Workshop on Numerical Computing

Floating-Point Arithmetic

Jeff Arnold, Intel Corporation

25 September 2012

Agenda

- Part I – Fundamentals
  - Motivation
  - Some properties of floating-point numbers
  - Standards
  - More about floating-point numbers
  - A trip through the floating-point numbers
- Part II – Techniques
  - Error-free transformations
  - Summation
  - Dot product
  - Polynomial evaluation

Motivation

- Why is floating-point arithmetic important?
- Reasoning about floating-point arithmetic
- Why do standards matter?
- Techniques which improve floating-point
  - Accuracy
  - Versatility
  - Performance

Why is Floating-Point Arithmetic Important?

- It is ubiquitous in scientific computing
  - Most research in HEP can’t be done without it
- Need to implement algorithms which
  - Get the best answers
  - Get the best answers quickly
  - Get the best answers all the time
- A rigorous approach to floating-point is seldom taught in programming courses
  - Too many think floating-point arithmetic is
    - Approximate in a random ill-defined sense
    - Mysterious
    - Often wrong

Reasoning about Floating-Point Arithmetic

It’s important because

- One can prove algorithms are correct
  - One can even prove they are portable
- One can estimate the round-off and approximate errors in calculations
- This knowledge increases confidence in floating-point calculations and results

Some Properties of Floating-Point Numbers

- They aren’t the same as the real numbers encountered in mathematics
  - They do not form a field
  - Some common rules of arithmetic are not always obeyed
  - There are only a finite number of them
  - They are all rational numbers
    - but they are only a subset of the rationals
    - thus none of them are irrational

Some Properties of Floating-Point Numbers

- Even if a and b are floating-point numbers, $a \oplus b$ may not be
  - Similarly for $\ominus$, $\otimes$ and $\oslash$
- Operations may not associate:
  - $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$
  - Similarly for $\ominus$ and $\otimes$
- Operations may not distribute:
  - $a \otimes (b \oplus c) \neq (a \otimes b) \oplus (a \otimes c)$

Standards

There have been three major standards affecting floating-point arithmetic:

- IEEE 754-1985 Standard for Binary Floating-Point Arithmetic
- IEEE 854-1987 Standard for Radix Independent Floating-Point Arithmetic
- IEEE 754-2008 Standard for Floating-Point Arithmetic
  - We will concentrate on this one since it is current

IEEE 754-1985

Standardized/specified

- Formats
- Rounding modes
- Operations
- Special values
- Exceptions

IEEE 754-1985

- Only described binary floating-point arithmetic
- Two basic formats specified:
  - single precision (mandatory)
  - double precision
- An extended format was associated with each basic format
  - Double extended: the IA32 “80-bit” format

IEEE 854-1987

- “Radix-independent”
  - But essentially only radix 2 or 10 considered
- Established constraints on the relationships between
  - Number of bits of precision
  - Mininum and maximum exponent
- Established constraints between various formats

The Need for a Revision

- Standardize common practices
  - Quadruple precision
- Standardize effects of new/improved algorithms
  - Radix conversion
  - Correctly rounded elementary functions
- Remove ambiguities
- Improve portability

IEEE 754-2008

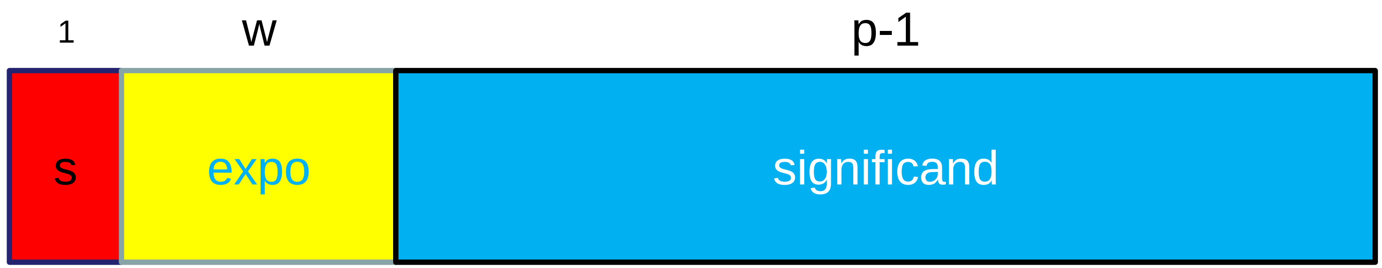
- Merged 754-1985 and 854-1987
  - But tried not to invalidate hardware which conformed to 754-1985
- Standardized
  - Quadruple precision
  - Fused multiply-add (FMA)
- Resolve ambiguities
  - Aids portability between implementations

IEEE 754-2008

Formats

- Interchange
  - Used to exchange floating-point data between implementations/platforms
  - Fully specified as bit strings
    - Does not address endianness
- Extended and Extendable formats
  - Encodings not specified
  - May match interchange formats
- Arithmetic formats
  - A format which represents operands and results for all operations required by the standard

Format of a Binary Floating-point Number



IEEE Name	Format	Storage Size	w	p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

IEEE 754-2008

Formats

- Basic formats:
  - Binary with lengths of 32, 64 and 128 bits
  - Decimal with lengths of 64 and 128 bits
- Other formats:
  - Binary with a length of 16 bits
    - $p = 11$
    - $e_{min} = -14$, $e_{max} = +15$
  - Decimal with a length of 32 bits

Larger Formats

- Parameterized based on size k:
  - $k \geq 128$ and must be a multiple of 32
  - $p = k - roundnearest(4 \times log_2(k)) + 13$
  - $w = k - p$
  - $e_{max} = 2^{w-1} - 1$
- For example, on all conforming platforms, Binary1024 will have:
  - $k = 1024$
  - $p = 1024 - 40 + 13 = 997$
  - $w = 27$
  - $e_{max} = +67108863$

IEEE 754-2008

- Radix
  - Either 2 or 10
- Representation specified by
  - Radix
  - Sign
  - Exponent
    - Biased exponent
    - e_{min} must be equal to $1 - e_{max}$
  - Significand
    - “hidden bit” format used for normal values

We're not going to consider every possible format

For this workshop, we will limit our discussion to

- Radix 2
- Binary32, Binary64 and Binary128 formats
  - Covers SSE and AVX
    - I.e., modern processors
  - Not considering "double extended" format
    - "IA32 x87" format
  - Not considering decimal formats
- Round to nearest even

Value of a Floating-Point Number

The value of a floating-point number is determined by 4 quantities:

- sign $s \in \{0,1\}$
- radix β
  - Sometimes called the "base"
- precision p
  - the digits are x_i, $0 \le i < p$, where $0 \le x_i < \beta$
- exponent e is an integer
  - $e_{min} \le e \le e_{max}$

Value of a Floating-Point Number

The value of a floating-point number can be expressed as

$$x = (-)^s \beta^e \sum_{i=0}^{p-1} x_i \beta^{-i}$$

where the significand is

$$m = \sum_{i=0}^{p-1} x_i \beta^{-i}$$

with

$$0 \le m < \beta$$

Value of a Floating-Point Number

The value can also be written

$$x = (-)^s \beta^{e-p+1} \sum_{i=0}^{p-1} x_i \beta^{p-i-1}$$

where the integral significand is

$$M = \sum_{i=0}^{p-1} x_i \beta^{p-i-1}$$

with

$$0 \leq M < \beta^p$$

Operations specified by IEEE 754-2008

- Addition, subtraction
- Multiplication
- Division
- Remainder
- Square root
- All with correct rounding
  - correct rounding: return the correct finite result using the current rounding mode

Operations

- Conversion to/from integer
  - Conversion to integer must be correctly rounded
- Conversion to/from decimal strings
  - Conversions must be monotonic
  - Under some conditions, binary→decimal→binary conversions must be exact

Special Values

- Zero
  - signed
- Infinity
  - signed
- NaN
  - Quiet NaN
  - Signaling NaN
  - NaNs do not have a sign: they aren't a number
    - the sign bit is ignored
  - NaNs can "carry" information

Exceptions Specified by IEEE 754-2008

- Underflow
  - Absolute value of a non-zero result is less than $\beta^{e_{min}}$ (i.e., it is subnormal)
  - Some ambiguity: before or after rounding?
- Overflow
  - Absolute value of a result greater than the largest finite value $\Omega = 2^{e_{max}} \times (2 - 2^{1-p})$
  - Result is $\pm\infty$
- Division by zero
  - x/y where x is finite and non-zero and $y = 0$
- Inexact
  - Result, after rounding, is not exact
- Invalid

Exceptions Specified by IEEE 754-2008

- Invalid
  - An operand is a sNaN
  - $\sqrt{x}$ where $x < 0$
    - however $\sqrt{-0} = -0$
  - $(-\infty) + (+\infty)$, $(+\infty) + (-\infty)$
  - $(-\infty) - (-\infty)$, $(+\infty) - (+\infty)$
  - $(\pm 0) \times (\pm\infty)$
  - $(\pm 0)/(\pm 0)$ or $(\pm\infty)/(\pm\infty)$
  - some floating-point $\rightarrow$integer or decimal conversions

Rounding Modes in IEEE 754-2008

- round to nearest
  - round to nearest even
    - in the case of ties, select the result whose significand is even
    - required for binary and decimal
    - the default rounding mode for binary
  - round to nearest away
    - required only for decimal
- round toward $+\infty$
- round toward $-\infty$
- round toward 0

Transcendental and Algebraic Functions

The standard **recommends** the following functions be correctly rounded:

- e^x, $e^x - 1$, 2^x, $2^x - 1$, 10^x, $10^x - 1$
- $log_\alpha(\Phi)$ for $\alpha = e, 2, 10$ and $\Phi = x, 1 + x$
- $\sqrt{x^2 + y^2}$, $1/\sqrt{x}$, $(1 + x)^n$, x^n, $x^{1/n}$
- $\sin(x)$, $\cos(x)$, $\tan(x)$, $\sinh(x)$, $\cosh(x)$, $\tanh(x)$ and the inverse functions
- $\sin(\pi x)$, $\cos(\pi x)$
- And more...

Transcendental Functions

Why this may be difficult to do...

Consider $2^{1.e4596526bf94dp-31}$

- The correct answer is
  $1.0052fc2ec2b537ffffffffffffff4$...
- You need to know the result to 115 bits to determine the correct rounding.
- "The Table-Makers Dilemma"
  - Rounding $\approx f(x)$ gives same result as rounding $f(x)$
- See publications from ENS group

Table-Makers Dilemma

"No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits."

W. Kahan

Convenient Properties

Exact operations

- If $\frac{y}{2} \leq x \leq 2y$ and subnormals are available, then $x - y$ is exact
  - Sterbenz's lemma
- But what about catastrophic cancellation?
  - Subtracting nearly equal numbers loses accuracy
- The subtraction itself does not introduce any error
  - it may amplify a pre-existing error

Convenient Properties

Exact operations

- Multiplication/division by 2^n is exact
  - In the absence of under/overflow
- Multiplication of numbers with significands having sufficient low-order 0 digits
  - Precise splitting and Dekker's multiplication

Walking Through Floating-point Numbers

- `0x0000000000000000` — +zero
- `0x0000000000000001` — smallest subnormal
- `...`
- `0x000fffffffffffff` — largest subnormal
- `0x0010000000000000` — smallest normal
- `...`
- `0x001fffffffffffff`
- `0x0020000000000000` — 2 X smallest normal

Walking Through Floating-point Numbers

- `0x0020000000000000` — 2 X smallest normal
- `. . .`
- `0x7fefffffffffffff` — largest normal
- `0x7ff0000000000000` — +infinity
- `0x7ff0000000000001` — NaN
- `. . .`
- `0x7fffffffffffffff` — NaN
- `0x8000000000000000` — -zero

Walking Through Floating-point Numbers

- `0x8000000000000000` — -zero
- `0x8000000000000001` — “smallest” negative subnormal
- ...
- `0x800fffffffffffff` — “largest” negative subnormal
- `0x8010000000000000` — “smallest” negative normal
- ...
- `0xfff0000000000000` — -infinity
- `0xfff0000000000001` — NaN
- ...
- `0xffffffffffffffff` — NaN

End of Part I

Time for a break...

Q & A

CERN openlab

Part II -- Techniques

- Error-Free Transformations
- Summation
- Dot Products
- Polynomial Evaluation
- Data Interchange

Notation

- Floating-point operations are written:
  - $\oplus$ addition
  - $\ominus$ subtraction
  - $\otimes$ multiplication
  - $\oslash$ division
- $a \oplus b$ represents the floating-point addition of a and b
  - a and b are floating-point numbers
  - the result is a floating-point number
  - in general, $a + b \neq a \oplus b$
- A generic floating-point operation on x is written $\circ(x)$

Error-Free Transformations

An error-free transformation (EFT) is an algorithm which determines the rounding error associated with a floating-point operation.

- Addition/subtraction

$$a + b = (a \oplus b) + t$$

- Multiplication

$$a \times b = (a \otimes b) + t$$

- There are others

Error-Free Transformations

- Under most conditions, the rounding error is itself a floating-point number
  - $a + b = s + t$ where $s = a \oplus b$
  - all values are floating-point numbers
  - This is still a powerful analytical tool even when t is not a floating-point number
- An EFT can be implemented using only floating-point computations in the working precision
- Rounding error is often called the approximation error

EFT for Addition: FastTwoSum

Compute $a + b = s + t$ where

- $|a| \geq |b|$
- $s = a \oplus b$

```
void
FastTwoSum( const double a, const double b,
            double* s, double* t ) {
    //  Requires that |a| ≥ |b|
    //  No unsafe optimizations!
    *s = a + b;
    *t = b - ( *s - a );
    return;
}
```

EFT for Addition: TwoSum

Compute $a + b = s + t$ where

- $s = a \oplus b$

```
void
TwoSum( const double a, const double b,
        double* s, double* t ) {
    //  No unsafe optimizations!
    *s = a + b;
    double z = *s – b;
    *t = ( a - z ) + ( b - ( *s - z ) );
    return;
}
```

EFTs for Addition

- A realistic implementation of `FastTwoSum` requires 3 floating-point operations and a branch
- `TwoSum` takes 6 floating-point operations but requires no branches
- `TwoSum` is usually faster on modern processors
- Recall that this discussion is restricted to radix 2 and round to nearest even
  - this is required to prove `TwoSum`

Precise Splitting Algorithm

- Known as Veltkamp's algorithm
- Calculates x_h and x_l such that $x = x_h + x_l$ exactly
- For $\delta < p$, where δ is a parameter,
  - The significand of x_h fits in $p - \delta$ digits
  - The significand of x_l fits in δ digits
- No information is lost in the transformation

Precise Splitting

Code fragment

```
void
Split( const double x, const int delta,
       double* x_h, double* x_l ) {
    //  No unsafe optimizations!
    unsigned long c = (1UL << delta) + 1;
    *x_h = ( c * x ) + ( x - ( c * x ) );
    *x_l = x - x_h;
    return;
}
```

Precise Multiplication

- Dekker's algorithm
- Computes s and t such that $a \times b = s + t$ where s= $a \otimes b$

Precise Multiplication Algorithm

```
#define SHIFT_POW 27 /* ⌈p/2⌉ for Binary64 */
void
Mult( const double a, const double b,
      double* s, double* t ) {
    double a_high, a_low, b_high, b_low;
    //  No unsafe optimizations!
    Split( a, SHIFT_POW, &a_high, &a_low );
    Split( b, SHIFT_POW, &b_high, &b_low );
    *s = x * y;
    *t = -*s + a_high * b_high ;
    *t += a_high * b_low + a_low * b_high;
    *t += a_low * b_low;
    return;
}
```

Summation Techniques

- Traditional
- Sorting and Insertion
- Compensated
- Distillation
- Multiple accumulators

- Reference: Higham

Summation Techniques

Condition number

$$C_{sum} = \frac{|\sum a_i|}{\sum |a_i|}$$

- If C_{sum} is “not too large,” the problem is not ill-conditioned and traditional methods may suffice
- But if C_{sum} is “too large,” we want results appropriate to higher precision without actually using a higher precision
- But if higher precision is available, use it!

Traditional Summation

- $s = \sum_{i=0}^{n} x_i$
- Code fragment

```
double
Sum( const double* x, const int n ) {
    int i;
    for ( i = 0; i < n; i++ ) {
        Sum += x[ i ];
    }
    return Sum;
}
```

Traditional Summation

What can go wrong?

- Catastrophic cancellation
  - loss of significance
  - magnitude of operands nearly equal but signs differ: $x \approx -y$
- Small terms encountered when running sum is large
  - the smaller terms don't affect the result
  - but later large magnitude terms may reduce the running sum

Sorting and Insertion

- Reorder the operands
  - Increasing magnitude
  - Decreasing magnitude
- Insertion
  - First sort by magnitude
  - Remove x_1 and x_2 and compute their sum
  - Insert that sum on the list keeping it sorted
  - Repeat until only 1 element is left on the list
- Many variations
  - If lots of cancellation, sorting by decreasing magnitude can be better
  - Sterbenz's lemma

Compensated Summation

- Based on `FastTwoSum` and `TwoSum` techniques
- Knowledge of the exact rounding error in a floating-point addition is used to correct the summation

Compensated Summation

- Code fragment

```
double
Kahan( const double* x, const int n ) {
    double sum = x[ 0 ];
    double c = 0.0;
    double y;
    int i;
    for ( i = 1; i < n; i++ ) {
        y = x[ i ] + c;
        FastTwoSum( sum, y, &sum, &c );
    }
    return sum;
}
```

Compensated Summation

- Many variations known
- Consult the extensive literature:
  - Kahan
  - Knuth
  - Priest
  - Pichat and Neumaier
  - Rump, Ogita and Oishi
  - Shewchuk
  - Arénaire Project (ENS)

Other Summation Techniques

- Distillation
  - Separate accumulators based on exponents of operands
  - Additions are always exact until the accumulators are finally added
- Long accumulators
  - Emulate greater precision
  - double-double

Choice of Summation Technique

- Performance
- Error bound
  - independent of n?
- Condition number
  - Is it known?
  - Difficult to determine?
  - Some algorithms allow it to be determined simultaneously with the sum
  - It can be used to evaluate the suitability of the result
- No one technique fits all situations all the time

Dot Product

- Use of EFTs
- Recast to summation
- Compensated dot product

Dot Product

- Condition number:

$$C_{dot\ product} = \frac{2\sum_{i=1}^{n}|a_i \cdot b_i|}{\left|\sum_{i=1}^{n} a_i \cdot b_i\right|}$$

- If C is not too large, a traditional algorithm can be used

Dot Product

- The dot product of 2 vectors of dimension n can be reduced to computing the sum of $2n$ floating-point numbers
  - Split each element
  - Form products
  - Sum accurately
- Algorithms can be constructed such that the result computed in precision p is as accurate as though the dot product was computed in precision $2p$ and then rounding back
- Consult the work of Ogita, Rump and Oishi

Polynomial Evaluation

- Horner’s method
- Use of EFTs
- Compensated

Polynomial Evaluation

Horner's method

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

where x and all a_i are all floating-point numbers

Polynomial Evaluation

- Code fragment

```
double
Horner( const double* a, const int n,
        double x ) {
    int i;
    double p = 0.0;
    for ( i = n; i >= 0; i-- ) {
        p = p * x + a[ i ];
    }
    return p;
}
```

Polynomial Evaluation

Compensated Horner's method:

- Let $p_0 = \texttt{Horner(a,n,x)}$
- Determine $\pi(x)$ and $\sigma(x)$ where
  - $\pi(x)$ and $\sigma(x)$ are polynomials of degree $n-1$ with coefficients π_i and σ_i
  - such that

$$p(x) = p_0 + \pi(x) + \sigma(x)$$

Polynomial Evaluation

Compensated Horner's method:

- $p(x) = p_0 + \pi(x) + \sigma(x)$
- Error analysis shows that under certain conditions, $p(x)$ is as accurate as evaluating p_0 in twice the working precision
- Even if those conditions are not met, one can apply the method recursively to $\pi(x)$ and $\sigma(x)$

Data Interchange

Moving floating-point data between platforms without loss of information?

- Exchange binary data
- Use of %a and %A
  - Encodes the internal bit patterns via hex digits
- Formatted decimal strings
  - Requires sufficient decimal digits to guarantee “round-trip” reproducibility
  - Depends on accuracy of run-time binary↔decimal conversion routines on all platforms

Bibliography

- D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys, 23(1):5–47, March 1991.
- J.-M. Muller et al, *Handbook of Floating-Point Arithmetic*, Birkäuser, Boston, 2010.
- N. J. Higham, *Accuracy and Stability of Numerical Algorithms (2nd edition)*, SIAM, 2002.

Bibliography

- Publications from CNRS/ENS Lyon/INRIA/Arénaire project (J.-M. Muller et al)
- Publications from Institute for Reliable Computing (Institut für Zuverlässiges Rechnen), Technische Universität Hamburg-Harburg (Siegfried Rump)

Q & A

CERN openlab