FZ Hands-On Handout

Martin Burtscher, Franck Cappello, Sheng Di, Hanqi Guo, Samuel Li, Xin Liang,
Peter Lindstrom, Dingwen Tao, Kento Sato, Seung Woo Son, Robert Underwood, Kai Zhao *

February 14, 2024

*names listed alphabetically

Introduction

Welcome to the FZ Developer's meeting. During the hands-on portion, we hope that you will explore the compressors that are most interesting to you given the overview we presented. Each section of the tutorial is intended to be self-contained, so the order in which you visit the sections shouldn't matter. The developers of these compressors will be walking arround to answer questions.
You have two options to work on the hands-on: via SSH (preferred) or on your own laptop via Docker

Connecting to the Machines

We have provided a remote system that you can log into via SSH to run the hands on. This is the reccomended way to access the tutorial. The password is `CUT LENT ROVE CAL LEFT CUBA GLUE OMEN EEL BEAR GAL`

```
ssh -p 80 exouser@10.8.0.28
```

Using Your Laptop/Supercomputer

We have a pre-built contianer that runs on modern versions of Linux, MacOS, and Windows. This runs best on amd64/x86_64 processors (e.g. Intel/AMD processors), but it is possible run on arm64/aarch64/m series silicon as well. Please note that you may not be able to run all compressors in this way if you lack access to the appropriate hardware.
While the commands listed here are for podman, equivelent commands exist for Singularity, Apptainer, and Docker.

```
#Linux
sudo apt install podman #Debian/Ubuntu
sudo dnf install podman #RHEL/Fedora (probably installed)

#Windows
choco.exe install podman
podman machine init
podman machine start

#MacOS
brew install podman qemu
podman machine init
podman machine start

# all platforms
podman run -it --rm --platform linux/amd64  ghcr.io/robertu94/sz-zfp-zchecker:latest
```

Installation via Spack/Conda

We maintain spack and in many cases conda packages for this software as well. Please ask us about this if you need this for your environment.

Contents

1 SZ2/SZ3

contact: Kai Zhao, Sheng Di

1.1 CLI Interface (SZ2&SZ3)

```
# print help for the sz command
sz

# compress a 3D array in sz, with an absolute error bound of 1e-3
# -f : FP32
# -M ABS -A 1E-3: absolute error bound of 1e-3
# -i source_file : input
# -z compressed_file : output
# -3 500 500 100: array[100][500][500], reverse of C sequence
sz -f -M ABS -A 1E-3 -i /usecases/Hurricane/Uf48.dat -z  \
    /usecases/Hurricane/Uf48.dat.sz -3 500 500 100

# decompress the above data in sz, and print various statistics
# -i source_file : input, used for statistics
# -s compressed_file : input
# -x decompressed_file : output
# -3 500 500 100: array[100][500][500], reverse of C sequence
# -a : print statistics
sz -f  -i /usecases/Hurricane/Uf48.dat -s /usecases/Hurricane/Uf48.dat.sz -x \
    /usecases/Hurricane/Uf48.dat.sz.out -3 500 500 100  -a

# the command line of sz3 is backward compatible with sz2
# you can replace 'sz' with 'sz3' in the above lines to use sz3
```

1.2 C/C++ Interface (SZ3)

```
#include "SZ3/api/sz.hpp"
#compress
SZ3::Config conf(100, 500, 500); // array[100][500][500]
conf.cmprAlgo = SZ3::ALGO_INTERP_LORENZO;
conf.errorBoundMode = SZ3::EB_ABS; // refer to def.hpp for all supported error bound mode
conf.absErrorBound = 1E-3; // absolute error bound 1e-3
size_t cmpSize = 100 * 500 * 500; // cmpSize is overwritten by compressed data size
/**
 * @tparam T source data type
 * @param config compression configuration
 * @param data source data
 * @param cmpSize compressed data size in bytes
 * @return compressed data, remember to 'delete []' when the data is no longer needed.
 */
char *cmpData = SZ_compress<float>(conf, data, cmpSize);

#decompress
auto decompressedData = new float[100*500*500];
SZ3::Config conf1;
/**
 * @param conf configuration placeholder. Overwritten by the compression configuration
 * @param cmpData compressed data
```

```
 * @param cmpSize compressed data size in bytes
 * @param decData pre-allocated memory space for decompressed data
 */
SZ_decompress<float>(conf1, cmpData, cmpSize, decData);

```

1.3 Python Interface (SZ3)

```
#python support is experimental
#you need to locate 1) sz3c lib file and 2) pysz.py
#sz3c lib can be found in your_cmake_folder/tools/sz3c/
#pysz.py can be found in sz3/tools/pysz/

import numpy as np
from pathlib import Path
from pysz import SZ
import sys

# prepare your data in numpy array format
data = np.fromfile('Uf48.bin.dat', dtype=np.float32)
data = np.reshape(data, (100, 500, 500))


lib_extention = {
    "darwin": "libSZ3c.dylib",
    "windows": "SZ3c.dll",
}.get(sys.platform, "libSZ3c.so")
# Please change the path to the SZ dynamic library file in your system
sz = SZ("/path/to/your/sz/lib/folder".format(lib_extention))

"""
Compress data with SZ
:param data: original data, numpy array format, dtype is FP32 or FP64
:param eb_mode:# error bound mode, integer (0: ABS, 1:REL)
:param eb_abs: optional, abs error bound, double
:param eb_rel: optional, rel error bound, double
:param eb_pwr: optional, pwr error bound, double
:return: compressed data, numpy array format, dtype is np.uint8
         compression ratio
"""
data_cmpr, cmpr_ratio = sz.compress(data, 0, 1e-3, 0, 0)
print("compression ratio = {:5G}".format(cmpr_ratio))

# decompress, both input and output data are numpy array
data_dec = sz.decompress(data_cmpr, data.shape, data.dtype)

# verify
sz.verify(data, data_dec)
```

2 cuSZ

For this section you will need an NVIDIA GPU.

contact: Jiannan Tian
cuSZ is a GPU adoption of SZ. cuSZ incorporates several practical components, including fully parallelized prediction & error quantization stage and coarsely parallelized Huffman encoding (the released version).

2.1 CLI Interface

1. The interpretation of data shape is important to utilize the dimensional information. cuSZ puts two ways to specify data shape,
   - fastest to slowest (inner to outer loop if sequential, also CUDA `dim3`)
     `--dim3 [X]x[Y]x[Z]` or `--len [X]x[Y]x[Z]`
   - slowest to fastest (outer to inner loop if sequential)
     `--size [Z]x[Y]x[X]`
2. Two predictors are supported:
   - Lorenzo predictor of higher throughput
     `--predictor lorenzo`
   - cubic spline interpolation of higher quality (3D only)
     `--predictor spline`
3. Two modes are supported: `abs` and `r2r` (relative to data range).

```
# print help for the cusz command
cusz

# print even longer help for the cusz command
cusz -h

# Currently, only 32-bit fp-type is supported.
# To compress, -z is needed.
# By default, Lorenzo predictor is used.
# To use 3D cubic spline interpolation,
# --predictor spline should be appended in the command line.
cuszi -t f32 -m rel -e [ErrorBound] -i [/PATH/TO/DATA] --dim3 [X]x[Y]x[Z] -z --report time

# To decompress, -x is needed
# The optional --compare [/PATH/TO/DATA] can be specified to
# compared the original and reconstructed data.
cuszi -i [/PATH/TO/DATA].cusza -x --report time --compare [/PATH/TO/DATA]
```

2.2 C/C++ Interface

cuSZ uses namespace `psz`, which stands for GPU-**p**arallelized **SZ**.

```
// header zone
#include "cusz.h"
#include "context.h"

pszheader header;   // declare metadata

uint8_t* ptr_compressed;    // to access internal compressor buffer
size_t compressed_len;      // to be changed by compressor

T *d_uncomp, *h_uncomp;
T *d_decomp, *h_decomp;

auto oribytes = sizeof(T) * len;
cudaMalloc(&d_uncomp, oribytes), cudaMallocHost(&h_uncomp, oribytes);
cudaMalloc(&d_decomp, oribytes), cudaMallocHost(&h_decomp, oribytes);

// User handles loading from filesystem & transferring to device.
// io::read_binary_to_array(fname, h_uncomp, len);
// cudaMemcpy(d_uncomp, h_uncomp, oribytes, cudaMemcpyHostToDevice);

// initialize CUDA stream
cudaStream_t stream; cudaStreamCreate(&stream);

// use default setup for cuSZ framework
pszframe* work = pszdefault_framework();

// alternative finer-grained setup
// pszframe* work = new pszframe{
//     .predictor = pszpredictor{.type = Lorenzo},
//     .quantizer = pszquantizer{.radius = 512},
//     .hfcoder = pszhfrc{.style = Coarse},
//     .max_outlier_percent = 20};

pszcompressor* comp = psz_create(work, F4);

// specify compression mode: Rel or Abs
// specify error bound: any fractional number
pszctx* ctx = new pszctx{.mode = Rel, .eb = 2.4e-4};
// x, y, z, w (w is a placeholder for future use)
pszlen uncomp_len = pszlen{3600, 1800, 1, 1};
pszlen decomp_len = uncomp_len;

// C++ time-record type
psz::TimeRecord compress_timerecord;
psz::TimeRecord decompress_timerecord;

{ // compression scope
psz_compress_init(comp, uncomp_len, ctx);
psz_compress(comp, d_uncomp, uncomp_len, &ptr_compressed, &compressed_len, &header,
    (void*)&compress_timerecord, stream);

// User can interpret the collected time information in other ways.
psz::TimeRecordViewer::view_compression(
    &compress_timerecord, oribytes, compressed_len);
```

```
55
56  // verify header
57  printf("header.%-*s : %p\n", 12, "(addr)", (void*)&header);
58  printf("header.%-*s : %u, %u, %u\n", 12, "{x,y,z}", header.x, header.y,
59      header.z);
60  printf("header.%-*s : %lu\n", 12, "filesize", psz_utils::filesize(&header));
61  } // end of compression scope
62
63  // If needed, User should perform a memcopy to transfer `ptr_compressed`
64  // before `compressor` is destroyed.
65  uint8_t* compressed_buf;
66  cudaMalloc(&compressed_buf, compressed_len);
67  cudaMemcpy(compressed_buf, ptr_compressed, compressed_len, cudaMemcpyDeviceToDevice);
68
69  { // decompression scope
70  psz_decompress_init(comp, &header);
71  psz_decompress(
72      comp, ptr_compressed, compressed_len, d_decomp, decomp_len,
73      (void*)&decompress_timerecord, stream);
74  // print kernel time records
75  psz::TimeRecordViewer::view_decompression(
76      &decompress_timerecord, oribytes);
77  } // end of decompression scope
78
79  // demo: offline checking (de)compression quality.
80  psz::eval_dataquality_gpu(d_decomp, d_uncomp, len, compressed_len);
81
82  psz_release(comp);
83
84  cudaFree(compressed_buf);
85  cudaFree(d_uncomp), cudaFreeHost(h_uncomp);
86  cudaFree(d_decomp), cudaFreeHost(h_decomp);
87
88  cudaStreamDestroy(stream);
```

3 ZFP

contact: Peter Lindstrom
ZFP is a compressed format for representing multidimensional floating-point and integer arrays. ZFP provides compressed-array classes that support high throughput read and write random access to individual array elements. ZFP also supports serial and parallel compression of whole arrays, e.g., for applications that read and write large data sets to and from disk. For documentation, source code, and other resources, see https://zfp.llnl.gov.

3.1 CLI Interface

```
# print help for the zfp command
zfp

# load a 3d array of floats, use zfp with an absolute error bound of 1e-4,
# and collect and print various statistics
zfp -s -i /usecases/Hurricane/Uf48.dat -f -3 500 500 100 -a 1e-4

# use zfp's fixed-rate mode to compress to 13 bits/value
zfp -s -i /usecases/Hurricane/Uf48.dat -f -3 500 500 100 -r 13

# use zfp's fixed-precision mode with 10 bits of precision
zfp -s -i /usecases/Hurricane/Uf48.dat -f -3 500 500 100 -p 10

# use zfp's reversible (lossless) mode to compress to stdout, decompress from stdin,
# and encode the array size and compression mode in a header
zfp -h -i /usecases/Hurricane/Uf48.dat -f -3 500 500 100 -R -z - | \
zfp -h -z - -o /tmp/Uf48.out
```

3.2 C Interface

```
#include "zfp.h"

// input: (double* array, size_t nx, size_t ny, size_t nz, double tolerance)

// initialize metadata for the 3D array a[nz][ny][nx]
zfp_type type = zfp_type_double;                          // array scalar type
zfp_field* field = zfp_field_3d(array, type, nx, ny, nz); // array metadata

// initialize metadata for a compressed stream
zfp_stream* zfp = zfp_stream_open(NULL);                  // compressed stream and parameters
zfp_stream_set_accuracy(zfp, tolerance);                  // set tolerance for fixed-accuracy mode
//  zfp_stream_set_precision(zfp, precision);             // alternative: fixed-precision mode
//  zfp_stream_set_rate(zfp, rate, type, 3, zfp_false);   // alternative: fixed-rate mode
//  zfp_stream_set_reversible(zfp);                       // alternative: lossless mode

// allocate buffer for compressed data
size_t bufsize = zfp_stream_maximum_size(zfp, field);     // capacity of compressed buffer
void* buffer = malloc(bufsize);                           // storage for compressed stream

// associate bit stream with allocated buffer
bitstream* stream = stream_open(buffer, bufsize);         // bit stream to compress to
zfp_stream_set_bit_stream(zfp, stream);                   // associate with compressed stream

```

```
// compress array
zfp_stream_rewind(zfp);                                 // rewind stream to beginning
size_t zfpsize = zfp_compress(zfp, field);              // return value is compressed byte size

// decompress array
zfp_stream_rewind(zfp);                                 // rewind stream for decompression
assert(zfp_decompress(zfp, field) == zfpsize);          // decompress stream
```

3.3 Python Interface

Install ZFP's Python bindings using `pip3 install --user zfpy`.

```
import zfpy
import numpy as np

# initialize 100 * 80 NumPy array to Hilbert matrix
my_array = np.array([1/(1+x+y) for y in range(100) for x in range(80)]).reshape((100, 80))

# compress array to within an absolute error tolerance of 1e-6
compressed_data = zfpy.compress_numpy(my_array, tolerance=1e-6)

# decompress to another NumPy array
decompressed_array = zfpy.decompress_numpy(compressed_data)

# verify that error tolerance is satisfied
np.testing.assert_allclose(my_array, decompressed_array, atol=1e-6)

# compression ratio
decompressed_array.nbytes / len(compressed_data)
```

3.4 C++ Interface

```
#include <climits>
#include <cmath>
#include <iostream>
#include "zfp/array2.hpp"
#include "zfp/codec/gencodec.hpp"

double f(size_t x, size_t y) { return 1. / (1 + x + y); }

template <class Array>
void evaluate(Array& a, std::string name)
{
  // initialize array to Hilbert matrix
  for (size_t y = 0; y < a.size_y(); y++)
    for (size_t x = 0; x < a.size_x(); x++)
      a(x, y) = f(x, y);

  // optional: compress any cached data
  a.flush_cache();

  // compute error
  double abserr = 0;
  double relerr = 0;
  for (size_t y = 0; y < a.size_y(); y++)
```

```
    for (size_t x = 0; x < a.size_x(); x++) {
      double d = std::abs(a(x, y) - f(x, y));
      abserr += d;
      relerr += d / std::abs(f(x, y));
    }
  abserr /= a.size();
  relerr /= a.size();

  // output error
  std::cout << std::defaultfloat << a.rate() << "-bit " << name << " array" << std::endl;
  std::cout << "  mean absolute error = " << std::scientific << abserr << std::endl;
  std::cout << "  mean relative error = " << std::scientific << relerr << std::endl;
}

int main()
{
  const size_t nx = 60;  // array width
  const size_t ny = 100; // array height

  // scalar type for uncompressed array
  typedef float real;  // single precision
//typedef __fp16 real; // alternative: half precision
  const double rate = sizeof(real) * CHAR_BIT; // storage size in bits/value

  // uncompressed 2D array storing scalars of type real
  zfp::array2< double, zfp::codec::generic2<double, real> > a(nx, ny, rate);
  evaluate(a, "uncompressed");

  // zfp compressed 2D array with equal storage size
  zfp::array2<double> b(nx, ny, rate);
  evaluate(b, "zfp");

  return 0;
}
```

4 MGARD

Contact: Xin Liang, Jieyang Chen
MGARD is a highly functional, performant, portable, and extendable compressor for scientific data. Built upon wavelet and finite element theories, MGARD features rigorous error controls on both raw data and derived quantities, diverse functionalities including both compression and refactoring, utility support for both uniform/non-uniform structured data and unstructured data, and unified interfaces on CPU and GPUs. For detailed documentation, publication, and source code, please refer to https://github.com/CODARcode/MGARD.

4.1 CLI Interface

```
# print help for the mgard-x command
mgard-x

Options
    -z: compress data
        -i <path to data file to be compressed>
        -c <path to compressed file>
        -t <s|d>: data type (s: single; d:double)
        -n <ndim>: total number of dimensions
           [dim1]: slowest dimension
           [dim2]: 2nd slowest dimension
           ...
           [dimN]: fastest dimension
        -u <path to coordinate file>
        -m <abs|rel>: error bound mode (abs: absolute; rel: relative)
        -e <error>: error bound
        -s <smoothness>: smoothness parameter
        -l choose lossless compressor (0:Huffman 1:Huffman+LZ4 2:Huffman+Zstd)
        -d <auto|serial|openmp|cuda|hip|sycl>: device type
        -v enable verbose (show timing and statistics)

    -x: decompress data
        -c <path to compressed file>
        -o <path to decompressed file>
        -d <auto|serial|cuda|hip>: device type
        -v enable verbose (show timing and statistics)

# Example: compress a 3D array in MGARD with an absolute error bound of 1e-3
mgard-x -z -i /usecases/Hurricane/Uf48.dat -c /usecases/Hurricane/Uf48.dat.mgard -t s -n 3 100 500 500
    -s 0 -l 2 -d auto -m abs -e 1e-3

# Example: decompress the 3D array
mgard-x -x -c /usecases/Hurricane/Uf48.dat.mgard -o /usecases/Hurricane/Uf48.dat.mgard.out -d auto

# Example: refactor a 3D array
mdr-x -z -i /usecases/Hurricane/Uf48.dat -c refactored.mgard -t s -n 3 100 500 500 -l 2 -h 1 -d auto

# Retrieve and reconstruct the data under absolute error bound 1e-3
mdr-x -x -c refactored.mgard -o /usecases/Hurricane/Uf48.dat.mgard.out -a /usecases/Hurricane/Uf48.dat
```

4.2 C/C++ Interface

MGARD uses namespace `mgard_x` with a sub-namespace `MDR` for refactoring APIs.

High-level APIs for error-controlled compression and decompression:

```
#include "mgard/compress_x.hpp"

// prepare data buffers
mgard_x::DIM num_dims = 3;
mgard_x::SIZE n1, n2, n3;
std::vector<mgard_x::SIZE> shape{n1, n2, n3};
mgard_x::SIZE in_byte = n1 * n2 * n3 * sizeof(double);
mgard_x::SIZE out_byte;
//... load data into in_array
double *in_array = ...;
void *compressed_array = NULL;
void *decompressed_array = NULL;
// tol: error tolerance
// s: smoothness parameter
double tol = 0.01, s = 0;

// MGARD config parameters
mgard_x::Config config;

// Compressing with high-level API
mgard_x::compress(num_dims, mgard_x::data_type::Double, shape, tol, s, mgard_x::error_bound_type::REL,

// Decompressing with high level API
mgard_x::decompress(compressed_array, out_byte, decompressed_array, config, false);
```

High-level APIs for refactoring and error-controlled progressive retrieval:

```
#include "mgard/mdr_x.hpp"
...

// prepare data buffers
mgard_x::DIM num_dims = 3;
mgard_x::SIZE n1, n2, n3;
std::vector<mgard_x::SIZE> shape{n1, n2, n3};
mgard_x::SIZE in_byte = n1 * n2 * n3 * sizeof(double);
mgard_x::SIZE out_byte;
//... load data into in_array
double *in_array = ...;

mgard_x::Config config;
mgard_x::MDR::RefactoredMetadata refactored_metadata;
mgard_x::MDR::RefactoredData refactored_data;

// Refactor with high-level API
mgard_x::MDR::MDRefactor(D, mgard_x::data_type::Double, shape, in_array, refactored_metadata,
    refactored_data, config, false);

// Save refactored_metadata and refactored_data to files
```

```
...

mgard_x::MDR::ReconstructedData reconstructed_data;
// Read in refactored_metadata from file
...
// Progressively reconstruct for each error bound
for (double tol : tolerances) {
    // Specify error bound and smoothness parameter for each subdomain
    for (auto &metadata : refactored_metadata.metadata) {
          metadata.requested_tol = tol;
          metadata.requested_s = s;
    }
    // Determine required data components for reconstruction
    mgard_x::MDR::MDRequest(refactored_metadata, config);
    // Read in required data components from files
    ...
    // Reconstruct with high-level API
    mgard_x::MDR::MDReconstruct(refactored_metadata, refactored_data, reconstructed_data,
        config, false, original_data);

    // reconstructed_data now contains progressively reconstructed data
    double out_data = reconstructed_data.data;
}
```

5 SPERR

Contact: Samuel Li (shaomeng@ucar.edu)
Repo: github.com/NCAR/SPERR/
Wiki: github.com/NCAR/SPERR/wiki

SPERR uses *wavelet transforms* to decorrelate the data, encodes the quantized coefficients, and explicitly corrects any data point exceeding a prescribed point-wise error (PWE) tolerance. Most often, SPERR produces the smallest compressed bitstream honoring a PWE tolerance.
A SPERR bitstream can be used to reconstruct the data fields in two additional fashions: *flexible-rate* decoding and *multi-resolution* decoding.

- *Flexible-rate* decoding: any prefix of a SPERR bitstream (i.e., a sub-bitstream that starts from the very beginning) produced by a simple truncation is still valid to reconstruct the data field, though at a lower quality. This ability is useful for applications such as tiered storage and data sharing over slow connections, to name a few.
- *Multi-resolution* decoding: a hierarchy of the data field with coarsened resolutions can be obtained together with the native resolution. This ability is useful for quick analyses with limited resources.

On a Unix-like system with a working C++ compiler and CMake, SPERR can be built from source and made available to users in just a few commands. See this README for detail.

5.1 CLI Interface

Upon a successful build, four CLI utility programs are placed in the `./bin/` directory; three of them are most relevant here: `sperr2d`, `sperr3d`, and `sperr3d_trunc`. Each of them can be invoked with the `-h` option to display a help message.

5.1.1 `sperr2d`

`sperr2d` is responsible for compressing and decompressing a 2D data slice. Its help message contains all the options `sperr2d` takes:

```
$ ./bin/sperr2d -h

Usage: ./bin/sperr2d [OPTIONS] [filename]

Positionals:
  filename TEXT:FILE          A data slice to be compressed, or
                              a bitstream to be decompressed.

Options:
  -h,--help                   Print this help message and exit

Execution settings:
  -c Excludes: -d             Perform a compression task.
  -d Excludes: -c             Perform a decompression task.

Input properties:
  --ftype UINT                Specify the input float type in bits. Must be 32 or 64.
  --dims [UINT,UINT]          Dimensions of the input slice. E.g., `--dims 128 128`
                              (The fastest-varying dimension appears first.)

Output settings:
  --bitstream TEXT            Output compressed bitstream.
```

```
  --decomp_f TEXT             Output decompressed slice in f32 precision.
  --decomp_d TEXT             Output decompressed slice in f64 precision.
  --decomp_lowres_f TEXT      Output lower resolutions of the decompressed slice in f32 precision.
  --decomp_lowres_d TEXT      Output lower resolutions of the decompressed slice in f64 precision.
  --print_stats Needs: -c     Show statistics measuring the compression quality.

Compression settings:
  --pwe FLOAT Excludes: --psnr --bpp
                              Maximum point-wise error (PWE) tolerance.
  --psnr FLOAT Excludes: --pwe --bpp
                              Target PSNR to achieve.
  --bpp FLOAT:FLOAT in [0 - 64] Excludes: --pwe --psnr
                              Target bit-per-pixel (bpp) to achieve.
```

Examples:

1. Compress a 2D slice in 512×512 dimension, single-precision floats with a PWE tolerance of 10^{-2}:
   `./bin/sperr2d -c --ftype 32 --dims 512 512 --pwe 1e-2 \`
   `--bitstream ./out.stream ./in.f32`

2. Perform the compression task described above, plus also write out the compress-decompressed slice, and finally print statistics measuring the compression quality:
   `./bin/sperr2d -c --ftype 32 --dims 512 512 --pwe 1e-2 \`
   `--decomp_f ./out.decomp --print_stats --bitstream ./out.stream ./in.f32`

3. Decompress from a SPERR bitstream, and write out the slice in native and coarsened resolutions:
   `./bin/sperr2d -d --decomp_f ./out.decomp --decomp_lowres_f ./out.lowres ./sperr.stream`
   In this example, the output file `out.decomp` will be of the native resolution, and six other files (`out.lowres.256x256, out.lowres.128x128, etc.`) will also be produced with their filenames indicating the coarsened resolution.

5.1.2 sperr3d

`sperr3d` is responsible for compressing and decompressing a 2D data volume. Compared to `sperr2d` which compresses the input 2D slice as a whole, `sperr3d` divides an input 3D volume into smaller chunks, and then compresses each chunk individually. This treatment allows for compressing and decompressing all the small chunks in parallel. `sperr3d` uses 256^3 as the default chunk dimension, but any number in the order of low hundreds and preferably divides the full volume is a good number. Command line options `--chunks` and `--omp` control the chunking and parallel execution behavior respectivelly.
The help message of `sperr3d` details all the options it takes:

```
$ ./bin/sperr3d -h

Usage: ./bin/sperr3d [OPTIONS] [filename]

Positionals:
  filename TEXT:FILE          A data volume to be compressed, or
                              a bitstream to be decompressed.

Options:
  -h,--help                   Print this help message and exit

Execution settings:
  -c Excludes: -d             Perform a compression task.
  -d Excludes: -c             Perform a decompression task.
  --omp UINT                  Number of OpenMP threads to use. Default (or 0) to use all.
```

```
Input properties (for compression):
  --ftype UINT                Specify the input float type in bits. Must be 32 or 64.
  --dims [UINT,UINT,UINT]     Dimensions of the input volume. E.g., `--dims 128 128 128`
                              (The fastest-varying dimension appears first.)

Output settings:
  --bitstream TEXT            Output compressed bitstream.
  --decomp_f TEXT             Output decompressed volume in f32 precision.
  --decomp_d TEXT             Output decompressed volume in f64 precision.
  --decomp_lowres_f TEXT      Output lower resolutions of the decompressed volume in f32 precision.
  --decomp_lowres_d TEXT      Output lower resolutions of the decompressed volume in f64 precision.
  --print_stats Needs: -c     Print statistics measuring the compression quality.

Compression settings:
  --chunks [UINT,UINT,UINT]   Dimensions of the preferred chunk size. Default: 256 256 256
                              (Volume dims don't need to be divisible by these chunk dims.)
  --pwe FLOAT Excludes: --psnr --bpp
                              Maximum point-wise error (PWE) tolerance.
  --psnr FLOAT Excludes: --pwe --bpp
                              Target PSNR to achieve.
  --bpp FLOAT:FLOAT in [0 - 64] Excludes: --pwe --psnr
                              Target bit-per-pixel (bpp) to achieve.
```

Examples:

1. Compress a 3D volume in $384 \times 384 \times 256$ dimension, double-precision floats, using a PWE tolerance of 10^{-9} and chunks of $192 \times 192 \times 256$:
   `./bin/sperr3d -c --omp 4 --ftype 64 --dims 384 384 256 --chunks 192 192 256 \`
   `--pwe 1e-9 --bitstream ./out.stream ./in.f64`

2. Perform the compression task described above, plus also write out the compress-decompressed volume, and finally print statistics measuring the compression quality:
   `./bin/sperr3d -c --omp 4 --ftype 64 --dims 384 384 256 --chunks 192 192 256 \`
   `--pwe 1e-9 --decomp_d ./out.decomp --print_stats --bitstream ./out.stream ./in.f64`

3. Decompress from a SPERR bitstream, and write out the volume in native and coarsened resolutions:
   `./bin/sperr3d -d --decomp_d ./out.decomp --decomp_lowres_d ./out.lowres ./sperr.stream`
   In this example, the output file `out.decomp` will be of the native resolution, and five other files (`out.lowres.192x192x128, out.lowres.96x96x64, etc.`) will also be produced with their filenames indicating the coarsened resolution.

> To support multi-resolution decoding in 3D cases, the individual chunks (`--chunks`) need to 1) approximate a cube, so that there are the same number of wavelet transforms performed on each dimension, and 2) divide the full volume in each dimension.

5.1.3 `sperr3d_trunc`

`sperr3d_trunc` is responsible for properly truncating a multi-chunked 3D bitstream, i.e., locating the offset of each chunk in a bitstream, and truncating each chunk individually.
The help message of `sperr3d_trunc` details its options:

```
$ ./bin/sperr3d_trunc -h

Usage: ./bin/sperr3d_trunc [OPTIONS] [filename]
```

```
Positionals:
  filename TEXT:FILE          The original SPERR3D bitstream to be truncated.

Options:
  -h,--help                   Print this help message and exit

Truncation settings:
  --pct UINT REQUIRED         Percentage (1--100) of the original bitstream to truncate.
  --omp UINT                  Number of OpenMP threads to use. Default (or 0) to use all.

Output settings:
  -o TEXT                     Write out the truncated bitstream.

Input settings:
  --orig32 TEXT               Original raw data in 32-bit precision to calculate compression
                              quality using the truncated bitstream.
  --orig64 TEXT               Original raw data in 64-bit precision to calculate compression
                              quality using the truncated bitstream.
```

Examples:

1. Produce a truncated version of a bitstream using 10% of the original length:
   `./bin/sperr3d_trunc --pct 10 -o ./stream.10 ./bitstream`
2. Perform the task above, plus evaluate compression quality using the truncated bitstream:
   `./bin/sperr3d_trunc --pct 10 -o ./stream.10 --orig64 ./data.f64 ./bitstream`

> SPERR bitstreams without using multi-chunks (i.e., `--dims` equals `--chunks` in 3D, and all 2D cases) can safely be truncated by any tool (e.g., `head` on Linux), not necessarily using `sperr3d_trunc`.

5.2 C++ Interface

5.2.1 2D Compression and Decompression

C++ class `sperr::SPECK2D_FLT` is responsible for 2D compression and decompression. The sample code walks through necessary steps to perform a compression and decompression task, and a more concrete example can be found here.

```cpp
//
// Example of using a sperr::SPECK2D_FLT() to compress a 2D slice.
// This is a 6-step process.
//
#include "SPECK2D_FLT.h"

// Step 1: create an encoder:
auto encoder = sperr::SPECK2D_FLT();

// Step 2: specify the 2D slice dimension (the third dimension is left with 1):
encoder.set_dims({128, 128, 1});

// Step 3: copy over the input data from a raw pointer (float* or double*):
encoder.copy_data(ptr, 16'384);      // 16,384 is the number of values.
// Step 3 alternative: one can hand a memory buffer to the encoder to avoid a memory copy;
// use either version is cool.
```

```
encoder.take_data(std::move(input)); // input is of type std::vector<doubles>.

// Step 4: specify the compression quality measured in one of three metrics;
// only the last invoked quality metric is honored.
encoder.set_tolerance(1e-9);         // PWE tolerance = 1e-9
encoder.set_bitrate(2.2);            // Target bitrate = 2.2
encoder.set_psnr(102.2);             // Target PSNR = 102.2

// Step 5: perform the compression task:
encoder.compress();

// Step 6: retrieve the compressed bitstream:
auto bitstream = std::vector<uint8_t>();
encoder.append_encoded_bitstream(bitstream);
```

```
//
// Example of using a sperr::SPECK2D_FLT() to decompress a bitstream.
// This is a 5-step process.
//
#include "SPECK2D_FLT.h"

// Step 1: create a decoder:
auto decoder = sperr::SPECK2D_FLT();

// Step 2: specify the 2D slice dimension (the third dimension is left with 1):
// This information is often saved once somewhere for many same-sized slices.
decoder.set_dims({128, 128, 1});

// Step 3: pass in the compressed bitstream as a raw pointer (uint8_t*):
decoder.use_bitstream(ptr, 16'384);  // 16,384 is the length of the bitstream.

// Step 4: perform the decompression task:
decoder.decompress(multi_res);  // a boolean, if to enable multi-resolution decoding

// Step 5: retrieve the decompressed volume:
std::vector<double> vol = decoder.view_decoded_data();
auto hierarchy = decoder.view_hierarchy();     // if multi-resolution was enabled
// Step 5 alternative: one can take ownership of the data buffer to avoid a memory copy.
std::vector<double> vol = decoder.release_decoded_data();
auto hierarchy = decoder.release_hierarchy();  // if multi-resolution was enabled
```

5.2.2 3D Compression and Decompression

C++ class `sperr::SPERR3D_OMP_C` is responsible for 3D compression, and `sperr::SPERR3D_OMP_D` is responsible for 3D decompression. The sample code walks through necessary steps to perform a compression and decompression task, and a more concrete example can be found here.

```
//
// Example of using a sperr::SPERR3D_OMP_C() to compress a 3D volume.
// This is a 6-step process.
//
#include "SPERR3D_OMP_C.h"

// Step 1: create an encoder:
auto encoder = sperr::SPERR3D_OMP_C();
```

```
9
10  // Step 2: specify the volume and chunk dimensions, respectively:
11  encoder.set_dims_and chunks({384, 384, 256}, {192, 192, 128});
12
13  // Step 3: specify the number of OpenMP threads to use:
14  encoder.set_num_threads(4);
15
16  // Step 4: specify the compression quality measured in one of three metrics;
17  // only the last invoked quality metric is honored.
18  encoder.set_tolerance(1e-9);         // PWE tolerance = 1e-9
19  encoder.set_bitrate(2.2);            // Target bitrate = 2.2
20  encoder.set_psnr(102.2);             // Target PSNR = 102.2
21
22  // Step 5: perform the compression task:
23  // The input data is passed in in the form of a raw pointer (float* or double*),
24  // and the total number of values will be passed in here too.
25  encoder.compress(ptr, 384 * 384 * 256);
26
27  // Step 6: retrieve the compressed bitstream:
28  std::vector<uint8_t> stream = encoder.get_encoded_bitstream();
```

```
1   //
2   // Example of using a sperr::SPERR3D_OMP_D() to decompress a bitstream.
3   // This is a 5-step process.
4   //
5   #include "SPERR3D_OMP_D.h"
6
7   // Step 1: create a decoder:
8   auto decoder = sperr::SPERR3D_OMP_D();
9
10  // Step 2: specify the number of OpenMP threads to use:
11  decoder.set_num_threads(4);
12
13  // Step 3: pass in the compressed bitstream as a raw pointer (uint8_t*):
14  decoder.use_bitstream(ptr, 16'384);  // 16,384 is the length of the bitstream.
15
16  // Step 4: perform the decompression task:
17  // Note that the pointer to the bitstream is passed in again!
18  decoder.decompress(ptr, multi_res);  // a boolean, if to enable multi-resolution decoding
19
20  // Step 5: retrieve the decompressed volume:
21  auto [dimx, dimy, dimz] = decoder.get_dims();  // dimension of the volume
22  std::vector<double> vol = decoder.view_decoded_data();
23  auto hierarchy = decoder.view_hierarchy();     // if multi-resolution was enabled
24  // Step 5 alternative: one can take ownership of the data buffer to avoid memory copies.
25  std::vector<double> vol = decoder.release_decoded_data();
26  auto hierarchy = decoder.release_hierarchy();  // if multi-resolution was enabled
```

To achieve higher performance with repeated compression and decompression tasks, the encoder and decoder objects are better to be re-used rather than repeatedly destroyed and created.

5.3 C Interface

SPERR provides a C wrapper with a set of C functions. All of the C interface is in the header file SPERR_C_API.h, which itself documents the C functions and parameters, etc. The following example code walks through key steps to use the C API to perform compression and decompression, while more concrete examples are available for 2D and 3D cases.

5.3.1 Example: 2D

```c
/*
 * Example of using the SPERR C API to perform 2D compression and decompression tasks.
 */
#include "SPERR_C_API.h"

/* Step 1: create variables to keep the output: */
void* stream = NULL;  /* caller is responsible for free'ing it after use. */
size_t stream_len = 0;

/* Step 2: call the 2D compression function:
 * Assume that we have a buffer of 128 * 128 floats (in float* type) to be compressed,
 * using PWE tolerance = 1e-3.
 */
int ret = sperr_comp_2d(ptr,           /* memory buffer containing the input */
                        1,             /* the input is of type float; 0 means double. */
                        128,           /* dimx */
                        128,           /* dimy */
                        3,             /* compression mode; 3 means fixed PWE */
                        1e-3,          /* actual PWE tolerance */
                        0,             /* not using a header for the output bitstream */
                        &stream,       /* will hold the compressed bitstream */
                        &stream_len);  /* length of the compressed bitstream */
assert(ret == 0);

/*
 * Now that the 2D compression is completed, one can decompress the bitstream to
 * retrieve the raw values, as the rest of this example shows.
 */

/* Step 3: create a pointer to hold the decompressed values: */
void* output = NULL;  /* caller is responsible for free'ing it after use. */

/* Step 4: call the 2D decompression function: */
int ret2 = sperr_decomp_2d(stream,      /* compressed bitstream */
                           stream_len,  /* compressed bitstream length */
                           1,           /* decompress to floats. 0 means to doubles. */
                           128,         /* dimx */
                           128,         /* dimy */
                           &output);    /* decompressed data is stored here */
assert(ret2 == 0);
free(output);   /* cleanup */
free(stream);   /* cleanup */
```

5.3.2 Example: 3D

```c
/*
 * Example of using the SPERR C API to perform 3D compression and decompression tasks.
 */
#include "SPERR_C_API.h"

/* Step 1: create variables to keep the output: */
void* stream = NULL;  /* caller is responsible for free'ing it after use. */
size_t stream_len = 0;

/* Step 2: call the 3D compression function:
 * Assume that we have a buffer of 256^3 floats (in float* type) to be compressed,
 * using PWE tolerance = 1e-3 and chunk dimension of 128^3.
 */
int ret = sperr_comp_3d(ptr,           /* memory buffer containing the input */
                        1,             /* the input is of type float; 0 means double. */
                        256,           /* dimx */
                        256,           /* dimy */
                        256,           /* dimz */
                        128,           /* chunk_x */
                        128,           /* chunk_y */
                        128,           /* chunk_z */
                        3,             /* compression mode; 3 means fixed PWE */
                        1e-3,          /* actual PWE tolerance */
                        4,             /* use 4 OpenMP threads */
                        &stream,       /* will hold the compressed bitstream */
                        &stream_len);  /* length of the compressed bitstream */
assert(ret == 0);

/*
 * Now that the 3D compression is completed, one can decompress the bitstream to
 * retrieve the raw values, as the rest of this example shows.
 */

/* Step 3: create a pointer to hold the decompressed values,
 * and also variables to hold the volume dimensions.
 */
void* output = NULL;  /* caller is responsible for free'ing it after use. */
size_t dimx = 0, dimy = 0, dimz = 0;

/* Step 4: call the 3D decompression function: */
int ret2 = sperr_decomp_3d(stream,      /* compressed bitstream */
                           stream_len,  /* compressed bitstream length */
                           1,           /* decompress to floats. 0 means to doubles. */
                           4,           /* use 4 OpenMP threads */
                           &dimx,       /* dimx of the decompressed volume */
                           &dimy,       /* dimy of the decompressed volume */
                           &dimz,       /* dimz of the decompressed volume */
                           &output);    /* decompressed data is stored here */
assert(ret2 == 0);
free(output);   /* cleanup */
free(stream);   /* cleanup */
```

6 LC Framework

contact: Martin Burtscher
LC is a framework for automatically generating high-speed lossless and guaranteed-error-bounded lossy data compression and decompression algorithms. It supports CPUs and GPUs.
The framework code and tutorial are available at https://github.com/burtscher/LC-framework/

6.1 Overview

LC consists of the following three parts:

- Component library
- Preprocessor library
- Framework

Both libraries contain separate encoders and decoders for CPU and GPU execution. The user can extend these libraries. The framework takes preprocessors and components from these libraries and chains them into a pipeline to build a compression algorithm. It similarly chains the corresponding decoders in the opposite order to build the matching decompression algorithm. Moreover, the framework can automatically search for effective algorithms for a given input file or set of files by testing user-selected sets of components in each pipeline stage.

6.2 Quick-Start Guide and Tutorial

6.2.1 Installation

To download LC, run the following Linux commands:

```
git clone https://github.com/burtscher/LC-framework.git
cd LC-framework/
```

If you want to run LC on the CPU, generate the framework as follows:

```
./generate_Host_LC-Framework.py
```

If, instead, you want to run LC on the GPU, generate the framework as follows:

```
./generate_Device_LC-Framework.py
```

In either case, run the printed command to compile the generated code. For the CPU, use:

```
g++ -O3 -march=native -fopenmp -DUSE_CPU -I. -std=c++17 -o lc lc.cpp
```

For the GPU, use:

```
nvcc -O3 -arch=sm_70 -DUSE_GPU -Xcompiler "-O3 -march=native -fopenmp" -I. -o lc lc.cu
```

You may have to adjust these commands and flags to your system and compiler. For instance, the sm_70 should be changed to match your GPU's compute capability.
The generate_Hybrid_LC-Framework.py script is only for testing and should not be used as it generates slow code.

6.2.2 Usage Examples for Lossless Compression Algorithms

The following examples assume you have a file called *input.dat* in the current directory and want to find a good compression algorithm for it. See below for a description of the available preprocessors and components. Assume you believe that using bit shuffling (BIT) and run-length encoding (RLE) at 4-byte granularity make a good compressor. Then you can see how well it compresses by entering the following command (note the two pairs of quotes):

```
./lc input.dat CR "" "BIT_4 RLE_4"
```

This will produce output that lists the compression ratio at the end. If you want to see whether running the RLE component at 1-byte granularity performs better, try:

```
./lc input.dat CR "" "BIT_4 RLE_1"
```

To find out which components (and preprocessors) are available, simply run:

```
./lc
```

If you want to see more stats on the input and output data as well as throughput information in addition to the compression ratio, switch the mode from CR to AL:

```
./lc input.dat AL "" "BIT_4 RLE_1"
```

Note that using AL also turns on verification to make sure the decompressed data is bit-by-bit equivalent to the original data.
One of the key strengths of LC is its ability to automatically search for a good compression algorithm. For example, if you want LC to try all available components in the second stage, type:

```
./lc input.dat CR "" "BIT_4 .+"
```

The ".+" is a regular expression that matches the names of all components in the library. You can use it to select any subset of the available components. Of course, you can also use a regular expression for the first pipeline stage by entering:

```
./lc input.dat CR "" ".+ .+"
```

This is not limited to two stages. To search for the best 3-stage pipeline, use:

```
./lc input.dat CR "" ".+ .+ .+"
```

Note that the search time increases exponentially with the number of stages. Before you perform a search, you can check the size of the search space using the PR mode as follows:

```
./lc input.dat PR "" ".+ .+ .+ .+"
```

The output lists the number of algorithms that will be tested as well as which components will be considered in each pipeline stage. If this number is too large, i.e., the search would take too long, try reducing the search space by limiting the number of components to be considered:

```
./lc input.dat CR "" "DIFF_4 .+ .+ R.+|C.+|H.+"
```

If available, we recommend using the GPU version of LC as it tends to be much faster than the CPU version. To further speed up the search, LC includes a genetic algorithm (GA) to quickly search for a good but not necessarily the best algorithm. If you want to run the GA to find a good pipeline with 5 stages, enter the following command:

```
./scripts/ga_search.py -s 5 input.dat
```

If you are interested in the throughput in addition to the compression ratio, use the EX mode of LC like this:

```
./lc input.dat EX "" ".+ .+"
```

The output includes the Pareto front at the end, allowing the user to pick the best algorithm for a given compression or decompression throughput. The six columns list the algorithm, the compression ratio, the CPU compression throughput, the CPU decompression throughput, the GPU compression throughput, and the GPU decompression throughput. The throughputs are given in gigabytes per second.
All CR and EX runs with more than one algorithm also write their results to a CSV file that can be opened with most spreadsheet applications to view and postprocess the results.
EX- and AL-runs with one algorithm write the compressed data to a file called LC.encoded and the decompressed data to a file called LC.decoded.
In summary, LC supports the following modes:

- AL: This mode provides the most detailed output but only works for a single pipeline.
- PR: This mode prints the search space and then quits.
- CR: This mode searches for the best compressing algorithm.
- EX: This mode searches for the algorithms on the Pareto front, taking into account both the compression ratio and the compression/decompression throughput.
- TS: This mode is for testing only and should not be used.

6.2.3 Usage Examples for Lossy Floating-Point Compression Algorithms

To generate lossy algorithms with LC, preprocessors are needed. They must be fully specified (no regular expressions are allowed) and cannot be searched for automatically as they require user-specified parameters such as the error bound.
To find a good lossy compression algorithm for IEEE-754 32-bit single-precision floating-point data that are quantized with a maximum point-wise absolute error bound of 0.01 and then losslessly compressed with three components, enter:

```
./lc input.dat CR "QUANT_ABS_0_f32(0.01)" ".+ .+ R.+|C.+|H.+"
```

To do the same with a point-wise relative error bound, use:

```
./lc input.dat CR "QUANT_REL_0_f32(0.01)" ".+ .+ R.+|C.+|H.+"
```

The preprocessors work with the CR, EX, and AL modes. However, since both EX and AL verify the result, the default lossless verification will likely fail for lossy compression. LC includes a set of verifiers that can be selected in lieu of the default verifier. For an point-wise absolute error bound of 0.001, use:

```
./lc input.dat EX "QUANT_ABS_0_f32(0.001)" ".+ R.+|C.+|H.+" "MAXABS_f32(0.001)"
```

See the ./verifiers/ directory for additional available verifiers or the description below.
These quantizers replace any lost bits with zeros. If you prefer those bits be replaced by random data to minimize autocorrelation, use:

```
./lc input.dat CR "QUANT_ABS_R_f32(0.01)" ".+ .+ R.+|C.+|H.+"
```

or

```
./lc input.dat CR "QUANT_REL_R_f32(0.01)" ".+ .+ R.+|C.+|H.+"
```

6.2.4 Standalone Compressor and Decompressor Generation

Once you have determined a good lossless or lossy compression algorithm (e.g., "TUPL4_1 RRE_1 CLOG_1"), you can generate a standalone compressor and a standalone decompressor that are optimized for this algorithm.
To generate the CPU version, run:

```
./generate_standalone_CPU_compressor_decompressor.py "" "TUPL4_1 RRE_1 CLOG_1"
```

To generate the GPU version, run:

```
./generate_standalone_GPU_compressor_decompressor.py "" "TUPL4_1 RRE_1 CLOG_1"
```

In either case, run the printed commands to compile the generated code. For the CPU, use:

```
g++ -O3 -march=native -fopenmp -I. -std=c++17 -o compress compressor-standalone.cpp
g++ -O3 -march=native -fopenmp -I. -std=c++17 -o decompress decompressor-standalone.cpp
```

For the GPU, use:

```
nvcc -O3 -arch=sm_70 -DUSE_GPU -Xcompiler "-march=native -fopenmp" -I. -o compress compressor-standalone
nvcc -O3 -arch=sm_70 -DUSE_GPU -Xcompiler "-march=native -fopenmp" -I. -o decompress decompressor-standa
```

You may have to adjust these commands and flags to your system and compiler. For instance, the sm_70 should be changed to match your GPU's compute capability.
At this point, you can compress files with:

```
./compress input_file_name compressed_file_name [y]
```

and decompress them with:

```
./decompress compressed_file_name decompressed_file_name [y]
```

Both commands accept an optional "y" parameter at the end. If it is specified, the compressor and decompressor will measure and print the throughput.

6.3 Available Components, Preprocessors, and Verifiers

All LC compression pipelines start with zero or more preprocessors and end with one or more components. The preprocessors require parentheses after their names and may take parameters. The components do not take any parameters and cannot have parentheses.

6.3.1 Available Components

The LC framework breaks the input data up into chunks of 16 kB, each of which is compressed independently and in parallel using the selected components. All components are lossless. Most components support different word sizes. The number at the end of their names indicates the word size in bytes. For example, "_4" means the word size is 4 bytes (e.g., ints or floats). To structure the description of the components, we group them into the following four categories: mutators, shufflers, predictors, and reducers. The goal of the first three types is to better expose patterns so that the reducer components can compress the data more effectively. Only reducer components make sense the the last pipeline stage.
Mutators computationally transform each value. This is done independently of other values and does not compress the data.

- NUL: This component performs the identity transformation, meaning it outputs the input verbatim. It is useful in that it allows longer pipelines to also cover algorithms with fewer stages.
- TCMS: This component converts each value from twos-complement to magnitude-sign representation, which is often easier to compress because it tends to yield more leading zero bits.
- DBEFS: This component operates on IEEE-754 floating-point values. It first de-biases the exponent and then rearranges the data fields from sign, exponent, fraction order to (de-biased) exponent, fraction, sign order.
- DBESF: This component operates on IEEE-754 floating-point values. It first de-biases the exponent and then rearranges the data fields from sign, exponent, fraction order to (de-biased) exponent, sign, fraction order.

Shufflers rearrange the order of the values but perform no computation on them. Some shufflers reorder the bits or bytes within a word. None of them compress the data.

- BIT: This component is often referred to as "bit shuffle" or "bit transpose". It takes the most significant bit of each value in the input and outputs them together, then it takes the second most significant bit of each value and outputs them, and so on down to the least significant bit. This improves compressibility if the values tend to have the same bits in certain positions.
- TUPLk: This component assumes the data to be a sequence of k-tuples, which it rearranges by listing all first tuple values, then all second tuple values, and so on. For example, a tuple size of k = 3 changes the linear sequence x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4 into x1, x2, x3, x4, y1, y2, y3, y4, z1, z2, z3, z4. This is beneficial as values belonging to the same "dimension" often correlate more with each other than with other values from within the same tuple.

Predictors guess the next value by extrapolating it from prior values and then subtracting the prediction from the actual value, which yields a residual sequence. If the predictions are accurate, the residuals cluster around zero, making them easier to compress than the original data. Predictors per se do not compress the data.

- DIFF: This component computes the difference sequence (also called "delta modulation") by subtracting the previous value from the current value and outputting the resulting difference. If neighboring values correlate with each other, this tends to produce a more compressible sequence.
- DIFFMS: This component computes the difference sequence like DIFF does but outputs the result in sign-magnitude format, which is often more compressible because it tends to produce values with many leading zero bits.

Reducers are the only components that can compress the data. They exploit various types of redundancies to do so.

- CLOG: This component breaks the data up into 32 subchunks, determines the smallest amount of leading zero bits of all values in a subchunk, records this count, and then stores only the remaining bits of each value. This compresses data with leading zero bits.
- HCLOG: This component works like CLOG except it first applies the TCMS transformation to all values in a subchunk that yield no leading zero bits when using CLOG.
- RLE: This component performs run-length encoding. It counts how many times a value appears in a row. Then it counts how many non-repeating values follow. Both counts are emitted and followed by a single instance of the repeating value as well as all non-repeating values.
- RRE: This component creates a bitmap in which each bit specifies whether the corresponding word in the input is a repetition of the prior word or not. It outputs the non-repeating words and a compressed version of the bitmap that is repeatedly compressed with the same algorithm.
- RZE: This component creates a bitmap in which each bit specifies whether the corresponding word in the input is zero or not. It outputs the non-zero words and a compressed version of the bitmap like RRE does.

6.3.2 Available Preprocessors

Preprocessors operate on the entire data (i.e., there is no chunking) and can be lossy or lossless. Some preprocessors support different data types. The end of their names indicates the data type for which they are designed. For example, "_f32" means the preprocessor targets 32-bit floating-point values. To structure the description of the preprocessors, we group them into lossy or lossless preprocessors.

Lossless

These quantizers support INFs and NaNs. The end of the quantizer name indicates the data type for which it is designed.

- NUL: This preprocessor performs the identity transformation, meaning it outputs the input verbatim. It takes no parameters.
- LOR1D: This preprocessor performs a 1-dimensional Lorenzo transformation, i.e., it computes a difference sequence.

6.3.3 Lossy Quantizers

All quantizers require a parameter that specifies the maximally allowed error bound EB. They take an optional second parameter specifying a threshold. Any value whose magnitude is at or above the threshold is compressed losslessly and not quantized. The quantizers support INFs and NaNs. The end of the quantizer name indicates the data type for which it is designed.

- QUANT_ABS_0: These preprocessors quantize 32- and 64-bit floating-point values based on the provided point-wise absolute error bound. All values that end up in the same quantization bin are decompressed to the same value. These preprocessors guarantee that the original value V is decoded to a value V' such that V - EB <= V' <= V + EB.
- QUANT_ABS_R: These preprocessors quantize 32- and 64-bit floating-point values based on the provided point-wise absolute error bound. Each value from the same quantization bin is decompressed to a random value within the provided error bound to minimize autocorrelation. These preprocessors guarantee that the original value V is decoded to a value V' such that V - EB <= V' <= V + EB.
- QUANT_R2R: These preprocessors quantize 32- and 64-bit floating-point values just like their QUANT_ABS counterparts except the provided error bound is first multiplied by the range of values occurring in the input, where the range is the maximum value minus the minimum value.
- QUANT_REL_0: These preprocessors quantize 32- and 64-bit floating-point values based on the provided point-wise relative error bound. All values that end up in the same quantization bin are decompressed to the same value. These preprocessors guarantee that the original value V is decoded to a value V' with the same sign such that |V| / (1 + EB) <= |V'| <= |V| * (1 + EB).
- QUANT_REL_R: These preprocessors quantize 32- and 64-bit floating-point values based on the provided point-wise relative error bound. Each value from the same quantization bin is decompressed to a random value within the provided error bound to minimize autocorrelation. These preprocessors guarantee that the original value V is decoded to a value V' with the same sign such that |V| / (1 + EB) <= |V'| <= |V| * (1 + EB).

6.3.4 Available Verifiers

Some verifiers support different data types. The end of their names indicates the data type for which they are designed.

- LOSSLESS: This verifier is the default. It passes verification if the decompressed output matches every bit of the original input.
- PASS: This verifier always passes verification and is only useful for debugging.
- MAXABS: This verifier takes a point-wise absolute error bound as parameter and only passes verification if every output value is within the specified error bound.
- MAXR2R: This verifier works like MAXABS except the provided error bound is first multiplied by the range of values occurring in the input, where the range is the maximum value minus the minimum value.
- MAXREL: This verifier takes a point-wise relative error bound as parameter and only passes verification if every output value is within the specified error bound.

- MSE: This verifier takes a mean squared error as parameter and only passes verification if the mean squared error of the output values is within the error bound.
- PSNR: This verifier takes a peak-signal-to-noise ratio (PSNR) as parameter and only passes verification if the PSNR of the output values is above the specified lower bound.

For learning how to add your own components and preprocessors, please see https://github.com/burtscher/LC-framework/?tab=readme-ov-file#adding-and-removing-components.

7 TEZip

TEZip is not installed in the software envionment due to its comparatively complex dependencies and compatibility with the GPUs in the testing envionment

contact: Amarjit Singh
TEZip(Time Evolutionary Zip) is developed in RIKEN R-CCS and designed to compress time evolutionary data by using a deep learning for prediction. TEZip compression/decompression procedures consist of three steps, model training, compression and decompression. For more details, please refer to the readthedocs document full documentation

7.1 CLI Interface

```
# 1. Creating training data
# Comandline options
#   [Learning_image_directory]: The directory path containing the training images to be
#       dumped into the Hkl file (e.g., ./data)
#   [Output_directory]: Directory path to output Hkl file (e.g., ./data_hkl).
#   -v: Specifies the path of the directory used for verification among the directories
#       specified in the first argument. Without this option, randomly determined.
python train_data_create.py [Learning_image_directory] [Output_directory]

# 2. Model Training
# Comandline options
#   -l: Execute learning mechanism.
#       [Output_directory] is "Path to the output directory of the model"
#       [Directory_for_training_data] is "Path to the training data directory(.hkl)"
#   -f: Forced CPU mode flag
#      "-f" to the runtime will disable the GPU and force it to run on the CPU
#   -v: Flag for screen output
#      "-v" to the runtime, the learning status, such as losses and epochs during
#      learning, will be output to the console.
python tezip.py -l [Output_directory] [Directory_for_training_data]

# 3. Compression
# Comandline options
#   -c: Run the compression mechanism
#       [Model_directory] Path of the directory of trained models
#       [Directory_of_images_to_be_compressed] Directory path of the image to be
#           compressed
#       [Output_directory] Output directory path for compressed data
#   -w: Specifying the keyframe switching criteria
#       SWP(Static Window-based Prediction)to specify how many keyframes to infer from
#       one keyframe of execution. If it is specified at the same time as -t  it will
#       cause an error termination
#   -t: Specify the criteria for keyframe switching
#       Specify the threshold value of MSE(Mean Square Error) for execution switching in
#       DWP(Dynamic Window-based Prediction). If it is specified at the same time as
#       -w  it will cause an error termination
#   -p: Number of images for warm-up
#       The more keyframes you specify for LSTM recording  the larger the
#       "key_frame.dat" will be and the smaller the "entropy.dat" will likely be.
#       However, if you set the number of keyframes to 0 or 1 when running DWP  the MSE
```

```
#       will become larger and the final number of keyframes may become larger
#   -m: Selecting an error-bound mechanism
#       Select the error bouncing mechanism from the following four types
#           abs absolute error bound
#           rel relative bound ratio
#           absrel Do both of the above
#           pwrel point wise relative error bound
#       If you select multiple items or select non-existent items  the #program will
#       exit with an error
#   -b: Threshold for error bouncing mechanism
#       Specify the tolerance threshold of the error bouncing mechanism. If "-m" is
#       specified as absrel,  enter two values.
#       If an inappropriate number of inputs are given for the one specified by  -m
#       the program will exit with an error.If the input contains  0 the error
#       bouncing mechanism will not be executed and the data will be fully lossy
#       compressed.
#   -f: Forced CPU mode flag
#       By adding  -f  to the runtime  you can disable the GPU and force it to run on
#       the CPU
#   -v: Flag for screen output
#       When  -v  is added at runtime  the status during execution  such as the value
#       of MSE after inference and the time taken for the compression process  will be
#       output to the console
#   -n: Flag to disable Entropy Coding for compression process
python tezip.py -c [Model_directory] [Directory_of_images_to_be_compressed]
[Output_directory] -p [Number_of_warm-up_sheets] -w or -t [-w
Number_of_inferences_to_be_made_from_a_single_keyframe , -t
MSE_threshold_for_keyframe_switching] -m [Error-bound_mechanism_name] -b
[Threshold_for_error_bouncing_mechanism]

# 4. Decompressoin
# Comandline options
#   -u: Run the learning mechanism
#       [Model_directory] Directory path of trained models
#       [Directory_of_compressed_data]  Directory path for compressed data (.dat), etc.
#       [Output_directory] Output directory path for unzipped data
#   -f: Forced CPU mode flag
#       By adding  -f  to the runtime, you can disable the GPU and force it to run on
#       the CPU.
#   -v: Flag for screen output
#       By adding  -v  at runtime, the processing time during decompression is output
#       to the console
python tezip.py -u [Model_directory] [Directory_of_compressed_data] [Output_directory]

```

7.2 C/C++ Interface

TEZip does not have its own C/C++ interface but may be called via LibPressio's C/C++.

8 DCTZ

contact: Seung Woo Son
DCTZ is a lossy compressor based on DCT (discrete cosine transform) tailored to work with floating point datasets, single- or double-precision, with an error-bound capability (primarily on relative error bound). The compression pipeline has been adapted from JPEG.

8.1 CLI Interface

DCTZ has a single test program that goes through compression/decompression as a single test scenario. There are two versions of test programs: dctz-ec-test and dctz-qt-test. dctz-ec-test is a more conservative compressor, whereas dctz-qt-test includes an extra quantization process to improve compression ratios while guaranteeing the user-defined error bounds. The test scripts, test-dctz.sh or test-dctz-f.sh, for a couple of sample datasets are available under the "tests" folder in the DCTZ repository.

```
# print help for dctz compressor
# both dctz-ec-test and dctz-qt-test has the same CLI
dctz-ec-test --help
Test case: ../dctz-ec-test -d|-f [err bound] [var name] [srcFilePath] [dimension sizes...]

# commandline options
#  precision: -d | -f                    double or float
#  error bound: 1E-3, 1E-4, or 1E-5
#  var name: unused but
#  data file path
#  dimensions list

# sample CLI for evaluating the CESM-ATM-taylor
# the error bound is set to 1E-3
dctz-qt-test -f 1E-3 var CESM-ATM-tylor/1800x3600/CLDHGH_1_1800_3600.dat 3600 1800

# The above command will generate a compressed file named CLDHGH_1_1800_3600.dat.z
# and the decompressed file named CLDHGH_1_1800_3600.dat.z.r.

```

9 LibPressio

contact: Robert Underwood
LibPressio provides a generic abstraction accross the various compressors including ALL of the compressors listed above. It will eventually serve as the front end to FZ. In general, all features are supported from all interfaces.
The material presented here adapts a subset of the LibPressio tutorial. The full tutorial covers GPU based compressors, using compressors with user provided metrics, usage with I/O libraries, automatic optimization of compressors with OptZConfig, and much more. You can also find full documentation and more creatives uses of LibPressio.
Below we present the basics for the commandline, Python, C++, and C interfaces. We also have support for R, Julia, and Rust.

9.1 CLI interface

LibPressio allows you to rapid experiment with various compressors

```
#print help for the libpressio command
pressio

# load a 3d array of float, use SZ3 with an absolute error bound of 1e-4
# and collect various metrics and print them out
pressio \
    -i /usecases/Hurricane/Uf48.dat -d 500 -d 500 -d 100 -t float
    -b compressor=sz3 -o abs=1e-4
    -m error_stat -m size -m time -M all

# same but with an HDF5 file
pressio \
    -i /path/to/foo.hdf5 -I /some/dataset
    -b compressor=sz3 -o abs=1e-4
    -m error_stat -m size -m time -M all

# same but with an numpy file
pressio \
    -i /path/to/foo.npy -T numpy
    -b compressor=sz3 -o abs=1e-4
    -m error_stat -m size -m time -M all


# same but with ZFP, and just error statistics
pressio \
    -i /usecases/Hurricane/Uf48.dat -d 500 -d 500 -d 100 -t float
    -b compressor=zfp -o abs=1e-4
    -m error_stat -M all

# use ZFP's fixed rate mode
pressio \
    -i /usecases/Hurricane/Uf48.dat -d 500 -d 500 -d 100 -t float
    -b compressor=zfp -o zfp:rate=13
    -m error_stat -M all

# SZ3 but with a REL error bound
pressio \
    -i /usecases/Hurricane/Uf48.dat -d 500 -d 500 -d 100 -t float
```

```
    -b compressor=zfp -o rel=1e-4
    -m error_stat -M all

#print the builtin help for ZFP
pressio -a help -b compressor=zfp

#print the settings configured for ZFP
pressio \
    -a settings -O all \
    -b compressor=zfp -o rel=1e-4
```

9.2 Python Interface

You can use `pressio_new client:python sz3 /usecases/Hurricane/Uf48.dat` to generate a starter code. You can run `pressio_new` to see the list of available starter codes. The code that appears below is a commmented version of what this tool outputs with one line added to compute error statistics:

```python
import libpressio as lp
import numpy as np
from pprint import pprint

# read in the file as a binary array
input = np.fromfile("/usecases/Hurricane/Uf48.dat", dtype=np.float32).reshape(100, 500, 500)

# configure the compressor
compressor = lp.PressioCompressor.from_config({
    "compressor_id": "pressio",
    "early_config": {
        "pressio:compressor": "sz3"
    },
    "compressor_config": {
        "pressio:abs": 1e-5
    }
})

# run the compressor
decompressed = input.copy()
compressed = compressor.encode(input)
decompressed = compressor.decode(compressed, decompressed)

# collect metrics results
pprint(compressor.get_metrics())
```

9.3 C++ Interface

You can use `pressio_new client:cpp sz3 /usecases/Hurricane/Uf48.dat` to generate a starter code. You can run `pressio_new` to see the list of available starter codes. The code that appears below is a commmented version of what this tool outputs with one line added to compute error statistics:

```cpp
#include <libpressio_ext/cpp/libpressio.h>
#include <libpressio_meta.h>
#include <iostream>
#include <string>

using namespace std::string_literals;
```

```cpp

int main() {
    //load 3rd party compressors definitions.  These support compressors with licences that
    //prevent it from being used with LibPressio directly (e.g. the GPL or a proritary licence)
    pressio library;
    libpressio_register_all();
    pressio_compressor compressor = library.get_compressor("pressio");

    //the data is a 3d array of type float.  100 is the slowest advancing dimension
    //this is called column major order or fortran order
    pressio_dtype type = pressio_float_dtype;
    std::vector<size_t> dims {500,500,100};
    pressio_data metadata = pressio_data::owning(type, dims);

    // while using an I/O plugin is overkill here, we can change posix to "hdf5"
    // to load an HDF5 dataset, or "numpy" to read a .npy file.
    pressio_io io = library.get_io("posix");
    if(!io) {
        std::cerr << library.err_msg() << std::endl;
        exit(library.err_code());
    }
    if(io->set_options({
      {"io:path", "/usecases/Hurricane/Uf48.dat"s},
    })) {
        std::cerr << io->error_msg() << std::endl;
        exit(io->error_code());
    }
    pressio_data* input = io->read(&metadata);
    pressio_data compressed = pressio_data::empty(pressio_byte_dtype, {});
    pressio_data output = pressio_data::owning(type, dims);

    //configure the compressor to use SZ3 with an absolute error bound.
    //we can change sz3 to zfp here and choose what compressor to use
    if(compressor->set_options({
        {"pressio:compressor", "sz3"},
        {"pressio:abs", 1e-5}
    })) {
        std::cerr << io->error_msg() << std::endl;
        exit(io->error_code());
    }

    //run the compressor
   if(compressor->compress(input, &compressed) != 0) {
           std::cerr << compressor->error_msg() << std::endl;
           exit(compressor->error_code());
   }
   if(compressor->decompress(&compressed, &output) != 0) {
           std::cerr << compressor->error_msg() << std::endl;
           exit(compressor->error_code());
   }

    //collect the metrics results and print them them to the screen
  std::cout << compressor->get_metrics_results() << std::endl;
        delete input;
```

```
}
```

9.4 C interface

You can use `pressio_new client:c sz3 /usecases/Hurricane/Uf48.dat` to generate a starter code. You can run `pressio_new` to see the list of available starter codes. The code that appears below is a commmented version of what this tool outputs with one line added to compute error statistics:

```c
#include <stdlib.h>
#include <stdio.h>
#include <libpressio.h>
#include <libpressio_meta.h>

int main() {
        struct pressio* library = pressio_instance();
        //load 3rd party compressors definitions.  These support compressors with licences that
        //prevent it from being used with LibPressio directly (e.g. the GPL or a proritary licence)
        libpressio_register_all();

        //load the "pressio" meta compressor that provides some standardization about what
        //compressors error bounds to support.  For example, if a compressor support an ABS bound
        //"pressio" will provide a REL bound for free
        struct pressio_compressor* compressor = pressio_get_compressor(library, "pressio");
        if(!compressor) {
                fprintf(stderr, "%s\n", pressio_error_msg(library));
                exit(pressio_error_code(library));
        }

        //the data is a 3d array of type float.  100 is the slowest advancing dimension
        //this is called column major order or fortran order
        enum pressio_dtype type = pressio_float_dtype;
        size_t dims[] = {500,500,100};
        struct pressio_data* metadata = pressio_data_new_owning(type,
            sizeof(dims)/sizeof(dims[0]), dims);

        // while using an I/O plugin is overkill here, we can change posix to "hdf5"
        // to load an HDF5 dataset, or "numpy" to read a .npy file.
        struct pressio_io* io = pressio_get_io(library, "posix");
        if(!io) {
                fprintf(stderr, "%s\n", pressio_error_msg(library));
                exit(pressio_error_code(library));
        }
        struct pressio_options* io_opts = pressio_options_new();
        pressio_options_set_string(io_opts, "io:path", "/usecases/Hurricane/Uf48.dat");
        if(pressio_io_set_options(io, io_opts)) {
                fprintf(stderr, "%s\n", pressio_io_error_msg(io));
                exit(pressio_io_error_code(io));
        }
        pressio_options_free(io_opts);
        struct pressio_data* input = pressio_io_read(io,metadata);
        if(!input) {
                fprintf(stderr, "%s\n", pressio_io_error_msg(io));
                exit(pressio_io_error_code(io));
        }
        pressio_data_free(metadata);
```

```
48          pressio_io_free(io);
49
50          //setup output locations for the compressed and decompressed data
51          struct pressio_data* compressed = pressio_data_new_empty(pressio_byte_dtype, 0, NULL);
52          struct pressio_data* output = pressio_data_new_owning(type,
53              sizeof(dims)/sizeof(dims[0]), dims);
54
55          //configure the compressor to use SZ3 with an absolute error bound.
56          //we can change sz3 to zfp here and choose what compressor to use
57          struct pressio_options* compressor_opts = pressio_options_new();
58          pressio_options_set_string(compressor_opts, "pressio:compressor", "sz3");
59          //this line was added to compute error statistics when decompression is run
60          pressio_options_set_string(compressor_opts, "pressio:metric", "error_stat");
61          pressio_options_set_double(compressor_opts, "pressio:abs", 1e-5);
62          if(pressio_compressor_set_options(compressor, compressor_opts)) {
63                  fprintf(stderr, "%s\n", pressio_compressor_error_msg(compressor));
64                  exit(pressio_compressor_error_code(compressor));
65          }
66          pressio_options_free(compressor_opts);
67
68          //run the compressor
69          if(pressio_compressor_compress(compressor, input, compressed) != 0) {
70                  fprintf(stderr, "%s\n", pressio_compressor_error_msg(compressor));
71                  exit(pressio_compressor_error_code(compressor));
72          }
73          if(pressio_compressor_decompress(compressor,compressed,output) != 0) {
74                  fprintf(stderr, "%s\n", pressio_compressor_error_msg(compressor));
75                  exit(pressio_compressor_error_code(compressor));
76          }
77
78          //collect the metrics results and print them them to the screen
79          struct pressio_options* metrics = pressio_compressor_get_metrics_results(compressor);
80          char* metrics_str = pressio_options_to_string(metrics);
81          fprintf(stdout, "%s\n", metrics_str);
82          free(metrics_str);
83          pressio_options_free(metrics);
84
85          //free memory
86          pressio_data_free(input);
87          pressio_data_free(compressed);
88          pressio_data_free(output);
89          pressio_compressor_release(compressor);
90          pressio_release(library);
91  }
```