Distributed and Parallel Computing

Lecture 09

Alan P. Sexton

University of Birmingham

Spring 2017

Floating Point Number Representation

Single Precision IEEE-754 floating point numbers has:

- 1 sign bit (S)
- an 8 bit biased exponent field with a bias of 127 (E)
    - 0 and 255 reserved for special numbers
    - E=127 corresponds to an exponent of $E - 127 = 0$
- a 23 bit mantissa field (M)
- for normal numbers, an implicit (hidden) initial 1 bit in the mantissa is assumed (24 binary bits $\approx$ 7 decimal digits)
- a number of special bit patterns in S, E and M for positive and negative zeros, positive and negative infinities, NaN and sub-normal numbers

For normal numbers the interpretation is:

$$(-1)^S \left(1 + 2^{-23} M\right) 2^{E-127}$$

Subnormal Numbers

Normal numbers use the implicit 1 bit in the mantissa. What if only normal numbers are represented?

- The smallest representable number greater than 0 would be

$$\left(1 + 2^{-23} \times 0\right) 2^{-127} = 2^{-127} \approx 6 \times 10^{-39}$$

- The next smallest would be

$$\left(1 + 2^{-23} \times 1\right) 2^{-127} = 2^{-127} + 2^{-150}$$

- Thus much larger distance from 0 to succ(0) than from succ(0) to succ(succ(0))
- Sub-normal numbers: if exponent is -127 (i.e. E=0), don't use implicit 1 in mantissa

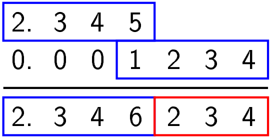
Rounding

- Unit in Last Place ($ULP(x)$): distance between the two floating point numbers that are the closest pair (a, b) that straddle x: $a \leq x \leq b$
- IEEE-754 requires that operations round to nearest representable floating point number: that the computed result be within 0.5 ULPs of the mathematically correct result.
- Consider $a + b$, where $a \gg b$, e.g. in a 4 digit decimal notation:

$$\begin{array}{l} \boxed{2.\;\;3\;\;4\;\;5} \\ 0.\;\;0\;\;0\;\boxed{1\;\;2\;\;3\;\;4} \\ \hline \end{array}$$

Rounding

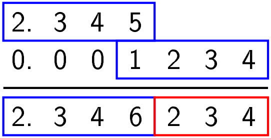
- Unit in Last Place ($ULP(x)$): distance between the two floating point numbers that are the closest pair (a, b) that straddle x: $a \leq x \leq b$
- IEEE-754 requires that operations round to nearest representable floating point number: that the computed result be within 0.5 ULPs of the mathematically correct result.
- Consider $a + b$, where $a \gg b$, e.g. in a 4 digit decimal notation:



$$\begin{array}{ccccccc} 2. & 3 & 4 & 5 & & & \\ 0. & 0 & 0 & 1 & 2 & 3 & 4 \\ \hline 2. & 3 & 4 & 6 & 2 & 3 & 4 \end{array}$$

Rounding

- Unit in Last Place (*ULP(x)*): distance between the two floating point numbers that are the closest pair (a, b) that straddle x: $a \leq x \leq b$
- IEEE-754 requires that operations round to nearest representable floating point number: that the computed result be within 0.5 ULPs of the mathematically correct result.
- Consider $a + b$, where $a \gg b$, e.g. in a 4 digit decimal notation:



2. 3 4 5

0. 0 0 1 2 3 4

2. 3 4 6 2 3 4

- Thus, in this system, $(2.345 + 0.001234) - 2.345 = 0.001$

Sequence of Additions

Again in 4 digit decimal notation, consider summing a vector containing 1000.0 in the first element and then 10,000 elements of value 0.1.

- Mathematically, result should be 2000.0
- Incrementally adding from first returns 1000.0

$$\begin{array}{rl}
\boxed{1\quad 0\quad 0\quad 0.} & 0 \\
0. & \boxed{1\quad 0\quad 0\quad 0} \\
\hline
\boxed{1\quad 0\quad 0\quad 0.} & 0 \\
0. & \boxed{1\quad 0\quad 0\quad 0} \\
\hline
\vdots &
\end{array}$$

Worst case error in summing N numbers: $O(N)$

Fixing Sequence of Additions

To fix this, we can try adding in increasing order:

- Sort vector first
- Now incremental additions should add the smaller values together first then combine accumulated larger values with larger values from later in the vector

Problem:

- Accumulated small values may get significantly larger than later values in the vector
- Thus may help in some cases, but not a full solution

Kahan Sumation

Idea: Accumulate the sum, but calculate a correction term and add it to the next number at each step:

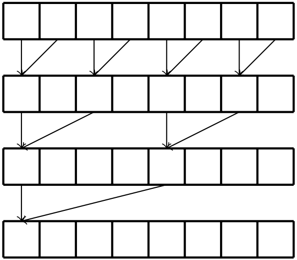
```
float fk_add(float * flt_arr)
{
  long i;
  float sum, correction, corrected_next_term, new_sum;

  sum = flt_arr[0];
  correction = 0.0;
  for (i = 1; i < ARR_SIZE; i++)
  {
    corrected_next_term = flt_arr[i] - correction;
    new_sum = sum + corrected_next_term;
    correction = (new_sum - sum) - corrected_next_term;
    sum = new_sum;
  }
  return sum;
}
```

- Worst case error in summing N numbers: $O(1)$, i.e. dependent only on the precision of the number representation
- But, not easy to parallelise

Fixing Sequence of Additions

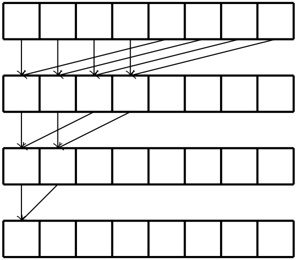
Try sorting, then reducing



- adds similarly sized pairs in early steps
- but later steps add increasingly different sized values
- Worst case error in summing N numbers: O(log(N))
- But: poor thread blocking, memory access patterns

Fixing Sequence of Additions

Can use more efficient compressed thread reduction pattern:



- but this adds values widely separated in vector
- Careful ordering helps
- Possible to use reduction for first few steps and switch to (unrolled) Kahan Summation to finish off

Absolute and Relative Errors

Given a value v, and its computed approximation $\hat{v}$:

- The *absolute error* in $\hat{v}$ is $|v - \hat{v}|$
- The *relative error*, where $v \neq 0$, in $\hat{v}$ is $\frac{|v-\hat{v}|}{|v|}$

The relative error is usually the more useful quantity.

- $|a| \gg |b| \Rightarrow a + b$ has a large absolute error
- $|b| \ll 1 \Rightarrow \frac{a}{b}$ has large relative and absolute errors
- $a \approx b \Rightarrow a - b$ has a large relative error (cancellation errors)

Example

We can compute $e = 2.7182818\ldots$, the base of natural logarithms, with the formula:

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

n	val
10^1	2.593742
10^2	2.704814
10^3	2.716924
10^4	2.718146
10^5	2.718268
10^6	2.718281
10^7	2.718282
10^8	2.718282
10^9	2.718282
10^{10}	2.718282
10^{11}	2.718282
10^{12}	2.718524
10^{13}	2.716110
10^{14}	2.716110
10^{15}	3.035035
10^{16}	1.000000
10^{17}	1.000000

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\frac{1}{\sqrt{x^2+1}-x} = \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\begin{aligned}\frac{1}{\sqrt{x^2+1}-x} &= \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)} \\ &= \frac{\sqrt{x^2+1}+x}{x^2+1-x^2}\end{aligned}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\begin{aligned}\frac{1}{\sqrt{x^2+1}-x} &= \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)}\\ &= \frac{\sqrt{x^2+1}+x}{x^2+1-x^2}\\ &= \sqrt{x^2+1}+x\end{aligned}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\begin{aligned}\frac{1}{\sqrt{x^2+1}-x} &= \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)}\\ &= \frac{\sqrt{x^2+1}+x}{x^2+1-x^2}\\ &= \sqrt{x^2+1}+x\end{aligned}$$

Now tiny roundoff error

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\cos^2 x - \sin^2 x}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\cos^2 x - \sin^2 x}$$

Normally fine when denominator is not 0, but when x is near $\pi/4$, we get cancellation error

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\cos^2 x - \sin^2 x}$$

Normally fine when denominator is not 0, but when x is near $\pi/4$, we get cancellation error

$$\frac{1}{\cos^2 x - \sin^2 x} = \frac{1}{\cos 2x}$$

Now no problem where denominator is close to, but not 0