HawkZip

Amol Bhagavathi, Avi Kaufman, Patryk Wojtkowski
University of Iowa, Department of Computer Science, Iowa City, Iowa
(Dated: April 25, 2025)

In this project, we investigate different algorithms and optimization techniques for improving the compression ratio and throughput of a CPU error-bounded lossy compressor (HawkZip). HawkZip is originally a parallel error-bounded lossy compressor that exploits the idea of Shared Address Space via OpenMP parallel programming interface. In this work we implement two different algorithms (1) Delta encoding + quantization and (2) 2D-Lorenzo Prediction. Our benchmarks were taken on the CESM-ATM data set and were compared across different hardware. Our findings show that the 2D-Lorenzo implementation yields up to 4x compression ratio but throughput decreased to 33% of its original value. Alternatively, the Delta-encoding + Zstd path improves compression ratio, while compressing at lower than baseline speed but decoding quicker, making it the more balanced choice. These results suggest an unequal trade-off between compression ratio and throughput.

I. INTRODUCTION

HawkZip aims to raise compression efficiency while respecting a strict, user–supplied error bound ε. Our best performing implementation keeps the original *block-based quantization* of the provided baseline, but adds two complementary ideas:

a. Block-local 1-D prediction. Within every fixed 32-value block we store the first quantized code q_0 unchanged and then record only the differences

$$\delta_i = q_i - q_{i-1}, \qquad i = 1, \ldots, 31, \tag{1}$$

where $q_i = \text{round}\big(x_i/(2\varepsilon)\big)$. Because the majority of δ_i are small integers, each block needs far fewer bits than if we had stored all q_i directly. The deltas are bit-plane packed exactly as in the baseline, so the kernel remains embarrassingly parallel.

b. Second stage with Zstd. After all blocks are packed we run the resulting byte stream through the Zstandard codec. An eight-byte header `(origSize, compSize)` is prepended so that decompression is a single call to `ZSTD_decompress` followed by the inverse delta-decode. The extra pass adds runtime overhead but removes the residual byte-level redundancy left by bit-plane packing.

Taken together, these two steps deliver the best compression-ratio / throughput trade-off we observed.

Another way of improving compression ratio is to exploit the structure of the data we are compressing. The benchmark we test on is the CESM-ATM climate grid of which the data contained exhibits strong 2 dimensional correlations. A simple way of exploiting these correlations is raising our 1D Lorenzo prediction to a 2D Lorenzo predictor:

$$\hat{x}_{i,j} = x_{i-1,j} + x_{i,j-1} - x_{i-1,j-1}, \tag{2}$$

In this method we predict each point from its top, left, and corner neighbors. The residual is the difference between the true value and the prediction:

$$\begin{aligned} r_{i,j} &= x_{i,j} - \hat{x}_{i,j} \\ &= x_{i,j} - \big(x_{i-1,j} + x_{i,j-1} - x_{i-1,j-1}\big) \\ &= x_{i,j} - x_{i-1,j} - x_{i,j-1} + x_{i-1,j-1}. \end{aligned} \tag{3}$$

The resulting residuals tend to be smaller than in the 1D case allowing us to achieve higher compression ratios for the same error-bound. However, each prediction now includes an addition and a subtraction, which increases the computational overhead. Additionally, data dependency is greatly increased, leading to difficulty in parallelization.

In order to overcome this data dependency, we modify our parallelization to a "wavefront" parallelization. In the wavefront method, we split up threads along the diagonal (see Figure 1). Although this method allows us to effectively parallelize our code, not all threads do the same amount of work, thus we do not achieve the same efficiency increase that we see with other simple block parallelism methods.

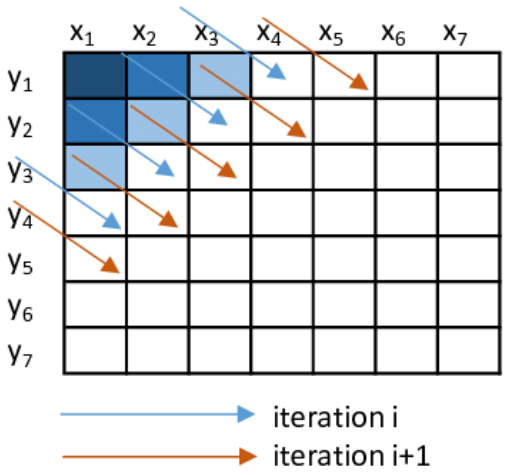

Figure 1. "Wavefront Parallelism"

Just like the first method we run the result through the lossless compressor Zstd which further increases compression ratio. Alternative lossless compression were tested as well and are documented in the benchmarks section.

II. METHODOLOGY

The enhanced `hawkZip` keeps the baseline's block-parallel layout (`BLOCK_SIZE`=32, OpenMP threads working on disjoint blocks) but introduces three changes. The first is within each block we store *forward deltas* rather than absolute quantized values. Second, these deltas are packed plane-by-plane using a bit-width chosen per block. Third, the resulting byte stream is passed through the ZSTD lossless compressor and wrapped with an 8-byte header. The baseline stops after the first, and therefore forgoes the entropy reduction of deltas and the additional space saving from Zstd.

Compression kernel. Each thread quantises its 32-value block using

$$q_i = \mathrm{round}\big(x_i/(2\varepsilon)\big),$$

records the initial code and then the deltas $\delta_0 = q_0$, $\delta_i = q_i - q_{i-1}$. The magnitudes $|\delta_i|$ populate a work array while the signs are packed into a 32-bit mask. The maximum magnitude in the block defines a fixed bit-width that is stored as a one-byte header. Threads compute contiguous write offsets with a prefix sum, emit the sign mask and then the b bit-planes. When all blocks are written, the contiguous payload is compressed with `ZSTD_compress`; the original and compressed sizes occupy the first eight bytes of the final buffer.

Decompression kernel. The decoder reads the header, expands the Zstd payload into a scratch buffer, and scans the per-block bit-widths to recover read offsets. For a non-zero block it reads the sign mask, unpacks the b bit-planes back into $|\delta_i|$, restores the signed deltas, and reconstructs the quantised codes with $Q_0 = \delta_0$ and $Q_i = Q_{i-1} + \delta_i$. The original floats follow from $x_i = Q_i\,(2\varepsilon)$. Compared to the baseline, the main extra work is the prefix-sum of deltas and the Zstd (de)compression call.

For specific optimizations to the kernels, we used both loop unrolling and bitwise operations. Specifically, we used loop unrolling in the compression kernel function that converts the original floating-point data into quantized integer values. This is because, like mentioned in class, loop unrolling works best when there are no data dependencies between iterations, and this loop met that condition. By unrolling this loop to process 4 elements at a time, we reduced the loop control overhead, which allows the compiler to better schedule the independent arithmetic operations across available execution units.

The second optimization applied was to replace modulo with bitwise operations. When dividing by a power of 2, the modulo operation can be replaced with a bitwise AND using a mask of one less than the divisor. This optimization eliminates the need for expensive division instructions. The performance benefit is most notable in the compression algorithm's inner loops where these operations occur frequently during bit manipulation and indexing calculations.

III. EVALUATION

We tested our implementation on 3 different processor. An Apple M4 Pro with 12 cores, an Apple M1 with 8 Cores, and an Intel Xeon Gold 6240R with 48 cores.

The benchmarks in Tables I–V contrasts the baseline approach with our 1-D–delta + bit-plane + Zstd path on three machines and eight error bounds. On every platform the enhanced version yields markedly higher compression ratios—up to nearly 13× more than baseline at the looser 10^{-1} tolerance. Even after accounting for the Zstd overhead, decompression remains similar to the baseline. We chose a very aggressive compression level of 42 for Zstd. While making our throughput suffer it provided the best compression ratio which is what we were trying to achieve. Lowering the level would increase throughput ,especially at tighter bounds, but at the cost of lowering compression ratio.

Table VI shows that the optimizations applied after the initial rewrite push performance higher. On the Intel host the same algorithm has an increase in throughput and posts smaller but consistent gains at tighter bounds, demonstrating that loop unrolling and using bitwise instead of modulo operations provide performance gains.

To confirm that Zstd is the most cost-effective lossless compressor we swapped in three popular alternatives. LZ4 (Table VII) is a pure LZ77/ANS that compresses fastest but has a sacrifice in compression ratio. Deflate (Table VIII) combines Huffman coding with LZ77; its ratios sit between LZ4 and Zstd while speed is similar to LZ4 on these data. BLOSC (Table IX) adds shuffle and bit-shuffle filters before an internal codec; its results are similar to LZ4. Zstd's hybrid design—Huffman + LZ77 + ANS—makes it more aggressive than the other codecs, and although this costs a bit of compression speed, the much higher compression ratio easily justifies the trade-off.

Finally, Table XII reports Peak-Signal-to-Noise Ratio and Structural-Similarity scores for both the baseline and our best implementation. Because delta coding and Zstd operate strictly on already-quantised integers, neither step changes the pointwise error profile generated by the original quantiser. PSNR and SSIM remain identical to the baseline at every tested error level, confirming that the substantial space savings come at no cost to reconstructed data quality.

IV. BENCHMARKS

Table I. Baseline- M1

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	69.034585	0.90 – 2.12 GB/s	3.69 – 11.53 GB/s	Success
10^{-2}	12.229069	0.58 – 1.27 GB/s	1.14 – 1.47 GB/s	Success
10^{-3}	5.7939684	0.34 – 0.74 GB/s	0.42 – 0.70 GB/s	Success
10^{-4}	3.630575	0.23 – 0.49 GB/s	0.28 – 0.46 GB/s	Success
10^{-5}	2.639421	0.18 – 0.45 GB/s	0.21 – 0.35 GB/s	Success
10^{-6}	2.071815	0.11 – 0.30 GB/s	0.12 – 0.30 GB/s	Success
10^{-7}	1.705065	0.13 – 0.31 GB/s	0.10 – 0.25 GB/s	Fail - 2180 unbounded data points
10^{-8}	1.448677	0.12 – 0.22 GB/s	0.08 – 0.21 GB/s	Fail - 810297 unbounded data points

Table II. Baseline – M4 Pro

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	69.034584	2.822064 GB/s	15.287634 GB/s	Success
10^{-2}	12.229069	2.499445 GB/s	3.979484 GB/s	Success
10^{-3}	5.739684	1.614807 GB/s	2.052337 GB/s	Success
10^{-4}	3.630575	1.114585 GB/s	1.339622 GB/s	Success
10^{-5}	2.639421	0.886160 GB/s	1.029110 GB/s	Success
10^{-6}	2.071815	0.721236 GB/s	0.838482 GB/s	Success
10^{-7}	1.705065	0.614232 GB/s	0.686790 GB/s	Fail – 2180 unbounded points
10^{-8}	1.448677	0.530419 GB/s	0.588519 GB/s	Fail – 810 297 unbounded points

Table III. Baseline – fast × Intel

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	69.034584	0.721746 GB/s	3.783187 GB/s	Success
10^{-2}	12.229069	0.443785 GB/s	0.809685 GB/s	Success
10^{-3}	5.739684	0.414760 GB/s	0.498180 GB/s	Success
10^{-4}	3.630575	0.334685 GB/s	0.365711 GB/s	Success
10^{-5}	2.639421	0.300431 GB/s	0.298054 GB/s	Success
10^{-6}	2.071815	0.214613 GB/s	0.258815 GB/s	Success
10^{-7}	1.705065	0.219631 GB/s	0.219831 GB/s	Fail – 2180 unbounded points
10^{-8}	1.448677	0.191390 GB/s	0.211466 GB/s	Fail – 810 297 unbounded points

Table IV. Improved bit-packing + zstd – M4 Pro

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	882.863831	0.233542 GB/s	6.009615 GB/s	Success
10^{-2}	93.002228	0.064096 GB/s	2.926385 GB/s	Success
10^{-3}	24.706043	0.025805 GB/s	1.619896 GB/s	Success
10^{-4}	10.965947	0.017989 GB/s	1.385923 GB/s	Success
10^{-5}	6.148518	0.013491 GB/s	1.038323 GB/s	Success
10^{-6}	3.919807	0.011139 GB/s	0.860603 GB/s	Success
10^{-7}	2.800466	0.009988 GB/s	0.732289 GB/s	Fail – 2180 unbounded points
10^{-8}	2.167144	0.009144 GB/s	0.681800 GB/s	Fail – 810 297 unbounded points

Table V. Improved bit-packing + zstd – fast × Intel

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	882.863831	0.117608 GB/s	3.448156 GB/s	Success
10^{-2}	93.002228	0.029232 GB/s	1.379796 GB/s	Success
10^{-3}	24.706043	0.010572 GB/s	0.887924 GB/s	Success
10^{-4}	10.965947	0.008082 GB/s	0.657649 GB/s	Success
10^{-5}	6.148518	0.005976 GB/s	0.541203 GB/s	Success
10^{-6}	3.919807	0.004807 GB/s	0.466743 GB/s	Success
10^{-7}	2.800466	0.004488 GB/s	0.386956 GB/s	Fail – 2180 unbounded points
10^{-8}	2.167144	0.003967 GB/s	0.319505 GB/s	Fail – 810 297 unbounded points

Table VI. Improved bit-packing + zstd + optimizations – fast × Intel

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	882.863831	0.118241 GB/s	3.514180 GB/s	Success
10^{-2}	93.002228	0.028908 GB/s	1.559655 GB/s	Success
10^{-3}	24.706043	0.012684 GB/s	0.913543 GB/s	Success
10^{-4}	10.965947	0.008100 GB/s	0.705471 GB/s	Success
10^{-5}	6.148518	0.005797 GB/s	0.538375 GB/s	Success
10^{-6}	3.919807	0.005405 GB/s	0.469676 GB/s	Success
10^{-7}	2.800466	0.004775 GB/s	0.414802 GB/s	Fail – 2180 unbounded points
10^{-8}	2.167144	0.004221 GB/s	0.344821 GB/s	Fail – 810 297 unbounded points

Table VII. Improved bit-packing + LZ4 + optimizations – fast × Intel

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	349.161438	2.674074 GB/s	3.842393 GB/s	Success
10^{-2}	38.316273	1.369662 GB/s	2.602196 GB/s	Success
10^{-3}	12.316072	0.794248 GB/s	1.113704 GB/s	Success
10^{-4}	6.681936	0.556312 GB/s	0.756065 GB/s	Success
10^{-5}	4.253409	0.447663 GB/s	0.653156 GB/s	Success
10^{-6}	2.989870	0.384186 GB/s	0.572959 GB/s	Success
10^{-7}	2.282271	0.312013 GB/s	0.465046 GB/s	Fail – 2180 unbounded points
10^{-8}	1.842388	0.292136 GB/s	0.614147 GB/s	Fail – 810 297 unbounded points

Table VIII. Improved bit-packing + Deflate (zlib) + optimizations – fast × Intel

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	695.372253	0.908334 GB/s	4.118064 GB/s	Success
10^{-2}	71.075035	0.162133 GB/s	1.289256 GB/s	Success
10^{-3}	20.222336	0.062801 GB/s	0.682434 GB/s	Success
10^{-4}	9.719940	0.050608 GB/s	0.468146 GB/s	Success
10^{-5}	5.599777	0.039194 GB/s	0.357449 GB/s	Success
10^{-6}	3.643720	0.032123 GB/s	0.293240 GB/s	Success
10^{-7}	2.641372	0.029882 GB/s	0.250266 GB/s	Fail – 2180 unbounded points
10^{-8}	2.065564	0.027206 GB/s	0.209956 GB/s	Fail – 810 297 unbounded points

Table IX. Improved bit-packing + BLOSC + optimizations – fast × Intel

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	316.101410	2.437554 GB/s	4.172485 GB/s	Success
10^{-2}	34.939205	1.088659 GB/s	1.624938 GB/s	Success
10^{-3}	11.480110	0.569267 GB/s	0.916302 GB/s	Success
10^{-4}	6.290413	0.373300 GB/s	0.659182 GB/s	Success
10^{-5}	4.117020	0.313708 GB/s	0.581767 GB/s	Success
10^{-6}	2.786250	0.285597 GB/s	0.529044 GB/s	Success
10^{-7}	2.051002	0.316712 GB/s	0.492523 GB/s	Fail – 2180 unbounded points
10^{-8}	1.630627	0.306777 GB/s	0.486449 GB/s	Fail – 810 297 unbounded points

Table X. 2D Lorenzo (wavefront parallelization) with ZSTD

Error Bound	Compression Ratio	Compression Throughput	Compr. Avg.	Decompression Throughput	Decompr. Avg.	Success/Fail Error Check
10^{-1}	703.774109	0.0433 – 0.1068 GB/s	0.1025 GB/s	0.0985 – 0.1318 GB/s	0.1225 GB/s	Success
10^{-2}	91.698326	0.0503 – 0.1041 GB/s	0.0991 GB/s	0.0578 – 0.1215 GB/s	0.1054 GB/s	Success
10^{-3}	26.712452	0.0562 – 0.1023 GB/s	0.0948 GB/s	0.0562 – 0.1153 GB/s	0.1102 GB/s	Success
10^{-4}	14.100206	0.0348 – 0.0965 GB/s	0.0857 GB/s	0.0942 – 0.1143 GB/s	0.1085 GB/s	Success

Table XI. Improved using SZ3

Error Bound	Compression Ratio	Compression Throughput	Decompression Throughput	Success/Fail Error Check
10^{-1}	5182.963407	0.521161 GB/s	2.318442 GB/s	Success
10^{-2}	466.011039	0.505737 GB/s	2.176678 GB/s	Success
10^{-3}	77.462852	0.444886 GB/s	1.571386 GB/s	Success
10^{-4}	21.341606	0.336091 GB/s	0.779575 GB/s	Success
10^{-5}	7.873913	0.284101 GB/s	0.362244 GB/s	Success
10^{-6}	4.333084	0.242647 GB/s	0.238292 GB/s	Success
10^{-7}	2.896805	0.159763 GB/s	0.164692 GB/s	Fail – 727 unbounded points
10^{-8}	2.093273	0.134134 GB/s	0.129433 GB/s	Fail – 34 022 unbounded points

Table XII. Quality Metrics (PSNR and SSIM)

Error Bound	PSNR (Baseline) [dB]	PSNR (Best) [dB]	SSIM (Baseline)	SSIM (Best)
10^{-4}	84.77	84.77	0.999713	0.999713
10^{-3}	64.76	64.76	0.979035	0.979035
10^{-2}	44.78	44.78	0.692282	0.692282
10^{-1}	27.68	27.68	0.085973	0.085973

V. CONCLUSION

We have presented two methods for improving the performance of the baseline compressor:

- A block-local 1-D prediction
- 2D-Lorenzo Prediction

Extensive benchmarks across all three test machines show that **block-local 1-D delta prediction followed by Zstd** is the most practical upgrade to the baseline. At the representative error bound 10^{-4} it *triples* the compression ratio while compressing at roughly half the baseline speed and decoding noticeably faster. This offers the best overall balance for workflows. The 2-D Lorenzo variant pushes the ratio still higher, but its encode time grows by an order of magnitude. This is useful for when storage space is the priority for usage.

Our survey of lossless back-ends confirms that **Zstandard**—with its Huffman + LZ77 + ANS pipeline—delivers the most favorable trade-off. The extra encode time is justified by the substantial gain in compression ratio..

Additionally, quality metrics remain identical to the baseline for every tested error bound, confirming that our method does not introduce any additional error beyond the specified ε

Finally, we would like to note how oversubscription yielded better throughput for our experiments by utilizing more threads than physical cores available. This boosted throughput on every platform we tested. The 12-core Apple M4 Pro reached its peak at 32 threads, while the 48-core Intel host performed best with 64 threads. This could be because each compression task is so lightweight that it often stalls on cache misses rather than arithmetic. The extra threads therefore give the scheduler work to run whenever another thread is waiting on memory, keeping the cores busy and hiding latency. Noticing this helped us increase throughput at no implementation cost.

[1] S. Di, X. Liang, Z. Chen, and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *Proc. IEEE IPDPS*, 2016, pp. 730–739.

[2] X. Wang, S. Di, and F. Cappello, "QCAT: Quality Characterization Analysis Tool," GitHub repository, `https://github.com/szcompressor/qcat` (accessed 24 Apr 2025).

[3] SZ Compressor Developers, "SZ3: Error-bounded Lossy Compressor for Scientific Data," GitHub repository, `https://github.com/szcompressor/SZ3` (accessed 24 Apr 2025).

[4] Y. Collet *et al.*, "Zstandard Compression Library," GitHub repository, `https://github.com/facebook/zstd`, version 1.5.5 (accessed 24 Apr 2025).

[5] Y. Collet, "LZ4 Lossless Compression Algorithm," GitHub repository, `https://github.com/lz4/lz4`, version 1.9.4 (accessed 24 Apr 2025).

[6] F. Alted, V. Haenel, *et al.*, "c-blosc: A High-Performance Compressor Optimised for Binary Data," GitHub repository, `https://github.com/Blosc/c-blosc` (accessed 24 Apr 2025).

[7] J. Metz, "Deflate (zlib) Compressed Data Format," online documentation, `https://github.com/libyal/assorted/blob/main/documentation/Deflate%20(zlib)%20compressed%20data%20format.asciidoc` (accessed 24 Apr 2025).