SPERR Interface and Examples

Samuel Li

February 15, 2024

1 Introduction

This document is the SPERR section of the handout for the FZ workshop hands-on session on February 15 2024, Sarasota, FL. It provides examples for the CLI interface, C++ interface, and C interface of SPERR (Section 2.1, 2.2, and 2.3, respectively).

2 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.

2.1 CLI Interface

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

2.1.1 `sperr2d`

`sperr2d` is responsible for compressing and decompressing a 2D data slice. In compression mode (`-c`), it takes as input a raw binary file consisting of single- or double-precision floating point values, and outputs a compressed bitstream. In decompression mode (`-d`), it takes as input a compressed bitstream, and outputs the decompressed binary file of single- or double-precision floating point values. 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 (choose one):
  --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.

2.1.2 sperr3d

`sperr3d` is responsible for compressing and decompressing a 2D data volume. Similar to `sperr2d`, it operates in either compression (`-c`) or decompression (`-d`) mode, converting between raw binary floating-point values and compressed bitstreams. Different from `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 chunking step allows for compressing and decompressing all the small chunks individually and in parallel. `sperr3d` uses 256^3 as the default chunk size, but any number from dozens to low hundreds would work well. (Chunk dimensions that can divide the full volume are preferred, but not mandatory.) 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.

2.1.3 sperr3d_trunc and Flexible-Rate Decoding

Compressed SPERR bitstreams support *flexible-rate* decoding, meaning that a sub-bitstream of it from the beginning can still be used to reconstruct the data field. Equally important, the reconstruction will have the best possible quality (in terms of average error) under the the size constraint of the sub-bitstream, though lower quality than using the full bitstream. Given a compressed bitstream, a sub-bitstream can be obtained by a simple truncation, for example, a truncation that keeps the first 10% of the full bitstream.
The chunking scheme used in 3D compression (see Section 2.1.2) brings some complication, because the bitstream representing each chunk needs to be truncated *individually*. `sperr3d_trunc` is thus introduced to properly truncate a multi-chunked SPERR bitstream. Specifically, it 1) locates the bitstream representing each chunk, 2) truncates the bitstream, and 3) stitches all truncated bitstreams together so `sperr3d` can properly decompress it.
The help message of `sperr3d_trunc` details its options:

SPERR bitstreams without using multi-chunks (i.e., `--dims` equals `--chunks` in 3D, and all 2D cases) can safely be truncated by any method. For example, the following command truncates a compressed bitstream `field.stream` to keep its first 5kB as `field2.stream`, which is also recognized by SPERR:
`head -c 5000 density.stream > density2.stream`

```
$ ./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`

2.2 C++ Interface

2.2.1 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);
```

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

```

```cpp
// 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
```

2.2.2 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.

```cpp
//
// 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();

// Step 2: specify the volume and chunk dimensions, respectively:
encoder.set_dims_and chunks({384, 384, 256}, {192, 192, 128});

// Step 3: specify the number of OpenMP threads to use:
encoder.set_num_threads(4);

// 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:
// The input data is passed in in the form of a raw pointer (float* or double*),
// and the total number of values will be passed in here too.
encoder.compress(ptr, 384 * 384 * 256);

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

```
//
// Example of using a sperr::SPERR3D_OMP_D() to decompress a bitstream.
// This is a 5-step process.
//
#include "SPERR3D_OMP_D.h"

// Step 1: create a decoder:
auto decoder = sperr::SPERR3D_OMP_D();

// Step 2: specify the number of OpenMP threads to use:
decoder.set_num_threads(4);

// 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:
// Note that the pointer to the bitstream is passed in again!
decoder.decompress(ptr, multi_res);  // a boolean, if to enable multi-resolution decoding

// Step 5: retrieve the decompressed volume:
auto [dimx, dimy, dimz] = decoder.get_dims();  // dimension of the 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 memory copies.
std::vector<double> vol = decoder.release_decoded_data();
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.

2.3 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.

2.3.1 Example: 2D

```
/*
 * 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 */
```

```c
                        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 */
```

2.3.2 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 */
```