Techniques for Floating-Point Arithmetic

Jeff Arnold

5 May 2014

Agenda

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, $a \oplus b \neq a + b$
- A generic floating-point operation on a is written $\circ(x)$

Error-Free Transformations

An error-free transformation (EFT) is an algorithm which transforms a (small) set of floating-point numbers into another (small) set of floating-point numbers of the same precision without any loss of information.

$$f(a, b) \longmapsto g(s, t)$$

Example:

An EFT for addition determines floating-point numbers s and t from the floating-point numbers a and b such that $a + b = s + t$ where $s = (a \oplus b)$. Thus, t is a floating-point number which is equal to the rounding error of the floating-point addition.

Error-Free Transformations

EFTs exist for

- Addition: $a + b = (a \oplus b) + t$
- Multiplication: $a \times b = (a \otimes b) + t$
- Splitting: $a = s + t$

EFTs are most useful when they can be implemented only using operations in the current working precision.

Additional EFTs can be derived by composition. For example, an EFT for dot products makes use of those for addition and multiplication.

An EFT for Addition: FastTwoSum

Compute $a + b = s + t$ where

- $|a| \geq |b|$
- $s = a \oplus b$
- t is a floating-point number

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

An EFT for Addition: TwoSum

Compute $a + b = s + t$ where

- $s = a \oplus b$
- t is a floating-point number

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

EFTs for Addition

- A realistic implementation of `FastTwoSum` requires a branch and 3 floating-point opertions
- `TwoSum` takes 6 floating-point operations but requires no branches
- Thus `TwoSum` is usually faster on pipelined (e.g., modern) processors
- The algorithm used in `TwoSum` is valid in radix 2 even if underflow occurs (i.e., if subnormals are available)

Precise Splitting Algorithm

- Given a floating-point number x, determine the floating-point numbers x_h and x_l such that $x = x_h + x_l$
- For $\delta \leq p$, where δ is a parameter,
    - The signficand of x_h fits in $p - \delta$ digits
    - The signficand of x_l fits in δ digits
    - δ is typically chosed to be $\lceil p/2 \rceil$
- No information is lost in the transformation
- This scheme is known variously as Veltkamp's algorithm or the Veltkamp-Dekker algorithm

Precise Splitting EFT

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

Precise Multiplication

- Dekker's algorithm
- Given floating-point numbers a and b, determine floating-point numbers s and t such that $a \times b = s + t$ where $s = a \otimes b$

Precise Multiplication EFT

```
#define DELTA 27 // For Binary64
void
Mult(const double a, const double b,
     double* s, double* t)
{   // No unsafe optimizations!
    double a_high, a_low, b_high, b_low;
    Split(a, DELTA, &a_high, &a_low);
    Split(b, DELTA, &b_high, &b_low);
    *s = a * b;
    *t = -*s + a_high * b_high;
    *t += a_high * b_low;
    *t += a_low * b_high;
    *t += a_low * b_low;
    return;
}
```

Summation Techniques

- Traditional
- Sorting and Insertion
- Compensated
- Distillation
- Multiple Accumulators
- Reference: Higham: *Accuracy and Stability of Numerical Algorithms*

Summation Techniques

Condition number:

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

- If C_{sum} is not too large, the problem is not ill-conditioned and traditional methods may be sufficient
- If C_{sum} is too large, we need to have results appropriate to a higher precision *without actually using a higher precision*
- Obviously, if higher precision is readily available, use it

Traditional Summation

$$s = \sum_{i=0}^{n-1} x_i$$

```
double
Sum(const double* x, const unsigned int n)
{   // No unsafe optimizations!
    double sum = x[0]
    for(int i = 1; i < n; i++) {
        sum += x[i];
    }
    return;
}
```

Sorting and Insertion

- Reorder the operands
  - By value or magnitude
  - Increasing or decreasing
- Insertion
  - First sort by magnitude
  - Remove x_1 and x_2 and compute their sum
  - Insert that value into the list keeping the list sorted
  - Repeat until only one element is in the list
- Many Variations
  - If lots of cancellations, sorting by decreasing magnitude may be better
  - Sterbenz' 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
- Developed by William Kahan

Compensated (Kahan) Summation

```
double
Kahan(const double* x, const unsigned int n)
{   // No unsafe optimizations!
    double s = x[0];
    double t = 0.0;
    for( int i = 1; i < n_values; i++ ) {
        double y = x[i] - t;
        double z = s + y;
        t = ( z - s ) - y;
        s = z;
    }
    return s;
}
```

Compensated Summation

Many variations known. Consult the literature:

- Kahan
- Knuth
- Priest
- Pichat and Neumaier
- Rump, Ogita and Oishi
- Shewchuk
- AriC project (CNRS/ENS Lyon/INRIA)

Other Summation Techniques

- Distillation
  - Separate accumulators based on exponents of operands
  - Additions are always exact until the accumulators are finally summed
- Long Accumulators
  - Emulate greater precision
  - For example, double-double and triple-double

Choice of Summation Technique

- Performance
- Error Bound
    - Is it (weakly) dependent on n?
- Condition Number
    - Is it known?
    - Is it difficult to determine?
    - Some algorithms allow it to be determined simultaneously with an estimate of the sum
    - Permits easy evaluation of 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:

$$C_{dot\ \ product} = \frac{2\sum|a_i \cdot b_i|}{|\sum a_i \cdot b_i|}$$

- If C is not too large, a traditional algorithm can be used
- If C is large, more accurate methods are required

Dot Product

- The dot product of 2 n-dimensional vectors 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 with precision p is as accurate as through the dot product was computed in precision $2p$ and then rounded to p
- Consult the work of Ogita, Rump and Oishi

Polynomal Evaluation

- Horner’s method
- Use of EFTs
- Compensated

Polynomal Evaluation

Horner's method

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

where x and all a_i are floating-point numbers

Polynomial Evaluation by Horner's Method

```
double
Horner(const double* a,
       const unsigned int n,
       double x)
{
    double p = a[n];
    for (unsigned int i = n - 1; i >= 0; i--) {
        p = p * x + a[i];
    }
    return p;
}
```

Use of EFTs in Horner's Method

The Horner scheme is basically a combination of additions and multiplications.

```
    p = p * x + a[i];
```

Applying the EFTs for those operations is straight-forward if tedious

Use of EFTs in Compensated Horner's Method

Graillat, Langlois and Louvet have developed an algorithm using `TwoSum` and `Mult` which computes

$$p = \sum_{i=0}^{n} a_i x_i - \left(\sum_{i=0}^{n-1} \pi_i x_i + \sum_{i=0}^{n-1} \sigma_i x^i \right)$$

The first term is simply the result of a Horner's evaluation.
The algorithm can be shown to be an EFT for Horner's evaluation:

$$Horner = p + \left(\sum_{i=0}^{n-1} \pi_i x_i + \sum_{i=0}^{n-1} \sigma_i x^i \right)$$

Use of EFTs in Compensated Horner's Method

Analysis of the algorithm shows that the result is the same as that computed by a classical Horner's evaluation with twice the precision which is then rounded to the working precision.

Reference: *Compensated Horner's Scheme*, S. Graillat, Ph. Langlois, N. Louvet. Université de Perpignan Via Domitia research report RR2005-04.

Bibliography

- J.-M. Muller et al, *Handbook of Floating-Point Arithmetic*, Birkäuser, Boston, 2010.
- N.J. Higham, *Accuracy and Stability of Numerical Algorithms (2^{nd} Edition)*, SIAM, Philadelphia, 2002.
- Publications from CNRS/ENS Lyon/INRIA/AriC project (J.-M. Muller et al).
- Publications from Institut für Zuverlässiges Rechnen (Institute for Reliable Computing), Technische Universität Hamburg-Harburg (Siegfried Rump et al).

Questions