Lecture 20:
The impact of floating point

David Bindel

10 Apr 2011

Logistics

- SPH
  - You have until midnight Friday
  - Prioritize understanding performance to code fiddling
  - Though performance of broken code is not so interesting
- Topic modeling
  - You have until May 2 – finish early!
  - Just upgraded Julia on C4.
- Final project also on the horizon!

Lessons from SPH

- Smart algorithms and tuning trump tuning alone
  - Binning does far more for performance than parallelism.
- There is no point in tuning non-bottlenecks
  - The leapfrog updates take effectively no time
  - Re-binning is also negligible compared to interactions

Lessons from SPH

- OpenMP is not magic
  - Starting/stopping thread times is expensive
  - Synchronization is expensive
  - Easy to slow things down by parallelizing!
  - Helps to be thoroughly aware of dependency structure
- Race conditions are sometimes soul-suckingly subtle
  - May not show up except by accidents of timing
  - Very difficult to detect (note: `valgrind` DRD tool may help)

Lessons from SPH

- Test and debug incrementally
  - The code and pray method does not scale
  - Automate the process of sanity checking your computations!
- Pointer chasing (and index chasing) are performance drags
  - Pays to have data locality and simple, regular access patterns
  - Can get significant wins from paying attention to data used in inner loops
- `calloc` and `free` are not free

Performance koans

- Tuning often starts not with code, but with data structures.
- You cannot judge performance without a model.
- You cannot optimize what you cannot measure.
- Eventually, a better algorithm will always beat a tuning trick.
- Your time is worth more than the computer's time.
- The fastest code to write may be someone else's library.

Enough with the warm-up act.
Floating point!

Why this lecture?

Isn't this really a lecture for the start of CS 3220?

- Except you might have forgotten some things
- 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

Some history

- Von Neumann and Goldstine, 1947 – can't solve linear systems accurately for $n > 15$ without carrying many digits ($n > 8$).
- Turing, 1949 – carrying d digits is equivalent to changing input data in the dth place (backward error analysis)
- Wilkinson, 1961 – rediscovered + publicized backward error analysis (1970 Turing Award)
- Backward error analysis of standard algorithms from 1960s on
- But varying arithmetics made portable numerical SW 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 reminder

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 $2^{e_{\max}+1} \leq x < \infty$ 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, toward 0 – error bounds and intervals
- 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
- 754-2008 *recommends* (does not require) correct rounding for a few transcendentals as well (sine, cosine, etc).

Denormalization and underflow

Denormalized numbers:

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

- Evenly fill in space between $\pm 2^{e_{\text{min}}}$
- 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:

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

- This model is *not* complete
  - 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 $x/2 \leq y \leq 2x$
- 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)
```

Can show backward error result:

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

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

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

Hooray for the modern era!

- Almost everyone implements IEEE 754 (at least 1985)
  - Old Cray arithmetic is essentially extinct
- We teach backward error analysis in basic classes
- We have good libraries for linear algebra, elementary functions

Back to the future?

- Almost everyone implements IEEE 754 (at least 1985)
  - Old Cray arithmetic is essentially extinct
  - But GPUs may lack gradual underflow
  - 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
- We teach backward error analysis in basic classes
  - ... which are often no longer required!
  - And anyhow, backward error analysis isn't everything.
- We have good libraries for linear algebra, elementary functions
  - But people will still roll their own.

Arithmetic speed

Single precision is faster than double precision

- Actual arithmetic cost may be comparable (on CPU)
- But GPUs generally prefer single
- And SSE instructions do more per cycle with single
- 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	
$\quad d = U^{-1}(L^{-1}r)$	$O(n^2)$ single-precision work
$\quad x = x + d$	$O(n)$ single-precision work
$\quad r = b - Ax$	$O(n^2)$ 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 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 (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× 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, 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, and also Hull, Farfrieve, and Tang.)

Parallel problems

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

Problem 1: Repeatability

Floating point addition is *not* associative:

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

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 barf — and garbage comes out.

Problem 1: Repeatability

What can we do?

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

Problem 2: Heterogeneity

- Local arithmetic faster than communication
- So be redundant about some computation
- What if the redundant computations are on different HW?
  - 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

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!

Start with "What Every Computer Scientist Should Know About Floating Point Arithmetic" (David Goldberg, with an addendum by Doug Priest). It's in the back of Patterson-Hennessey.