CS 5220

Floating Point

David Bindel

2024-11-19

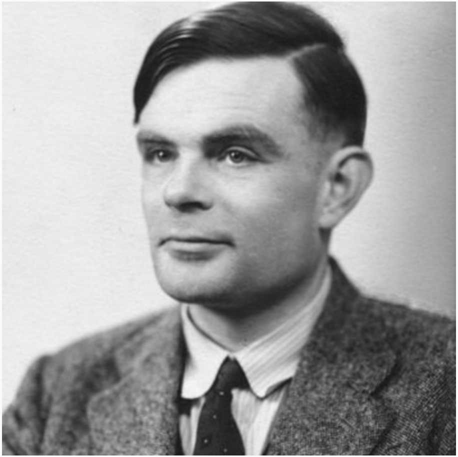
Von Neumann and Goldstine

Von Neumann and Goldstine

"Numerical Inverting of Matrices of High Order" (1947)

> ... *matrices of the orders 15, 50, 150 can usually be inverted with a (relative) precision of 8, 10, 12 decimal digits less, respectively, than the number of digits carried throughout.*

Turing



Turing

"Rounding-Off Errors in Matrix Processes" (1948)

Carrying d digits is equivalent to changing input data in the dth place (backward error analysis).

Wilkinson

Wilkinson

"Error Analysis of Direct Methods of Matrix Inversion" (1961)

Modern error analysis of Gaussian elimination

> *For his research in numerical analysis to facilitiate the use of the high-speed digital computer, having received special recognition for his work in computations in linear algebra and "backward" error analysis. — 1970 Turing Award citation*

Kahan

Kahan

IEEE-754/854 (1985, revised 2008, 2018)

> *For his fundamental contributions to numerical analysis. One of the foremost experts on floating-point computations. Kahan has dedicated himself to "making the world safe for numerical computations." — 1989 Turing Award citation*

IEEE floating point reminder

Normalized numbers:

$$(-1)^s \times (1.b_1 b_2 \dots b_p)_2 \times 2^e$$

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 $-1022 \leq e \leq 1023$)

Newer 16-bit formats: fp16 ($p = 10$); bfloat16 ($p = 7$)

Beyond normalized

- What if we can't represent an exact result?
- What about $2^{e_{\max}+1} \leq x < \infty$ 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*

- 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, to 0 – error bds + intervals
- 754 *recommends* (does not require) correct rounding for a few transcendentals as well (sine, cosine, etc).

Inexact

- If rounded result $\neq$ exact result, have *inexact exception*
    - Which most people seem not to know about…
    - … and which most of us who do usually ignore

Denormalization and underflow

Denormalized numbers:

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

- 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

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"
- $\sqrt{-1}$ generates Not-a-Number (*invalid exception*)

But every basic op produces *something* well defined.

Basic rounding model

Model of roundoff in a basic op:

$$\mathrm{fl}(a \odot b) = (a \odot b)(1 + \delta), \quad |\delta| \leq \epsilon.$$

- This model is *not* complete
    - Misses overflow, underflow, divide by zero
    - Also, some things are done exactly!
    - Example: $2x$ exact, as is $x + y$ if $x/2 \leq y \leq 2x$
- But useful as a basis for backward error analysis

Example: Horner's rule

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)
```

Example: Horner's rule

Can show backward error result:

$$\mathrm{fl}(p) = \sum_{k=0}^{n} \hat{c}_k x^k$$

where $|\hat{c}_k - c_k| \leq (n+1)\epsilon|c_k|$.

Backward error + sensitivity gives forward error. Can even compute running error estimates!

Hooray for the modern era!

- Everyone almost implements IEEE 754
    - Old Cray arithmetic is essentially extinct
- We teach backward error analysis in basic classes
- Good libraries for LA, elementary functions

Back to the future?

- But GPUs have funky (low-precision) formats!
- Hard to write portable exception handlers
- Exception flags may be inaccessible
- Some features might be slow
- Compiler might not do what you expected

Back to the future?

- We teach backward error analysis in basic classes
  - … which are often no longer required!
  - And anyhow, bwd error isn't everything.
- Good libraries for LA, elementary functions
  - But people will still roll their own.

Arithmetic speed

Single faster than double precision

- Actual arithmetic cost may be comparable (on CPU)
- But GPUs generally prefer single (or lower)
- And AVX instructions do more per cycle with single
- And memory bandwidth is lower

NB: FP16 originally intended for storage only!

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

Example: Helpful extra precision

```
/*
 * Assuming all coordinates are in [1,2), check on which
 * side of the line through A and B is the point C.
 */
int check_side(float ax, float ay, float bx, float by,
               float cx, float cy)
{
    double abx = bx-ax, aby = by-ay;
    double acx = cx-ax, acy = cy-ay;
    double det = acx*aby-abx*aby;
    if (det == 0) return  0;
    if (det <  0) return -1;
    if (det >  0) return  1;
}
```


This is not robust if the inputs are double precision!

Single or double?

What to use for:

- Large data sets? (single for performance, if possible)
- Local calculations? (double by default, except 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?

Simulating extra precision

Can simulate extra precision. Example:

```
// s1, s2 = two_sum(a, b) -- Dekker's version
if abs(a) < abs(b) { swap(&a, &b); }
double s1 = a+b;          /* May suffer roundoff */
double s2 = (a-s1) + b;   /* No roundoff! */
```

Simulating extra precision

Idea applies more broadly (Bailey, Bohlender, Dekker, Demmel, Hida, Kahan, Li, Linnainmaa, Priest, Shewchuk, ...)

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

Exceptional arithmetic speed

Time to sum 1000 doubles on my laptop:

- Initialized to 1: 1.3 microseconds
- Initialized to inf/nan: 1.3 microseconds
- Initialized to 10^{-312}: 67 microseconds

$50\times$ performance penalty for gradual underflow!

Exceptional arithmetic

Why worry? One reason:

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

Also limits range of simulated extra precision.

Exceptional algorithms, take 2

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.

(See Demmel and Li; Hull, Farfrieve, and Tang.)

Three problems

What goes wrong with floating point in parallel (or just high performance) environments?

Problem 0: Mis-attributed Blame

To blame is human. To fix is to engineer. — Unknown

Three variants:

- "Probably no worries about floating point error."
- "This is probably due to floating point error."
- "Floating point error makes this untrustworthy."

Problem 1: Repeatability

Floating point addition is *not* associative:

$$\mathrm{fl}(a + \mathrm{fl}(b + c)) \neq \mathrm{fl}(\mathrm{fl}(a + b) + c)$$

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

Problem 1: Repeatability

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

Problem 1: Repeatability

What can we do?

- Apply error analysis agnostic to ordering
- Write slower debug version with specific ordering
- Soon(?): Call the *reproducible BLAS*

Problem 2: Heterogeneity

- Local arithmetic faster than communication
- So be redundant about some computation
- What if redundant computations use different HW?
    - Different nodes in the cloud?
    - GPU and CPU?
- Problems
    - Different exception handling on different nodes
    - Different branches due to different rounding

What can we do?

- Avoid FP-dependent branches
- Communicate FP results affecting branches
- Use reproducible kernels

Claim: DNNs robust to low precision!

- Overflow an issue (hence bfloat16)
- Same pressure has revived block FP?
- More experiments than analysis

Recap

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!

References

- “What Every Computer Scientist Should Know About Floating Point Arithmetic” (David Goldberg + addendum by Doug Priest)
- “Revisiting ‘What Every Computer Scientist Should Know About Floating Point Arithmetic’ ” (Lafage)
- *Numerical Computing with IEEE Floating Point Arithmetic* (Overton)
- *Handbook of Floating Point Arithmetic* (Muller et al)
- *Accuracy and Stability of Numerical Algorithms* (Higham)