Cortado: Progressive Retrieval of Lossily Compressed Data with Guaranteed Error Bounds

Brandon Alexander Burtchell
Department of Computer Science
Texas State University
San Marcos, TX, USA
burtchell@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

***Abstract*—Many important scientific datasets are too large to be downloaded for local processing. Error-bounded lossy compression can help, especially because it allows trading off higher compression ratios for lower data quality. However, this forces providers to preselect the error bounds at which to offer the data. Moreover, users who download a dataset but later need a higher-fidelity version must download an entirely new version. As a remedy, we introduce Cortado, a novel technique to progressively refine the error bound of a dataset. It is unique in that it never violates the requested error bound, refinement of an existing file always yields bit-for-bit the same result as downloading a fresh version, and each refinement requires only a single encoding/decoding phase. We compare Cortado with state-of-the-art approaches and find that it is the only progressive compressor that guarantees the evaluated error bounds on all tested inputs while achieving the highest throughputs, comparable reconstruction quality, and on-par compression ratios.**

***Index Terms*—data compression, lossy compression, progressive compression, floating-point data**

I. Introduction

Scientific floating-point data is often compressed to reduce storage space and transmission times. Unfortunately, such data tends to not compress well using lossless approaches, typically yielding no more than a factor of two in savings [1]. As a consequence, error-bounded lossy compression has been proposed to achieve much higher compression ratios at the cost of a controlled degree of information loss [2]. However, using lossy compression leaves data providers with the dilemma of choosing the error bounds at which to store the data. Furthermore, users who download a dataset but later need a higher-fidelity version are forced to download an entirely new version—which includes redundant information from the original version. These issues are exacerbated in big data applications, where the size of datasets can easily exceed terabytes or petabytes.

Progressive compression/retrieval methods address these issues by recoding the data needed to refine a file [3]. Here, refinement refers to incremental improvement of the data quality (i.e., lowering the error). By differentially encoding multiple versions in a hierarchical manner, users can refine a current version by downloading just enough information to improve it—minimizing redundancy. Such "progressive" operation can be crucial and is listed as one of five Priority Research Directions in the 2021 DOE ASCR workshop report [4]. Existing methods generally progress in terms of *resolution* (number of data points) or *precision* (number of bits per data point) [5]. However, the prior approaches leave important opportunities for improvement.

First, some progressive retrieval methods—typically residual-based methods—require multiple decompression passes to refine an input [6]–[8]. If the intervals of the progression sequence are too close and the sequence is too long, decoding to a high-fidelity representation may require significant computational overhead compared to the single decompression pass of traditional, non-progressive retrieval.

Second, some progressive retrieval approaches introduce and propagate floating-point error [6]. This is also an issue with some conventional lossy floating-point compressors due to lack of associativity, rounding issues, or compiler optimizations. In fact, a majority of the state-of-the-art scientific lossy compressors do not successfully guarantee the error bound [9]. Therefore, progressive methods that are based on such compressors will suffer the same issues.

Third, some progressive compressors encode the data at a small, predetermined set of error bounds at compression time, leaving users with access to only these few representations. Thus, these progressive compressors echo one of the original issues with traditional lossy compression.

In this paper, we introduce **Cortado**, a new progressive retrieval technique that progresses in terms of guaranteed error bounds (i.e., precision) and addresses the above challenges in the following ways. First, Cortado allows refinement to guaranteed error bounds with a *single decoding phase* per request. Second, Cortado utilizes a quantizer implemented exclusively with integer operations that does not suffer from floating-point error or the propagation thereof. Third, Cortado supports retrieval of near-arbitrary error-bounded versions of a dataset down to the finest representation stored on the server.

Our comparisons with leading GPU and CPU approaches show that Cortado is the only progressive compressor that truly guarantees the user-selected error bound. It does so while delivering the highest encoding and end-to-end retrieval throughput, maintaining competitive compression ratios, and delivering on-par if not better reconstruction quality. The code is open-source and available at github.com/burtscher/Cortado.

This paper makes the following main contributions:

- It explains the proposed guaranteed-error-bound-based progressive retrieval method.
- It discusses how Cortado *requantizes* data to coarser error bounds without error propagation or accumulation.
- It demonstrates Cortado's strict error-bound guarantees, high throughput, good compression ratio, and comparable reconstruction quality relative to the state of the art.

The rest of this paper is structured as follows. Section II provides an overview of the LC framework upon which Cortado is based as well as the fundamentals of conventional lossy and progressive compression. Section III details the elements of Cortado. Section IV summarizes related work. Section V outlines the evaluation methodology. Section VI presents and discusses the results. Section VII summarizes and concludes.

II. Background

This section describes the LC framework as well as the fundamentals of lossy compression and progressive compression.

A. LC Framework

Cortado's compressor is based on components from the LC framework [10]. LC generates compression algorithms from a library of data transformations called components that are chained together to form pipelines. Fig. 1 visualizes how a compression pipeline accepts and passes an input through its components to yield a compressed output. The corresponding decompression algorithm simply performs the inverse transformations in reverse order.

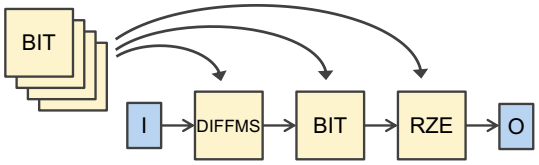

Fig. 1. A compression pipeline with 3 stages generated by the LC framework (I = input, O = output)

LC can be used to exhaustively search through combinations of components to find a suitable compressor for a file or target domain. Through such a process, LC has helped generate several recent lossless compressors, including LICO [11], SPratio, SPspeed, DPratio, and DPspeed [1], as well as lossy compressors such as cuSZ-Hi [12], PFPL [13], and SLEEK [14]. LC was also used to create PRISM [15], a state-of-the-art progressive compressor. We similarly employed the LC framework to find a suitable lossless compression pipeline for Cortado (see Section III-B).

LC compressors operate largely independently on 16 kB chunks of data, run on CPUs and GPUs, and produce bit-for-bit the same compressed and decompressed files on both types of devices. On the GPU, the encoder uses Merrill and Garland's decoupled look-back technique [16] to propagate the cumulative size of prior compressed chunks to the next thread block so it knows where to write its output.

B. Error-bounded Lossy Compression

Point-wise error bounds ensure that each original value and corresponding decompressed value do not deviate by more than a specified threshold. The most frequently used point-wise error-bound types are absolute (ABS) [17], normalized absolute (NOA), and relative (REL) [18] (not covered).

The point-wise absolute error of a value is the difference between the original value and its reconstructed value. For a data value x, we express its error as $e_{abs} = |x_{original} - x_{reconstructed}|$. To guarantee an absolute error bound of ε, each value in the reconstructed file must satisfy $e_{abs} \leq \varepsilon$.

The point-wise *normalized* absolute error of a value is the point-wise absolute error normalized by the value range of the file $R = x_{max} - x_{min}$. Thus, the NOA error of a data value x is $e_{noa} = |\frac{e_{abs}}{R}|$. With a NOA error bound of ε, each value of the reconstructed file must satisfy $e_{noa} \leq \varepsilon$.

Cortado supports both ABS and NOA error bounds. The rest of the paper focuses on NOA error bounds so as not to introduce bias between datasets containing values at different ranges and distributions.

C. Progressive Compression

Progressive compressors can be broadly categorized as precision- or resolution-based—though many combine ideas from both categories. This subsection summarizes the common techniques. Specific compressors are discussed in Section IV.

Precision-based methods progress in terms of the number of bits used to represent each data point. For example, this can be achieved via bitplaning (a.k.a. bit shuffling), where the most significant bit of each value is output as a bitplane, followed by the second bit of each value and so on. The user can download data by fetching each bitplane in order—progressively refining the precision of all values. Bitplaning has been utilized in traditional scentific data compressors (e.g., PFPL [13], SPERR [19], SPratio [1], ZFP [20]) for its ability to improve compression by, for example, grouping the leading zeros of small values. Alternatively, precision progression can also be achieved with residuals [6], where an input is quantized at multiple error bounds and, instead of saving each copy with redundant data, the subsequent error-bounded versions are represented only by the residual (i.e., difference) between itself and the coarser version. Typically, these residuals are much easier to compress than the raw values they yield.

In contrast, resolution-based methods progress in terms of the number of data points. Depending on the application, previous methods have used resolution trees [21], [22], mesh deformation [23], [24], etc. Many leading approaches for scientific data employ repeated multilinear interpolation to preserve more spacial correlation between points [25], [26]. In short, the data is treated as a piecewise multilinear function defined on the input grid. The data is refactored into levels of increasing resolution by computing the interpolant at successively larger strides and encoding coefficients based on the difference between the original values and the interpolant at each level.

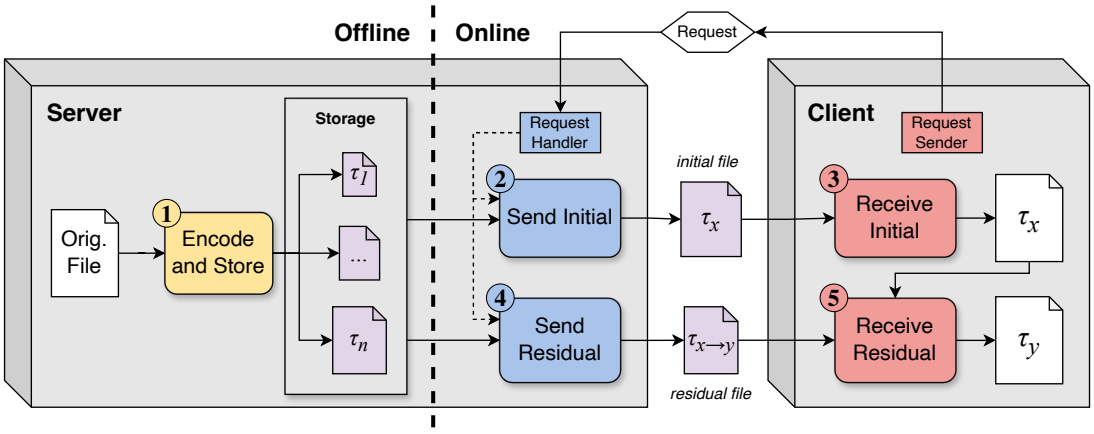

Fig. 2. Overview of Cortado: First, the server encodes and stores a file at one or a few error bounds (1). When a client requests a file at an initial error bound, the server generates and sends it (2) and the client receives it (3). Subsequent client requests for finer error bounds are satisfied by computing and sending the residual between the client's current error bound and the requested error bound (4). The client receives this residual and applies it (5). Compressed files are shaded in violet. Uncompressed files are unshaded. τ signifies an error bound.

III. Approach

Fig. 2 provides a high-level overview of Cortado. It shows the delegation of server and client duties in separate boxes. The dashed line demarcates the aspects of Cortado that occur offline (when the data provider prepares the data) versus online (each client request).

Suppose the server has a file to compress and make available via progressive retrieval. Starting with the original file on the left, Cortado first encodes and stores (1) the file at one or a few error bounds. Only one file is necessary, but multiple files encoded near commonly-requested error bounds can reduce the average file-read time when the server fulfills a request.

When a client requests a file at a particular error bound, the server handles the request in one of two ways. If the client is requesting the file for the first time, the server generates and sends the *initial file* based on the next finer available error bound (2), which is simply the file quantized and compressed at the client's requested error bound. The client subsequently receives the initial file and decompresses it (3). If instead the client is requesting to refine a previously downloaded file, the server computes and sends the *residual file* (4). As the client receives the residual, it subsequently decompresses it and applies it to the existing copy (5).

There are important opportunities for latency hiding during a server-to-client data transfer, especially for large inputs. Mainly, after the server's GPU has compressed a set of data chunks in parallel, it can send the compressed chunks over the network while compressing the next set of chunks. Accordingly, the client can decompress and process each chunk as it is received. The residual send/receive has further opportunities for latency hiding as discussed in Subsection III-E.

The rest of this section details Cortado's key aspects. First, we describe the quantization technique and lossless compression algorithm that enable the approach. Then, we discuss how these underlying elements are used in Cortado's main operations: server encode and store (1), initial send/receive (2)/(3), and residual send/receive (4)/(5).

A. Quantization

Cortado's quantizers are based on SLEEK's quantizer, which, unlike conventional quantization methods, does not suffer from overflow or floating-point rounding errors, thus guaranteeing the error bound without the need for verification or storing outliers [14]. This is accomplished by emulating all floating-point operations with integer instructions and lowering the error bound to the nearest power of two. Due to these characteristics, the quantizer's output bin numbers can be dequantized and requantized in terms of another error bound without introducing any additional information loss. In other words, requantization always yields bit-for-bit the same result as direct quantization of the original data. We leverage this crucial feature when computing the residual (arithmetic difference) between bin numbers at different error bounds.

Unfortunately, SLEEK's quantizer maps floats to integer bins that center around zero to ensure zero is reproducible. This is a desirable characteristic for scientific data, since many files contain values that are close to zero. However, zero-centered bins can result in errors when an input's error bound is refined and the bin boundaries of the old and new error bounds do not align. To avoid this issue while maintaining the ability to recover zero, Cortado uses different bin boundaries for the server and the client.

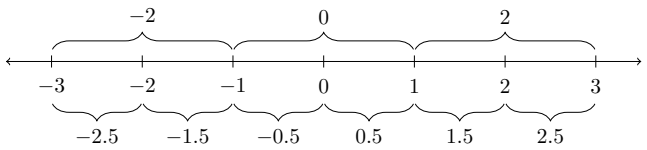

Fig. 3. Bin ranges and corresponding decoded values for the server (below) and client (above)

Fig. 3 visualizes an example of server bin ranges (below) and client bin ranges (above) on a number line. On the server, the bin sizes are halved and not zero-centered (i.e., bin boundaries are at zero so zero is not recoverable). On the client, the (de)quantizer maps the server's bin numbers to ones that are double the size—but zero-centered. Therefore, when the client decodes, the server bins with boundaries on zero (e.g., $[-1, 0)$ and $[0, 1)$) will be grouped into a single client bin (e.g., $[-1, 1)$) that decodes to zero.

Now that server bins are not zero-centered, they can be requantized to other powers of two and will always have aligned bin boundaries. For the rest of the paper, we will use the client error bound to refer to both itself and the underlying non-zero-centered halved server error bound.

B. Compression Algorithm

Cortado compresses server files (1), initial files (2), and residual files (4) using the "DIFFMS_4 BIT_4 RZE_1" algorithm from the LC framework. We selected this compression pipeline as it yields the highest geometric-mean compression ratio for server files, initial files, and residual files out of the 133,308 three-stage pipelines we tested on a subset of the evaluated inputs (see Section V).

DIFFMS_4 operates on 4-byte words (i.e., `int`) and computes the difference sequence (a.k.a. "delta modulation") by subracting the previous value from the current value and outputting the result in magnitude-sign format. If the input data is smooth, the output will be values close to zero.

BIT_4 is a bit shuffler that operates on 4-byte words and takes the most significant bit of each value in the input and outputs them together. Then it does the same with the second bit and so on. Ideally, this component shuffles the leading zero bits of values towards the beginning, producing many zero values.

Lastly, RZE_1 operates on bytes and creates a bitmap where each bit specifies whether the corresponding byte in the input is a zero or not. It outputs the non-zero bytes and a compressed version of the bitmap that is repeatedly compressed with the same algorithm. Since this component operates on BIT_4's output, the data chunks likely contain many zero bytes, which can be marked in the bitmap instead of being stored in full.

The client's receive operations (3, 5) decompress the received files using the corresponding decompression algorithm: "iRZE_1 iBIT_4 iDIFFMS_4", where each component performs the inverse operation. iRZE_1 decompresses its bitmap and decompresses the zeros accordingly. Then, iBIT_4 unshuffles the bits. Lastly, iDIFFMS_4 computes the prefix sum on the difference sequence to recover the original values.

The remaining subsections describe how Cortado's compression, decompression, quantization, and dequantization routines can be chained together to fulfill client requests for data at near-arbitrary error bounds. In LC, each of these routines are typically independent and defined as their own GPU kernels. For Cortado, we fuse the underlying kernels and optimize their memory usage to maximize performance.

C. Server Encode and Store

The encode and store (1) operation is performed offline. Equation 1 formally defines the constraints on the set of error bounds for the server files.

$$\tau_S = (\tau_1, \tau_2, \ldots, \tau_n), \text{ where } \begin{cases} \tau_1 > \tau_2 > \cdots > \tau_n \\ \infty \geq \tau_i \geq 0 \end{cases} \quad (1)$$

For example, if a server stored $n = 3$ representations with error bounds $\tau_S = (2^{-4}, 2^{-10}, 2^{-16})$, clients can access the file at any error bound in the range $[2^{-16}, \infty]$. Recall that the server's quantizer halves each error bound and offsets the bins to have boundaries on zero. Although clients may request arbitrary error bounds, the server will always provide the next available power of two to guarantee the error bound. In practice, the server would carefully choose a set of error bounds depending on storage constraints and expected client requests, since the size of each compressed server file will have a great effect on the file-read time whenever it is used to prepare an initial file or a residual file.

Optionally, one of the provided server representations can be the losslessly compressed file (i.e., $\tau_n = 0$). Although this typically requires substantial storage overhead compared to a lossy representation, it allows the client to access the lossless representation as a contingency.

D. Initial Send/Receive

Suppose a client requests an initial file at error bound τ_x. The initial send (2) consists of the following steps. The server begins by finding the file with the next finer τ_i in τ_S. Then, it decompresses and dequantizes this file. Next, the server requantizes (i.e., coarsens) it in terms of τ_x, compresses the result, and sends it to the client.

Requantization improves compression since files at coarser error bounds generally contain less information and compress more. Since Cortado converts from server to client bins anyway, coarsening improves compression and reduces transfer times at essentially no additional runtime (just a few extra integer arithmetic instructions per value with an embarrassingly-parallel implementation). The same is true for the residual send (4).

The initial receive (3) is less involved. When the client receives the initial file, it decompresses and dequantizes it.

E. Residual Send/Receive

Now suppose the client already has a file at an error bound τ_x. Recall that the client can refine the file to any available error bound τ_y as long as $\tau_x > \tau_y \geq \tau_n$, where τ_n is the tightest error bound offered by the server. The residual send (4) performs the following operations. First, the server recomputes the file at the client's current error bound τ_x and then requantizes this τ_x version in terms of the requested error bound τ_y. Next, the server similarly prepares a file at the requested error bound τ_y. Lastly, for each value, the server computes the difference between the files at τ_x and τ_y, yielding the residual file. The server then compresses the residual file and sends it to the client.

The residual receive (5) consists of the following steps. While the residual send is performed on the server, the client requantizes its current τ_x version in terms of τ_y. This step has no dependencies, so it can be performed asynchronously while the server is preparing the residual file. When the client receives the residual file, it decompresses it and applies it to its requantized version by adding the received differences.

IV. Related Work

Magri and Lindstrom [6] introduce a general framework for progressive compression that guarantees successively tighter error bounds at every progression level. Each level only needs to encode the difference between the previous level's error-bounded reconstruction and the current level's tighter error-bounded reconstruction. The approach is general because any lossy compressor can be used to encode/decode each level. However, the example compressors the authors compare in their case study (e.g., ZFP [20], SZ3 [27], SPERR [19]) are all floating-point-based, and thus the framework can propagate error (albeit a provably manageable amount). Furthermore, this approach presents a tradeoff between the granularity of the progression and the overhead of storage and compute time. A finer granularity stores more residuals and leads to more decoding passes per retrieval. Cortado shares the residual-based nature of Magri and Lindstrom's approach but differs in key ways. Mainly, instead of the encoded file consisting of predetermined residuals, Cortado computes residuals *on the fly* when a client makes a request. Therefore, Cortado only requires a single decoding/re-encoding pass per request, irrespective of the original and target error bound. Also, Cortado's quantizer does not propagate floating-point error unlike other state-of-the-art lossy compressors that could be used in this framework. While we describe Cortado's single lossless compression algorithm in Section III, any other lossless compressor could theoretically be swapped in.

MGARD [25], [26] refactors the data via multilinear interpolation and compresses each layer. MGARD+ [28] makes enhancements via level-wise coefficient quantization at different error bounds. PMGARD [7], [29] (a.k.a. MDR) introduces progressive retrieval in terms of both precision and resolution simultaneously by optimizing MGARD's data refactoring technique and performing bitplane encoding. PMGARD requires multiple decoding passes to apply each set of coefficients to the data. It uses an optimized data loader that decides the necessary bitplanes each layer needs to satisfy a client's request (e.g., error bound, compressed size). Notably, the MGARD family can be configured to control point-wise error bounds as well as derived quantities of interest (QoIs).

HP-MDR [8] introduces a GPU-optimized bitplane encoder and utilizes GPU-MGARD [30] kernels to port PMGARD to GPUs. HP-MDR further optimizes end-to-end retrieval by hiding the latency of its operations via pipelining. Like PMGARD, HP-MDR can optionally control QoIs.

Several recent progressive compressors utilize similar interpolation-based techniques as the MGARD family. IPComp [31] is based on SZ3 [27] and uses interpolation and predictive negabinary bitplane encoding to support resolution- and precision-based progressive retrieval. However, IPComp currently only supports CPU execution and thus has limited throughput potential. PRISM [15] is a GPU-based approach that utilizes an interpolation-based quantizer followed by bitplaning to support resolution- and precision-based progressive retrieval. Like PMGARD, both IPComp and PRISM use an optimized data loader. Cortado is residual-based, and thus fundamentally differs from these approaches.

Hoang et al. [21] build a precision-resolution tree by restructuring a scalar-field input into a tree and then bitplaning each node. We proposed a similar resolution-based tree-building approach that encodes the sums of groups of values to allow users to recover averages of the data when progressively decoding [22]. Both approaches encode the data at several intervals, requiring multiple decoding passes when retrieving a new representation. However, the latter cannot guarantee error bounds at any of the intermediate progressive levels.

V. Experimental Methodology

We test Cortado on datasets from SDRBench [32], [33], a suite that provides snapshots of scientific datasets from a number of domains, including hurricane simulations and cosmology. The files of each dataset are described in Table I. Note that the EXAALT dataset contains files with different dimensions, but we still treat it as a single dataset in the evaluations. All tested files consist of single-precision floating-point values.

TABLE I
Input datasets

Dataset	# Files	File Dimensions	Filesize (MB)
CESM (3D)	33	26 × 1,800 × 3,600	642.7
EXAALT (1)	3	5,423 × 3,137	64.9
EXAALT (2)	3	83 × 1,077,290	341.1
ISABEL	13	100 × 500 × 500	19.1
NYX	6	512 × 512 × 512	512.0

We ran Cortado on each input 9 times and collect the compression ratio and median throughputs. The results present the geometric mean of the geometric mean of each dataset so as not to overemphasize results from datasets with more files.

We define the sequence of requested error bounds in terms of the *granularity* of the progression Δ as $\tau_R^{\Delta} = (\tau_1, \tau_2, ..., \tau_m)$ where $\tau_i = 2^{-\Delta i}$ for $i = [1..m]$ and $m = 32/\Delta$. We borrow this way of defining an error bound progression from Magri and Lindstrom [6]. With the exception of the comparison in Subsection VI-A, all experiments use $\Delta = 2$, yielding a request progression $\tau_R^{\Delta=2} = (2^{-2}, 2^{-4}, ..., 2^{-32})$. The error bounds from this progression are then normalized to the range of values of each file (i.e., NOA).

Our case study in Subsection VI-A compares Cortado to the state of the art. On the GPU, we compare with HP-MDR and PRISM. On the CPU, we compare with IPcomp and M-SZ3—the SZ3 implementation of Magri and Lindstrom's framework. We borrow the implementation of M-SZ3 from the

HP-MDR artifact [8], [34], which is parallelized with OpenMP and configured to use 32 threads.

We ran the GPU codes on an NVIDIA GeForce RTX 4090 and an NVIDIA A100 (40 GB). We compiled them with `nvcc` for the respective compute capabilities. We ran the CPU codes on an AMD Ryzen Threadripper 2950X (16 cores, 2 threads per core) with 48 GB of main memory.

VI. Results

This section presents the experimental results. First, we compare Cortado to state-of-the-art progressive compressors. Then, we evaluate the three main operations of Cortado independently. The results in this section focus on the computation time to make them independent of the network speed.

A. Comparisons

We compare Cortado to several leading progressive retrieval methods. In particular, we focus on end-to-end metrics across the lifespan of a file that is progressively refined by the error bounds (EBs) in $\tau_R^{\Delta=4}$, meaning a client initially requests a file at 2^{-4}, then refines it to 2^{-8}, then 2^{-12}, and so on. We configure PRISM to encode each file to support retrieval up to a NOA EB of 2^{-32}. We configure Cortado to use a server file progression of $\tau_S = (2^{-10}, 2^{-20}, 2^{-32})$.

Table II summarizes each compressor's ability to strictly guarantee each EB in $\tau_R^{\Delta=4}$ on the files of each dataset. We list the number of files in each dataset next to its name in parenthesis. Compressors that guarantee an EB for all files in a dataset have the corresponding cell text bolded and colored blue. HP-MDR is missing CESM results since it failed on these inputs due to out-of-memory errors (denoted by "×"). HP-MDR is missing counts for the tighter EBs because it exits when the EB is violated. Similarly, IPComp limits single-precision inputs to an encoded NOA EB of 10^{-6} so we stop its progression at 2^{-16}, which is the last EB in $\tau_R^{\Delta=4}$ within this limit. We denote these unsupported EBs with "–".

TABLE II
Summary of Strictly-Guaranteed EBs across Compressors

		2^{-4}	2^{-8}	2^{-12}	2^{-16}	2^{-20}	2^{-24}	2^{-28}	2^{-32}
Cortado	CESM (33)	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**
	EXAALT (6)	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
	ISABEL (13)	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**
	NYX (6)	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
PRISM	CESM (33)	**33**	**33**	**33**	32	31	0	0	0
	EXAALT (6)	1	1	1	1	1	0	0	0
	ISABEL (13)	10	7	8	7	7	0	0	0
	NYX (6)	**6**	2	0	0	0	0	0	0
HP-MDR	CESM (33)	×	×	×	×	×	×	×	×
	EXAALT (6)	**6**	**6**	**6**	**6**	5	–	–	–
	ISABEL (13)	**13**	**13**	**13**	**13**	**13**	–	–	–
	NYX (6)	**6**	**6**	**6**	**6**	**6**	–	–	–
M-SZ3	CESM (33)	19	19	18	18	18	18	17	17
	EXAALT (6)	1	1	1	1	1	0	0	1
	ISABEL (13)	8	8	8	8	8	8	8	8
	NYX (6)	0	0	0	0	0	0	0	0
IPComp	CESM (33)	19	19	19	19	–	–	–	–
	EXAALT (6)	3	3	3	3	–	–	–	–
	ISABEL (13)	8	8	8	8	–	–	–	–
	NYX (6)	0	0	0	0	–	–	–	–

To the best of our knowledge, PRISM and M-SZ3 should support strict guarantees on tighter EBs. However, PRISM violates the EB on all tested files in all datasets starting at 2^{-24}. Interestingly, for M-SZ3, the dataset itself seems to have the biggest influence, with the number of files with guaranteed EBs remaining relatively constant along the progression. Nonetheless, M-SZ3 never guarantees all EBs for all files in each dataset. For the EBs HP-MDR and IPComp do support, they also fail to provide guarantees on some files. Generally, all compressors except Cortado particularly struggle to guarantee EBs for NYX inputs. Overall, Cortado is the only compressor that is able to guarantee all EBs for all files—even at the tightest tested EBs that others fail on or do not support.

Of course, guaranteed EBs are not the only signifier of quality since even a single value with a violation invalidates an entire file, which may be acceptable in some scenarios. Thus, we also study the overall reconstruction quality via the Peak Signal-to-Noise Ratio (PSNR) in Fig. 4. Higher PSNR values are better. On one input from CESM and two inputs from EXAALT, Cortado achieves lossless compression at some of the tightest EBs (starting at 2^{-24} for CESM and at 2^{-28} for EXAALT). This yields a PSNR of ∞, which would prevent the computation of the geometric mean. As a remedy, we overwrite ∞ with a PSNR of 350, which is just above the highest-observed finite PSNR for these inputs (328). With this adjustment, we can still observe general PSNR trends without omitting outliers or introducing significant bias.

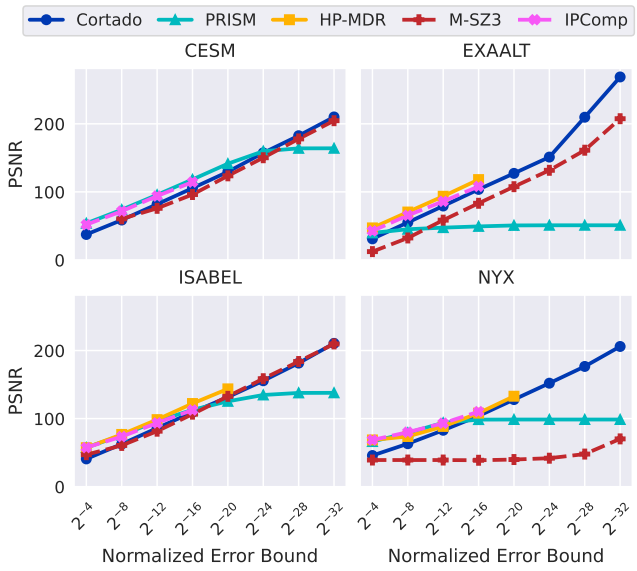

Fig. 4. Peak Signal-to-Noise Ratio

Overall, the PSNR results show that Cortado is always at least on-par with other compressors. Cortado, HP-MDR, IPComp, and M-SZ3 (except on NYX) all follow a similar linear improvement as the EB tightens. In contrast, PRISM falls off this line and plateaus early at different EBs for each dataset, suggesting that it is sensitive to different data characteristics and fails to preserve the quality at tighter EBs. Like with the guaranteed EBs, Cortado is the only compressor that maintains a linear improvement of PSNR for all tested EBs—even beyond where HP-MDR's and IPComp's support

ends—and all tested datasets.

Fig. 5 presents the compression ratio (CR) of each compressor's encoded file (collection of server files for Cortado). The relative CRs are consistent across datasets, with the exception of M-SZ3 on the NYX inputs, which exhibits an impressive CR that overshadows the others. However, recall that M-SZ3 fails to guarantee any EB on any NYX input (Table II) and achieves relatively poor PSNR (Fig. 4). Evidently, M-SZ3's underlying compressor (SZ3) is failing to encode important information for NYX inputs. Otherwise, IPComp is the overwhelming leader in encoded CR (though it also does not guarantee all EBs). This is consistent with trends in lossy compression: GPU compressors tend to sacrifice some CR to deliver high speeds via parallelism [2].

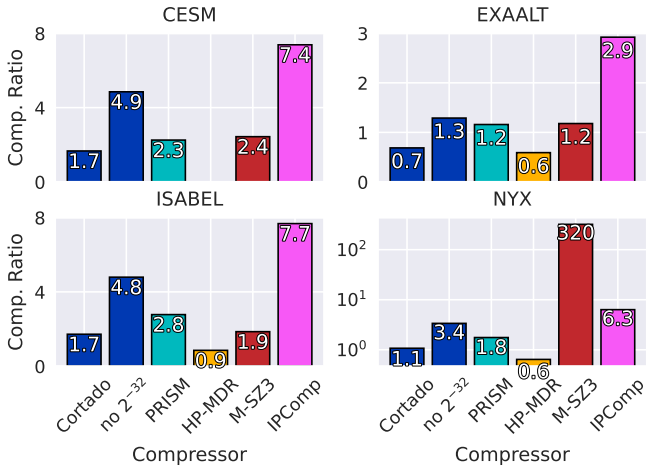

Fig. 5. Server-Encoding Compression Ratio

Recall that IPComp only encodes enough information to support retrieval up to a NOA EB of 10^{-6}. Similarly, PRISM fails to guarantee EBs beyond 2^{-20} and plateaus in PSNR at or before this EB (Fig. 4). To make the comparison to these compressors fairer, we also show the CR when Cortado similarly omits the tightly-bounded server file of 2^{-32} and instead only stores $\tau_S = (2^{-10}, 2^{-20})$. With this configuration (labeled "no 2^{-32}"), Cortado surpasses PRISM's CR.

Fig. 6 compares the encoding throughputs of the compressors on each dataset. Expectedly, the GPU compressors exhibit higher throughputs than the CPU compressors due to their high degree of parallelism. Notably, despite being configured to encode the original file 3 times to produce τ_S, Cortado still maintains the highest average server-encoding throughput. This is further improved if Cortado omits 2^{-32} from τ_S. We attribute this to Cortado's omission of the expensive prediction/interpolation step that the other GPU codes perform.

The CPU compressors are naturally much slower. Particularly, M-SZ3, despite having a parallel implementation, yields the lowest overall encoding throughput since it must run SZ3's encoding/decoding procedures eight times to compute the residuals at each predetermined EB in $\tau_R^{\Delta=4}$.

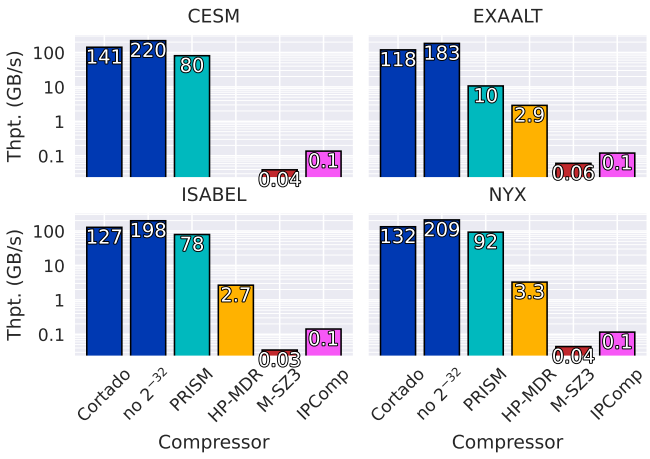

Fig. 6. Server-Encoding Throughput (RTX 4090, Threadripper 2950X)

Fig. 7 shows the end-to-end retrieval CR. Note the differing log-scale y-axes for each dataset. All compressors naturally start high and, with the exception of M-SZ3 on NYX, exponentially decrease in overall CR as the total retrieved data accumulates via refinement to tighter EBs. For the most part, Cortado, PRISM, and IPComp are quite comparable as they each decrease in CR at similar rates. That said, IPComp tends to send slightly less data (yielding higher CRs) than PRISM and Cortado. M-SZ3 tends to have the highest overall retrieval CR. However, as discussed above, M-SZ3 has issues with EB guarantees and reconstruction quality. As we explore below, it is also comparatively slower in terms of end-to-end retrieval throughput. Lastly, we observe that HP-MDR consistently delivers the lowest retrieval CR across EBs.

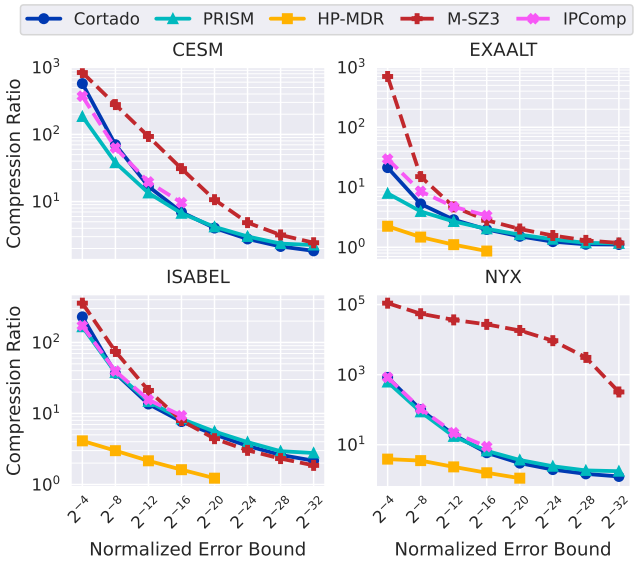

Fig. 7. End-to-end Compression Ratio of Progressive Retrieval

In Cortado, each data transmission consists of both the server send (②, ④) and the client receive (③, ⑤). Naturally,

the client must wait for data to be sent from the server, but this dependency only exists at the LC-chunk level. Thus, once the server compresses a chunk (or multiple chunks in parallel), it can send the chunk(s) while it compresses the remaining chunks. As long as the transmission speed is slower than the throughput of both operations, these operations can have their latency hidden. Since our throughput evaluations omit the network transfer time, we approximate this latency hiding by reporting the throughput of whichever operation is slower.

Fig. 8 shows the end-to-end retrieval throughput, which is the original file size divided by the accumulated runtime of the requests it takes to fulfill an EB. Note that the y-axis uses a logarithmic scale and is shared across dataset subplots. The compressors are consistently ranked in terms of retrieval throughput across datasets, with Cortado being the fastest and IPComp the slowest. This is similar to the server-encoding throughputs, where the GPU approaches are in a different class of performance than the CPU approaches.

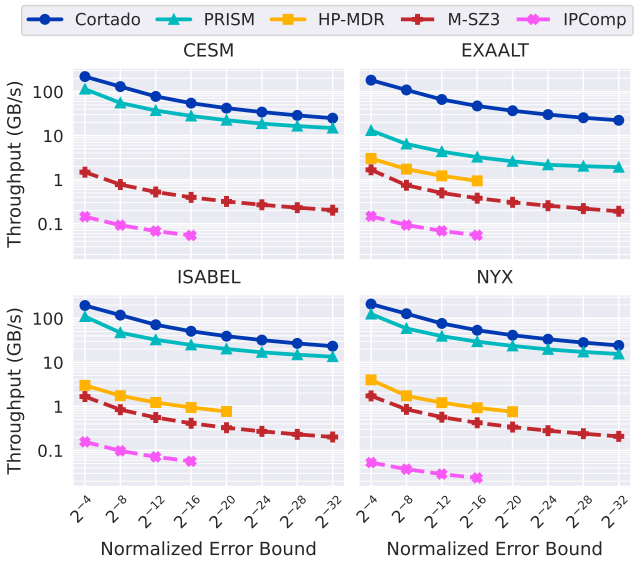

Fig. 8. End-to-end Throughput of Progressive Retrieval (RTX 4090, Threadripper 2950X)

Unlike server encoding, M-SZ3 is faster than IPComp for retrieval, which we attribute to M-SZ3's multi-threaded OpenMP implementation and the fact that Magri and Lindstrom's framework performs less work in each decompression pass than its corresponding compression procedure. Comparatively, IPComp exhibits a relatively symmetric encoding and decoding throughput.

Overall, Cortado is the only tested compressor that strictly guarantees all EBs on all inputs while achieving comparable PSNR and end-to-end retrieval CR. Moreover, Cortado surpasses other solutions in terms of encoding and retrieval throughput. The remaining subsections independently evaluate Cortado's three main operations.

B. Server Encode and Store

The server-encode operation (1) is essentially a conventional lossy compression, albeit at the non-zero-centered and halved server EBs. Fig. 9 shows the CR for each EB in the $\tau_R^{\Delta=2}$ progression. Note that the y-axis uses a logarithmic scale. Recall that, in practice, a server would only store a file at a few EBs. The CRs exhibited by this lossy compressor start quite high and expectedly decrease as the EB tightens.

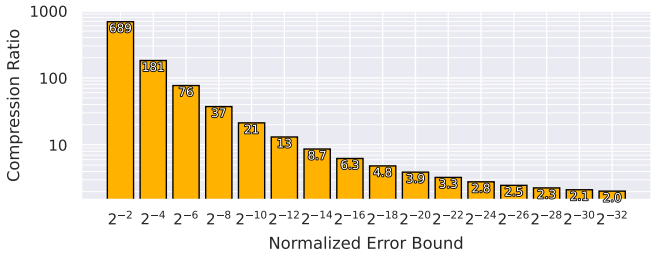

Fig. 9. Server-File Compression Ratio across EBs

Fig. 10 shows the average throughput when the server encodes a file at each EB in the tested progression. Evidently, the choice in EB does not have a dramatic impact on throughput. Nonetheless, the throughput tends to slightly decrease as the EB tightens since tighter EBs result in less compressible data and thus require writing more bytes. This trend is evident on both the RTX 4090 and the A100, with the A100 generally being slower. Prior work also shows LC (de)compressors to run more slowly on the A100 (40 GB) due to its relatively (to the RTX 4090) lower clock rate and lower SM count, which it exchanges for high memory bandwidth/capacity and more FP64 units [13]. Cortado does not benefit from these characteristics since the evaluated inputs fit on the tested GPUs and Cortado's quantizer does not perform floating-point operations. Thus, the A100 tends to perform worse.

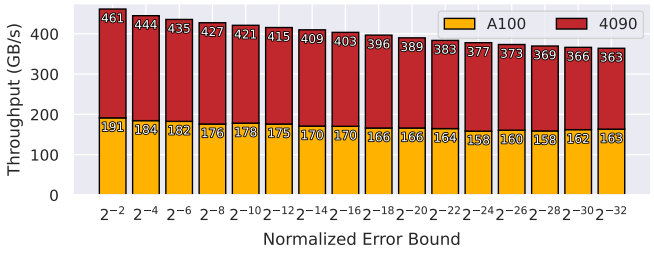

Fig. 10. Server-Encode Throughput across EBs

Note that, since the server-encode operation is performed offline, the runtime of the server encoder is less crucial than the online operations of Cortado (next subsections).

C. Initial Send/Receive

The initial-send operation (2) decompresses and dequantizes the next finer server file, then requantizes and compresses at the client's requested EB. Accordingly, the client runs

the initial-receive operation (3), which decompresses and dequantizes the received initial file.

Fig. 11 shows the CRs of all initial files at all measured client EBs. Note that the y-axis uses a logarithmic scale. Like the server files, the CR exponentially decreases as the EB tightens. However, since the underlying server quantizer halves a given EB, the CR of each initial file is relatively higher than its corresponding server file at the "same" EB.

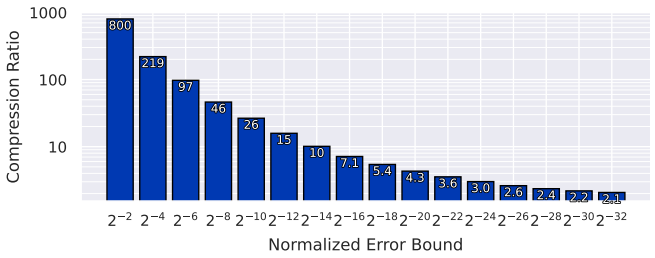

Fig. 11. Initial-File Compression Ratio across EBs

Fig. 12 shows the effect of the server EB and the request EB on the server's throughput when sending an initial file (2) on the RTX 4090. Here, the server EB (the EB at which the server decompresses and dequantizes) moves along the rows, and the client's request EB (the EB at which the server will requantize and compress) moves along the columns. Only the bottom left cells are populated since a request EB must be equal or coarser than a server EB.

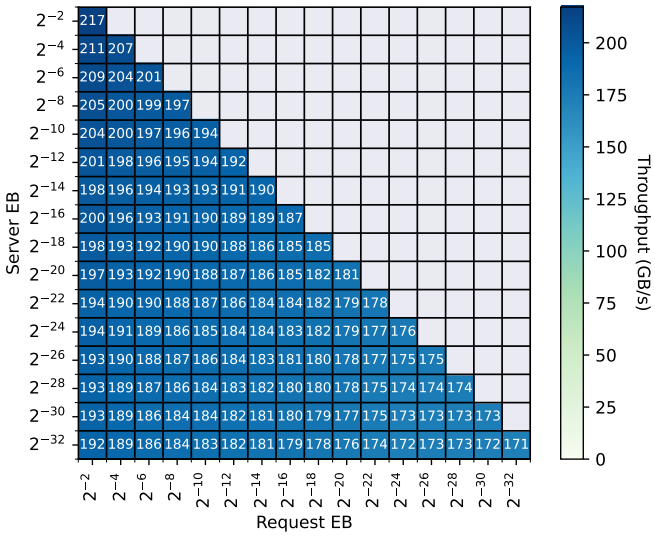

Fig. 12. Initial-Send Throughput across Server/Request EBs (RTX 4090)

The throughput slightly decreases along any column, demonstrating the throughput impact of the decompression phase of the initial send. In other words, compressed files at tighter EBs are a little slower to decompress because they are larger and require more work to process. Similarly, along any row, the throughput decreases marginally as the request EB tightens. So, like with decompression, files compressed at tighter EBs are somewhat slower to compress since more data is written, and, in the case of the RZE_1 stage of Cortado's compression algorithm, more data is iterated over during each recursive bitmap phase. Nonetheless, the decrease in throughput along both axes is minor, which shows that the impact of the client's request and the server EB on the throughput is not substantial. Cortado's compression throughput is robust to the compressibility of the data. Regardless of the context, the initial send is able to deliver relatively high throughputs. On the A100, the trends are the same, except the throughputs are lower by a factor of 2.2—ranging from 74 GB/s to 94 GB/s.

Fig. 13 shows the client's throughput when receiving an initial file (3). Like the server encode (Fig. 10) and initial send (Fig. 12), the request EB only slightly affects the initial-receive throughput. Interestingly, these initial-receive throughputs are a little lower than the server-encoding throughputs, despite the fact that decompression algorithms are typically faster than their corresponding compression algorithm. This is because Cortado's DIFFMS_4 stage in the compressor is embarrassingly parallel whereas the iDIFFMS_4 stage in the decompressor requires a parallel prefix sum, which takes longer to compute. Nonetheless, the initial-receive throughputs are always faster than any corresponding initial send.

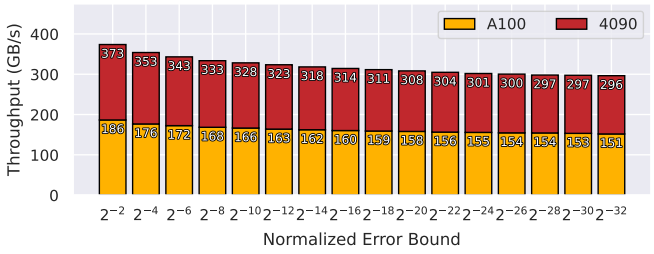

Fig. 13. Initial-Receive Throughput across EBs

Overall, the initial files compress well and are relatively fast to send and receive, with the server and request EB parameters having little impact on throughput. If we factor in the ability of the client to handle LC chunk(s) in parallel while the server prepares more chunks, the real end-to-end throughput capabilities are remarkable, as observed in Subsection VI-A. We use the initial send/receive metrics as a baseline in the next subsection, which evaluates the residual send/receive.

D. Residual Send/Receive

The residual-send operation (4) decompresses and dequantizes the next finer server file, requantizes to both the client's source (existing) EB (which is further requantized in terms of the destination EB) and the destination (requested) EB, then computes the residual and compresses it. Meanwhile, the residual-receive operation (5) requantizes the existing file to the destination EB. After (or while) receiving the residual file, it decompresses and applies the residual to the existing file.

Fig. 14 summarizes the CRs of residual files across source/destination EB combinations. Along the diagonal (highlighted yellow), we include the CR of the initial file as a

baseline. Each other cell is colored according to how much the CR improves over the baseline by sending the residual instead of the entire (initial) file.

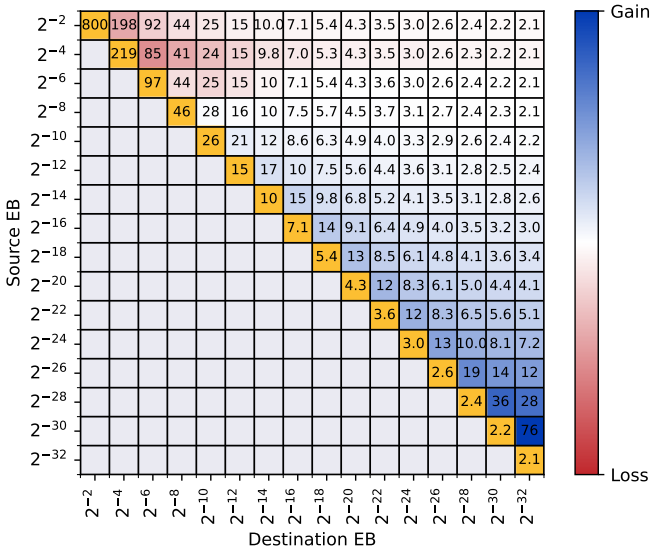

Fig. 14. Residual-File Compression Ratio across Source/Destination EBs

Overall, a residual file tends to have a better CR than an initial file that fulfills the same EB request. The residual files that have a relative CR loss are clustered towards the top-left of the graph—at the coarser source/destination EBs. This tends to not be an issue as these EBs, 2^{-2} through 2^{-6}, are more likely to be fulfilled by initial requests. Furthermore, they produce files with up to about 4 to 64 distinct values, respectively. For many applications, these EBs may be too coarse to be useful and, therefore, would be uncommon client requests. Even so, the relative loss of these residuals is small. For example, the highest relative loss refining from 2^{-4} to 2^{-6} is a mere 13%. In contrast, the gains tend to be much higher—e.g., 36x the baseline in the extreme case of refining from 2^{-30} to 2^{-32}.

The residual-send throughput depends on three parameters: the source EB, the destination EB, and the server EB from which the former two are requantized. For brevity's sake, we only present the residual-send throughputs from the tightest-tested server EB of 2^{-32} in Fig. 15. The trends are consistent at other server EBs, with throughputs trending higher at tighter server EBs—up to a maximum of 196 GB/s for a server EB of 2^{-4}. Like the preceding CR results in Fig. 14, we include the baseline along the diagonal highlighted in yellow. For this figure, the baseline corresponds to the 2^{-32} row from the initial-send results in Fig. 12.

Generally, since a residual send involves more work than an initial send, the throughput tends to be slightly lower—with some exceptions when computing the residual to/from tight EBs (bottom right). However, the throughput loss is small enough to be made up for by the CR improvement, which crucially alleviates the bottleneck of data communication.

Fig. 16 shows the receive throughput for each source/destination EB. Again, we observe that the residual

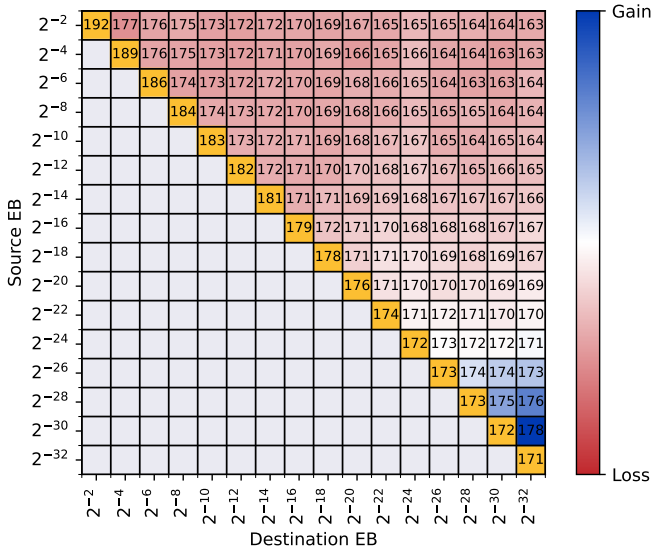

Fig. 15. Residual-Send Throughput across Source/Destination EBs for Server EB of 2^{-32} (RTX 4090)

files' relative gains in CR (compared to initial files) introduce a tradeoff in the residual-receive throughput. Like the residual send, the slight throughput loss is negligible if the communication speed is less than the send/receive throughput.

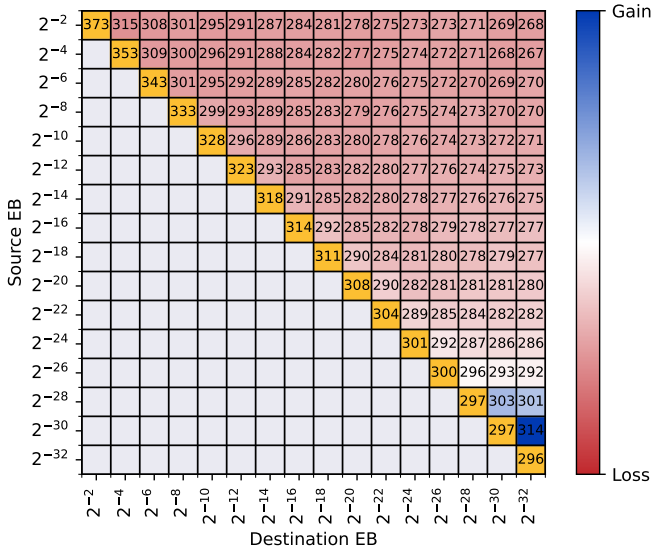

Fig. 16. Residual-Receive Throughput across Source/Destination EBs (RTX 4090)

Furthermore, as mentioned in Subsection III-E, part of the residual-receive operation (i.e., requantizing the client's existing file from the source to the destination EB) has no data dependency and can be performed simultaneously with the residual send. Therefore, in practice, the latency of this phase would be completely hidden and the end-to-end throughput would be much higher—making it even more competitive with the baseline of an initial send/receive. The server-client

model that enables this is unique to Cortado and the reason Cortado can fulfill near-arbitrary EB requests with minimal performance impact from the necessary on-the-fly requantization. As observed in Subsection VI-A, this latency hiding contributes to Cortado consistently achieving the highest end-to-end throughput compared to the state of the art.

VII. Summary and Conclusions

Scientific floating-point data is often compressed with error-bounded lossy compressors to boost the compression ratio while controlling information loss. To enable users to refine lossily-downloaded files, progressive compression methods encode the data in a hierarchical manner that minimizes redundant information. However, previous approaches suffer from problems such as violation of error-bound guarantees, error propagation/accumulation, support of only predetermined error-bound progressions, and multiple decompression passes per request. Cortado addresses these issues with its integer-based quantizer and on-the-fly requantization to near-arbitrary error bounds. With its server-client model and fast GPU (de)compression algorithm, Cortado consistently outperforms previous approaches in terms of encoding and retrieval throughput and achieves on-par compression ratios, all while being the only evaluated approach that strictly guarantees the requested error bound and produces reconstructions with comparable PSNR. Furthermore, Cortado supports progression down to lossless as a contingency for users. To the best of our knowledge, Cortado is the first progressive compressor to support this combination of features while delivering high throughputs and compression ratios.

VIII. Acknowledgments

This work has been supported by the U.S. National Science Foundation (NSF) under Award CCF-2403380, by the Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research (ASCR) under Award DE-SC0022223, and by an equipment donation from NVIDIA Corporation.

References

[1] N. Azami, A. Fallin, and M. Burtscher, "Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 395–409. [Online]. Available: https://doi.org/10.1145/3669940.3707280

[2] S. Di, J. Liu, K. Zhao, X. Liang, R. Underwood, Z. Zhang, M. Shah, Y. Huang, J. Huang, X. Yu, C. Ren, H. Guo, G. Wilkins, D. Tao, J. Tian, S. Jin, Z. Jian, D. Wang, M. H. Rahman, B. Zhang, S. Song, J. Calhoun, G. Li, K. Yoshii, K. Alharthi, and F. Cappello, "A Survey on Error-Bounded Lossy Compression for Scientific Datasets," *ACM Comput. Surv.*, vol. 57, no. 11, pp. 287:1–287:38, Jun. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3733104

[3] G. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, Feb. 1992. [Online]. Available: https://ieeexplore.ieee.org/document/125072/

[4] S. Klasky, J. Thayer, and H. Najm, "Data Reduction for Science: Brochure from the Advanced Scientific Computing Research Workshop," Tech. Rep. None, 1770192, Apr. 2021. [Online]. Available: https://www.osti.gov/servlets/purl/1770192/

[5] D. Hoang, P. Klacansky, H. Bhatia, P.-T. Bremer, P. Lindstrom, and V. Pascucci, "A Study of the Trade-off Between Reducing Precision and Reducing Resolution for Data Analysis and Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1193–1203, Jan. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8440822

[6] V. A. P. Magri and P. Lindstrom, "A General Framework for Progressive Data Compression and Retrieval," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–11, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10308629/

[7] X. Liang, Q. Gong, J. Chen, B. Whitney, L. Wan, Q. Liu, D. Pugmire, R. Archibald, N. Podhorszki, and S. Klasky, "Error-controlled, progressive, and adaptable retrieval of scientific data with multilevel decomposition," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3476179

[8] Y. Li, W. Li, Q. Gong, Q. Liu, N. Podhorszki, S. Klasky, X. Liang, and J. Chen, "HP-MDR: High-performance and Portable Data Refactoring and Progressive Retrieval with Advanced GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, Nov. 2025, pp. 2076–2093. [Online]. Available: https://dl.acm.org/doi/10.1145/3712285.3759845

[9] A. Fallin and M. Burtscher, "Lessons Learned on the Path to Guaranteeing the Error Bound in Lossy Quantizers," Jul. 2024. [Online]. Available: http://arxiv.org/abs/2407.15037

[10] M. Burtscher, N. Azami, A. Fallin, B. Burtchell, A. Rodriguez, B. Jerald, Y. Liu, and A. Mongandampulath Akathoott, "LC Git Repository," Apr. 2025. [Online]. Available: https://github.com/burtscher/LC-framework

[11] N. Azami, R. Lawson, and M. Burtscher, "LICO: An Effective, High-Speed, Lossless Compressor for Images," in *2024 Data Compression Conference (DCC)*, Mar. 2024, pp. 462–471, iSSN: 2375-0359. [Online]. Available: https://ieeexplore.ieee.org/document/10533820

[12] S. Wu, J. Pan, J. Liu, J. Tian, Z. Qiu, J. Huang, K. Zhao, X. Liang, S. Di, Z. Chen, and F. Cappello, "Boosting Scientific Error-Bounded Lossy Compression through Optimized Synergistic Lossy-Lossless Orchestration," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, Nov. 2025, pp. 2024–2037. [Online]. Available: https://dl.acm.org/doi/10.1145/3712285.3759798

[13] A. Fallin, N. Azami, S. Di, F. Cappello, and M. Burtscher, "Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds," in *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium*, June 2025. [Online]. Available: https://doi.org/10.1109/IPDPS64566.2025.00083

[14] A. M. Akathoott, A. Rodriguez, and M. Burtscher, "SLEEK: Compressing Memory Copies for Floating-Point Data on GPUs," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2026, pp. 115–129, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/document/11575376

[15] B. Lu, Z. Liu, H. Zhao, D. Luo, W. Huang, Y. Gu, J. Liu, G. Tan, and D. Tao, "PRISM: An Efficient GPU-Based Lossy Compression Framework for Progressive Data Retrieval with Multi-Level Interpolation," in *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '26. New York, NY, USA: Association for Computing Machinery, Jan. 2026, pp. 164–176. [Online]. Available: https://dl.acm.org/doi/10.1145/3774934.3786438

[16] D. Merrill and M. Garland, "Single-pass Parallel Prefix Scan with Decoupled Look-back," NVIDIA, Tech. Rep. NVR-2016-002, Mar. 2016. [Online]. Available: https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back

[17] 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 (IPDPS)*, May 2017, pp. 1129–1139, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/document/7967203

[18] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An Efficient Transformation Scheme for Lossy Data Compression with Point-Wise Relative Error Bound," in *2018 IEEE International Conference on*

Cluster Computing (CLUSTER), Sep. 2018, pp. 179–189, iSSN: 2168-9253. [Online]. Available: https://ieeexplore.ieee.org/document/8514879

[19] S. Li, P. Lindstrom, and J. Clyne, "Lossy Scientific Data Compression With SPERR," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2023, pp. 1007–1017, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/document/10177487

[20] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec. 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6876024

[21] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P.-T. Bremer, and V. Pascucci, "Efficient and Flexible Hierarchical Data Layouts for a Unified Encoding of Scalar Field Precision and Resolution," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 603–613, Feb. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9222049

[22] B. A. Burtchell and M. Burtscher, "Building n-dimensional trees for resolution-based progressive compression," in *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC Workshops '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 307–313. [Online]. Available: https://doi.org/10.1145/3731599.3767373

[23] P. Alliez and M. Desbrun, "Progressive compression for lossless transmission of triangle meshes," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, Aug. 2001, pp. 195–202. [Online]. Available: https://dl.acm.org/doi/10.1145/383259.383281

[24] D. Cohen-Or, D. Levin, and O. Remez, "Progressive Compression of Arbitrary Triangular Meshes," in *Proceedings Visualization '99 (Cat. No.99CB37067)*, Oct. 1999, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/10189410/

[25] 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, Dec. 2018. [Online]. Available: https://doi.org/10.1007/s00791-018-00303-9

[26] ——, "Multilevel techniques for compression and reduction of scientific data—the multivariate case," *SIAM Journal on Scientific Computing*, vol. 41, no. 2, pp. A1278–A1303, 2019. [Online]. Available: https://doi.org/10.1137/18M1166651

[27] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappello, "SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors," *IEEE Transactions on Big Data*, vol. 9, no. 2, pp. 485–498, Apr. 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9866018

[28] X. Liang, B. Whitney, J. Chen, L. Wan, Q. Liu, D. Tao, J. Kress, D. Pugmire, M. Wolf, N. Podhorszki, and S. Klasky, "MGARD+: Optimizing Multilevel Methods for Error-Bounded Scientific Data Reduction," *IEEE Transactions on Computers*, vol. 71, no. 7, pp. 1522–1536, Jul. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9479913/

[29] J. Wang, X. Liang, B. Whitney, J. Chen, Q. Gong, X. He, L. Wan, S. Klasky, N. Podhorszki, and Q. Liu, "Improving Progressive Retrieval for HPC Scientific Data using Deep Neural Network," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, Apr. 2023, pp. 2727–2739. [Online]. Available: https://ieeexplore.ieee.org/document/10184595/

[30] J. Chen, L. Wan, X. Liang, B. Whitney, Q. Liu, D. Pugmire, N. Thompson, J. Y. Choi, M. Wolf, T. Munson, I. Foster, and S. Klasky, "Accelerating Multigrid-based Hierarchical Scientific Data Refactoring on GPUs," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Portland, OR, USA: IEEE, May 2021, pp. 859–868. [Online]. Available: https://ieeexplore.ieee.org/document/9460526/

[31] Z. Yang, S. Di, L. Zhang, R. Li, X. Li, J. Huang, J. Liu, F. Cappello, and K. Zhao, "IPComp: Interpolation Based Progressive Lossy Compression for Scientific Applications," in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '25. New York, NY, USA: Association for Computing Machinery, Sep. 2025, pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3731545.3731578

[32] 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)*, Dec. 2020, pp. 2716–2724. [Online]. Available: https://ieeexplore.ieee.org/document/9378449

[33] "SDRBench," 2020. [Online]. Available: https://sdrbench.github.io/

[34] Y. Li, "MasterVChicken/Multi-Component-Compression-Framework: sc25-hpmdr-v1," Aug. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16938522