Compressed Climate Data on TileDB

Yeskendir Zharkynbek
CS 245
KAUST

***Abstract*—I successfully used the SZ3 compression framework in Python to compress a large 3D climate dataset and store it in TileDB. I used a pre-built pipeline that operates through three main stages: prediction, quantization, and encoding. This process minimized the data size while guaranteeing that the results satisfy the required relative error boundaries. Using the Interp.lorenzo algorithm, I achieved a high compression ratio of 40 at a relative error bound of $\epsilon = 10^{-2}$**

I. Introduction

Scientific work often creates huge, multi-dimensional datasets that are expensive to store and use. Climate data, like the Red Sea temperature information I used, is a perfect example: it involves floating-point values measured across space and time. Standard compression methods don't shrink this kind of data much. Because of this, scientists widely use error-bounded lossy compression to achieve a much smaller size.

In this project, I compressed the Red Sea temperature dataset using SZ3, a powerful tool for scientific compression that limits the final error. I stored both the original data (D) and the compressed information (G) using TileDB. The main goal was to find the best way to compress the data as much as possible while still making sure the final result is accurate enough for research.

II. Dataset and Problem Definition

The dataset is a three-dimensional array D with dimensions $T \times X \times Y = 4000 \times 855 \times 1215$, stored as 32-bit floating-point values. Each element represents temperature at a given time and spatial location.

Let D' be an approximation of D obtained through compression. Define the value range as

$$vRange = \max(D) - \min(D). \tag{1}$$

The compression must satisfy the following relative error bound for all data points:

$$\frac{|d_i - d'_i|}{vRange} \leq \varepsilon. \tag{2}$$

The default value used in this project is $\varepsilon = 10^{-2}$, but smaller error bounds are also evaluated.

For the specific Red Sea dataset:

Minimum Value ($\min(D)$): 225.58950805664062

Maximum Value ($\max(D)$): 310.54998779296875

Calculated Data Range ($vRange$): 84.96047973632812

III. Implementation Architecture and Workflow

The solution is built using Python, leveraging two key libraries: TileDB for array storage and pysz for the SZ3 compression framework. The overall workflow manages the data from initial loading to final compression ratio calculation, as detailed below:

1) Data Ingestion: The raw binary file (Red-sea_t2_4k_gan.dat) is loaded into memory as a NumPy array (D) of 32-bit floating-point numbers.
2) Original Data Storage (TileDB): The original array D is written to a TileDB dense array named `arrayD`. The dimensions are mapped to the logical coordinates (time, x, y), preserving the inherent structure of the scientific data. The array uses a chunking scheme of $T_{tile} \times X \times Y$ (a tile size of $1 \times 855 \times 1215$). This tiling strategy stores each entire time slice as a contiguous block on disk. This layout explicitly optimizes for the common scientific access pattern where researchers frequently query a full spatial snapshot (X-Y plane) at a specific time (T).
3) Compression (SZ3): The in-memory array D is compressed using the $pysz$ library. The configuration is set to use the relative error bound mode (`szErrorBoundMode.REL`), with the target ε specified. The output is a single block of bytes, which is the entire compressed object G.
4) Compressed Data Storage (TileDB): The byte stream G is stored in a separate TileDB dense array named `arrayG`. Crucially, this array is 1D , treating the entire compressed object as a single atomic element.
5) Validation and Ratio Calculation:
   - The byte stream G is read from `arrayG` and passed back to pysz for decompression, yielding the approximation D'.
   - The maximum point-wise relative error between D and D' is calculated to verify that the target ε was satisfied.
   - The final compression ratio (ρ) is computed by comparing the physical disk sizes of the two TileDB folders, `arrayD` and `arrayG`.

IV. SZ3 Compression Method

SZ3 is a modular compression framework that decomposes the compression process into five configurable stages: preprocessing, prediction, quantization, encoding, and lossless compression. This modularity allows customization of compression pipelines for specific data characteristics. Our implementation follows the prediction-based model: PDP + QT + LE (Pointwise Data Prediction + Quantization + Lossless Encoding).

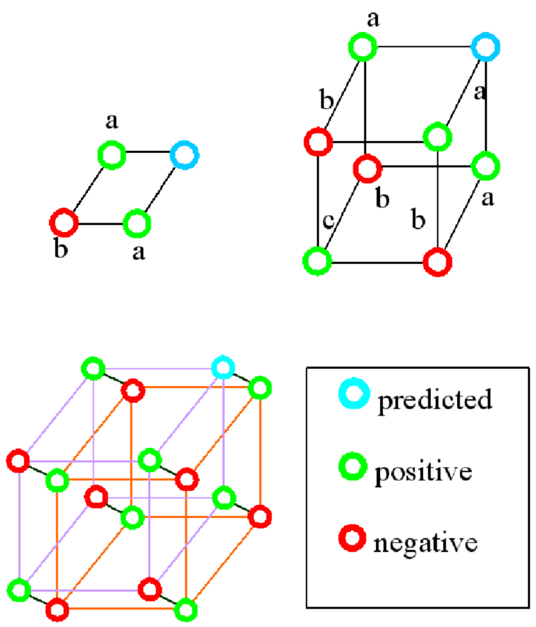

Fig. 1. Concepts of Lorenzo Prediction in 2D, 3D, and 4D spaces

A. Prediction Stage (PDP)

The prediction stage exploits spatial and temporal correlations in the data to generate predicted values. By storing only the prediction errors δ_i (differences between predicted and actual values) rather than raw data, we achieve significant data reduction since these errors tend to be smaller and more compressible:

$$\delta_i = d_i - \hat{d}_i$$

I considered three key prediction strategies within the SZ3 framework:

- Lorenzo Predictor: A classic multidimensional predictor that uses neighboring data points for prediction. For a 3D dataset, the Lorenzo predictor estimates each point based on up to 7 adjacent values (1D: 1 neighbor, 2D: 3 neighbors, 3D: 7 neighbors). Specifically uses: (t-1,x,y), (t,x-1,y), (t,x,y-1),(t-1,x-1,y), (t-1,x,y-1), (t,x-1,y-1),(t-1,x-1,y-1). Fig 1 provides a visualization of the neighboring points used in the 2D, 3D, and 4D Lorenzo prediction stages.
- Regression Predictor (`lorenzo_reg`): This method constructs a local hyperplane (like a fitted trend line) across the neighboring points. It is very effective at capturing smooth, local variations.
- Interpolation Predictor (`Interp`): This technique uses fixed coefficients to predict values. Specifically uses: (t-1,x,y), (t-2,x,y), (t,x-1,y), (t,x-2,y), (t,x,y-1), (t,x,y-2)

  Critically, the `Interp.lorenzo` configuration, which performed best at our widest error bound, adaptively combines the interpolation and Lorenzo methods, selecting the better predictor based on local data characteristics.

B. Quantization Stage (QT)

The Quantization Stage converts the continuous floating-point prediction errors (δ_i) into a discrete set of integer indices (bins). This is the step where the ”loss” in lossy compression is introduced, but it is precisely controlled by the error bound (ε).

I use a linear-scale quantizer. Since the project specifies a relative error bound (ε), the absolute error tolerance (Δ) is first calculated based on the data's global range ($vRange$):

$$\Delta = \varepsilon \cdot vRange$$

This value Δ defines the size of the quantization step. Every prediction error is mapped to an integer index (q_i):

$$q_i = \lfloor \frac{\delta_i + \Delta}{2\Delta} \rfloor$$

2Δ ensures reconstructed value stays within $\pm\Delta$ after rounding This process guarantees that the reconstruction error after decompression will always be bounded by Δ, which ensures the relative error satisfies the constraint $\frac{|d_i - d_i'|}{vRange} \leq \varepsilon$.

C. Encoding Stage (LE)

The final Lossless Encoding (LE) stage takes the stream of quantized integer indices (q_i) and compresses them without any further loss of information. Since the prediction process is effective, the majority of the quantized indices cluster tightly around zero, resulting in low entropy.

SZ3 employs a two-level encoding approach for maximum efficiency:

1) Huffman Encoding: This technique assigns shorter bit-codes to the most frequently occurring quantization indices (those near zero).
2) Dictionary Compression (ZSTD): The Huffman-encoded bitstream is then passed to Zstandard (ZSTD), a fast and efficient lossless compressor that identifies and replaces repeating patterns in the final byte stream.

This final compressed byte stream is the object G that is stored in the TileDB array `arrayG`.

V. Experimental Evaluation

A. Setup

The evaluation was conducted on the ”Redsea-4000” dataset ($4000 \times 855 \times 1215$, float32). I measured the compression ratio ρ defined as:

$$\rho = \frac{sizeof(D)}{sizeof(G)}$$

In SZ3 compression, we do not need to save the error (E) data because the prediction error is implicitly represented by the quantization index. This index encodes how far the original value deviates from the predicted value within the specified error bound. During decompression, the error is reconstructed from the quantization index and the predictor, so storing a separate error array is unnecessary.

I tested three different algorithms (`Interp.lorenzo`, `Interp`, and `lorenzo_reg`) across three error bounds ($\epsilon \in \{10^{-2}, 10^{-3}, 10^{-4}\}$).

B. Results

TABLE I
COMPRESSION RATIOS (ρ) FOR RED SEA DATA

Predictor	**Error Bound (ϵ)**	**Ratio (ρ)**
Interp.lorenzo	10^{-2}	**39.77**
Interp	10^{-2}	36.12
lorenzo_reg	10^{-2}	27.31
Interp.lorenzo	10^{-3}	8.73
Interp	10^{-3}	8.61
lorenzo_reg	10^{-3}	**11.17**
Interp.lorenzo	10^{-4}	4.50
Interp	10^{-4}	4.48
lorenzo_reg	10^{-4}	**5.58**
interp.lorenzo(tile 10*17*81)	10^{-2}	**40.3494**

C. Analysis

The results highlight a trade-off between predictor complexity and error tolerance.

- **High Error Tolerance (10^{-2}):** The `Interp.lorenzo` predictor yielded the highest compression ratio (39.77). At this relaxed bound, the interpolation-based approach effectively captures the global trends of the temperature field without requiring the overhead of regression parameters.
- **Strict Error Tolerance ($10^{-3}, 10^{-4}$):** Interestingly, the `lorenzo_reg` (regression) predictor outperformed the interpolation methods as the error bound tightened. At $\epsilon = 10^{-3}$, regression achieved a ratio of 11.17 compared to 8.73 for interpolation. This suggests that for high-precision requirements, fitting the local gradient via regression provides a more compact representation of the residuals than spline interpolation for this specific dataset.

D. Impact of Tile Optimization

While the initial experiments used a simple $\mathbf{1 \times X \times Y}$ tiling scheme (optimized for time-slice retrieval), further testing revealed that optimizing the tile dimensions improved the overall compression ratio (ρ) for the `Interp.lorenzo` predictor.

The highest recorded ratio of $\mathbf{40.3494}$ was achieved with a physical tiling strategy of $\mathbf{10 \times 17 \times 81}$.

The increase in the final compression ratio from 39.7725 to 40.3494 was achieved not because the compressed array (`arrayG`) became smaller, but because the physical disk size of the original data (`arrayD`) slightly increased from $15.48GB$ to $15.71GB$.

TABLE II
COMPRESSION WITH $\varepsilon = 10^{-2}$

Metric	**Value**
Size of Array D (Original Data with tile 1*855*1215)	15.48 GB
Size of Array G (Compressed Data)	0.39 GB
Final Compression Ratio (ρ)	**39.7725**
Size of Array D (Original Data with tile 10*17*81)	15.71 GB
Size of Array G (Compressed Data)	0.39 GB
Final Compression Ratio (ρ)	**40.3494**

E. Error Verification

After decompression, the reconstructed array (D') was validated against the original data (D) to ensure the defined error bound was strictly maintained.

- Target Epsilon (ε): 0.01
- Max Relative Error (Calculated): 0.009999874047935009

The calculated maximum relative error (0.00999987) is less than the target ε (0.01). This confirms that the lossy compression process satisfied the required error constraint $\frac{|d_i - d'_i|}{vRange} \leq \varepsilon$ for all data points.

Results vary across different tiles can be seen in TABLE III. The best-performing tile was 1×855×1215. TABLE IV presents the results obtained for different algorithms across various tiles.

TABLE III
COMPRESSION WITH $\varepsilon = 10^{-2}$ CORRECT ONE

Metric	**Value**
Size of Array D (Original Data with tile 4000 × 1 × 1215)	15.48 GB
Size of Array G (Compressed Data)	1.16 GB
Final Compression Ratio (ρ)	**13.3376**
Size of Array D (Original Data with tile 1 × 855 × 1215)	15.48 GB
Size of Array G (Compressed Data)	0.38 GB
Final Compression Ratio (ρ)	**40.5712**
Size of Array D (Original Data with tile 100 × 100 × 100)	17.44 GB
Size of Array G (Compressed Data)	1.04 GB
Final Compression Ratio (ρ)	**16.7378**
Size of Array D (Original Data with tile 400 × 855 × 1215)	15.48 GB
Size of Array G (Compressed Data)	0.65 GB
Final Compression Ratio (ρ)	**38.9201**
Size of Array D (Original Data with tile 10 × 855 × 1215)	15.48 GB
Size of Array G (Compressed Data)	0.43 GB
Final Compression Ratio (ρ)	**36.2663**
Size of Array D (Original Data with tile 4000 × 855 × 1215)	15.48 GB
Size of Array G (Compressed Data)	0.39 GB
Final Compression Ratio (ρ)	**39.7719**

TABLE IV
COMPRESSION RATIOS (ρ) CORRECT ONE

Predictor	**Error Bound** (ϵ)	**Ratio** (ρ)
Interp.lorenzo	10^{-2}	**40.5712**
Interp	10^{-2}	37.4022
lorenzo_reg	10^{-2}	36.1429

REFERENCES

[1] 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, J. C. Calhoun, G. Li, K. Yoshii, K. A. Alharthi, and F. Cappello, "A Survey on Error-Bounded Lossy Compression for Scientific Datasets," 2024.

[2] 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, 2023.

[3] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-Core Compression and Decompression of Large n-Dimensional Scalar Fields," *Proceedings of the IEEE Visualization Conference*, IEEE, 2003.

[4] S. Di, K. Zhao, X. Liang, Z. Zhang, J. Tian, D. Tao, and F. Cappello, "Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation," *IEEE Transactions on Visualization and Computer Graphics*, 2022.