Beating Floating Point at its Own Game: Posit Arithmetic

John L. Gustafson
Professor, A*STAR and National University of Singapore

Supercomputing Frontiers 2017
Why worry about floating point?

Find the scalar product $a \cdot b$:

\[
\begin{align*}
a &= (3.2e8, 1, -1, 8.0e7) \\
b &= (4.0e7, 1, -1, -1.6e8)
\end{align*}
\]

Note: All values are integers that can be expressed exactly in the IEEE 754 Standard floating-point format (single or double precision)

Single Precision, 32 bits: $a \cdot b = 0$
Double Precision, 64 bits: $a \cdot b = 0$

Correct answer: $a \cdot b = 2$

Most linear algebra is unstable with floats!
What’s wrong with IEEE 754? A start:

- No guarantee of identical results across systems
- It’s a guideline, not a standard
- Breaks the laws of algebra:
  \[ a + (b + c) \neq (a + b) + c \quad a \cdot (b + c) \neq a \cdot b + a \cdot c \]
- Overflow to infinity, or underflow to zero, create an infinite loss of accuracy.

IEEE floats are weapons of math destruction.
What else is wrong with IEEE 754?

• Exponents usually take too many bits
• Accuracy is flat across a vast range, then falls off a cliff
• Subnormal numbers are a headache ("gradual underflow")
• Divides are messy and slow
• Wasted bit patterns: “negative zero,” too many NaN values

Do we really need 9,007,199,254,740,990 ways to say something is *Not a Number*??
IEEE Standard (1985)  
Type 1 Unums (2013)  
Type 2 Unums (2016)  
Type 3 Unums (2017)

Rounded: cheap, uncertain, “good enough”  
Floats, $f = n \times 2^m$  
$m, n$ are integers  
“Guess” mode, flexible word size  

Rigorous: more work, certain, mathematical  
Intervals $[f_1, f_2]$, all $x$ such that $f_1 \leq x \leq f_2$  
Unums, ubounds, sets of uboxes  
Sets of Real Numbers (SORNs)

Posits  
Valid

If you mix the two esthetics, you end up satisfying neither.
noun  Philosophy

a statement that is made on the assumption that it will prove to be true.
Metrics for Number Systems

- Decimals of Accuracy: \(-\log_{10}(\log_{10}(x_j / x_{j+1}))\)
- Dynamic range: \(\log_{10}(\text{maxreal} / \text{minreal})\)
- Percentage of operations that are exact (closure under + – × ÷ √ etc.)
- Average accuracy loss when inexact
- Entropy per bit (maximize information)
- Accuracy benchmarks: simple formulas, linear equation solving, math kernels…
Posit Arithmetic: Beating floats at their own game

Fixed size, \( n\text{bits} \).

Note: The “ubit” is only for the \textit{valid} type; posits round, if necessary.

\( es = \text{exponent size} = 0, 1, 2, \ldots \text{ bits} \).
Posit Arithmetic Example

Here, $e_s = 3$. Float-like circuitry is all that is needed (integer add, integer multiply, shifts to scale by $2^k$)

Posits do not underflow or overflow. There is no NaN.

Simpler, smaller, faster circuits than IEEE 754
Posits use the Projective Reals

• Like Type 2 unums, posits map reals to standard signed integers.
• This eliminates “negative zero” and other IEEE float issues
Example with $nbits = 3$, $es = 1$.

Value at 45° is always

$$useed = 2^{es}$$

If bit string < 0, set sign to – and negate integer.
Rules for inserting new points

Between $\pm \text{maxpos}$ and $\pm \infty$, scale up by $\text{useed}$. (New regime bit)

Between 0 and $\pm \text{minpos}$, scale down by $\text{useed}$. (New regime bit)

Between $2^m$ and $2^n$ where $n - m \geq 2$, insert $2^{(m + n)/2}$. (New exponent bit)
At \( nbins = 5 \), fraction bits appear.

Between \( x \) and \( y \) where \( y \leq 2x \), insert \((x + y)/2\).

Existing values stay put as *trailing* bits are added.

Appending bits increases *accuracy* east and west, *dynamic range* north and south!
Posits v. Floats: a *metrics-based* study

• Compare *quarter-precision* IEEE-style floats

• Sign bit, 4 exponent bits, 3 fraction bits

• smallsubnormal = 1/512; maxfloat = 240.

• Dynamic range of five orders of magnitude

• Two bit patterns that mean zero

• Fourteen bit patterns that mean “Not a Number” (NaN)
Float accuracy tapers only on left

- Min: 0.52 decimals
- Avg: 1.40 decimals
- Max: 1.55 decimals

Graph shows decimals of accuracy from minfloat to maxfloat.
Posit accuracy tapers on both sides

- Min: 0.22 decimals
- Avg: 1.46 decimals
- Max: 1.86 decimals

Graph shows decimals of accuracy from minpos to maxpos. But posits cover seven orders of magnitude, not five.
Both graphs at once

Decimal accuracy

Where most calculations occur

Posits

 Floats

Set elements
### Matching float dynamic ranges

<table>
<thead>
<tr>
<th>Size, bits</th>
<th>Float exponent size</th>
<th>Float dynamic range</th>
<th>Posit es value</th>
<th>Posit dynamic range</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5</td>
<td>$6 \times 10^{-8}$ to $7 \times 10^{4}$</td>
<td>1</td>
<td>$4 \times 10^{-9}$ to $3 \times 10^{8}$</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>$1 \times 10^{-45}$ to $3 \times 10^{38}$</td>
<td>3</td>
<td>$6 \times 10^{-73}$ to $2 \times 10^{72}$</td>
</tr>
<tr>
<td>64</td>
<td>11</td>
<td>$5 \times 10^{-324}$ to $2 \times 10^{308}$</td>
<td>4</td>
<td>$2 \times 10^{-299}$ to $4 \times 10^{298}$</td>
</tr>
<tr>
<td>128</td>
<td>15</td>
<td>$6 \times 10^{-4966}$ to $1 \times 10^{4932}$</td>
<td>7</td>
<td>$1 \times 10^{-4855}$ to $1 \times 10^{4855}$</td>
</tr>
<tr>
<td>256</td>
<td>19</td>
<td>$2 \times 10^{-78984}$ to $2 \times 10^{78913}$</td>
<td>10</td>
<td>$2 \times 10^{-78297}$ to $5 \times 10^{78296}$</td>
</tr>
</tbody>
</table>

**Note:** Isaac Yonemoto has shown that 8-bit posits suffice for neural networks, with es = 0
8-bit posits speed *neural nets*

Sigmoid functions take 1 cycle in posits, vs. dozens of cycles with float math libraries.

(Observation by I. Yonemoto)
ROUND 1

Unary Operations

$1/x, \sqrt{x}, x^2, \log_2(x), 2^x$
Closure under Reciprocation, $1/x$
Closure under Square Root, $\sqrt{x}$

- **Floats**
  - 7.03% exact
  - 40.6% inexact
  - 52.3% NaN

- **Posits**
  - 8.20% exact
  - 42.2% inexact
  - 49.6% NaN
Closure under Squaring, $x^2$

Decimal loss per calculation

<table>
<thead>
<tr>
<th></th>
<th>Floats</th>
<th>Posits</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.3% exact</td>
<td></td>
<td>15.2% exact</td>
</tr>
<tr>
<td>43.8% inexact</td>
<td></td>
<td>84.8% inexact</td>
</tr>
<tr>
<td>5.47% NaN</td>
<td></td>
<td>0% NaN</td>
</tr>
<tr>
<td>25.0% overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.5% underflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All losses, sorted
Closure under $\log_2(x)$

- **Floats**
  - 7.81% exact
  - 39.8% inexact
  - 52.3% NaN

- **Posits**
  - 8.98% exact
  - 40.6% inexact
  - 50.4% NaN

The graph compares the decimal loss per calculation between Floats and Posits, showing a higher percentage of inexact results and NaN values for Floats compared to Posits.
Closure under $2^x$

Decimal loss per calculation

<table>
<thead>
<tr>
<th></th>
<th>Floats</th>
<th>Posits</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.81% exact</td>
<td>8.98% exact</td>
<td></td>
</tr>
<tr>
<td>56.3% inexact</td>
<td>90.6% inexact</td>
<td></td>
</tr>
<tr>
<td>5.47% NaN</td>
<td>0.391% NaN</td>
<td></td>
</tr>
<tr>
<td>15.6% overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.8% underflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All losses, sorted

0.000 0.005 0.010 0.015 0.020 0.025 0.030
0 50 100 150 200

1233

Floths

Posits
ROUND 2

Two-Argument Operations

\[ x + y, \; x \times y, \; x \div y \]
Addition Closure Plot: Floats

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>18.533%</td>
<td>exact</td>
</tr>
<tr>
<td>70.190%</td>
<td>inexact</td>
</tr>
<tr>
<td>0.000%</td>
<td>underflow</td>
</tr>
<tr>
<td>0.635%</td>
<td>overflow</td>
</tr>
<tr>
<td>10.641%</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Inexact results are magenta; the larger the error, the brighter the color.

Addition can **overflow**, but cannot **underflow**.
Addition Closure Plot: Posits

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25.005%</td>
<td>exact</td>
</tr>
<tr>
<td>74.994%</td>
<td>inexact</td>
</tr>
<tr>
<td>0.000%</td>
<td>underflow</td>
</tr>
<tr>
<td>0.000%</td>
<td>overflow</td>
</tr>
<tr>
<td>0.002%</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Only one case is a NaN:

$$\pm \infty + \pm \infty$$

With posits, a NaN always stops the calculation. (Valids handle NaNs as sets.)
All decimal losses, sorted

Decimal loss per calculation

- Floats
  - 18.5% exact
  - 70.2% inexact
  - 10.6% NaN
  - 0.635% overflow
  - 0% underflow

- Posits
  - 25.0% exact
  - 75.0% inexact
  - 0.00153% NaN

All decimal losses, sorted
Multiplication Closure Plot: Floats

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.272%</td>
<td>exact</td>
</tr>
<tr>
<td>58.279%</td>
<td>inexact</td>
</tr>
<tr>
<td>2.475%</td>
<td>underflow</td>
</tr>
<tr>
<td>6.323%</td>
<td>overflow</td>
</tr>
<tr>
<td>10.651%</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Floats score their first win: more exact products than posits…

but at a **terrible cost**!
### Multiplication Closure Plot: Posits

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>18.002%</td>
<td>exact</td>
</tr>
<tr>
<td>81.995%</td>
<td>inexact</td>
</tr>
<tr>
<td>0.000%</td>
<td>underflow</td>
</tr>
<tr>
<td>0.000%</td>
<td>overflow</td>
</tr>
<tr>
<td>0.003%</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Only two cases produce a NaN:

\[ \pm \infty \times 0 \]
\[ 0 \times \pm \infty \]
The sorted losses tell the real story

<table>
<thead>
<tr>
<th>Decimals</th>
<th>Losses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floats</td>
<td>22.3% exact, 51.2% inexact, 10.7% NaN, 12.5% overflow, 3.34% underflow</td>
</tr>
<tr>
<td>Posits</td>
<td>18.0% exact, 82.0% inexact, 0.00305% NaN</td>
</tr>
</tbody>
</table>

All decimal losses, sorted
Division Closure Plot: Floats

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>22.272%</td>
<td>exact</td>
</tr>
<tr>
<td>58.810%</td>
<td>inexact</td>
</tr>
<tr>
<td>3.433%</td>
<td>underflow</td>
</tr>
<tr>
<td>4.834%</td>
<td>overflow</td>
</tr>
<tr>
<td>10.651%</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Like multiplication, but with less symmetry.
Division Closure Plot: Posits

<table>
<thead>
<tr>
<th></th>
<th>exact</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>18.002%</td>
<td>exact</td>
<td>81.995%</td>
<td>inexact</td>
<td>0.000%</td>
<td>underflow</td>
<td>0.000%</td>
<td>overflow</td>
<td>0.003%</td>
</tr>
</tbody>
</table>

Posits do not have denormalized values. Nor do they need them.

Hidden bit = 1, always. Simplifies hardware.
ROUND 3

Higher-Precision Operations

32-bit formula evaluation
128-bit triangle area calculation
LINPACK solved with… 16 bits!
# Accuracy on a 32-Bit Budget

Compute: \[
\left( \frac{27/10 - e}{\sqrt{2} + \sqrt{3}} \right)^{67/16} \approx 302.8827196
\]

with \( \leq 32 \) bits per number.

<table>
<thead>
<tr>
<th>Number Type</th>
<th>Dynamic Range</th>
<th>Answer</th>
<th>Error or Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE 32-bit float</td>
<td>( 2 \times 10^{33} )</td>
<td>( 302.912\ldots )</td>
<td>( 0.0297\ldots )</td>
</tr>
<tr>
<td>Interval arithmetic</td>
<td>( 10^{12} )</td>
<td>[18.21875, 33056.]</td>
<td>( 3.3\ldots \times 10^4 )</td>
</tr>
<tr>
<td>Type 1 unums</td>
<td>( 4 \times 10^{33} )</td>
<td>(302.75, 303.)</td>
<td>( 0.25 )</td>
</tr>
<tr>
<td>Type 2 unums</td>
<td>( 10^{99} )</td>
<td>( 302.887\ldots )</td>
<td>( 0.0038\ldots )</td>
</tr>
<tr>
<td>Posits, ( es = 3 )</td>
<td>( 3 \times 10^{144} )</td>
<td>( 302.88231\ldots )</td>
<td>( 0.00040\ldots )</td>
</tr>
<tr>
<td>Posits, ( es = 1 )</td>
<td>( 10^{36} )</td>
<td>( 302.8827819\ldots )</td>
<td>( 0.000062\ldots )</td>
</tr>
</tbody>
</table>

Posits beat floats at both dynamic range and accuracy.
Thin Triangle Area

Find the area of this thin triangle

\[ b = \frac{7}{2} + 3 \times 2^{-111} \]
\[ a = 7 \]
\[ c = \frac{7}{2} + 3 \times 2^{-111} \]

using the formula

\[
s = \frac{a+b+c}{2}; \quad A = \sqrt{s(s-a)(s-b)(s-c)}
\]

and 128-bit IEEE floats, then 128-bit posits.

Answer, correct to 36 decimals:

\[ 3.14784204874900425235885265494550774 \ldots \times 10^{-16} \]

From “What Every Computer Scientist Should Know About Floating-Point Arithmetic,”
A Grossly Unfair Contest

IEEE quad-precision floats get only *one digit* right:

3.63481490842332134725920516158057683⋯×10^{−16}

128-bit posits get **36 digits** right:

3.14784204874900425235885265494550774⋯×10^{−16}

To get this accurate an answer with IEEE floats, you need the *octuple* precision (256-bit) format.

Posits don’t even need 128 bits. They can get a very accurate answer with only 119 bits.
LINPACK: \( Ax = b \)

16-bit posits versus 16-bit floats

- 100 by 100; random \( A_{ij} \) entries in \((0, 1)\)
- \( b \) chosen so \( x \) should be all 1s exactly

**IEEE 16-bit Floats**
- Dynamic range: \( 10^{12} \)
- Maximum error: 0.011
- Decimal accuracy: 1.96

**16-bit Posits**
- Dynamic range: \( 10^{16} \)
- Maximum error: NONE
- Decimal accuracy: \( \infty \)

Note: work funded in part by DARPA under contract BAA 16-39
# 64-bit Float versus 16-bit posit

## 64-bit IEEE Floats

<table>
<thead>
<tr>
<th>Float</th>
<th>124344978758017532527446746826171875</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.999999999999</td>
<td>837907438404727145098149776458740234375</td>
</tr>
<tr>
<td>1.000000000000</td>
<td>193178806284777238033711910247802734375</td>
</tr>
<tr>
<td>0.999999999999</td>
<td>8501198916756038670428097248077392578125</td>
</tr>
<tr>
<td>0.999999999999</td>
<td>11182158029987476766109466552734375</td>
</tr>
<tr>
<td>0.999999999999</td>
<td>00079927783735911361873149871826171875</td>
</tr>
</tbody>
</table>

## 16-bit Posits

<table>
<thead>
<tr>
<th>Posit</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Remember this from the beginning?

Find the scalar product \( a \cdot b \):

\[
\begin{align*}
a &= (3.2\times10^8, 1, -1, 8.0\times10^7) \\
b &= (4.0\times10^7, 1, -1, -1.6\times10^8)
\end{align*}
\]

Posit answer: \( a \cdot b = 2 \) (correct)

IEEE floats require 128-bit precision to get it right. Posits (es = 3) need only 25-bit precision to get it right. **Fused dot product** is 3–6 times faster than the float method.  

* “Hardware Accelerator for Exact Dot Product,”
  David Biancolin and Jack Koenig, ASPIRE Laboratory, UC Berkeley
Type 1, 2 unum hardware challenges…

- Type 1 unums need *variable size*; require unpacked form for simple indexing
- Type 2 unums need *table look-up*; only scale to about 20 bits
- But: Posits and Valids are ready to go *now*!

—I think you should be more explicit here in step two."
Building posit chips: The race is on

- Like IEEE floats, but *simpler* and *less area* (!)
- Multiplier, adder designs are done
- REX Computing, and a handful of startups are working on it; Intel is showing interest
- Looks ideal for GPUs; more arithmetic per chip
Posit pairs beat *intervals* at their own game, too: *Valid* mode

“**Posit**” mode: Round unum if operation yields an inexact.

“**Valid**” mode: Rigorous bounds; “NaN” answers are sets
32-bit precision may suffice now!

- Early computers used 36-bit floats.
- IBM System 360 went to 32-bit.
- It wasn’t quite enough.
- What if 32-bit posits could replace 64-bit floats for structural analysis, circuit simulation, etc.? 
- Potential 2x shortcut to exascale. Or more.
Summary

• Posits beat floats at their own game: superior accuracy, dynamic range, closure
• Bitwise-reproducible answers (at last!)
• Proven better answers with same number of bits
• …or, equally good answers with fewer bits
• Simpler, more elegant design can reduce silicon cost, energy, and latency.

Who will produce the first posit arithmetic chip?