CS 267 Applications of Parallel Computers

Lecture 13:

Floating Point Arithmetic

James Demmel

http://www.cs.berkeley.edu/~demmel/cs267_Spr99

Outline

° A little history

° IEEE floating point formats

° Error analysis

° Exception handling

• Using exception handling to go faster

° How to get extra precision cheaply

° Cray arithmetic - a pathological example

° Dangers of Parallel and Heterogeneous Computing

A little history

° Von Neumann and Goldstine - 1947

- "Can't expect to solve most big [n>15] linear systems without carrying many decimal digits [d>8], otherwise the computed answer would be completely inaccurate." - WRONG!

° Turing - 1949

- "Carrying d digits is equivalent to changing the input data in the d-th place and then solving Ax=b. So if A is only known to d digits, the answer is as accurate as the data deserves."
- Backward Error Analysis

° Rediscovered in 1961 by Wilkinson and publicized

° Starting in the 1960s- many papers doing backward error analysis of various algorithms

° Many years where each machine did FP arithmetic slightly differently

- Both rounding and exception handling differed
- Hard to write portable and reliable software
- Motivated search for industry-wide standard, beginning late 1970s
- First implementation: Intel 8087

° ACM Turing Award 1989 to W. Kahan for design of the IEEE Floating Point Standards 754 (binary) and 854 (decimal)

- Nearly universally implemented in general purpose machines

Defining Floating Point Arithmetic

° Representable numbers

- Scientific notation: +/- d.d…d x r^{exp}
- sign bit +/-
- radix r (usually 2 or 10, sometimes 16)
- significand d.d…d (how many base-r digits d?)
- exponent exp (range?)
- others?

° Operations:

- arithmetic: +,-,x,/,...
  - how to round result to fit in format
- comparison (<, =, >)
- conversion between different formats
  - short to long FP numbers, FP to integer
- exception handling
  - what to do for 0/0, 2*largest_number, etc.
- binary/decimal conversion
  - for I/O, when radix not 10

° Language/library support for these operations

IEEE Floating Point Arithmetic Standard 754 - Normalized Numbers

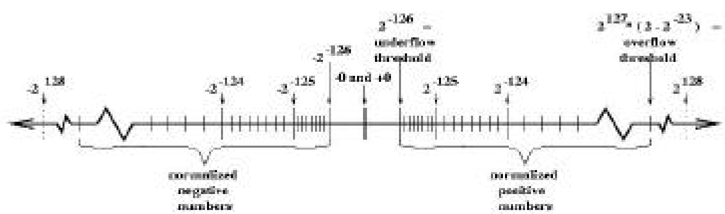
° **Normalized Nonzero Representable Numbers: +- 1.d…d x 2^{exp}**

- **Macheps** = Machine epsilon = $2^{-\#significand\ bits}$ = relative error in each operation
- **OV** = overflow threshold = largest number
- **UN** = underflow threshold = smallest number

Format	# bits	#significand bits	macheps	#exponent bits	exponent range
----------	--------	----------------------	------------	--------------------	----------------------
Single	32	23+1	2^{-24} (~10^{-7})	8	2^{-126} - 2^{127} (~10^{+-38})
Double	64	52+1	2^{-53} (~10^{-16})	11	2^{-1022} - 2^{1023} (~10^{+-308})
Double Extended (*80 bits on all Intel machines*)	>=80	>=64	<=2^{-64}(~10^{-19})	>=15	2^{-16382} - 2^{16383} (~10^{+-4932})

° **+- Zero: +-, significand and exponent all zero**

- Why bother with -0 later



Rules for performing arithmetic

° **As simple as possible:**

- Take the exact value, and round it to the nearest floating point number (**correct rounding**)
- Break ties by rounding to nearest floating point number whose bottom bit is zero (**rounding to nearest even**)
- Other rounding options too (**up, down, towards 0**)

° **Don't need exact value to do this!**

- Early implementors worried it might be too expensive, but it isn't

° **Applies to**

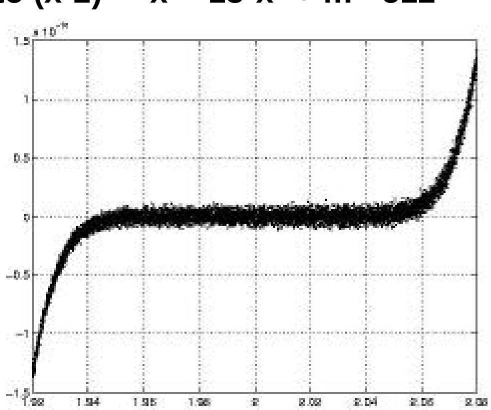
- +,-,*,/
- sqrt
- conversion between formats
- rem(a,b) = remainder of a after dividing by b
  - a = q*b + rem, q = floor(a/b)
  - cos(x) = cos(rem(x,2*pi)) for |x| >= 2*pi
  - cos(x) is *exactly* periodic, with period rounded(2*pi)

Error Analysis

- **Basic error formula**
  - fl(a op b) = (a op b)*(1 + d) where
    - op one of +,-,*,/
    - |d| <= macheps
    - assuming no overflow, underflow, or divide by zero
- **Example: adding 4 numbers**
  - $fl(x_1+x_2+x_3+x_4) = \{[(x_1+x_2)*(1+d_1) + x_3]*(1+d_2) + x_4\}*(1+d_3)$
    $= x_1*(1+d_1)*(1+d_2)*(1+d_3) + x_2*(1+d_1)*(1+d_2)*(1+d_3)$
    $+ x_3*(1+d_2)*(1+d_3) + x_4*(1+d_3)$
    $= x_1*(1+e_1) + x_2*(1+e_2) + x_3*(1+e_3) + x_4*(1+e_4)$
    where each $|e_i| <\sim 3*macheps$
  - get exact sum of slightly changed summands $x_i*(1+e_i)$
  - Backward Error Analysis - algorithm called numerically stable if it gives the exact result for slightly changed inputs
  - Numerical Stability is an algorithm design goal

Example: polynomial evaluation using Horner's rule

- **Horner's rule to evaluate $p = \sum_{k=0}^{n} c_k * x^k$**
  - $p = c_n$, for k=n-1 downto 0, $p = x*p + c_k$
- **Numerically Stable**
- **Apply to $(x-2)^9 = x^9 - 18*x^8 + \ldots - 512$**

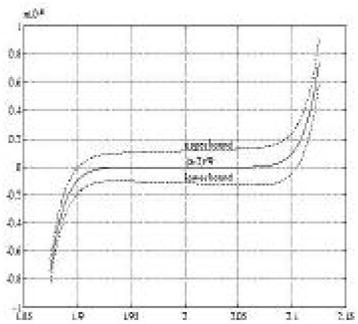


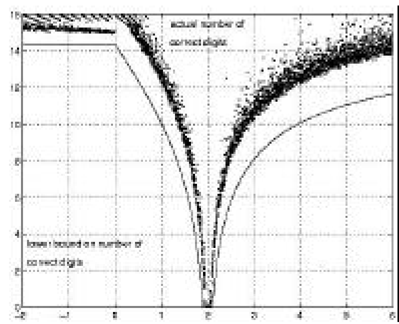
Example: polynomial evaluation (continued)

° $(x-2)^9 = x^9 - 18*x^8 + \ldots - 512$

° We can compute error bounds using

- fl(a op b)=(a op b)*(1+d)





What happens when the "exact value" is not a real number, or is too small or too large to represent accurately?

You get an "exception"

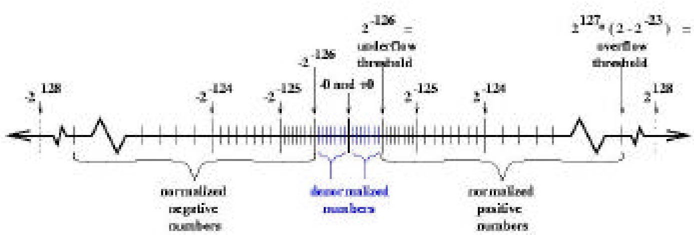
Exception Handling

- What happens when the “exact value” is not a real number, or too small or too large to represent accurately?
- 5 Exceptions:
  - **Overflow** - exact result > OV, too large to represent
  - **Underflow** - exact result nonzero and < UN, too small to represent
  - **Divide-by-zero** - nonzero/0
  - **Invalid** - 0/0, sqrt(-1), …
  - **Inexact** - you made a rounding error (very common!)
- Possible responses
  - Stop with error message (unfriendly, not default)
  - Keep computing (default, but how?)

IEEE Floating Point Arithmetic Standard 754 - “Denorms”

- Denormalized Numbers: +-0.d…d x 2^{min_exp}
  - sign bit, nonzero significand, minimum exponent
  - Fills in gap between UN and 0
- Underflow Exception
  - occurs when exact nonzero result is less than underflow threshold UN
  - Ex: UN/3
  - return a denorm, or zero
- Why bother?
  - Necessary so that following code never divides by zero

    if (a != b) then x = a/(a-b)



IEEE Floating Point Arithmetic Standard 754 - +- Infinity

° **+- Infinity: Sign bit, zero significand, maximum exponent**

° **Overflow Exception**
- occurs when exact finite result too large to represent accurately
- Ex: 2*OV
- return +- infinity

° **Divide by zero Exception**
- return +- infinity = 1/+-0
- sign of zero important!

° **Also return +- infinity for**
- 3+infinity, 2*infinity, infinity*infinity
- Result is exact, not an exception!

IEEE Floating Point Arithmetic Standard 754 - NAN (Not A Number)

° **NAN: Sign bit, nonzero significand, maximum exponent**

° **Invalid Exception**
- occurs when exact result not a well-defined real number
- 0/0
- sqrt(-1)
- infinity-infinity, infinity/infinity, 0*infinity
- NAN + 3
- NAN > 3?
- Return a NAN in all these cases

° **Two kinds of NANs**
- Quiet - propagates without raising an exception
- Signaling - generate an exception when touched
  - good for detecting uninitialized data

Exception Handling User Interface

- Each of the 5 exceptions has the following features
  - A sticky flag, which is set as soon as an exception occurs
  - The sticky flag can be reset and read by the user

    reset overflow_flag and invalid_flag

    perform a computation

    test overflow_flag and invalid_flag to see if any exception occurred
  - An exception flag, which indicate whether a trap should occur
    - Not trapping is the default
    - Instead, continue computing returning a NAN, infinity or denorm
    - On a trap, there should be a user-writable exception handler with access to the parameters of the exceptional operation
    - Trapping or "precise interrupts" like this are rarely implemented for performance reasons.

Exploiting Exception Handling to Design Faster Algorithms

- Paradigm:

  1) Try fast, but possibly "risky" algorithm
  2) Quickly test for accuracy of answer (use exception handling)
  3) In rare case of inaccuracy, rerun using slower "low risk" algorithm

- Quick with high probability
  - Assumes exception handling done quickly
- Ex 1: Solving triangular system Tx=b
  - Part of BLAS2 - highly optimized, but risky
  - If T "nearly singular", expect very large x, so scale inside inner loop: slow but low risk
  - Use paradigm with sticky flags to detect nearly singular T
  - Up to 9x faster on Dec Alpha
- Ex 2: Computing eigenvalues, up to 1.5x faster on CM-5

  For k= 1 to n
  $d = a_k - s - b_k^2/d$
  if |d| < tol, d = -tol
  if d < 0, count++

  vs.

  For k= 1 to n
  $d = a_k - s - b_k^2/d$ ... ok to divide by 0
  count += signbit(d)

- Demmel/Li (www.cs.berkeley.edu/~xiaoye)

Summary of Values Representable in IEEE FP

	Sign	Exponent	Fraction
° +- Zero	+-	0…0	0……………………0
° Normalized nonzero numbers	+-	Not 0 or all 1s	anything
° Denormalized numbers	+-	0…0	nonzero
° +-Infinity	+-	1….1	0……………………0
° NANs	+-	1….1	nonzero

- Signaling and quiet
- Many systems have only quiet

Simulating extra precision

- What if 64 or 80 bits is not enough?
  - Very large problems on very large machines may need more
  - Sometimes only known way to get right answer (mesh generation)
  - Sometimes you can trade communication for extra precision
- Can simulate high precision efficiently just using floating point
- Each extended precision number s is represented by an array $(s_1,s_2,\ldots,s_n)$ where
  - each s_k is a FP number
  - $s = s_1 + s_2 + \ldots + s_n$ in exact arithmetic
  - $s_1 >> s_2 >> \ldots >> s_n$
- Ex: Computing $(s_1,s_2) = a + b$

  if $|a|<|b|$, swap them

  $s_1 = a+b$ *… roundoff may occur*

  $s_2 = (a - s_1) + b$ *… no roundoff!*
  - s_1 contains leading bits of a+b, s_2 contains trailing bits
- Systematic algorithms for arbitrary precision
  - Priest / Shewchuk (www.cs.berkeley.edu/~jrs)
- Current effort to define extra precise BLAS this way
  - www.netlib.org/cgi-bin/checkout/blast/blast.pl

Cray Arithmetic

° **Historically very important**
  - Crays among the fastest machines
  - Other fast machines emulated it (Fujitsu, Hitachi, NEC)

° **Sloppy rounding**
  - fl(a + b) not necessarily (a + b)(1+d) but instead
    fl(a + b) = a*(1+d_a) + b*(1+d_b)   where $|d_a|,|d_b|$ <= macheps
  - Means that fl(a+b) could be either 0 when should be nonzero, or twice too large when a+b "cancels"
  - Sloppy division too

° **Some impacts:**
  - arccos(x/sqrt(x^2 + y^2)) can yield exception, because x/sqrt(x^2 + y^2) >1
    - not on any other computer
  - Best available eigenvalue algorithm fails
    - Need Π_k (a_k - b_k) accurately
    - Need to preprocess by setting each a_k = 2*a_k - a_k (kills bottom bit)

° **Latest Cray (=SGI) machine partially adopt IEEE (but SV1?)**

Hazards of Parallel and Heterogeneous Computing

° **What new bugs arise in parallel floating point programs?**

° **Ex 1: Nonrepeatability**
  - Makes debugging hard!

° **Ex 2: Different exception handling**
  - Can cause programs to hang

° **Ex 3: Different rounding (even on IEEE FP machines)**
  - Can cause hanging, or wrong results with no warning

° **See www.netlib.org/lapack/lawns/lawn112.ps**

Hazard #1: Nonrepeatability due to nonassociativity

° Consider s= all_reduce(x,"sum") = x1 + x2 + … + xp

° Answer depends on order of FP evaluation

- All answers differ by at most p*macheps*(|x1| + … + |xp|)
- Some orders may overflow/underflow, others not!

° How can order of evaluation change?

- Change number of processors
- In reduction tree, have each node add first available child sum to its own value
  - order of evaluation depends on race condition, unpredictable!

° Options

- Live with it, since difference likely to be small
- Build slower version of all_reduce that guarantees evaluation order independent of #processors, use for debugging

Hazard #2: Heterogeneity: Different Exception Defaults

° Not all processors implement denorms fast

- DEC Alpha 21164 in "fast mode" flushes denorms to zero
  - in fast mode, a denorm operand causes a trap
  - slow mode, to get underflow right, slows down all operations significantly, so rarely used
- SUN Ultrasparc in "fast mode" handles denorms correctly
  - handles underflow correctly at full speed
  - flushing denorms to zero requires trapping, slow

° Imagine a NOW built of DEC Alphas and SUN Ultrasparcs

- Suppose the SUN sends a message to a DEC containing a denorm: the DEC will trap
- Avoiding trapping requires running either DEC or SUN in slow mode
- Good news: most machines converging to fast and correct underflow handling

Hazard #3: Heterogeneity: Data Dependent Branches

° Mixed Cray/IEEE machines may round differently

° Different "IEEE machines" may round differently
  - Intel uses 80 bit FP registers for intermediate computations
  - IBM RS6K has MAC = Multiply-ACcumulate instruction
    - d = a*b+c with one rounding error, i.e. a*b good to 104 bits
  - SUN has neither "extra precise" feature
  - Different compiler optimizations may round differently (yuck)

° Impact: same expression can yield different values on different machines

```
{ Compute s redundantly
     or
{ s = reduce_all(x,min)
if (s > 0) then
     compute and return a
else
     communicate
     compute and return b
end
```

° Taking different branches can yield nonsense, or deadlock
  - How do we fix this example? Does it always work?

Further References on Floating Point Arithmetic

° Notes for Prof. Kahan's CS267 lecture from 1996
  - www.cs.berkeley.edu/~wkahan/ieee754status/cs267fp.ps
  - Note for Kahan 1996 cs267 Lecture

° Prof. Kahan's "Lecture Notes on IEEE 754"
  - www.cs.berkeley.edu/~wkahan/ieeestatus/ieee754.ps

° Prof. Kahan's "The Baleful Effects of Computer Benchmarks on Applied Math, Physics and Chemistry
  - www.cs.berkeley/~wkahan/ieee754status/baleful.ps

° Notes for Demmel's CS267 lecture from 1995
  - www.cs.berkeley.edu/~demmel/cs267/lecture21/lecture21.html