LLNL-JRNL-652576

Fixed-Rate Compressed Floating-Point Arrays

P. Lindstrom

April 2, 2014

IEEE Transactions on Visualization and Computer Graphics

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Fixed-Rate Compressed Floating-Point Arrays

Peter Lindstrom, *Senior Member, IEEE*

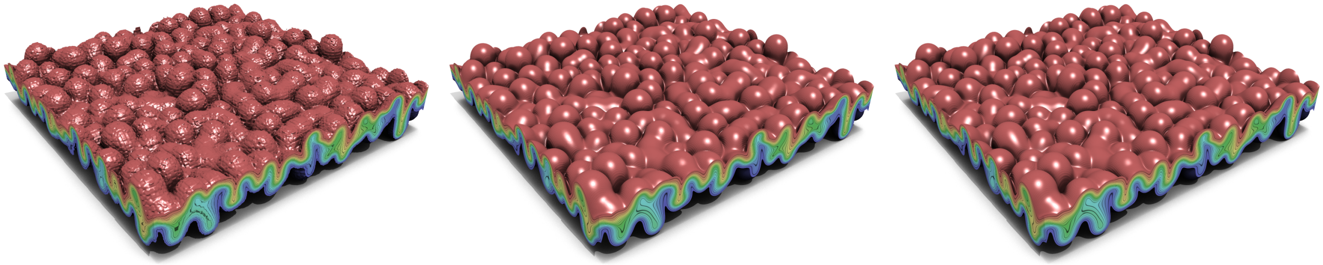

(a) 1 bit/double (b) 4 bits/double (c) 64 bits/double (no compression)

Fig. 1: Interval volume renderings of compressed double-precision floating-point data on a 384 × 384 × 256 grid. At 4 bits/double (16x compression) the image is visually indistinguishable from full 64-bit precision.

Abstract—Current compression schemes for floating-point data commonly take fixed-precision values and compress them to a variable-length bit stream, complicating memory management and random access. We present a fixed-rate, near-lossless compression scheme that maps small blocks of 4^d values in d dimensions to a fixed, user-specified number of bits per block, thereby allowing read and write random access to compressed floating-point data at block granularity. Our approach is inspired by fixed-rate texture compression methods widely adopted in graphics hardware, but has been tailored to the high dynamic range and precision demands of scientific applications. Our compressor is based on a new, lifted, orthogonal block transform and embedded coding, allowing each per-block bit stream to be truncated at any point if desired, thus facilitating bit rate selection using a single compression scheme. To avoid compression or decompression upon every data access, we employ a software write-back cache of uncompressed blocks. Our compressor has been designed with computational simplicity and speed in mind to allow for the possibility of a hardware implementation, and uses only a small number of fixed-point arithmetic operations per compressed value. We demonstrate the viability and benefits of lossy compression in several applications, including visualization, quantitative data analysis, and numerical simulation.

Index Terms—Data compression, floating-point arrays, orthogonal block transform, embedded coding.

1 Introduction

Current trends in high-performance computing point to an exponential increase in core count and commensurate decrease in memory bandwidth per core. Similar bandwidth shortages are already observed for I/O, inter-node communication, and between CPU and GPU memory. This trend suggests that the performance of future computations will be dictated in large part by the amount of data movement. Moreover, with large data sets often being generated remotely, e.g. on shared compute clusters or in the cloud, the cost of transferring the results of the computation for visual exploration, quantitative analysis, and archival storage can be substantial.

This increase in compute power has also led to a new challenge in visualization: with insufficient I/O bandwidth to store the simulation results at high enough temporal or spatial fidelity for off-line analysis, *in situ* visualization is needed that runs in tandem with the simulation. Here the visualization has to compete with the simulation for the same memory and bandwidth resources, putting further strain on the system.

One approach to alleviating this data movement bottleneck is to remove any redundancy in the data, e.g. using data compression. With abundant compute power at our disposal, using otherwise wasted compute cycles to compress the data makes sense if it can be done quickly enough to feed the compute-starved cores. However, for scientific applications that predominantly work with large arrays of floating-point numbers, lossless compression affords only modest data reductions.

Lossy compression has long been accepted in computer graphics, for instance for reduced storage of textures, and dedicated hardware for texture *de*-compression is now common in GPUs and mobile devices [1]. Such compressed formats and related efforts in visualization on rendering from compressed storage [12, 15, 41] have primarily been motivated from the standpoint of preserving visual fidelity, whereas quantitative analysis and numerical simulation place stricter requirements on tolerable errors. Moreover, these techniques tend to be highly asymmetric, with compression speed sacrificed in favor of as-fast-as-possible decompression. However, many tasks in visualization require online computation of derived fields, while simulation evolves fields over time. Both are uses cases where compressed reads cannot happen without prior compressed writes.

The goal of this work is to develop a compressed floating-point array primitive, analogous to compressed textures, that supports fast compression and decompression, but tailored to the high-precision, numerical data common in science and engineering applications. To be effective, we require lossy compression. We note that while this may seem a controversial proposition, there is nothing "magic" about the 64-bit precision currently used. As we will show, many visualization and simulation tasks can cope with far less precision—and even fewer bits per value, by using compression.

A major obstacle to be cleared first is the support for random access. Current lossy compression methods have primarily been designed to produce the shortest bit stream possible using variable-length coding, and as a result do not easily handle random access, especially when the compressed stream has to be updated whenever the data is modified. Even methods like Samplify's APAX compressor [48] that have some support for fixed rate achieve this only via a feedback loop that periodically adapts the level of compression, and likewise don't support random access.

- *Peter Lindstrom is with the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory. E-mail: pl@llnl.gov.*

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

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

To respond to these needs, we present a fixed-rate compression scheme that supports random read and write access to d-dimensional floating-point arrays at the granularity of small blocks of 4^d values (e.g. $4 \times 4 \times 4$ values in 3D). Our compressed array primitive is wrapped by a C++ interface that to the user appears like a regular linear or multidimensional array, and which uses a software cache, whose size can be specified by the user, under the hood to limit the frequency of compression and decompression. The resulting C++ class can be substituted into applications very easily with minimal code changes, thus reducing the memory footprint and bandwidth consumption, or increasing the size of arrays that can fit in memory. We show that while currently executed in software, our compressed array incurs only a minor performance penalty in visualization applications.

Although the main ingredients of our compressor—orthogonal block transform and embedded coding—are not new ideas, each stage of the compressor has been carefully designed to achive high speed, implementation simplicity, symmetric performance, SNR scalability, fine bit rate granularity, and the level of quality demanded by applications that process single- or double-precision data, such as quantitative data analysis, visualization, and numerical simulation. Moreover, our design is simple in order to accommodate a hardware implementation. At up to 400 MB/s throughput, our software codec is an order of magnitude slower than per-core memory bandwidth, but virtually instant in relation to current I/O rates. Part of its speed is due to a new, simplified block transform that has an elegant lifted implementation, and which we believe will have applications beyond floating-point compression.

To our knowledge, this is the first solution that serves as a general-purpose compressed representation of floating-point arrays, that supports both efficient, fine-grained read and write access enabled by fixed-rate compression, and that allows the user to specify the exact number of bits to allocate to each array.

2 Previous Work

We here review current state of the art in compression of floating-point data, as well as related work on fixed-rate compression of images.

2.1 Lossless floating-point compression

The majority of work on compression of double-precision data has focused on *lossless* compression, justifiably so as such precision is most often used in situations that demand accuracy. With few exceptions, these methods use linear prediction and encode the smaller residuals using some variant of non-statistical [5,9,39] or statistical [14,21,30] variable-length codes (e.g. entropy codes). The methods by Burtscher and co-workers [5,39] are notable in that they rely on hash functions to extract non-linear relationships. The issue of what makes a good predictor has been further explored by Fout and Ma [13]. Although important in many applications, lossless methods rarely achieve more than 1.5x compression on double-precision data, and therefore have only limited impact on bandwidth reduction.

2.2 Lossy floating-point compression

The idea of using compression to effectively increase bandwidth [49] and the amount of data that can be stored in memory for visualization is at least two decades old [14, 35, 45]. Like the pioneering work of Ning and Hesselink [35], Schneider and Westermann [41] proposed a lossy compression method based on vector quantization (VQ) to render volume data directly from compressed storage. While supporting random access reads, generating good VQ codebooks on the fly can be expensive, and VQ is not easily amenable to rate control. Several related efforts have focused on volume rendering from compressed storage [11, 15, 31], primarily with the goal of minimizing the perceptual error in rendered images, and with the assumption of compress-once, read-only access. These approaches are predominantly asymmetric, with fast decompression but slow compression [12]. Our scheme bears some resemblance to the DCT-based coders of Yeo and Liu [50] and Laurance and Monro [28], but uses smaller blocks for more fine-grained access and a more efficient transform and coding scheme, while also supporting fixed-rate coding to enable random-access writes.

The majority of these prior methods have proven effective in volume rendering applications by exploiting the data access pattern (e.g. as slices), limited data precision (e.g. 8 bits), an anticipated transfer function, or the need for perceptually but not necessarily numerically accurate results [8]. It is, however, unknown how these methods would fare on nonvisual, quantitative tasks other than volume rendering. Indeed, one contribution of our paper is such an evaluation of lossy compression on analysis and simulation tasks.

Compression has also been recognized within the high-performance computing community as a potential way of reducing data movement, e.g. for accelerating I/O and communication, but even for reducing memory bus traffic within simulations [2,3,19,22,25,27]. This trend is notable, as computational scientists are warming up to the prospect of using lossy compression, not only for visualization and data analysis, but also on the simulation state itself.

In the recent study by Laney et al. [27], the simulation state is stored compressed and is then decompressed in its entirety at the beginning of each time step to simulate the effects of inline compression. Such an approach does not reduce memory bandwidth, however, unless the entire uncompressed state fits in cache. In this paper we consider a tighter integration of compression, where the state is decompressed piecemeal on demand to a small cache, and is possibly written back to persistent compressed storage once or multiple times per time step.

2.3 Image, Texture, and Buffer Compression

A substantial body of work exists on compression of images, textures, and GPU buffers. Many of today's image formats represent a spectrum of compression techniques that we could draw upon for encoding 2D and 3D arrays. For instance, the GIF format uses vector quantization on RGB color vectors; PNG and JPEG-LS use linear prediction; JPEG, like our solution, uses block transform coding; JPEG XR relies on lapped transforms; while JPEG2000 uses higher-order wavelets. These formats also represent a progression of increasing complexity and quality. We find lapped transforms [32], in which the basis functions extend across block boundaries, unsuitable since they preclude blocks from being compressed and decompressed independently. For the same reasons, wavelets other than the Haar basis have wide stencils and cascading data dependencies at coarser resolution that span block boundaries.

In part to address these issues, many texture compression formats have been proposed [10, 20, 36, 43]. Like our representation, these partition the texture up into small blocks and allocate a fixed number of bits of compressed storage per block. Unfortunately, these formats are unsuitable for our purposes, as they exploit the low dynamic range and precision (typically 8-bit) of natural images and the limitations of human vision in order to preserve visual similarity. Moreover, few of them support bit rate selection beyond one or two fixed settings.

Today's GPUs store data other than textures, such as depth and color buffers. These buffers, with few exceptions [38], demand lossless persistent storage [1, 37, 44], but it is often feasible to read from them using a lossy transmission mode or to benefit from lossless transmission of portions that compress well. The goal in buffer compression is to reduce bandwidth rather than storage. In this paper we achieve both via lossy compression.

3 Compression Scheme

Our compression scheme for 3D double-precision data is inspired by ideas developed for texture compression of 2D image data. As in most texture compression formats, we divide the 3D array into small, fixed-size blocks of dimensions $4 \times 4 \times 4$ that are each stored using the same, user-specified number of bits, and which can be accessed entirely independently. At a high level, our method compresses a block by performing the following sequence of steps: (1) align the values in a block to a common exponent; (2) convert the floating-point values to a fixed-point representation; (3) apply an orthogonal block transform to decorrelate the values; (4) order the transform coefficients by expected magnitude; and (5) encode the resulting coefficients one "bit plane" at a time. We will detail each of these steps below.

3.1 Conversion to Fixed-Point

With a hardware implementation in mind, we begin by converting the double-precision values in a $4 \times 4 \times 4$ block to a common fixed-point format, as in [46, 48]. This alignment of values, aka. block-floating-point storage, is done by expressing each value with respect to the largest floating-point exponent in a block, which is stored uncompressed at the head of the block, resulting in normalized values in the range $(-1, +1)$. We use a Q3.60 fixed-point two's complement format that allows numbers in the range $[-8, +8)$ to be represented, i.e. a 64-bit signed integer i represents the value $2^{-60}i$. Although the floating-point values after exponent normalization lie in a smaller range $(-1, +1)$, the subsequent block transform stage requires additional precision to represent intermediate values and final transform coefficients. Note that the implicit leading one bit of non-zero floating-point numbers is represented explicitly in this fixed-point format.

3.2 Block Transform

We transform the fixed-point values to a basis that allows the spatially correlated values to be mostly decorrelated, as this results in many near-zero coefficients that can be compressed efficiently. As is common for regularly gridded data, we employ a separable transform in d dimensions that can be expressed as d 1D transforms along each dimension, resulting in a basis that is the tensor product of 1D basis functions. Our goal is thus to find a suitable 1D basis.

Many discrete orthogonal block transforms have been proposed, each with their pros and cons. Examples include the discrete Haar wavelet transform (HWT), the slant transform (ST), the family of discrete cosine transforms (DCT), among which DCT-II from JPEG is the most common, the high-correlation transform (HCT) used in H.264, the Walsh-Hadamard transform (WHT), and the discrete Hartley transform (DHT); see, e.g., [47]. Another orthogonal basis is the Gram polynomial (aka. discrete Chebyshev polynomial) basis (GP).

We make the observation that in the case of transformations of 4-vectors, all of the above transforms can be expressed as orthogonal matrices of the form

$$A = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ c & s & -s & -c \\ 1 & -1 & -1 & 1 \\ s & -c & c & -s \end{pmatrix} \tag{1}$$

$$s = \sqrt{2}\sin\tfrac{\pi}{2}t \qquad c = \sqrt{2}\cos\tfrac{\pi}{2}t \tag{2}$$

where $t \in [0, 1]$ is a parameter. Based on these definitions, $t = \{0, \frac{2}{\pi}\tan^{-1}\frac{1}{3}, \frac{1}{4}, \frac{2}{\pi}\tan^{-1}\frac{1}{2}, \frac{1}{2}\}$ corresponds to HWT (90 degrees phase shifted), ST, DCT-II, HCT, and WHT, respectively, with DHT coinciding with WHT and GP with ST (modulo sign differences and/or permutation of the basis vectors). To our knowledge, this is the first such parametric description that unifies several well-known orthogonal transforms.

We now form a separable basis for 3D blocks by taking tensor products of the basis vectors (rows) of A:

$$B_{ijk}(x, y, z) = b_i(x) \otimes b_j(y) \otimes b_k(z) \tag{3}$$

with $0 \le i, j, k \le 3$, $||B|| = 1$, and where b_i is the i^{th} basis vector of A (and similarly for b_j and b_k). Transforming a block to this basis is equivalent to performing a sequence of independent 1D transforms along x, y, and z.

We will later order the basis functions by sequency [47], i.e. by $i + j + k$, which can be thought of as a generalization of the zig-zag ordering used in JPEG to 3D arrays. We may think of the indices i, j, k as encoding the polynomial degree of the corresponding 1D basis functions (this is certainly true for the Gram basis), with $0 \le i + j + k \le 9$ representing the total degree. This divides the basis functions into ten equivalence classes.

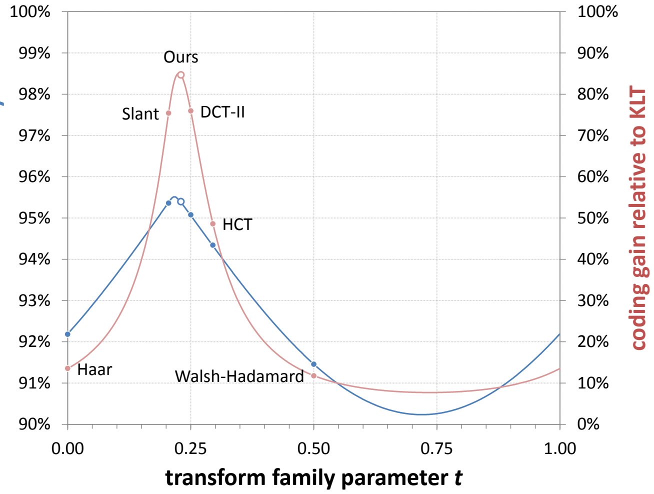


Fig. 2: Decorrelation efficiency (blue; note the vertical scale) and coding gain (red) for 21 scalar fields achieved by several transforms parameterized by $t \in [0, 1]$. Evidently our efficient transform is near optimal.

```
          w -=   x;   w /= 2;   x +=   w;
          y -=   z;   y /= 2;   z +=   y;
          z -=   x;   x *= 2;   x +=   z;
y *= s;   y -= c*w;   w *= 2;   w += c*y;   w /= s;
```

Listing 1: C implementation of the lifted transform $A'\begin{pmatrix} x & y & z & w \end{pmatrix}^T$ for general values of s and c. In our transform $s = \frac{1}{2}$, and therefore `y *= s` and `w /= s` are replaced with `y /= 2` and `w *= 2`.

3.2.1 Lifted Implementation

A naïve implementation of the transform would either unfold the block into a 64-vector and apply multiplication by the 64×64 basis matrix, or perform a sequence of forty-eight 4×4 matrix multiplications by taking advantage of separability. Fortunately any orthogonal matrix can be decomposed into a sequence of plane rotations, each of which can be expressed efficiently using the *lifting scheme* [7] via in-place additions, subtractions, and multiplications. The unique structure of our basis A leads to a very efficient lifted implementation.

In order to eliminate unnecessary sign changes, we slightly rewrite the basis by negating the last two rows of A and let A' denote this modified basis. This change affects only the signs of the transform coefficients and has no impact on orthogonality or error. The resulting forward transform A' can then be implemented as shown in Listing 1. The inverse transform simply reverses the sequence of steps, with addition and multiplication interchanged with subtraction and division.

Although the polyphase decomposition of the transform is not unique [7], we have chosen this particular sequence of lifting steps in order to minimize range expansion of the intermediate quantities. This is an important consideration since the transform is implemented in fixed point and could otherwise overflow. For inputs in $(-1, +1)$, each output (and intermediate) quantity lies in $(-2, +2)$, and after three separable passes in $(-8, +8)$. The tradeoff is some slight loss in precision due to the irreversible divisions by two. Note that multiplications and divisions by two can be implemented as bit shifts and, while not exploited here, the sixteen 4-vectors can be transformed in parallel, with the independent lifting steps enabling additional concurrency.

3.2.2 A New, Computationally Efficient Transform

The above lifted transform, while efficient, requires both multiplication and division by s. Although the division can be implemented as a multiplication by the reciprocal of the constant s, these integer multiplications are nevertheless the most expensive operations involved in the transform. Using the judicious choice $s = \frac{1}{2}$, we turn these multiplications into bit shifts. Remarkably, this choice is not only attractive from a performance standpoint, but is also near optimal in terms of decorrelation efficiency and coding gain, two measures that are commonly used to assess the effectiveness of orthogonal transforms [6].

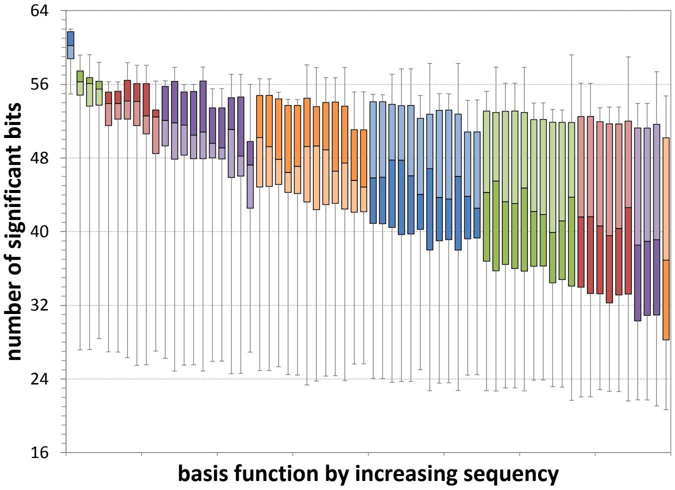


Fig. 3: Box plot showing the distributions of the number of significant bits for the coefficients associated with the 64 basis functions (grouped and color coded by sequency) based on data from 30 different fields. This plot confirms that the energy is concentrated in the low frequencies.

The decorrelation efficiency η and coding gain γ are both computed from the 64×64 covariance matrix $\Sigma = [\sigma_{ij}^2]$ of the transformed signal, where each of the 64 entries in a block is considered to be a random variable that we seek to decorrelate to improve compression. These quantities are given by

$$\eta = \frac{\sum_i \sigma_{ii}^2}{\sum_i \sum_j |\sigma_{ij}^2|} \qquad \gamma = \frac{\sum_i \sigma_{ii}^2}{64\left(\prod_i \sigma_{ii}^2\right)^{1/64}} \tag{4}$$

Using 21 scalar fields from different simulations, comprising nearly 0.8 billion floating-point values in total, we computed $\eta(t)$ and $\gamma(t)$ as functions of the transform parameter t (cf. Eq. (2)). To obtain comparable units for the different quantities represented by the fields, we first normalized the variance of each field to unity and then computed the aggregate covariance matrix across all fields, from which η and γ were obtained. For ease of interpretation, we normalized γ by the maximum coding gain possible, as given by the (data-dependent) Karhunen-Loève Transform (KLT), which produces a diagonal covariance matrix. As Fig. 2 shows, our choice $t = \frac{2}{\pi} \cot^{-1} \sqrt{7} \approx 0.230$ is close to optimal, both with respect to η and γ, for these fields. Furthermore, this choice allows us to replace generic multiplications and divisions with bit shifts (Listing 1).

The resulting transform involves 8 additions, 6 bit shifts (by one), and 2 integer multiplications per transformed 4-vector. Amortized over each numerical value, the complete 3D transform uses 6 additions, 4.5 shifts, and 1.5 multiplications per compressed scalar. This compares quite favorably with the 64 multiplications and 63 additions per scalar required by a naïve matrix-vector product implementation of the transform. It is also reasonably competitive with the Lorenzo predictor employed in FPZIP [30], which requires 7 floating-point additions per compressed value.

The 64-bit fixed-point multiplications could be implemented as four 32-bit multiplications (with 64-bit products) and some bit shifting. We note that in our transform, $c = \frac{\sqrt{7}}{2} \approx 1.323 \approx \frac{10837}{2^{13}}$. This rational approximation is accurate to 7 digits and results in the bottom 32 bits of c being zero, allowing the fixed-point multiplication to be executed as only two instead of four 32-bit integer multiplications. We use this value of c for our results.

A hardware implementation might prefer an even simpler approximation $c \approx \frac{5}{4}$. Multiplication by this constant can be efficiently performed using one addition and a shift by two, allowing the entire transform to be implemented using only integer addition and bit shifting. Note that the orthogonality of the basis and invertibility of the transform are independent of the choice of c, and as long as $c \leq \frac{\sqrt{7}}{2}$ no further range expansion occurs. c does, however, affect the norm of the basis vectors and thus the relative magnitudes of the transform coefficients. We also observed that this crude approximation of c tends to result in some degradation in quality.

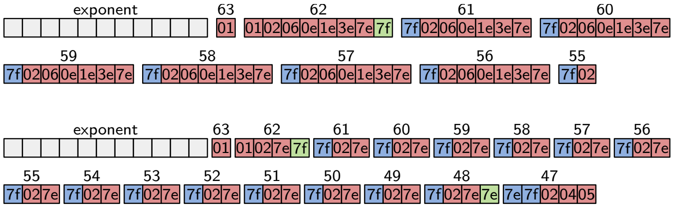


Fig. 4: Single-block bit streams for balanced (top) and unbalanced (bottom) trees partitioned by bit plane number. The boxes show tree node indices in hexadecimal and bit type: group test (red), sign (green), and value (blue). Notice the increase in value and sign bits on the bottom.

3.3 Embedded Coding

Given a collection of transform coefficients, most of which are expected to be small in magnitude, how should they be encoded to yield the best quality for a given bit budget? Ideally, the codec should allow for many different bit rates (as prescribed by the user). Toward this end, we turn to embedded encoding, which produces a stream of bits that are ordered roughly by their impact on error. We note that due to the orthogonality of our transform, the root mean square (RMS) error in the signal domain equals the RMS error in the transform domain, and in this sense all coefficients are equally important. More precisely, all coefficient bits within the same "bit plane" have the same impact on error, and hence our strategy is to encode one bit plane at a time.

In embedded coding, each bit encodes some partial information about the signal, and any prefix of this stream can be decoded independently to yield a valid solution (with any unencoded bit set to zero by the decoder). In this manner, we may use a single codec to produce a full (near) lossless encoding of each block, and this stream can then be truncated to satisfy the user-specified bit rate. Moreover, any already compressed block can be degraded in fidelity by simply further truncating the bit stream, i.e. with no need for recompression.

Several embedded coding strategies exist, among which embedded zerotrees [42] and set partitioning [40] are among the better known, both having been designed for coding wavelet coefficients. Hong and Ladner [17] showed that both of these schemes can be thought of as variants of *group testing*—a procedure originally invented to test consolidated blood samples for infectious disease among larger populations. We employ a custom group testing procedure to encode our transform coefficients.

The main idea is to perform a set of *significance tests*, with each test returning which (signed) coefficients are larger in magnitude than a given threshold. Using progressively smaller thresholds that are powers of two, this amounts to determining which coefficients have a one bit in the current bit plane, not counting those coefficients that were found to be significant in a previous pass. The idea behind group testing is to test not individual coefficients but groups of them as a whole, with the assumption that most of them are insignificant. Thus, a single bit can be used to encode that a whole collection of coefficients are insignificant with respect to the current threshold.

If at least one coefficient in a group is significant, the group is refined by splitting it into two smaller groups, and the procedure is applied recursively until each significant group consists of a single coefficient. For each such significant coefficient, we first encode its sign and then place it onto the list of significant coefficients, whose remaining, less significant bits will be encoded verbatim for subsequent bit planes. The remaining, insignificant groups are then further refined in subsequent passes, until each coefficient is significant or zero, or until we have exhausted the bit budget.

How should we group coefficients for group testing? Unlike in [18], we cannot group corresponding coefficients across blocks, since we must allow each block to be decompressed independently. Instead, we make the observation that the signal energy tends to decrease with frequency, and consequently we expect the magnitudes of coefficients to be ordered by sequency. This is indeed the case, as evidenced by the box plot shown in Fig. 3. Thus, placing the coefficients in sequency order results in a nearly sorted list. For any significant group, we expect the significant coefficients to come from the lower-sequency half, and that in each refined group the coefficients will have roughly the same magnitude. Once a group is found to be significant, we next test the subgroup expected to be *insignificant*, since when this is the case we may infer that the other subgroup must be significant, eliminating the need to test that group and encode the redundant result.

Assuming groups are always split in equal halves, this recursive structure gives rise to a complete binary tree, where each internal node corresponds to a group of leaves (its descendants). The partitioning into groups corresponds to a cut through this tree, and each group test amounts to encoding, using one bit, a node on the cut. If the bit is zero (insignificant), the node is left on the cut until the next bit plane is coded. Otherwise, it is refined by replacing it with its children. Thus, for each bit plane we make a breadth-first traversal of the tree and refine the cut as needed. Once a leaf node is found to be significant, we mark it as such and remove it from the cut. Each bit plane pass then sends one value bit for each significant node followed by group test bits that refine the cut.

We note that a balanced tree structure is not necessarily optimal. For instance, as seen in Fig. 3, the DC component (far left in the plot) usually has a greater magnitude than the other coefficients. In a balanced tree, the group refinement that first exposes the DC component as the only significant coefficient splits the coefficients into $\log_2 N + 1$ progressively smaller groups, (where $N = 64$ is the block size), and each of the $\log_2 N = 6$ insignificant groups are then tested in subsequent passes, even though they are often all insignificant (see Fig. 4).

To address this issue, we use an unbalanced tree that locates the low-sequency components near the root, and where groups of higher-sequency coefficients are stored progressively deeper in the tree. This results in fewer groups initially, allowing more value bits to be coded (Fig. 4, bottom). We found that this unbalancing of the tree gave a 2–6 dB increase in PSNR at low bit rates. For efficiency, we represent the cut in the tree as a single 127-bit mask, the set of significant coefficients as a 64-bit mask, and the (fixed) tree structure as an array of per-node “pointers” to the left child (siblings have consecutive coefficient indices).

4 Caching

As presented, the proposed scheme would for each floating-point value accessed require decompressing its corresponding block and then compressing and storing the block upon each write (or update) access. This would be prohibitively expensive if implemented in software, but also if compression were done in hardware. Not only would this approach incur frequent computation associated with (de)compression, but precious memory bandwidth would also be spent on transferring compressed data over the memory bus.

To alleviate this bandwidth pressure, we use a small, direct-mapped software cache of decompressed blocks. We use a write-back policy with a “dirty bit” stored with each cached block, such that compression is only invoked for blocks evicted from the cache that have been modified. That is, for tasks that do not modify the floating-point array (e.g. many analysis tasks), only decompression is needed, ensuring that the fidelity of the data is not impacted after the initial one-time compression stage. This user-configurable cache represents the only memory overhead in our scheme beyond the compressed blocks.

Many analysis tasks and simulation kernels “stream through” the 3D array sequentially in an outer loop and access only immediate neighbors in an inner loop. A similar access pattern is used when gathering or scattering values between data with different *centerings*, such as between nodes (vertices) and cells (elements). To best support such access patterns, our default cache size accommodates two layers of blocks. Assuming no cache conflicts, this ensures that only compulsory cache misses are incurred when making a sequential pass over the array elements and accessing only immediate neighbors.

field	DR	lossless				FPZIP 32-bit precision					FPZIP 16-bit precision				
		ratio				ratio	PSNR		accuracy		ratio	PSNR		accuracy	
		GZIP	FPC	FPZIP	ZFP		FPZIP	ZFP	FPZIP	ZFP		FPZIP	ZFP	FPZIP	ZFP
ρ	0.00	2.11	1.92	**2.74**	2.61	14.67	**120**	77	31.2	**53.0**	37.3	22.7	**53.2**	15.0	**42.0**
p	1.62	1.06	1.13	**1.38**	1.37	4.47	**156**	141	32.0	**32.5**	27.2	59.5	**67.9**	16.0	**19.6**
u	0.77	1.05	1.16	**1.46**	1.43	5.36	**148**	130	32.0	**32.8**	30.6	52.2	**66.0**	16.0	**21.0**
v	0.76	1.05	1.17	**1.46**	1.43	5.36	**148**	130	32.0	**32.8**	31.0	51.5	**65.8**	16.0	**21.0**
w	0.65	1.05	1.17	**1.46**	1.44	5.45	**147**	131	32.0	**32.8**	31.3	51.0	**67.7**	16.0	**21.1**
μ	7.30	1.04	1.05	1.11	**1.21**	2.52	156	**205**	32.0	**35.2**	6.7	60.0	**110.4**	16.0	**19.4**
D	7.32	1.04	1.05	1.12	**1.21**	2.53	156	**204**	32.0	**35.1**	6.7	60.0	**109.9**	16.0	**19.4**
T	0.00	1.15	1.34	**1.65**	1.63	8.72	**126**	95	32.0	**34.6**	101.5	30.4	**36.7**	15.8	**19.8**
u	0.25	1.05	1.19	**1.45**	1.44	5.25	**135**	129	32.0	**32.7**	36.9	39.0	**62.1**	16.0	**21.0**
v	0.27	1.06	1.17	**1.45**	1.43	5.28	**137**	131	32.0	**32.6**	37.3	40.8	**64.9**	16.0	**20.8**
w	0.32	1.06	1.18	**1.45**	1.43	5.21	**135**	132	32.0	**32.5**	36.9	39.2	**65.3**	16.0	**20.7**
H_2	0.18	1.12	1.27	**1.59**	1.58	6.84	**126**	111	**32.0**	31.7	50.8	28.0	**51.1**	15.5	**20.8**
O_2	0.06	1.16	1.29	1.60	**1.61**	6.53	**127**	105	32.1	**32.2**	52.2	31.8	**45.9**	16.1	**21.5**
O	6.08	1.03	1.07	1.25	**1.31**	3.27	141	**174**	32.0	**33.0**	12.0	44.7	**90.5**	15.9	**19.2**
OH	6.17	1.03	1.08	1.26	**1.33**	3.31	141	**170**	32.0	**32.6**	11.7	44.7	**89.3**	16.2	**19.2**
H_2O	6.02	1.06	1.12	1.30	**1.38**	3.51	124	**161**	31.9	31.9	11.9	34.1	**86.6**	16.4	**19.5**
H	5.89	1.03	1.08	1.24	**1.31**	3.28	140	**171**	32.0	**33.1**	11.7	44.1	**88.8**	16.1	**19.4**
HO_2	5.59	1.03	1.08	1.24	**1.31**	3.25	138	**151**	32.0	**33.3**	11.7	42.1	**67.5**	16.1	**19.5**
H_2O_2	5.91	1.03	1.08	1.24	**1.31**	3.24	140	**156**	32.0	**33.3**	11.5	44.2	**72.6**	16.1	**19.5**
N_2	0.00	1.24	1.43	**1.82**	1.81	10.98	**113**	83	32.0	**34.2**	226.3	**16.8**	-1.3	**16.6**	15.8
aggregate		1.09	1.18	1.41	**1.44**	4.41	138	**139**	32.0	**34.1**	18.1	41.8	**68.1**	16.0	**21.0**

Table 1: Compression ratios, peak signal to noise ratio, and accurary in bits for ZFP and other compressors (the best results appear in bold). DR denotes the median base-2 dynamic range across 4 × 4 × 4 blocks.

5 Results

We evaluated the quality and speed of our compressor, code named ZFP, on various double-precision fields obtained from four physics simulations: Miranda (an LLNL hydrodynamics code), S3D (a Sandia combustion code), pF3D (an LLNL laser-plasma interaction code), and LULESH (an LLNL shock hydrodynamics proxy application). We ran our experiments on a single core of an iMac with 3.4 GHz Intel Core i7 processors and 32 GB of 1600 MHz DDR3 RAM. We use *bpd* (bits per double) to refer to the amortized storage cost of each value.

5.1 Quality

In order to assess the quality provided by our lossy compression scheme, we report on two quality measures: the peak signal to noise ratio (PSNR) and accuracy. We define PSNR Q for a discrete signal x of length N and approximate signal $\tilde{x}$ as

$$Q(x, \tilde{x}) = 10 \log_{10} \frac{\left[\frac{1}{2}(\max_i x_i - \min_i x_i)\right]^2}{\frac{1}{N}\sum_i (x_i - \tilde{x}_i)^2} \tag{5}$$

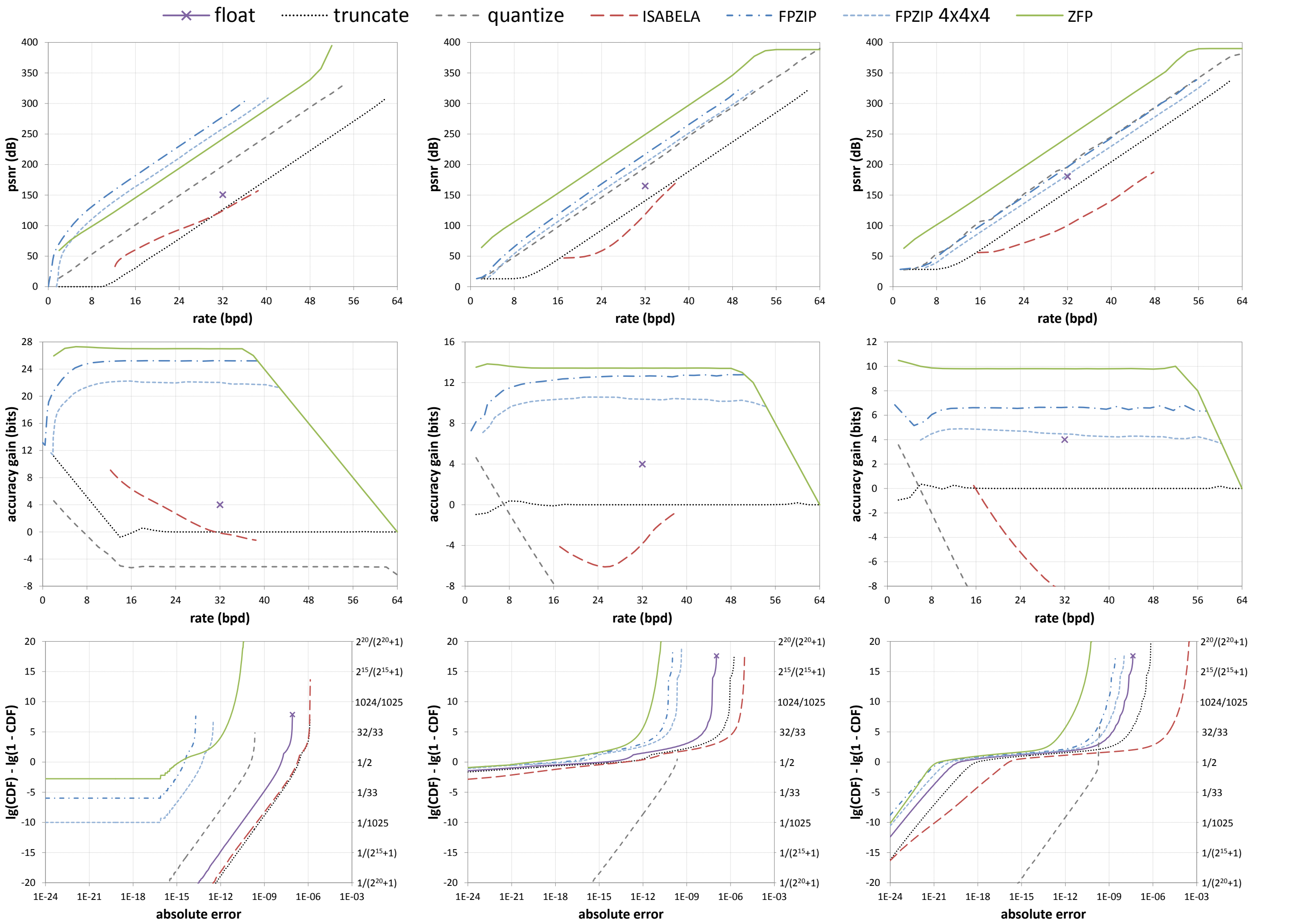
Note that Q accounts for the absolute error. For applications that are more concerned with element-wise relative error, we measure the *accuracy* as the number of bits of agreement between two floating-point numbers. Because this measure is not straightforward to define when one number is zero, when the two numbers are of opposite sign, or when they narrowly straddle a power of two, we adopt the following definition. Let $I(x)$ denote the binary integer representation of x obtained by converting the sign-magnitude floating-point representation to a two's complement integer, such that $I(x) < I(y) \iff x < y$. The accuracy α of $\tilde{x}$ with respect to x is then given by

$$\alpha(x, \tilde{x}) = 64 - \log_2(|I(x) - I(\tilde{x})| + 1) \tag{6}$$

where $|I(x) - I(\tilde{x})|$ measures the number of floating-point values between x and $\tilde{x}$. α does indeed relate to the number of bits of agreement when the signs and exponents of x and $\tilde{x}$ agree, but also handles numbers with different exponents or signs in an intuitive manner.

Rather than reporting the accuracy directly, which tends to vary linearly with the rate, we plot its difference with the rate, which we call the *accuracy gain*. This gain is the additional number of bits inferred, e.g. via prediction or orthogonal transform, and in a sense is the amount of redundant information exposed and discarded by the compressor. Note that it is possible for the accuracy gain to be negative.

We compare ZFP with several alternatives. One straightforward way to reduce precision is to store values in single (float) precision. This is common in visualization and analysis, which usually demand less



(a) S3D temperature (T) (b) S3D oxygen mass fraction (O) (c) Miranda viscocity (μ)

Fig. 5: Peak signal to noise ratio (top) and median accuracy gain (middle) vs. rate (bpd = bits per double) and cumulative normalized-error distribution at 32 bpd (bottom) for our ZFP method, ISABELA, original FPZIP, FPZIP applied to $4 \times 4 \times 4$ blocks, and naïve approaches like mantissa truncation, uniform quantization, and conversion to single-precision floats. The CDFs have been transformed nonlinearly to emphasize the tails.

precision. Such conversion usually results in a positive accuracy gain of four bits; three of these are due to the difference in number of exponent bits between single and double precision, and the fourth results from rounding rather than truncating the mantissa when converting to float. Two other common approaches are to truncate the mantissa (and even exponent) by discarding (zeroing) least significant bits (with bounded relative error), and to uniformly quantize numbers between the extremal values (with bounded absolute error).

We also compare with two other lossy floating-point compressors: FPZIP [29, 30] and ISABELA [24, 25] (using B-splines with $W_0 = 1024, C = 30$, and varying error tolerance ϵ), as well as with lossless compressors GZIP and FPC [4, 5] (with a 16 MB hash). FPZIP is primarily a lossless predictive coder, but supports a lossy mode by losslessly compressing truncated mantissas. Hence FPZIP bounds the relative error, as does ISABELA via a tolerance parameter. We found FPZIP to always outperform ISABELA, and we will focus our comparison primarily on FPZIP. Because FPZIP is a streaming variable-rate coder, it does not support localized random access. We attempted a more apples-to-apples comparison by applying FPZIP to the same 4^3-sized blocks used by ZFP (while excluding header information). We will refer to this method as FPZIP $4 \times 4 \times 4$.

Table 1 lists compression ratios (uncompressed size divided by compressed size), PSNR, and median accuracy for several fields for ZFP and the other compressors. This table also lists for each field its average local dynamic range represented as the median of values $\log_2(\max |x_i| / \min |x_i|)$, where the local extrema are computed over $4 \times 4 \times 4$ blocks. In "lossless" mode we encoded 58 bit planes for ZFP, which always provided a median accuracy of 64 bits and PSNR of at least 313 dB (i.e. better than floating-point epsilon), and we relaxed the fixed-rate requirement. Even though ZFP was not designed for lossless compression, it performed slightly better than FPZIP and significantly better than FPC in aggregate (harmonic mean compression ratio). We then ran FPZIP in 16- and 32-bit precision mode (truncating mantissas), and assigned a fixed rate to ZFP so that it compressed to the same size as FPZIP. In 32-bit mode, we found FPZIP to give higher PSNR on the easier to compress fields (as hinted by the lossless compression ratios and low dynamic range), while ZFP did better on less compressible data. In virtually all cases, ZFP gave a higher accuracy, however. At higher compression ratios, ZFP also gave significantly higher PSNR. The one anomaly is the single-exponent N_2 field, for which ZFP was only allowed 7 bits per block in addition to the common exponent.

Fig. 5 plots the PSNR and median accuracy gain for three of these scalar fields: the S3D temperature and oxygen mass fraction fields, and the Miranda viscocity field. These three data sets represent different exponent distributions in the double-precision numbers, from the very low entropy temperature field (only three exponents), to a data set with hundreds of exponents (for predominantly positive numbers), to one with a roughly equal number of negative and positive values. Although the mass fractions should all be non-negative, numerical error caused some of them to be slightly negative.

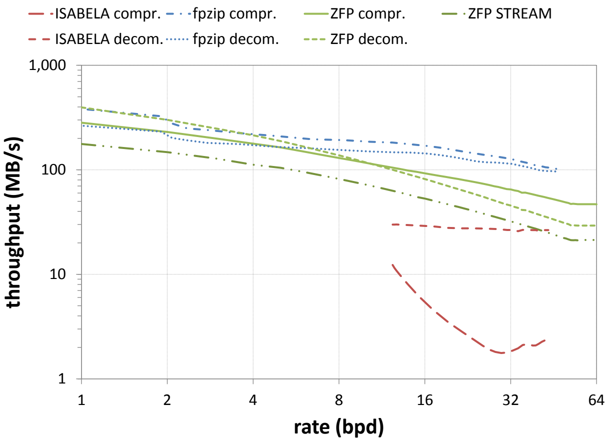


Fig. 6: Raw compression and decompression throughput in number of uncompressed bytes input or output per second. Higher rates require more bit plane passes in ZFP and thus more time.

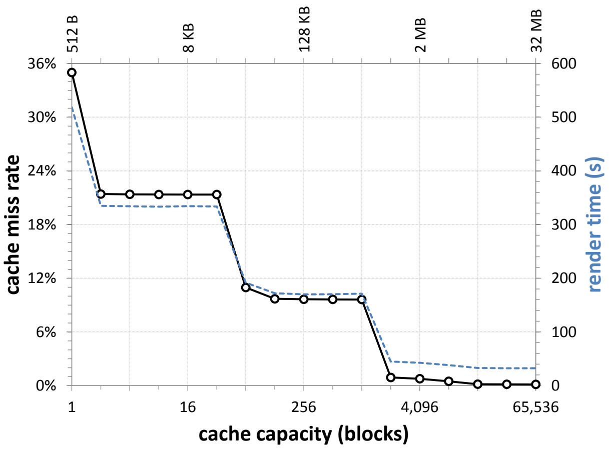


Fig. 7: Cache utilization and total rendering time (dashed) as a function of cache size for 4 bpd ray tracing. The full, uncompressed scalar field is 288 MB, and took 28 seconds to render without compression.

This figure shows that PSNR increases linearly with rate (as expected) for most methods, until reaching infinity when there is no loss. We observe a plateau in PSNR for ZFP reached around 56 bpd. This occurs because the uncompressed data has only 52 bits of mantissa, while we reserve four non-mantissa bits in our fixed-point representation for the sign and extra precision needed by the block transform. Using more than 56 bpd is beneficial only for blocks with non-uniform exponents, where the mantissa is shifted into the least 8 significant bits. For values closer to zero, exponent differences within a block may exceed eight, in which case ZFP loses some low-order mantissa bits, preventing fully lossless compression. However, such low-order bits would also be lost in arithmetic operations like addition whenever the exponents of two operands differ; a loss that is generally accepted.

We note that other than for the easily compressible temperature field, ZFP outperforms FPZIP by 30–50 dB (i.e. 1.5–2.5 decimal digits), and the blocked version of FPZIP and ISABELA by even more—in spite of ZFP being a fixed-rate compressor. ZFP is disadvantaged by easy-to-compress data, in that blocks that compress very well must still be padded to the fixed bit budget. We notice that the PSNR for ZFP exhibits a surprising consistency (e.g. $Q \approx 240$ dB at 32 bpd) across the data sets, and is largely unaffected by the compressibility of the data. In contrast, PSNR for FPZIP varies by as much as 80 dB across these three data sets for a fixed bit rate. We see this predictable behavior as a strength of ZFP that is likely to facilitate rate selection.

Because FPZIP bounds the relative error, it is perhaps not surprising that it does not always perform well in terms of absolute error. The middle row of Fig. 5, which plots the relative error in terms of accuracy gain, reveals a surprise, however. Here FPZIP is again outperformed by ZFP. Moreover, we see that ZFP excels at very low bit rates. This ability to keep both absolute and relative errors low is another strength of ZFP. Because the accuracy gain plus the bit rate equal the accuracy $\alpha \leq 64$, the plot is bounded by the diagonal line $y = 64 - x$, which explains the convergence of the curves to this line.

The bottom row of Fig. 5 shows at 32 bpd the cumulative distribution of absolute errors normalized by field range (CDF values are shown on the right vertical axis). We see that the maximum error (the rightmost point on each curve) correlates quite well with PSNR. Evidently ZFP achieves low maximum errors even for difficult-to-compress blocks, in spite of the fixed-rate constraint.

We also evaluated our efficient transform with respect to the discrete cosine transform within the framework of our compressor. On average, we found our transform to give a 1.5–4.0 dB improvement in PSNR at low to mid bit rates, and even more in the near-lossless regime.

5.2 Speed and Cache Utilization

We here evaluate the raw speed of our compressor when applied to the $384 \times 384 \times 256$ Miranda pressure field (other fields gave similar results) while performing strided data accesses to the uncompressed array. Fig. 6 shows the throughput in number of uncompressed bytes input from or output to main memory per second, which at 1 bpd tops out around 400 MB/s for our decompressor and 280 MB/s for the compressor. This is roughly comparable to the speed of FPZIP. As the rate increases, progressively more bit planes are processed and output, resulting in a gradual reduction in throughput.

We also evaluated the speed of ZFP array accesses using the well-known STREAM benchmark [33] on arrays initialized with the pressure field to avoid trivial compression of constant values. The overhead of translating linear array indices and caching data reduces throughput by a nearly constant 1.7x over raw (de)compression.

To test the effectiveness of caching decompressed blocks, we used our compressed array in a ray tracer. We measured both cache misses and total rendering time vs. cache size (Fig. 7) while rendering the data set in Fig. 1(b). Other than good correlation between cache misses and rendering time, this figure shows three distinct plateaus. For small caches, many misses occur during initialization when the array is traversed in raster order to precompute which cells intersect the interval volume, as determined by 8 adjacent cell corner values. When smaller than the 96-block wide domain, the cache must be reloaded each time the innermost loop restarts. A similar issue occurs when a whole slice does not fit in cache. We find that in spite of 18 million rays cast, more than half of the cache misses occur during initialization, as consecutive primary rays tend to traverse similar regions of the domain. Once the third, lowest plateau is reached, very few cache misses occur. Using a cache of 16 K blocks and 4 bpd compression, the rendering time of 33 seconds is only slightly longer than the 28 seconds taken without compression, though using 11 times less memory: 18 MB of compressed data and 8 MB of cache, vs. 288 MB uncompressed. Remarkably only 24 MB of compressed data had to be decompressed to satisfy the 3.6 GB worth of floating-point read accesses made.

6 Applications

We now evaluate our compressor in a number of applications. We note that the total programming effort to modify all of these applications to use our compressor was less than 90 minutes.

6.1 Quantitative and Visual Analysis

We begin by assessing the viability of lossy compression for quantitative analysis, which normally is done at reduced (single) precision, and hence we expect some loss in precision to be acceptable. We first consider computing the Fourier spectrum of the density field at the mid plane of a Rayleigh-Taylor instability simulation [26]. This is a common analysis method used in hydrodynamics for detecting turbulence, which should manifest itself as an exponentially decaying spectrum with a slope of $-5/3$ at middle frequencies.

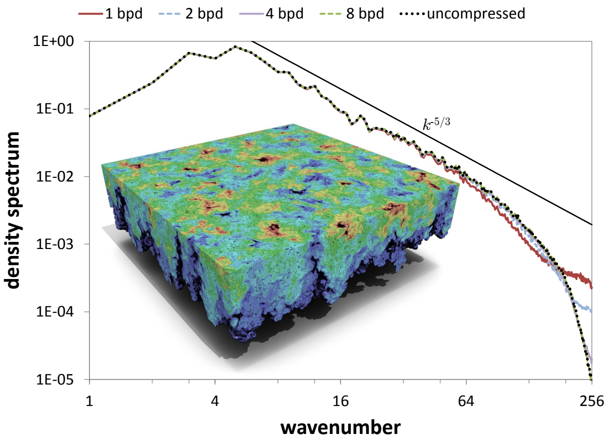


Fig. 8: Density spectrum computed over the 2D mid plane. The spectrum follows a $k^{-5/3}$ power law in the middle region that typifies turbulent behavior. The spectrum is well represented at 4 bpd and above.

Fig. 8 shows the spectrum for a late time step in the simulation, where turbulence has set in, and confirms the expected power law. We applied our lossy compressor to this 512^3 double-precision field and reran the spectral analysis The figure shows that low-frequency modes are well represented, but the loss in precision and blocking introduce high-frequency noise, where the spectra disagree. At 4 bpd (16x compression), however, the spectrum is well represented at all but the very highest frequencies, and at 8 bpd the difference is virtually undetectable. Using 4-bit uniform quantization, we obtained comparable results to using 1 bpd compression. We note that the sharp drop in power at the highest frequencies is common in simulation codes, as numerical accuracy demands that the field is well resolved and varies reasonably smoothly at the finest grid scale.

A common objection to block transform coding is the potential for artifacts, which usually manifest themselves as visible discontinuities between blocks (cf. JPEG artifacts). Such discontinuities are also quite evident in the 1 bpd field shown in Fig. 1(a), though perhaps not surprisingly so given the high compression rate.

We note that even minor discontinuities in a scalar field are usually exposed when computing its derivatives. To test the presence for such artifacts, we computed unstable manifolds of scalar fields using a variation of the technique presented in [16]. Each such manifold consists of the set of points in the domain whose downward gradient integral lines converge on the same minimum, and collectively these manifolds segment the domain into spatially coherent pieces.

Fig. 10 shows these unstable manifolds for the 2D mid plane of the vertical velocity field (w) from the Miranda run. Similar segmentations were done by Laney et al. [26] to extract "bubble and spike" features that characterize the turbulent behavior of the Rayleigh-Taylor instability. Each segment corresponds to a region of downward motion due to gravity associated with a spike during fluid mixing. We notice the spurious extrema and segments in Fig. 10(a), where 64x compression was used. These could possibly be removed using persistence-based simplification. (Differences in segment color are primarily due to minor shifts in the locations of minima.) Nevertheless, most segments are still recognizable, whereas results based on 1-bit truncation or quantization would clearly be nonsensical. At 4 bpd, the segmentation is essentially indistinguishable from ground truth.

Fig. 10(d) quantifies the error in segmentation using an information theoretic measure. Here the error is the *normalized variation of information* $0 \leq \frac{V(X,Y)}{H(X,Y)} \leq 1$, where $V(X,Y) = H(X,Y) - I(X;Y)$ is the variation of information [34] (a metric) between segmentations X and Y, $H(X,Y)$ is the joint entropy of corresponding segmentation labels, and $I(X;Y)$ is their mutual information. This error converges quickly to zero as the bit rate is increased. We conclude that our block transform does not appear to introduce any discontinuities in the field or its first derivatives at 4 bpd and above.

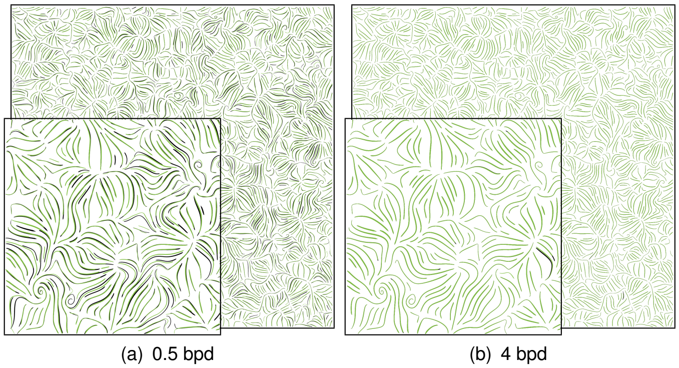


Fig. 9: Streamlines extracted from compressed Rayleigh-Taylor velocity fields. The green lines from each compressed field overlay the black lines from the uncompressed field.

6.2 Visualization

We take the discrete approximation to gradients performed above for Morse segmentations one step further and compute streamlines in bilinearly interpolated vector fields using 4th order Runge-Kutta integration. Any errors in the vector field should be exposed by diverging streamlines with respect to the uncompressed field, as small errors have the potential to grow over the course of integration.

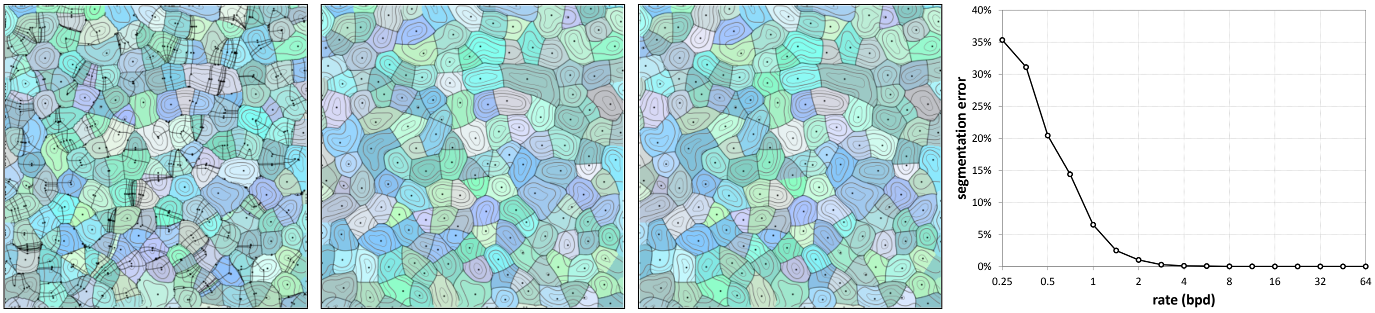
Fig. 9 shows streamlines for the uncompressed field in black, overlaid by green streamlines computed for the compressed vector fields. Again we use the mid plane of the RTI simulation, where the vector field is given by the horizontal velocity (u, v), although we extracted this 2D field from the compressed 3D field. At half a bit per double, many per-streamline errors are evident, yet the field is qualitatively a fair approximation of the full 64-bit field. Very few errors (black lines) are evident in the 4 bpd vector field plot. Again, block artifacts do not seem to be significant enough to cause concern.

As discussed above, we also integrated our compressor with a volumetric ray tracer, which computes interval volumes (the region between two isosurfaces) and spawns secondary rays for shadow casting and ambient occlusion to simulate global illumination (see Fig. 1, for instance). Although the artifacts in Fig. 1(a) are readily visible, the 4 bpd rendering, which also uses gradients for lighting, is virtually indistinguishable from the 64 bpd uncompressed field.

6.3 Blast Wave Simulation

We conclude with a more challenging problem that involves both frequent reads from and writes to the compressed array—a fluid dynamics simulation. We used LULESH [23], a shock hydrodynamics code that simulates a point explosion known as a Sedov blast. LULESH is a Lagrangian (moving mesh) C++ code that uses a logically regular 3D grid. The initial point explosion generates a shock wave that propagates radially and distorts the mesh as it travels through the domain. The Sedov problem can be solved in closed form, giving $r(t) \propto t^{2/5}$, where r is the radial position of the shock wave and t is time.

Using our compressor within LULESH presents a new challenge—the fields are not only read at reduced accuracy, but are also updated several times each time step. This periodic decompression and lossy compression introduces errors that may propagate and grow over time, potentially causing the simulation to diverge. Our goal was, thus, to assess whether such divergence was observed in practice, and what levels of compression (if any) were deemed acceptable. We note that this experiment is similar to the one carried out by Laney et al. [27], but differs in that in our case compression and decompression occur over the course of each time step, as dictated by our caching scheme, whereas Laney et al. applied full decompression and compression of the entire state at the beginning and end of each time step. Thus, our experiment is a more challenging stress case, as compression may occur more than once per time step. Furthermore, this example involves fields with a sharp discontinuity near the shock wave that could be difficult to preserve.



(a) 1 bpd: 6.5% error (b) 4 bpd: 0.09% error (c) no compression (d) error vs. bit rate

Fig. 10: (a–c) Morse segmentations of the vertical velocity field from the Rayleigh-Taylor simulation. (d) Segmentation error of unstable manifolds in terms of normalized variation of information as a function of compressed bit budget.

One important consideration is the cache size to use. Since kernels in LULESH perform gathers and scatters between nodes and cells in raster order, we conjectured that a cache holding two layers of blocks would be sufficient. Using such a cache, LULESH required write-back (lossy compression) of each block 1.18 times per time step on average. Halving the cache size, we observed a dramatic increase in this number to 20.9. Consequently, we used two block layers in our evaluation, representing $\frac{1}{8}$ of the 64^3 domain.

We note that although per mesh element errors may be observed relative to a run with no compression, the important quantity of interest is the actual shock position $r(t)$, which we measured as the radial position (relative to the point source) of the mesh element of maximal density. For the 1,000 time steps executed, we computed the Pearson correlation coefficient between t and $r(t)^{5/2}$, which theory predicts to be linearly related. For a run using 16 bpd compression, we found the correlation with theory to be 99.994%, with a relative error in final shock position of 0.06% compared to the uncompressed run. Reducing the precision to 12 bpd and 8 bpd, this error increased to 0.80% and 7.38%, respectively. We conclude that in spite of over a thousand applications of lossy compression to each mesh element, the outcome of this simulation did not change appreciably at 4–5x compression.

7 Discussion

Although some of the ideas proposed here partially overlap with prior work on compressed representations, we would like to discuss some of the unique aspects and strengths of our approach, potential use cases not covered already, as well as limitations.

7.1 Limitations

Our representation is inherently limited to regularly gridded data, and it is unclear how it could be adapted to unstructured grids. However, we believe that it could find utility in adaptive mesh refinement codes that use nested regular grids. For effective compression, the data should exhibit some smoothness at fine scales, which is common in simulation data, though observational data may be more noisy. The fact that we handle shocks adequately is a promising sign, however.

As designed, our scheme cannot easily bound the maximum error incurred. Our approach has been to minimize the RMS error, which we believe is a good compromise between mean and maximum error. Moreover, by relaxing the fixed-rate constraint, we can easily support a fixed-quality mode, where either an absolute or relative error tolerance is met. Doing so requires essentially no changes to the compressor.

7.2 Benefits

Perhaps one of the main strengths of our approach is its ease of integration. Our compressed array primitive supports a simple C++ interface, including (i, j, k) indexing and flat 1D array view, which allows it to be dropped into existing codes with minimal coding effort. The applications considered here collectively exceed well over 10,000 lines of code, yet our integration of compression required only just over an hour of total programming work to swap out array declarations and add controls for rate selection and cache size.

We have so far discussed using our compressor for storing static fields in visualization and analysis, and evolving fields in simulation. There is a third important use case: representing large, constant tables of numerical data, such as (multidimensional) equation of state and opacity tables queried by the simulation. These can occupy gigabytes to terabytes and require distributed storage. Using our compressed representation, memory is freed up and communication is reduced while sharing these tables across nodes.

Although not stressed here, our compressor allows for a smooth tradeoff between quality and memory usage. Thus, for very large data sets that normally do not fit in main memory, the analysis could nearly always proceed compressed, albeit at reduced accuracy.

Because the bit stream is embedded, it can be truncated at any point (or lengthened via zero padding). Thus no decompression followed by re-compression is needed to change the precision of a coded block, which could introduce further loss. Rather, a single unified format can be used for simulation, visualization, analysis, off-line storage, etc., with each user choosing only how many bits of precision to retain.

Finally, we are not aware of any other fixed-rate compression scheme for floating-point data, let alone a compressed array primitive. In addition to facilitating random access, the fixed storage size also allows the ordering of compressed blocks to be optimized to improve locality of reference, e.g. for out-of-core computations.

8 Conclusion

We presented a compressed representation of 3D floating-point arrays that supports random-access reads and writes. To achieve high compression ratios, our scheme is lossy, but by allowing the user to specify the exact amount of compression our method can approach lossless mode. In spite of being constrained to meet a fixed rate, our method compares favorably to state-of-the-art variable-length compressors—especially at low bit rates. Our compressor achieves high quality through a new and efficient orthogonal block transform, and can be implemented using only integer addition and shifting. It is hence very fast, and can through caching nearly entirely hide the overhead of decompression. Although we focused on compression of 3D arrays of double-precision numbers, our approach is straightforward to generalize to single-precision values and arrays of other dimensions.

Using several examples we demonstrated that 16x compression or more can often be tolerated in visualization and analysis applications, while 4x compression is possible in simulations that demand repeated state updates and, thus, frequent compression and decompression.

Future work is needed to make our approach more widely applicable, including resolving issues like thread safety, investigating more effective caching, finding good layouts of the compressed blocks, extending the compressor to time-varying data, and further tuning the method for hardware implementation. Some applications may want to adapt the rate spatially. While supported by our compressor, variable-length blocks would require more complicated memory management. We are particularly intrigued by the idea of applying our approach to texture compression, noting that further simplifications can be made to the compressor when working with two-dimensional integer arrays.

References

[1] T. Akenine-Möller and J. Ström. Graphics processing units for handhelds. *Proceedings of the IEEE*, 96(5):779–789, 2008.

[2] A. H. Baker, H. Xu, J. M. Dennis, M. N. Levy, D. Nychka, S. A. Mickelson, J. Edwards, M. Vertenstein, and A. Wegener. A methodology for evaluating the impact of data compression on climate simulation data. In *ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 203–214, 2014.

[3] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt. Integrating online compression to accelerate large-scale data analytics applications. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 20–24, 2013.

[4] M. Burtscher. FPC version 1.1, 2006. http://www.csl.cornell.edu/~burtscher/research/FPC/.

[5] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *Data Compression Conference*, pages 293–302, 2007.

[6] R. J. Clarke. *Transform coding of images*. Academic Press, Inc., 1985.

[7] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998.

[8] K. Engel, M. Hadwiger, J. Kniss, A. Lefohn, C. Salama, and D. Weiskopf. Real-time volume graphics, 2004. ACM SIGGRAPH Course #28.

[9] V. Engelson, D. Fritzson, and P. Fritzson. Lossless compression of high-volume numerical data from simulations. In *Data Compression Conference*, pages 574–586, 2000.

[10] S. Fenney. Texture compression using low-frequency signal modulation. In *Graphics Hardware*, 2003.

[11] N. Fout, H. Akiba, K.-L. Ma, A. E. Lefohn, and J. Kniss. High-quality rendering of compressed volume data formats. In *Eurovis*, pages 77–84, 2005.

[12] N. Fout and K.-L. Ma. Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1600–1607, 2007.

[13] N. Fout and K.-L. Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2295–2304, 2012.

[14] J. Fowler and R. Yagel. Lossless compression of volume data. In *IEEE Symposium on Volume Visualization*, pages 43–50, 1994.

[15] S. Guthe and W. Strasser. Real-time decompression and visualization of animated volume data. In *IEEE Visualization*, pages 349–356, 2001.

[16] A. Gyulassy, P.-T. Bremer, and V. Pascucci. Computing Morse-Smale complexes with accurate geometry. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2014–2022, 2012.

[17] E. S. Hong and R. E. Ladner. Group testing for image compression. *IEEE Transactions on Image Processing*, 11(8):901–911, 2002.

[18] E. S. Hong, R. E. Ladner, and E. A. Riskin. Group testing for block transform image compression. In *Asilomar Conference on Signals, Systems and Computers*, pages 769–772, 2001.

[19] N. Huebbe, A. Wegener, J. Kunkel, Y. Ling, and T. Ludwig. Evaluating lossy compression on climate data. In *International Supercomputing Conference*, pages 343–356, 2013.

[20] K. Iourcha, K. Nayak, and Z. Hong. System and method for fixed-rate block-based image compression with inferred pixel values, 1999. http://www.google.com/patents/US5956431.

[21] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless compression of predicted floating-point geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[22] J. Iverson, C. Kamath, and G. Karypis. Fast and effective lossy compression algorithms for scientific datasets. In *Euro-Par Parallel Processing*, pages 843–856, 2012.

[23] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *IEEE International Parallel & Distributed Processing Symposium*, pages 919–932, 2013.

[24] S. Lakshminarasimhan. ISABELA version 0.2, June 2014. http://freescience.org/cs/ISABELA/ISABELA.html.

[25] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *Euro-Par Parallel Processing*, pages 366–379, 2011.

[26] D. Laney, P.-T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1053–1060, 2006.

[27] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener. Assessing the effects of data compression in simulations using physically motivated metrics. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 76:1–12, 2013.

[28] N. K. Laurance and D. M. Monro. Embedded DCT coding with significance masking. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 2717–2720, 1997.

[29] P. Lindstrom. FPZIP version 1.1.0, June 2014. https://computation.llnl.gov/casc/fpzip/.

[30] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[31] E. B. Lum, K.-L. Ma, and J. Clyne. Texture hardware rendering of time-varying volume data. In *IEEE Visualization*, pages 263–270, 2001.

[32] H. S. Malvar. Extended lapped transforms: Properties, applications, and fast algorithms. *IEEE Transactions on Signal Processing*, 40(11):2703–2714, 1992.

[33] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995. http://www.cs.virginia.edu/stream/ref.html.

[34] M. Meila. Comparing clusterings by the variation of information. In *Learning Theory and Kernel Machines*, pages 173–187, 2003.

[35] P. Ning and L. Hesselink. Vector quantization for volume rendering. In *ACM Workshop on Volume Visualization*, pages 69–74, 1992.

[36] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. Adaptive scalable texture compression. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics*, pages 105–114, 2012.

[37] J. Pool, A. Lastra, and M. Singh. Lossless compression of variable-precision floating-point buffers on GPUs. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 47–54, 2012.

[38] J. Rasmusson, J. Ström, and T. Akenine-Möller. Error-bounded lossy compression of floating-point color buffers using quadtree decomposition. *The Visual Computer*, 26(1):17–30, 2010.

[39] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference*, pages 133–142, 2006.

[40] A. Said and W. A. Pearlman. A new, fast, and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):243–250, 1996.

[41] J. Schneider and R. Westermann. Compression domain volume rendering. In *IEEE Visualization*, pages 293–300, 2003.

[42] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, 1993.

[43] J. Ström and T. Akeninie-Möller. iPACKMAN: High-quality, low-complexity texture compression for mobile phones. In *Graphics Hardware*, pages 63–70, 2005.

[44] J. Ström, P. Wennersten, J. Rasmusson, J. Hasselgren, J. Munkberg, P. Clarberg, and T. Akenine-Möller. Floating-point buffer compression in a unified codec architecture. In *Graphics Hardware*, pages 75–84, 2008.

[45] A. Trott, R. Moorhead, and J. McGinley. Wavelets applied to lossless compression and progressive transmission of floating point data in 3-D curvilinear grids. In *IEEE Visualization*, pages 385–388, 1996.

[46] B. E. Usevitch. JPEG2000 extensions for bit plane coding of floating point data. In *Data Compression Conference*, page 451, 2003.

[47] R. Wang. *Orthogonal Transforms*. Cambridge University Press, 2012.

[48] A. Wegener. Block floating point compression of signal data, 2012. http://www.google.com/patents/US8301803.

[49] J. Woodring, S. Mniszewski, C. Brislawn, D. DeMarle, and J. Ahrens. Revisiting wavelet compression for large-scale climate data using JPEG 2000 and ensuring data precision. In *IEEE Large Data Analysis and Visualization*, pages 31–38, 2011.

[50] B.-L. Yeo and B. Liu. Volume rendering of DCT-based compressed 3D scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43, 1995.