Accuracy-Aware Development Environment for High-Efficiency Math

Dr. John Gustafson

Director

Intel Labs

Integers → Fixed-point → Floating-point → Accuracy-aware

Terminology reminders

- *Precision* = Digits available to store a number ("32-bit" or "4 decimal", for example)
- *Accuracy* = Number of **valid** digits in a result ("to three significant digits", for example)
- ULP = Unit of Least Precision.

Precision is not a goal.
Precision is the means, accuracy is the en

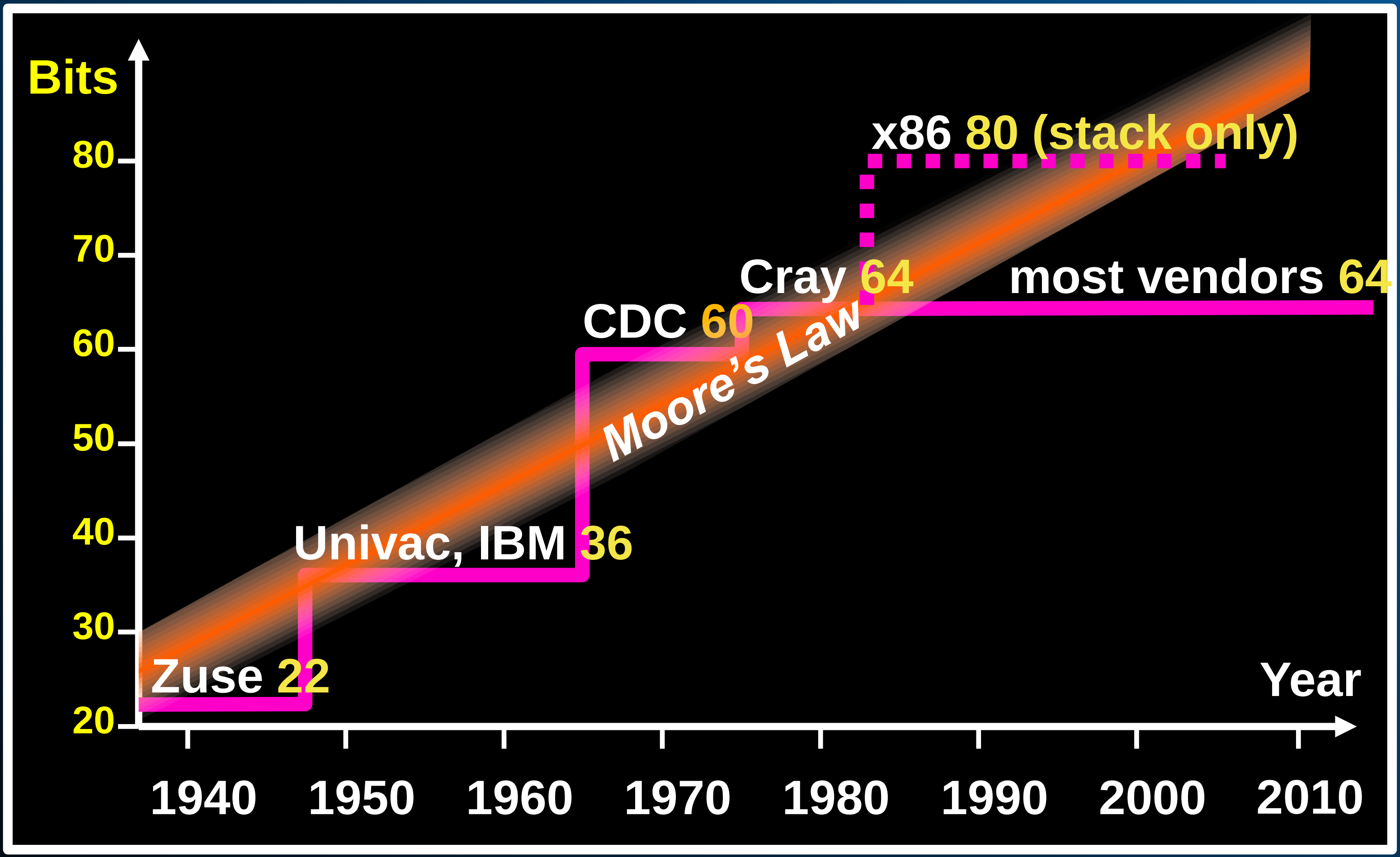
Intel Labs

The Problem

- Current floating-point math wastes energy, power, time, and storage, by using *worst-case precision* everywhere.
- Widespread issue, beyond just HPC; precision excess is prevalent in search, cloud, games, graphics, financial calculations, speech recognition…
- FP is hard to use because programming bugs and rounding errors look alike!
- Constraint: We *must* continue to support the IEEE standard.

Using 64-bit everywhere is *speculation*

Is it enough? Is it too much? We're *guessing*.



Intel Labs

Benefits of efficient math reach across a wide range of applications

Angry Birds (box3d games)

Google's Page Rank

Source: Andrew Kormornicki, IBM

Financial Sector

Black Scholes Pricing Formula

$$c = S\,N(d_1) - Xe^{-r(T-t)}\,N(d_2)$$

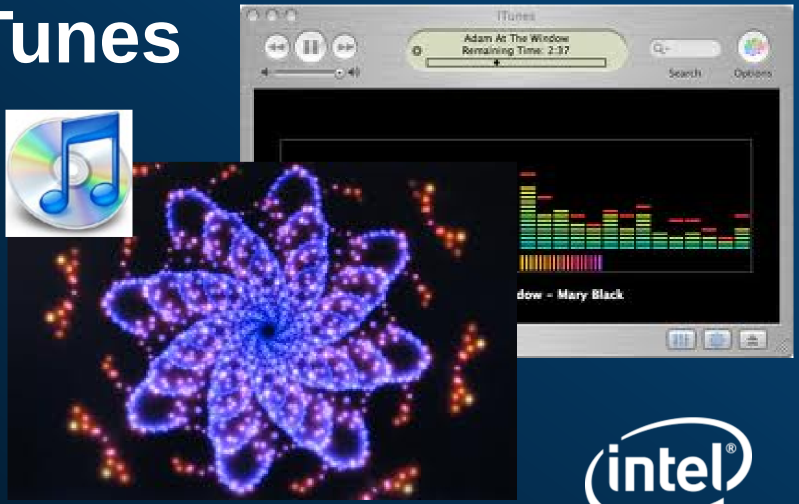
where

$$d_1 = \frac{\ln(S/X) + (r \cdot s^2/2)(T-t)}{s\sqrt{T-t}}$$

$$d_2 = d_1 - s\sqrt{T-t}$$

c = call option price
S = current stock price
X = exercise price
$e^{-r(T-t)}$ = continuously compounded risk free rate
s = standard deviation of stock price returns
T = maturity date
t = date option is being valued
N(x) = cumulative probability distribution function for a standardized normal variable

iTunes



Intel Labs

The Opportunity

- We can reduce bandwidth requirements by enabling **safe use of reduced precision**.
- Tools can help guarantee accurate calculations involving real numbers; make computers “self aware” of accuracy
- We can maintain the IEEE Standard legacy, but right-size the precision we use.

Reduce power, get better answers, and improve performance, *all at once*.

Intel Labs

When you don’t know accuracy (1)…

Sleipner Oil Rig Collapse. Loss: **$700 million.**

See http://www.ima.umn.edu/~arnold/disasters/sleipner.html

Intel Labs

When you don’t know accuracy (2)…

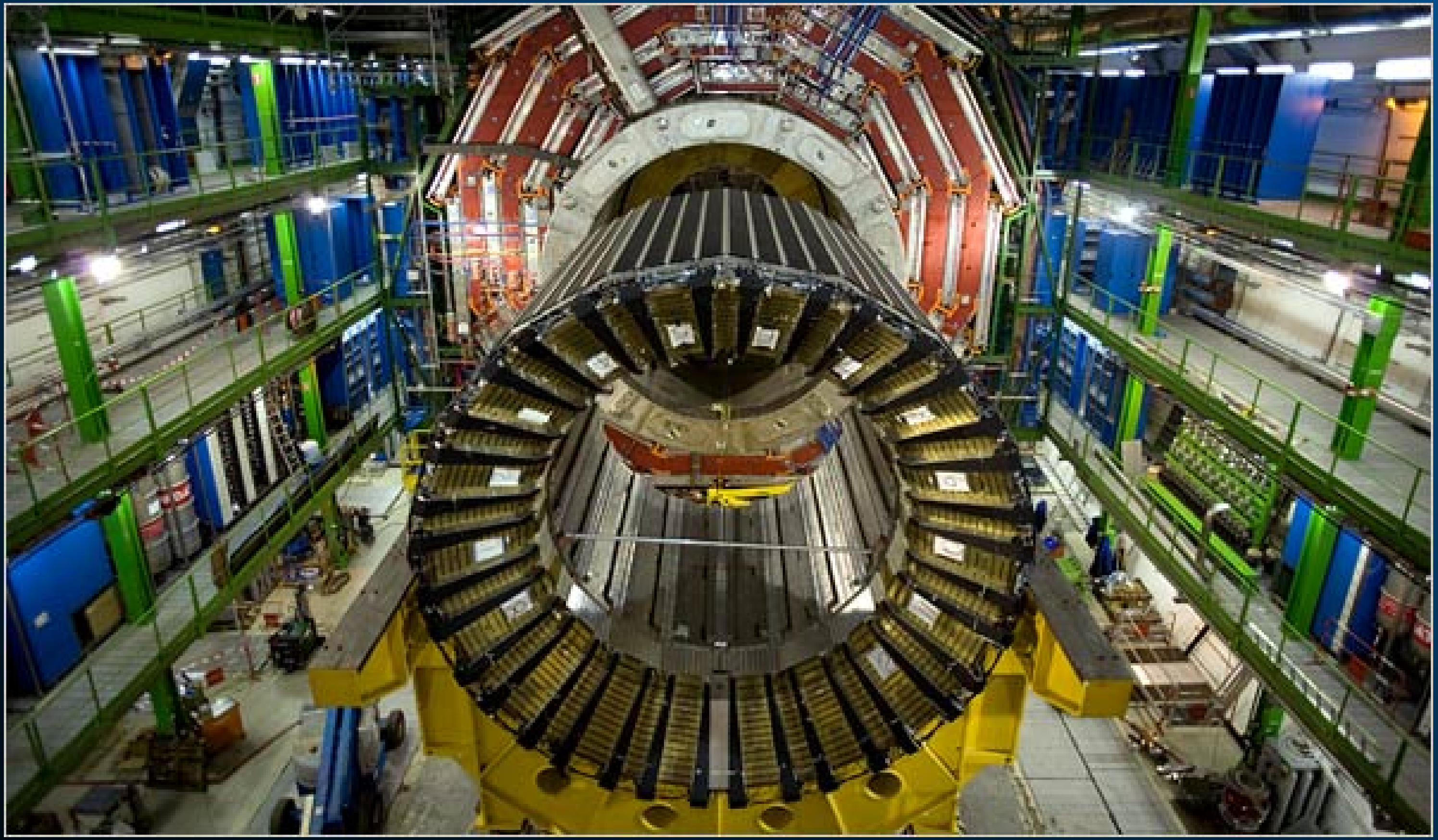
Vancouver stock exchange index undervalued by 50%

See http://ta.twi.tudelft.nl/usersvuik/wi211/disasters.html

intel
Intel Labs

When you don't know accuracy (3)…

2011: CERN faces need for 1.9x more memory



Source of data: Andrezej Nowak, CERN

Intel Labs

…and inaccuracy can *really* hurt

Patriot missile accident **killed 28 Americans.**

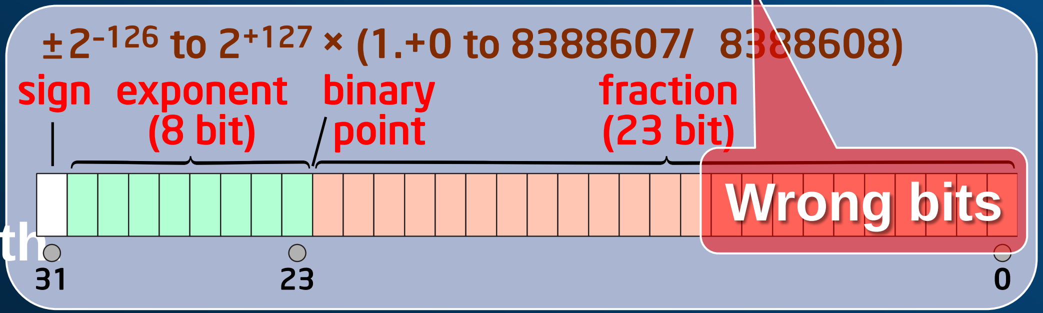
See http://www.fas.org/spp/starwars/gao/im92026.htm

Quick Tutorial on Rounding Error

"0.1 second" in single precision isn't exact.
It's 0.100000002384185791015625, rounded. Every addition will be a little bit off.

Accumulating seconds, 0.1 at a time, for 100 hours, will be off by at least *three minutes*!

As other speakers have said:
(a + b) + c is NOT the same as a + (b + c) in floating-point math



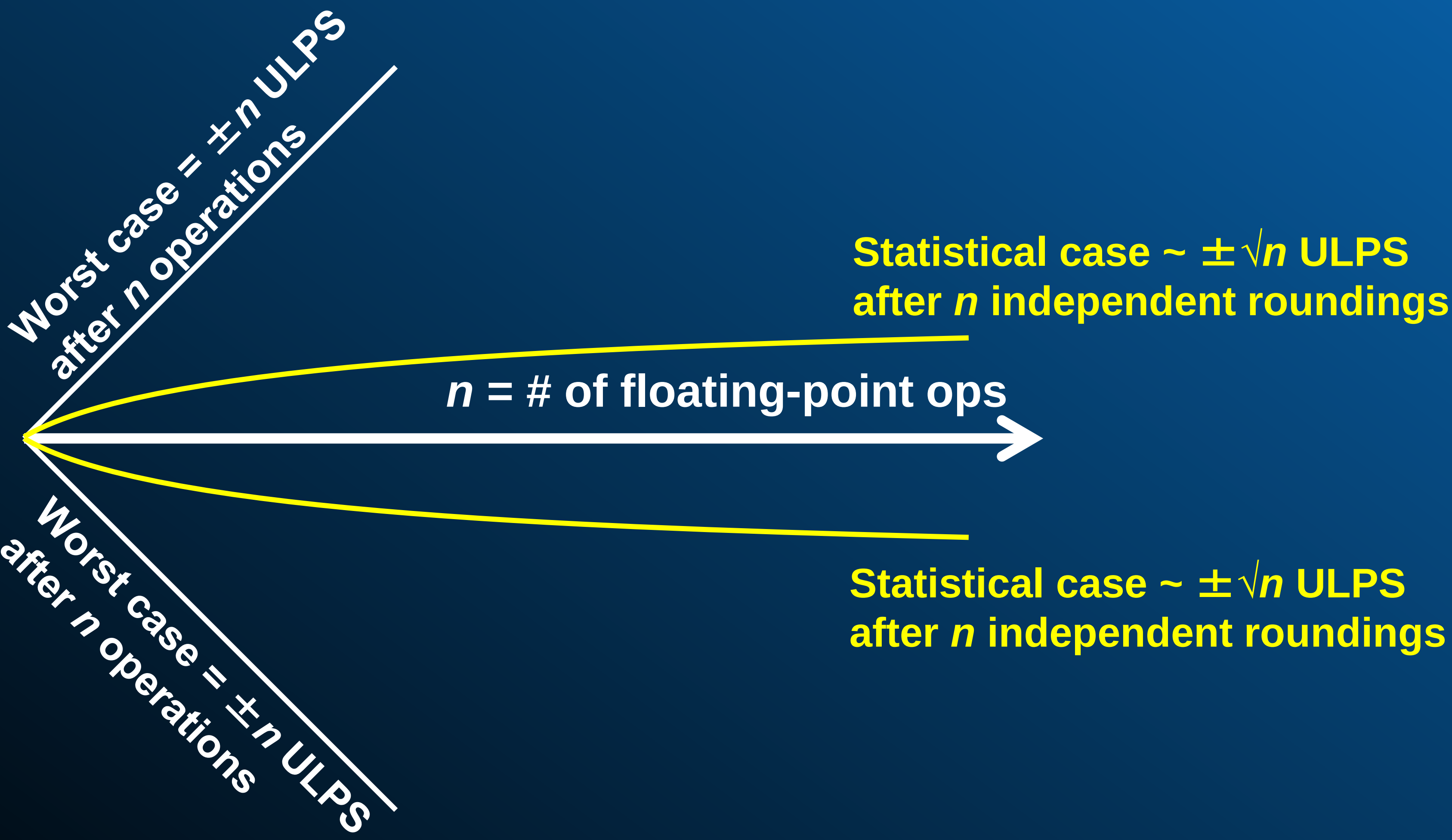
a = 1.0
b = 100000000.
c = –100000000.

(a + b) rounds down to = 100000000. Add c, get 0.0.
(b + c) = 0 exactly, with no rounding. Add a, get 1.0.

So floating point math fails algebra.
Big headache for parallel programming. Bug, or rounding error?
Accuracy-awareness solves this problem.

Intel Labs

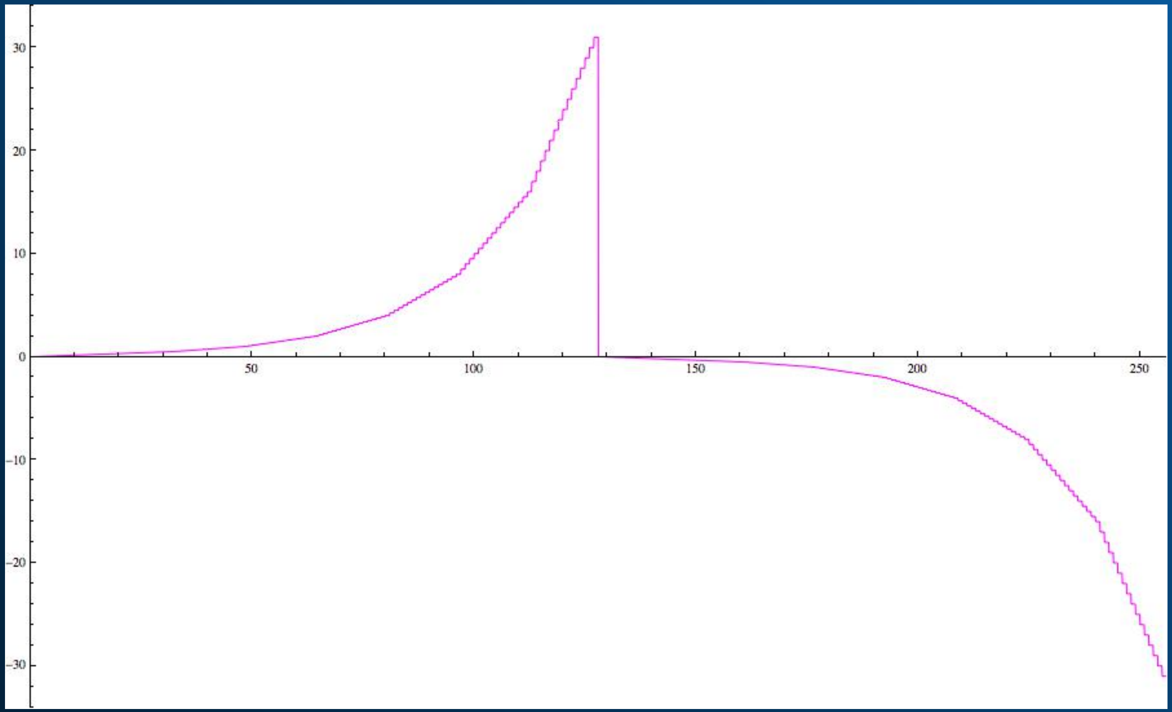
"Unbiased rounding" won't save you



- Rounding biases are *not* always statistically independent!
- And at petaflops/sec, "creeping crud" accumulates *fast.*

Intel Labs

IEEE-style floating point at any precision (via Mathematica)



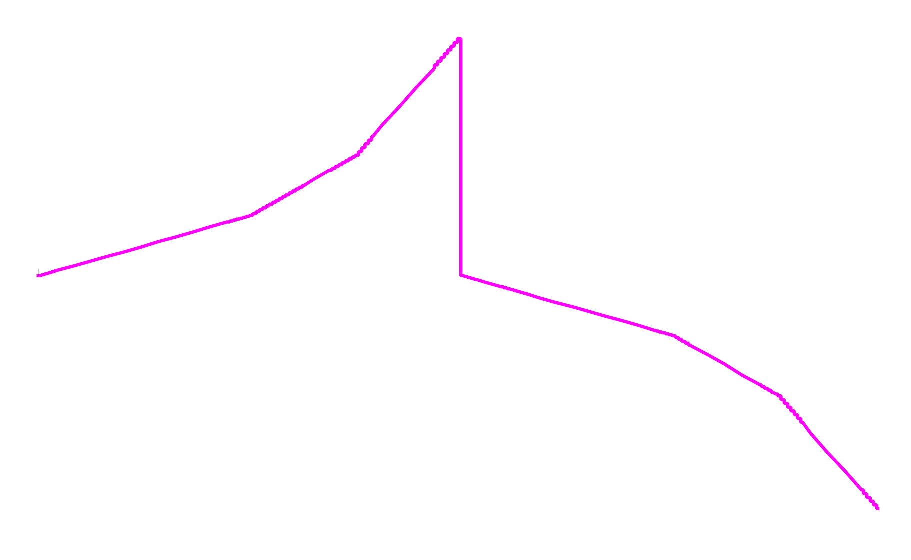

```
mbits = 8;
imax = 2^mbits - 1;
expobits = 3;
fracbits = mbits - 1 - expobits;
hidden = 2^fracbits;
signmask = 2^(mbits - 1);
fracmask = hidden - 1;
expomask = (signmask - 1) - fracmask;
sign[b_] := BitAnd[dbt, signmask]/signmask;
expo[b_] := BitAnd[dbt, expomask]/hidden;
frac[b_] := BitAnd[dbt, fracmask];
bias = 2^(expobits - 1) - 1;

ftox[b_] := N[(-1)^sign[b] Which[
   expo[b] == 0, frac[b]*2^(1 - bias - fracbits),
   expo[b] > 0, 2^(expo[b] - bias) (1 + frac[b]/hidden)]]
```

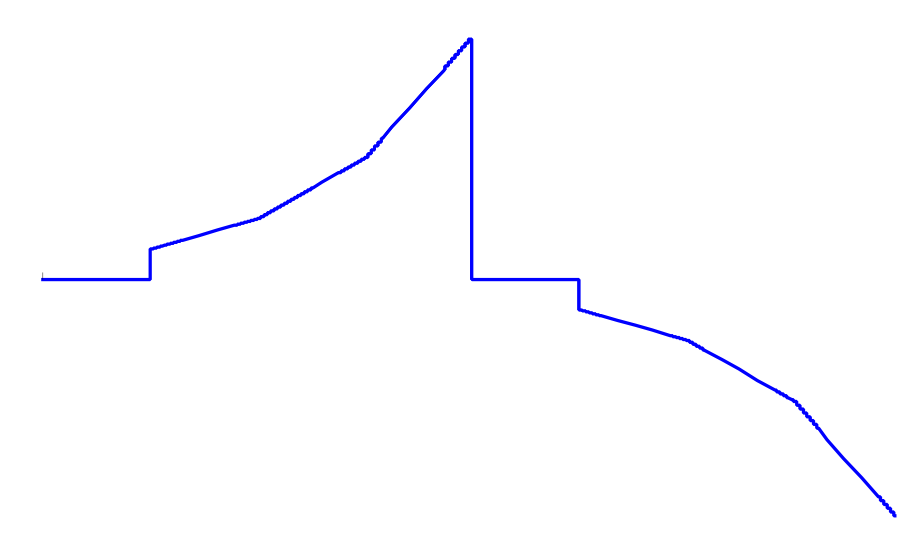
Intel Labs

Denormalized floats aren’t ‘weird’

What’s weird is *not* using them.
Using 8-bit floats to illustrate:



With denormalized numbers.
“Gradual underflow”



Easier hardware? Not really!
Clip to zero.
GPUs do this. Why?

Intel Labs

Sources of numerical inaccuracy

- Machine-caused errors
  - Cumulative rounding (“creeping crud”)
  - Left-digit destruction (subtracting similar numbers)
  - Operations on values of very different magnitudes (catastrophic accuracy destruction; like,$10^{16} + 3.14$ gives 10^{16}.)
  - I/O; conversion of decimal to binary numbers and back
- Programmer-caused errors
  - Naïve algorithms
  - Poor guarding of user input, e.g. sin(x) allowing $x = 10^{+300}$
- Nature-caused errors (soft errors)

Intel Labs

Miscellaneous Principles for Roundoff Control

- Don't differentiate numerically. Find an integral formulation of the problem!
- Use high-precision accumulators; this may allow reduced-precision data stored in DRAM
- If you need bitwise reproducibility for debugging, use parallel random number generators, and binary sum collapse even when summing on a single processor
- Lean on high-quality library routines, even for simple things like dot products, instead of writing your own (side effect: improves maintainability of code)

"How do you know your answer is correct?"

"(Laughter) "What do you mean?"*(This is the most common response)*

"We used double precision."

"It's the same answer we've always gotten."

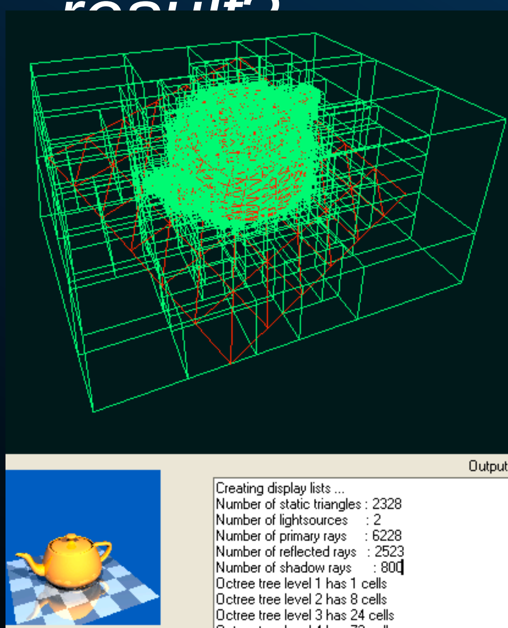
"It's the same answer others get."

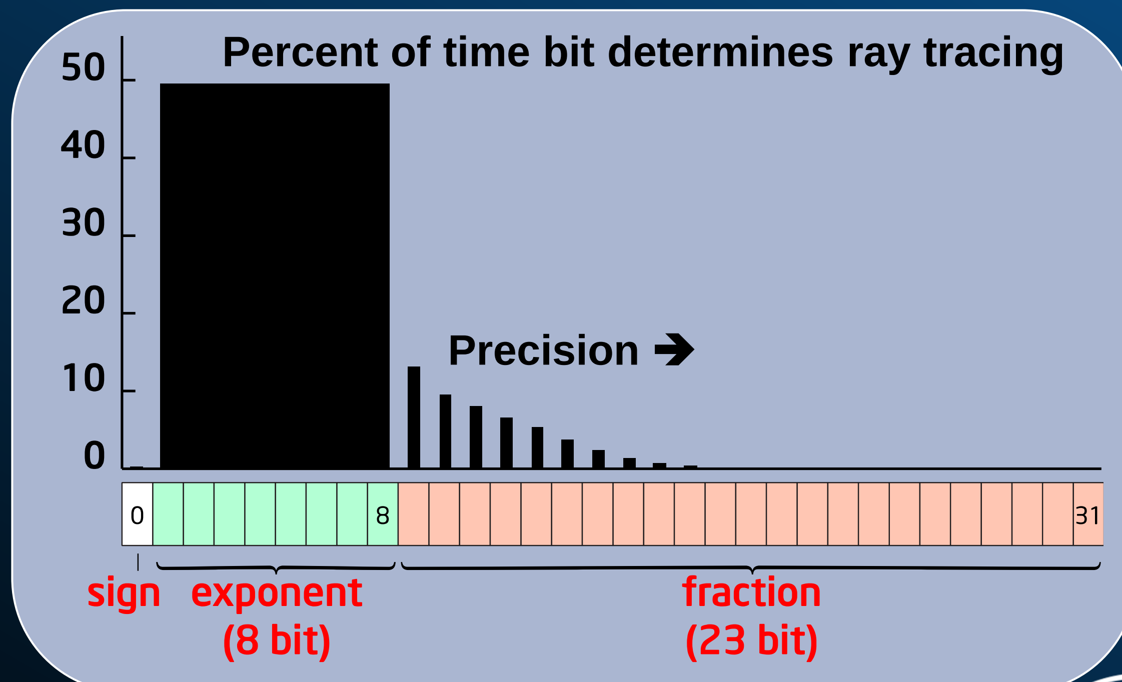
"It agrees with special-case analytic answers."

Intel Labs

It’s unlikely a code uses the best precision

- Too few bits gives unacceptable errors
- Too many bits wastes memory, bandwidth, energy
- 15 decimals *everywhere* to get 4 decimals in *result?*





Source: http://mantawiki.sci.utah.edu/manta/index.php/Main_Page

Intel Labs

Excess precision burdens DRAM energy, which is improving *too slowly*

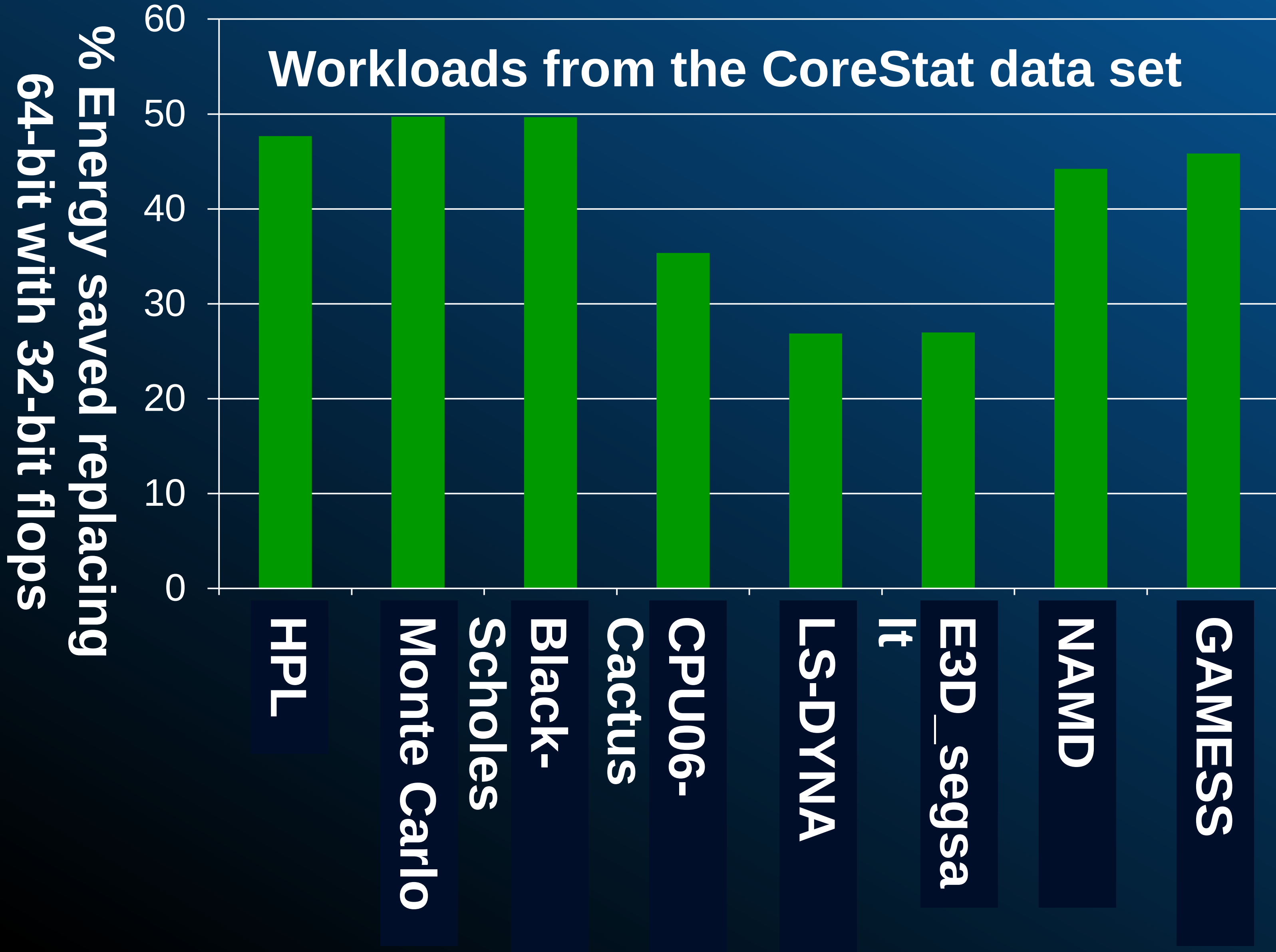
Operation	**Approximate energy consumed today**
64-bit multiply-add	64 pJ
Read/store register data	6 pJ
Read 64 bits from DRAM	**4200 pJ**
Read 32 bits from DRAM	***2100 pJ***

Simply using single precision in DRAM instead of double saves as much energy as *30* on-chip floating-point operations.

Source: S. Borkar, Intel. Data is for 32 nm technology ca. 2010

Intel Labs

Energy savings from halving flop-width



Intel Labs

Accuracy-Aware Debugging Ideas

- Annotate FP code to track rounding history of each value. Initial values flagged as exact or inexact.
- Attach “odometer” for how many adds, mults, divs, roots, etc. Triad operations inherit total histories of parents plus current operation.
- Distinguish left-digit destruction (ldd) from rounding
  - $3.14159 - 3.14000 = 1.59\times10^{-3}$ (left digit destruction)
  - $10000000 + 0.5 = 10000000$ (rounding)
- Halt options: relative error too high, absolute error too high, conditional test indeterminate.

Intel Labs

Example: Quadratic Equation

- Programmer needs to solve $ax^2 + bx + c = 0$
- Recalling elementary school math, naïvely uses $r_1, r_2 = (-b \pm (b^2 - 4ac)^{1/2})/(2a)$
- But $(b^2 - 4ac)^{1/2}$ might be very close to $\pm b$, resulting in left-digit destruction for one root.

Let's try this for $a = 3$, $b = 100$, $c = 2$, and seven-decimal precision.

Intel Labs

Operation trace

`t1=b*b`	b^2	0.1000000×10^5
`t2=4*a`	$4a$	0.1200000×10^2
`t2=t2*c`	$4ac$	0.2400000×10^2
`t2=t1–t2`	$b^2 - 4ac$	0.9976000×10^4
`if t2≤0, then print`	Exit if solution is	
`“degenerate or`	degenerate or involves	
`non-real answer”`	imaginary numbers	
`stop`		
`end if`		
`t2=sqrt(t2)`	$(b^2-4ac)^{1/2}$	0.9987993×10^2
`r1=–b+t2`	$-b+(b^2-4ac)^{1/2}$	0.1200700×10^0
`r1=r1/2`	$(-b+(b^2-4ac)^{1/2})/2$	0.6003500×10^{-1}
`r1=r1/a`	First root	0.2001167×10^{-1}
`r2=–b–t2`	$-b-(b^2-4ac)^{1/2}$	-0.1998799×10^3
`r2=r2/2`		-0.9993995×10^2
`r2=r2/a`	Second root	-0.3331332×10^2
`print r1, r2`		
`end`		

Intel Labs

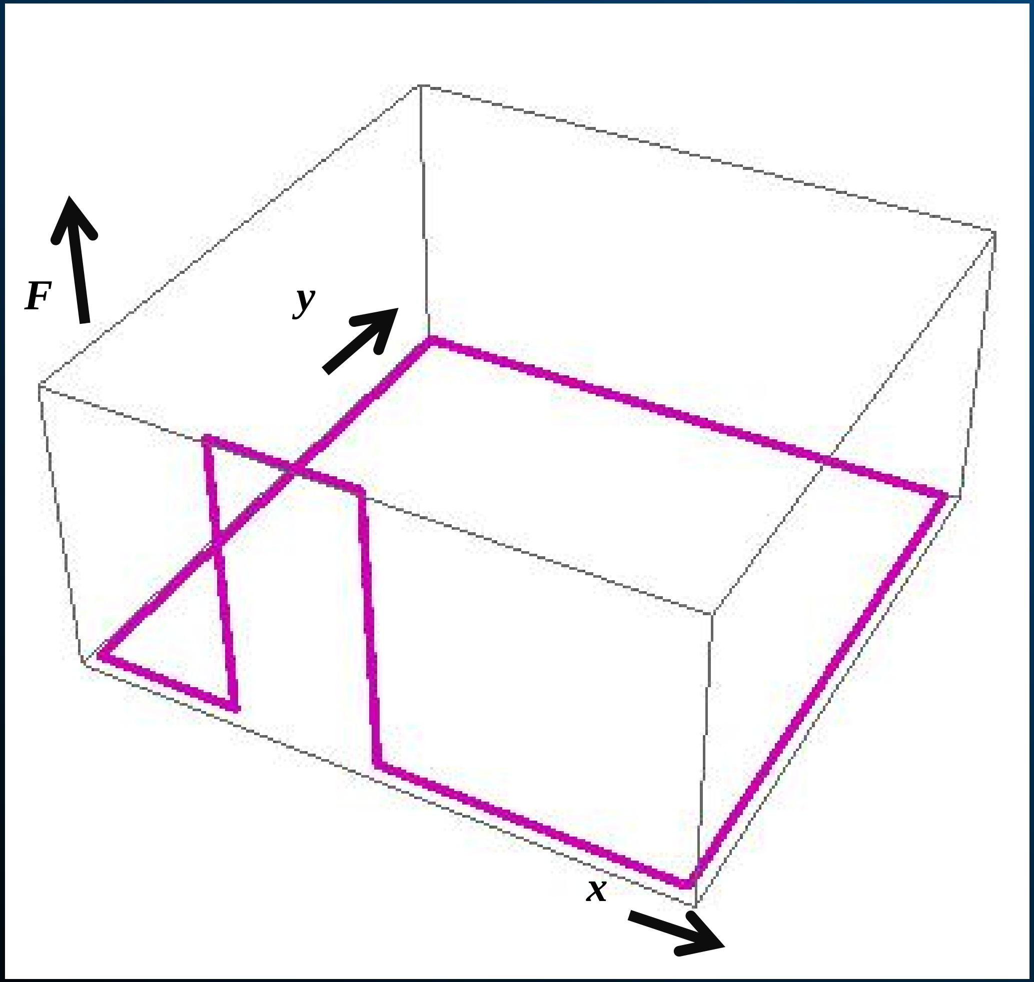
Tracing with accuracy-aware tool

	HULPs		S.M. digits			Value		±
	ldd	*	/	√				
`t1 = b * b`	0	1	0.1000000 × 10^5	0	0	1	0	0
`t2 = 4 * a`	0	2	0.1200000 × 10^2	0	0	1	0	0
`t2 = t2 * c`	0	2	0.2400000 × 10^2	0	0	2	0	0
`t2 = t1 – t2`	0	4	0.9976000 × 10^5	1	0	3	0	0
`if t2 ≤ 0, print “degen./complex”;stop`			Test is safe; no ambiguity					
`t2=sqrt(t2)`	1	7	~ 0.9987993 × 10^2	1	0	3	0	1
`r1=–b+t2`	100	5	~ 0.1200700 × 10^0	2	2	3	0	1
`r1=r1/2`	100	5	~ 0.6003500 × 10^{-1}	2	2	3	1	1
`r1=r1/a`	101	7	~ 0.2001167 × 10^{-1}	2	2	3	2	1
`r2=–b–t2`	2	7	~–0.1998799 × 10^3	2	0	3	0	1
`r2=r2/2`	3	7	~–0.9993995 × 10^2	2	0	3	1	1
`r2=r2/a`	4	7	~–0.3331332 × 10^2	2	0	3	2	1
`print r1, r2`								
`end`								

Can we use *16-bit* floating point?

- Three decimals of accuracy
- Dynamic range of 10 orders of magnitude
- 4x savings over 64-bit flops *if* it can be made numerically safe
- What about replacing 64-bit floating point with 16-bit interval bounds (32 bits total, still a 2x savings but mathematically rigorous)?
- Intel's Ivy Bridge chip will be first to support 16-bit floating point storage format (but not native ops)

Example: Laplace's Equation



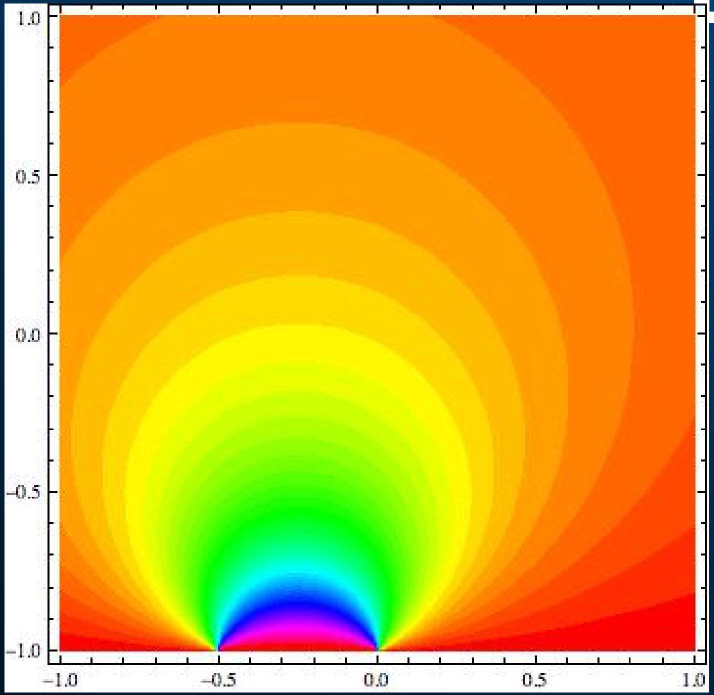
- Magenta line specifies boundary condition.
- Inside the unit square,

$$\nabla^2 F = 0$$

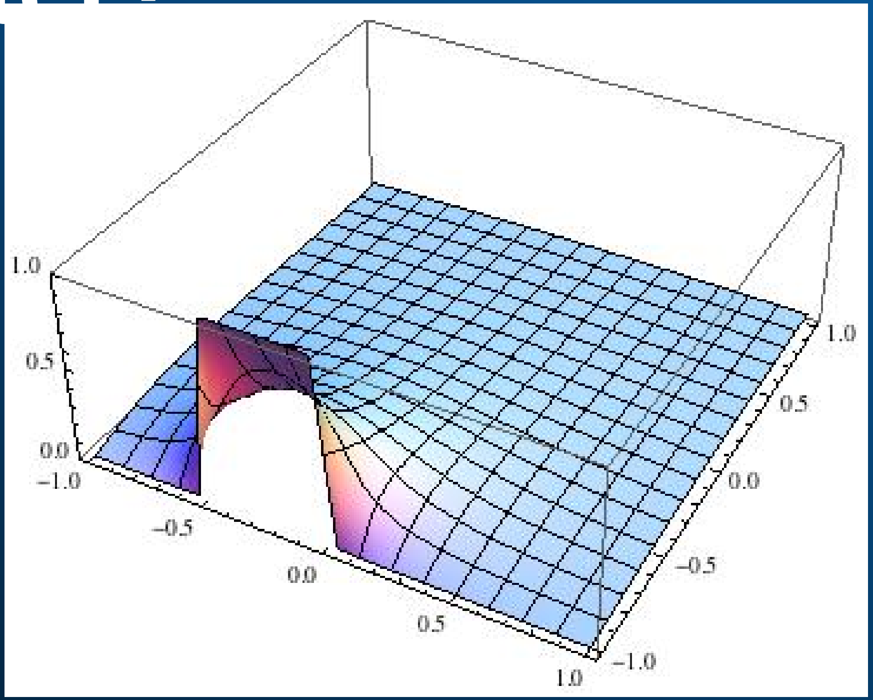
- (Classic problem for relaxation methods, but multigrid has lowest arithmetic complexity.)

Intel Labs

Laplace's Solvers: Which is Better?



64-bit floating point method seems to have converged. 15 decimals, some of them probably correct.

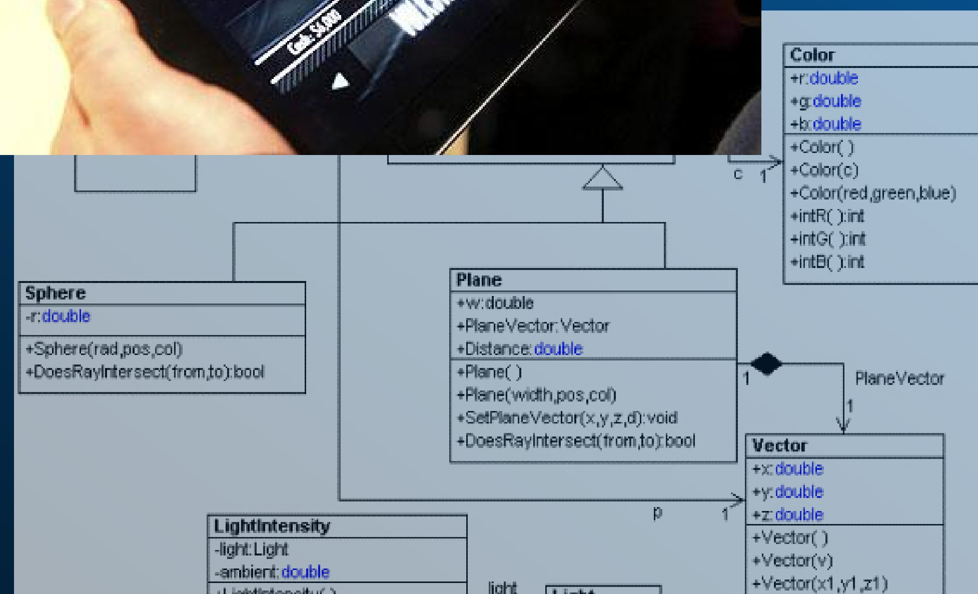


16-bit *interval* arithmetic provably bounds answer to 3 decimals, uses half the storage and bandwidth and energy

Intel Labs

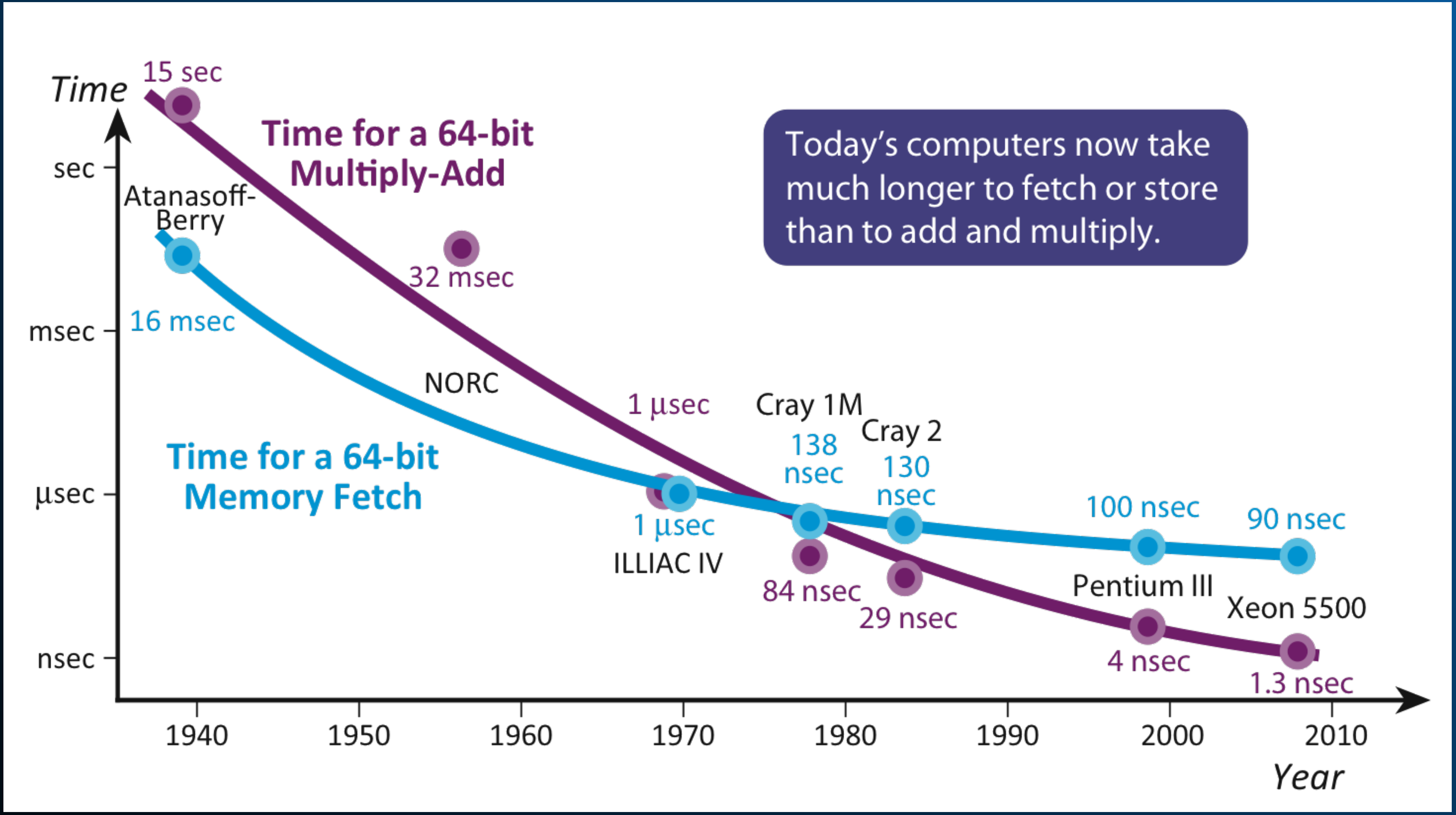
A Developer Scenario

- Developer compiles app with tool to track accuracy, display results with “± n.nn” outputs
- Discovers 95% of app only needs 16-bit ops; tool identifies 5% where 32-bit needed.
- Developer rewrites app for 16-bit ops, *removes* accuracy tracking for production version
- 4x speed in Ivy Bridge, more frames per second, less power throttling in large data center servers



Intel Labs

Right-Sizing Precision Can Relieve the "Memory Wall"



Each halving of precision relieves the memory wall by about 2x.

Intel Labs

Experimental Results

- Speech recognition (sphinx3, from SPECfp) uses 22 bits more precision than needed to get required accuracy
- Computational fluid dynamics (lbm, from SPECfp) uses 19 bits more precision than needed
- Shock hydrodynamics (weapons design, from DARPA Challenge Problems) uses 15 bits more precision than needed

This is the *uniform* amount we can reduce precision, safely. More improvement is possible for operation-by-operation trimming.

Intel Labs

Some Approaches

Approach	Background
Interval arithmetic	Rigorous, historically in Intel MKL, decades of papers on how to use it
Accuracy-tracking software tools	Quick to develop and apply; Berkeley backing; leads to hardware efficiency; **doesn't change answers**, just monitors
Support 128-bit precision	Easy way to check accuracy; not that expensive for on-chip scratch results
Rational arithmetic	Represents fractions perfectly; great way to check + – * / operations

Intel Labs

Five Plausible Schemes (Kahan)

"Can the effects of roundoff upon a floating-point computation be assessed without submitting it to a mathematically rigorous and (if feasible at all) time-consuming error-analysis? In general, *No*.

"This mathematical fact of computational life has not deterred advocates of schemes like these:

1. Repeat the computation in arithmetics of increasing precision, increasing it until as many as desired of the results' digits agree.
2. Repeat the computation in arithmetic of the same precision but rounded differently, say *Down*, and then *Up*, and maybe *Towards Zero* too, besides *To Nearest*, and compare three or four results.
3. Repeat the computation a few times in arithmetic of the same precision rounding operations randomly, some *Up*, some *Down*, and treat results statistically.
4. Repeat the computation a few times in arithmetic of the same precision but with slightly different input data each time, and see how widely results spread.
5. Perform the computation in *Significance Arithmetic*, or in *Interval Arithmetic*.

Interval Math: Due for a Revival?

$A \le x \le B$, or x is in [A, B], where A and B are representable, exact floating-point numbers

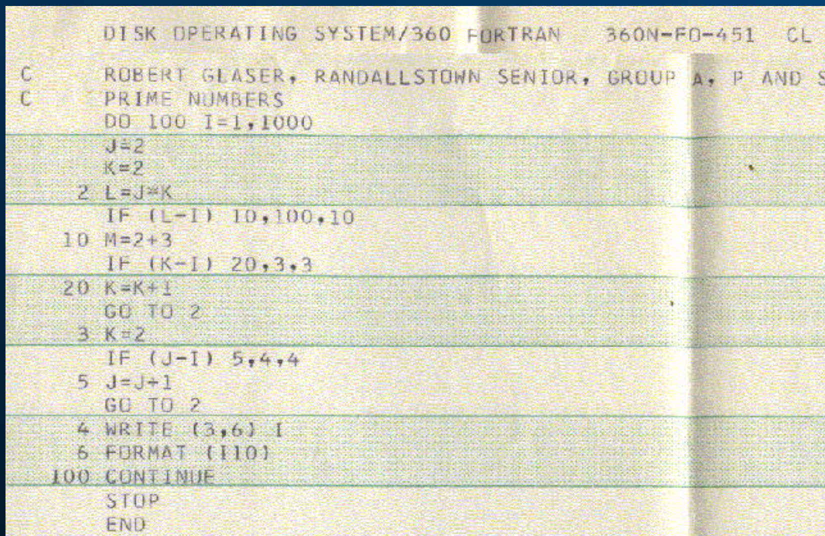
- Interval Arithmetic has been tried for decades, but often produces bounds too loose to be useful.
- In many other areas of computing, speed has been turned into improved quality of answer, not reduction in total task time.
- Midpoint-radius storage ($x \pm r$) is more bit-efficient than [A,B] because when bounds are tight, A and B have redundant bits
- By doing more flops AND using many cores, we can keep the bounds tight, and produce rigorous, high-quality answers for the first time.

Intel Labs

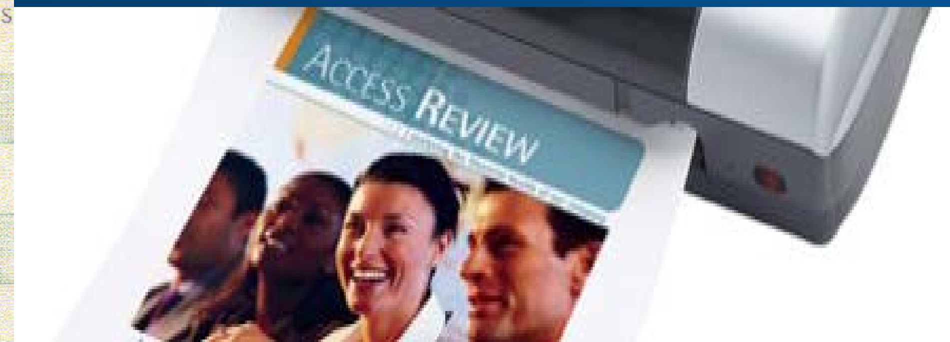
Rigorous bound approaches exist for

- Radiation transfer (graphics, heat)
- Pin-connected truss structures (general structural analysis in the limit of fine structures)
- N-body dynamics (useful for provable CFD?)
- PDEs like Laplace where bounding the forcing function leads to bounds on the answer
- This could be a “Golden Age” for algorithm research! We need all new methods.

Analogy: Printing in 1970 versus 2012



```
      DISK OPERATING SYSTEM/360 FORTRAN    360N-FO-451   CL
C     ROBERT GLASER, RANDALLSTOWN SENIOR, GROUP A, P AND S
C     PRIME NUMBERS
      DO 100 I=1,1000
      J=2
      K=2
    2 L=J*K
      IF (L-I) 10,100,10
   10 M=2+3
      IF (K-I) 20,3,3
   20 K=K+1
      GO TO 2
    3 K=2
      IF (J-I) 5,4,4
    5 J=J+1
      GO TO 2
    4 WRITE (3,6) I
    6 FORMAT (I10)
  100 CONTINUE
      STOP
      END
```



We use faster technology for** better **prints, not to do low-quality prints in milliseconds.

Intel Labs

The Single-Use Expression problem

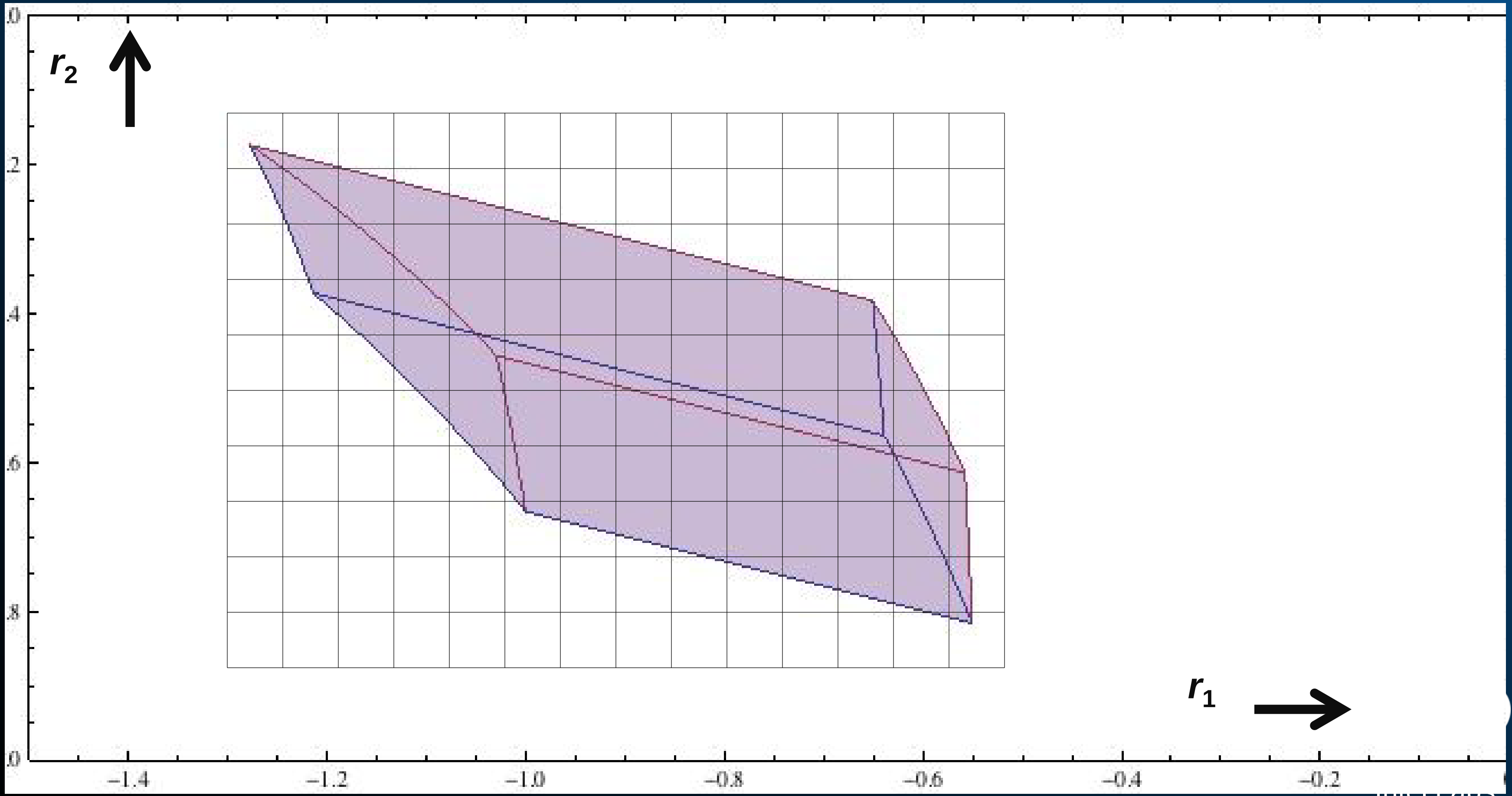
- What wrecks interval arithmetic is simple things like

$$F(x) = x - x.$$

- The answer should be 0, or maybe $[-\varepsilon, +\varepsilon]$. But if x is the interval [3, 4], then interval $x - x$ stupidly evaluates to [–1, +1], which doubles the uncertainty (interval width) and makes the interval solution far inferior to the point arithmetic method.
- Interval proponents say we should seek expressions where each variable only occurs once (SUE = Single Use Expression). But that's impractical or impossible in general.
- One approach, "mincing", not only solves the problem but gives us something to do with all those millions of cores!

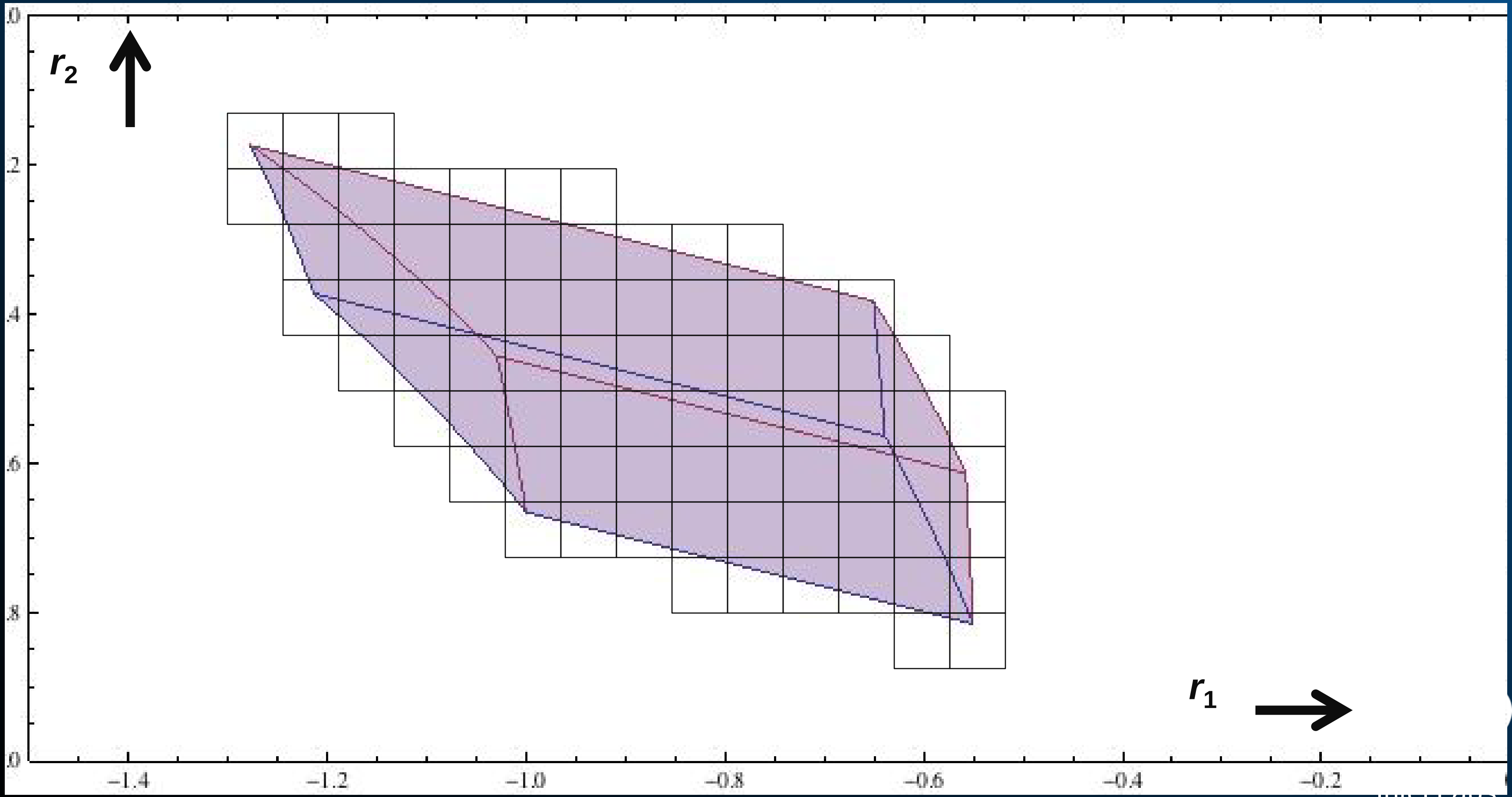
Rigorous Quadratic Equation Bounds-1

- Find roots $\underline{r}_1$, r_2 for interval *a*, *b*, *c* values in $ax^2+bx+c=0$.
- Completely contain possible answer set, without



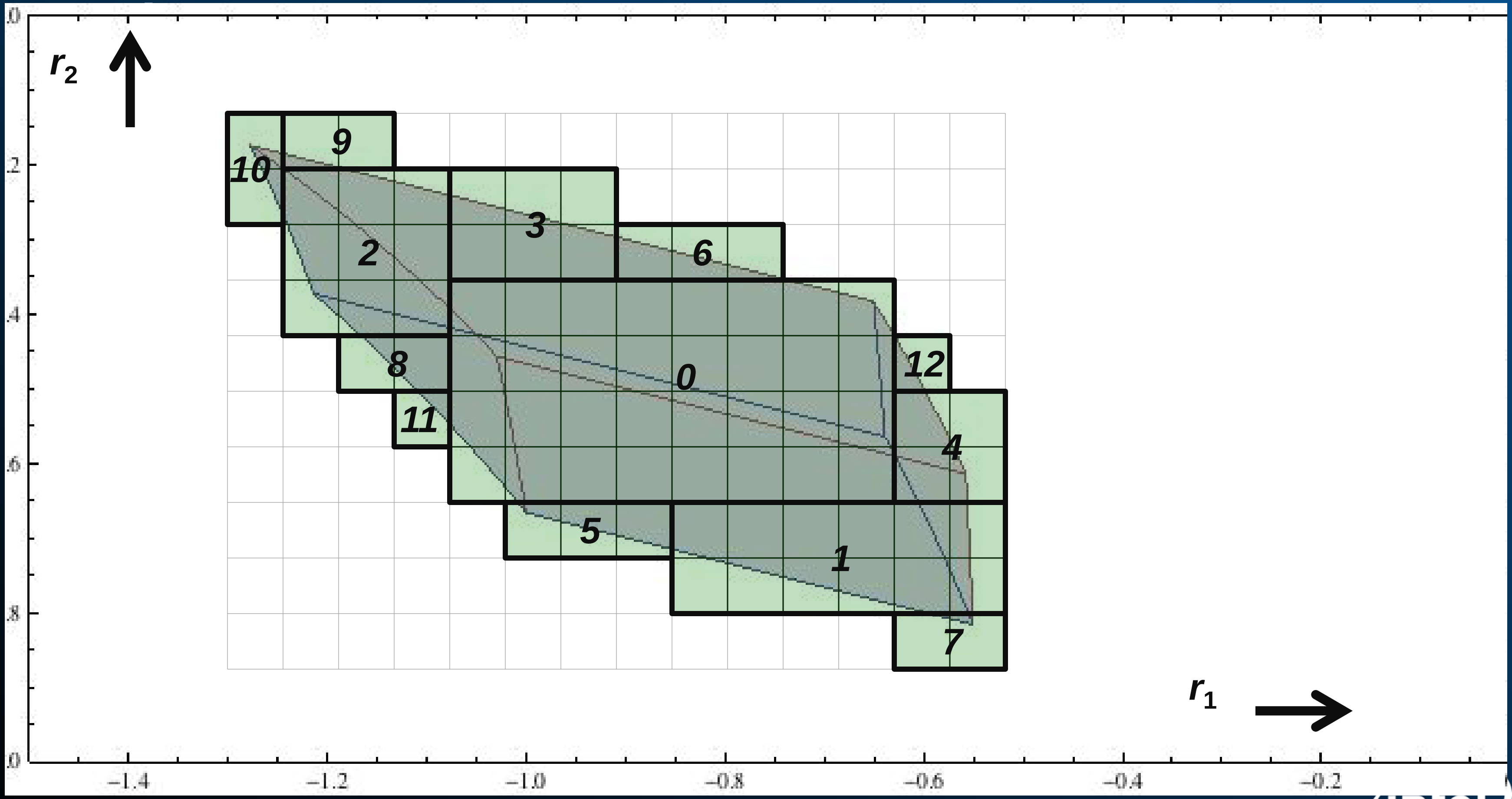
Rigorous Quadratic Equation Bounds-2

- Remove all squares not part of the cover set.



Rigorous Quadratic Equation Bounds-3

- Assign processors different 2D intervals in that cover set, each propagating to the next computing



Benefits of this approach

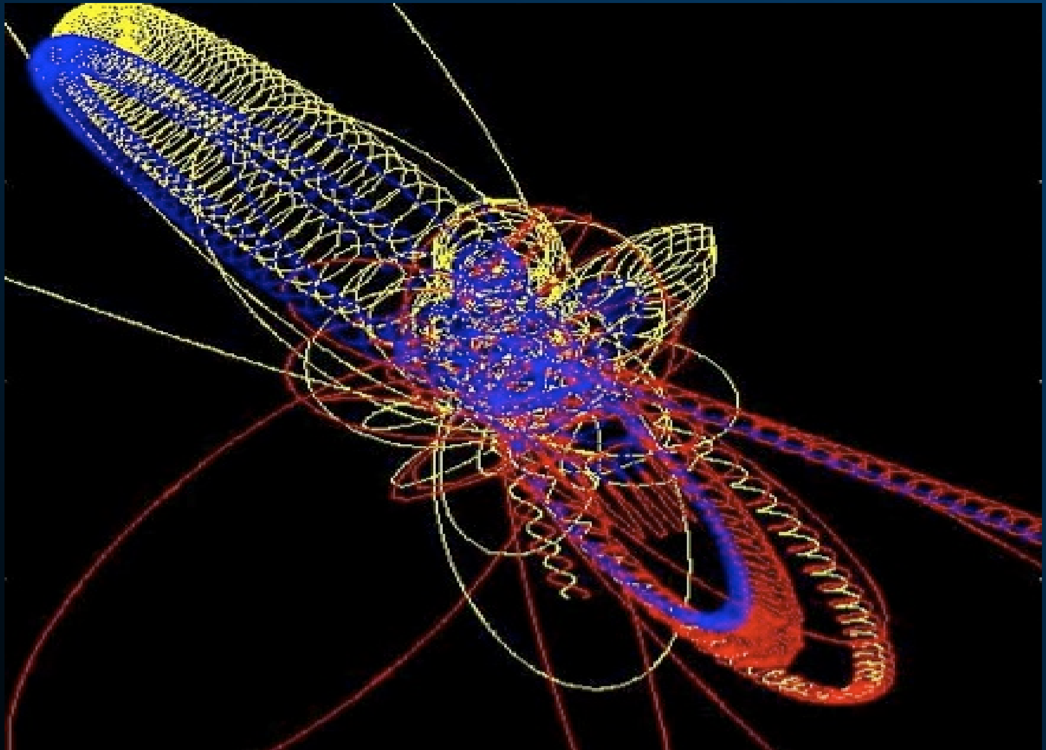
1. This is a new direction of scaling a problem. The more processors and speed, the higher the answer quality. A single core gets a rigorous “containment” of the answer, but looser than a powerful computer can get.

2. Provides resiliency check for floating-point math; error shows up as a value that is *not contiguous* when the starting set was contiguous. (Like a voting scheme, except there is no useless redundancy; every computation helps get answer)

3. Drastically increases the ratio of useful floating-point operations to memory operations, helping with “the memory wall”!

Even the 3-Body Problem is *Highly Parallel*

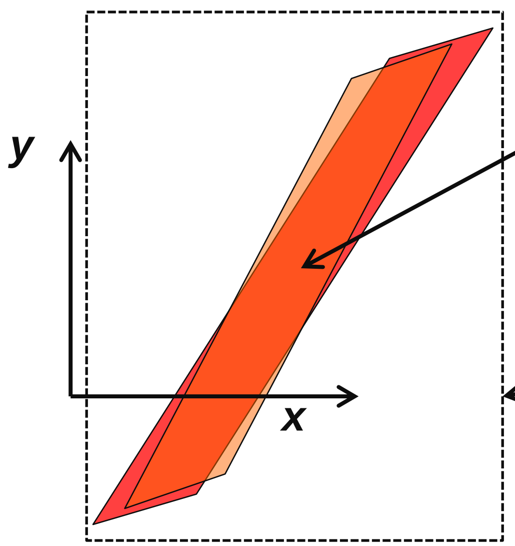
- Appears “Embarrassingly Serial” with only 18 variables, yet simulation involves a huge number of serial steps.
- However: each step produces an irregular containment set. Use all available cores to track.
- Far more ops per data point. Billions of cores usefully employed. Provable bounds on the answer.



Intel Labs

Linear Solvers: Challenging Once Again?

- Even 2 equations in 2 unknowns involves computational geometry… intersecting 8 half-planes (2 parallelograms).
- "Ill-posed" problems much less of a problem with intervals!
- Ultimate solution is the minimum "containment set."
- Box around that solution leads to "the wrapping problem"



Answer is the set of all *x*-*y* floating-point squares containing *any part* of the overlap.

Sloppy bound is the box interval containing the overlap.

Idea Convergence

Automate much of the work of a numerical analyst

New parallel techniques for tight, rigorous bounds

Hardware with built-in *X±r* accuracy tracking field

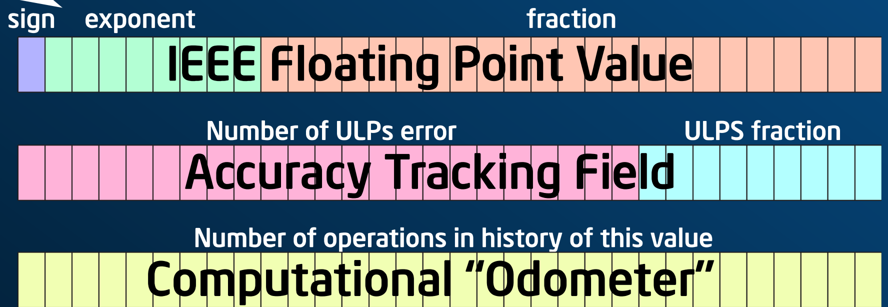
Intel Labs test chip

Accuracy fields attached to FP

Interval arithmetic made practical

Numbers that know their own accuracy and history

Accuracy-Aware Arithmetic



Note: This involves *NO* changes to IEEE arithmetic.

Intel Labs

Summary

- DRAM bandwidth is precious.
- So is DRAM.
- Making arithmetic accuracy-aware *reduces the number of bits we need to move*, saving time, bandwidth, & energy.
- This addresses a chronic problem with floating-point math being treacherous.

Intel Labs

Perhaps "accuracy-aware arithmetic" is the next step in mathematical computing.

Integers → Fixed-point → Floating-point → Accuracy-aware