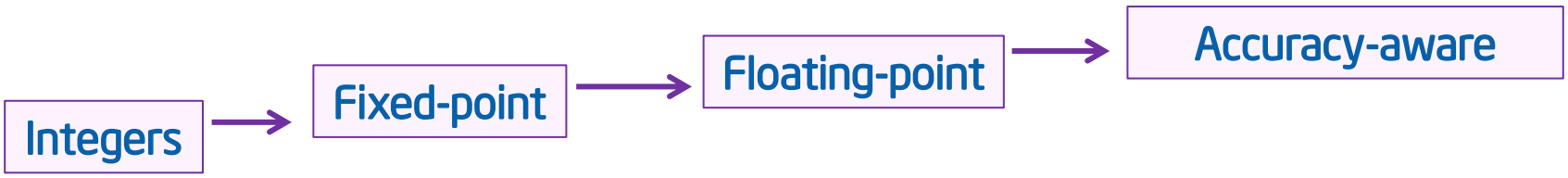
Accuracy-Aware High-Efficiency Math

Tim Mattson
Principle Engineer
Intel Labs



Disclaimer

READ THIS … its very important

- The views expressed in this talk are those of the speaker and not his employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if I say anything really stupid, it's my fault … don't blame my collaborators.

Floating Point numbers are not Real

- Floating point numbers are NOT a closed set.
  - "A op B" can generate results that don't fit in a floating point format.

```
float A = 0.01f;

If (100 * A   !=   1.0) printf("oops");
```

Output: "oops"

```
float, c, b = 1000.2f;
c = b - 1000.0;
printf (" %f", c);
```

Output: 0.200012

- 0.01 and 0.2 do not have exact binary representations … so the computer rounds to the nearest floating point number.

Floating point arithmetic is not associative

- Floating point numbers are:
  - NOT Associative: A * (C * B) ≠ (A * C) * B
  - NOT Distributive: A*(B+C) ≠ A*B + A*C
- A simple test:
  - Fill 2 arrays each with 10000 random values between 0.0 and 1.0
  - Shift one up by 100 and one down by 0.001
  - Mix the arrays together, sum them and subtract a large number (500000)
  - Results:
    - 170.968750 with 1 thread
    - 171.968750 with 2 threads
    - 172.750000 with 4 threads

Floating point arithmetic is not associative

- Floating point numbers are:
  - NOT Associative: A * (C * B) ≠ (A * C) * B
  - NOT Distributive: A*(B+C) ≠ A*B + A*C
- A simple test:
  - Fill 2 arrays each with 10000 random values between 0.0 and 1.0
  - Shift one up by 100 and one down by 0.001
  - Mix the arrays together, sum them and subtract a large number (500000)
  - Results:
    - 170.968750 with 1 thread
    - 171.968750 with 2 threads
    - 172.750000 with 4 threads

Which of these answers is right?

They are all equally "right" … the true answer is 177.750

You can't pick one random order of FLOPS and arbitrarily call it the "right one".

"How do you know the answer to a floating point computation is correct?"

Common responses:

- *Laughter … "of course they are correct … you must be joking"*
- "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."

… But this is not a joke. It is a very serious question

When you don't know accuracy (1)…

Sleipner Oil Rig Collapse (8/23/91). Loss: **$700 million.**

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

Inaccurate linear elastic model used with NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.

Third party names are the property of their owners

When you don't know accuracy (2)…

Vancouver stock exchange index undervalued by 50%

(Nov. 25, 1983)

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

Index managed on an IBM/370. 3000 trades a day and for each trade, the index was truncated to the machine's REAL*4 format, loosing 0.5 ULP per transaction. After 22 months, the index had lost half its value.

Third party names are the property of their owners

…and inaccuracy can *really* hurt

Patriot missile incident (2/25/91). Failed to stop a scud missile from hitting a barracks, **killing 28 Americans.**

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

System counted time in 1/10 sec increments … which doesn't have an exact binary representation. Over time, error accumulates. The incident occurred after 100 hours of operation … at which point the accumulated errors in time variable resulted in a 600+ meter tracking error.

Third party names are the property of their owners

The Problem

- How often do we have “working” software that is “silently” producing inaccurate results?
  - We don’t know … nobody is keeping count.
- But we do know this is an issue for 2 reasons:
  (see Kahan’s desperately needed Remedies…)
  - Numerically Naïve (and unchallenged) formulas in text books (e.g. solving quadratic equations).
  - Errors found after years of use (Rank estimate in use since 1965 and in LINPACK, LAPACK, and MATLAB (Zlatko Drmac and Zvonimir Bujanovic 2008, 2010). Errors in LAPACK’s _LARFP found in 2010.)
- Solution? We need programmers to understand numerical analysis … but that isn’t going to happen.

> Computer Science has changed over my lifetime. Numerical Analysis seems to have turned into a sliver under the fingernails of computer scientists
>
> Prof. W. Kahan, Desperately needed Remedies … Oct. 14, 2011

How should we respond?

- Programmers should conduct mathematically rigorous analysis of their floating point intensive applications to validate their correctness.
- But this won't happen … training of modern programmers all but ignores numerical analysis. The following tricks* help and are better than nothing …
  1. Repeat the computation with arithmetic of increasing precision, increasing it until a desired number of digits in the results agree.
  2. Repeat the computation in arithmetic of the same precision but rounded differently, say *Down* then *Up* and perhaps *Towards Zero,* then compare results.
  3. Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results varry.

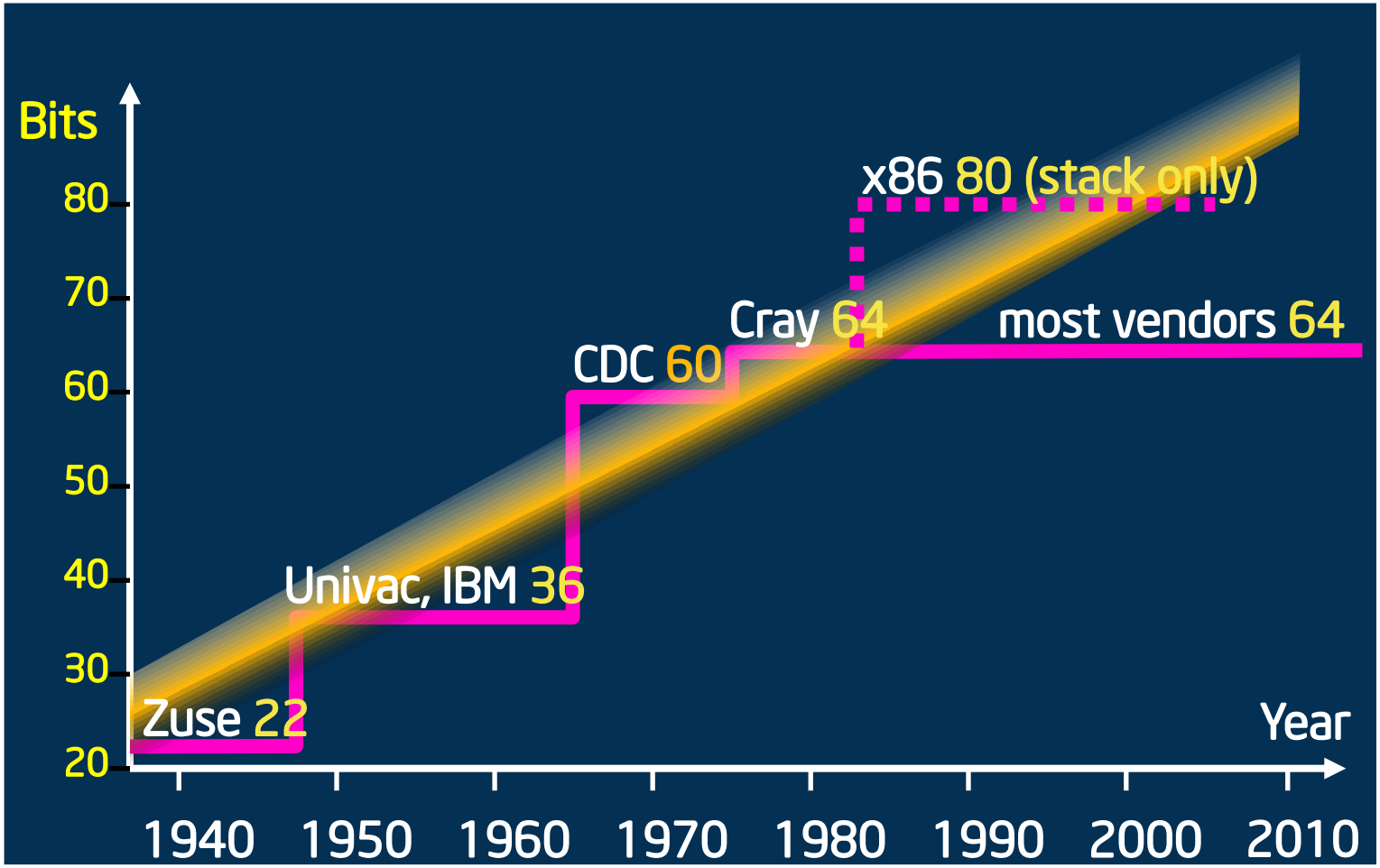
These are useful techniques, but they don't go far enough. How can the discerning skeptic confidently use FLOPs?

*Source: W. Kahan: How futile are mindless Assessments of Roundoff in floating-point computation?

Outline

- What is the problem?
- Solutions
  - Use so many bits you can pretend there is no problem
  - Change how we model real arithmetic on computers
  - Use only the bits you need
  - Let the hardware solve the problem
- Conclusion

Solution: use lots of bits and hope for the best …



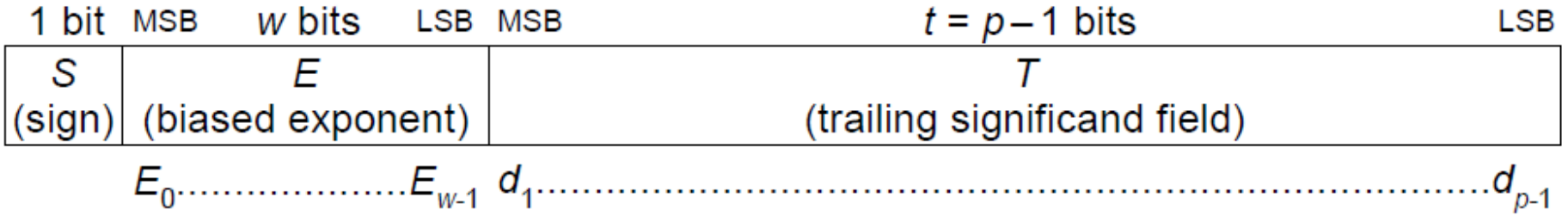
Is 64 bits enough? Is it too much? We're *guessing*.

Third party names are the property of their owners

Quad Precision

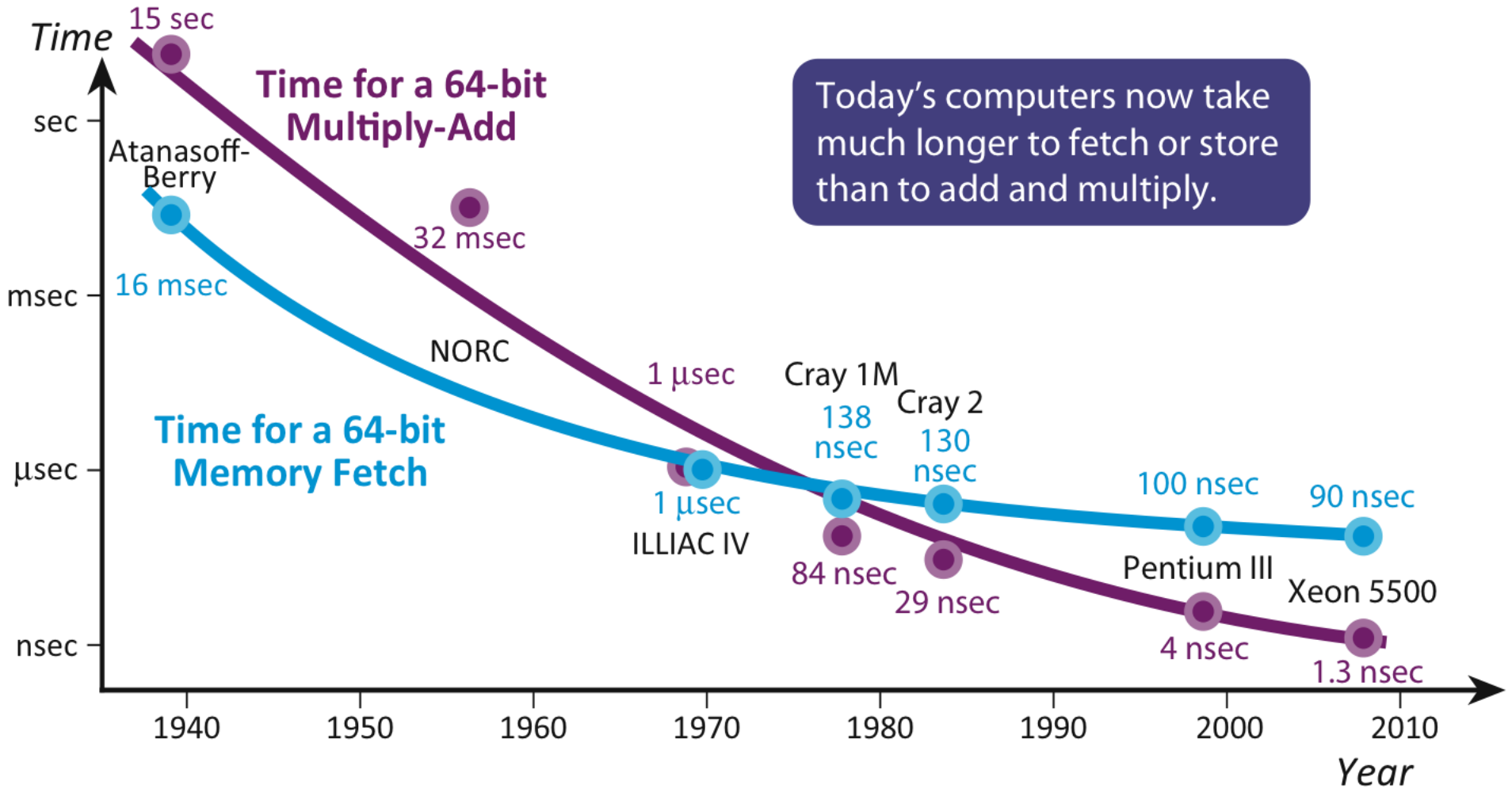
- IEEE 754™ defines a range of formats including quad (128)

	binary32	binary64	binary128
P, digits	24	53	113
emax	+127	+1023	+16383



- There are pathological cases where you lose all the precision in an answer, but the more common case is that you lose only half the digits.
- Hence, for 32 or 64 bit input data, quad precision (113 significant bits) is probably adequate to make most computations safe (Kahan 2011).

Wider floating point formats turn compute bound problems into memory bound problems



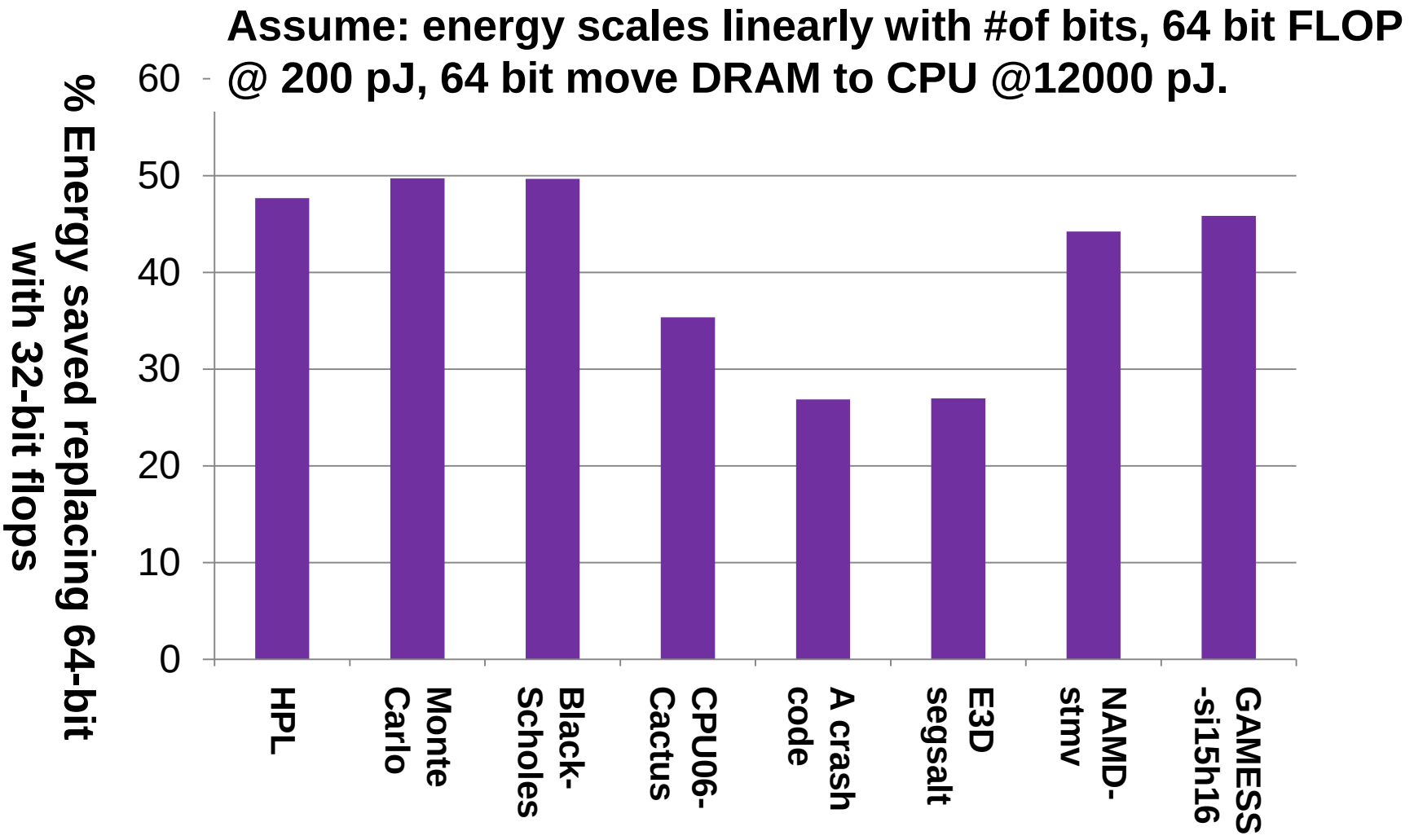
Energy implications of floating point numbers: 32 bit vs. 64 bit numbers

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

energy savings: replace 64 bit flops with 32 bit flops



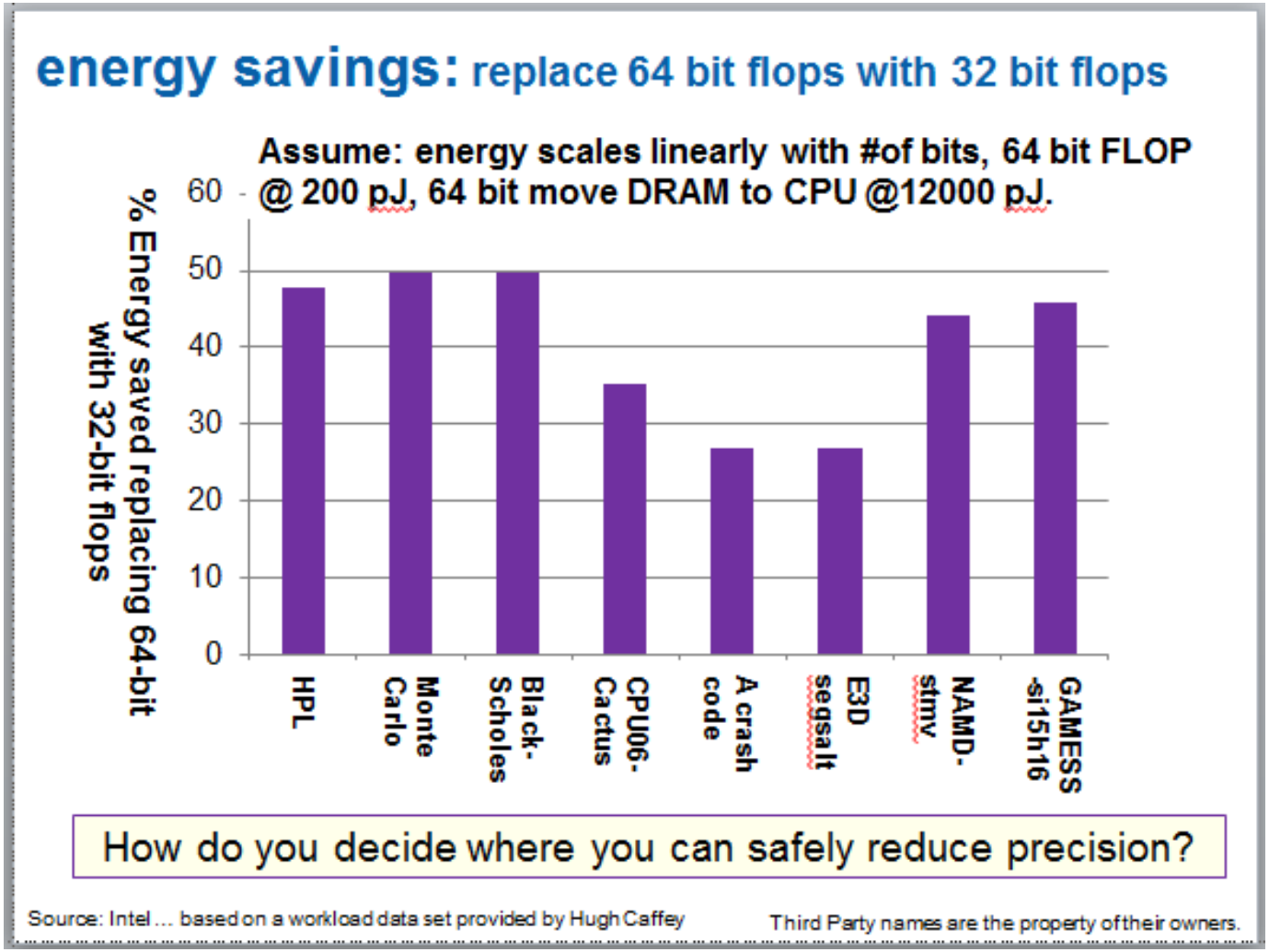
How do you decide where you can safely reduce precision?

Source: Intel … based on a workload data set provided by Hugh Caffey

Third Party names are the property of their owners.

Maybe we don't want Quad after all?

- If Performance/Watt is the goal, using Quad everywhere to avoid careful numerical analysis is probably a bad idea.



Outline

- What is the problem?
- Solutions
  - Use so many bits you can pretend there is no problem
  - Change how we model real arithmetic on computers
  - Use only the bits you need
  - Let the hardware solve the problem
- Conclusion

Interval Numbers

- Interval number: the range of possible values within a closed set

$$\boldsymbol{x} \equiv [\underline{x}, \overline{x}] := \{x \in R \mid \underline{x} \leq x \leq \overline{x}\}$$

- Representing real numbers:
  - A single floating point number

    $1/3 \approx 0.333333$

  - An interval that bounds the real number

    $1/3 \in [0.33333,\ 0.33334]$

- Representing physical quantities:
  - An single value (e.g. an average)

    $radius_{earth} \approx 6371$ km

  - The range of possible values

    $radius_{earth} \in [6353, 6384]\ km$

Interval Arithmetic

Let $\boldsymbol{x = [a, b]}$ and $\boldsymbol{y = [c, d]}$ be two interval numbers

1. Addition $x + y = [a, b] + [c, d] = [a + c, b + d]$

2. Subtraction $x - y = [a, b] - [c, d] = [a - d, b - c]$

3. Multiplication $xy = [min(ac,ad,bc,bd), max(ac,ad,bc,bd)]$

4. Reciprocal $1 / y = [1/d, 1/c]$

5. Division $x/y = \begin{cases} x \cdot 1/y & c, d \neq 0 \quad y \notin 0 \\ [\infty, -\infty] & c, d \neq 0 \quad y \in 0 \end{cases}$

Properties of Interval Arithmetic

Let ***x***, ***y*** and ***z*** be interval numbers

1. Commutative Law

$$x + y = y + x$$

$$xy = yx$$

2. Associative Law

$$x + (y + z) = (x + y) + z$$

$$x(yz) = (xy)z$$

3. ***Distributive Law does not always hold***, *but*

$$x(y + z) \subseteq xy + xz$$

Functions and Interval arithmetic

- Interval extension of a function

$$[f]([x]) \supseteq \{f(y) | y \in [x]\}$$

- Naively can just replace variables with intervals. But be careful … you want an interval extension that produces bounds that are as narrow as possible. For example …

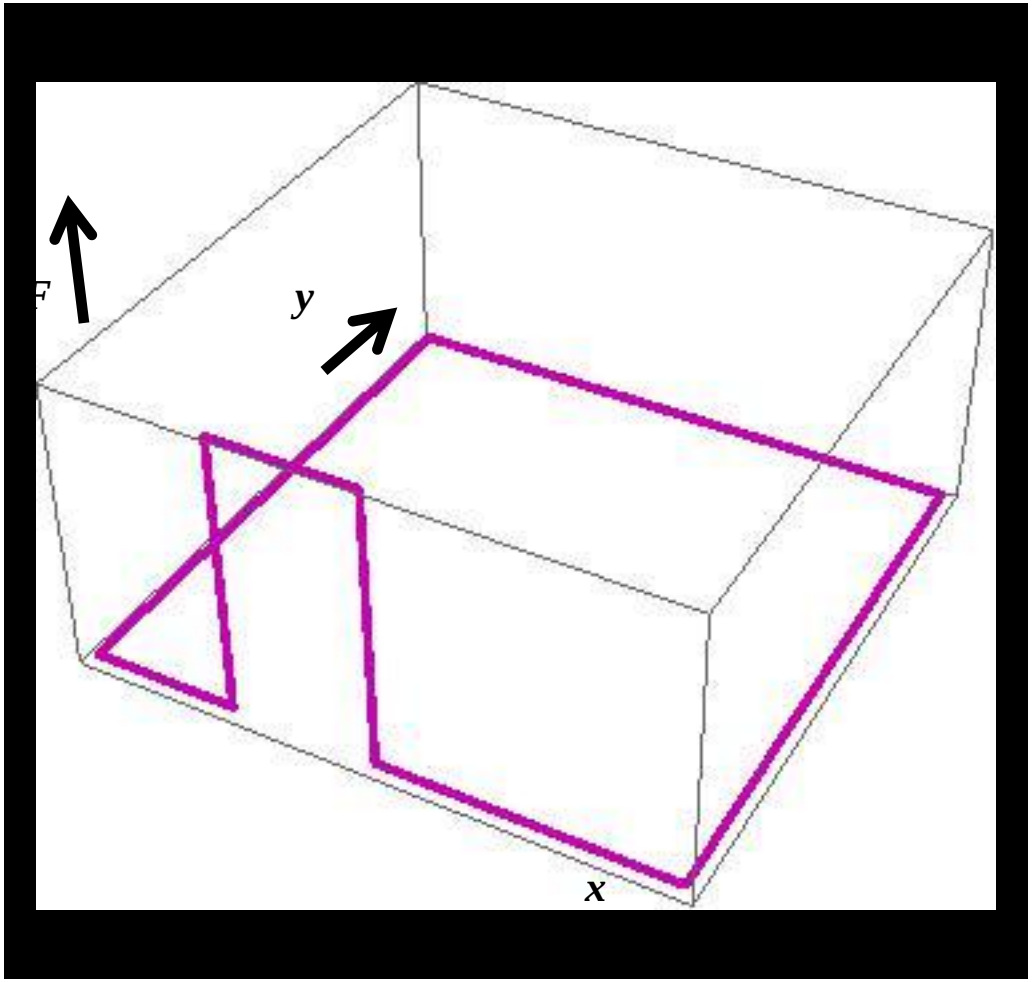
$$f(x) = x - x \qquad let\ x = [1,2]$$

$$f[x] = [1-2, 2-1] = [-1,1]$$

- An interval extension with tighter bounds can be produced by modifying the function so the variable x appears only once.

$$f(x) = x - x = x(1-1) = 0$$

Working with Intervals: Example: Laplace's Equation*



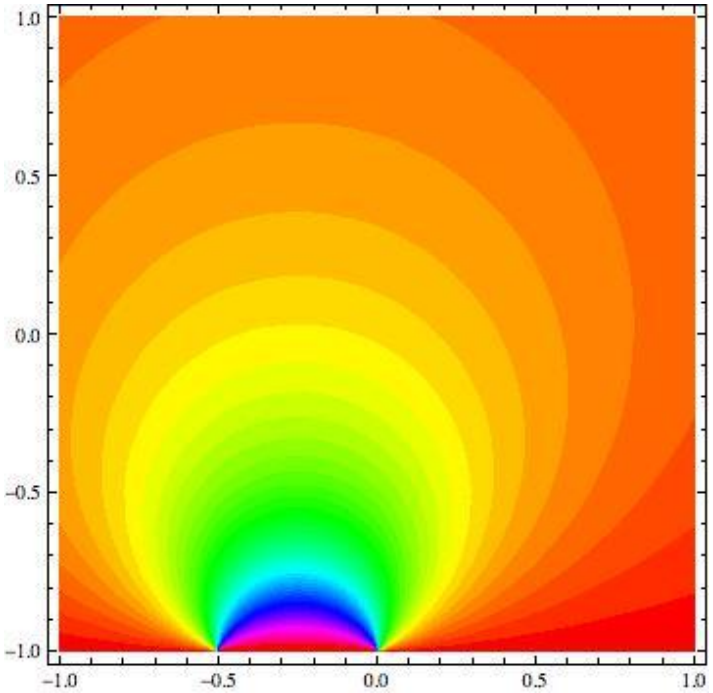
- Magenta line specifies boundary condition.
- Inside the unit square,

$$\nabla^2 F = 0$$

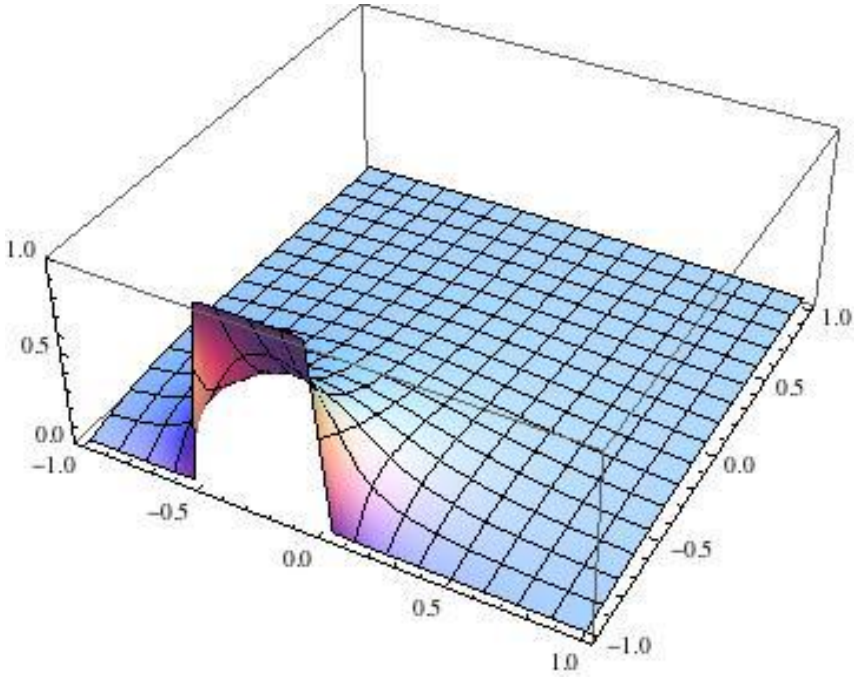
- (Classic problem for relaxation methods, but multigrid has lowest arithmetic complexity.)

*Source: John Gustafson of AMD

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, memory bandwidth and energy

*Source: John Gustafson of AMD

Interval Math: Due for a Revival?

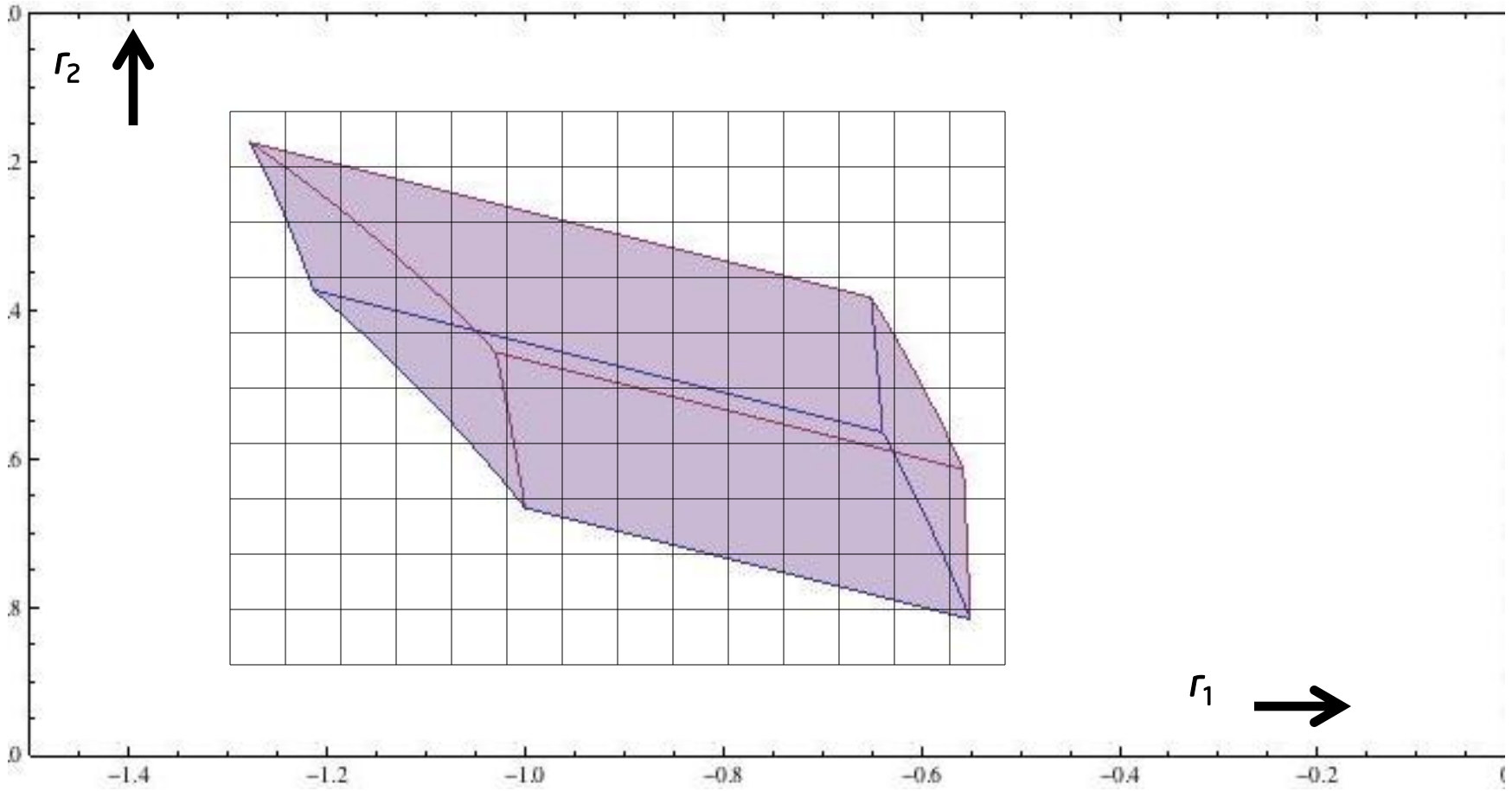
- Interval Arithmetic has been tried for decades, but often produces bounds too loose to be useful.
- In many other areas of computing, speed has been turned into improved quality of answer, not reduction in total task time.
- Midpoint-radius storage ($x \pm r$) is more bit-efficient than [A,B] because 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.

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
- 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.

Rigorous Quadratic Equation Bounds-1

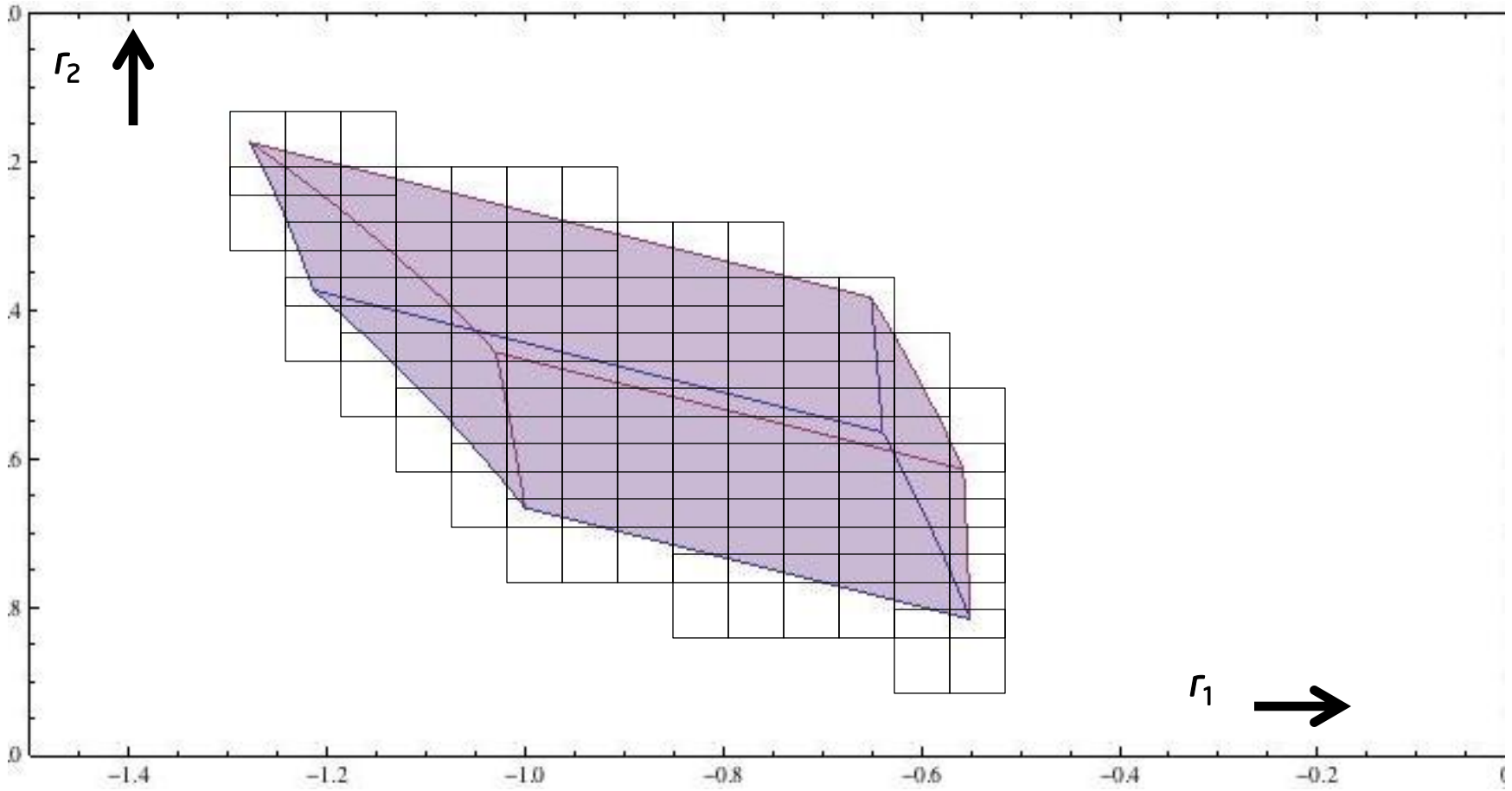
- Find roots r_1, r_2 for interval a, b, c values in $ax^2+bx+c=0$.
- Completely contain possible answer set, without waste.



*Source: John Gustafson of AMD

Rigorous Quadratic Equation Bounds-2

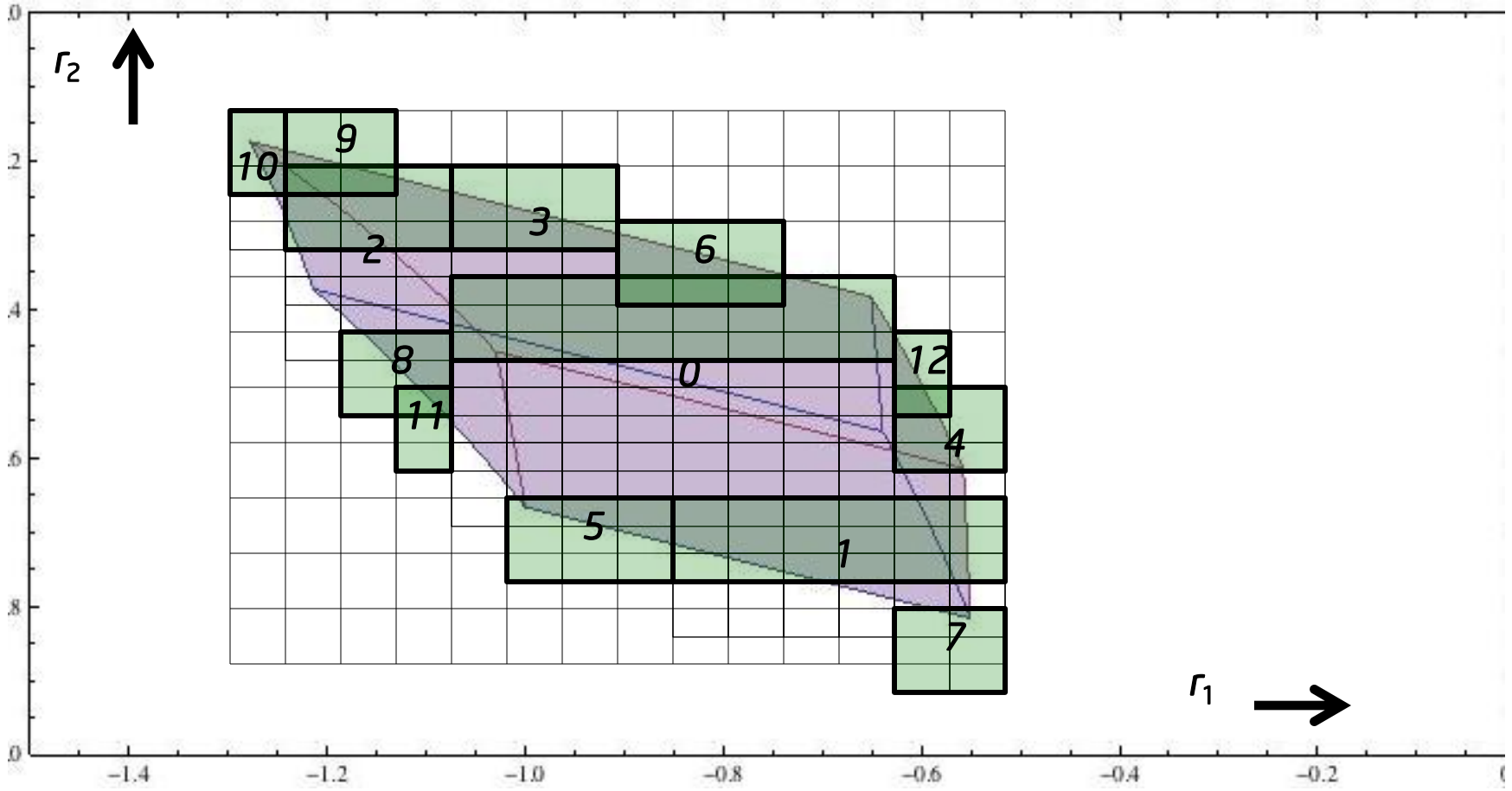
- Remove all squares not part of the cover set.



*Source: John Gustafson of AMD

Rigorous Quadratic Equation Bounds-3

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



*Source: John Gustafson of AMD

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”!

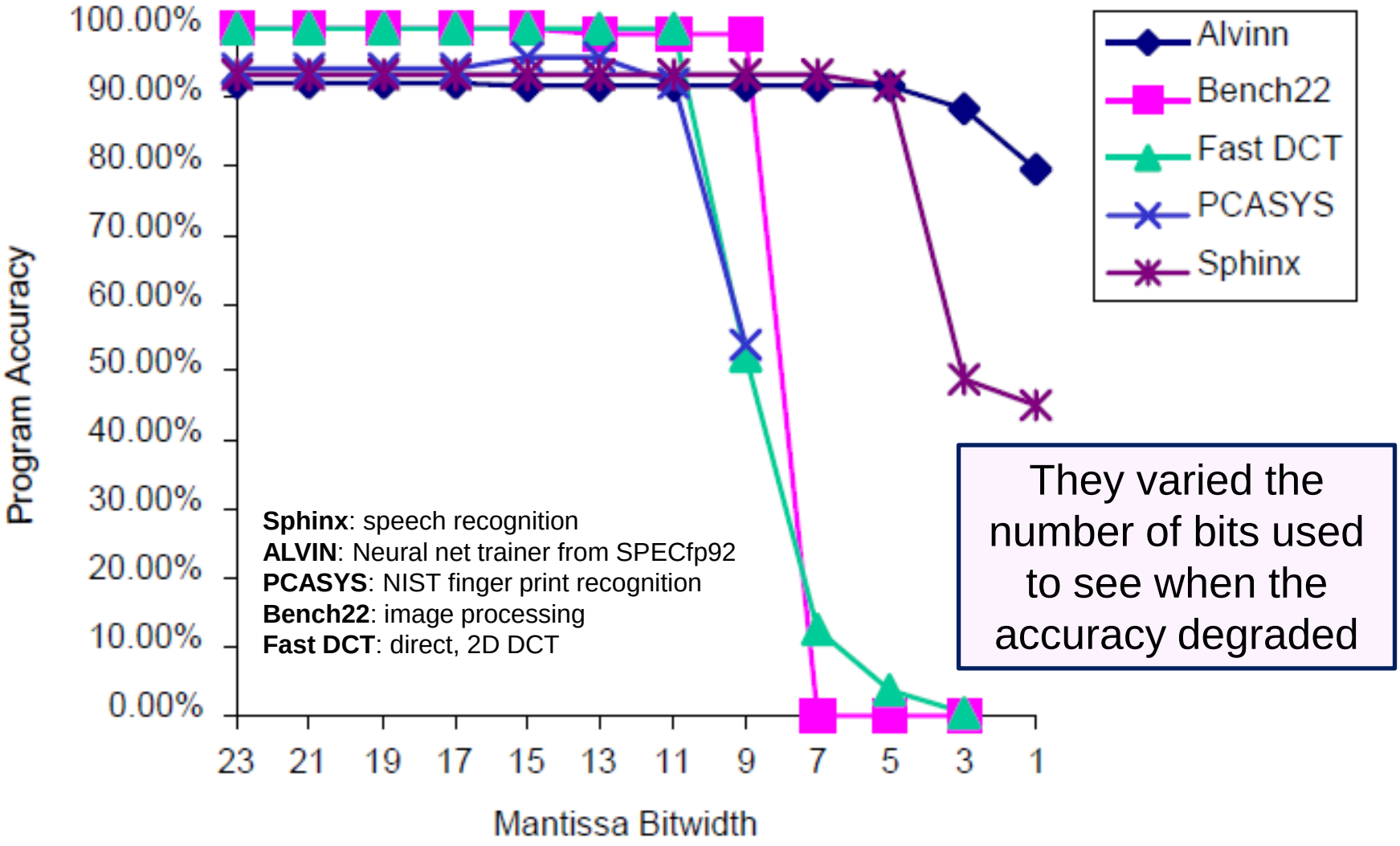
The problems with Intervals

- Interval ops are expensive:
  - Using directed rounding from IEEE754, you can do mathematically rigorous interval arithmetic on modern microprocessors … but they are slowed compared to “normal” floating point ops.
- Designing useful interval extensions of functions can be prohibitively difficult.
  - Just replacing floats with intervals results in bounds that are needlessly large (e.g. remember the division problem for intervals that contain zero).
- While many interval algorithms are well known, there are many problems that we don’t know how to effectively solve with interval arithmetic. It isn’t a general solution.

Outline

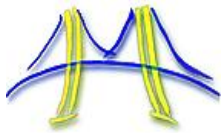
- What is the problem?
- Solutions
  - Use so many bits you can pretend there is no problem
  - Change how we model real arithmetic on computers
  - Use only the bits you need
  - Let the hardware solve the problem
- Conclusion

How many bits do we really need?



J.Y.F. Tong, D. Nagle, and R. Rutenbar, "Reducing Power by Optimizing the Necessary Precision Range of Floating Point Arithmetic," in *IEEE Transactions on VLSI systems*, Vol. 8, No.3, pp 273-286, June 2000. [2] M. Stevenson, J. Babb,

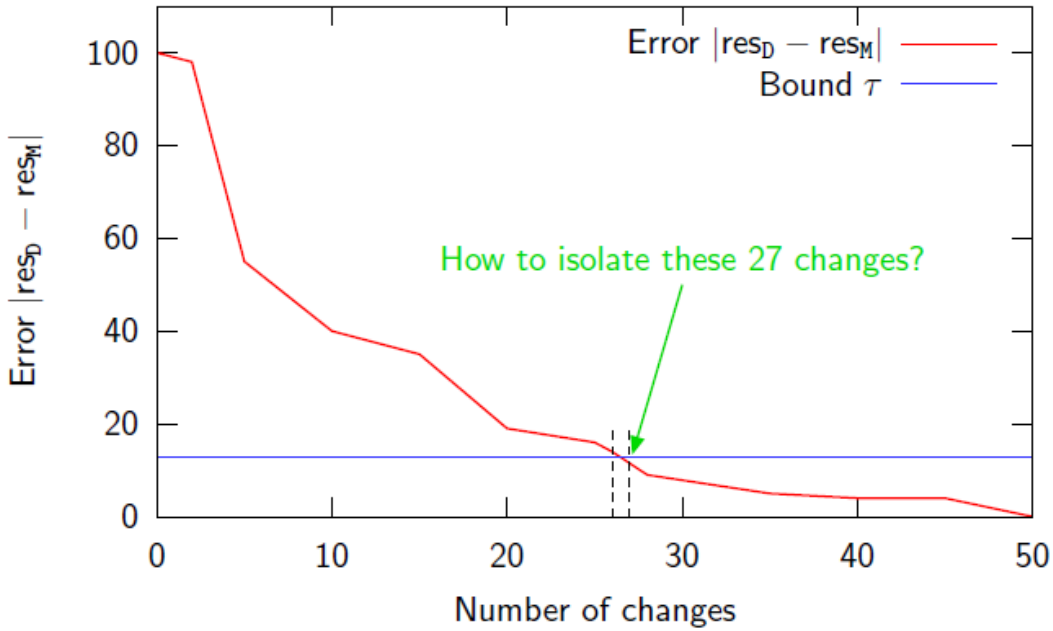
Selectively reducing precision … without armies of numerical analysts doing the work "by hand"



General principle of Delta-Debugging

- Principle: find a locally minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)

Adapt methods from "delta debugging" to automatically find where reduced precision can be used (Koushik Sen's group at UC Berkeley).



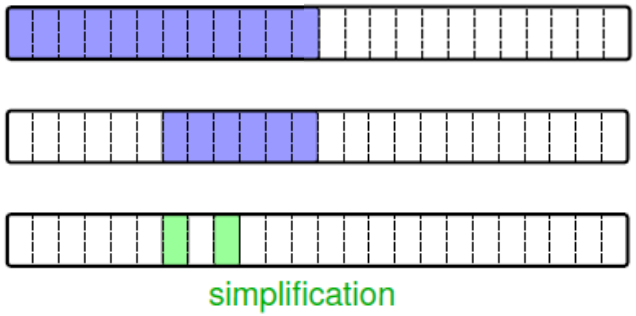
Source: Techniques for the automatic debugging of scientific floating point programs, Bailey, Demmel, Kahan, Revy, Sen, SCAN'2010, Lyon France 2010

Search procedures

Source: Techniques for the automatic debugging of scientific floating point programs, Bailey, Demmel, Kahan, Revy, Sen, SCAN'2010, Lyon France 2010

General principle of Delta-Debugging

- Principle: find a locally minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, higher precision, ...)
  - implementation like binary search



↪ apply half the changes and check if the output is still accurate

↪ if no, go back to the other state and discard the other half

↪ if the input is still inaccurate, increase the granularity of the splitting

  - the changes may be not consecutive

Current transformations:
- Float → double
- Double → double-double
- Rounding: rn→(ru, rd, rz)

Manage precision for performance, correctness and power

Source: Techniques for the automatic debugging of scientific floating point programs, Bailey, Demmel, Kahan, Revy, Sen, SCAN'2010, Lyon France 2010

More realistic example (D.H. Bailey)

- Calculate the arc length of the function g defined as

$$g(x) = x + \sum_{0 \le k \le 5} 2^{-k} \sin(2^k \cdot x), \quad \text{over } (0, \pi).$$

- Summing for $x_k \in (0, \pi)$ divided into n subintervals

$$\sqrt{h^2 + (g(x_k + h) - g(h))^2},$$

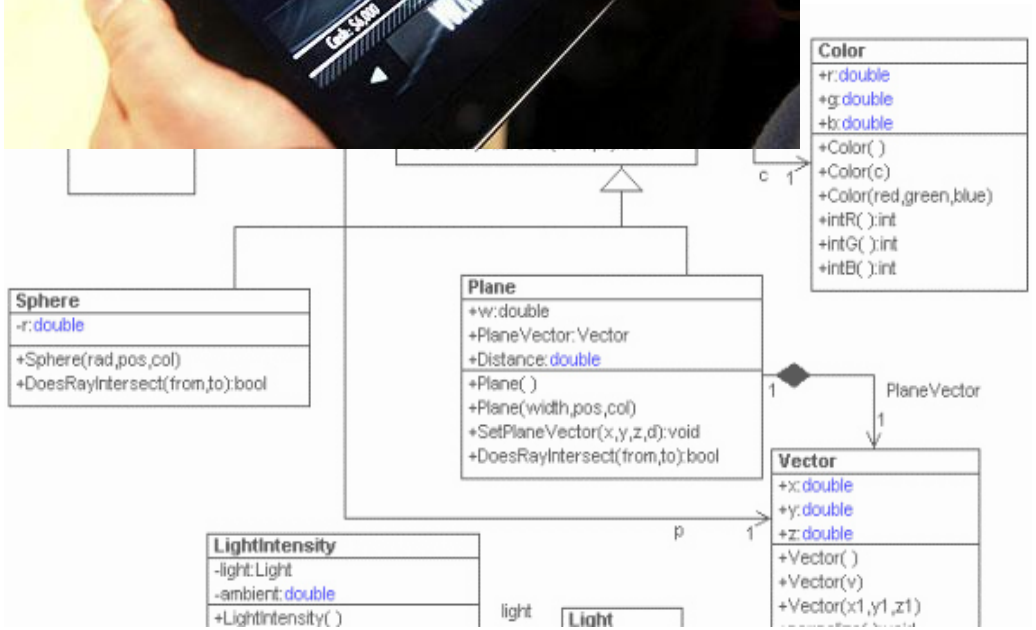
with $h = \pi/n$ and $x_k = k \cdot h$. If $n = 1000000$, we have

result = 5.795776322412856 (all double-double) → 20x slower
= 5.795776322413031 (all double)
= 5.795776322412856 (only the summand is in double-double)

- only 1 change is necessary: found in ≈ 30 sec.

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

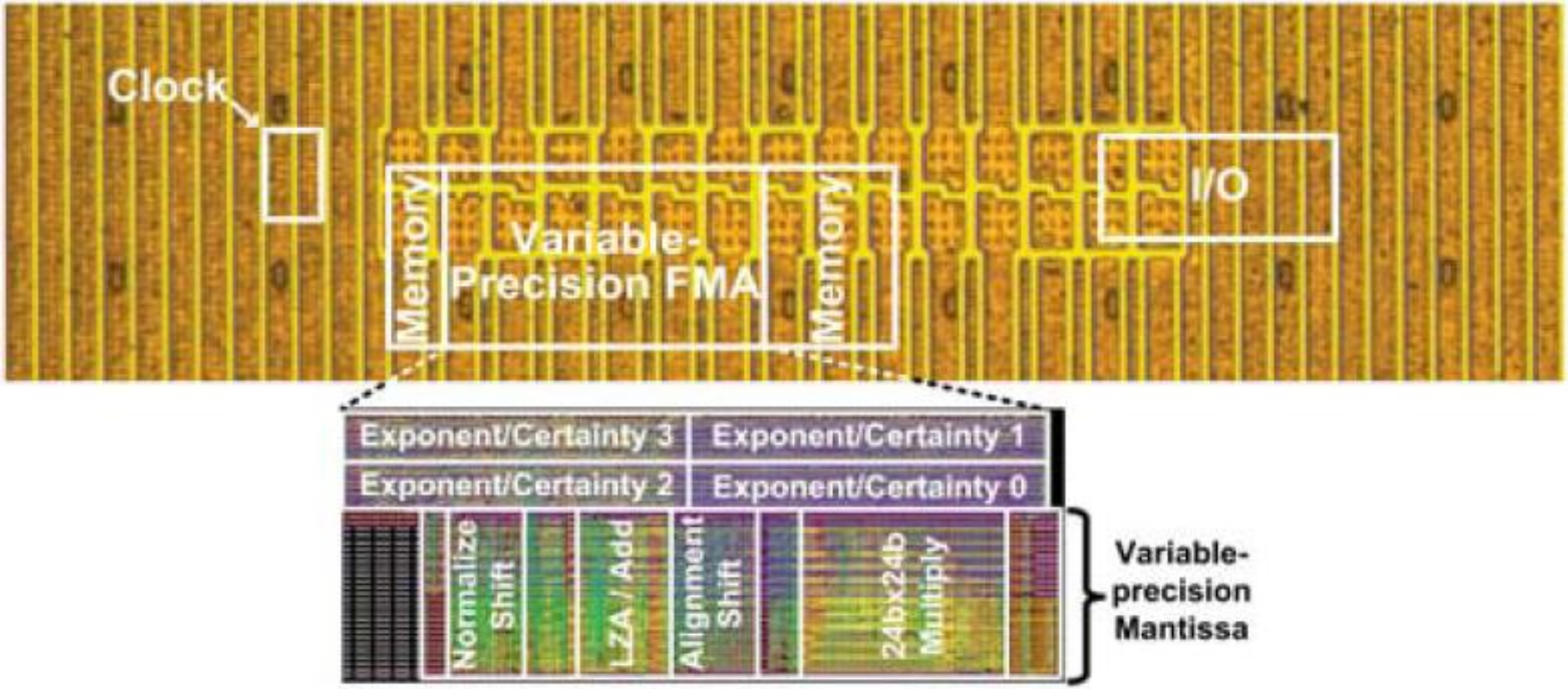


Outline

- What is the problem?
- Solutions
  - Use so many bits you can pretend there is no problem
  - Change how we model real arithmetic on computers
  - Use only the bits you need
  - Let the hardware solve the problem
- Conclusion

Hardware support for assured accuracy

- An Intel labs chip … supports variable precision math with uncertainty tracking. Can use with software that runs at low precision, tracks accuracy and reruns computations automatically if the error grow too large.



H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, S. Borkar, "A 1.45 GHz 52-162 GFLOPR./S variable-0precision floating-point fused multiply-add unit with certainty tracking in 32 nm CMOS", ICCSS, p. 182, 2012

Computing the FMA and uncertainty

- How does the chip define the FMA/uncertainty computation?
- Operands A, B and C (A*B+C) with their uncertainties. ΔR is a rounding error.

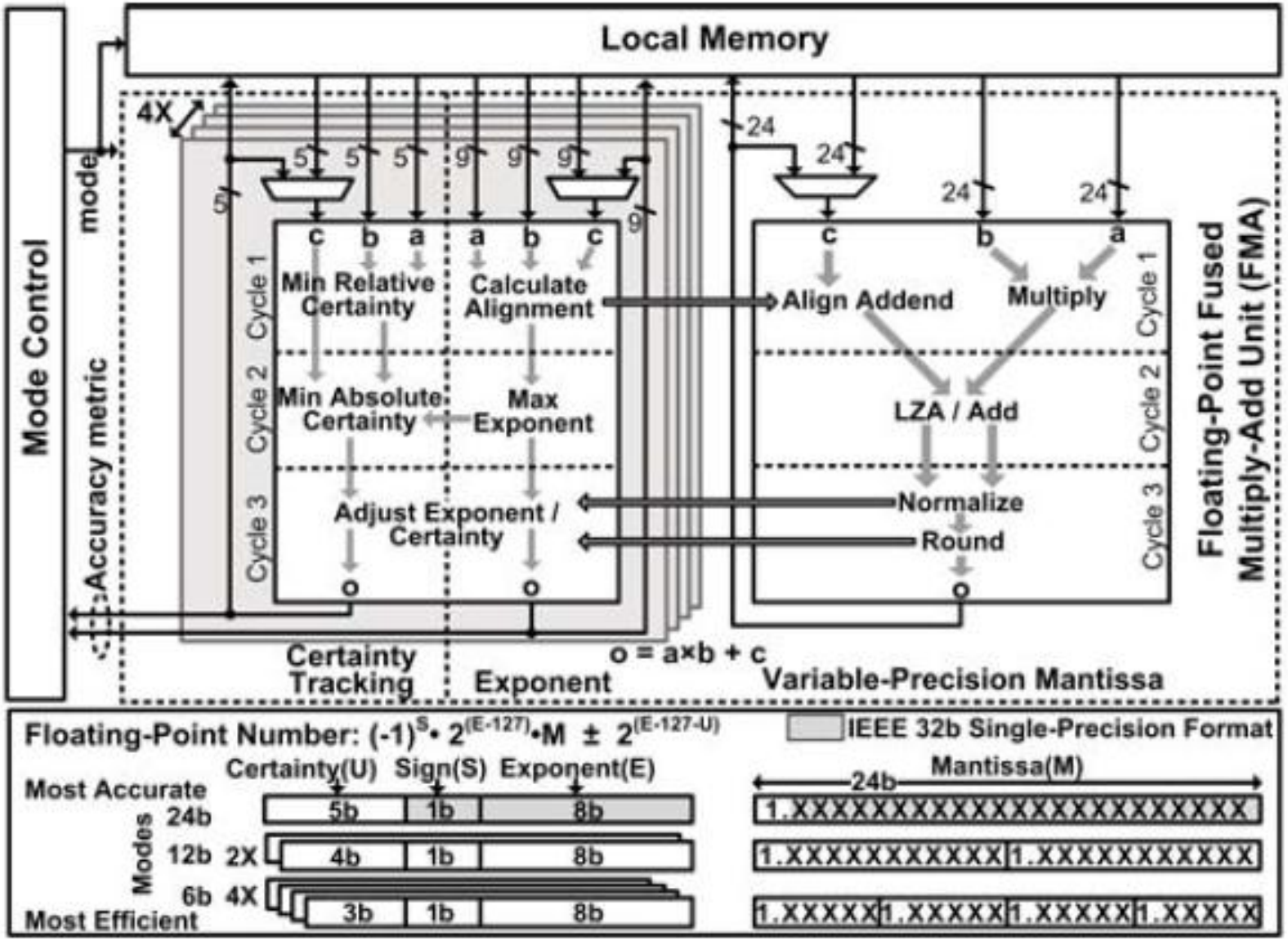
$$O \pm \Delta O = (A \pm \Delta A) * (B \pm \Delta B) + (C \pm \Delta C) \pm \Delta R$$

- Expand the above and gather terms to find O and ΔO .

$$O \pm \Delta O = \underbrace{(A * B + C}_{O} + (\Delta A * \Delta B) \pm \underbrace{(B * \Delta A + A * \Delta B + \Delta C + \Delta R)}_{\Delta O}$$

FMA “recentered” to account for the error term

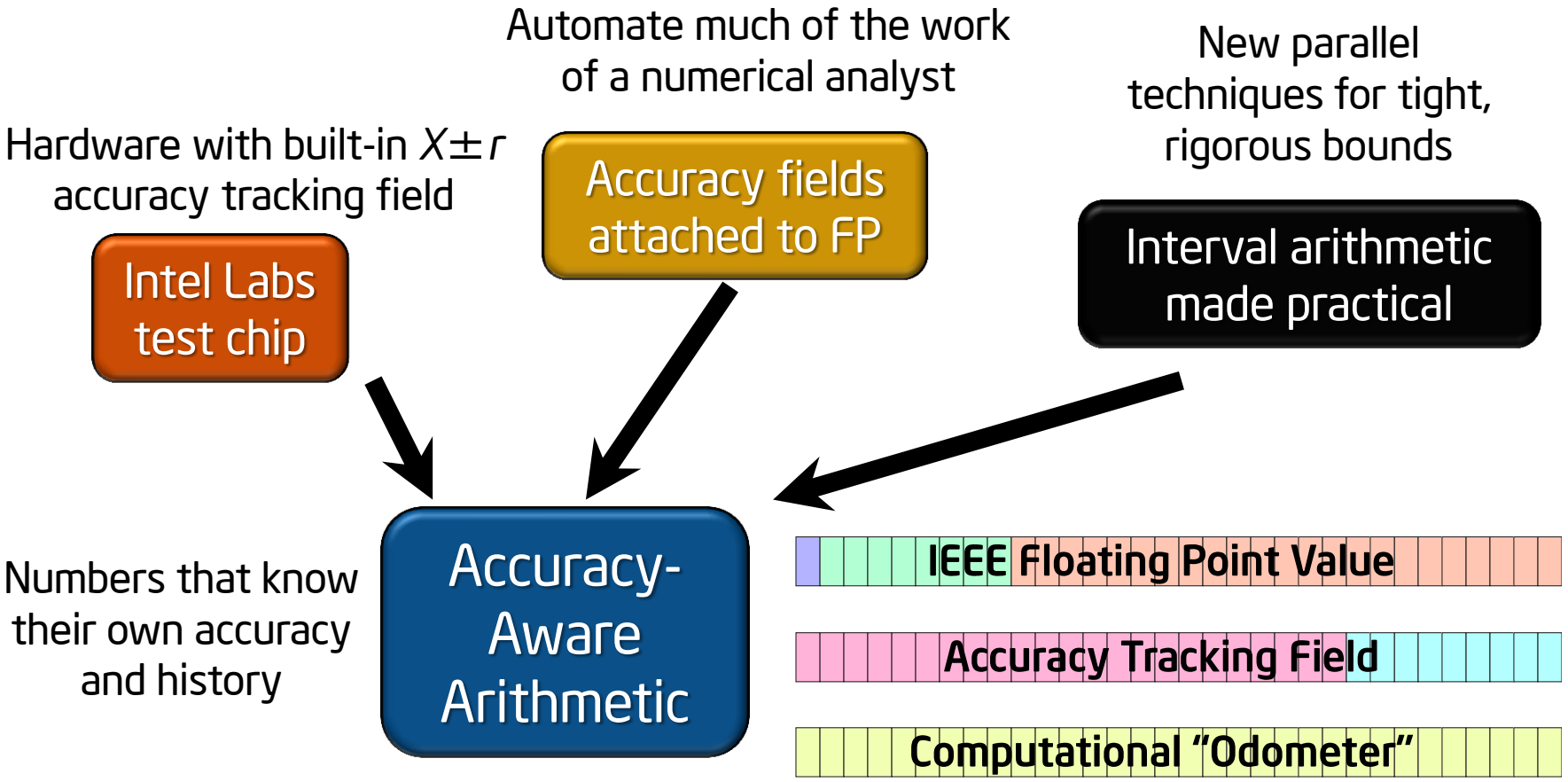
A variable precision FMA with "Certainty tracking"



Outline

- What is the problem?
- Solutions
  - Use so many bits you can pretend there is no problem
  - Change how we model real arithmetic on computers
  - Use only the bits you need
  - Let the hardware solve the problem
- Conclusion

None of these ideas alone solves the problem in all cases, but maybe if we combined them into a single integrated system, we'd have a solution?



Summary

- We do not know if results from our floating point intensive applications are correct.
  - We could know … IEEE 754™ combined with good numerical analysis can solve the problem, but programmers don't know numerical analysis and nothing suggests this will change.
- We need to rethink how we use floating point arithmetic and create the right tools to support assured accuracy.
- Assured accuracy has many benefits
  - Knowing (not guessing and hoping) that are programs are correct.
  - If we know how many bits we need, maybe we can use less … and less bits saves energy

Third party names are the property of their owners