Building n-Dimensional Trees for Resolution-Based Progressive Compression

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

Floating-point data is typically compressed at strict error bounds to reduce storage cost while facilitating scientific analyses. Unfortunately, this tends to yield large compressed files. In some cases, however, a user might not need the data at a high fidelity. Progressive compression addresses this issue by refactoring the data into a hierarchical series of increasing fidelity, allowing users to download the data at an initial fidelity and subsequently retrieve higher fidelities. This paper studies a resolution-based progressive compression approach that achieves competitive compression ratios against traditional compression methods. Furthermore, it studies how the progression of resolution affects the quality of the data.

CCS Concepts

• **Theory of computation → Data compression**.

Keywords

Data compression, lossy compression, progressive compression

ACM Reference Format:
Brandon Alexander Burtchell and Martin Burtscher. 2025. Building n-Dimensional Trees for Resolution-Based Progressive Compression. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3731599.3767373

1 Introduction

Scientific data is often compressed to reduce storage space. Still, the compressed data can be quite large, leading to long download times. In some cases, a user might not initially need a high-fidelity version of the data but still requires access to a lossless or error-bounded lossy version as a contingency. Progressive compression addresses this situation. It is a compression technique that refactors an input into a hierarchical series of progressively more lossy representations. Thus, a user could initially download a file at a low fidelity and then progressively download more of the fidelity in the future if needed. Importantly, an effective progressive compression method does not simply maintain multiple copies of the data at different fidelities. Rather, it encodes the extra information stored in each representation such that the user does not have to download redundant information in subsequent progressive retrievals. Typically, progressive compression varies an input by resolution or precision. We study a method that varies resolution and compresses each resolution level with a potentially different compression algorithm.

To find suitable compression algorithms for each level, we use the LC framework [9]. 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. LC has been used to generate several state-of-the-art lossless and lossy compressors including SPratio, SPspeed, DPratio, DPspeed, and PFPL [8, 11].

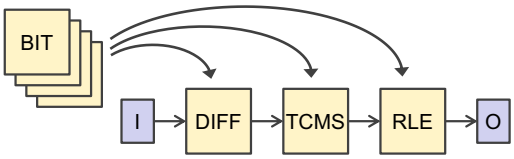


Figure 1: A compression pipeline with 3 stages generated by the LC framework (I = input, O = output)

Only some of the components, called *reducers*, compress the data. The others rearrange or modify the data to expose redundancies that can be leveraged by subsequent reducer components in the pipeline. For this study, we use 69 total components—including 36 reducers. With 3 stages, LC can generate $69 \times 69 \times 36 = 171{,}396$ pipelines.

This paper makes the following main contributions:

- It outlines a tree-based resolution-varying progressive compression approach.
- It demonstrates that the proposed approach achieves competitive compression ratios with traditional lossy compression methods that represent the same data.
- It analyzes the effect of varying resolution on the data quality.

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 demonstrates the proposed approach. Section 4 outlines the experimental methodology. Section 5 presents and analyzes the results. Section 6 summarizes and concludes.

2 Related Work

Progressive JPEG was introduced with the JPEG standard [16]. It utilizes regular JPEG's discrete wavelet transform but scans the image multiple times to encode successively improving quality levels

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767373

of the image data. The subsequent levels only encode the lower significant bits of the values to avoid storing redundant information.

MGARD [4, 5] decomposes scalar fields into a hierarchy of components of varying scale and resolution and adaptively quantizes each component's coefficients to provide error-bounded compression. PMGARD [14, 17] introduces progressive retrieval in terms of both precision and resolution simultaneously. This is achieved by optimizing MGARD's data refactoring technique and performing bit-plane encoding.

Hoang et al. [13] build a precision-resolution tree by restructuring a scalar-field input into a tree (varying resolution), then splitting each node into a sequence of bit planes (varying precision). While the idea of building trees is shared, we opt to sum each group of values instead of rearranging the original values. This allows users to recover averages of the data during partial decoding instead of equidistant samples of the data. Furthermore, we utilize a specific quantization method that allows for the summation of bin numbers to enable our approach on floating-point data.

Yang et al. [19] introduce IPComp, which employs interpolation and bitplanes to offer progressive reconstruction. Our averaging approach to decoding is similar to IPComp's linear interpolation, but we differ in the quantization and compression techniques.

Magri and Lindstrom [15] introduce a general framework for progressive compression that guarantees successively tighter error bounds at every progressive 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, error-bounded compressor can be used to encode/decode at each progressive level.

Early progressive compression works target 3D triangle meshes [6, 10]. Generally, these works use mesh decimation and vertex prediction techniques to vary the number of vertices while maintaining a good approximation of the original mesh. Our approach differs in target application and, therefore, in the progression techniques.

3 Approach

This section discusses the progressive approach, which consists of quantization, tree encoding, and layer compression. Section 3.1 discusses the necessary quantization step for floating-point support. Sections 3.2 and 3.3 present the proposed tree encoding/decoding approach via a simple example on 1D data and a discussion on generalizations for n-dimensional support, respectively. Lastly, Section 3.4 outlines the way each progressive layer is compressed.

3.1 Quantization

We first quantize the data from float values to integer bins using the point-wise NOA (normalized absolute) quantizer from PFPL [11], which accepts a user-defined error bound and normalizes it to the range of values (minimum to maximum) in the input. The quantizer encodes values either lossily as an integer that represents an approximation of the original value or losslessly if it cannot otherwise meet the selected error bound [12].

3.2 1D Demonstration

Fig. 2 demonstrates a simple example of the tree encoding scheme used to build progressive layers. The diagram is arranged in the order of the computation from top to bottom. Suppose we quantized a floating-point input that had 8 elements and yielded the values in layer 0. To form progressive layers, we can sum each pair of values to create a parent. For example, layer 0 sums the first two values—1 and 2—to create their parent with a value of 3. In this example, the encoder repeats this process to build 3 layers up to the root. Importantly, the root (36) is the sum of all values in the original quantized input. To avoid overflow, we cast the original quantized 32-bit integers to the 64-bit `long long` type before performing addition. In the event of an incomplete pair (i.e., an odd-sized layer), the extra value would simply be passed up to the next layer as if it had a neighboring zero.

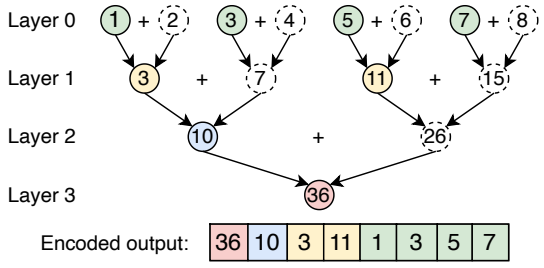


Figure 2: Building progressive layers from 1D data

The tree representation holds some redundant information that can be omitted from the final encoded output. Namely, we can drop one child from each summed group. In this 1D example, we drop the right child of every pair (dashed circles), and emit values to the encoded output in order of root (layer 3) to base (layer 0). The encoded output is stored in order from layer 3 to 0 in `long long` format. Thus, for this example, the output has the same number of elements as the input but the size in bytes is doubled.

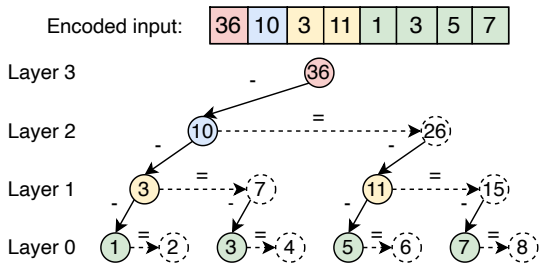


Figure 3: Full decoding

Fig. 3 demonstrates the decoding technique if the user chose to download the entire encoded data produced by the example in Fig. 2. Again, the diagram is arranged in computation order: starting with the encoded input at the top and ending with the reconstructed output in layer 0. To begin, we retrieve the top layer from the encoded input (layer 3 with root 36). To decode each layer, we retrieve the left children from the next layer (layer 2 with left child 10), then subtract the left child from its parent to recover

the right child. To fully decode to the original quantized input, we repeat this down to layer 0.

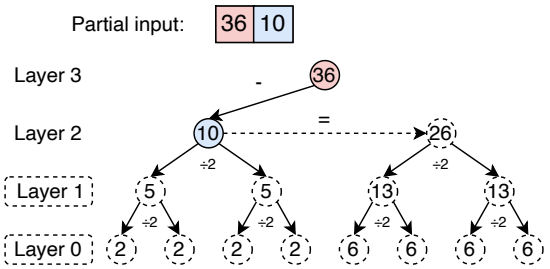


Figure 4: Partial progressive decoding from layer 2

Now suppose the user instead only downloaded layers 3 and 2. Fig. 4 demonstrates how to approximate from the partially decoded data in layer 2. We simply divide each value in the previous layer by the number of children (i.e., 10 divided by 2) to approximate the values of its children. By repeating this down to layer 0, we end up with an approximation of the same size as the original but a quarter of the "resolution". Both the full and partial decoded versions can then be de-quantized to yield a floating-point reconstruction.

3.3 *n*-Dimensional Generalizations

While the preceding subsection walked through a simple 1D example, our approach also supports higher dimensions. To generalize, the 1D pairs become *n*-dimensional *groups* of 2^n values. When building each layer, we can sum up the values in the group to create the parent. When encoding, we write every value of each group except the last (i.e., the bottom-right in 2D, the bottom-right-back in 3D, etc.). Thus, the total encoded output will have the same number of elements as the original input (plus extra elements in the event of an odd layer size in one or more dimensions, e.g., a 2-by-3 layer). Fig. 5 demonstrates the encoding approach on a 4-by-4 2D example consisting of four 2-by-2 (color-coded) groups. The dropped values from each group are marked with a stroked box.

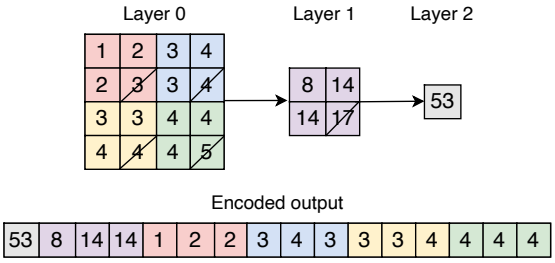


Figure 5: Building progressive layers from 2D data

Like in 1D, the layers are emitted in order of the top layer (layer 2 in the example) to layer 0. Importantly, the values of each group are emitted to the encoded output together rather than being encoded in the order of the original file (e.g., row-by-row in 2D). For instance, in Fig. 5, layer 0's encoded output starts by writing the top-left group (red) instead of writing the entire first row, which spans two groups (i.e., elements 1, 2, 3, 4). Reordering the values in such a way can group more similar values together, which typically leads to higher compression ratios.

3.4 Layer Compression

After the tree is built and the redundant values from each layer have been dropped, we can compress the data to save storage space. Since the layers may exhibit different data patterns and characteristics, it is prudent to explore the effect of compressing each layer's encoded representation with an independent compression algorithm to maximize the compression ratio. We use LC to generate and search for lossless compression pipelines for this purpose. Section 5.2 discusses which compression pipelines perform well for each layer. In the decoding stage, each retrieved layer must thus be decompressed by LC before decoding the tree.

Note that LC handles data in 16 kB chunks. In order to avoid diminishing returns in compression ratio and to save encoding/decoding time, we stop building the progressive layers when the next higher layer's emitted size would be smaller than 16 kB.

4 Experimental Methodology

We aim to evaluate the proposed approach in terms of compression ratio and quality of the data reconstruction. These metrics are system-independent, so we do not list system specifications. For our evaluation, we use a total of 58 files from the following SDRBench [3, 20] datasets: CESM-ATM (3D), EXAALT Copper, ISABEL [1], and NYX [2, 7]. The files of each dataset are described in Table 1. Note that we split the EXAALT Copper dataset to separate the fields with different dimensions.

Table 1: 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

For the quantizer, we tested the following normalized absolute error bounds [11]: 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}. This paper shows the results for 10^{-6}. The other error bounds yield similar trends but higher compression ratios and lower initial quality since they accept more initial loss.

To search for LC pipelines, we start with a cursory search for 2-stage pipelines for each layer at each of the tested error bounds. Then, we pin the 1st stage of each layer's best 2-stage pipelines to the 1st stage of our 3-stage pipelines. This technique saves search time while yielding well-compressing algorithms. Performing an exhaustive 3-stage pipeline search per layer would be orders of magnitude slower. For this particular study, we pinned the following components to stage 1, yielding $10 \times 69 \times 36 = 24{,}840$ pipelines: NUL, BIT_8, DIFFMS_8, DIFFNB_8, HCLOG_8, RARE_4, RAZE_4, RRE_8, RZE_2, and RZE_4. Note that we include the NUL component,

Table 2: Running compression ratio (NOA error bound = 10^{-6})

Layer	Pipeline	CESM	EXAALT (1)	EXAALT (2)	ISABEL	NYX
8	DIFFNB_8 RAZE_8 RARE_8	-	-	21673.78	-	-
7	DIFFMS_8 BIT_8 RZE_2	-	-	7387.46	-	-
6	RZE_4 TUPL6_4 RARE_4	-	4007.67	2511.69	-	-
5	DIFFMS_8 BIT_8 RZE_1	30576.39	1081.40	849.86	-	38789.05
4	DIFFMS_8 RZE_4 RAZE_4	3966.49	306.16	254.73	6321.55	5277.55
3	DIFFMS_8 RZE_4 RAZE_4	599.06	87.09	76.39	1013.22	755.25
2	DIFFMS_8 BIT_8 RZE_1	109.39	23.74	22.76	175.72	120.02
1	DIFFMS_8 BIT_8 RZE_1	20.26	6.62	6.53	31.73	19.82
0	DIFFMS_8 BIT_8 RZE_1	3.56	1.86	1.88	6.03	3.64
Same Pipeline All Progr. Layers	DIFFMS_8 BIT_8 RZE_1	3.56	1.85	1.88	6.03	3.64
Same Pipeline All Inputs, Lossy	DIFFNB_8 BIT_4 RZE_1	5.40	1.88	1.94	6.00	3.68
Pipeline Per Input, Lossy	Exhaustive	5.42	2.20	1.96	7.22	3.75
Pipeline Per Input, Lossless	Exhaustive	2.02	1.85	1.46	2.39	1.42

which simply passes its input on to the next component, in order to consider 1- and 2-stage pipelines in the search.

5 Results

This section presents the results. Subsections 5.1 and 5.2 analyze the compression ratios. Subsection 5.3 investigates the effect of resolution progression on data quality.

5.1 Running Compression Ratio

To understand the compression ratio in terms of how much accumulated encoded data the user must download to retrieve each resolution layer, we can use the *running compression ratio*, which is defined as the original file size divided by the accumulated sizes of the target layer and all layers above it (higher is better). Table 2 presents the geometric mean of the running compression ratios for each dataset. It also displays several baselines. First is the running compression ratio down to layer 0 if each progressive layer was compressed by the single-best pipeline rather than their own independent pipelines. The final three baselines are traditional, non-progressive approaches: the lossy compression ratio from the single-best pipeline across all inputs, the lossy compression ratio from each file's best pipeline, and the lossless compression ratio from each file's best pipeline. For all baselines but the first, we utilize LC's full three-stage search space of 171,396 pipelines. Since each dataset has different dimensions, the number of generated progressive layers differs per column. If a layer is not reached we leave the cell blank.

Starting at each dataset's topmost layer, we see extremely high compression ratios. As we progress downwards—meaning the user retrieves extra data to improve the resolution—the running compression ratio expectedly goes down. Importantly, all layers except layer 0 always have a running compression ratio greater than all the traditional-compression baselines. With the exception of the CESM dataset, decoding down to layer 0—that is, recovering the equivalent data of a traditional lossy method—always achieves a running compression ratio that is within 5% of the single-pipeline lossy baseline. Furthermore, progressive retrieval down to layer 0 always beats lossless compression.

Importantly, we observe no meaningful difference in the running compression ratio down to layer 0 between the approach that uses different pipelines per layer and the approach that uses the same pipeline per layer. While not shown, the running compression ratios of layers 1 and above with this single pipeline had a negligible loss. Notably, the best pipeline to use across all pipelines is the same as the best pipeline to use only on layer 0 (and also layers 1 and 2): DIFFMS_8 BIT_8 RZE_1. This is likely because layer 0 is the largest layer and influences the running compression ratio the most. We discuss the constituent components of this pipeline in Section 5.2.

We can conclude the following from these observations. First, for all but one of the evaluated datasets, the proposed approach achieves compression ratios that are on par with the traditional approach of compressing all files with the same pipeline, while offering users the added capability of progressive retrieval. Second, all evaluated datasets exhibit significantly higher compression ratios if decoded down to layer 1 or higher, meaning the partial retrieval of a file does not incur a penalty on compression ratio. Lastly, using a different pipeline per layer is not necessary for these inputs, as using a single well-compressing pipeline for all layers yields almost the same compression ratios.

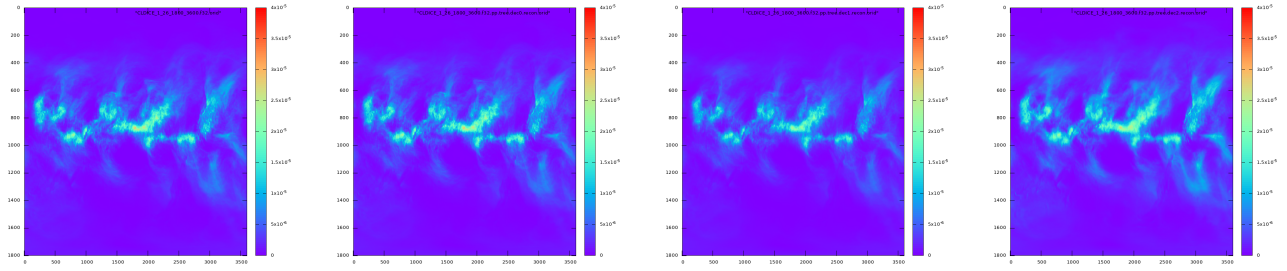
5.2 Per-Layer Compression Ratio

To understand how each layer compresses, Table 3 shows the *independent* compression ratios of each layer. The per-layer compression ratio is the original size of the layer (i.e., elements of the encoded output with the same color in Fig. 2) divided by its compressed size. Note that higher layers have a significantly smaller size than lower layers before LC compression.

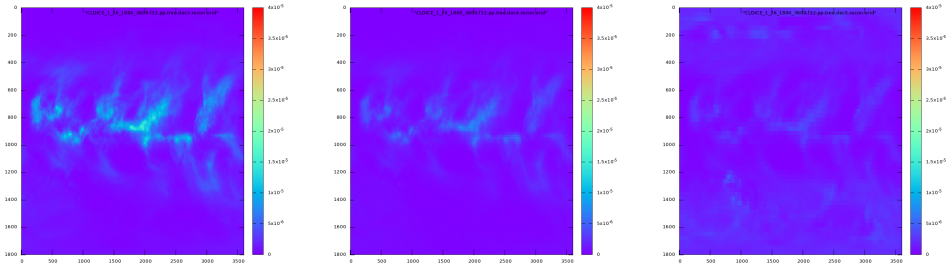
There is a dramatic increase in compression ratios as we go down from the top layer to the lower layers with more values. This is the opposite of what was observed for the running compression ratios, which is likely due to the lower layers having more leading zeros in each value since fewer bin-number additions have occurred—making these values easier to compress. Furthermore, the higher per-layer compression ratios for lower layers is exacerbated by the dimensionality of the inputs. For instance, CESM, ISABEL, and NYX are 3D datasets, and the compression-ratio increase from each dataset's topmost layer to the 0^{th} layer is much more dramatic than

Table 3: Per-layer compression ratio (NOA error bound = 10^{-6})

Layer	Pipeline	CESM	EXAALT (1)	EXAALT (2)	ISABEL	NYX
8	DIFFNB_8 RAZE_8 RARE_8	-	-	2.04	-	-
7	DIFFMS_8 BIT_8 RZE_2	-	-	2.11	-	-
6	RZE_4 TUPL6_4 RARE_4	-	2.00	2.15	-	-
5	DIFFMS_8 BIT_8 RZE_1	2.34	2.21	2.42	-	2.37
4	DIFFMS_8 RZE_4 RAZE_4	2.41	2.52	2.47	3.63	2.61
3	DIFFMS_8 RZE_4 RAZE_4	2.97	2.86	2.79	4.43	3.01
2	DIFFMS_8 BIT_8 RZE_1	4.02	3.06	3.13	5.87	3.90
1	DIFFMS_8 BIT_8 RZE_1	5.51	3.45	3.47	8.50	5.20
0	DIFFMS_8 BIT_8 RZE_1	7.56	3.87	3.97	13.08	7.85



(a) Original data (b) Layer 0 (c) Layer 1 (d) Layer 2

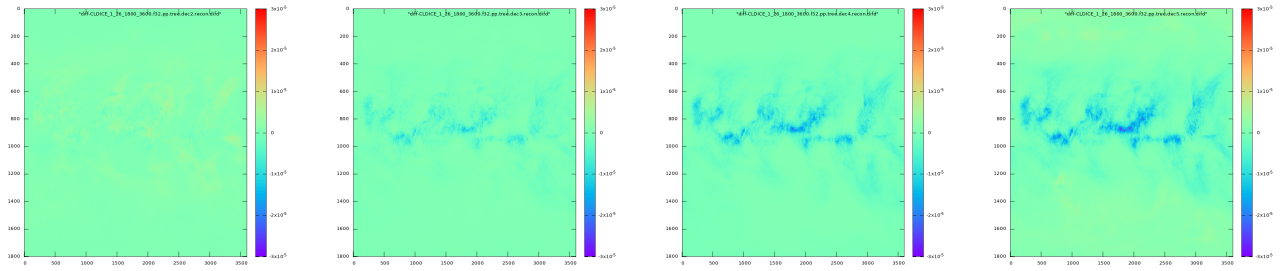


(e) Layer 3 (f) Layer 4 (g) Layer 5

Figure 6: CLDICE progressive layers (NOA error bound = 10^{-6})

in the 2D EXAALT datasets. This suggests that the LC pipelines are exploiting the grouping of nearby values performed by the tree encoding phase, and that there is more correlation between values in the lower layers. Specifically, the 3D inputs group and emit 7 nearby values at a time while the 2D inputs group and emit 3 values at a time. While this grouping technique seems to benefit compression ratios, it necessarily means that each layer drops more values when summing a group to create a parent. Luckily, this does not seem to incur a major difference in data quality as will be investigated in Section 5.3.

In LC, each component is available with a set of word sizes that it can operate at—typically 1, 2, 4, and 8 bytes. Regarding the pipelines for each layer, we observe that layers 0 through 2 pick the same pipeline: DIFFMS_8 BIT_4 RZE_1. Recall that this is also the pipeline that, when used on all layers, yields the best single-pipeline progressive running compression (see Fig. 2). Since these layers contain the most data to compress, this pipeline is arguably the most crucial to the overall compression of the files. DIFFMS_8 outputs the difference sequence of 8-byte (i.e., 64-bit `long long`) values in magnitude-sign format. Then, BIT_8 rearranges the data to output the first bit of every 8-byte value, then the second bit of all values, and so on. If the difference sequence contains values with many leading zeros, then BIT_8 moves those zeros to the beginning of the file. The last component, RZE_1 performs repeated zero elimination on 1-byte values. Thus, for layers 0 through 2, it seems that the values are quite smooth, yielding a difference sequence with many leading zeros, which are exploited by the second and third stages.



(a) Layer 2 (b) Layer 3 (c) Layer 4 (d) Layer 5

Figure 7: Differences of reconstructed layers from original CLDICE file (NOA error bound = 10^{-6})

The other layers have different pipelines yet include similar components. However, as discussed in Section 5.1, the same-pipeline-all-layers compression ratios are so close to the unique-pipeline-per-layer approach that the unique pipelines may not be necessary.

5.3 Quality

While the running compression ratios comparatively surpass or are on-par with the baseline traditional compression methods, it is necessary to analyze the effect of the progression on the quality and usability of the data.

Fig. 6 visualizes the 13^{th} 2D slice of the CLDICE input from the 3D CESM-ATM dataset at each reconstructed progressive layer. We choose this slice as it is in the middle of the first dimension of this $26 \times 1{,}800 \times 3{,}600$ input. The value range for this file is approximately $[-2.6 \times 10^{-22}, 7.8 \times 10^{-5}]$, and the average is approximately 1.6×10^{-6}. For visibility's sake, we plot each image from 0 to 4×10^{-5}. Naturally, the resolution degrades as we progress up to the lower-resolution layers. We also observe that each value progressively approaches the average of all values. However, in all layers, we can see characteristics of the turbulent region, which presumably are of the most interest.

Fig. 7 shows the difference between the reconstructed progressive layers 2, 3, 4, and 5 and the original data, visualized on the same 13^{th} slice of CLDICE. We omit layers 0 and 1 because the differences are so small that they are imperceptible in our charts. We display the differences with a symmetrical range of $[-3.0 \times 10^{-5}, 3.0 \times 10^{-5}]$. Evidently, the differences trend in the negative direction, meaning the differing values in the reconstructions tend to be less than the original value. This makes sense, since the turbulent region consists of higher values than the average value (1.6×10^{-6}). As the progressive reconstructions approach the average value, the reconstructed values necessarily tend to be less than the original values. For the other inputs, we also observe this tendency to approach the average value. Still, the magnitude of the differences are relatively small relative to the range of the input.

These visualizations demonstrate the utility of the proposed approach to preview data. That is, a user could download a low-resolution version and have enough information to decide whether to download a higher resolution for computation or which part of the dataset is likely of interest and should be downloaded at a higher resolution.

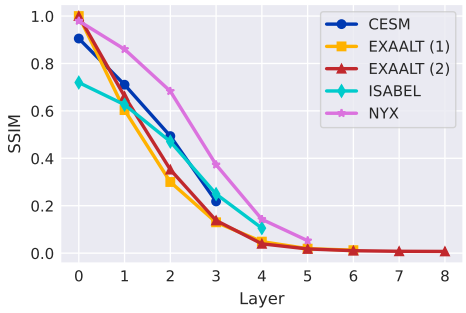


Figure 8: SSIM across layers (NOA error bound = 10^{-6})

Fig. 8 presents the structural similarity index measure (SSIM) [18] for each file as it progresses through its layers. SSIM rates a reconstruction's similarity to its original version from 0 to 1, with 1 representing an exact reconstruction. Note that, since layer 0's quality is established by the quantizer's base error bound, there is already some loss of SSIM at layer 0.

These results essentially quantify the qualitative findings from Figs. 6 and 7. Naturally, the SSIM decreases as we progress through the layers. However, there is no significant difference in the SSIM's rate of change between these datasets of different dimensions and sizes. We observe the same trends for all other tested error bounds, albeit starting at lower SSIMs for looser error bounds. Users could employ such a metric to determine what range of layers is meaningful. Considering that layers 1 and above all have higher running compression ratios than the traditional approaches (see Section 5.1), any layer that is usable would deliver a shorter download time.

6 Summary and Conclusions

This work investigates a resolution-based progressive compression method that combines an n-dimensional tree-encoding technique with per-layer customized compression to achieve similar compression ratios as traditional non-progressive compression at the

same error bounds. Our visualizations show that even the lowest-resolution layers maintain characteristics of the data that could be useful to users. Moreover, we observe reasonable SSIM degradation across all evaluated datasets as we progress through the layers.

The approach's n-dimensional support bolsters its possibility for usage in real applications. Additionally, the partial decoding method makes it possible to implement support for user interruption—where a user could stop a progressive download at any moment and still retrieve a usable version of the data. However, there are some tradeoffs. Mainly, the quantization method cannot guarantee an error bound for layers beyond the base layer. The summing of quantized values has the possibility of mixing losslessly-encoded floating-point values and lossy integer bins—which can introduce unpredictable errors. To address these tradeoffs, it is worth exploring techniques that progress in terms of guaranteed error bounds.

Acknowledgments

This work has been supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Research (ASCR), under contract DE-SC0022223. We also acknowledge the DOE NNSA ECP project and the ECP CODAR project with regards to SDRBench.

References

[1] 2004. IEEE Visualization 2004 Contest: Data Set. Retrieved August 24, 2025 from http://vis.computer.org/vis2004contest/data.html

[2] 2017. Nyx Project. Retrieved August 24, 2025 from https://amrex-astro.github.io/Nyx/

[3] 2020. SDRBench. Retrieved August 22, 2025 from https://sdrbench.github.io/

[4] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2018. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science* 19, 5 (Dec. 2018), 65–76. doi:10.1007/s00791-018-00303-9

[5] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2019. Multilevel Techniques for Compression and Reduction of Scientific Data—The Multivariate Case. *SIAM Journal on Scientific Computing* 41, 2 (2019), A1278–A1303. doi:10.1137/18M1166651

[6] Pierre Alliez and Mathieu Desbrun. 2001. Progressive compression for lossless transmission of triangle meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 195–202. doi:10.1145/383259.383281

[7] Ann S. Almgren, John B. Bell, Mike J. Lijewski, Zarija Lukić, and Ethan Van Andel. 2013. Nyx: A MASSIVELY PARALLEL AMR CODE FOR COMPUTATIONAL COSMOLOGY. *The Astrophysical Journal* 765, 1 (Feb. 2013), 39. doi:10.1088/0004-637X/765/1/39 Publisher: The American Astronomical Society.

[8] Noushin Azami, Alex Fallin, and Martin Burtscher. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 395–409. doi:10.1145/3669940.3707280

[9] Martin Burtscher, Noushin Azami, Alex Fallin, Brandon Burtchell, Andrew Rodriguez, Benila Jerald, Yiqian Liu, and Anju Mongandampulath Akathoott. 2025. LC Git Repository. https://github.com/burtscher/LC-framework

[10] Daniel Cohen-Or, David Levin, and Offir Remez. 1999. Progressive Compression of Arbitrary Triangular Meshes. In *Proceedings Visualization '99 (Cat. No.99CB37067)*. 1–7. doi:10.1109/VISUAL.1999.10189410

[11] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. 2025. 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*. doi:10.1109/IPDPS64566.2025.00083

[12] Alex Fallin and Martin Burtscher. 2024. Lessons Learned on the Path to Guaranteeing the Error Bound in Lossy Quantizers. doi:10.48550/arXiv.2407.15037

[13] Duong Hoang, Brian Summa, Harsh Bhatia, Peter Lindstrom, Pavol Klacansky, Will Usher, Peer-Timo Bremer, and Valerio Pascucci. 2021. Efficient and Flexible Hierarchical Data Layouts for a Unified Encoding of Scalar Field Precision and Resolution. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (Feb. 2021), 603–613. doi:10.1109/TVCG.2020.3030381

[14] Xin Liang, Qian Gong, Jieyang Chen, Ben Whitney, Lipeng Wan, Qing Liu, David Pugmire, Rick Archibald, Norbert Podhorszki, and Scott Klasky. 2021. Error-controlled, progressive, and adaptable retrieval of scientific data with multilevel decomposition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3458817.3476179

[15] Victor A. P. Magri and Peter Lindstrom. 2023. A General Framework for Progressive Data Compression and Retrieval. *IEEE Transactions on Visualization and Computer Graphics* (2023), 1–11. doi:10.1109/TVCG.2023.3327186

[16] G.K. Wallace. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics* 38, 1 (Feb. 1992), xviii–xxxiv. doi:10.1109/30.125072

[17] Jinzhen Wang, Xin Liang, Ben Whitney, Jieyang Chen, Qian Gong, Xubin He, Lipeng Wan, Scott Klasky, Norbert Podhorszki, and Qing Liu. 2023. Improving Progressive Retrieval for HPC Scientific Data using Deep Neural Network. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2727–2739. doi:10.1109/ICDE55515.2023.00209

[18] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (April 2004), 600–612. doi:10.1109/TIP.2003.819861

[19] Zhuoxun Yang, Sheng Di, Longtao Zhang, Ruoyu Li, Ximiao Li, Jiajun Huang, Jinyang Liu, Franck Cappello, and Kai Zhao. 2025. IPComp: Interpolation Based Progressive Lossy Compression for Scientific Applications. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3731545.3731578

[20] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors. In *2020 IEEE International Conference on Big Data (Big Data)*. 2716–2724. doi:10.1109/BigData50022.2020.9378449