Software Aspects of IEEE Floating-Point Computations for Numerical Applications in High Energy Physics

J.M. Arnold
Intel Compiler and Languages
Developer Products Division
Software and Services Group
Intel Corporation

11 May 2010

Agenda

- Standards
- Floating-Point Numbers
- Common Formats and Hardware
- Rounding Modes and Errors
- They're Not Real!
- Techniques for Improving Accuracy
- Compiler Options
- Pitfalls and Hazards
- References
- Q & A

Standard for Floating-Point Arithmetic

IEEE 754-2008

- It's the most widely-used standard for floating-point computation
- It is followed by most modern hardware and software implementations
- Some software assumes IEEE 754 compliance
- Replaces earlier standards such as IEEE 74-1985

IEEE 754-2008

The standard defines

- Arithmetic formats
  - finite numbers, infinities, NANs
- Interchange formats
  - encodings as bit strings
  - binary formats
- Rounding algorithms
- Operations
- Exception handling

What is a Floating-Point Number?

value $x = (-1)^s \beta^e \times m$

where

sign	$s \in \{0, 1\}$
radix	$\beta \in \{2, 10\}$
exponent	$e \in \{e_{min}, e_{max}\}$
significand	$m = \sum_{i=0}^{p-1} d_i \beta^{-i}$
digits	$d_i \in [0, \beta - 1], d_0 \neq 0$ generally

What is a Floating-Point Number?

Some examples for $\beta = 2$:

$$
\begin{aligned}
4.0 &= (-1)^0 \times 2^2 \times 1.0 \cdots 0 \\
-0.1 &= (-1)^1 \times 2^{-4} \times 1.\underline{1001} \cdots \\
0.01 &= (-1)^0 \times 2^{-7} \times 1.\underline{01000111101011100001} \cdots
\end{aligned}
$$

Special Floating-Point Values

- ± 0
  - Yes, there is a -0
  - $+0 == -0$ but $1.0/\pm 0.0 \Rightarrow \pm\infty$
- $\pm\infty$
- NaN
  - Not a number. E.g., $\sqrt{-1}$
- Denormals
  - $|x| < \beta^{e_{min}}$
  - $0 < m < 1$ $(d_0 = 0)$

Common Floating-Point Formats

	β	p	e_{min}	e_{max}	Size
float	2	24	-126	+127	32 bits
double	2	53	-1022	+1023	64 bits
extended	2	64	-16382	+16383	80 bits
quad	2	113	-16382	+16383	128 bits

- extended is found in x87-style hardware
- on Itanium, extended is 82 bits
- quad is typically emulated in software

x87 Floating-Point Hardware

- Introduced with the Intel 8087 floating-point co-processor
- 8 floating-point registers implemented as a stack
- Supports single, double and extended formats
- Rounding precision only controls the size of the significand, not the exponent range
- Potential exists for "double rounding" problems

  Consider $1848874847.0 \otimes 19954562207.0$:

  The result is 36893488147419103232 using x87

  but 36893488147419111424 using SSE

  36893488147419107329 is exact

SSE Floating-Point Hardware

- Supports float and double formats
- The number of SSE registers and their sizes vary by processor but the format of float and double remain the same
- Permits better reproducibility because all results are either float or double; no extended significand or increased exponent range as with x87 hardware
- Supported by both SISD and SIMD instructions

Rounding Modes

There are four rounding modes

- Round to nearest even
  - round to the nearest floating-point number
  - if midway between numbers, round to the floating-point number with the even significand
  - this is the default
- Round toward $+\infty$
- Round toward $-\infty$
- Round toward 0
  - also called chopping or truncation

Rounding Modes

- Many math libraries and other software make assumptions about the current rounding mode of a process
- Don't change the default unless you really know what you're doing
- And if you know what you're doing, you probably won't change it

Errors

- ulp $\Rightarrow$ units in the last place

  for $x \in [\beta^e, \beta^{e+1}], ulp(x) = \beta^{e-p+1}$
- Fundamental operations produce correctly rounded results

  they have an absolute error $\leq 0.5\, ulp$ provided no exceptions occur
- Compilers and math libraries may trade accuracy for performance
  - "fast" math libraries
  - reduced accuracy math libraries
  - rearrangements such as $x/y \Rightarrow x * (1.0/y)$

Floating-Point Numbers are not Real!

- In each interval $[\beta^e, \beta^{e+1})$, there are β^{p-1} floating-point numbers

  but there are many more real numbers in that interval

- Even if a and b are floating-point numbers, $a + b$ may not be a floating-point number

- Floating-point operations may not associate

  $(a \oplus b) \oplus c$ may not equal $a \oplus (b \oplus c)$

- Floating-point operations may not distribute

  $a \otimes (b \oplus c)$ may not equal $(a \otimes b) \oplus (a \otimes c)$

Floating-Point Numbers are not Real!

For example, if

$$a = 10^{+30}$$
$$b = -a$$
$$c = 1.0$$

then

$$(a \oplus b) \oplus c = 1.0$$
$$a \oplus (b \oplus c) = 0.0$$

Techniques for Improving Accuracy

- Accurate summation
  - adding values while avoiding
    - loss of precision
    - catastrophic cancellation
- Accurate multiplication
- Accurate interchange of data

Accurate Summation Techniques

- Use double precision
- Sort the values before adding
  - sort by value or absolute value
  - sort by increasing or decreasing
- Process positive and negative values separately
- Dekker's extended-precision addition algorithm

Dekker's Extended-Precision Addition Algorithm

Compute s and t such that $s = a \oplus b$ and $a + b = s + t$

```
void Dekker(const double a, const double b,
            double* s, double* t) {
    if (abs(b) > abs(a)) {
        double temp = a;
        a = b;
        b = temp;
    }
    // Don't allow value-unsafe optimizations
    *s = a + b;
    double z = *s - a;
    *t = b - z;
    return;
}
```

Kahan's Summation Algorithm

Sum a series of numbers accurately

```
double Kahan(const double a[], const int n) {
    double s = a[0];
    double t = 0;
    for(int i = 1; i < n; i++) {
        double y = a[ i ] - t;
        double z = s + y;
        t = ( z - s ) - y;
        s = z;
    }
    return s;
}
```

Accurate Multiplication

- Veltkamp splitting

  split $x \Rightarrow x_h + x_l$ where the number of non-zero digits in each significand is $\approx p/2$

  this can be done exactly using normal floating-point operations

- Dekker's multiplication scheme

  $z = x * y \Rightarrow z_h + z_l$

  again, this can be done exactly using normal floating-point operations

Veltkamp Splitting

```
void vSplitting(const double x, const int sp,
                double* x_high, double* x_low) {
    unsigned long C = ( 1UL << sp ) + 1;
    double a = C * x;
    double b = x - a;
    *x_high = a + b;
    *x_low = x - *x_high;
}
```

Dekker Multiplication

```
void dMultiply(double x, double y, double* r_1, double* r_2) {
    const int SHIFT_POW = 27; // 53/2 for double precision
    double x_high, x_low, y_high, y_low;
    double a, b, c;
    vSplit(x, SHIFT_POW, &x_high, &x_low);
    vSplit(y, SHIFT_POW, &y_high, &y_low);
    *r_1 = x * y;
    a = x_high * y_high - *r_1;
    b = a + x_high * y_low;
    c = b + x_low * y_high;
    *r_2 = c + x_low * y_low;
}
```

Accurate Interchange

- Use binary files
- Reading and writing using `%f` isn't good enough

  internal ⇒ external ⇒ internal may not recover the same value

- Use `%a` (or `%A`) formatting to print floating-point data
  - the value is formatted as `[-]0xh.hhhh...p±d`
  - the usual length modifiers apply (e.g., `%l` or `%L`)
  - major limitation: not all linuxes support `%a` for input
  - an example where $x = 0.1, y = x * x, z = 0.01$

$$x = 0.100000 \text{ (0x1.999999999999ap-4)}$$
$$y = 0.010000 \text{ (0x1.47ae147ae147bp-7)}$$
$$z = 0.010000 \text{ (0x1.47ae147ae147cp-7)}$$

Compiler Options

Compiler Options Control

- Value safety
- Expression evaluation
- Precise exceptions
- Floating-point contractions
- "Force to zero"
    - denormals are forced to 0
    - may improve performance, especially if hardware doesn't support denormals

Value Safety

Transformations which may affect results

- Reassociation

  $(x + y) + z \Rightarrow x + (y + z)$

- Distribution

  $x * (y + z) \Rightarrow x * y + x * z$

- Vectorized loops
- Reductions
- OpenMP reductions

Compiler Options – `icc`

The `-fp-model` keyword controls floating-point semantics

- `fast[=1|2]`; default is `fast=1`

  allows “value-unsafe” optimizations

- precise

  allows value-safe optimizations only

- source — double — extended

  precision of intermediate results

- except

  strict exception semantics

- may be specified more than once

Compiler Options – `icc`

To improve the reproducibility of results

- `-fp-model precise`

  value-safe optimizations only

- `-fp-model source`

  intermediate precisions as written

- `-ftz`

  no denormals; e.g., abrupt underflows

- but performance relative to `-O3` will be affected

Compiler Options – `gcc`

Same capabilities as with `icc` but option names are different

- `-funsafe-math-optimizations`

  allows unsafe optimizations; a "composite" option

- `-fassociative-math`

  allows reassociations

- `-ffast-math`

  a "composite" option

- `-freciprocal-math`

  replace divides by multiplication

- and many more

  very few are enabled by any `-O` switch

Compiler Options – `gcc`

Compared with `icc`, `gcc` is more conservative, cautious and strict about its choice of defaults for floating-point optimizations

Be Aware of Approximation Errors

- Neither 0.1 nor 0.01 can be presented exactly as floating-point numbers

  $(0.1) \otimes (0.1) \neq fl(0.01)$

Testing for Equality

- Testing floating-point numbers for equality can be problematic
  - especially if the values are computed
    roundoff error
  - even if they are constants
    approximation error
  - beware of never-ending loops
    `while (a != b) {...}`
  - consider using $\leq$, $\geq$ etc depending on the nature of the algorithm

Testing for Equality

- Testing floating-point numbers for equality can be problematic
  - using absolute errors is usually wrong
    `if (abs(a-b) < 1.0-8){...}`
  - use relative errors
    `if (abs(a-b)/b < epsilon){...}`
    but avoid dividing by 0!
  - you may want to use $ulp(a)$ and $ulp(b)$
  - consider writing an `AlmostEqual` routine

Be Aware of Consistency Errors

Assume an x87 hardware environment

```
...
x = ...
y = ... // result probably in a floating-point register
if ( x != y ) {
  ...
  // no changes to x or y but y may have been written to memory
  ...
}
if ( x == y ) { // result may be inconsistent with previous test
  ...
}
```

A Recent Example

Itanium hardware environment with Fused Multiply-Add (FMA)

```
...
a += b*c - d*e
...
```

To make better use of FMA, the compiler changed this into

```
...
a = ( a - d*e ) + b*c
...
```

and the answer changed and a ROOT stress test failed! Using `-fp-model strict` solved the problem.

References

1. To write good floating-point code, you *must* read "What Every Computer Scientist Should Know About Floating-Point Arithmetic," by David Goldberg. ACM Computing Surveys 23, 1, 5-48 (1991)

2. An excellent recent text: "Handbook of Floating-Point Arithmetic," by J-M Muller et al. (Birkhäuser, 2010)

3. "Art of Computer Programming, Volume 2: Seminumerical Algorithms." Donald Knuth.

Recent Papers from Intel

1. "Consistency of Floating-Point Results" by Corden and Kreitzer
2. "Floating-Point Calculations and the ANSI C, C++ and Fortran Standards" by Corden and Kreitzer

Questions?