Lecture plan

- Last time: GPU review, Cloud computing
- This time: Using the CRC, Floating Point

Why Floating Point?

- And might care about using single precision for speed
- And might wonder when your FP code starts to crawl
- And may want to run code on a current GPU
- And may care about mysterious hangs in parallel code
- And may wonder about reproducible results in parallel

A little bit of history (of course)

- Von Neumann and Goldstine, 1947: can't solve linear systems accurately for $n > 15$ without carrying many digits ($n > 8$).
- Turing, 1949: carrying n digits is equivalent to changing input data in the nth place (backward error analysis)
- Wilkinson, 1961: rediscovered and publicized backward error analysis (1970 Turing Award)
- Backward error analysis of standard algorithms from 1960s
- But varying arithmetics made portable numerical software hard!
- IEEE 754/854 floating point standards (published 1985; Turing award for W. Kahan in 1989)
- Revised IEEE 754 standard in 2008

IEEE floating point

Normalized numbers: $(-1)^s \times (1.b_1 b_2 \ldots b_p)_2 \times 2^e$ Have 32-bit single, 64-bit double numbers consisting of

- Sign s
- Precision p (p $=$ 23 or 52)
- Exponent $e(-126 \leq e \leq 126 or -1022 \leq e \leq 1023)$

Questions:

- What if we can't represent an exact result?
- What about $2^{e_{max}+1} \leq x << \leq$ or $0 \leq x < 2^{e_{min}}$?
- What if we compute $1/0$?
- What if we compute $\sqrt{-1}$?

Rounding

Basic ops $(+, -, \times, /, \sqrt{\ })$, require correct rounding

- Policy: As if computed to infinite precision, then rounded. Don't actually need infinite precision for this!
- Different rounding rules possible:
  - Round to nearest even (default)
  - Round up, down, toward 0: error bounds and intervals
- If rounded result $\neq$ exact result, have *inexact exception*
- 754-2008 *recommends* (does not require) correct rounding for a few transcendentals as well (sine, cosine, etc)

Denormalization and underflow

Denormalized numbers: $(-1)^s \times (0.b_1 b_2 \ldots b_p)_2 \times 2^{e_{min}}$

- Evenly fill in space between $\pm 2^{e_{min}}$
- Gradually lose bits of precision as we approach zero
- Denormalization results in an underflow exception
- Except when an exact zero is generated

Infinity and NaN

Other things can happen:

- $2^{e_{max}} + 2^{e_{max}}$ generates ∞ (overflow exception)
- $1/0$ generates ∞ (divide by zero exception) ... should really be called "exact infinity" exception
- $\sqrt{-1}$ generates Not-a-Number (invalid exception)

But every basic operation produces something well defined.

Basic rounding model

Model of roundoff in a basic op:
$fl(a \cdot b) = (a \cdot b)(1 + \delta), |\delta| \leq \epsilon_{\text{machine}}$

- Too optimistic: misses overflow, underflow, or divide by zero
- Also too pessimistic: some things are done exactly!
- Example: $2x$ exact, as is $x + y$ if $x/2 \leq y \leq 2x$

This model is not complete but useful as a basis for backward error analysis

Example: Horner's rule (polynomial evaluation)

Evaluate $p(x) = \sum_{k=0}^{n} c_k x^k$:

```
p = c(n)
for k = n-1 downto 0
p = x*p + c(k)
```

Can show backward error result: $p(x) = \sum_{k=0}^{n} \hat{c}_k x^k$: where $|\hat{c}_k \leq c_k| \leq (n+1)\epsilon_{\text{machine}}|c_k|$

The Modern Era: IEEE 754

- Almost everyone implements IEEE 754 (at least since 1985)
- Old Cray / Vax arithmetic is essentially extinct
  - VAX had 4: F (32 bit single precision), D (64 bit double precision), G Floating (64 bit double precision, wide exponent), H (128 bit quad precision)
  - Cray-1: 49 bit signed magnitude mantissa, 15 bit biased exponent
- Backward error analysis in Numerical Analysis
- Good libraries for linear algebra, elementary functions

IEEE 754 Floating point standard

- arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- interchange formats: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form
- rounding rules: properties to be satisfied when rounding numbers during arithmetic and conversions
- operations: arithmetic and other operations (such as trigonometric functions) on arithmetic formats
- exception handling: indications of exceptional conditions (such as division by zero, overflow, etc.)

Single Precision

- If $e = 255$ and $f \neq 0$, then v is NaN regardless of s
- If $e = 255$ and $f == 0$, then $v = (-1)^s \infty$
- If $0 < e < 255$, then $v = (-1)^s 2^{e-127}(1.f)$
  Normalized number
- If $e == 0$ and $f \neq 0$, the $v = (-1)^s 2^{-126}(0.f)$
  Denormalized numbers
- If $e == 0$ and $f == 0$ the $v = (-1)^s 0$
  Zero

Double Precision

- If $e = 2047$ and $f \neq 0$, then v is NaN regardless of s
- If $e = 2047$ and $f = 0$, then $v = (-1)^s \infty$
- If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023}$ (1.f)
  Normalized number
- If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-1022}$ (0.f)
  Denormalized numbers
- If $e = 0$ and $f = 0$ then $v = (-1)^s 0$
  Zero

Notes on single and double precision

- The leading 1 of the fractional part is not stored for normalized numbers (*hidden bit*)
- Representation allows for $+0$ and -0 indicating direction of 0 (allow determination that might matter if rounding was used)
- Denormalized numbers allow graceful underflow towards 0

But...

- Almost everyone implements IEEE 754 (at least since 1985)
  - But GPUs may lack gradual underflow, do sloppy division
  - And it's impossible to write portable exception handlers
  - And even with C99, exception flags may be inaccessible
  - And some features might be slow
  - And the compiler might not do what you expected
- Good libraries for linear algebra, elementary functions
  - But people will still write their own (!)

Arithmetic speed

- Single precision is faster than double precision
- Actual arithmetic cost may be comparable (on CPU)
- But GPUs generally prefer single precision
- And SSE instructions do more per cycle with single precision
- And memory bandwidth is lower

Mixed-precision arithmetic

- Idea: use double precision only where needed
- Example: iterative refinement and relatives
- Or use double-precision arithmetic between single-precision representations (may be a good idea regardless)

Example: Mixed-precision iterative refinement

Factor $A = LU$	$O(n^3)$ single-precision work
Solve $x = U^{-1}(L^{-1}b)$	$O(n^2)$ single-precision work
$r = b - Ax$	$O(n^2)$ double-precision work

While $||r||$ too large:

$d = U^{-1}(L^{-1}r)$	$O(n^2)$ single-precision work
$x = x + d$	$O(n)$ single-precision work
$r = b - Ax$	$O(n^2)$ double-precision work

Single or double?

What to use for:

- Large data sets? (single for performance, if possible)
- Local calculations? (double by default, except maybe on GPU)
- Physically measured inputs? (probably single)
- Nodal coordinates? (probably single)
- Stiffness matrices? (maybe single, maybe double)
- Residual computations? (probably double)
- Checking geometric predicates? (double or more)

Simulating extra precision

What if we want higher precision than is fast?

- Double precision on a GPU?
- Quad precision on a CPU?

Can simulate extra precision. Example:

```
if abs(a) < abs(b), swap a and b
double s1 = a+b; /* May suffer roundoff */
double s2 = (a-s1) + b; /* No roundoff!  */
```

Idea applies more broadly:

- Used in fast extra-precision packages
- And in robust geometric predicate code
- And in XBLAS

Exceptional arithmetic speed

Time to sum 1000 doubles on a laptop:

- Initialized to 1: 1.3 microseconds
- Initialized to Inf/NaN: 1.3 microseconds
- Initialized to 10^{-312}: 67 microseconds

50× performance penalty for gradual underflow!
Why worry? Some GPUs don't support gradual underflow at all!
One reason:

```
if (x != y)
z = x/(x-y);
```

Also limits range of simulated extra precision

Exceptional algorithms, again

A general idea (works outside numerics, too):

- Try something fast but risky
- If something breaks, retry more carefully

If risky usually works and doesn't cost too much extra, this improves performance

Problems in parallel programs: Repeatability

Floating point addition is not associative:
$fl(a + fl(b + c)) \neq fl(fl(a + b) + c)$
So answers depends on the inputs, but also

- How blocking is done in multiply or other kernels
- Maybe compiler optimizations
- Order in which reductions are computed
- Order in which critical sections are reached

Worst case: with nontrivial probability we get an answer too bad to be useful, not bad enough for the program to die and garbage comes out.

Problems in parallel programs: Repeatability

What to do?

- Apply error analysis agnostic to ordering
- Write a slower version with specific ordering for debugging

Problems in parallel programs: Heterogeneity

Local arithmetic faster than communication

- So be redundant about some computation
- What if the redundant computations are on different machines?
- Different nodes in the cloud?
- GPU and CPU?
- Problem case: different exception handling on different nodes
- Problem case: take different branches due to different rounding

Summary

So why care about the vagaries of floating point?

- Might actually care about error analysis
- Or using single precision for speed
- Or maybe just reproducibility
- Or avoiding crashes from inconsistent decisions!