LA-UR-24-21705

Approved for public release; distribution is unlimited.

Title:	Modifying an Implicit Monte Carlo transport code for 16-bit precision
Author(s):	Long, Alex Roberts Pakin, Scott D.
Intended for:	Applied Computer Science Meeting, 2024-02-26/2024-03-01 (Albuquerque, New Mexico, United States)
Issued:	2024-02-23

Modifying an Implicit Monte Carlo transport code for 16-bit precision

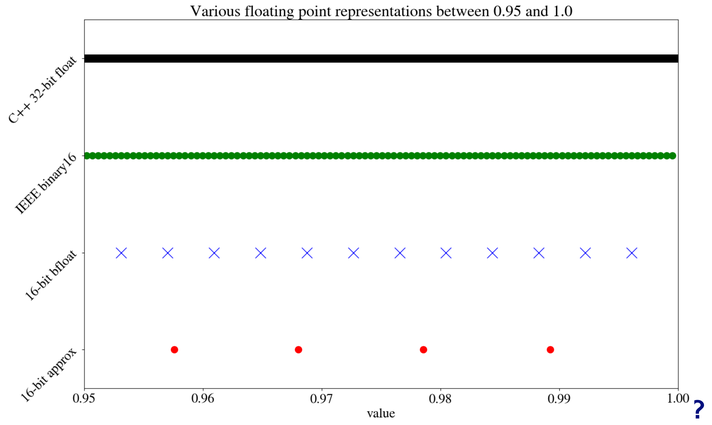
Alex Long (XCP-3), Scott Pakin (CCS-7)

The rise of small formats

GPUs for machine-learning and deep neural networks have increased hardware resources dedicated to lower-precision floating point formats. HPC codes at LANL predominantly use 64-bit floats (double) format. Potential benefits of reduced-precision for HPC codes include :

- More operations per second (faster simulations) (`float` ~2x `double`, `__half` ~2x `double`, `__half2` ~4x `double`)
- Reduced memory use (bigger simulations)
- More values can be stored in cache (faster simulations)
- Fewer registers used in GPU kernels (faster simulations)

Reduced-precision can require fundamental changes to an algorithm. For example, numbers very close to one often appear in Monte Carlo transport. The figure below plots representable



Two driving questions:

1) Can reduced precision produce comparable results to double precision in an Implicit Monte Carlo (IMC) code?
2) What is the speedup compared to double precision?

We will explore these questions using CUDA's 16-bit floats (half and bfloat) in the IMC proxy-app Branson

"Mixed Precision" vs. "Reduced Precision"

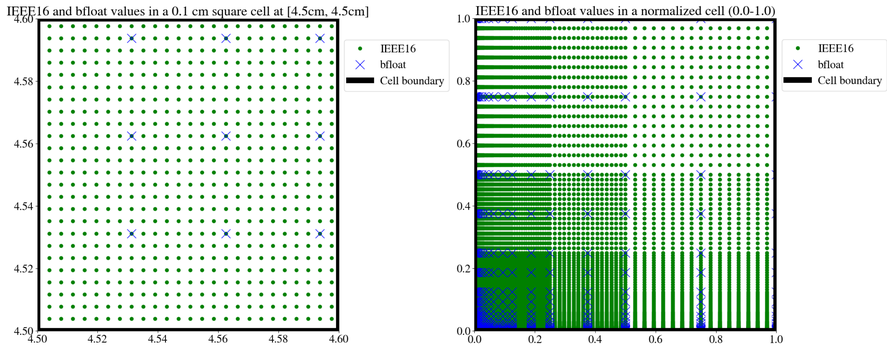
- Goal: target emerging platforms without hardware support for higher precision, i.e. no mixed-precision (e.g. Cerebras WSE-2 has no 64-bit double, Singular S1 has only 16-bit)

Changes for 16-bit floats

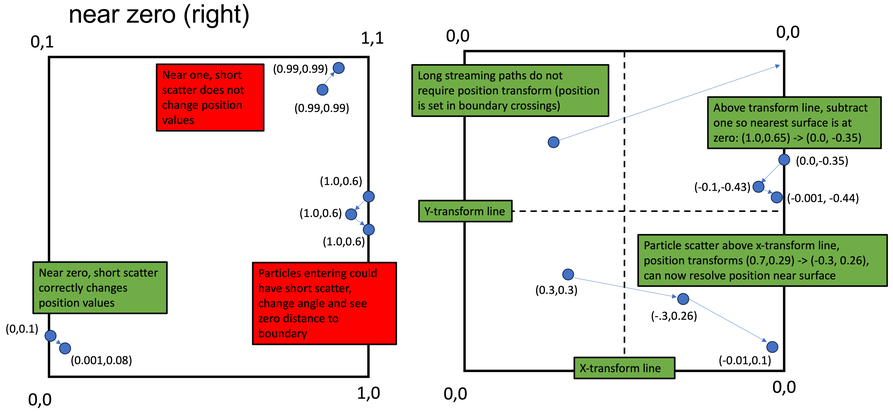
- Approach: rescale units and geometry to resolve important physics (scatters can be O(10^{-9} cm))
- Scale units to get them in range of 16-bit half type:
  1. Increase energy magnitude: jerk -> μjerk
  2. Increase length scale magnitude: cm -> μm (opacity is 1/cm, so it decreases in magnitude)
- Modify tracking routines to account for coincident events (scatters before face crossings, multiple surface crossings)
- Perform additional checks to keep values away from INF and NaN (small*(large*large)) -> (large*(small*large))
- Accumulate mean free paths traveled and deposit energy when leaving a cell (avoid accumulating small values)

Representing particle positions in mesh cells

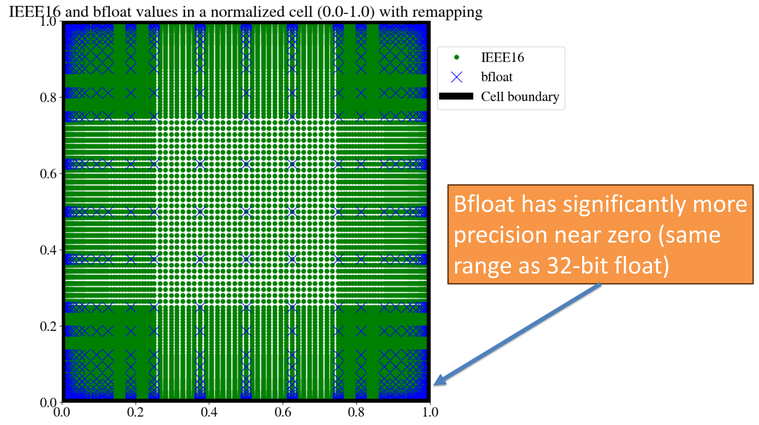
- Directly porting a traditional 64-bit mesh implementation limits representable positions away from zero, as seen in cell at 4.5 cm off axis (left figure)
- Remapping each spatial cell to local, relative coordinate system gives many more representable positions inside the cell (right figure, at 1/64 density for visualization)



- Even using relative positions, distances to scatter are not representable near 1.0
- To make use of increased precision near zero, particle positions are transformed such that short interactions happen near zero (right)



- Applying the position transform, the number of representable particle positions goes up by a factor of 3.3x and increases near surfaces (center figure, at 1/64 density for visualization)



Energy correction

In addition to particle tracking changes, an extra step is required in sourcing particles to correct for half precision representations of energy. In double, the energy of particle is:

$$E_{d,particle} = \frac{E_{d,cell}}{N_{particles}} \quad (1)$$

Where the "d" subscript indicates double precision. This value is converted to half precision, unfortunately that means:

$$\sum_{i=1}^{N_{particles}} E_{h,particle} \neq E_{d,cell} \quad (2)$$

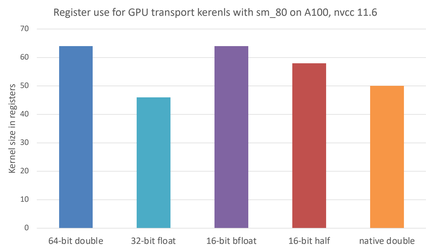
Particle energies must be corrected to better match the total cell energy

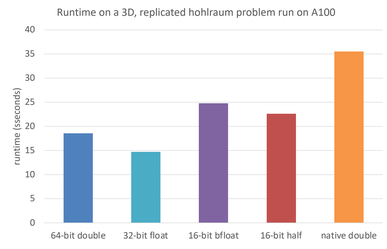
Results

Correctness

- Matched double precision to 1% on Marshak wave after 100 timesteps and 1 million particles per timestep
- Failed (for now) on full reduced-precision implementation—absorbed energy tally requires increased precision
- Implementing Kahan Summation on GPU is difficult as it requires two values to be atomically updated simultaneously or some kind of critical section

Performance





Discussion

Performance currently slower for 16-bit types, this work revealed performance is sensitive to atomic operations

Potential improvements to porting workflow:

- Native C++ support for various 16-bit floats (or at least __host__ versions of these functions)
- Special functions to help you determine when precision or range are important

Useful programming concepts for reduced precision:

- Custom classes to make _half look like a standard C++ type
- Templating to abstract type when possible
- C++ 17 with "if constexpr" for small specializations in templated functions

Future work

- Use the _half2 type that packs two 16-bit floats into a single 32-bit value—CUDA uses 32-bit registers for _half values so packing two particles into a _half2 could produce double performance assuming no branch divergence
- Use software approximations functions for 16-bit from previous LDRD (e.g. exp function accurate in (0,-16.0)

Los Alamos NATIONAL LABORATORY
NNSA National Nuclear Security Administration
Managed by Triad National Security, LLC, for the U.S. Department of Energy's NNSA.
ROSY ID: d920f099