universität innsbruck

2021 Data Compression Conference

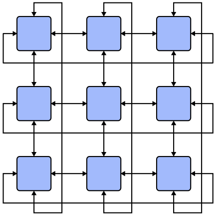
ndzip

A High-Throughput Parallel Lossless Compressor for Scientific Data

Fabian Knorr, Peter Thoman and Thomas Fahringer

Distributed and Parallel Systems Group
University of Innsbruck, Austria

Introduction



Algorithms in **High Performance Computing (HPC)** commonly work with large multi-dimensional grids of floating point data. Some important algorithms are limited by network bandwidth.

- Distributed Matrix Transpose
- Cooley-Tukey Fast Fourier Transform
- ...

Data compression can transparently increase effective bandwidth.

- Must be **lossless** in the general case
- Saturating the interconnect requires **high throughput**

Specialized Floating-Point Compressors

General-purpose byte-oriented compressors are **not a good fit** for floating-point data.

- Grid data is often **smooth**, but values are still **individually unique**
- Effective decorrelation **requires interpretation** of the floating-point representation
- Most well known compressors have **asymmetric performance**

Typical building blocks of existing specialized compressors are:

1. **Prediction** of each floating point value, local or global
2. A **difference operator** yielding a residual from the prediction
3. An **encoding scheme** favoring small residuals.

Existing specialized algorithms [6][3][1][2] are either trading throughput for higher compression ratios or are not optimized for modern hardware.

ndzip Overview

ndzip is a novel, lossless block compression scheme for multi-dimensional grids of univariate floating-point data.

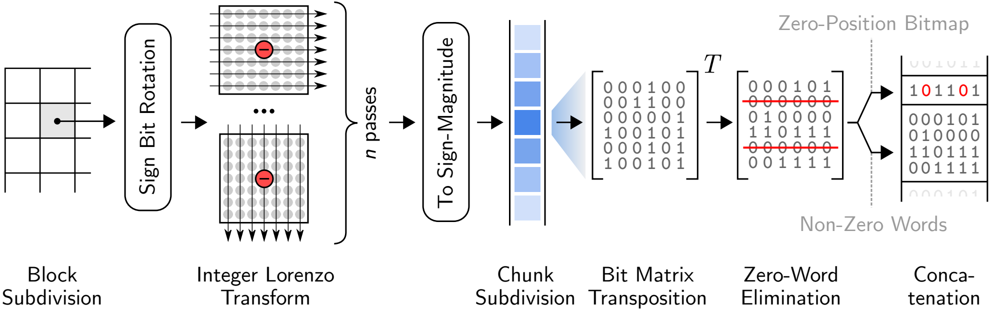
Its design enables **efficient, highly parallel** implementation on modern hardware through

- **Locality:** values are decorrelated only from direct neighbors
- **Parallelism:** coarse-grained between blocks, fine-grained within compression stages
- **Dimensionality-awareness:** grid size is an input for multidimensional decorrelation

We present the ndzip algorithm and an implementation on **x86_64 hardware** using the AVX2 vector extension.

ndzip Compression Pipeline

ndzip subdivides the grid into **fixed-size blocks**, which are compressed independently.

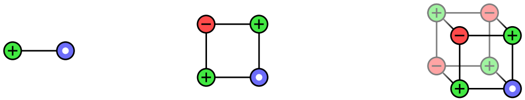


Decompression simply reverses each compression step; ndzip is *symmetric*.

The Lorenzo Predictor [4]

Predict values from all known neighbors in a length-2 hypercube:

1D | **2D** | **3D**



- positive coefficient
- negative coefficient
- prediction

1D: $\begin{bmatrix} 1 & p_1 \end{bmatrix} \rightarrow p_1 = 1$

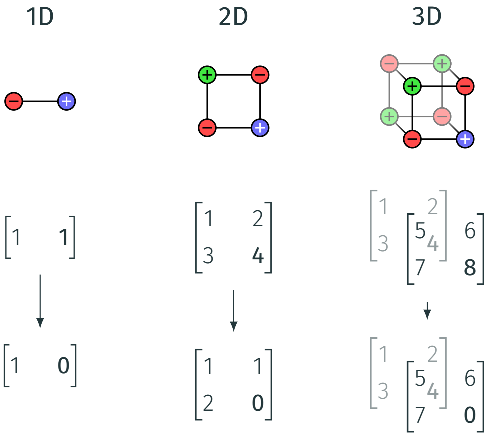
2D: $\begin{bmatrix} 1 & 2 \\ 3 & p_2 \end{bmatrix} \rightarrow p_2 = -1 + 2 + 3 = 4$

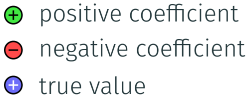
3D: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & p_3 \end{bmatrix} \rightarrow p_3 = 1 - 2 - 3 + 4 - 5 + 6 + 7 = 8$

Very effective [6], but reconstruction during decompression **limits parallelism**.

New: Integer Lorenzo Transform

Calculating the prediction *residuals* directly without an intermediate step yields a separable transform in the multi-dimensional case.





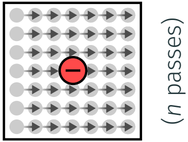
Since this transform is not reversible in floating-point arithmetic, it is **approximated in the integer domain**.

Vectorized Integer Lorenzo Transform

The Integer Lorenzo Transform is separable: An n-dimensional transform is equivalent to performing a one-dimensional transform along each of the n dimensions.

Forward Transform

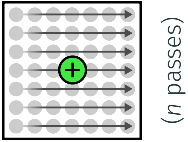
The forward transform is fully parallel in each dimension. Each vector instruction computes 8 single-precision or 4 double-precision deltas simultaneously.



difference

Inverse Transform

The inverse transform has a dependence on the predecessor value in each row. Separability exposes $n - 1$ dimensions of parallelism in each step. The 1-dimensional case cannot be efficiently parallelized on this hardware.



prefix sum

Residual Value Encoding

Small integer residuals have many **redundant sign bits**, which can be encoded efficiently using the vertical bit-packing scheme introduced in [7].

1. Turn redundant bits into zero-bits with a sign-magnitude representation
2. For each 32- (64-) word block, transpose the 32×32 (64×64) bit matrix
3. Eliminate zero-rows and prepend a header bitmap encoding the omitted rows

$$
\begin{array}{l} \text{Word 0} \\ \text{Word 1} \\ \text{Word 2} \\ \text{Word 3} \\ \text{Word 4} \\ \text{Word 5} \\ \text{Word 6} \\ \text{Word 7} \end{array}
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}^T
=
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}
\rightleftharpoons
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}
\begin{array}{l} \text{Head} \\ \text{Bit 0} \\ \text{Bit 4} \\ \text{Bit 5} \\ \text{Bit 6} \\ \text{Bit 7} \end{array}
$$

Vectorized Residual Value Encoding

Vertical bit packing is complex to implement efficiently, but operates at a 32-bit granularity and requires little branching in the compaction step.

Naive implementation: 32×32 nested loop with one `shift+and+or` *per bit*

Complexity, autovectorized: **772 (5398) instructions** for single (double) precision.

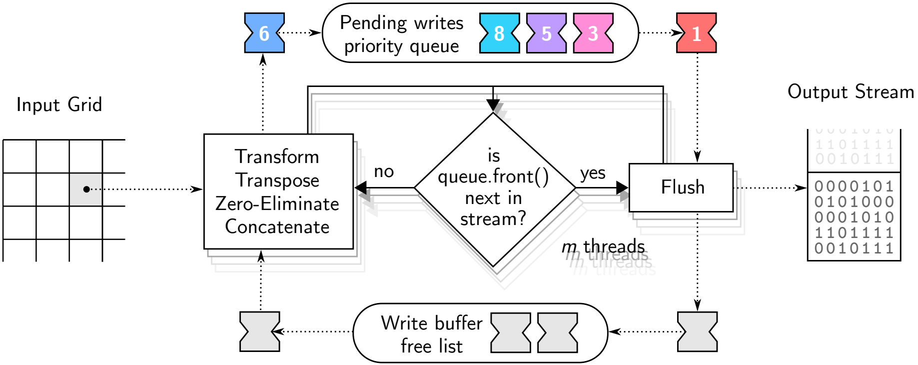
Manually vectorized two-stage implementation:

1. Transpose equivalent 32×4 *byte* matrix with `permute+unpack` vector operations $\Rightarrow$ results in a 4×32 matrix, where each element is an 8-bit column vector
2. For each output row, extract 32 bits in parallel using one `shift+vpmovmskb` (*move byte mask*) operation each

Complexity: **124 (625) instructions** for single (double) precision.

Thread Parallelism Between Blocks

Compression requires synchronization to determine output positions



Decompression can use simple work-sharing with meta-information from the compressor

Test Setup

Test Data from various scientific domains [5]:

dataset	single	double	extent
msg_sppm	✓	✓	34,874,483
msg_sweep3d	✓	✓	15,716,403
snd_thunder	✓		7,898,672
ts_gas	✓		4,208,261
ts_wesad	✓		4,588,553
hdr_night	✓		8,192 × 16,384
hdr_palermo	✓		10,268 × 20,536
hubble	✓		6,036 × 6,014
rsim	✓	✓	2,048 × 11,509
spitzer_fls_irac	✓		6,456 × 6,389
spitzer_fls_vla	✓		8,192 × 8,192
spitzer_frontier	✓		3,874 × 2,694

dataset	single	double	extent
asteroid	✓		500 × 500 × 500
astro_mhd	✓		128 × 512 × 1024
astro_mhd		✓	130 × 514 × 1026
astro_pt	✓	✓	512 × 256 × 640
flow		✓	16 × 7,680 × 1,0240
hurricane	✓		100 × 500 × 500
magrecon	✓		512 × 512 × 512
miranda	✓		1,024 × 1,024 × 1,024
redsea	✓	✓	50 × 500 × 500
sma_disk	✓		301 × 369 × 369
turbulence	✓		256 × 256 × 256
wave	✓	✓	512 × 512 × 512

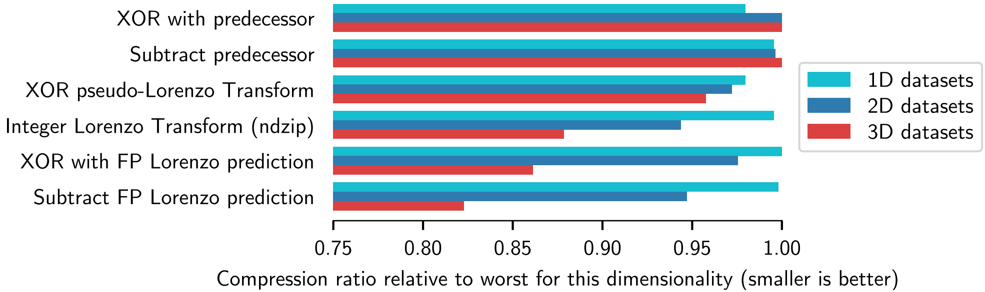
Hardware: AMD Ryzen 9 3900X (12 cores, 24 threads), 64 GB DDR4-3200 RAM

Approximation Quality of the Integer Lorenzo Transform

Integer approximation slightly lowers the achieved compression ratio, but still profits from higher dimensionality.

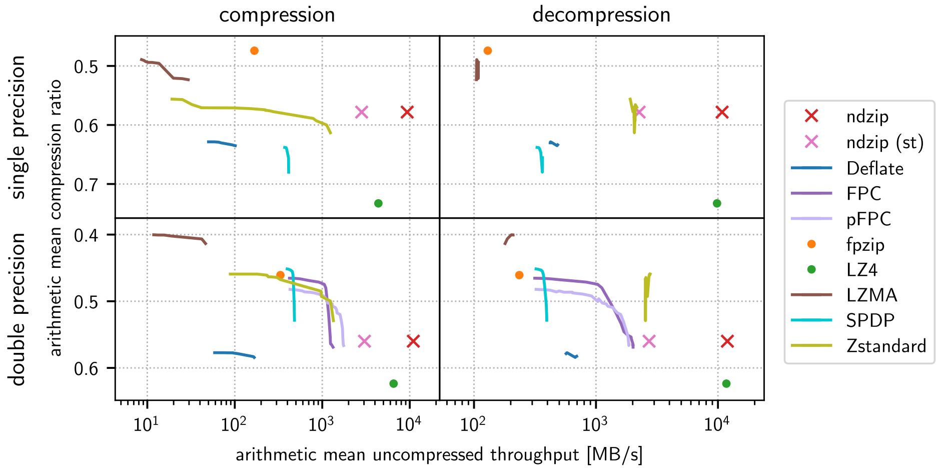
Recall

The Integer Lorenzo Transform is an approximation of the floating-point Lorenzo predictor, necessary for efficient parallel decompression.



Compressor Efficiency

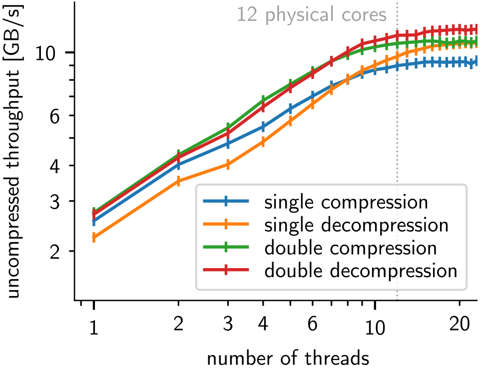
ndzip is **6× faster** than the second-fastest specialized, parallel compressor pFPC



Parallel Scaling

ndzip profits significantly from many-threaded execution. Decompression, which requires no synchronization, is the most threading-friendly.

Reference: The throughput of optimized memory-to-memory copy is 16.3 GB/s on this system, as reported by the STREAM benchmark.



Conclusion & Future Directions

ndzip is a novel, lossless block compression schemes for floating-point data.

For the targeted hardware, we demonstrated an implementation that achieves throughput unprecedented by existing specialized floating-point compressors. This is achieved with

- A design that exposes **data locality** and **multiple levels of parallelism**
- The novel, data-parallel **Integer Lorenzo Transform** for decorrelation
- A hardware-friendly **residual coding scheme**

Future Directions

We are currently working on a GPU implementation, which profits from the same design decisions. Stay tuned!

Part of the Celerity project

Celerity[1]

High-level C++ for Accelerator Clusters

ndzip was developed as part of the Celerity project, a **distributed-memory runtime** for accelerator clusters. Celerity automatically derives communication and execution schedules for programs while providing an expressive C++ API to the user.

DPS Distributed and Parallel Systems

universität innsbruck

[1]`https://celerity.github.io`

Thank You!

ndzip is available at `https://github.com/fknorr/ndzip`.

If you have questions, feel free to contact me at `fabian@dps.uibk.ac.at`.

- M. Burtscher and P. Ratanaworabhan.
  FPC: A high-speed compressor for double-precision floating-point data.
  IEEE Tr. on Computers, 58(1):18–31, 2008.
- M. Burtscher and P. Ratanaworabhan.
  pFPC: A parallel compressor for floating-point data.
  In *2009 Data Compression Conference*, pages 43–52. IEEE, 2009.
- S. Claggett, S. Azimi, and M. Burtscher.
  SPDP: An automatically synthesized lossless compression algorithm for floating-point data.
  In *2018 DCC*, pages 335–344. IEEE, 2018.
- L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak.
  Out-of-core compression and decompression of large n-dimensional scalar fields.
  In *Computer Graphics Forum*, volume 22, pages 343–348. Wiley Online Library, 2003.

F. Knorr, P. Thoman, and T. Fahringer.
Datasets for Benchmarking Floating-Point Compressors.
arXiv e-prints, page arXiv:2011.02849, Nov. 2020.

P. Lindstrom and M. Isenburg.
Fast and efficient compression of floating-point data.
IEEE Transactions on Vis. and Comp. graphics, 12(5):1245–1250, 2006.

A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher.
MPC: a massively parallel compression algorithm for scientific data.
In *2015 IEEE International Conference on Cluster Computing*, pages 381–389. IEEE, 2015.