ZFP: compressed floating-point arrays

ECP Annual Meeting 2021

March 30, 2021

Peter Lindstrom
What is ZFP?

- **Answer #1**: A new, efficient **number format** for small vectors and tensors
  - Alternative to IEEE 754/SSE/AVX/tensor core registers, bfloats, posits, flexpoint, ...

- **Answer #2**: An implementation of **multidimensional arrays** with user-specifiable memory footprint or accuracy
  - Alternative to std::vector, Eigen/GSL/Kokkos/NumPy arrays, ...

- **Answer #3**: A fast, **streaming compressor** for large floating-point & integer arrays
  - Alternative to gzip, bzip2, blosc, fpzip, JPEG, ...
ZFP is a compressed number format for multi-dimensional floating-point arrays

- **ZFP compactly represents small vectors and tensors of real values**
  - Encodes $d$-dimensional block of $4^d$ values as variable-length bit string
  - **Fixed-length code** obtained via bit string truncation
    - Analogous to float approximation $1/5 = 0.001100110011... \approx 0.0011$
    - May incur round-off error
    - “Common” blocks have shorter codes $\Rightarrow$ less or no round-off error
  - **H/W friendly encoder**: integer additions and bitwise operations
  - **Replaces IEEE 754** as number format for numerical computations
    - Usually orders of magnitude **more accurate** than IEEE 754
ZFP multi-dimensional arrays offer in-memory compressed storage with high-speed read and write access

- ZFP provides C++ classes for multi-dimensional arrays
  - Read & write **random access** at block granularity
    - Block decomposition is transparent to user
  - User specifies **memory footprint** or **error tolerance**
  - **Conventional API:** C++ operator overloading hides complexity of (de)compression
    - `double a[n] ↔ std::vector<double> a(n) ↔ zfp::array<double> a(n, bits_per_value)`
    - C, experimental NumPy APIs are also available
**ZFP’s C++ compressed arrays can replace STL vectors and C arrays with minimal code changes**

// example using STL vectors

```cpp
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 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

```cpp
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 supports fast, parallel (de)compression of whole arrays

- ZFP also supports streaming compression for I/O, communication, storage
  - Supports absolute and relative error tolerances and lossless compression
  - Serial, OpenMP, CUDA, HIP, and FPGA implementations
    • Up to 160 GB/s parallel throughput
  - C, C++, Python, Fortran bindings
    • 3rd party Julia & Rust bindings available
  - I/O & viz support: ADIOS, Conduit, HDF5, Intel IPP, OpenZGY, Silo, TTK, VTK-m, …

- ZFP has other nice properties
  - Supports spatially adaptive compression
  - Supports progressive reconstruction (aka. SNR scalability)
  - Resilient to data corruption
ZFP GPU compression achieves up to 160 GB/s throughput

ZFP throughput on NVIDIA P100, AMD MI60

- CUDA compress
- CUDA decompress
- HIP compress
- HIP decompress

throughput (uncompressed GB/s)
rate (compressed bits/value)
ZFP improves accuracy in finite difference computations using less storage than IEEE 754 and POSITS.
Contrary to conventional floating-point, finite-difference accuracy using ZFP increases with grid resolution.

finite differences over 3D field

$\mathcal{O}(h^{\frac{5}{6}})$ compression error
$\mathcal{O}(h^5)$ truncation error
$\mathcal{O}(h^{-1})$ roundoff error

$L_2$ error in $u_x$ step size $h$

- 32-bit float
- 28-bit zfp

- $1E^{-15}$
- $1E^{-14}$
- $1E^{-13}$
- $1E^{-12}$
- $1E^{-11}$
- $1E^{-10}$
- $1E^{-9}$
- $1E^{-8}$
- $1E^{-7}$
- $1E^{-6}$
- $1E^{-5}$
- $1E^{-4}$
- $9.8E^{-4}$
- $2.0E^{-3}$
- $3.9E^{-3}$
- $7.8E^{-3}$
- $1.6E^{-2}$
- $3.1E^{-2}$
- $6.3E^{-2}$
- $1.3E^{-1}$

- 9.8E-04
- 2.0E-03
- 3.9E-03
- 7.8E-03
- 1.6E-02
- 3.1E-02
- 6.3E-02
- 1.3E-01
We have developed rigorous error bounds for ZFP, both for static data and in iterative methods.

Work by Alyson Fox and James Diffenderfer
ZFP variable-rate C++ arrays allocate bits where needed

<table>
<thead>
<tr>
<th>8 bits/value</th>
<th>16 bits/value</th>
<th>32 bits/value</th>
<th>64 bits/value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate = 8.1</td>
<td>rate = 12.5</td>
<td>rate = 16.0</td>
<td>rate = 21.8</td>
</tr>
<tr>
<td>prec = 40.0</td>
<td>prec = 40.0</td>
<td>prec = 40.0</td>
<td>prec = 40.0</td>
</tr>
</tbody>
</table>
ZFP adaptive arrays improve accuracy in PDE solution over IEEE by 6 orders of magnitude using less storage.
ZFP’s variable-rate arrays improve accuracy per bit stored and in some applications reduce time to solution

<table>
<thead>
<tr>
<th></th>
<th>fixed-rate read-write</th>
<th>variable-rate read-write</th>
<th>variable-rate read-only</th>
<th>uncompressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00 bits/value</td>
<td>3.01 hours</td>
<td>0.74 bits/value</td>
<td>2.31 hours</td>
<td>32.00 bits/value</td>
</tr>
</tbody>
</table>
Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.

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.