Alternatives to IEEE: NextGen Number Formats for Scientific Computing

IPAM Workshop II: HPC and Data Science for Scientific Discovery

Peter Lindstrom

Jeff Hittinger, Matt Larsen, Scott Lloyd, Markus Salasoo

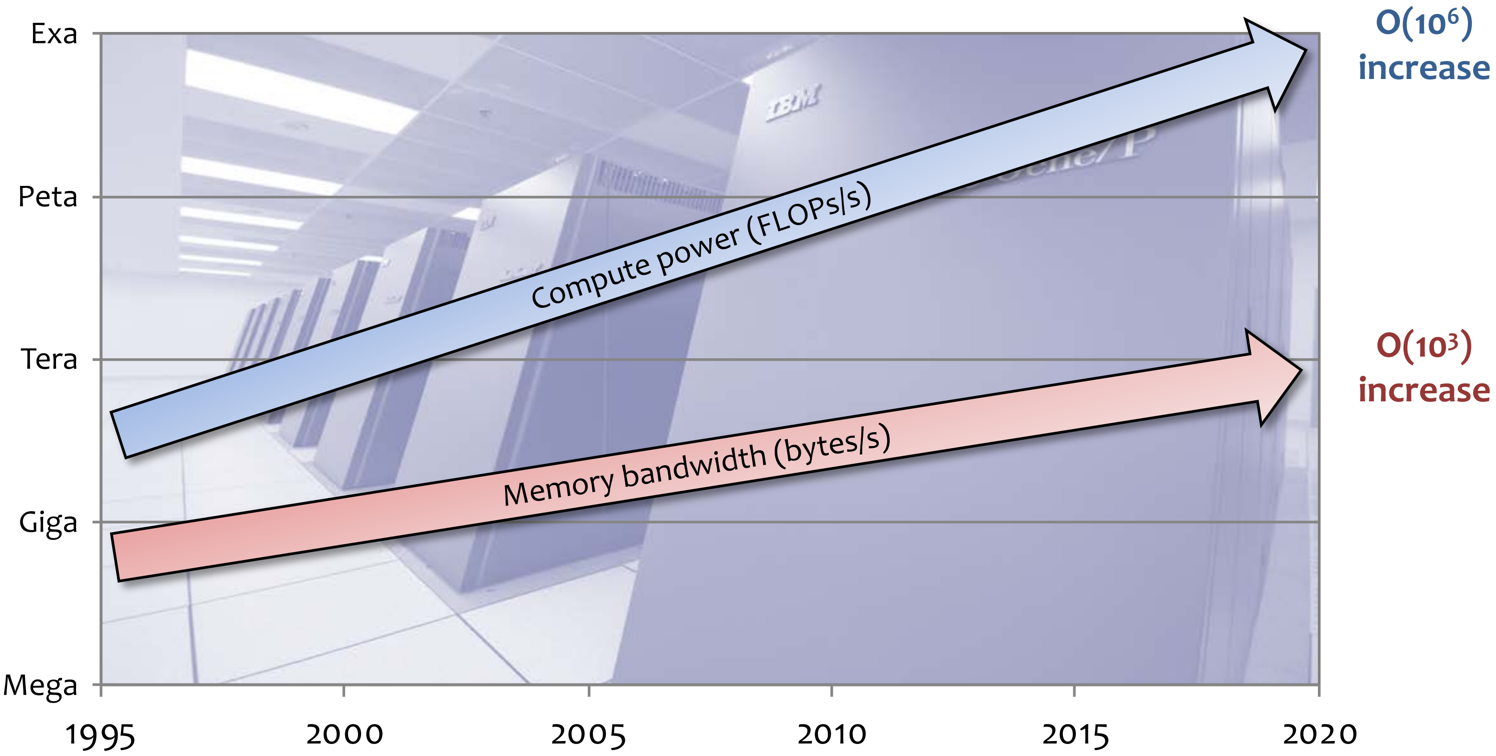
October 15, 2018

LLNL-PRES-759606

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Data movement will dictate performance and power usage at exascale



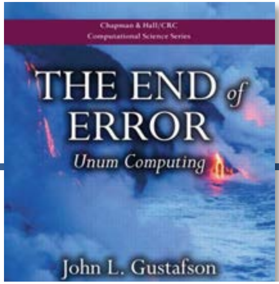
Uneconomical floating types and excessive precision are contributing to data movement

- IEEE assumes "one exponent size fits all"
  - **Insufficient precision** for some computations, e.g. catastrophic cancellation
  - **Insufficient dynamic range** for others, e.g. under- or overflow in exponentials
  - For these reasons, double precision is preferred in scientific computing

- Many bits of a double are contaminated by error
  - Error from round-off, truncation, iteration, model, observation, …
  - Pushing error around and computing on it is wasteful

- To significantly reduce data movement, we must **make every bit count!**
  - Next generation floating types must be re-designed from ground up

POSITS [Gustafson 2017] are a new float representation that improves on IEEE



	IEEE 754 floating point	Posits: UNUM version 3.0
Bit Length	{16, 32, 64, 128}	fixed length but arbitrary
Sign	sign-magnitude (−0 ≠ +0)	two's complement (−0 = +0)
Exponent	fixed length (biased binary)	variable length (Golomb-Rice)
Fraction Map	linear ($\varphi(x) = 1 + x$)	linear ($\varphi(x) = 1 + x$)
Infinities	{−∞, +∞}	±∞ (single point at infinity)
NaNs	many (9 quadrillion)	one
Underflow	gradual (subnormals)	gradual (natural)
Overflow	1 / FLT_TRUE_MIN = ∞ (oops!)	never (exception: 1 / 0 = ±∞)
Base	{2, 10}	$2^{2^m} \in \{2, 4, 16, 256, \dots\}$

In terms of encoding scheme, IEEE and POSITS differ in one important way

- Both represent real value as $y = (-1)^s\, 2^e\, (1 + f)$
- **IEEE:** fixed-length **binary** encoding of exponent
- **POSITS:** variable-length **Golomb-Rice** encoding of exponent
  - Exponent, e, encoded as m trailing bits in binary ($e \bmod 2^m$), $1 + \lfloor e / 2^m \rfloor$ leading bits in unary
  - More fraction bits are available when exponent is small

IEEE float	IEEE double	POSIT(7)
$e = D(1\ x) = 1 + x$ 1-bit exponent "sign" 7-bit exponent magnitude x	$e = D(1\ 000\ x) = 1 + x$	$e = D(1\ 0\ x) = x$
	$e = D(1\ 001\ x) = 129 + x$	$e = D(1\ 10\ x) = 128 + x$
	$e = D(1\ 010\ x) = 257 + x$	$e = D(1\ 110\ x) = 256 + x$
	…	…
	$e = D(1\ 111\ x) = 897 + x$	$e = D(1\ 11111110\ x) = 896 + x$

Related work: Universal coding of positive integers [Elias 1975]

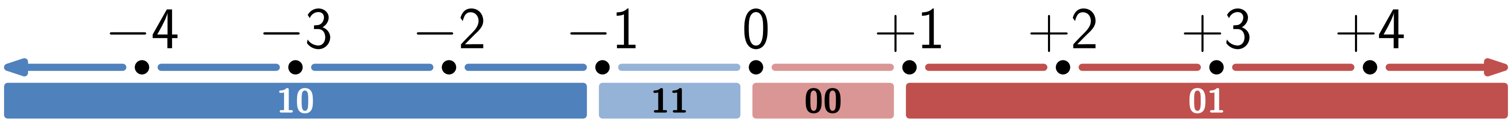
- Decompose integer $z \geq 1$ as $z = 2^e + r$
  - $e \geq 0$ is **exponent**
  - $r = z \bmod 2^e$ is e-bit **residual**

- Elias proposed three "prefix-free" variable-length codes for integers
  - Elias γ code is equivalent to POSIT(0) [Gustafson & Yonemoto 2017]
  - Elias δ code is equivalent to URR [Hamada 1983]

	Elias γ code	Elias δ code	Elias ω code
Exponent	unary	γ	ω (recursive)
Code(1)	0	0	0
Code($2^e + r$)	1^e 0 r	1 $\gamma(e)$ r	1 $\omega(e)$ r
Code($17 = 2^4 + 1$)	1111 0 0001	1 11 0 00 0001	1 1 1 0 0 00 0001

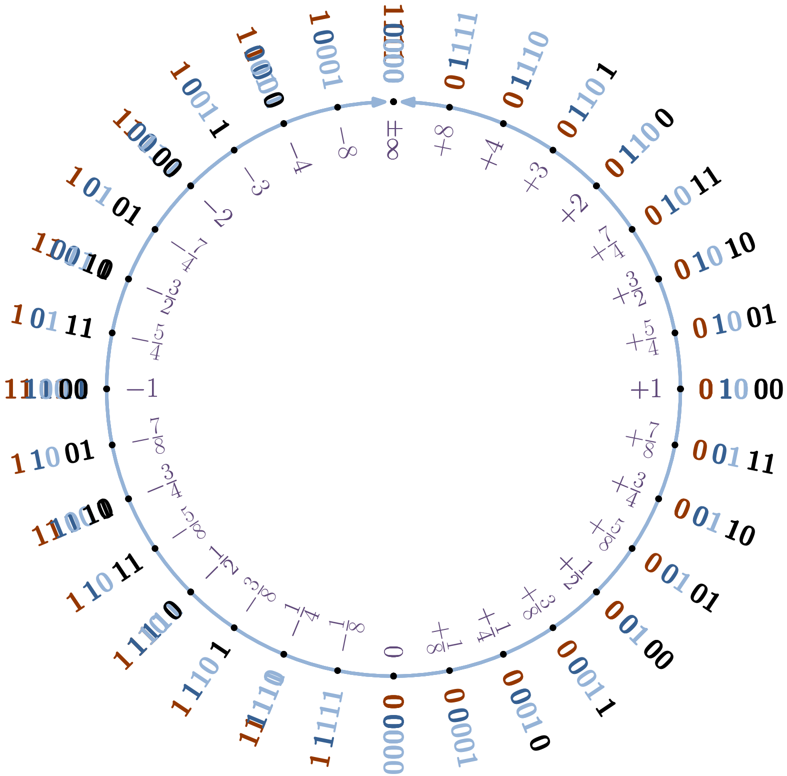
We extend Elias codes from positive integers to reals

- From $z \in \mathbb{N}$ (positive integers) to $y \in \mathbb{R}$ (reals)
  - **Rationals**: $1 \le z < y < z + 1$ encoded by appending fraction bits (as in IEEE, POSITS)
  - **Subunitaries**: $0 < y < 1$ encoded via two's complement exponent (as in POSITS)
  - **Negatives**: $y < 0$ encoded via two's complement binary representation (as in POSITS)
  - 2-bit prefix tells where we are on the real line (as in POSITS)



POSITS, Elias recursively refine intervals (0, ±1), (±1, ±∞)
Example: base-2 posits (aka. Elias γ)

- sign bit
- exponent sign
- exponent value
- fraction value



Beyond POSITS and Elias: NUMREP templated framework

- NUMREP is a modular C++ framework for defining number systems
  - **Exponent coding scheme** (unary, binary, Golomb-Rice, gamma, omega, …)
  - **Fraction map** (linear, reciprocal, exponential, rational, quadratic, logarithmic, …)
    - *Conjugate* fraction maps for sub- & superunitaries enable reciprocal closure
  - Handling of **under- & overflow**
  - **Rounding** rules (to nearest, toward $\{-\infty, 0, +\infty\}$)

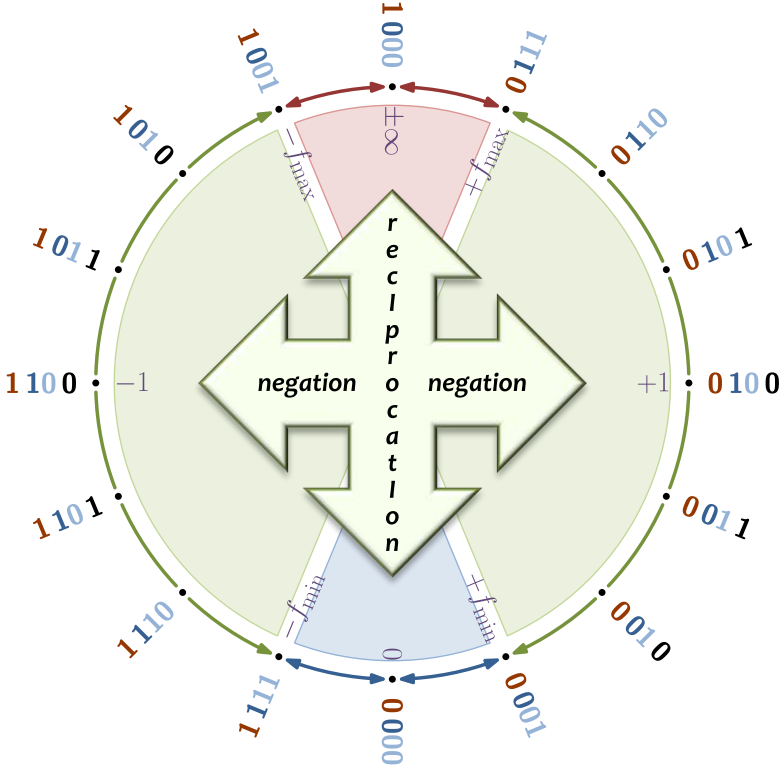
- NUMREP unifies IEEE, POSITS, Elias, URR, LNS, … , under a single schema
  - Uses auxiliary **arithmetic type** to perform numerical computations (e.g. IEEE, MPFR)
  - Uses **operator overloading** to mimic intrinsic floating types
  - Supports 8, 16, 32, 64-bit types

Lindstrom et al., "Universal coding of the reals: Alternatives to IEEE floating point," *Conference for Next Generation Arithmetic*, 2018

NUMREP is a powerful but complex framework—let's take a simplified view

Many useful NUMREPS are given by simple interpolation and extrapolation rules (plus closure under negation)

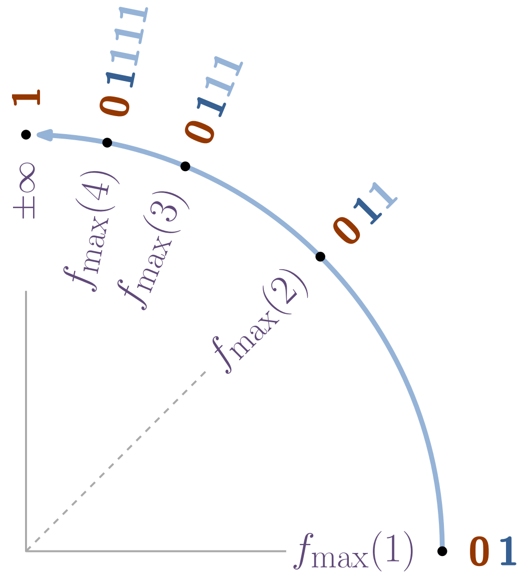
- extrapolation on $(f_{\max}, \infty)$
- interpolation on $(f_{\min}, f_{\max})$
- reciprocation on $(0, f_{\min})$



Extrapolation gives next $f_{\max}$ between previous $f_{\max}$ and $\pm\infty$ when adding one more bit of precision

- **Extrapolation rule:** $1 \le f_{\max}(p) < f_{\max}(p+1) < \infty$

type	$f_{\max}(p+1)$	sequence
Unary	$1 + f_{\max}(p)$	1, 2, 3, 4, 5, …
Elias γ	$2 \times f_{\max}(p)$	1, 2, 4, 8, 16, …
POSITS (base b)	$b \times f_{\max}(p)$	1, b, b^2, b^3, b^4, …
Elias δ	$f_{\max}(p)^2$	1, 2, 4, 16, 256, …
Elias ω	$2^{f_{\max}(p)}$	1, 2, 4, 16, 65536, …



- **Reciprocation rule:** $f_{\min}(p) = 1 / f_{\max}(p)$

Interpolation rule generates new number g(*a*, *b*) between two finite positive numbers *a* < *b*

- For POSITS, Elias {γ, δ, ω}:

$$g(a,b) = \begin{cases} \dfrac{a+b}{2} & \text{if } b \le 2a \\ 2^{g(\lg a,\, \lg b)} & \text{otherwise} \end{cases}$$

- Examples
  - $g(2, 4) = 3$ — arithmetic mean
  - $g(4, 16) = 2^{g(2,4)} = 2^3 = 8$ — geometric mean
  - $g(16, 65536) = 2^{g(4,16)} = 2^{2^{g(2,4)}} = 2^{2^3} = 2^8 = 256$ — "hypergeometric" mean (for ω only)

- For LNS:

$$g(a,b) = \sqrt{ab}$$

FLEX (FLexible EXponent): NUMREP on a shoestring

- Templated on ***generator*** that implements two rules
  - Interpolation function, $g(a, b)$
  - Extrapolation recurrence, $a_{i+1} = f(a_i) = g(a_i, \infty)$

- Rules + bisection implement encoding, decoding, rounding, fraction map
  - Recursively bisect $(-\infty, +\infty)$ using applicable interpolation or extrapolation rule
    - $g(-\infty, +\infty) = 0$
    - $g(0, \pm\infty) = \pm 1$
  - Exploit symmetry of negation, reciprocation
  - **Encode:** bracket number and recursively test if less (0) or greater (1) than bisection point
  - **Decode:** narrow interval toward lower (0) or upper (1) bound; return final lower bound
  - **Round:** round number to lower bound if less than bisection point, otherwise to upper
  - **Map:** given by interpolation rule

- NOTE: Not very performant, but great for experimentation & validation

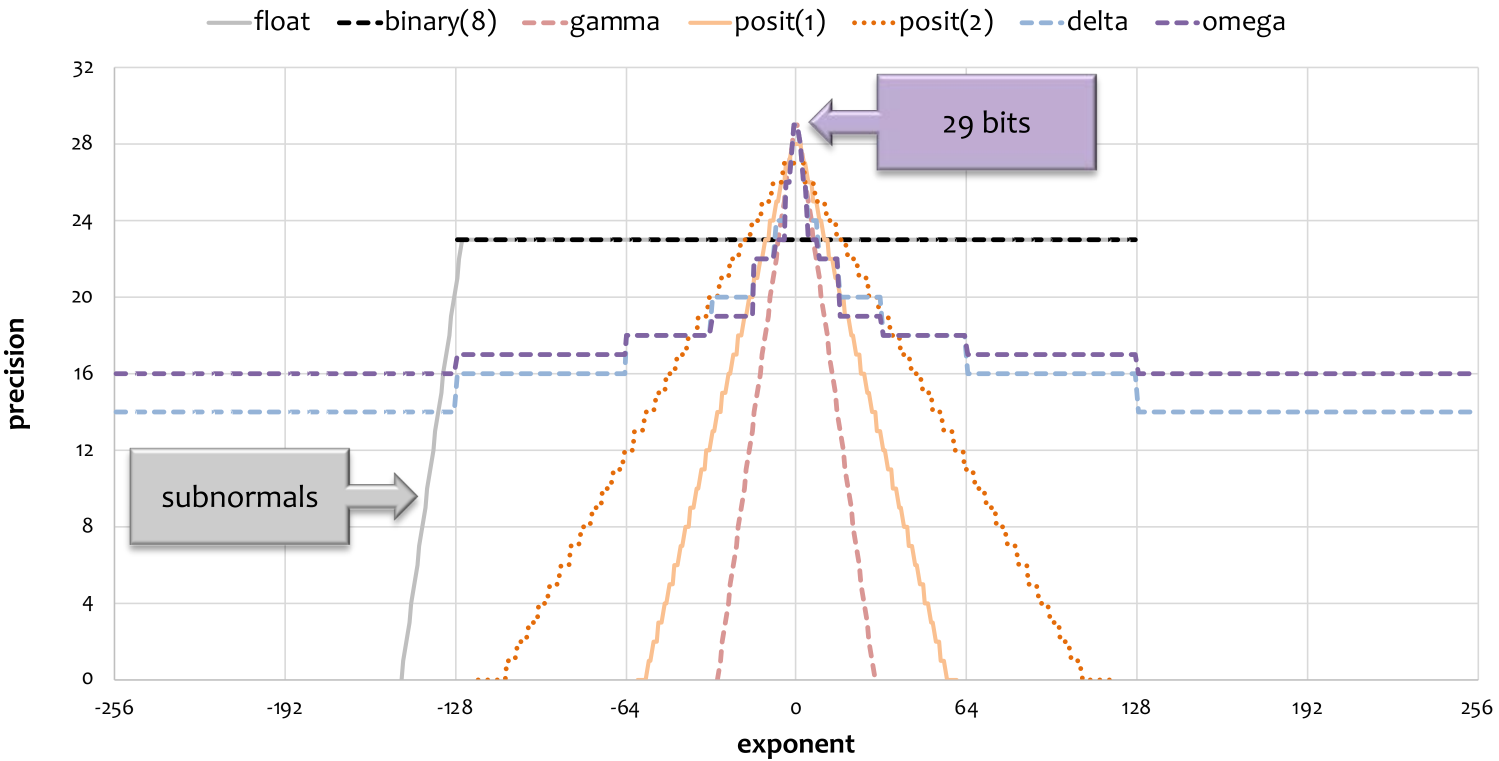
FLEX's POSIT implementation is 8 lines of code

```
// posit generator with m-bit exponent and base 2^(2^m)
template <int m = 0, typename real = flex_real>
class Posit {
public:
    real operator()(real x) const { return real(base) * x; }
    real operator()(real x, real y) const { return real(2) * x >= y ? (x + y) / real(2) : std::sqrt(x * y); }
protected:
    static const int base = 1 << (1 << m);
};
```

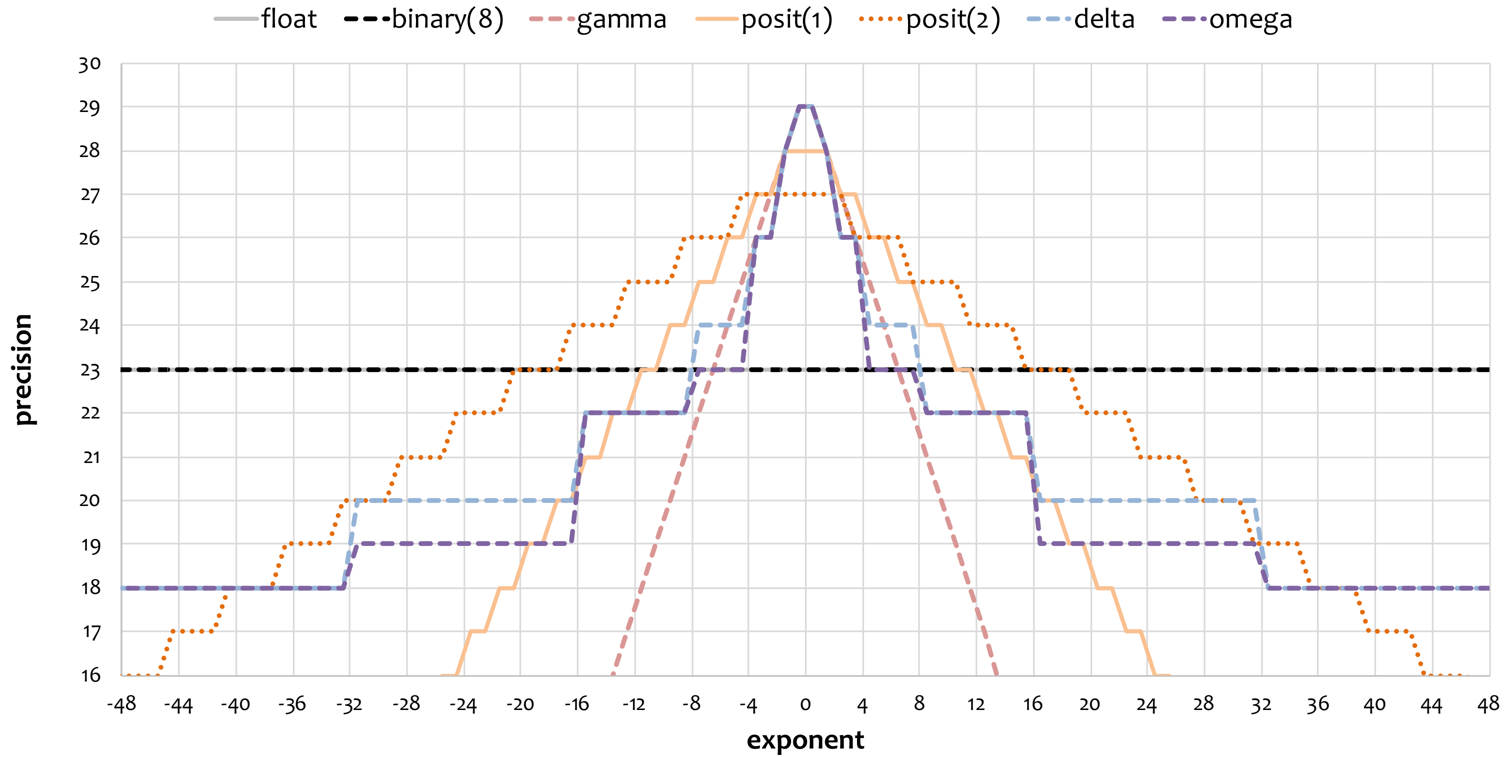
Elias delta and the Kessel Run: What is the distance to Proxima Centauri?

Quantity	Format	Sign	Exponent	Fraction	Error
1.3 parsecs	IEEE	0	01111111	01001100111101100101011	error = 1.2e−08
	ELIAS δ	0	$\underbrace{10}_{\text{2 bits}}$	01001100111101100101010**111000**	error = 4.8e−10
4.2 lightyears	IEEE	0	10000001	00001111011111101001000	error = 5.6e−08
	ELIAS δ	0	$\underbrace{11100}_{\text{5 bits}}$	00001111011111101001000**100**	error = 9.0e−11
4.0×10^{16} meters	IEEE	0	10110110	000111010010101000**10101**	error = 1.5e−08
	ELIAS δ	0	$\underbrace{1111111010111}_{\text{13 bits}}$	000111010010101001	error = 1.2e−06
2.4×10^{51} Planck lengths	IEEE	0	11111111	00000000000000000000000	error = ∞
	ELIAS δ	0	$\underbrace{11111111100101010}_{\text{17 bits}}$	**10101000110001**	error = 1.4e−05

Variable-length exponents lead to tapered precision: 6-9 more bits of precision than IEEE for numbers near one



Tapered precision (close-up): tension between precision and dynamic range

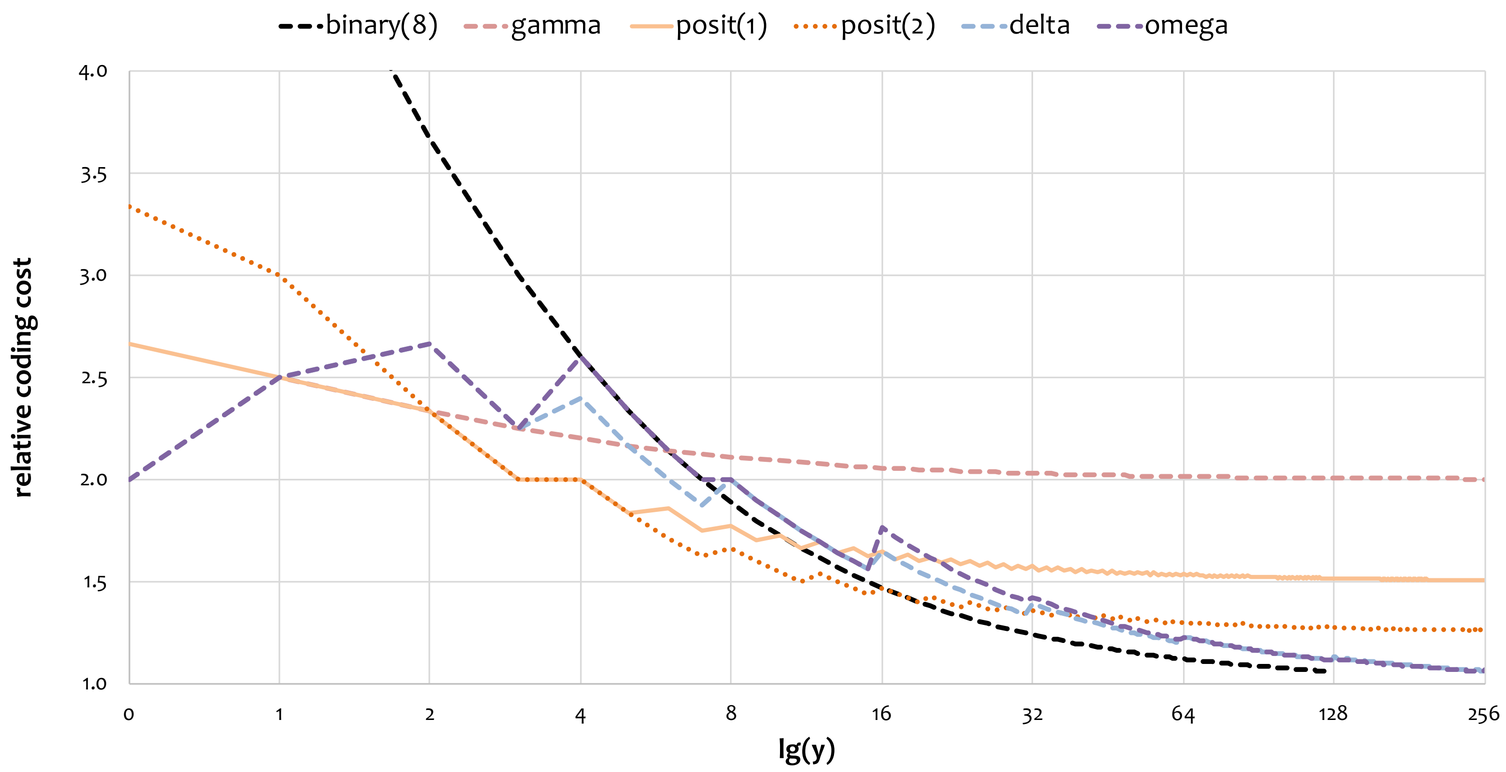


Elias delta and omega are asymptotically optimal universal representations

- Universality property (POSITS, Elias {γ, δ, ω}, but not IEEE)
  1. Each integer is represented by a ***prefix code***
     - No codeword is a prefix of any other codeword
     - Codeword length is self-describing—remaining available bits represent fraction
  2. Integer representation has ***monotonic length***
     - Larger integers have longer codewords
  3. Codeword length of integer *y* is $\boldsymbol{O(\lg y)}$
     - Codeword length is proportional to the number of significant bits

- Asymptotic optimality property (Elias {δ, ω})
  - Relative cost of coding exponent (leading-one position) vanishes in the limit

Elias delta and omega are asymptotically optimal: exponent overhead approaches zero in the limit



Fraction maps dictate how fraction bits map to values

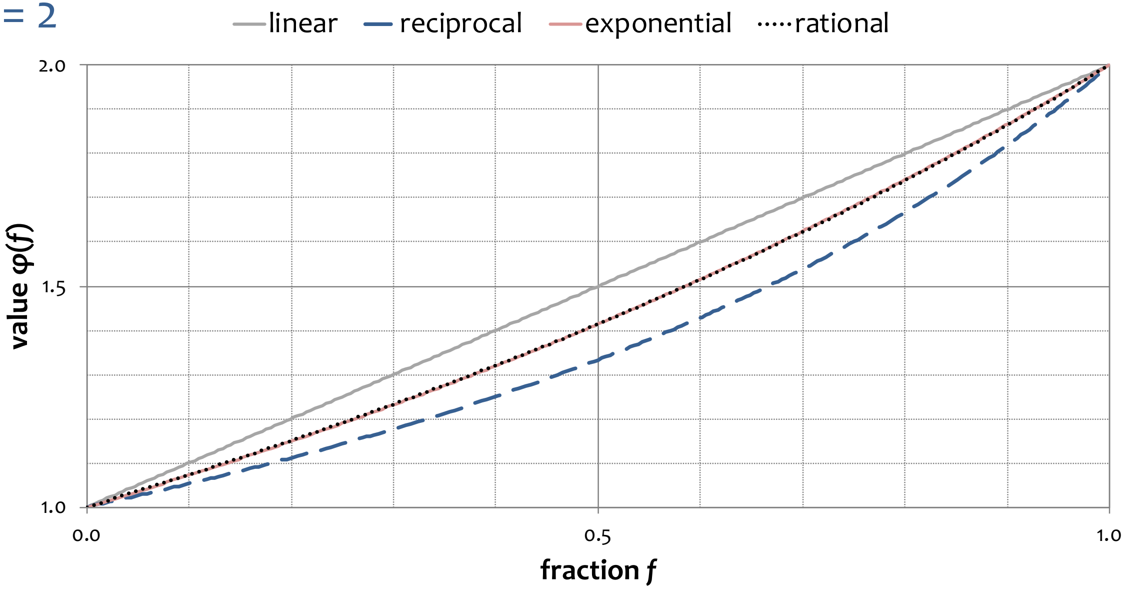
- So far we have assumed that fraction (significand), $f \in [0, 1)$, is mapped linearly
  - $y = (-1)^s\, 2^e\, (1 + f)$
  - We may replace $(1 + f)$ with transfer function $\varphi(f)$

- Fraction map, $\varphi : [0, 1) \longrightarrow [1, 2)$, maps fraction bits, f, to value, $\varphi(f)$
  - Subunitary map φ^-: $|y| < 1$
  - Superunitary map φ^+: $|y| \geq 1$

map	φ(f)	notes
Linear	$1 + f$	*introduces implicit leading-one bit*
Linear reciprocal	$2 / (2 - f)$	*conjugate with linear map ⇒ reciprocal closure*
Exponential	2^f	LNS *(logarithmic number system):* $y = 2^e\, 2^f = 2^{e+f}$
Linear rational	$(1 + p\, f) / (1 - q\, f)$	$p = \sqrt{2} - 1$, $q = (1 - p) / 2$

Conjugate pair (φ^-, φ^+) guarantees reciprocal closure

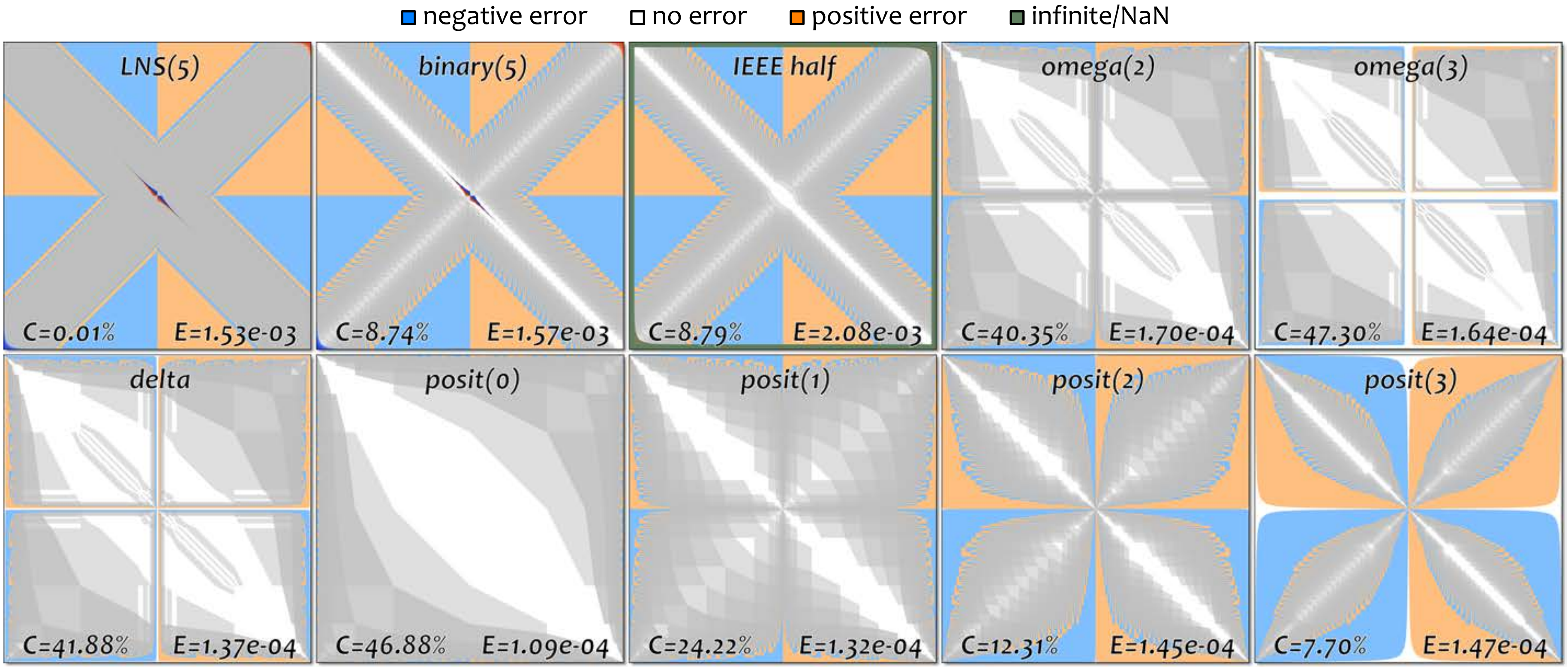
- Conjugacy: $\varphi^-(f)\ \varphi^+(1 - f) = 2 \quad 0 \le f \le 1$
  - E.g. $2 / (2 - f) \times (1 + (1 - f)) = 2$
- C^n continuity: $\varphi^{(n)}(1) / \varphi^{(n)}(0) = 2$



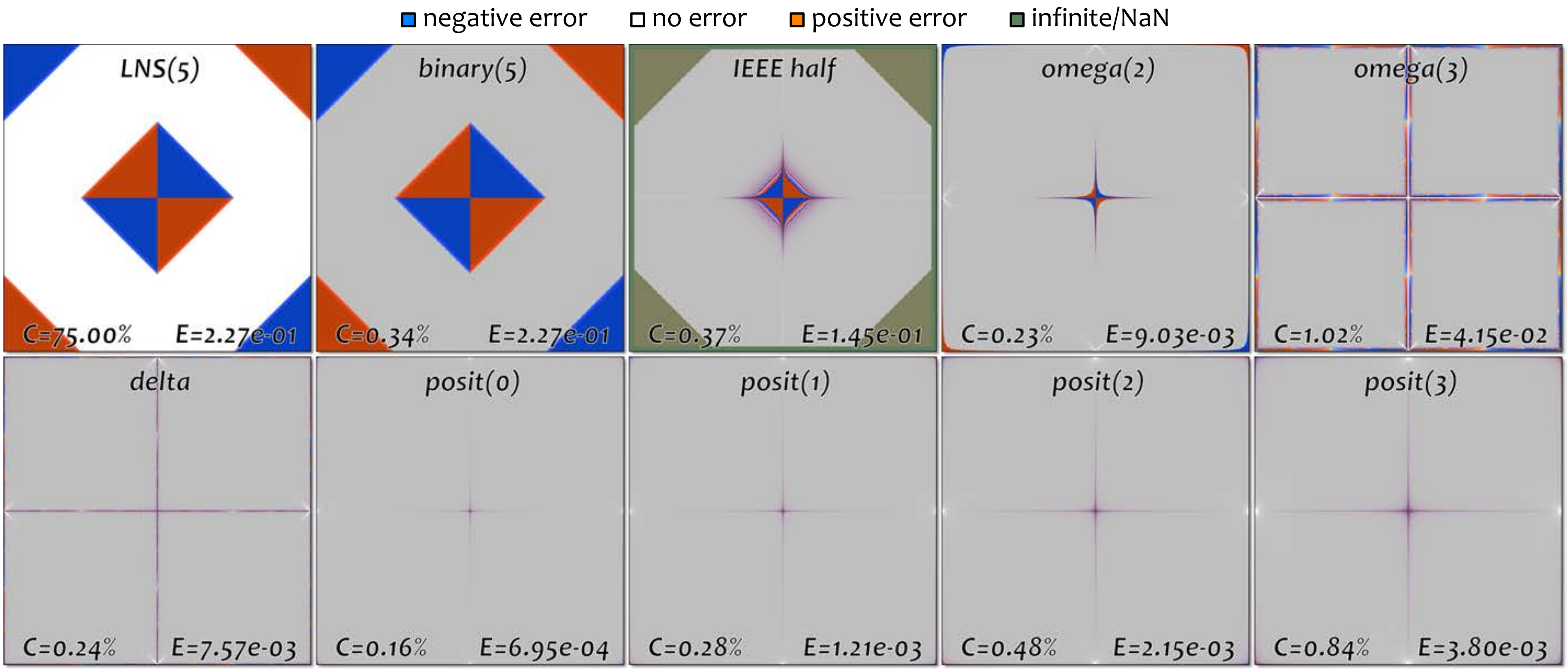
We present extensive results that span a spectrum of simple to increasingly complex numerical computations

- **Arithmetic closure** and relative error (vs. analytic solution) $z = x + y$
  - Addition and multiplication of 16-bit operands
- **Addition** of long floating-point sequences (vs. analytic solution) $s = \sum_i a_i$
  - Using naïve and best-of-breed addition algorithms
- **Matrix inversion** (vs. analytic solution) $A = H^{-1}$
  - LU-based inversion of Hilbert and Vandermonde matrices
- Symmetric matrix **eigenvalues** (vs. analytic solution) $\Lambda = QAQ^T$
  - QR-based dense eigenvalue solver
- Euler2D **PDE solver** (vs. IEEE quad precision) $\partial_t u + \nabla \cdot f(u) = 0$
  - Complex physics involving interacting shock waves

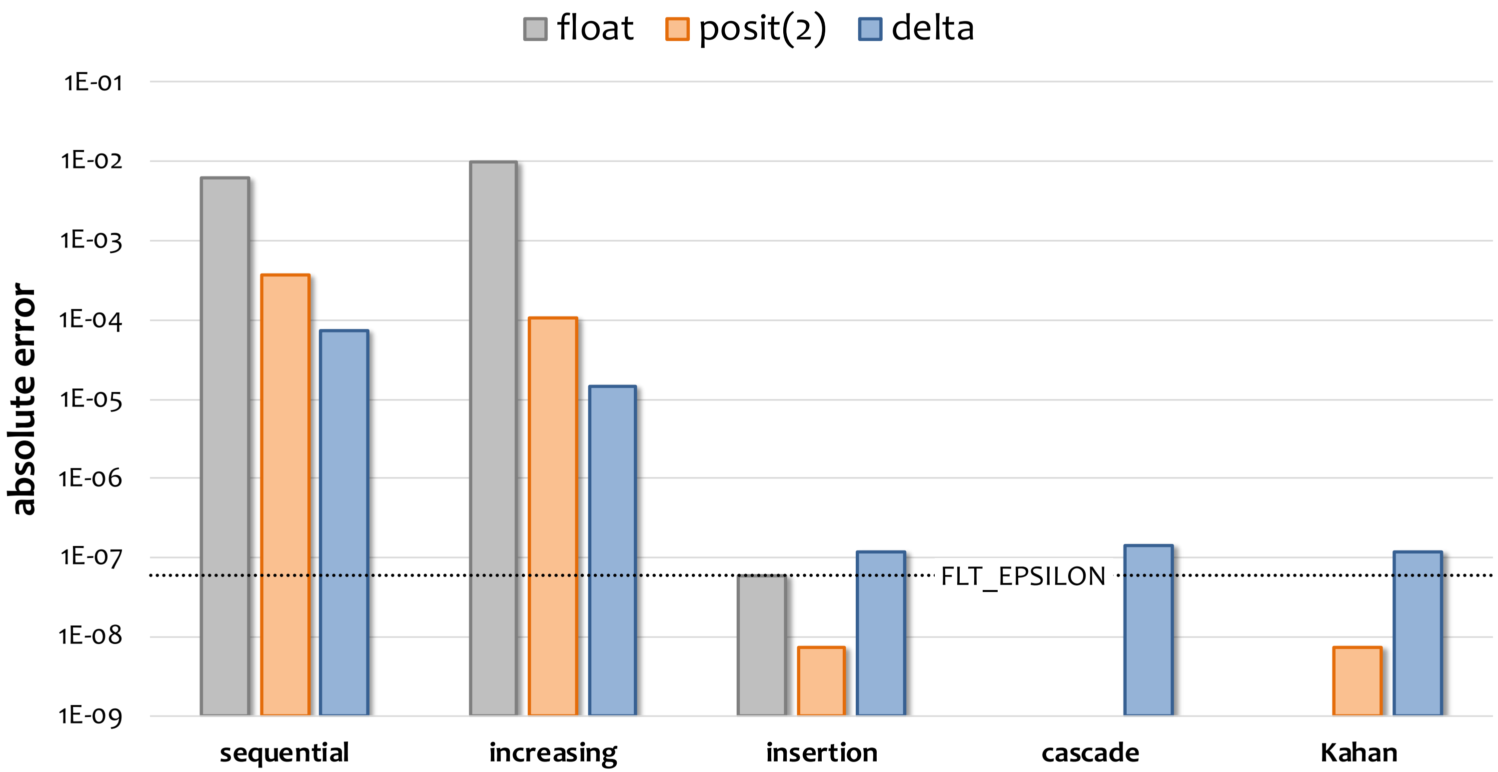
Closure rate (C) and mean relative error (E) for 16-bit addition suggest POSIT(0) (aka. Elias gamma) performs best



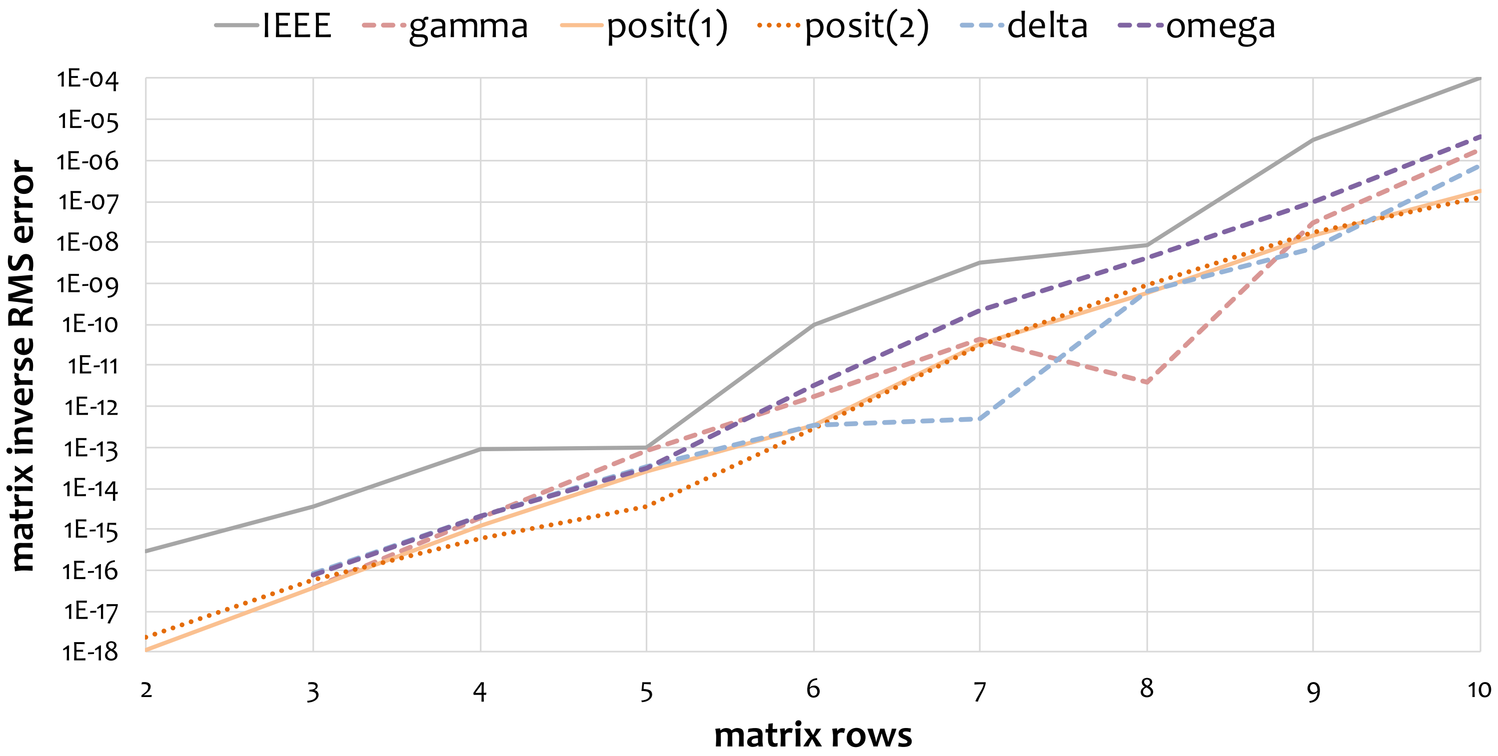
Optimal multiplicative closure of LNS does not imply superior relative errors



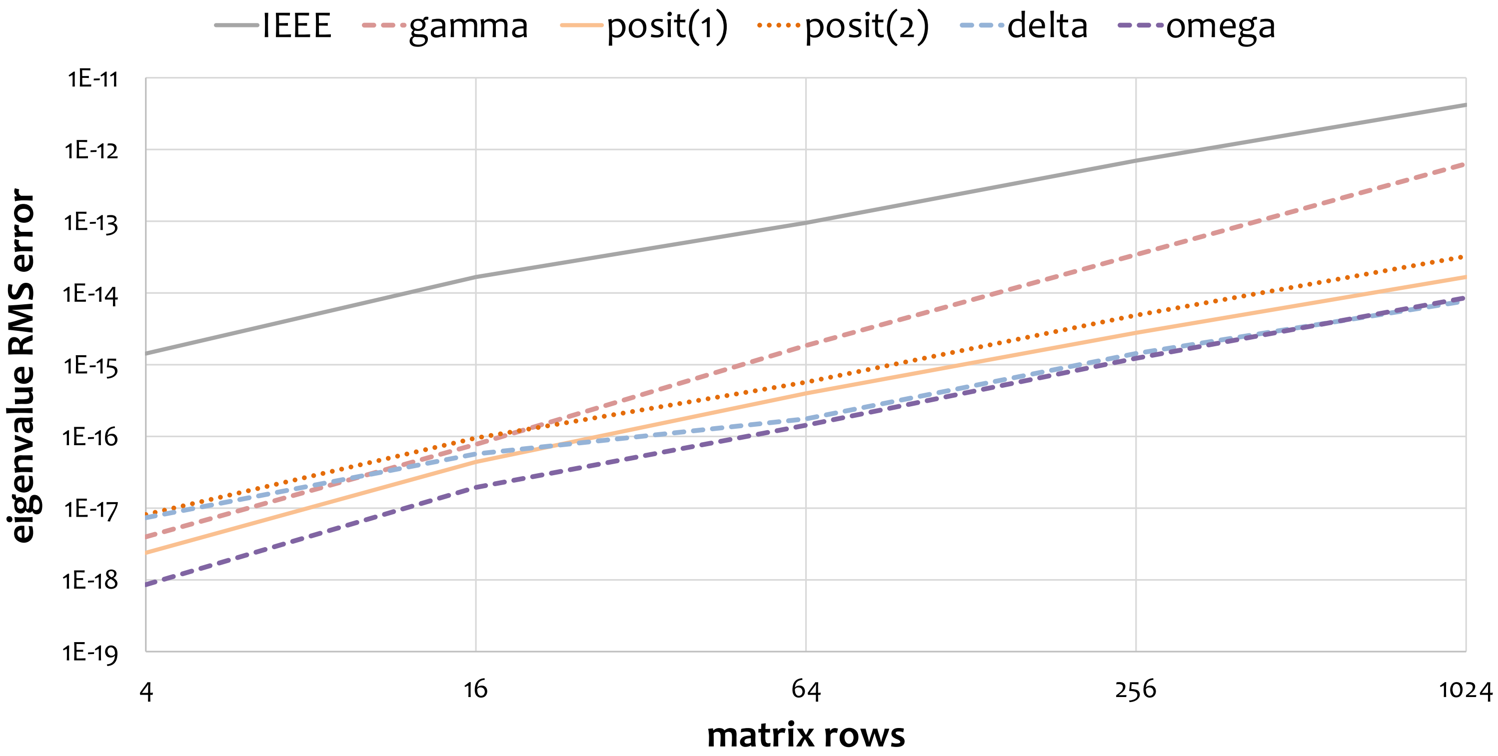
Simple summation algorithms benefit from tapered precision: Sum of Gaussian PDF and its first derivative



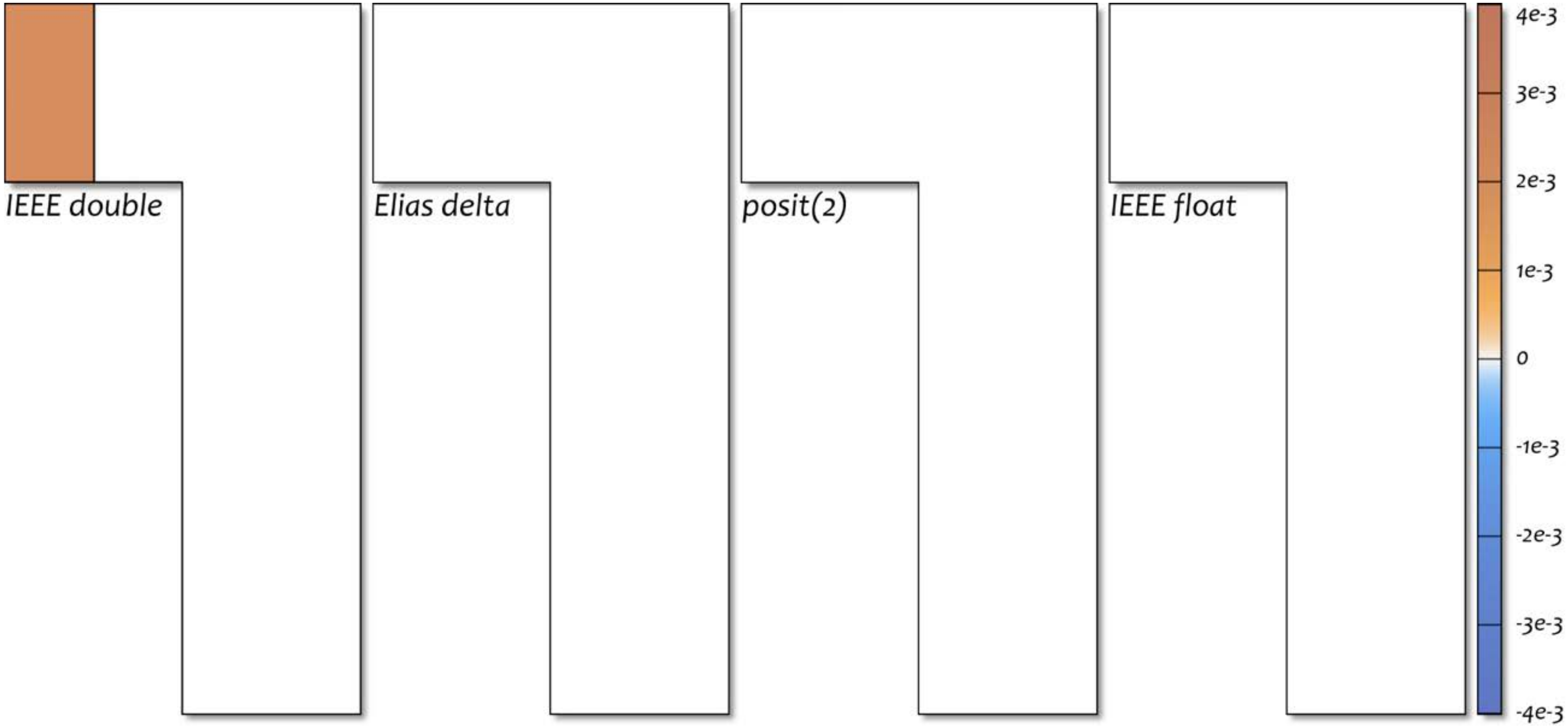
New types excel in 64-bit Hilbert matrix inversion (Eigen3)



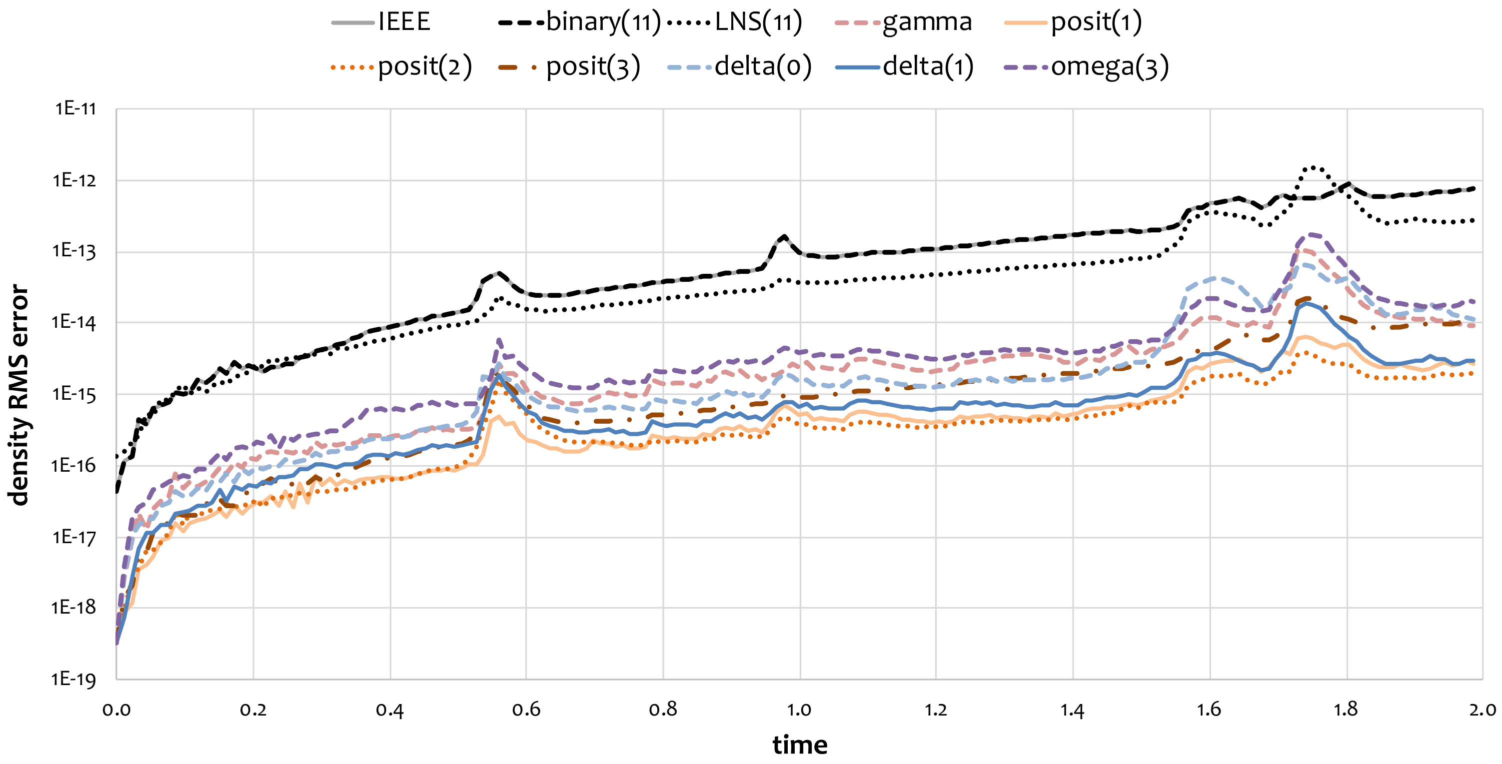
Elias outperforms IEEE by O(10^3) in 64-bit eigenvalue accuracy



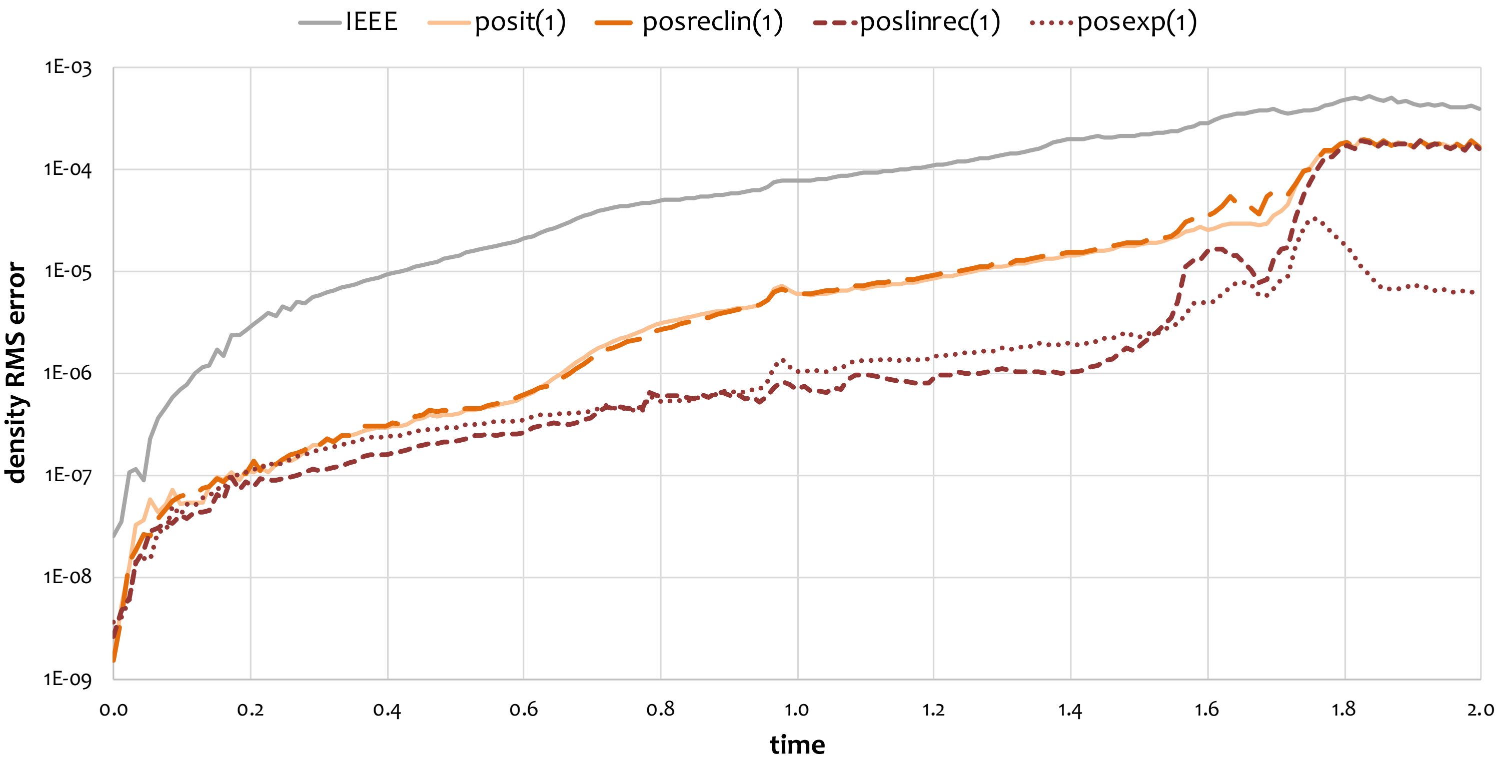
Euler2D shock propagation illustrates benefits of new types



IEEE consistently is the least accurate floating-point representation in numerical calculations (64-bit types)

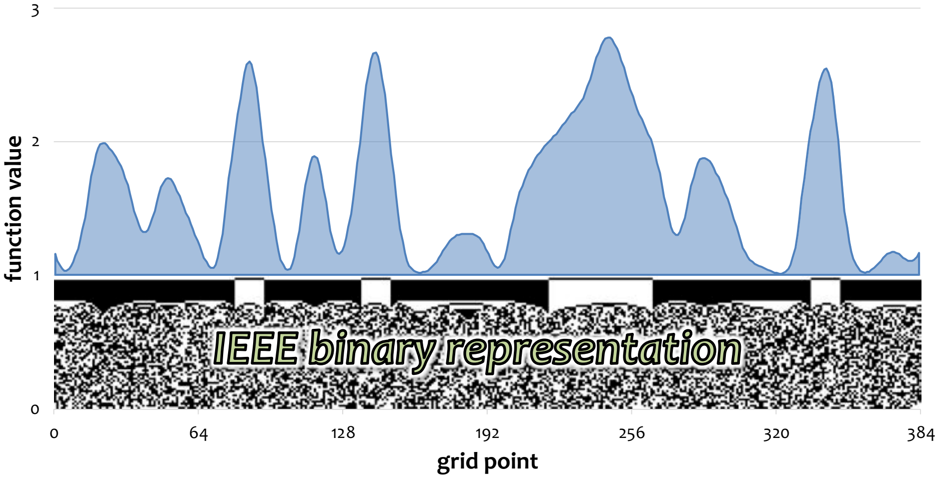


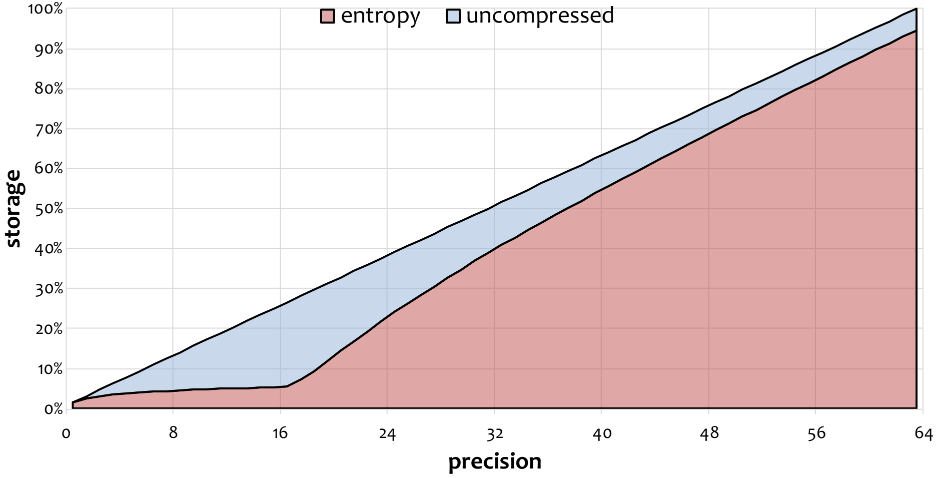
Nonlinear fraction maps sometimes improve accuracy substantially (32-bit types)



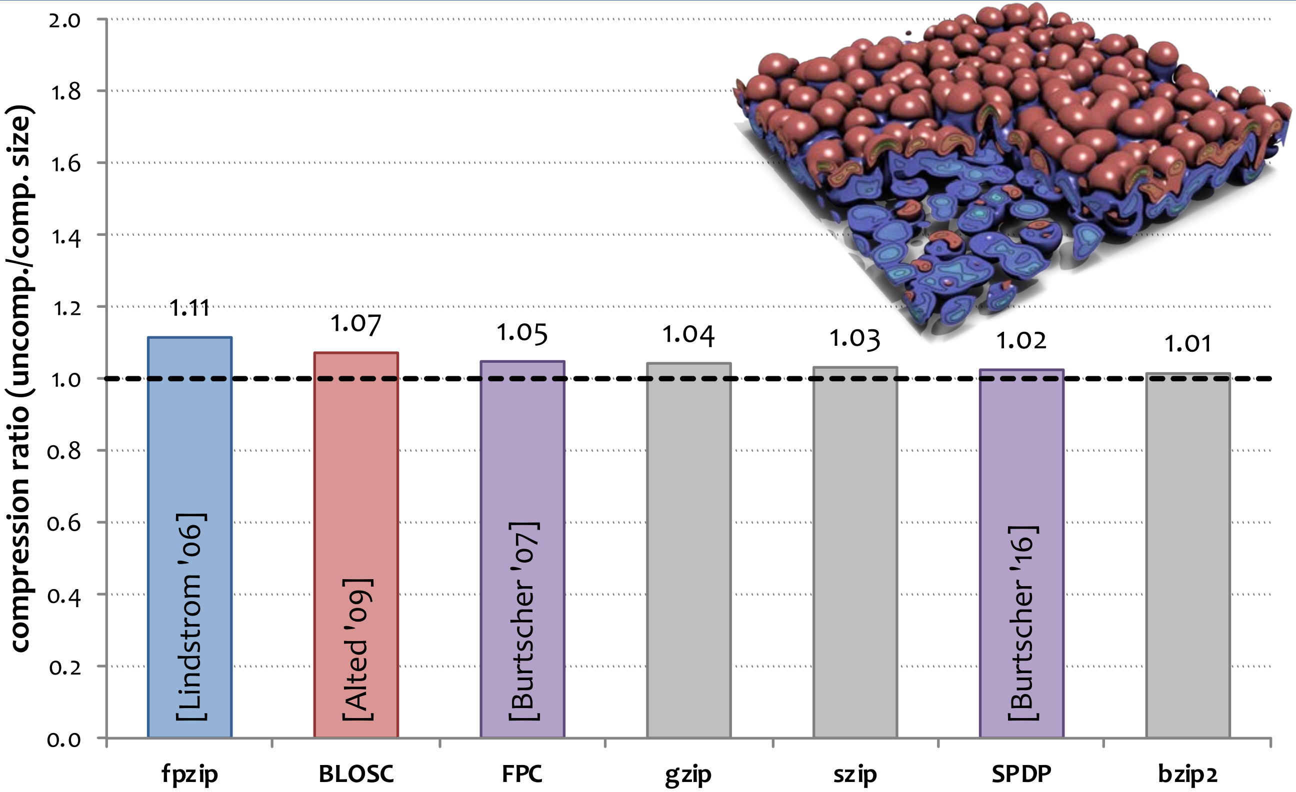
Beyond IEEE and posit representations of $\mathbb{R}$: Compressed representations of $\mathbb{R}^n$

- Many numerical computations involve continuous fields
  - E.g. partial differential equations over $\mathbb{R}^2$, $\mathbb{R}^3$
  - Nearby scalars on Cartesian grids have similar values
    - In statistical speak, fields exhibit ***autocorrelation***
    - Such redundancy is not exploited by any of the representations discussed so far



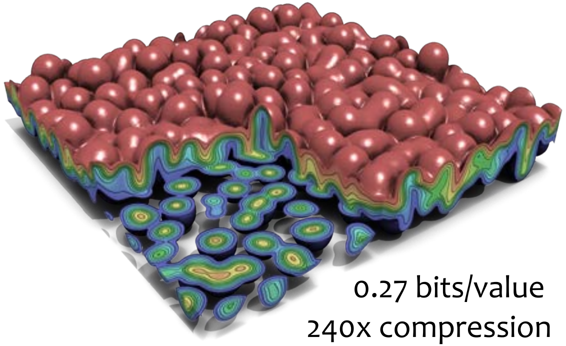


(64-bit) floating-point data does not compress well losslessly



Lossy compression enables greater reduction, but is often met with skepticism by scientists

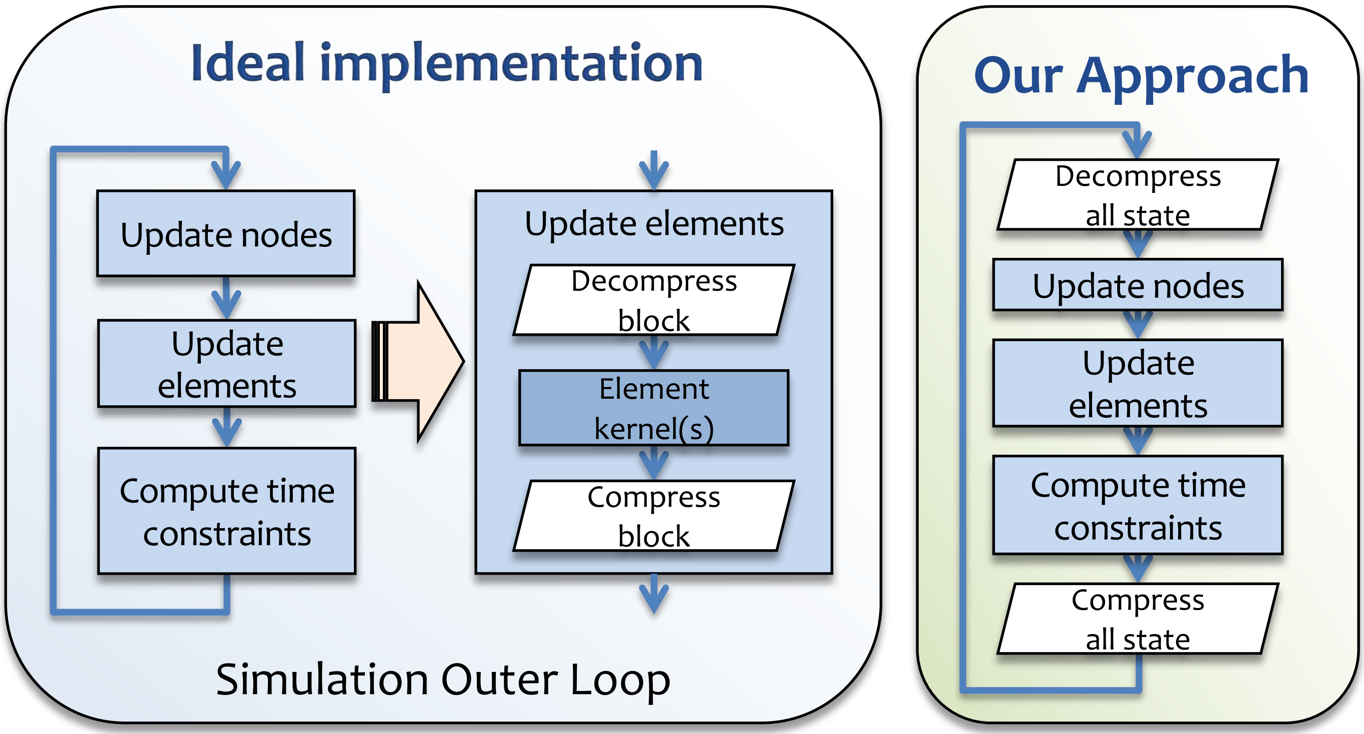
- Large improvements in compression are possible by allowing even small errors
  - Simulation often computes on meaningless bits
    - Round-off, truncation, iteration, model **errors abound**
    - Last few floating-point bits are effectively **random noise**



0.27 bits/value
240x compression

- Still, lossy compression often makes scientists nervous
  - Even though lossy **data reduction** is ubiquitous
    - **Decimation** in space and/or time (e.g. store every 100 time steps)
    - **Averaging** (hourly vs. daily vs. monthly averages)
    - **Truncation** to single precision (e.g. for history files)
  - State-of-the-art compressors support **error tolerances**

Can lossy-compressed in-memory storage of numerical simulation state be tolerated?



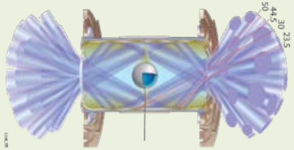
Using lossy FPZIP to store simulation state compressed, we have shown that 4x lossy compression can be tolerated

Lagrangian shock hydrodynamics

- QoI: *radial shock position*
- 25 state variables compressed over 2,100 time steps
- At **4x compression**, relative **error < 0.06%**

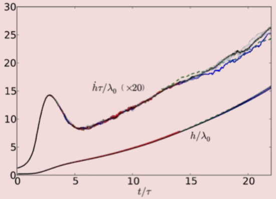
Laser-plasma multi-physics

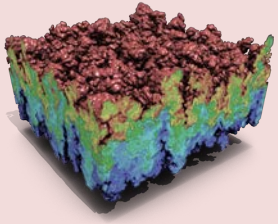
- QoI: *backscattered laser energy*
- At **4x compression**, relative **error < 0.1%**



High-order Eulerian hydrodynamics

- QoI: *Rayleigh-Taylor mixing layer thickness*
- 10,000 time steps
- At **4x compression**, relative **error < 0.2%**

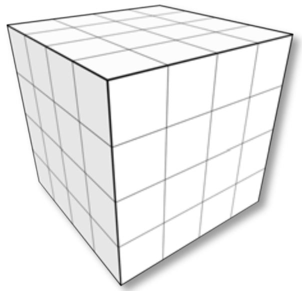




Lossy compression of state is viable, but streaming compression *increases* data movement—need inline compression

ZFP is an inline compressor for floating-point arrays

- Inspired by ideas from h/w texture compression
  - *d*-dimensional array divided into blocks of 4^d values
  - Each block is independently (de)compressed
    - e.g. to a user-specified number of bits or quality
  - Fixed-size blocks $\Rightarrow$ **random read/write access**
  - (De)compression is done **inline**, on demand
  - **Write-back cache** of uncompressed blocks limits data loss



- Compressed arrays via C++ operator overloading
  - Can be dropped into existing code by changing type declarations
  - double a[n] $\Leftrightarrow$ std::vector<double> a(n) $\Leftrightarrow$ zfp::array<double> a(n, **precision**)

Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, 2014

ZFP's C++ compressed arrays can replace STL vectors and C arrays with minimal code changes

// example using STL vectors

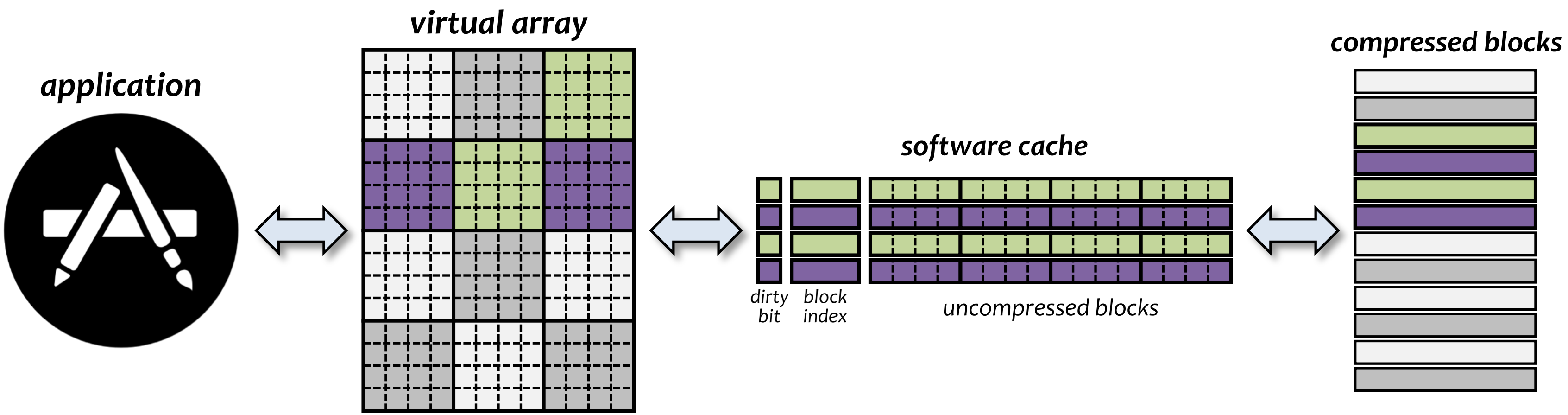
```
std::vector<double> u(nx * ny, 0.0);
u[x0 + nx*y0] = 1;
for (double t = 0; t < tfinal; t += dt) {
    std::vector<double> du(nx * ny, 0.0);
    for (int y = 1; y < ny - 1; y++)
        for (int x = 1; x < nx - 1; x++) {
            double uxx = (u[(x-1)+nx*y] - 2*u[x+nx*y] + u[(x+1)+nx*y]) / dxx;
            double uyy = (u[x+nx*(y-1)] - 2*u[x+nx*y] + u[x+nx*(y+1)]) / dyy;
            du[x + nx*y] = k * dt * (uxx + uyy);
        }
    for (int i = 0; i < u.size(); i++)
        u[i] += du[i];
}
```

// example using ZFP arrays

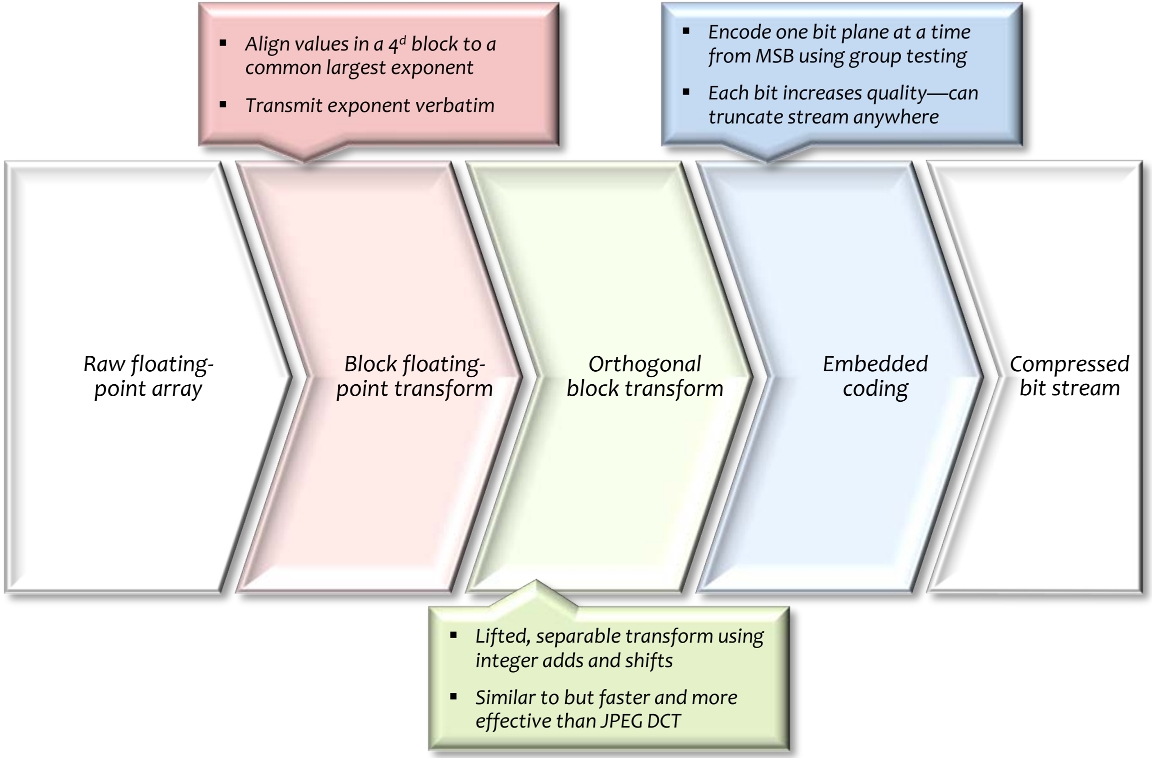
```
zfp::array2<double> u(nx, ny, bits_per_value);
u(x0, y0) = 1;
for (double t = 0; t < tfinal; t += dt) {
    zfp::array2<double> du(nx, ny, bits_per_value);
    for (int y = 1; y < ny - 1; y++)
        for (int x = 1; x < nx - 1; x++) {
            double uxx = (u(x-1, y) - 2*u(x, y) + u(x+1, y)) / dxx;
            double uyy = (u(x, y-1) - 2*u(x, y) + u(x, y+1)) / dyy;
            du(x, y) = k * dt * (uxx + uyy);
        }
    for (int i = 0; i < u.size(); i++)
        u[i] += du[i];
}
```

- required changes
- optional changes for improved readability

ZFP arrays limit data loss via a small write-back cache

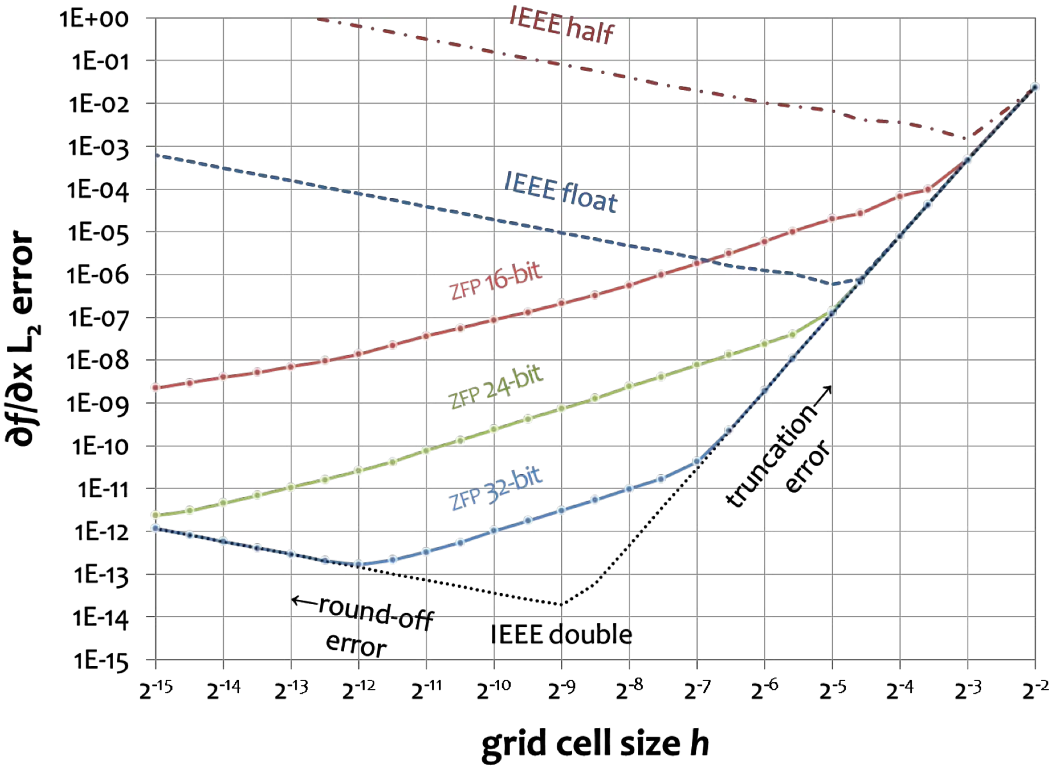


The ZFP compressor is comprised of three distinct components



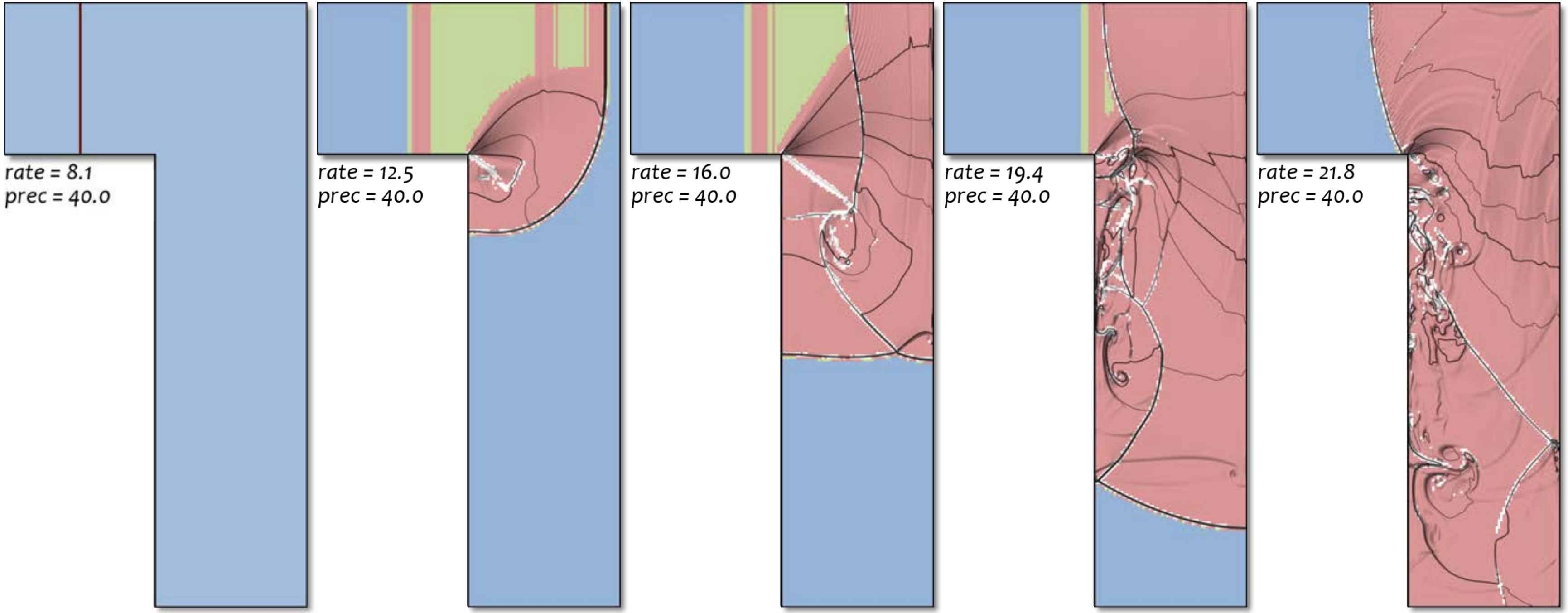
6-bit ZFP gives one more digit of accuracy than 32-bit IEEE in 2nd, 3rd derivative computations (Laplacian and highlight lines)

ZFP *improves* accuracy in finite difference computations using less precision than IEEE

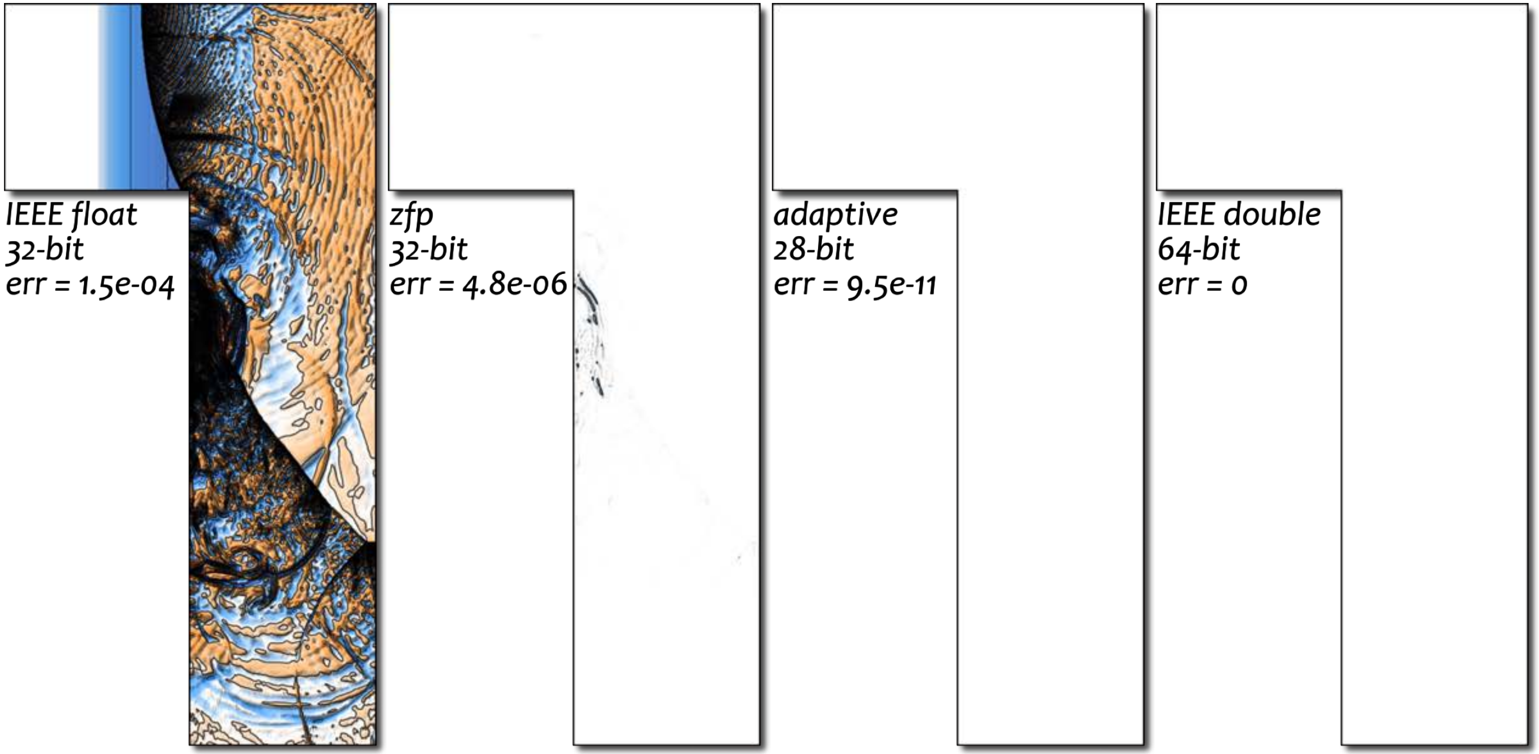


ZFP variable-rate arrays adapt storage spatially and allocate bits to regions where they are most needed

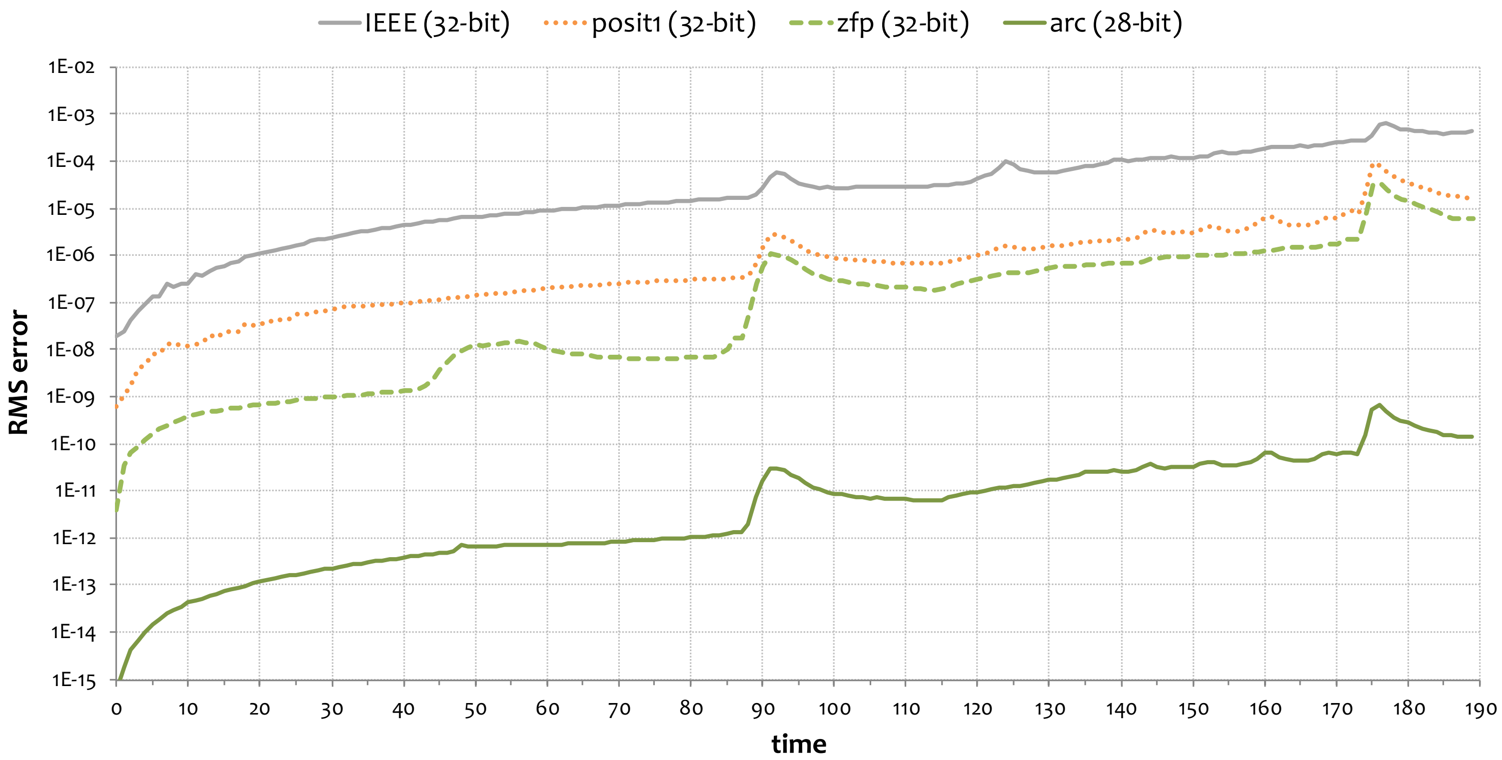
8 bits/value *16 bits/value* *32 bits/value* *64 bits/value*



ZFP adaptive arrays improve accuracy in PDE solution over IEEE by 6 orders of magnitude using less storage



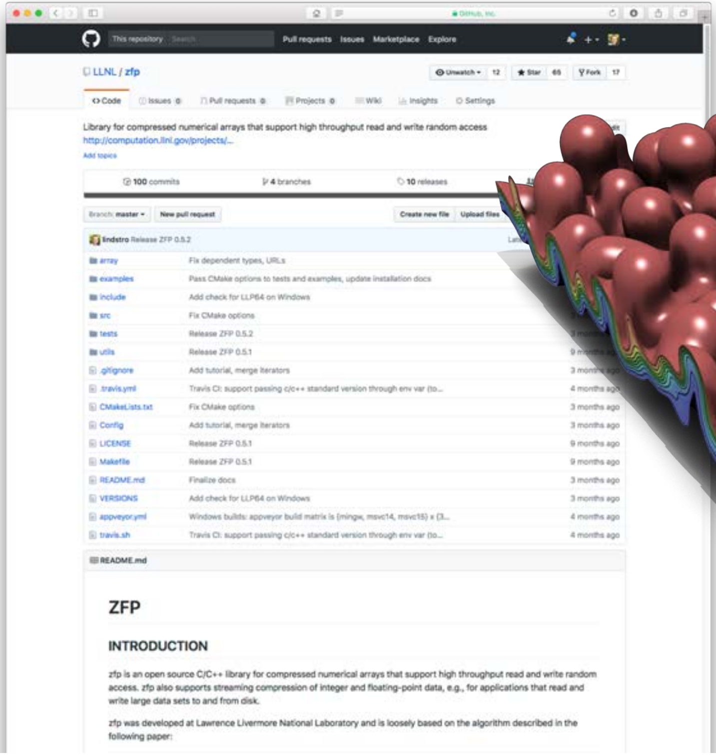
ZFP adaptive arrays improve accuracy in PDE solution over IEEE by 6 orders of magnitude using less storage

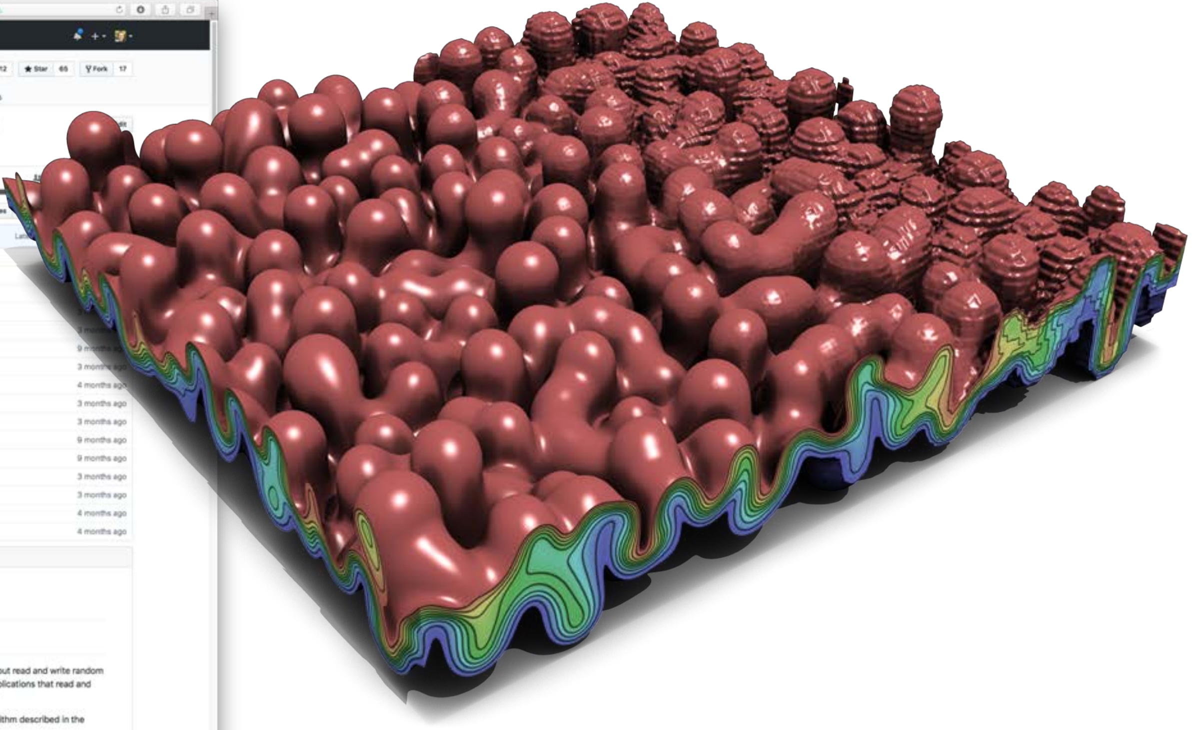


Conclusions: Novel scalar representations are driving research into new vector/matrix/tensor representations

- Emerging alternatives to IEEE floats show promising improvements in accuracy
  - **Tapered precision** allows for better balance between range and precision
  - New types have many desirable properties
    - No IEEE warts: *subnormals*, *excess of NaNs*, *signed zeros*, *overflow*, *exponent size dichotomy*, …
    - Features: *nesting*, *monotonicity*, *reciprocal closure*, *universality*, *asymptotic optimality*, …
- NumRep framework allows for experimentation with new number types in apps
  - **Modular design** supports plug-and-play of independent concepts
  - **Nonlinear maps** are expensive but worth investigating
  - POSITS are often among the best performing types; IEEE among the worst
- NextGen scalar representations still leave considerable room for improvement
  - ZFP fixed-rate compressed arrays consistently outperform IEEE and POSITS
  - ZFP **variable-rate arrays** optimize bit distribution and and **make very bit count**

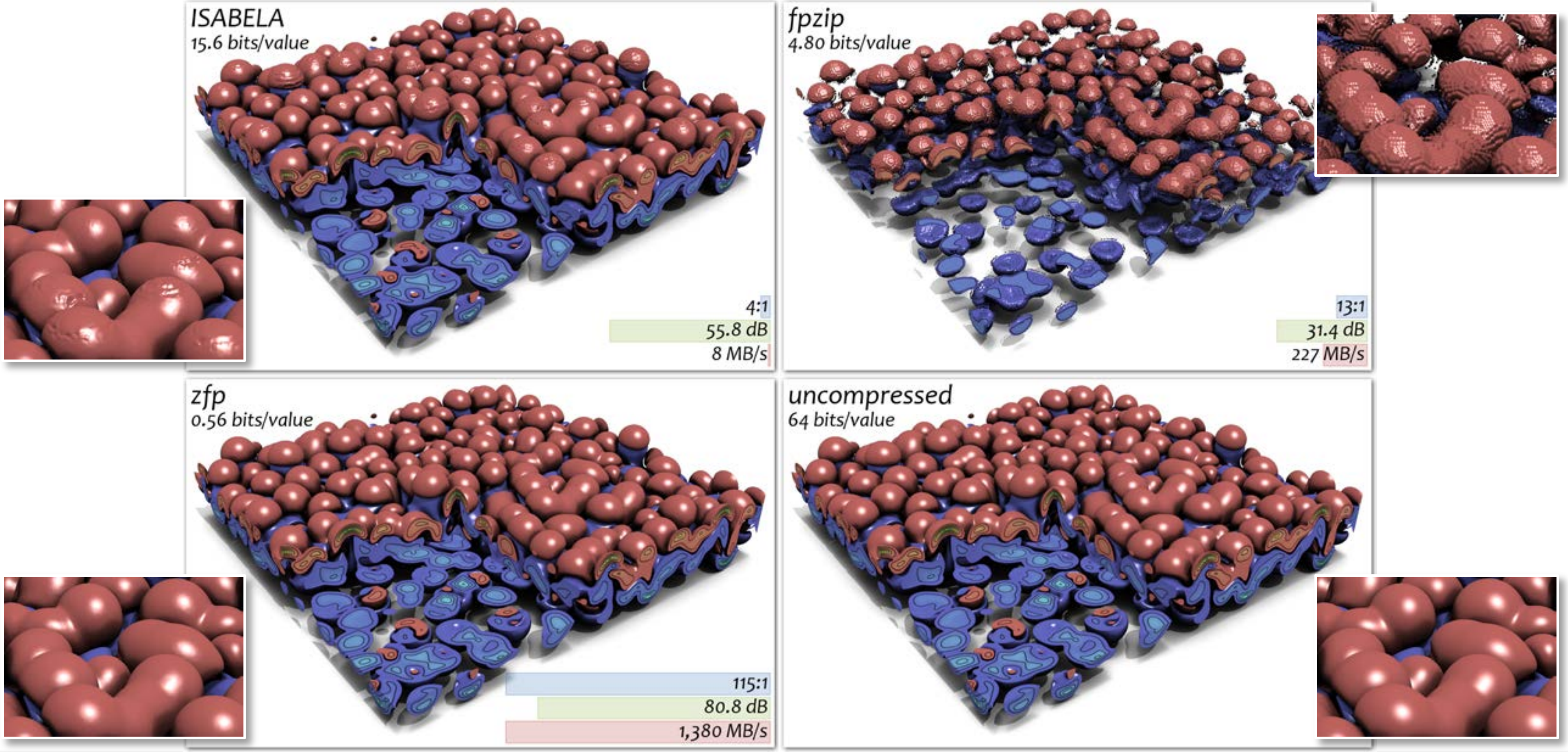
ZFP is available as open source on GitHub: https://github.com/LLNL/zfp



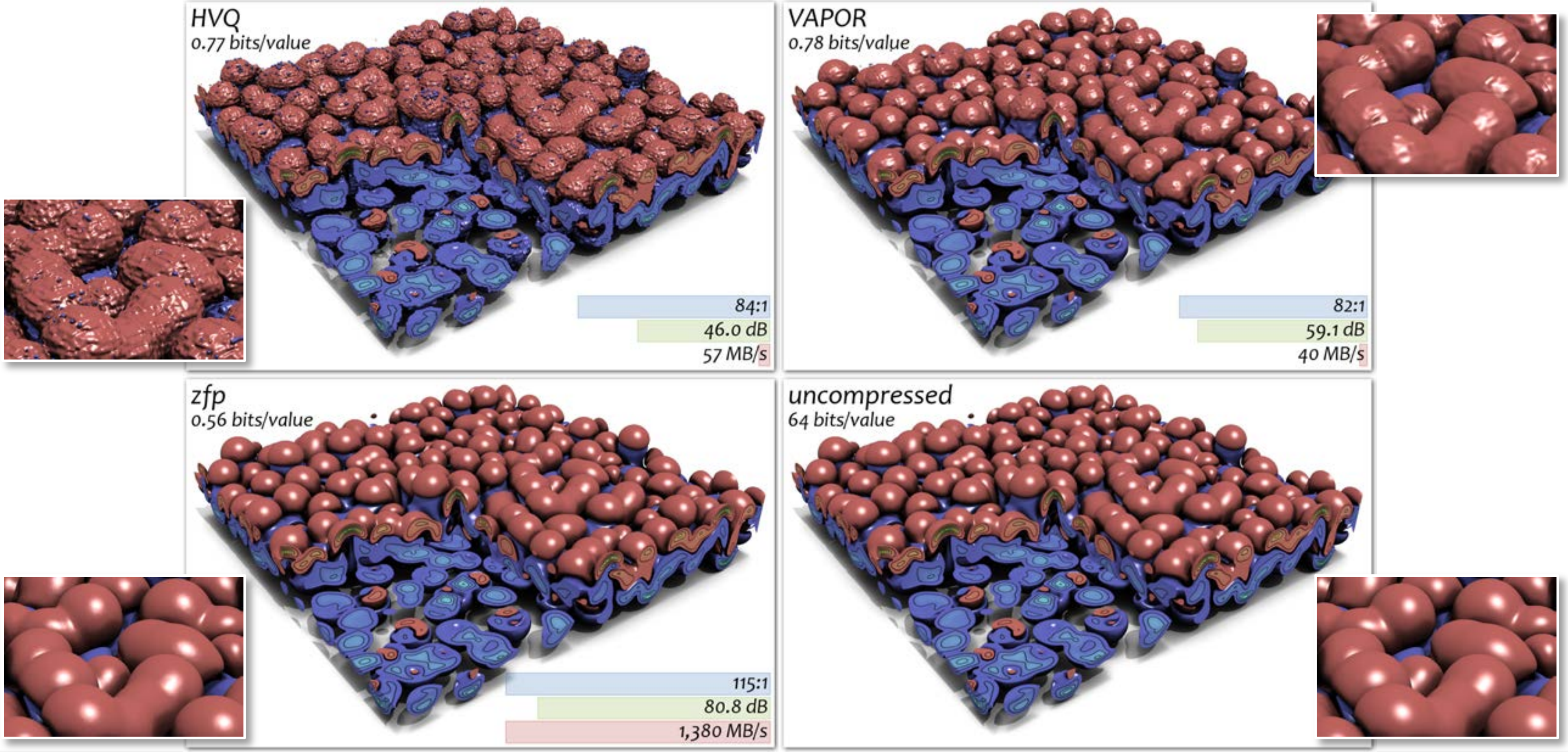


Lawrence Livermore National Laboratory

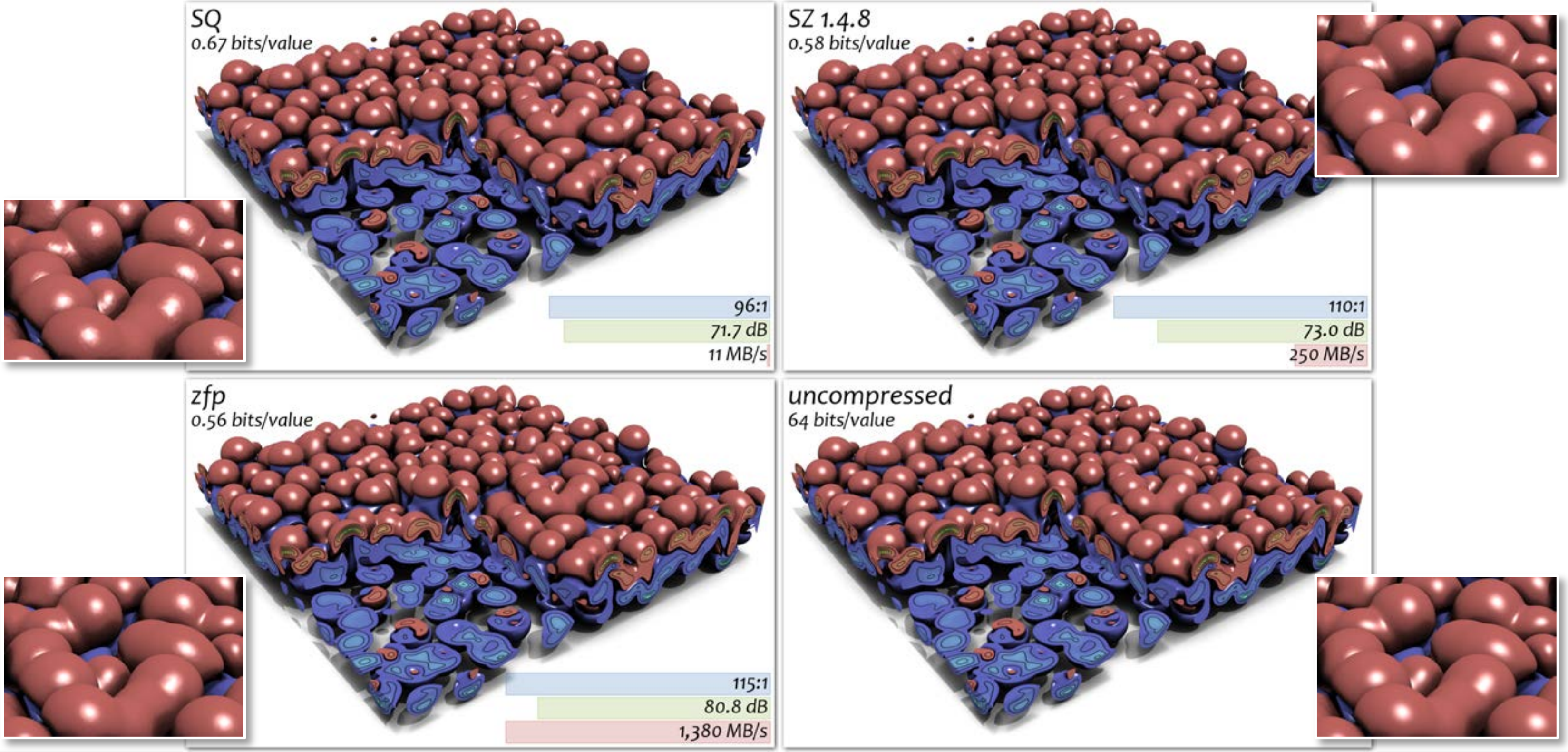
ZFP provides >100x compression with imperceptible loss in visual quality



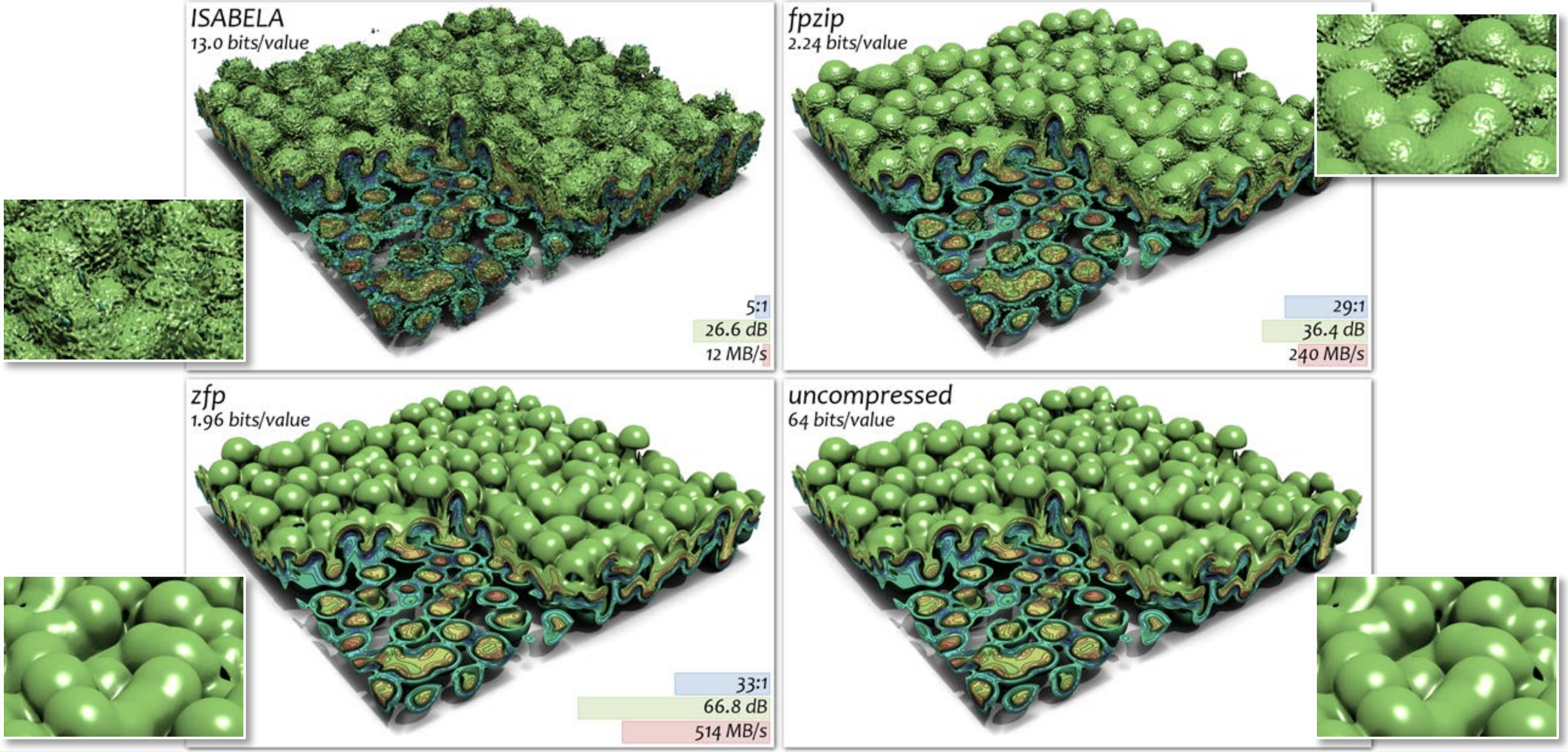
ZFP provides >100x compression with imperceptible loss in visual quality



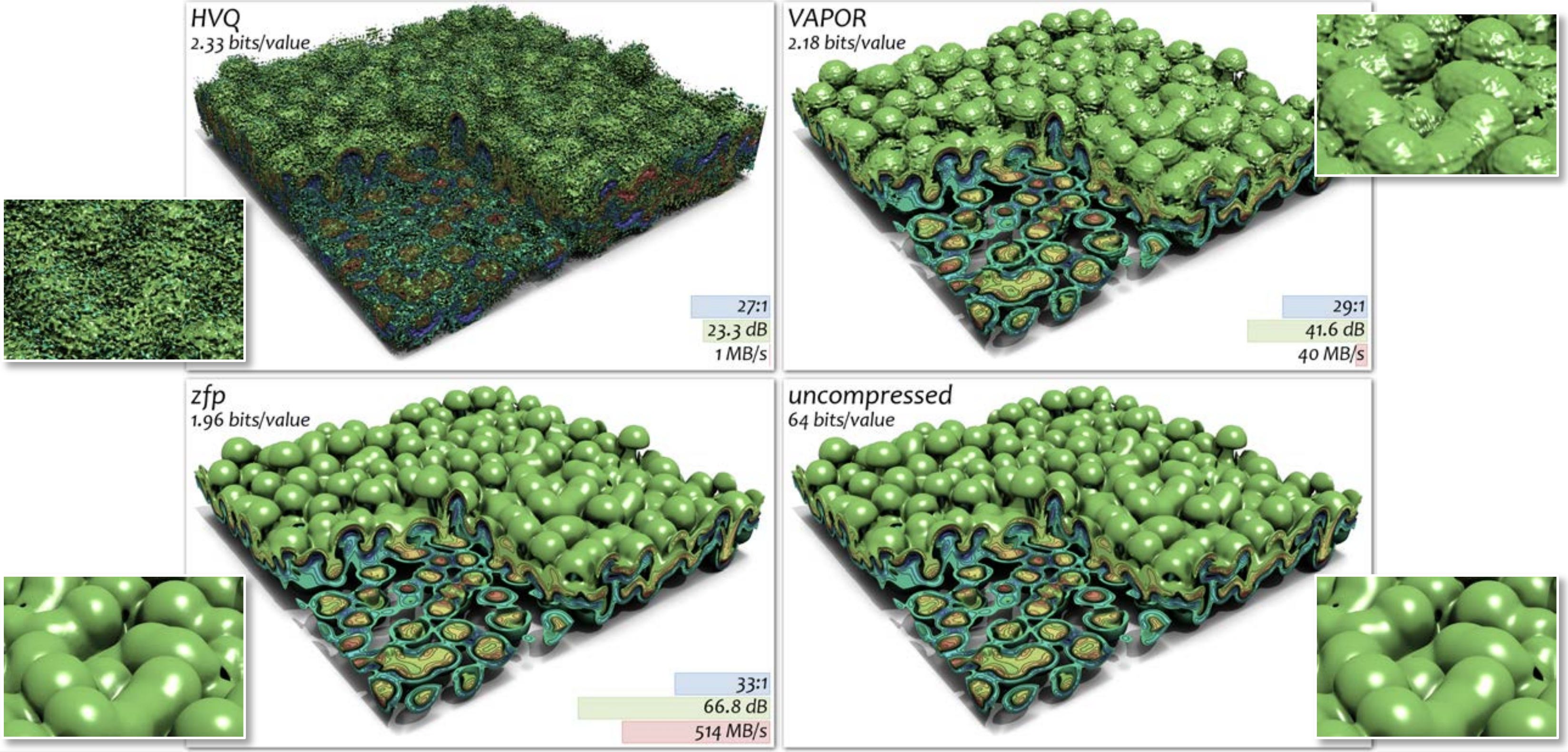
ZFP provides >100x compression with imperceptible loss in visual quality



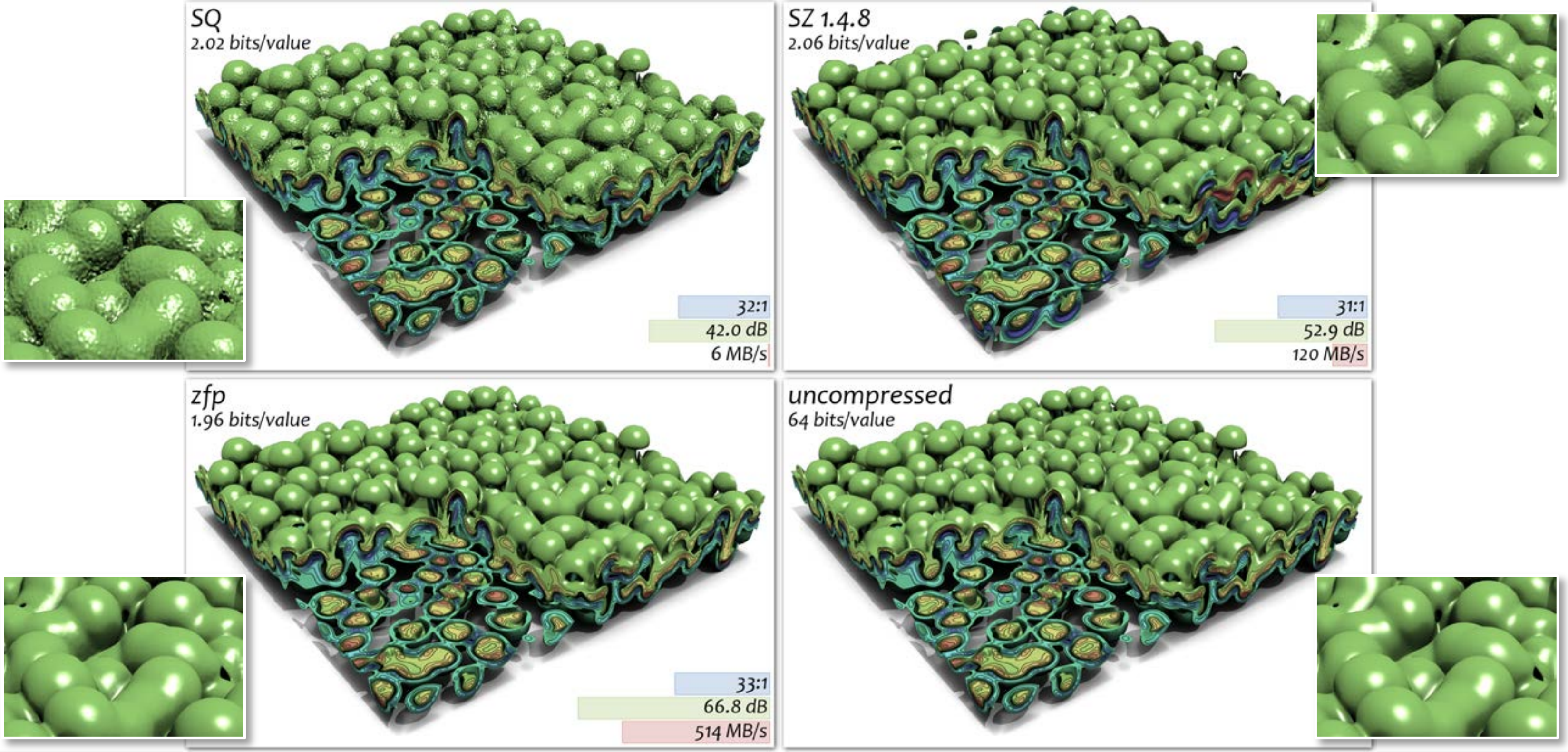
ZFP shows no artifacts in derivative computations (velocity divergence)



ZFP shows no artifacts in derivative computations (velocity divergence)



ZFP shows no artifacts in derivative computations (velocity divergence)



Parallel ZFP achieves up to 150 GB/s throughput