Recent Improvements in Generic Lossy Compression

Robert Underwood, Ph.D. School of Computing
How do we use compressors?

```c
confparams_cpr->errorBoundMode = ABS;
confparams_cpr->absErrBound = 1e-4;
unsigned char *bytes = SZ_compress(SZ_FLOAT, (float *)data, &outsize, dim[4],
                                 dim[3], dim[2], dim[1], dim[0]);

zfp_stream_set_accuracy(zfp, 1e-4);
size_t outsize = zfp_compress(zfp, in_field);

unsigned char *bytes = mgard_compress(data, outsize, mgard_dims[0],
                                       mgard_dims[1], mgard_dims[2], 1e-4f);
```
Compressors have too many interfaces!

Is this thread-safe???

Global configuration struct

Compressed Stream returned “implicity”

Pointwise absolute error bound named differently

Local configuration

Dimensions provided in opposite order

Uses 1’s for unused dimensions

Configuration in function arguments

Compressors have too many interfaces!

Is this thread-safe???

Global configuration struct

Compressed Stream returned “implicity”

Pointwise absolute error bound named differently

Local configuration

Dimensions provided in opposite order

Uses 1’s for unused dimensions

Configuration in function arguments

Compressors have too many interfaces!

Is this thread-safe???

Global configuration struct

Compressed Stream returned “implicity”

Pointwise absolute error bound named differently

Local configuration

Dimensions provided in opposite order

Uses 1’s for unused dimensions

Configuration in function arguments

Compressors have too many interfaces!

Is this thread-safe???

Global configuration struct

Compressed Stream returned “implicity”

Pointwise absolute error bound named differently

Local configuration

Dimensions provided in opposite order

Uses 1’s for unused dimensions

Configuration in function arguments
Compressors have too many interfaces!

Is this thread-safe???

Global configuration struct

Pointwise absolute error bound named differently

Compressed Stream returned “implicity”

Configuration in function arguments

Uses 1’s for unused dimensions

Dimensions provided in opposite order

Modifies input sometimes

Compressors! Only 3 Compressors!
LibPressio Provides a Common Interface

- Common Abstractions for:
  - Loading compressors
  - Configuration
  - Compression/Decompression
  - Representing Data
  - Error Reporting
  - Computing Metrics

```c
// get the compressor
struct pressio* library = pressio_instance();
struct pressio_compressor* sz = pressio_get_compressor(library,
    "sz");

// configure, validate, and assign the options
struct pressio_options* config =
    pressio_compressor_get_options(sz);
pressio_options_set_integer(config, "sz:error_bound_mode", REL);
pressio_options_set_double(config, "sz:rel_err_bound", 0.01);
pressio_compressor_set_options(sz, config);

// read in an input buffer
size_t dims[] = {500, 500, 100};
struct pressio_data* description =
    pressio_data_new_empty(pressio_float_dtype, 3, dims);
struct pressio_data* input_data =
    pressio_io_data_path_read(description, "CLOUDf48.bin.f32");

// create output buffers
struct pressio_data* compressed_data =
    pressio_data_new_empty(pressio_byte_dtype, 0, NULL);
struct pressio_data* decompressed_data =
    pressio_data_new_owning(pressio_float_dtype, 3, dims);

// compress and decompress the data
pressio_compressor_compress(sz, input_data, compressed_data);
pressio_compressor_decompress(sz, decompressed_data,
                      decompressed_data);
```
## What can we do generically?

<table>
<thead>
<tr>
<th>Task</th>
<th>Compressors</th>
<th>Lines LibPressio</th>
<th>Lines Native</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADIOS2</td>
<td>3</td>
<td>744</td>
<td>367</td>
<td>50.67%</td>
</tr>
<tr>
<td>Binding: Julia</td>
<td>1</td>
<td>299</td>
<td>25</td>
<td>91.64%</td>
</tr>
<tr>
<td>Binding: Python</td>
<td>2</td>
<td>768</td>
<td>363</td>
<td>52.73%</td>
</tr>
<tr>
<td>CLI</td>
<td>3</td>
<td>1649</td>
<td>756</td>
<td>54.15%</td>
</tr>
<tr>
<td>Configuration Optimizer</td>
<td>1</td>
<td>4683</td>
<td>1869</td>
<td>60.09%</td>
</tr>
<tr>
<td>HDF5 Filter</td>
<td>2</td>
<td>1469</td>
<td>438</td>
<td>70.18%</td>
</tr>
<tr>
<td>Z-Checker</td>
<td>7</td>
<td>3052</td>
<td>2647</td>
<td>86.73%</td>
</tr>
<tr>
<td>Parallel Compressor</td>
<td>2</td>
<td>2733</td>
<td>946</td>
<td>65.38%</td>
</tr>
</tbody>
</table>
## What can we do generically?

<table>
<thead>
<tr>
<th>Task</th>
<th>Compressors</th>
<th>Lines LibPressio</th>
<th>Lines Native</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADIOS2</td>
<td>3</td>
<td>744</td>
<td>367</td>
<td>50.67%</td>
</tr>
<tr>
<td>Binding: Julia</td>
<td>1</td>
<td>299</td>
<td>25</td>
<td>91.64%</td>
</tr>
<tr>
<td>Binding: Python</td>
<td>2</td>
<td>768</td>
<td>363</td>
<td>52.73%</td>
</tr>
<tr>
<td>CLI</td>
<td>3</td>
<td>1649</td>
<td>756</td>
<td>54.15%</td>
</tr>
<tr>
<td>Configuration Optimizer</td>
<td>1</td>
<td>4683</td>
<td>1869</td>
<td>60.09%</td>
</tr>
<tr>
<td>HDF5 Filter</td>
<td>2</td>
<td>1469</td>
<td>438</td>
<td>70.18%</td>
</tr>
<tr>
<td>Z-Checker</td>
<td>7</td>
<td>3052</td>
<td>2647</td>
<td>86.73%</td>
</tr>
<tr>
<td>Parallel Compressor</td>
<td>2</td>
<td>2733</td>
<td>946</td>
<td>65.38%</td>
</tr>
</tbody>
</table>
Configuration Search Why is this hard?

- Why not use binary search?
  - It doesn’t work consistently
  - The relationship between error bounds and metrics is often not monotonic
- Why not use analytic based methods/specialized compressors
  - Expensive to develop
  - Can we get sufficient results cheaper?
## Users Care About Scientific Results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Specialized Compressors</th>
<th>MGARD-QOI</th>
<th>FRaZ</th>
<th>OptZConfig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression Ratio, Compression Bandwidth</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Peak Signal to Noise Ratio (PSNR)</td>
<td>✓</td>
<td>✗*</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>$L_{\infty}$ norm</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Change in Weighted Mean (≈Bounded Linear Functional)</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Pearson' Coefficient ($R^2$)</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>P value for the Kolmogorov–Smirnov</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Spatial Relative Error</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

*While PSNR is not a Bounded Linear Functional (supported by MGARD-QOI), but can be bounded by MGARD $L_2$ norm (i.e. $s=0$)*
OptZConfig: Approach

Formulate compressor configuration as an optimization problem

\[
\max_{\tilde{c} \in U} \ Q(d_{f,t}, \tilde{d}_{f,t}(\tilde{c}; \theta_c); \theta_m)
\]

User Error Bound
Data for a Field and Timestep
Compressor Fixed-Parameters
Non-fixed Compressor Settings
Compression Function
Fixed Metric Parameters
Feasible Compressor Settings
OptZConfig VS. Specialized Compressors

• Not faster when tuning
• But allows quality and size tradeoffs that may beat specialized compressors
  • As high as 320% improvement in compression ratio
• While maintaining target quality
OptZConfig VS. FRaZ

- Relative to FRaZ
  - Upto 56x combined speedup
    - 3-15x from interiteration termination
    - \( \approx 2x \) from compressor threading
    - \( \approx 2x \) from search threading

- New Features
  - Supports user-defined objectives
  - Supports multiple input parameters
  - Extendable Search methods

<table>
<thead>
<tr>
<th>target</th>
<th>Speedup (1 ( \cdots ) 4 search threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.88x</td>
</tr>
<tr>
<td>50</td>
<td>1.97x</td>
</tr>
<tr>
<td>60</td>
<td>2.24x</td>
</tr>
</tbody>
</table>
OptZConfig VS. MGARD QOI

- Relative to MGARD-QOI mode
  - OptZConfig+SZ is much faster for one-off tasks
  - Even if precomputation is not required, 75% of the time it can still be faster
  - If OptZConfig is tuned, it is 1000x faster than MGARD-QOI tuned
OptZConfig: Results

- Can we extend our prior work (FRaZ) to bound simple user metrics and improve performance?
  - Yes, 56x performance improvement in tuning time over FRaZ
  - What are the advantages of differing methods to bound user methods?

<table>
<thead>
<tr>
<th>Method</th>
<th>Strength</th>
<th>Weakness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialized Compressors</td>
<td>● Fastest</td>
<td>● Often has over-preservation ● High Implementation costs</td>
</tr>
<tr>
<td>MGARD-QOI</td>
<td>● Scaling factor is valid for all data ● More general than specialized methods</td>
<td>● Scaling factor is expensive ● Restrictive assumptions</td>
</tr>
<tr>
<td>BlackBox Tuning (FRaZ/OptZConfig)</td>
<td>● Fast ● Most General</td>
<td>● Requires tuning</td>
</tr>
</tbody>
</table>
Auto-configuration Collaboration Opportunities

• How can we further drive down the costs of black box compression techniques to enable their use by a wider community?
  • faster search techniques that use fewer invocations of the compressors
  • proxy-model based approaches
  • techniques to use fewer tunings, etc...
• Ideal collaborators have experience
  • in numerical optimization/black-box optimization techniques
  • transfer learning inspired optimization approaches or a related subject
  • have interest in apply this to a inter-disciplinary problem of bounding user metrics.
Automatic Parallelization

• Many compressors aren’t parallel on the CPU* leaving performance opportunities
  • SZ CPU parallel implementation only supports 3d blocks of floats
  • ZFP doesn’t parallelize decompression
  • MGARD doesn’t have parallel compression or decompression at all

Can we solve this generically with LibPressio?

* GPU has a similar story but requires different approaches, we focus on CPU for now
Why is Automatic Parallelization hard?

- Naive solutions can make unneeded copies
  - Differing preconditions
  - ~10% performance penalty
- Up to August, parallel code that works with ZFP would crash SZ!
  - SZ wasn’t thread-safe
  - LibPressio provides metadata to catch this
- Will it provide enough benefit?
Automated Parallelization
Automated Parallelization

Highly optimized parallel chunking supporting n-d chunks; Contact me later for details, if you are interested
LibPressio and Automatic Parallelization

- **Compression**
  - Slightly slower than a hand tuned version for SZ and ZFP
  - Works for MGARD out of the box
    - doesn’t support 49 chunks on this dataset
  - Chunksize and threads requires tuning:
    - On this dataset 14 chunks and threads is idea
    - ~3-7% compression ratio penalty at this chunksize
LibPressio and Automatic Parallelization

• Decompression
  • native ZFP decompression is not parallelized
  • slightly slower than native, and works for ZFP, MGARD, et. al.
Open Questions in Compression Interfaces

• What are the remaining gaps in capabilities of compression interfaces for applications? I.e. better support for:
  • Asynchrony
  • Accelerators
  • streaming compression
  • Different data structures (i.e. point cloud, and/or sparse compression, etc...).

• Ideal collaborators have applications that are considering/using compression, but would like to consider ways to improve performance or evaluate different compression methodologies.
Recent Improvements in Generic Compression

Robert Underwood
robertu@g.clemson.edu
Clemson University
December 16th, 2021

My Curriculum Vita
Thank You!
Funding Acknowledgements

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations - the Office of Science and the National Nuclear Security Administration, Responsible for the planning and preparation of a capable Exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation’s Exascale computing imperative.

The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant No. 1619253 and 1910197.

We acknowledge the computing resources provided on Bebop, which is operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664. All opinions expressed in this paper are the authors and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE.