Lawrence Livermore National Laboratory

7000 East Avenue
Livermore CA 94550

Contact
Peter Lindstrom

ZFP:
Fast, Accurate Data Compression for Modern Supercomputing Applications

Prepared for:

2023 R&D 100 Award Entry

Lawrence Livermore National Laboratory

LLNL-MI-849560

Prepared by LLNL under Contract DE-AC52-07NA27344.

ZFP:
Fast, Accurate Data Compression for Modern Supercomputing Applications

1. PRODUCT/SERVICES CATEGORIES

A. Title

ZFP v1.0.0

B. Product category

Software/Services

2. R&D 100 PRODUCT/SERVICE DETAILS

A. Primary submitting organization

Lawrence Livermore National Laboratory

B. Co-developing organization

n/a

C. Product brand name

ZFP

D. Product introduction

This product was introduced to the market between January 1, 2022, and March 31, 2023. This product is not subject to regulatory approval. ZFP version 1.0.0 was released on August 1, 2022.

E. Price in U.S. Dollars

Free. ZFP is open source and free to users.

F. Short description

The ZFP software library provides a comprehensive solution to both lossy and lossless data compression. ZFP reduces the storage space of high-precision floating-point data without sacrificing accuracy. It was designed to be a compact number format for storing data arrays in-memory in compressed form while supporting high-speed random access.

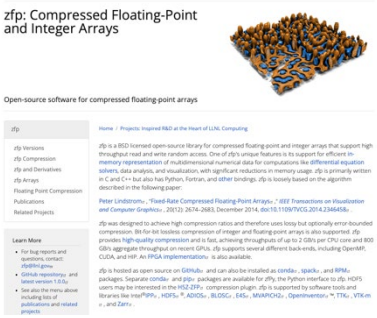
G. Type of institution represented

Government or Independent Lab/Institute

H. Submitter's relationship to product

Product developer

I. Photo



ZFP product information, documentation, and other details are available at zfp.llnl.gov.

J. Video

youtu.be/09Jl2ggiDuY

3. PRODUCT/SERVICE DESCRIPTION

A. What does the product or technology do?

ZFP [Lin14] is a new compressed number format for floating-point and integer arrays intended to reduce in-memory and offline storage and transfer time of large data sets that arise in high-performance computing (HPC). ZFP effectively expands available CPU and GPU memory by as much as 10x; reduces offline storage by one to two orders of magnitude; and—by reducing data volumes—speeds up data movement between memory and disk, distributed compute nodes, CPU and GPU memory, and even main memory and CPU registers. Such reductions in data movement are critical to today's HPC applications, whose performance is largely limited by data movement rather than compute power [SDM10, RD15].

Contrary to competing compressed formats designed for file storage, a unique feature of ZFP is that it supports high-speed, constant-time random access to array elements—both for read and write operations—suitable also for in-memory storage. This capability allows applications to work with ZFP arrays as though they were uncompressed and to substitute bulky floating-point arrays in the ubiquitous IEEE-754 format [Std19] with lean ZFP arrays with few code changes. Although ZFP supports *lossless compression* to ensure values are preserved exactly bit for bit, its primary use case is *lossy compression*, where small but usually negligible numerical errors are introduced to significantly boost compression levels. While such errors may at first seem alarming , similar rounding errors in fact occur in virtually every arithmetic operation that uses finite-precision floating-point arithmetic. In comparison with IEEE-754, ZFP increases accuracy per bit stored and supports user-specified error tolerances, allowing numerical errors to be controlled while simultaneously reducing storage and data movement.

ZFP provides C and C++ multidimensional array implementations for in-memory storage and an application programming interface (API) for compressing and decompressing entire floating-point arrays, e.g., for offline storage and data transfer. By decomposing array data into small blocks of numbers that are compressed independently, ZFP enables massive data parallelism to speed up (de)compression on multicore CPU and GPU devices. ZFP supports several back ends—serial, OpenMP, CUDA, and HIP—and programming languages—C, C++, Python, and Fortran—with additional back ends and language bindings available through third parties. The ZFP software library is available as open source on GitHub under a permissive BSD license, as is its ZHW field programmable gate array (FPGA) hardware implementation.

Compression Errors

Unlike the majority of today's lossy file compressors [ATW+18, BLP20, DC16, LI06, LLC23], quite a bit is known about the numerical errors introduced by ZFP's lossy compression modes. For instance, it is known that the ZFP error distributions are essentially normal (or Gaussian), but with finite support (i.e., errors are bounded) due to the mixing of uniform roundoff errors that occur in its decorrelating transform (see below) and the central limit theorem [Lin17]. This normality is attractive as it allows a user to reason about the propagation of such errors; e.g., the sum of two normal random variables is also normal. Moreover, the errors can be shown to be unbiased and uncorrelated [HBP+19], which is important in many physics applications to ensure conservation of mass, energy, and momentum, and in statistics.

Absolute error bounds for ZFP compressed data have been established [DFH+19], which allow scientists to set an acceptable error tolerance wherein ZFP reduces the data as much as possible while respecting the tolerance. While such error bounds are starting to become commonplace for lossy numerical compressors [ATW+18,

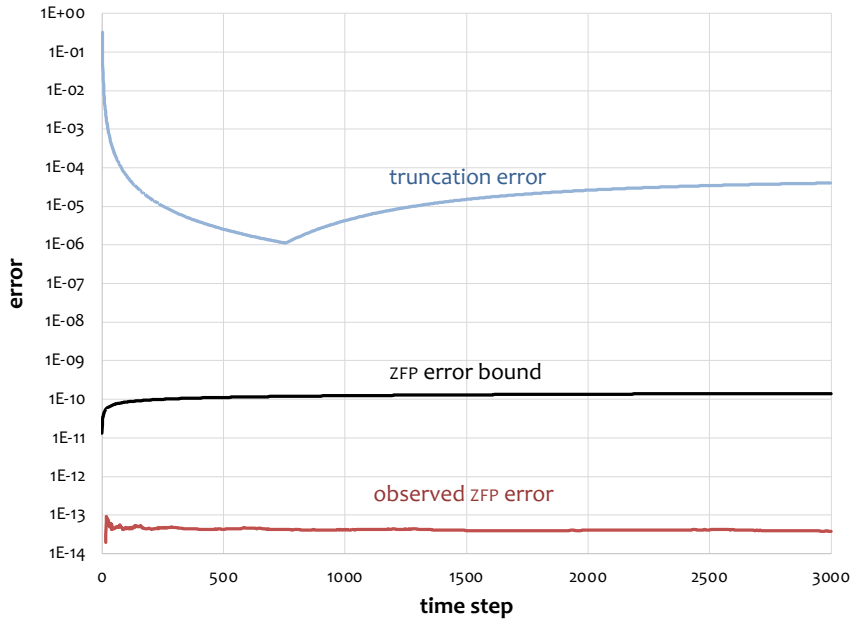


Figure 1: When ZFP is the primary representation of the evolving solution in a PDE solver, compression errors are introduced in each time step. Such errors could potentially accumulate over time and cause the solution to blow up. The ZFP team has established error bounds not just for a single application of compression but for the cumulative error over time in iterative solvers. This allows scientists to choose an appropriate compression ratio with the assurance that compression errors are far below other sources of error.

DC16, LLC23], bounds usually exist only for a single application of compression, as these compressors are primarily designed for compressing a given data set only once for offline storage. ZFP, on the other hand, is designed not only to compress numerical data but also to serve as an in-memory substitute for traditional floating-point numbers. As such, compression errors are introduced each time a piece of data is compressed, or *converted* to the ZFP format. This is of particular concern in iterative computations, e.g., partial differential equation (PDE) solvers, that use ZFP to store the evolving solution. In such scenarios, compression errors are incurred in each iteration (or time step) and may in principle cascade (Figure 1). Iterative error bounds [FDH+20], which are known only for ZFP, provide guarantees that errors do not grow uncontrollably over time and eventually "blow up."

An important stress case for lossy compressors and other reduced-precision representations is how numerical errors are magnified in "poorly conditioned" operations, such as when estimating spatial derivatives using finite differences in PDEs. By analyzing the ZFP back-end encoding scheme, one can show that in

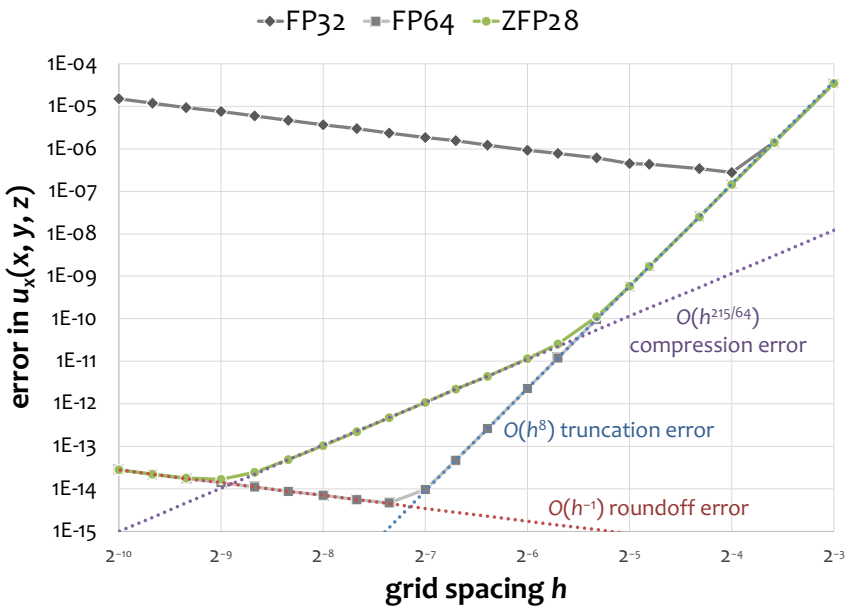


Figure 2: Rate of convergence of finite-difference derivative error as a function of grid spacing for conventional IEEE floating point (FP32, FP64) and ZFP at 28-bit storage. The total error plotted is the sum of truncation error, which decreases with finer grids, and roundoff error, which increases with finer grids. ZFP also incurs compression error (another form of roundoff error). Unlike standard number formats, ZFP's compression error decreases with finer grids and would surpass FP64 if ZFP were not first converted to FP64 for computing the finite-difference derivative estimate. At 28 bits, ZFP is up to 500 million times more accurate than 32-bit floating point.

d dimensions, the compression error in the n^{th} derivative asymptotically follows $O(h^{3d(2^{-1}-4^{-d})-n})$ as the grid spacing h approaches zero, or roughly $O(h^{4.36-n})$ in three dimensions (Figure 2). In other words, derivative accuracy *increases* rather than decreases with finer grids using ZFP, a behavior opposite that of traditional floating-point formats. In fact, ZFP accuracy would surpass even double precision (FP64) if ZFP were not first converted to FP64 for computing the finite-difference derivative estimate. We have shown that such computations can be performed directly on the partially decompressed ZFP format, further reducing errors [Lin19]. These are strong arguments for adopting ZFP in PDEs, where double precision is often needed to combat excessive rounding errors associated with single (FP32) and half (FP16) precision. The accompanying video (youtu.be/09Jl2ggiDuY) illustrates how computing on 12-bit ZFP gives qualitatively identical results to computing at full 64-bit precision, while conventional 32-bit floating-point precision is insufficient.

This result is significant: a more than 5x reduction using ZFP can be achieved with no change to the underlying application algorithm. In contrast, substantial R&D has been invested in mixed-precision algorithms [AAB+21] using FP16 and FP32 types (for 4x and 2x reduction, respectively), which in addition to explicit type declarations and conversions usually require special "tricks" such as careful rescaling to avoid over- and underflow (due to reduced dynamic range; ZFP retains the full dynamic range of FP64), extra passes of iterative refinement, and targeting special hardware like tensor cores—all of which demand nontrivial changes to the algorithm and implementation.

Another important benefit of ZFP is that it supports fine bit rate selection and, thus, a very flexible storage size. Per-value storage can be specified by the user in increments as fine as 1/32 bit. This feature effectively provides a continuous precision dial that avoids the dichotomy of the user's having to select among three discrete storage sizes (16, 32, and 64 bits) and distinct types using standard floating-point formats.

Compression Speed

One important consideration, especially for accelerating data movement via compression, is the speed of compression and decompression. That is, to realize a net performance gain in data transfers, the total time spent on compression (by the sender), transfer of compressed data, and decompression (by the receiver) must not exceed the time needed to transfer the data uncompressed. As documented extensively through numerous publications [LLH+18, LDT+19, NVB+20, LLC23, UBK+23], ZFP is one of if not *the* fastest lossy numerical compressor available. The ZFP CUDA-based GPU implementation achieves up to 700 GB/s throughput in compression and decompression, as shown in Figure 3. This is substantially faster than the throughput of I/O, supercomputer interconnects, and PCI Express for channeling data

between CPU and GPU. Several success stories of using ZFP to accelerate I/O [LCL16, TBP+21], communication [ZCK+21, ZKA+22, RZS+22], and CPU–GPU transfers [BAO19, NVB+20] have demonstrated speedups of up to 40x, 6x, and 2.2x, respectively.

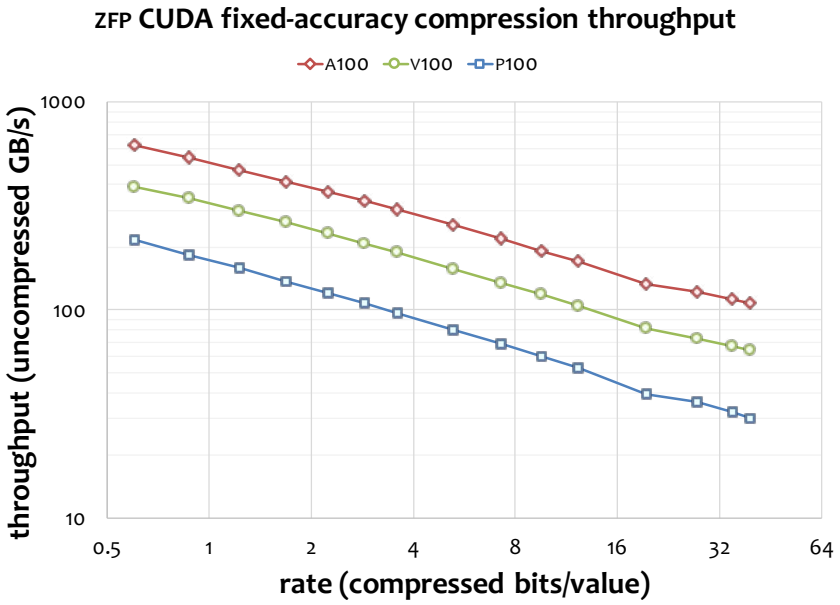


Figure 3: ZFP compression throughput on three generations of NVIDIA GPUs. Throughput varies with bit rate as compression time is linear in the number of bits output.

The relative simplicity and efficiency of the ZFP compression scheme has made it a candidate for implementation in HPC hardware, where one can entirely eliminate some operations not readily available in today's CPU instruction sets. For example, transposing a bit matrix—one of the costliest steps in ZFP—requires repeated data accesses and/or a large register file to hold intermediate results, but is done "instantly" in hardware via custom wiring. Moreover, a hardware implementation can more easily support fine-grained parallelism than one written in software, which reduces latency and boosts overall throughput. For these reasons, several independent research teams have developed FPGA hardware implementations of ZFP as a step toward on-chip compression [SJ19, SKJ20, HEE+22, LJ22, SKJ22, BWL+22], each with slight variations to further improve performance.

Community Impact

ZFP is recognized as one of the leading solutions for numerical data compression and has consequently seen widespread adoption in industry, academia, and national

labs. With over 1.5 million measurable downloads per year (from GitHub, Anaconda, and PyPI), the following features are largely responsible for ZFP's impact and rapid adoption:

- Provides unique features like random access, prescribed and guaranteed memory footprint, and fine-grained access not supported by other compressors (see comparison matrix on p.19).
- Is very effective at compressing data, competing with best of breed.
- Is the fastest floating-point compressor available.
- Is highly parallelizable by design.
- Has attractive error properties and error guarantees not available through other compressors.
- Provides C, C++, Python, and Fortran bindings, with other language bindings available through third parties (Rust, Julia, and WebAssembly, in particular).
- Supports numerous back ends, including serial, OpenMP, CUDA, and HIP, with additional implementations by Intel (for AVX support) and NEC (for their TSUBASA vector engine). Additionally, several FPGA hardware implementations have been published.
- Relies on best-practices software development using continuous integration and deployment; supports CMake and GNU make as build systems; is rigorously tested using more than 5,000 unit and functional tests; and employs code coverage analysis.
- Is well documented online via ReadTheDocs, with complete API documentation, tutorials, code examples, FAQ, and installation and troubleshooting guides. Searchable documentation exceeds 200 PDF pages.
- Is production ready with reliable performance and portability. ZFP runs and is continuously tested on Linux, macOS, and Windows. It conforms to C89 and C++98 language standards and has been built with numerous compilers (gcc, clang, Intel, XLC, PGI, MSVC, MingGW).
- Is easy to install and widely available through an array of package managers, including Spack, Anaconda, Pip, Linux RPM, and MacPorts.

- Is open source and freely available through a permissive BSD license.

As a self-contained software library with no dependencies on other software, ZFP's impact can in part be measured through adoption by customers in other high-impact commercial and academic applications. A few example applications that support ZFP include:

- ADIOS, a high-performance I/O library and past R&D 100 winner.
- BLOSC, one of the best-known frameworks for data compression.
- HDF5®, perhaps the most ubiquitous file format for science data. ZFP is one of only a handful of compressors shipped with HDF5® binaries.
- Intel® Integrated Performance Primitives and Intel® oneDTL.
- MVAPICH2, one of the leading MPI implementations of message passing.
- Open Inventor™, a commercial 3D visualization toolkit.
- OpenZGY, Schlumberger's open-source compressed format for seismic data based on the SEG-Y industry standard in the oil and gas industry. This format is based solely on ZFP.
- VTK-m, Kitware's high-performance visualization library.
- Zarr, one of the leading libraries for compressed array storage and an HDF5 competitor.

Among these applications, only ADIOS supports any of ZFP's competitors (discussed further below). Numerous other examples of ZFP customers are listed on the ZFP homepage (zfp.llnl.gov).

B. How does the product operate?

ZFP was inspired by texture compression formats [BAC96] designed for RGB images, which have long had hardware support on GPUs and mobile devices. Like ZFP, such formats partition images into small blocks (usually 4x4 pixels), each of which is compressed to a fixed number of bits. This so-called "fixed-rate compression" setup allows blocks to be quickly retrieved and decompressed on demand via random access. Unlike texture compression, which may spend a large amount of time optimizing the compression process to support fast read-only access, ZFP performance is symmetric for compression and decompression, with read and write accesses being equally fast. ZFP also has been designed for high-precision numerical data common in HPC, e.g., in PDEs, and supports 1D–4D integer and floating-point arrays.

ZFP exploits redundancy between numbers in conventional floating-point arrays, which for science and engineering applications tend to vary slowly with array index. For example, in gridded scalar fields that represent physical quantities, such as temperature and pressure in a weather model, values at adjacent grid points tend to be highly correlated. Such slow variations often cause nearby values to share the same floating-point exponent as well as several leading value bits. ZFP removes such redundancy in exponents using a *block-floating-point* representation, where the 4^d values of a d-dimensional block are expressed relative to a single common exponent. Redundancy in leading value bits is subsequently removed using a linear *decorrelating transform* similar to the discrete cosine transform used in JPEG image compression [Wal92]. This step, when followed by an integer conversion to "negabinary" (base –2), replaces without loss common leading bits with zero-bits, which can be compressed efficiently. The custom transform used by ZFP is very efficient and involves only 5 integer additions and subtractions each and 6 bit-shifts, compared to 16 multiplications and 12 additions for a naïve implementation. The final step losslessly encodes the transform coefficients by bit plane, from the most to least significant bit, and outputs a variable-length bit string whose length depends on how compressible the data is. The bit string can be truncated to any number of bits, e.g., to meet a fixed storage budget or requested error tolerance. This discarding of least significant bits is equivalent to *rounding* that already occurs in floating-point arithmetic and is the primary source of loss in accuracy. As with floating-point rounding, bit string truncation introduces a predictable and controllable level of error.

The selected ZFP "compression mode" determines how bit strings are truncated. In *fixed-rate mode*, the user specifies an exact storage size by fixing the number of compressed bits to use per block. In *fixed-precision mode*, ZFP transform coefficients

are encoded to a given number of bits of precision with the remaining bits zeroed, which results in variable-length storage. In *fixed-accuracy mode*, the user specifies an absolute error tolerance from which values may not deviate. In this use case, the number of bits of precision needed for coefficients is dictated by the value of the common block exponent relative to the error tolerance. Finally, ZFP also supports a *lossless compression mode* that leaves the bit string at its full length and ensures all remaining algorithmic steps in ZFP are fully reversible.

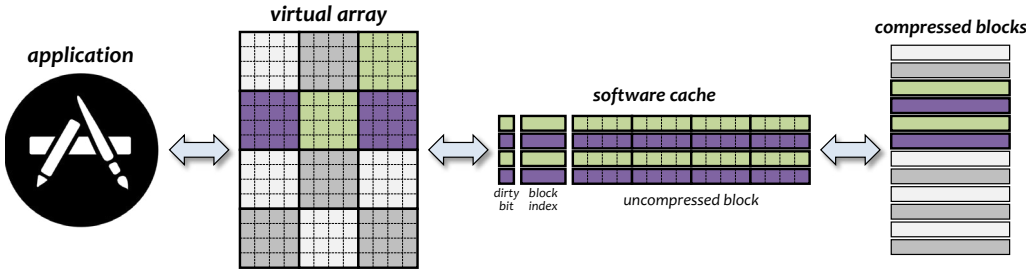


Figure 4: Conceptual illustration of a ZFP array, with persistent storage on the far right. The write-back software cache allows array accesses to be quickly serviced without (de)compressing the associated block upon each access.

ZFP provides C++ classes that implement multidimensional arrays. Using operator overloading, these arrays have a conventional API that one might expect of an array class that allows individual array elements to be referenced using standard indexing, e.g., an expression a[i][j] -= a[i + 1][j]; subtracts from a[i][j] the value of the corresponding element in the next row. Thus, the details of compression and decompression required to implement this expression are hidden from the user, allowing applications to work with ZFP arrays as though they were regular uncompressed arrays. Oftentimes, only changes to array declarations are needed for an existing application to use ZFP arrays in place of uncompressed arrays. To accelerate computations, ZFP makes use of a software cache of uncompressed blocks (stored in IEEE FP64 format) whose size is specified by the user, which avoids having to perform (de)compression each time an array element is accessed (see Figure 4).

C. Product comparison

As ZFP serves both as a general compressor of numerical data and as a number format for in-memory computations, two types of comparisons with the state of the art are made.

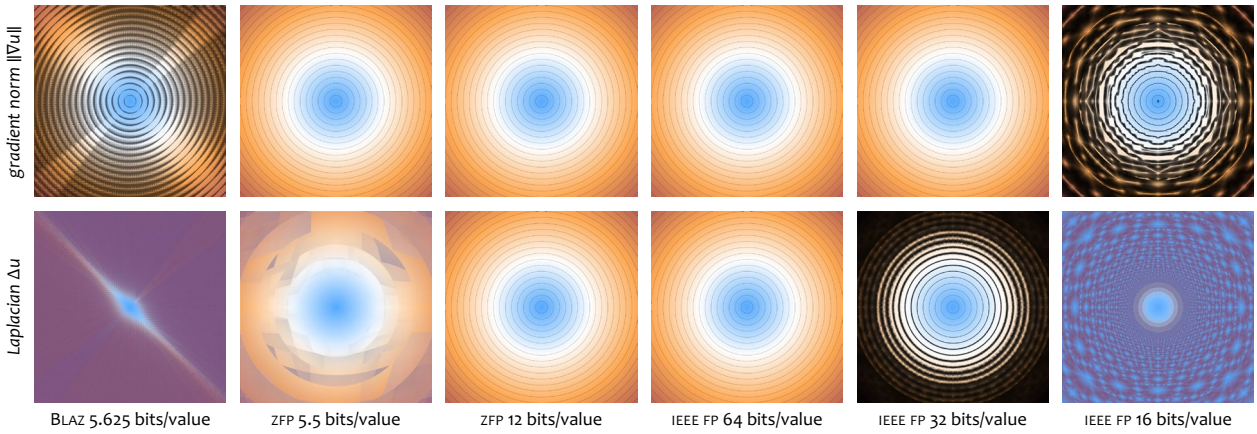


Figure 5: Visualization of first (top row) and second (bottom row) second-order finite-difference derivatives of the nonlinear function $u(\mathbf{r}) = \|\mathbf{r}\|^3$. The ideal result is concentric circular contour lines (contours are omitted when the function is poorly approximated). 12-bit ZFP is qualitatively identical to 64-bit IEEE floating point. Roundoff error in $u(\mathbf{r})$ is magnified in its estimated derivatives and visible as "smeared" contours for BLAZ, FP16, and FP32.

Comparison with Other Number Formats

One strength of ZFP is that its decorrelation of spatially correlated fields over small blocks of numbers allows repurposing redundant bits to increase numerical precision. High precision is needed, for example, to resolve differences between adjacent values when estimating derivatives using finite differences. Whereas number formats like IEEE floating point and POSITS [GY17] cannot exploit such correlations—because they store independent scalars—ZFP represents multiple values in a small block as a single unit, allowing correlations between them to be removed. Currently, the only block-based random-access format that competes with ZFP is the recently proposed BLAZ [Mar22], which supports only 2D arrays (vs. 1D–4D for ZFP) and only at a single fixed rate of 5.625 bits/value (vs. essentially any rate for ZFP).

Figure 5 illustrates the sensitivity of finite-difference derivative estimates to roundoff error in the function being differentiated (here, a radially symmetric function).

It is evident that conventional IEEE single (FP32) and half (FP16) precision are inadequate, whereas fixed-rate ZFP at only 12 bits of storage closely resembles double precision (FP64). Also evident is that, at similar bit rates, ZFP outperforms BLAZ, whose single rate proves inadequate for sensitive but common numerical computations like these. Additional comparisons, including with POSITS, are demonstrated in the accompanying video.

Comparison with Numerical Compressors

As secondary use case—a compressed format for archival storage (e.g., for storing simulation data on disk)—ZFP has several competitors. Two prominent ones that, like ZFP, are being developed with support from the U.S. Department of Energy's Exascale Computing Project are SZ [DC16] (a 2021 R&D 100 winner) and MGARD [ATW+18] (we here compare with SZ3 and MGARD 1.5—the most recent versions). We also compare with best-of-breed floating-point compressors designed to maximize compression at the expense of speed: TTHRESH [BLP20] and SPERR [LLC23]. Finally, we include in our comparison FPZIP [LI06], which is known to be one of the best lossless compressors of multidimensional floating-point data (FPZIP also supports lossy compression, but it is not its strength).

To evaluate effectiveness of lossy compression, one needs to account not only for compression ratio but also for the corresponding error introduced. This is usually done via *rate-distortion plots*, e.g., using signal-to-noise ratio ($SNR = 10\ \log_{10}\frac{\sigma}{E}$, where σ denotes standard deviation and E denotes root-mean-square error) vs. bit rate, R (in compressed bits of storage per array value). Like [LLC23], we favor accuracy gain plots ($gain = \log_2\frac{\sigma}{E} - R$) as they reveal important behaviors usually not visible in SNR plots. Accuracy gain directly tells how many bits of information were inferred by a compressor, i.e., the per-value savings in storage, and indicates when no more compression is possible when reaching a plateau where halving the error incurs one more bit of storage.

Figure 6 plots accuracy gain vs. rate over a range of error tolerances that have been successively halved for each data point. As is evident by the green lines, ZFP achieves close to the highest gains and is consistently outperformed in this use case only by SPERR. However, SPERR and TTHRESH are far more computationally expensive than ZFP, with implementations that are one to two orders of magnitude slower on CPU hardware, and several more orders of magnitude slower when compared with ZFP's GPU implementation. Also evident from this figure is how the ZFP curves vary rather smoothly and predictably in stark contrast to the wobbly SZ and MGARD curves, which occasionally have large gaps in rate or precipitous drops in accuracy gain to negative levels, indicating that the data is expanded rather than compressed. Such

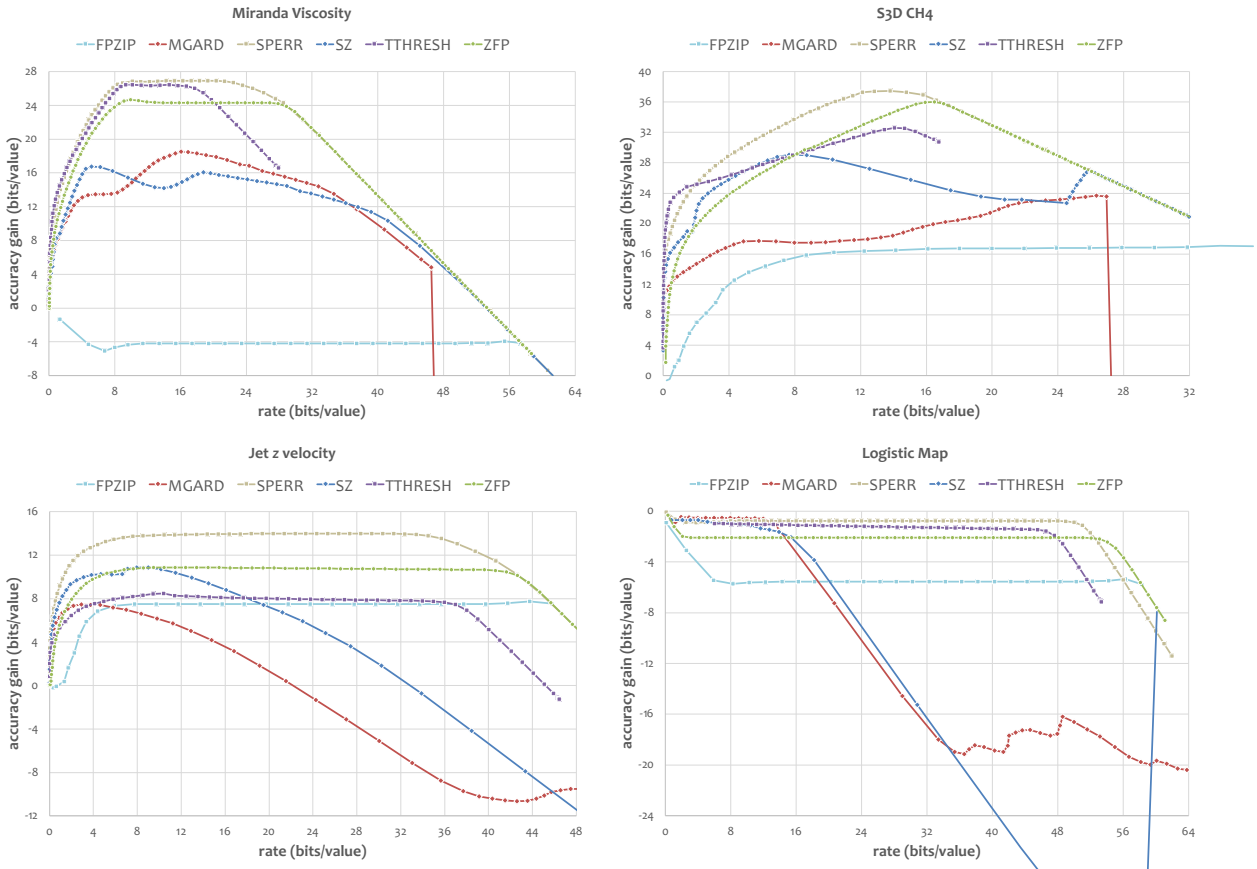


Figure 6: Accuracy gain (higher is better) as a function of bit rate (Miranda and S3D are from SDRBench [ZDL+20]). These plots highlight ZFP's predictable accuracy and rate as the error tolerance is halved for each data point. Such reliability and high performance give scientists confidence that small changes in compression parameters incur small changes in error and storage, while also being essential for error and convergence analysis.

kinks in the plots imply not only unpredictable storage and error that are sensitive to small changes in input parameters (such as error tolerance) but can also lead to unpredictable running times. Moreover, ZFP consistently provides higher accuracy than SZ and MGARD at mid to high rates—a range important for numerical data analysis, which demands more accuracy than, for example, visualization.

ZFP owes its predictable relationship between compressor parameters, bit rate, and error to excellent decorrelation that removes most data redundancy, and to a simple though efficient nonstatistical encoder that, unlike SZ and MGARD, does not have complex data dependencies from the use of statistical coders. Predictable and continuous response to input parameters avoids unpleasant surprises for end users, allows extrapolating compressor inputs to obtain a desired storage size or quality,

and significantly helps with error analysis such as convergence rate. Further evidence of ZFP's superior predictable performance and robustness of implementation is presented in a recent paper by one of ZFP's main competitors, who further concluded that, among several compressors evaluated, "Only ZFP's GPU implementation ran without crashes on our entire testing sets" [UBK+23]. This robustness comes from ZFP's careful design, focus on portability, and extensive unit tests.

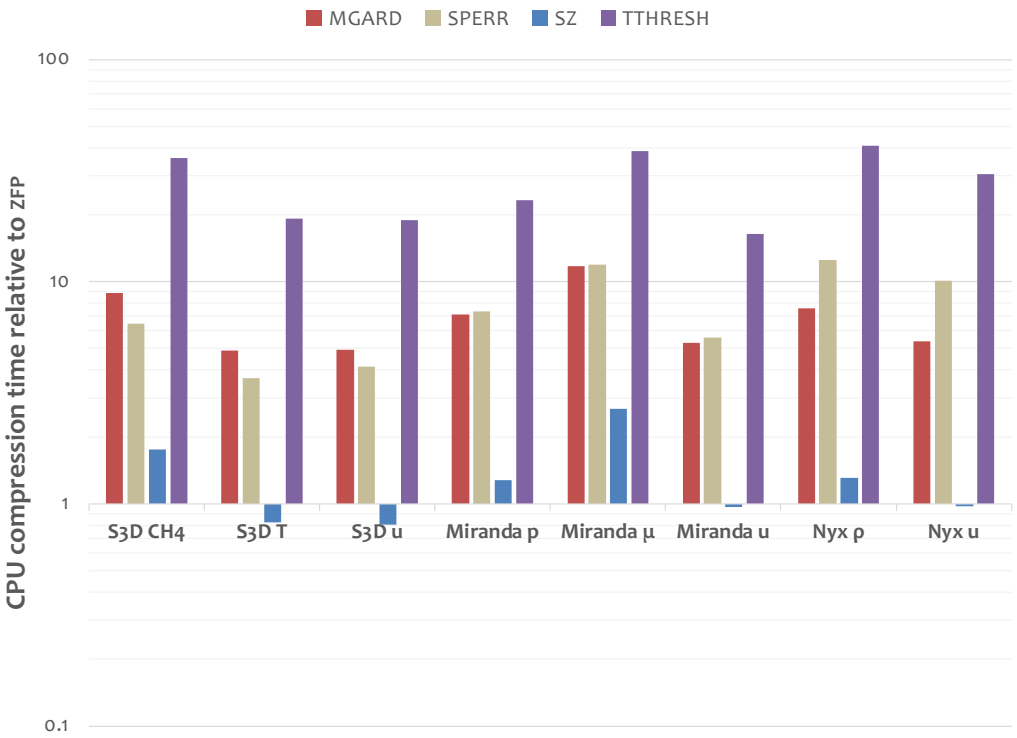


Figure 7: Comparison of serial CPU compression time between compressors for various SDRBench data sets. The time is reported as a ratio to ZFP compression time.

As Figure 7 shows, ZFP single-core CPU compression speed is on average 33% higher than sz's, with MGARD and SPERR being roughly 7–8x slower and TTHRESH clocking in at 28x slower than ZFP on average. Note that SPERR and TTHRESH do not have GPU implementations. ZFP has a particularly fast GPU implementation due to the massive data parallelism its block decomposition offers, resulting in a speedup of approximately 400x over its serial CPU implementation.

While compressors like SPERR, MGARD, and TTHRESH may be suitable for I/O, they lack the speed needed for more time-sensitive use cases, such as node-to-node communication and host-device transfers. Furthermore, no compressor other than ZFP supports random access or the level of fine-grained compression needed for on-demand data access. When tasked with compressing a selection of 4 x 4 x 4 blocks to the same level of accuracy, ZFP uses 10x less storage than the second-best compressor (SPERR) and as much as 70,000x less storage than the compressor with highest overhead (MGARD) (Figure 8). Not until blocks are enlarged to 16 x 16 x 16 do other compressors become competitive, at which point a decompressed block is 32 KB, effectively exhausting the entire L1 data cache. In contrast, a 16 bits/value compressed ZFP block ranges from 64 bits (1D) to 1024 bits (3D), essentially occupying one processor register to 1–2 cache lines; uncompressed blocks are 4x larger. As such, only ZFP is suitable for in-memory compression.

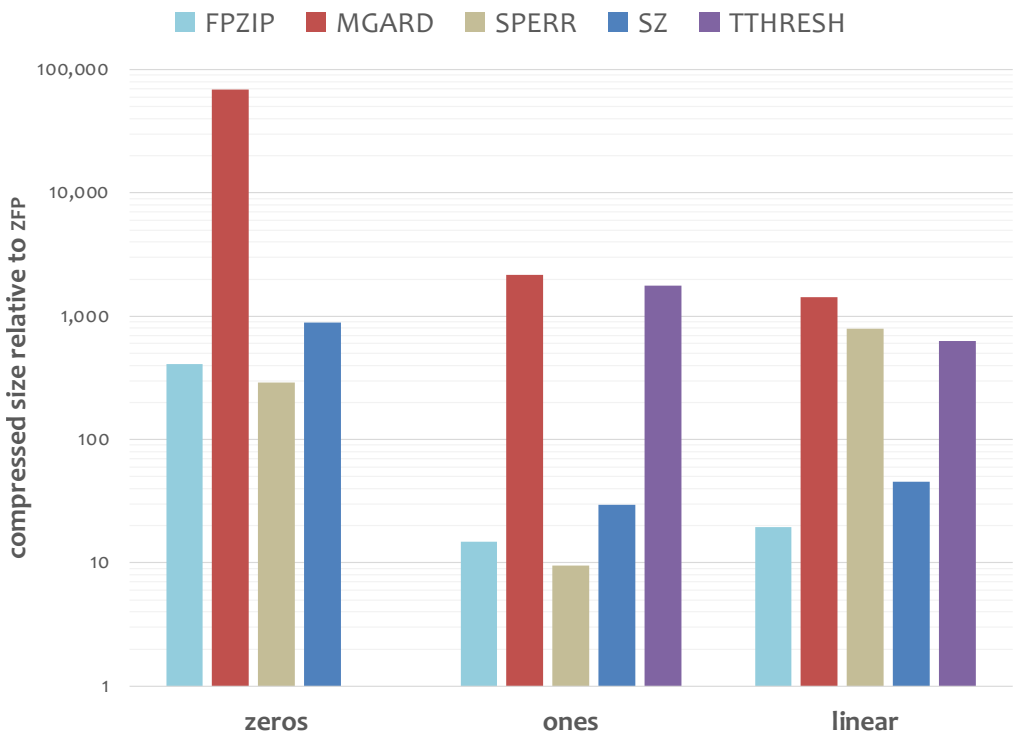


Figure 8: Storage overhead compared to ZFP for compressing a single 4 x 4 x 4 block.

D. Comparison Summary

Figure 9 summarizes the differences in features supported by ZFP and its competitors, as well as various metrics related to performance, accuracy, availability, and overall community impact. As is evident from this matrix, ZFP is not only one of the leading file compressors in terms of compression ratio, speed, and desirable and known error characteristics, but its support for in-memory compression with random access and a prescribed storage size makes it uniquely suitable for both offline and in-memory number format. Its high speed also accommodates use cases for accelerating data movement across the entire memory hierarchy of a computer that are not possible with other compressors. It is also evident in this matrix that the early research prototype BLAZ lacks both the features and flexibility in user-selected bit rate needed to achieve sufficient numerical accuracy and, thus, seriously contend with ZFP as compressed number format. Finally, in terms of community impact and adoption, ZFP is in a class by itself. In fact, we are unaware of any major applications of general utility that support one of our competitors but not ZFP. For a partial list of customers, see the ZFP website: zfp.llnl.gov.

Product Limitations

The ZFP algorithm and library have some limitations, some of which are intrinsic and some which we expect to address in future versions:

- The fine-grained array partitioning into blocks benefits parallelism and random access but places a limit on achievable compression ratios, as a minimal amount of data (e.g., block exponent and some leading values bits) must be stored with every block.

- ZFP trades some compression for performance and support for random access, allowing some compressors to achieve higher compression ratios, albeit at higher computational expense and with support for sequential access only.

- Individual scalars cannot be loaded/stored without loading/storing the blocks they belong to. This limits granularity of access somewhat and introduces overhead when only a single scalar in a block is needed.

- ZFP does not interface with prebuilt libraries, such as BLAS (not to be confused with the BLAZ data compressor) and LAPACK, that process arrays via floating-point pointers (e.g., double*). While ZFP provides proxy pointers that act like regular pointers, applications that explicitly pass data via pointers must be instrumented and recompiled to take advantage of such proxies.

- Hardware compression is bottlenecked by ZFP's back-end encoder and decoder, which often operate on bits one at a time. As explored in several FPGA adaptations of ZFP [HEE+22, LJ22, SJ19, SKJ20, SKJ22], one may trade compression ratio for a simpler but more performant coder.

		ZFP	BLAZ	SZ	MGARD	SPERR	TTHRESH	FPZIP	Notes
Core features	Data dimensions	1D-4D	2D	1D-4D	1D-5D	2D-3D	3D-4D	1D-3D	
	Floating-point types	✓	✓	✓	✓	✓	✓	✓	
	Integer types	✓		✓			✓		
	Random access	✓	✓						
	Fixed rate	✓	✓			✓	✓		
	User-specified rate	✓				✓	✓		
	Progressive access	✓			✓	✓	✓		
	Lossless compression	✓						✓	
	Array classes	✓	✓*						* Rudimentary C support only
Speed & backends	Speed	++	++	+	–	–	––		
	Parallelism/scalability	++	++	+	+	–	––	––	
	Predictable performance	++	++	–	++	+	+	+	
	OpenMP	✓		✓	✓	✓	*		
	CUDA	✓		✓*					* SZ2 only
	HIP	✓*			✓†				* Development branch only; † MGARD-X only
	AVX	✓*		✓					* Intel 3rd party implementation
	FPGA	✓*		✓					* Available as LLNL/zhw GitHub repository
	NEC VE	✓*							* NEC 3rd party implementation
Compression errors	Error distribution	normal	unknown	uniform	unknown	unknown	unknown	relative	
	Absolute error bound	✓		✓	✓	✓			
	Relative error bound	✓		✓	✓			✓	
	Iterative error bound	✓							
	Convergence theory	✓							
	Predictable quality	++	N/A	–	–	++	+	++	
Languages	Command line interface	✓		✓	✓	✓	✓	✓	
	C	✓	✓	✓		✓		✓	
	C++	✓		✓	✓	✓			
	Python	✓		✓					
	Fortran	✓		✓*					* SZ2 only
	Julia	✓*							* 3rd party implementation
	Rust	✓*							* 3rd party implementation

		ZFP	BLAZ	SZ	MGARD	SPERR	TTHRESH	FPZIP	Notes
Package distribution	Spack	✓		✓	✓	✓		✓	
	Anaconda	✓		✓				✓	
	Pip (PyPI)	✓							
	MacPorts	✓					✓		
	E4S	✓		✓					
	RPM (distros supported)	45		16	16			16	
I/O library support	HDF5	✓*		✓	✓				* Also bundled with HDF5 binary distribution
	ADIOS2	✓		✓	✓				
	MDIO	✓							
	BLOSC	✓							
	OpenZGY	✓							
	Zarr	✓*							* Via numcodecs
Metrics	GitHub stars	616	5	145*	23	10	38	75	* SZ2 + SZ3
	Conda downloads	>2 million	–	238*	–	–	–	30	* SZ2 + SZ3
	Google Scholar citations	528	0	411	67	0	98	497	
	GitHub dependents	31	0	0*	0	0	0	0	* SZ2 + SZ3

Figure 9: Comparison matrix between ZFP, its primary competing number format, BLAZ [Mar22], and state-of-the-art floating-point file compressors SZ [DC16], MGARD [ATW+18], SPERR [LLC23], TTHRESH [BLP20], and FPZIP [LI06].

4. SUMMARY

ZFP accelerates HPC calculations and lowers cost by dramatically reducing memory and disk footprint, bandwidth requirements, and power usage. ZFP is a compressed and highly efficient number format offering a unique alternative to conventional floating point in HPC applications. ZFP is suitable for in-memory, in-transit, and offline storage of correlated multidimensional array data, and offers significant reduction of data volumes for accelerating and enabling memory capacity- and bandwidth-limited computations. Additionally, ZFP increases accuracy of many numerically sensitive computations over alternative number formats and provides error controls that are intuitive to application scientists, who can easily substitute floating-point array data structures with ZFP to achieve exact reductions in storage or levels of accuracy. ZFP is highly performant and scalable through massive data parallelism. It runs on CPU, GPU, and accelerator hardware, and is available on Linux, macOS, and Windows through language bindings such as C, C++, Python, and Fortran. Freely available as open source, ZFP has become a de facto standard compression solution in the HPC community, with broad adoption in dozens of important commercial and academic applications.

5. REFERENCES

[AAB+21] A. Abdelfattah, H. Anzt, E. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. Higham, X. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. Tsai, U. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," International Journal on High Performance Computing Applications, 35(4):344–369, 2021, doi:10.1177/10943420211003313.

[ATW+18] M. Ainsworth, O. Tugluk, B. Whitney, S. Klasky, "Multilevel techniques for compression and reduction of scientific data-the univariate case," Computing and Visualization in Science, 19:65–76, 2018, doi:10.1007/s00791-018-00303-9. Source code: github.com/CODARcode/MGARD.

[BAC96] A. Beers, M. Agrawala, N. Chaddha, "Rendering from Compressed Textures," ACM SIGGRAPH 96, doi:10.1145/237170.237276.

[BAO19] M. Bamakhrama, A. Arrizabalaga, F. Overman, J.-P. Smeets, K. van der Sommen, R. van der Vossen, J. Wagensveld, "GPU Acceleration of Real-Time Control Loops," arXiv:1902.08018v1, 2019, doi:10.48550/arXiv.1902.08018.

[BLP20] R. Ballester-Ripoll, P. Lindstrom, R. Pajarola, "TTHRESH: Tensor Compression for Multidimensional Visual Data," IEEE Transactions on Visualization and Computer Graphics, 26(9):2891–2903, 2020, doi:10.1109/TVCG.2019.2904063. Source code: github.com/rballester/tthresh.

[BWL+22] M. Barrow, Z. Wu, S. Lloyd, M. Gokhale, H. Patel, P. Lindstrom, "ZHW: A Numerical CODEC for Big Data Scientific Computation," FPT 2022, doi:10.1109/ICFPT56656.2022.9974258. Source code: github.com/LLNL/zhw.

[DC16] S. Di, F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," IEEE IPDPS 2016, doi:10.1109/IPDPS.2016.11. Source code: github.com/szcompressor/SZ3.

[DFH+19] J. Diffenderfer, A. Fox, J. Hittinger, G. Sanders, P. Lindstrom, "Error Analysis of ZFP Compression for Floating-Point Data," SIAM Journal on Scientific Computing, 41(3):A1867–A1898, 2019, doi:10.1137/18M1168832.

[FDH+20] A. Fox, J. Diffenderfer, J. Hittinger G. Sanders, P. Lindstrom, "Stability Analysis of Inline ZFP Compression for Floating-Point Data in Iterative Methods," SIAM Journal on Scientific Computing, 42(5):A2701–A2730, 2020, doi:10.1137/19M126904X.

[GY17] J. Gustafson, I. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," Supercomputing Frontiers and Innovations, 4(2):71–86, 2017, doi:10.14529/jsfi170206.

[HBP+19] D. Hammerling, A. Baker, A. Pinard, P. Lindstrom, "A Collaborative Effort to Improve Lossy Compression Methods for Climate Data," IEEE DRBSD-5, 2019, 10.1109/DRBSD-549595.2019.00008.

[HEE+22] M. Habboush, A. H. El-Maleh, M. E. Elrabaa, S. AlSaleh, "DE-ZFP: An FPGA implementation of a modified ZFP compression/decompression algorithm," Microprocessors and Microsystems, 90:104453, 2022, doi:10.1016/j.micpro.2022.104453.

[LCL16] P. Lindstrom, P. Chen, E.-J. Lee, "Reducing disk storage of full-3D seismic waveform tomography (F3DT) through lossy online compression," Computers & Geosciences, 93:45–54, 2016, doi:10.1016/j.cageo.2016.04.009.

[LDT+19] X. Liang, S. Di, D. Tao, S. Li, B. Nicolae, Z. Chen, F. Cappello, "Improving Performance of Data Dumping with Lossy Compression for Scientific Simulation," IEEE CLUSTER 2019, doi:10.1109/CLUSTER.2019.8891037.

[LI06] P. Lindstrom, M. Isenburg, "Fast and Efficient Compression of Floating-Point Data," IEEE Transactions on Visualization and Computer Graphics, 12(5):1245–1250, 2006, doi:10.1109/TVCG.2006.143. Source code: github.com/LLNL/fpzip.

[Lin14] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," IEEE Transactions on Visualization and Computer Graphics, 20(12):2674–2683, 2014, doi:10.1109/TVCG.2014.2346458. Source code: github.com/LLNL/zfp.

[Lin17] P. Lindstrom, "Error Distributions of Lossy Floating-Point Compressors," JSM 2017 Proceedings. URL: osti.gov/biblio/1526183.

[Lin19] P. Lindstrom, "Compressed Numerics to Reduce Data Movement in Numerical Simulations," LDRD project final report. URL: ldrd-annual.llnl.gov/archives/ldrd-annual-2018/computing/fs/18-FS-018.

[LJ22] S.-M. Lim, S.-W. Jun, "Mobile nets can be lossily compressed: Neural network compression for embedded accelerators," Electronics, 11(6):858, 2022, doi:10.3390/electronics11060858.

[LLC23] S. Li, P. Lindstrom, J. Clyne, "Lossy Scientific Data Compression With SPERR," IPDPS 2023, doi:10.1109/IPDPS54959.2023.00104. Source code: github.com/NCAR/SPERR.

[LLH+18] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, Z. Qiao, "Understanding and Modeling Lossy Compression Schemes on HPC Scientific Data," IEEE IPDPS 2018, doi:10.1109/IPDPS.2018.00044.

[Mar22] M. Martel, "Compressed Matrix Computations," IEEE/ACM BDCAT 2022, doi:10.1109/BDCAT56447.2022.00016. Source code: github.com/mmartel66/blaz.

[NVB+20] L. Noordsij, S. van der Vlugt, M. Bamakhrama, Z. Al-Ars, P. Lindstrom, "Parallelization of Variable Rate Decompression through Metadata," Euromicro PDP 2020, doi:10.1109/PDP50117.2020.00045.

[RD15] D. Reed, J. Dongarra, "Exascale computing and big data," Communications of the ACM, 58(7):56–68, 2015, doi:10.1145/2699414.

[RZS+22] B. Ramesh, Q. Zhou, A. Shafi, M. Abduljabbar, H. Subramoni, D. Panda, "Designing Efficient Pipelined Communication Schemes using Compression in MPI Libraries," IEEE HiPC 2022, doi:10.1109/HiPC56025.2022.00024.

[SDM10] J. Shalf, S. Dosanjh, J. Morrison, "Exascale computing technology challenges," International Conference on High Performance Computing 2010, doi:10.1007/978-3-642-19328-6_1.

[SJ19] G. Sun, S.-W. Jun, "ZFP-V: Hardware-optimized lossy floating point compression," ICFPT 2019, doi:10.1109/ICFPT47387.2019.00022.

[SKJ20] G. Sun, S. Kang, S.-W. Jun, "BurstZ: a bandwidth-efficient scientific computing accelerator platform for large-scale data," ACM/IEEE SC 2020, doi:10.1145/3392717.3392746.

[SKJ22] G. Sun, S. Kang, S.-W. Jun, "BurstZ+: Eliminating the communication bottleneck of scientific computing accelerators via accelerated compression," ACM Transactions on Reconfigurable Technology and Systems, 15(2):1–34, 2022, doi:10.1145/3476831.

[Std19] IEEE Std 754-2019: IEEE Standard for Floating-Point Arithmetic, 2019, doi:10.1109/IEEESTD.2019.8766229.

[TBP+21] H. Tang, S. Byna, A. Petersson, D. McCallen, "Tuning Parallel Data Compression and I/O for Large-scale Earthquake Simulation," IEEE Big Data 2021, doi:10.1109/bigdata52589.2021.9671876.

[UBK+23] R. Underwood, J. Bessac, D. Krakowska, J. Calhoun, S. Di, F. Cappello, "Black-Box Statistical Prediction of Lossy Compression Ratios for Scientific Data," arXiv:2305.08801, 2023, doi:10.48550/ arXiv.2305.08801.

[Wal92] G. Wallace, "The JPEG still picture compression standard," IEEE Transactions on Consumer Electronics, 38(1):xciii–xxxiv, 1992, doi:10.1109/30.125072.

[ZCK+21] Q. Zhou, C. Chu, N. Kumar, P. Kousha, S. Ghazimirsaeed, H. Subramoni, D. Panda, "Designing High-Performance MPI Libraries with on-the-fly Compression for Modern GPU Clusters," IEEE IPDPS 2021, doi:10.1109/IPDPS49936.2021.00053.

[ZDL+20] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, F. Cappello, "SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors," IEEE Big Data 2020, doi:10.1109/BigData50022.2020.9378449. URL: sdrbench.github.io.

[ZKA+22] Q. Zhou, P. Kousha, Q. Anthony, K. Khorassani, A. Shafi, H. Subramoni, D. Panda, "Accelerating MPI All-to-All Communication with Online Compression on Modern GPU Clusters," ISC High Performance 2022, doi:10.1007/978-3-031-07312-0_1.

6. ADDITIONAL SUPPORTING INFORMATION

Video: youtu.be/09Jl2ggiDuY

Documentation:

Website: zfp.llnl.gov

User documentation: zfp.readthedocs.io/en/release1.0.0/

Open-source code: github.com/LLNL/zfp

7. AFFIRMATION

I/we certify that all of the information within this submission entry is accurate and represents the most up-to-date information available for this entry.

Signature

June 1, 2023

Date

8. CONTACTS

Principal investigator:

Peter Lindstrom
Computer Scientist
Lawrence Livermore National Laboratory
pl@llnl.gov
925.423.5925

Development team:

Danielle Asher
Computer Scientist
Lawrence Livermore National Laboratory
morrison32@llnl.gov
925.423.8352

Matthew Larsen
Developer
Former Lawrence Livermore National Laboratory Employee
larsen.matt1@gmail.com
530.902.1033

Markus Salasoo
Senior Software Engineer
Former Lawrence Livermore National Laboratory Employee
salasoom@gmail.com
518.698.6515

Stephen Herbein
Senior Systems Software Engineer
Former Lawrence Livermore National Laboratory Employee
stephen@herbein.net
302.893.8577

Mark Miller
Computer Scientist
Lawrence Livermore National Laboratory
miller86@llnl.gov
925.423.5901

Media and marketing contact:

Mary Holden-Sanchez
Business Development and Marketing Associate
Lawrence Livermore National Laboratory
holdensanchez2@llnl.gov
925.422.4614

Person who will handle R&D 100 Awards Event arrangements:

Peter Lindstrom
Computer Scientist
Lawrence Livermore National Laboratory
pl@llnl.gov
925.423.5925

Organization's LinkedIn profile URL:
linkedin.com/company/lawrence-livermore-national-laboratory

Organization's Twitter handle:
twitter.com/Livermore_Lab
twitter.com/Livermore_Comp

Organization's Facebook page URL:
facebook.com/livermore.lab

Additional social media URLs for your organization
instagram.com/livermore_lab