
H5Z-ZFP Documentation

Release 0.6.0

H5Z-ZFP

May 28, 2019

Contents

1	Installation	3
2	Interfaces	7
3	Endian Issues	11
4	Tests and Examples	13

H5Z-ZFP is a compression filter for HDF5 using the [ZFP compression library](#), supporting lossy compression of floating point and integer data to meet bitrate, accuracy, and/or precision targets. The filter uses the [registered](#) HDF5 filter ID, 32013. It supports single and double precision floating point and integer data *chunked* in 1, 2 or 3 dimensions. The filter will function on datasets of more than 3 dimensions, albeit at the probable expense of compression performance, as long as the chunking is such that no more than 3 dimensions of a chunk are non-unity.

Contents:

1.1 Prerequisites

- [ZFP Library](#) (or from [Github](#))
- [HDF5 Library](#)
- [H5Z-ZFP filter plugin](#)

1.1.1 Compiling ZFP

- There is a `Config` file in top-level directory of the ZFP distribution that holds `make` variables the ZFP Makefiles use. By default, this file is setup for a vanilla GNU compiler. If this is not the appropriate compiler, edit `Config` as necessary to adjust the compiler and compilation flags.
- An important flag you **will** need to adjust in order to use the ZFP library with this HDF5 filter is the `BIT_STREAM_WORD_TYPE` CPP flag. To use ZFP with H5Z-ZFP, the ZFP library **must** be compiled with `BIT_STREAM_WORD_TYPE` of `uint8`. Typically, this is achieved by including a line in `Config` of the form `DEFS += -DBIT_STREAM_WORD_TYPE=uint8`. If you attempt to use this filter with a ZFP library compiled differently from this, the filter's `can_apply` method will always return false. This will result in silently ignoring an HDF5 client's request to compress data with ZFP. Also, be sure to see [Endian Issues](#).
- After you have setup `Config`, simply run `make` and it will build the ZFP library placing the library in a `lib` sub-directory and the necessary include files in `inc` sub-directory.
- For more information and details, please see the [ZFP README](#).

1.1.2 Compiling HDF5

- If you want to be able to run the fortran tests for this filter, HDF5 must be configured with *both* the `--enable-fortran` and `--enable-fortran2003` configuration switches. Otherwise, any vanilla installation of HDF5 is acceptable.

- The Fortran interface to this filter *requires* a Fortran 2003 compiler because it uses `ISO_C_BINDING` to define the Fortran interface.

1.2 Compiling H5Z-ZFP

H5Z-ZFP is designed to be compiled both as a standalone HDF5 *plugin* and as a separate *library* an application can explicitly link. See *Plugin vs. Library Operation*.

Once you have installed the prerequisites, you can compile H5Z-ZFP using a command-line...

```
make [FC=<Fortran-compiler>] CC=<C-compiler>
      ZFP_HOME=<path-to-zfp> HDF5_HOME=<path-to-hdf5>
      PREFIX=<path-to-install>
```

where `<path-to-zfp>` is a directory containing ZFP `inc` and `lib` dirs and `<path-to-hdf5>` is a directory containing HDF5 `include` and `lib` dirs. If you don't specify a C compiler, it will try to guess one from your path. Fortran compilation is optional. If you do not specify a Fortran compiler, it will not attempt to build the Fortran interface.

The Makefile uses GNU Make syntax and is designed to work on OSX and Linux. The filter has been tested on gcc, clang, xlc, icc and pgcc compilers and checked with valgrind.

The command `make help` will print useful information about various make targets and variables. `make check` will compile everything and run a handful of tests.

If you don't specify a `PREFIX`, it will install to `./install`. The installed filter will look like...

```
$(PREFIX)/include/{H5Zzfp.h,H5Zzfp_plugin.h,H5Zzfp_props.h,H5Zzfp_lib.h}
$(PREFIX)/plugin/libh5zzfp.{so,dylib}
$(PREFIX)/lib/libh5zzfp.a
```

where `$(PREFIX)` resolves to whatever the full path of the installation is.

To use the installed filter as an HDF5 *plugin*, you would specify, for example, `setenv HDF5_PLUGIN_PATH $(PREFIX)/plugin`

1.3 H5Z-ZFP Source Code Organization

The source code is in two separate directories

- `src` includes the ZFP filter and a few header files
 - `H5Zzfp_plugin.h` is an optional header file applications *may* wish to include because it contains several convenient macros for easily controlling various compression modes of the ZFP library (*rate*, *precision*, *accuracy*, *expert*) via the *Generic Interface*.
 - `H5Zzfp_props.h` is a header file that contains functions to control the filter using *temporary Properties Interface*. Fortran callers are *required* to use this interface.

- `H5Zzfp_lib.h` is a header file for applications that wish to use the filter explicitly as a library rather than a plugin.
- `H5Zzfp.h` is an *all-of-the-above* header file for applications that don't care too much about separating out the above functionalities.
- `test` includes various tests. In particular `test_write.c` includes examples of using both the *Generic Interface* and *Properties Interface*. In addition, there is an example of how to use the filter from Fortran in `test_rw_fortran.F90`.

1.4 Silo Integration

This plugin is also part of the [Silo library](#). In particular, the ZFP library itself is also embedded in Silo but is protected from appearing in Silo's global namespace through a struct of function pointers (see [Namespaces in C](#)). If you happen to examine the source code for H5Z-ZFP, you will see some logic there that is specific to using this plugin within Silo and dealing with ZFP as an embedded library using this struct of function pointers wrapper. Just ignore this.

There are two interfaces to control the filter. One uses HDF5's *generic* interface via an array of `unsigned int cd_values` as is used in `H5Pset_filter()`. The other uses HDF5 *generic* properties added to the *dataset creation property list* used when the dataset to be compressed is being created. You can find examples of writing HDF5 data using both of these interfaces in `test_write.c`.

The plugin is designed to respond correctly when either interface is used.

The filter supports all of the standard ZFP controls for affecting compression including *rate*, *precision*, *accuracy*, and *expert* modes. For more information and details about these modes of controlling ZFP compression, please see the [ZFP README](#).

2.1 Generic Interface

For the generic interface, the following CPP macros are defined in the `H5Zzfp_plugin.h` header file:

```
H5Pset_zfp_rate_cdata(double rate, size_t cd_nelmts, unsigned int *cd_vals);
H5Pset_zfp_precision_cdata(unsigned int prec, size_t cd_nelmts, unsigned int *cd_
↪vals);
H5Pset_zfp_accuracy_cdata(double acc, size_t cd_nelmts, unsigned int *cd_vals);
H5Pset_zfp_expert_cdata(unsigned int minbits, unsigned int maxbits,
                        unsigned int maxprec, int minexp,
                        size_t cd_nelmts, unsigned int *cd_vals);
```

These macros utilize *type punning* to store the relevant ZFP parameters into a sufficiently large array (≥ 6) of `unsigned int cd_values`. It is up to the caller to then call `H5Pset_filter()` with the array of `cd_values` constructed by one of these macros.

However, these macros are only a convenience. You do not **need** the `H5Zzfp_plugin.h` header file if you want to avoid using it. But, you are then responsible for setting up the `cd_values` array correctly for the filter. For reference, the `cd_values` array for this ZFP filter is defined like so...

	cd_values index					
ZFP mode	0	1	2	3	4	5
rate	1	unused	rateA	rateB	unused	unused
precision	2	unused	prec	unused	unused	unused
accuracy	3	unused	accA	accB	unused	unused
expert	4	unused	minbits	maxbits	maxprec	minexp

A/B are high/low 32-bit words of a double.

Note that the `cd_values` used in the interface to `H5Pset_filter()` are **not the same** `cd_values` ultimately stored to the HDF5 dataset header for a compressed dataset. The values are transformed in the `set_local` method to use ZFP's internal routines for 'meta' and 'mode' data. So, don't make the mistake of examining the values you find in a file and think you can use those same values, for example, in an invocation of `h5repack`.

Because of the type punning involved, the generic interface is not suitable for Fortran callers.

2.2 Properties Interface

For the properties interface, the following functions are defined in the `H5Zzfp_props.h` header file:

```
herr_t H5Pset_zfp_rate(hid_t dcpl_id, double rate);
herr_t H5Pset_zfp_precision(hid_t dcpl_id, unsigned int prec);
herr_t H5Pset_zfp_accuracy(hid_t dcpl_id, double acc);
herr_t H5Pset_zfp_expert(hid_t dcpl_id,
    unsigned int minbits, unsigned int maxbits,
    unsigned int maxprec, int minexp);
```

These functions take a dataset creation property list, `hid_t dcpl_id` and create temporary (e.g. UNregistered) HDF5 property list entries to control the ZFP filter. Calling any of these functions removes the effects of any previous call to any one of these functions. In addition, calling any one of these functions also has the effect of adding the filter to the pipeline.

The properties interface is more type-safe. However, there is no way for the implementation of these properties to reside within the filter plugin itself. The properties interface requires that the caller link with with the filter *library*, `libh5zfp.a`. The generic interface does not require this.

Note that the properties and generic interfaces can be used when the filter is used either as a plugin or as a library. The difference is whether the application calls `H5Z_zfp_initialize()` or not.

2.3 Fortran Interface

A Fortran interface based on the properties interface, described above, has been added by Scot Breitenfeld of the HDF5 group. The code that implements the Fortran interface is in the file `H5Zzfp_props_f.F90`. An example of its use is in `test/test_rw_fortran.F90`. The properties interface is the only interface available for Fortran callers.

2.4 Plugin vs. Library Operation

The filter is designed to be compiled for use as both a standalone HDF5 *plugin* and as an explicitly linked *library*. When it is used as a plugin, all HDF5 applications are *required to find* the plugin dynamic library (named `lib*.so`, `dylib`) in a directory specified by the environment variable, `HDF5_PLUGIN_PATH`. Currently, the HDF5 library offers no mechanism for applications themselves to have pre-programmed paths in which to search for a

plugin. Applications are then always vulnerable to an incorrectly specified or unspecified `HDF5_PLUGIN_PATH` environment variable.

However, the plugin can also be used explicitly as a *library*. In this case, **do not** specify the `HDF5_PLUGIN_PATH` environment variable and instead have the application link to `libH5Zzfp.a` in the `lib` dir of the installation. Instead two initialization and finalization routines are defined:

```
int H5Z_zfp_initialize(void);  
int H5Z_zfp_finalize(void);
```

These functions are defined in the `H5Zzfp_lib.h` header file. Any applications that wish to use the filter as a *library* are required to call the initialization routine, `H5Z_zfp_initialize()` before the filter can be referenced. In addition, to free up resources used by the filter, applications may call `H5Z_zfp_finalize()` when they are done using the filter.

The ZFP library writes an endian-independent stream.

When reading ZFP compressed data on a machine with a different endian-ness than the writer, there is an unavoidable inefficiency. Upon reading data from disk and decompressing the read stream with ZFP, the correct endian-ness is returned in the result from ZFP before the buffer is handed back to HDF5 from the decompression filter. This happens regardless of reader and writer endian-ness incompatibility. However, the HDF5 library is expecting to get from the decompression filter the endian-ness of the data as it was stored to file (typically that of the writer machine) and expects to have to byte-swap that buffer before returning to any endian-incompatible caller. So, in the H5Z-ZFP plugin, we wind up having to un-byte-swap an already correct result read in a cross-endian context. That way, when HDF5 gets the data and byte-swaps it, it will produce the correct result. There is an endian-ness test in the Makefile and two ZFP compressed example datasets for big-endian and little-endian machines to test that cross-endian reads/writes work correctly.

Finally, *endian-targetting*, that is setting the file datatype for an endian-ness that is possibly different than the native endian-ness of the writer, is currently dis-allowed with H5Z-ZFP. Under these conditions, the `can_apply` method will return 0. This constraint can be relaxed, at the expense of an additional endian-UN-swap pass just prior to compression, at a future date if it becomes too onerous.

Tests and Examples

The tests directory contains a few simple tests of the H5Z-ZFP filter.

The test client, `test_write.c` is compiled a couple of different ways. One produces `test_write_plugin` which demonstrates the use of this filter as a standalone plugin. The other produces `test_write_lib`, demonstrates the use of the filter as an explicitly linked library. These test a simple 1D array with and without ZFP compression using either the *Generic Interface* or the *Properties Interface*. You can use the code there as an example of using the ZFP filter either as a plugin or as a library. The command `test_write_lib --help` or `test_write_plugin --help` will print a list of the example's options and how to use them.

There is a companion, `test_read.c` which is compiled into `test_read_plugin` and `test_read_lib` which demonstrates use of the filter reading data as a plugin or library.

Finally, there is a Fortran test example, `test_rw_fortran.F90`. The Fortran test writes and reads a 2D dataset. However, the Fortran test is designed to use the filter **only** as a library and not as a plugin. The reason for this is that the filter controls involve passing combinations of integer and floating point data from Fortran callers and this can be done only through the *Properties Interface*, which by its nature requires any Fortran application to have to link with an implementation of that interface. Since we need to link extra code for Fortran, we may as well also link to the filter itself alleviating the need to use the filter as a plugin.

In addition, a number tests are performed in the Makefile which test the plugin by using some of the HDF5 tools such as `h5dump` and `h5repack`.