 13: System 1 geometric-mean compression ratio and compression throughput on double-precision data with NOA error bounds, including Pareto fronts for the 4 error bounds

Both versions of SZ3 yield high compression ratios compared to the other compressors. As noted in the ABS section, the more parallelism-friendly algorithm used in SZ3_{OMP} is less effective than $\text{SZ3}_{\text{Serial}}$. SZ3_{OMP} is the second-fastest CPU code behind PFPL_{OMP}, which is 4.4 times faster on average. The next best compressor for both data types is PFPL. On the single-precision inputs, $\text{PFPL}_{\text{CUDA}}$ is the fastest for all tested error bounds. On the double-precision inputs, cuSZp is faster but yields a lower compression ratio and violates the error bound. At the tightest error bound, cuSZp yields a compression ratio of 13 compared to PFPL's compression ratio of 50.

Decompression: Figures 14 and 15 show scatter plots of the compression ratio versus the decompression throughput for the NOA error bounds on System 1 for the single- and double-precision inputs, respectively.

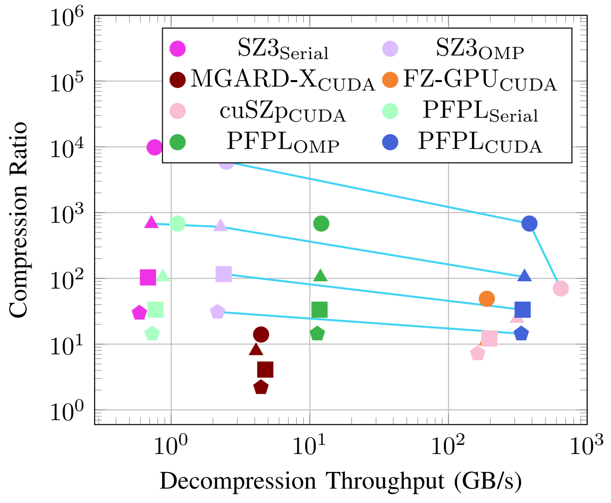


Fig. 14: System 1 geo-mean compression ratio and decompression throughput on single-precision data with NOA error bounds, including Pareto fronts for the 4 error bounds

The general trend for decompression is similar to the compression results. The only major difference is that cuSZp

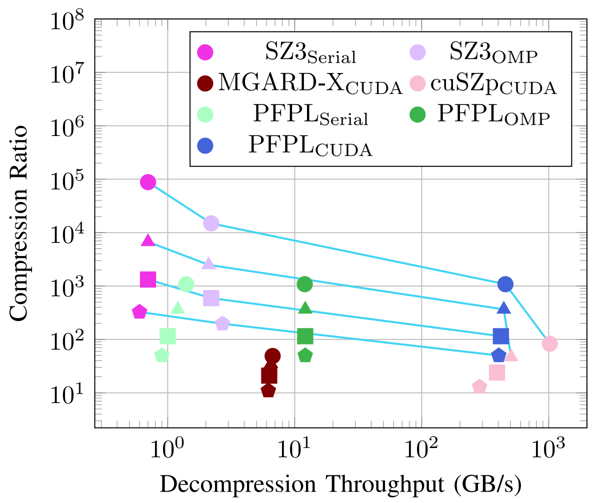


Fig. 15: System 1 geo-mean compression ratio and decompression throughput on double-precision data with NOA error bounds, including Pareto fronts for the 4 error bounds

now outperforms PFPL in decompression speed on one of the error bounds for the single-precision data. The reverse is true for the double-precision inputs, where cuSZp is fastest on three out of the four tested bounds. PFPL$_{\mathrm{OMP}}$ delivers high decompression throughputs on the CPU. On average, it is 5 times faster than the next fastest CPU code and 2.7 times faster than the slowest GPU code.

Takeaway 3. For the NOA error-bound type, if both throughput and compression ratio are important, PFPL is the preferred solution. SZ3 is the best choice if only the compression ratio matters. PFPL is much faster and compresses more than MGARD-X, the only other tested compressor that is CPU/GPU compatible.

E. Quality of Reconstructed Data

Figure 16 shows the relationship between the peak-signal-to-noise ratio (PSNR) and the compression ratio at different error bounds for all tested compressors and error-bound types. The inputs used for producing each PSNR chart match those of the respective result sections above. Higher PSNR values are better as they indicate higher reconstruction quality. A user may want to select the compressor that has the best PSNR.

PFPL yields a PSNR-to-compression-ratio relationship that falls in-between the CPU-only compressors and the GPU-compatible compressors, that is, PFPL delivers the best results among the GPU codes. In absolute terms, its PSNR is similar to that of the best CPU compressors, but at a lower compression ratio. These results demonstrate that the quantization method we use in PFPL delivers comparable data quality to the CPU-only compressors.

F. Other GPU Generations and CUDA Profiling

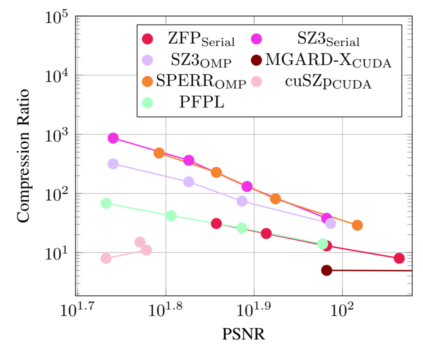
In addition to the aforementioned RTX 4090 and A100 GPUs, we also evaluated PFPL on a TITAN Xp, an RTX 2070 Super, and an RTX 3080 Ti. This experiment shows that the performance correlates primarily with the amount of compute provided by the GPU. For example, the RTX 2070 Super has a maximum number of threads per block of only 1024, which is fewer than any of the other GPUs. The corresponding decrease in the number of resident thread blocks, and thus available compute, causes the 2070 Super to perform similarly to the 3-year-older TITAN Xp. The RTX 4090 has more SMs and a higher clock speed than the A100, but the A100 has greater memory bandwidth. As shown in Section V-B, PFPL is faster on the RTX 4090 than on the A100 on both single- and double-precision inputs. The A100 also has more FP64 units than the RTX 4090. Yet, we do not see a large difference in performance because PFPL, and the other tested compressors, only execute a few floating-point operations in the quantizer. The rest of the quantizer and all other stages exclusively use integer operations.

Our profiling results back up the above findings and show that PFPL is not main-memory bound. After all, it reads the input from main memory only once, performs most of the work while the data resides in shared memory, then writes the output to main memory once. On the A100, we only utilize 15% of the available DRAM throughput while using the majority of the available compute power. The results for the RTX 4090 are similar, but the DRAM utilization is a little higher due to the lower available throughput.

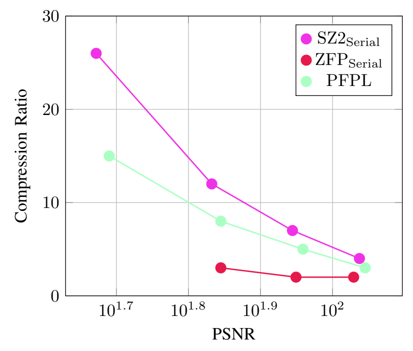
VI. Related Work

This section describes the seven state-of-the-art lossy floating-point compressors [10] with which we compare PFPL.

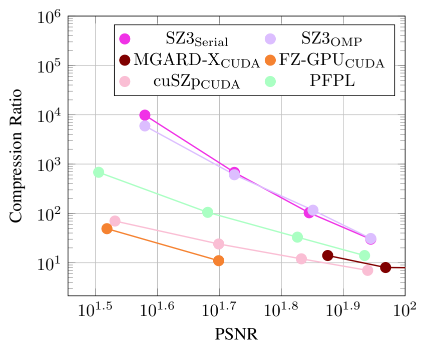
There are four main versions of SZ. They all have an overarching theme of using prediction in their compression pipeline. SZ2 [23] uses Lorenzo prediction [18] and linear regression followed by quantization. SZ3 [24, 26, 36] is an improvement that generally produces better compression ratios with similar throughput. Both SZ2 and SZ3 adopt entropy coding plus lossless compression after the lossy stage (e.g., Huffman [17] followed by GZIP [8] or ZSTD [7]). SZ2 and SZ3 are both CPU-only compressors. cuSZ [33, 34] is a CUDA implementation that employs a different, more GPU-friendly algorithm. It performs Lorenzo prediction and quantization followed by multi-byte Huffman coding. FZ-GPU [35] is a specialized version of cuSZ that fuses multiple kernels together for better throughput. Compared to FZ-GPU, cuSZp [15] yields higher compression ratios and higher decompression throughputs. It splits the data into blocks and then quantizes and predicts the values in all nonzero blocks, which are ultimately compressed by a fixed-length encoder. The fixed-length encoding is implemented using a bit-shuffle operation. Similar to the SZ compressors, we also use quantization as a lossy step, and our lossless stages also rearrange the data but utilizing different transformations. Compared with the SZ compressors, the key advantages of PFPL are (1) guaranteeing the error bound for ABS, REL, and NOA, (2) a higher throughput due to the complete kernel fusion, inlining of outliers, and other optimizations, and (3) full CPU/GPU compatibility.



(a) System 1 geo-mean compression ratio and PSNR on single-precision data for the four ABS error bounds



(b) System 1 geo-mean compression ratio and PSNR on single-precision data for the four REL error bounds



(c) System 1 geo-mean compression ratio and PSNR on single-precision data for the four NOA error bounds

Fig. 16: Compression ratio vs. PSNR results for the 3 error-bound types

ZFP [11, 27] is a widely used tranform-based compressor. It is specifically designed for in-memory array compression and supports on-the-fly random-access decompression. ZFP splits the input into blocks, converts each value into an integer, performs the aforementioned decorrelation, reorders the data, and converts the values from twos-complement to negabinary representation. Then, it groups the bits from most to least significant. Finally, the shuffled bits are losslessly compressed. Our approach only has a few commonalities with ZFP (e.g., converting to negabinary format).

MGARD [25] is the only other compressor that supports compression and decompression across CPUs and GPUs. This compressor uses multigrid hierarchical data refactoring to decompose the data and recompose it to a specified accuracy via selective loading based on the hierarchy after decomposition. Compared with MGARD, PFPL exhibits significantly higher compression ratios and throughput on both CPUs and GPUs but does not support progressive operation.

SPERR [21], a successor of SPECK [31], uses advanced wavelet transforms that are applied recursively to the input. SPERR detects outliers that do not meet the error bound and stores correction factors for those values. The coded wavelet coefficients and the outliers are compressed using ZSTD. Like some of the other transformation-based compressors, our approach does not have much in common with SPERR. We do, however, keep track of outliers that cannot be correctly handled within the error bound. Unlike SPERR, we leave these outliers inline with the rest of the values.

VII. Summary and Conclusions

We developed the Portable Floating-Point Lossy (PFPL) compressor to address three critical issues in existing lossy compressors: violated error bounds, missing CPU/GPU support, and low compression ratio or throughput.

We evaluated 8 lossy compressors on 2 systems using 7 single- and 3 double-precision input sets from the SDRBench suite, a total of 89 files. PFPL yields the highest CPU-parallel compression and decompression throughput compared to the codes from the literature. Furthermore, it outperforms all tested GPU codes in compression ratio. Consequently, PFPL is on the Pareto front in all sets of results, which is otherwise only the case for SZ, a CPU-only compressor that is orders of magnitude slower than PFPL on the GPU. PFPL achieves high throughputs and compression ratios even though it supports key features that the other lossy compressors lack.

- It is fully CPU/GPU compatible, which is otherwise only the case for MGARD-X, but MGARD-X does not support REL and does not guarantee the error bound.
- It supports the ABS, REL, and NOA error-bound types, which is otherwise only the case for SZ2, but SZ2 does not guarantee the error bound on REL.
- It guarantees the error bound for all supported error-bound types, which is otherwise only the case for SZ3, but SZ3 does not support REL.
- It combines a high throughput with a good compression ratio whereas the other studied tools either compress well or deliver a high throughput but not both.

Hence, PFPL is currently the only CPU/GPU compatible lossy compressor that guarantees point-wise absolute, relative, and normalized-absolute error bounds. We hope that PFPL will enable more scientists to lossily compress their data with confidence and promote the inclusion of error-bound guarantees in other lossy compressors.

VIII. Acknowledgments

This work has been supported in part by the Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under Award Numbers DE-SC0022223 and DE-AC02-06CH11357 as well as by an equipment donation from NVIDIA Corporation. This material was supported by the National Science Foundation under Grant Numbers CSSI/OAC #2311875 and CSSI/OAC #2104023.

References

[1] SZ Website. https://github.com/szcompressor/SZ.
[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[3] Noushin Azami, Alex Fallin, Brandon Burtchell, Andrew Rodriguez, Benila Jerald, Yiqian Liu, and Martin Burtscher. LC Git Repository. https://github.com/burtscher/LC-framework, 2025. Accessed: 2025-01-07.

[4] Allison H. Baker, Alexander Pinard, and Dorit M. Hammerling. On a Structural Similarity Index Approach for Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics*, 30(9):6261–6274, 2024.

[5] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

[6] Jieyang Chen, Lipeng Wan, Xin Liang, Ben Whitney, Qing Liu, David Pugmire, Nicholas Thompson, Jong Youl Choi, Matthew Wolf, Todd Munson, Ian Foster, and Scott Klasky. Accelerating Multigrid-based Hierarchical Scientific Data Refactoring on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 859–868, 2021.

[7] Yann Collet and Murray Kucherawy. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, February 2021.

[8] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[9] Sheng Di and Franck Cappello. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 730–739, Los Alamitos, CA, USA, may 2016. IEEE Computer Society.

[10] Sheng Di, Jinyang Liu, Kai Zhao, Xin Liang, Robert Underwood, Zhaorui Zhang, Milan Shah, Yafan Huang, Jiajun Huang, Xiaodong Yu, Congrong Ren, Hanqi Guo, Grant Wilkins, Dingwen Tao, Jiannan Tian, Sian Jin, Zizhe Jian, Daoce Wang, MD Hasanur Rahman, Boyuan Zhang, Jon C. Calhoun, Guanpeng Li, Kazutomo Yoshii, Khalid Ayed Alharthi, and Franck Cappello. A survey on error-bounded lossy compression for scientific datasets. https://arxiv.org/abs/2404.02840, 2024.

[11] James Diffenderfer, Alyson L. Fox, Jeffrey A. Hittinger, Geoffrey Sanders, and Peter G. Lindstrom. Error Analysis of ZFP Compression for Floating-Point Data. *SIAM Journal on Scientific Computing*, 41(3):A1867–A1898, 2019.

[12] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. PFPL Git Repository. https://github.com/burtscher/PFPL, 2025. Accessed: 2025-02-10.

[13] Alex Fallin and Martin Burtscher. Lessons learned on the path to guaranteeing the error bound in lossy quantizers, 2024.

[14] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[15] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck 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*, SC'23, Denver, CO, USA, 2023. Association for Computing Machinery.

[16] Yafan Huang, Kai Zhao, Sheng Di, Guanpeng Li, Maxim Dmitriev, Thierry-Laurent D Tonellot, and Franck 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)*, pages 651–661. IEEE, 2023.

[17] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[18] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Comput. Graph. Forum*, 22:343–348, 09 2003.

[19] F. Jager. Delta Modulation — A Method of PCM Transmission Using the One Unit Code. *Philips Res. Repts.*, 7, 01 1952.

[20] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. C. Bates, G. Danabasoglu, J. Edwards, M. Holland, P. Kushner, J.-F. Lamarque, D. Lawrence, K. Lindsay, A. Middleton, E. Munoz, R. Neale, K. Oleson, L. Polvani, and M. Vertenstein. ”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*, 96(8):1333 – 1349, 2015.

[21] Shaomeng Li, Peter Lindstrom, and John Clyne. Lossy Scientific Data Compression With SPERR. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1007–1017, 2023.

[22] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. An efficient transformation scheme for lossy data compression with point-wise relative error bound. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 179–189, 2018.

[23] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447, 2018.

[24] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447, 2018.

[25] Xin Liang, Ben Whitney, Jieyang Chen, Lipeng Wan,

Qing Liu, Dingwen Tao, James Kress, David Pugmire, Matthew Wolf, Norbert Podhorszki, and Scott Klasky. MGARD+: Optimizing Multilevel Methods for Error-Bounded Scientific Data Reduction. *IEEE Transactions on Computers*, 71(7):1522–1536, 2022.

[26] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M. Gok, Jiannan Tian, Junjing Deng, Jon C. Calhoun, Dingwen Tao, Zizhong Chen, and Franck Cappello. SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors. *IEEE Transactions on Big Data*, 9(2):485–498, 2023.

[27] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014.

[28] K. Masui, M. Amiri, L. Connor, M. Deng, M. Fandino, C. Höfer, M. Halpern, D. Hanna, A.D. Hincks, G. Hinshaw, J.M. Parra, L.B. Newburgh, J.R. Shaw, and K. Vanderlinde. A compression scheme for radio data in high performance computing. *Astronomy and Computing*, 12:181–190, 2015.

[29] D. Merrill and M. Garland. Single-pass parallel prefix scan with decoupled look-back. Technical Report NVR-2016-002, NVIDIA, March 2016.

[30] SDRBench Inputs, https://sdrbench.github.io/, 2023.

[31] Xiaoli Tang and William A. Pearlman. *Three-Dimensional Wavelet-Based Compression of Hyperspectral Images*, pages 273–308. Springer US, Boston, MA, 2006.

[32] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck 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 (IPDPS)*, pages 1129–1139, 2017.

[33] Jiannan Tian, Sheng Di, Xiaodong Yu, Cody Rivera, Kai Zhao, Sian Jin, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. Optimizing Error-Bounded Lossy Compression for Scientific Data on GPUs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 283–293, Los Alamitos, CA, USA, September 2021. IEEE Computer Society.

[34] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 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*, PACT '20, page 3–15, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck 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*, HPDC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1643–1654, 2021.

[37] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. SDR-Bench: Scientific Data Reduction Benchmark for Lossy Compressors. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2716–2724, 2020.