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

**Decompression** simply reverses each compression step; ndzip is symmetric.
Predict values from all known neighbors in a length-2 hypercube:

1D

\[
\begin{bmatrix}
1 & p_1 \\
3 & p_2 \\
\end{bmatrix}
\]

\[
p_1 = 1 \\
p_2 = -1 + 2 + 3 = 4
\]

2D

\[
\begin{bmatrix}
1 & 2 \\
5 & 4 \\
\end{bmatrix}
\]

\[
p_3 = 1 - 2 - 3 + 4 - 5 + 6 + 7 = 8
\]

3D

\[
\begin{bmatrix}
1 & 2 & 3 \\
5 & 4 & 6 \\
\end{bmatrix}
\]

Predictions with positive and negative coefficients:

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

**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.
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 $32 \times 32$ ($64 \times 64$) bit matrix
3. Eliminate zero-rows and prepend a header bitmap encoding the omitted rows

$$\
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
$$

**ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data**
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:** $32 \times 32$ 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 $32 \times 4$ byte matrix with `permute+unpack` vector operations
   ⇒ results in a $4 \times 32$ 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.
Compressio

 requires synchronization to determine output positions

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

from various scientific domains [5]:

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

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

### Hardware:

AMD Ryzen 9 3900X (12 cores, 24 threads), 64 GB DDR4-3200 RAM
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.
ndzip is 6× faster than the second-fastest specialized, parallel compressor pFPC.
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.
ndzip is a novel, lossless block compression scheme 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!
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.

\[1\text{https://celerity.github.io}\]
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.**  

M. Burtscher and P. Ratanaworabhan.  
**pFPC: A parallel compressor for floating-point data.**  

S. Claggett, S. Azimi, and M. Burtscher.  
**SPDP: An automatically synthesized lossless compression algorithm for floating-point data.**  

L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak.  
**Out-of-core compression and decompression of large n-dimensional scalar fields.**  
F. Knorr, P. Thoman, and T. Fahringer. 
Datasets for Benchmarking Floating-Point Compressors.  

P. Lindstrom and M. Isenburg.  
Fast and efficient compression of floating-point data.  

A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher.  
MPC: a massively parallel compression algorithm for scientific data.  