Motivation

• Floating point arithmetic is non-associative

\[(a + b) + c \neq a + (b + c)\]

• Parallel reductions will yield different answers whenever we change otherwise seemingly innocuous parameters

• Is this a problem?
  • Won’t answers only be different to floating point epsilon?

• Lattice QCD calculations can exhibit chaotic behavior
  • Small changes can cause divergent HMC phase space traversal

• Without exact reproducibility we lose the ability to repeat experiments (simulations)
  • This is a failure of the scientific method
  • To fully specify a given physics ensemble, one would need to fully specify the order of all summation operations
  • Debugging…. 
Sources of Chaos

- Data ordering
  - Structure of Arrays vs Array of Structures vs SIMD ordering
- Architecture specifics
  - Floating point rounding employed
  - Flush denormals to zero?
- Multi-process decomposition
  - Number of processes
  - Process grid (e.g., 4x4x4x4 vs 2x4x4x8)
- Hierarchical many-core processors (GPUs)
  - Thread block size
  - Number of thread blocks
  - Work items per thread
- Stencil application (Dslash)
  - Local gather vs halo gather
QUDA

• “QCD on CUDA” - http://lattice.github.com/quda (open source, BSD license)
• Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma**, CPS**, MILC**, TIFR, etc.
• Provides solvers for major fermionic discretizations, pure gauge algorithms, etc.
• Maximize performance
  – Mixed-precision methods
  – Autotuning for high performance on all CUDA-capable architectures
  – Multigrid solvers for optimal convergence
  – NVSHMEM for improving strong scaling
• Portable: HIP (merged), SYCL (in review) and OpenMP (in development)
• A research tool for how to reach the exascale (and beyond)
  – Optimally mapping the problem to hierarchical processors and node topologies
Tree Reduction Algorithm

- Classic parallel reduction algorithm
  - For data set \( v[N] \), launch \( N/2 \) threads
  - Each thread performs pairwise reduction \( u[t] = \text{reduce}(v[t], v[t + N/2]) \)
  - Store result and repeat with half the number of threads
  - Complete reduction performed in \( \log(N) \) steps

- Modern optimized form
  - Fixed set of \( T \) threads rake over the data
  - Each thread accumulates \( N/T \) terms locally
  - Then perform tree summation between threads

- If we change \( T \) we will alter the order of summation...

https://github.com/NVIDIA/cub
• Run same solver 10 times with different GPU thread counts
• BiCGStab(l) with Wilson fermions
  • $V = 16^3 \times 64$
  • 2 MPI processes, 2 GPUs
  • Target relative residual $2 \times 10^{-16}$
• Double precision reductions
  • 9 unique convergence histories
• Residual is insensitive to low-mode errors
  • “Equivalent” solutions may have drastically different error components
  • Low modes “tickle” instabilities in the MD integration

Solver Chaos

![Graph showing convergence histories and iteration counts](image)
Can we fix it using higher precision?

- Double precision is not the limit

```c
struct float64_t {
    unsigned int mantissa : 52;
    unsigned int exponent : 11;
    unsigned int sign : 1;
};
```

IEEE binary64

- 64-bits per real
- 53-bit mantissa => Precision $\varepsilon \sim 1 \times 10^{-16}$
- 8-bit exponent => Range $\in [2 \times 10^{-208}, 2 \times 10^{308}]$

```c
struct float128_t {
    unsigned int mantissa : 113;
    unsigned int exponent : 15;
    unsigned int sign : 1;
};
```

IEEE binary128

- 128-bits per real
- 113-bit mantissa => Precision $\varepsilon \sim 2 \times 10^{-34}$
- 15-bit exponent => Range $\in [3 \times 10^{-4932}, 1 \times 10^{4932}]$

- Most modern processors do not support IEEE fp128.......

Double-double

- Use two doubles to emulate a quad
  - Effective 107 bits of precision (nearly as much as binary128)

- double-double addition operation costs 20 double precision additions
  - But flops are free and everything’s bandwidth?

```c
__device__ __host__ __forceinline__
dbldbl add_dbldbl (dbldbl a, dbldbl b) {
  dbldbl z;
  double t1, t2, t3, t4, t5, e;
  t1 = dadd_rn (a.y, b.y);
  t2 = dadd_rn (t1, -a.y);
  t3 = dadd_rn (dadd_rn (a.y, t2 - t1), dadd_rn (b.y, -t2));
  t4 = dadd_rn (a.x, b.x);
  t2 = dadd_rn (t4, -a.x);
  t5 = dadd_rn (dadd_rn (a.x, t2 - t4), dadd_rn (b.x, -t2));
  t3 = dadd_rn (t3, t4);
  t4 = dadd_rn (t1, t3);
  t3 = dadd_rn (t1 - t4, t3);
  t3 = dadd_rn (t3, t5);
  z.y = e = dadd_rn (t4, t3);
  z.x = dadd_rn (t4 - e, t3);
  return z;
}
```

Double-double addition function
Reworking QUDA’s Reductions...

- Old
  - Host types hard coded to double precision
  - Only naive tree summation algorithm implemented

- New abstraction
  - Reduction type
    - Defines the precision of any sum reductions
  - Separate type for host scalar type
    - e.g., CG’s alpha, beta coefficients
  - Parallel summation algorithm
    - e.g., naive, Kahan, reproducible
  - All configured by CMake build system
Double-double Reductions

QUDA Implementation

Asymptotic 3% overhead

Performance on Norm2 reduction kernel, fp64 inputs
(Quadro GV100, CUDA 12.1)

Relative Deviation between CPU and GPU Norm2 reductions
Solver Chaos

- Run same solver 10 times with different GPU thread counts
- BiCGStab(l) with Wilson fermions
  - \( V = 16^3 \times 64 \)
  - 2 MPI processes, 2 GPUs
  - Target relative residual \( 2 \times 10^{-16} \)
- Double precision reductions
  - 9 unique convergence histories
- Double-double reductions
  - 6 unique convergence histories
Reproducible Summation

Ahrens et al, 2020

- Reproducible Summation (aka K-fold summation)
  - Bin the components of each number into bins of predefined exponent range
  - Each binned component is known as a “slice”
- We can sum the slices in the same bin exactly, so long as we don’t overflow
  - Abusing floating point to behave as integer (integer is associative)
  - Given a bin width of \( W \) bits, and precision \( P \) bits, we can sum \( 2^{P-W} \) slices exactly
    - E.g., FP32 (single precision)
      - \( P = 24, W = 13 \)
      - We can add \( 2^{11} \) slices together exactly
Reproducible Summation
Ahrens et al, 2020

- When summing slices, each summation represented by two bins:
  - Primary: where each slice’s value is summed to
  - Carry: store any overflow bits from summation to primary

- Algorithm is exact if we fully cover the range of the underlying floating-point representation
  - Not feasible for double precision (way too many bins required)

- Only retain a fixed number of bins $K$ (typically $K = 3$)
  - So each real number requires $K \times (\text{primary} + \text{carry})$ values

- Maximum bin based on set maximum value
  - Avoid pre-computing the set maximum by tracking maximum value to date
  - If new maximum encountered shift bins and drop least significant bins as needed

- Once summation of slices is complete, reconstruct the final floating point value

- Absolute error bound: $E \leq 2^{-(K-1)W} N \max x_i$
  - FP64: $E \leq 2^{-27} \varepsilon N \max x_i$ ($K = 3, W = 40$)
  - Compared to standard summation $E \leq (N - 1) \varepsilon \sum_i \|x_i\|$
Reproducible Summation

- Compare summation of same set under a random permutation
- Compare to exactly computed reference

Positive Uniform Random (N = $10^7$, 1000 permutations)

<table>
<thead>
<tr>
<th>Method</th>
<th>Maximum Relative Error</th>
<th>Maximum Absolute Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reproducible (double)</td>
<td>$3.1 \times 10^{-17}$</td>
<td>$3.7 \times 10^{-14}$</td>
</tr>
<tr>
<td>Kahan (double)</td>
<td>$4.4 \times 10^{-17}$</td>
<td>$1.2 \times 10^{-13}$</td>
</tr>
<tr>
<td>Parallel summation (double)</td>
<td>$1.4 \times 10^{-15}$</td>
<td>$4.3 \times 10^{-12}$</td>
</tr>
<tr>
<td>Reproducible (float)</td>
<td>$9.7 \times 10^{-9}$</td>
<td>$2.0 \times 10^{-5}$</td>
</tr>
<tr>
<td>Kahan (float)</td>
<td>$2.9 \times 10^{-8}$</td>
<td>$7.3 \times 10^{-4}$</td>
</tr>
<tr>
<td>Parallel summation (float)</td>
<td>$5.6 \times 10^{-7}$</td>
<td>$1.9 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

Sine Wave (N = $10^7$, 1000 permutations)
Reproducible Summation on GPUs
Algorithm as presented not efficient for parallel architectures

- Thread bin indices computed dynamically based on present summand value
  - Cannot dynamically index registers leading to spilling of bins into cache hampering performance
  - Solution: use switch table instead of direct array indexing

- Each thread’s local maximum may differ dramatically
  - Bin shifting overhead when reduction between threads is performed
  - Solution: when thread maximum is reset, reset for entire warp

- Each thread may load numbers of very different magnitude leading to different bin indices
  - E.g., index differs between threads in a warp
  - Not a problem on SIMT, perhaps a problem on SIMD?
Reproducible Summations on GPUs

QUDA Implementation

Performance on Norm2 reduction kernel, fp64 inputs

(Quadro GV100, CUDA 12.1)

Asymptotic 3% overhead

Relative Deviation between CPU and GPU Norm2 reductions

Reproducible reductions are bitwise identical as expected
Reproducible Summations on GPUs
QUDA Implementation

Aside: H100 can pull over 3 TB/s

Performance on Norm2 reduction kernel, fp64 inputs
(Quadro GV100, CUDA 12.1)

Relative Deviation between CPU and GPU Norm2 reductions
Reproducible reductions are bitwise identical as expected
• Run same solver 10 times with different GPU thread counts

• BiCGStab(l) with Wilson fermions
  • $V = 16^3 \times 64$
  • 2 MPI processes, 2 GPUs
  • Target relative residual $2 \times 10^{-16}$

• Double precision reductions
  • 9 unique convergence histories

• Double-double reductions
  • 6 unique convergence histories

• Reproducible reductions
  • 1 unique convergence history
Future Work

- Add support for reproducible 128-bit summation

- Optimize
  - Partial reduction memory writing (will improve performance for intermediate sizes)
  - MPI Allreduce (presently implemented using all gather and local sum)

- Reproducible algorithm abuses floating point to behave as integers
  - Why not just use actual integers?
  - Integers have a number of advantages
    - No wasted bits for storing the exponent
    - Fewer resources required for same precision (less registers, less memory traffic)
    - NVIDIA GPUs have hardware-accelerated warp-wide integer reductions (Ampere onwards)
Summary

• Lack of floating point associativity leads to lack of reproducibility for parallel computations

• Evolved QUDA’s reduction framework to allow for arbitrary reduction types and arbitrary summation algorithm

• 128-bit floating-point precision not sufficient to ensure reproducibility

• Deployed optimized reproducible reduction algorithm for bit-wise reproducible results

• Reproducibility doesn’t need to cost the earth

• Restoration of the Scientific Method

https://github.com/lattice/quda/tree/feature/reproducible