Efficient floating point: software and hardware

David Bindel Rajit Manohar

CS and ECE
Cornell University

17 Oct 2011

Why this talk?

Isn't this a lecture for an intro NA class?

- **Yes:**
  - IEEE 754-1985 is ubiquitous
  - IEEE 754R finished in 2008
  - Basic features and analysis are widely taught
- **No:**
  - Some aspects remain widely misunderstood.
  - Poor support for some features
    - In languages and compilers
    - In hardware

IEEE floating point representations

- 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 $-1022 \leq e \leq 1023$)
- Denormalized numbers (including ± 0):

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

- Representations for $\pm\infty$ and NaN.

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).

Basic rounding model

Model of roundoff in a basic op:

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

- 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

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 $\pm 2^{e_{\min}}$
- Required to guarantee $x - y = 0 \implies x = y$
- Gradually lose bits of precision as we approach zero
  - Violates basic roundoff model!
- 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.

Desiderata

Want *fast*, *accurate*, and *predictable* computation:

- Different precisions
  - Single and double for trading time/memory vs accuracy
  - `long double` for a little extra precision that still runs fast
  - Exactly represent single $\times$ single in double (+ a little)
- Preserving identities
  - $\mathrm{fl}(x - y) = 0 \iff x = y$ (gradual underflow)
  - $\mathrm{fl}(x - y)$ exact if x, y within $2\times$ (correct rounding)
- Exception handling
  - Run full speed without special checks for corner cases
  - Detect when more care was needed and recompute
  - Handle commonly understood cases (e.g. $1/\infty = 0$)
  - Add idioms for other exceptions (e.g. $0/0 \to a$)
- Rounding modes
  - Support error bounds, interval arithmetic

The best of all possible worlds?

- Different precisions
  - No consistency in choice of intermediate precisions:

    ```
    float a = ..., b = ..., c = ...
    float result = a*b + c;
    ```

    What precision is `a*b` here? What about FMA?
  - Choice of FP unit changes semantics (SSE vs x87)
  - `long double` is increasingly rare
- Preserving identities
  - Flush-to-zero destroys $\mathrm{fl}(x - y) = 0 \iff x = y$.
  - Optimizing compilers destroy others!
- Exception handling
  - Flags are inaccessible in many languages (optional in C99)
  - Custom trap handlers are *incredibly* painful to write
    - And may require actually slowing down the common case
    - In practice, only used for try/catch or abort
- Inconsistent access to rounding mode control

Mixed precision

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

Idea: Mix precisions to get both speed and accuracy

- Example: iterative refinement and relatives
- High *intermediate* precision often great for accuracy

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 coordinates 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!

What precision?

What to use for:

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

Preserving identities

What if we want higher precision than is fast in HW? Can simulate extra precision. Example:

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

Second line relies on fl($x - y$) exact if x,y within $2\times$.

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

But: Fails if optimizer doesn't maintain FP semantics!

Preserving identities

Flush-to-zero underflow breaks this code

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

Also limits range of simulated extra precision.

Most CPUs support gradual underflow, but...
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!
And some GPUs don't support gradual underflow at all!

Exceptional algorithms

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.)

The trouble with traps

One implementation idea: lightweight trap handlers

```
on nan trap, return 0 in
  for (int i = 0; i < n; ++i)
    y[i] = x[i]*log(x[i]);
```

I implemented this (as did Doug Priest at Sun):

- Trap handler decoded instruction
- Dynamically rewrote code to implement alternate action
- Required extra `fwait` instructions to compiler output
- Huge penalty to save / restore context

Slowed down the common case to the point of uselessness!

Exceptional interface

Proposal during IEEE 754 revision: exception handling menu

- Check flags
- `try/catch`-style handling
- Scaled exponent types (automatically handle over/underflow as needed)
- Presubstitution on potential NaN expressions
- Traps become a potential optimization, not a requirement

This did not go through, though much of the related discussion is archived; see, for example:

`http://grouper.ieee.org/groups/754/meeting-minutes/02-08-22.html`

Handoff to Rajit

- IEEE 754 is broadly supported, but
  - Many features are slow, hard to access
  - And perception as a hardware spec (vs. something supported in language) has limited usefulness of other features
- Maybe next gen hardware (and compilers) will help?