Floating Point Arithmetic is not Real

Tim Mattson

Git clone https://github.com/infn-esc/esc23.git

Exercises in esc23/hands-on/flop

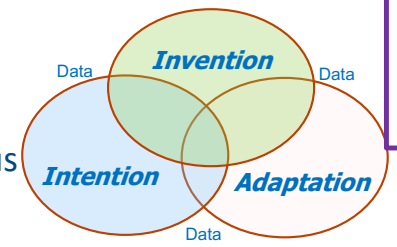
My 40 year Career

[tile]DB
2015
Sparse Array storage engine

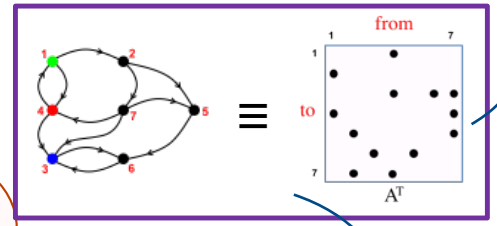

2015
BigDAWG Polystore system

2018-Aug'2023

oneAPI
Future of heterogenous computing



Machine Programming

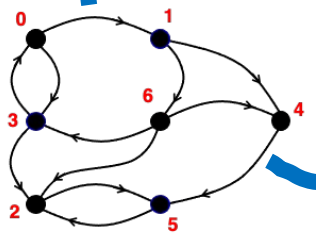


Unified Theory of data and computing

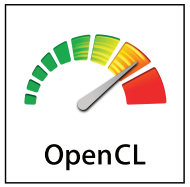
2023-2038
Retirement

Tim Mattson
RIP
1958-2038
Here lies the Christmas bat

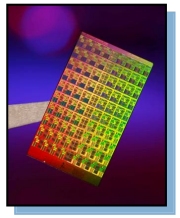
Write, teach, kayak, and think



2013
GraphBLAS

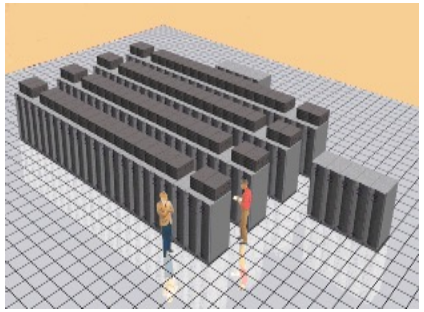


2008
Portable GPU programming


2007
World's First TFLOP chip

2002-2005
Director of life sciences

OpenMP
1997
Portable multithreading


1996
ASCI Red:
World's first TFLOP

PhD chemistry
1979-1985

PostDoc 1986
Caltech Concurrent Computation Project

Jobs with startups …
numerical analysis, signal processing,
scientific computing, and parallel computing
1987-1990

1990-1993
Linda and pre-MPI message passing

intel
1993
Supercomputing Systems Division

MPI
1994
Message Passing Interface

1994
FDIV Bug

“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

Arithmetic on computers

- We do arithmetic on a computer and expect it to work the same as doing arithmetic “by hand”.

- The details of how we do arithmetic on computers and the branch of mathematics that studies the consequences of computer arithmetic (numerical analysis) is fundamentally boring.
  - Even professionals who work on computer arithmetic (other than W. Kahan*) admit (maybe only in private) that its boring.

> 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

- It’s fine to take floating point arithmetic for granted … until something breaks.

*Professor William Kahan of UC Berkeley is the father of modern floating point arithmetic (IEEE-754)

Exercise: Tracking time in a digital system

- You are a software engineer working on a device that tracks objects in time and space.
- The device increments time in “clock ticks” of 0.01 seconds.
- Write a simple program that tracks time for a large number of clock-ticks and then outputs the actual time as a float. Assume you are working with an embedded processor that does not support the type double.
- Does your program work?

Real numbers on a computer are represented as Floating Point numbers

- Many decimal numbers do not have an exact representation as floating point numbers.

```
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.

Real numbers on a computer are represented as Floating Point numbers

- Many decimal numbers do not have an exact representation as floating point numbers.


```
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.

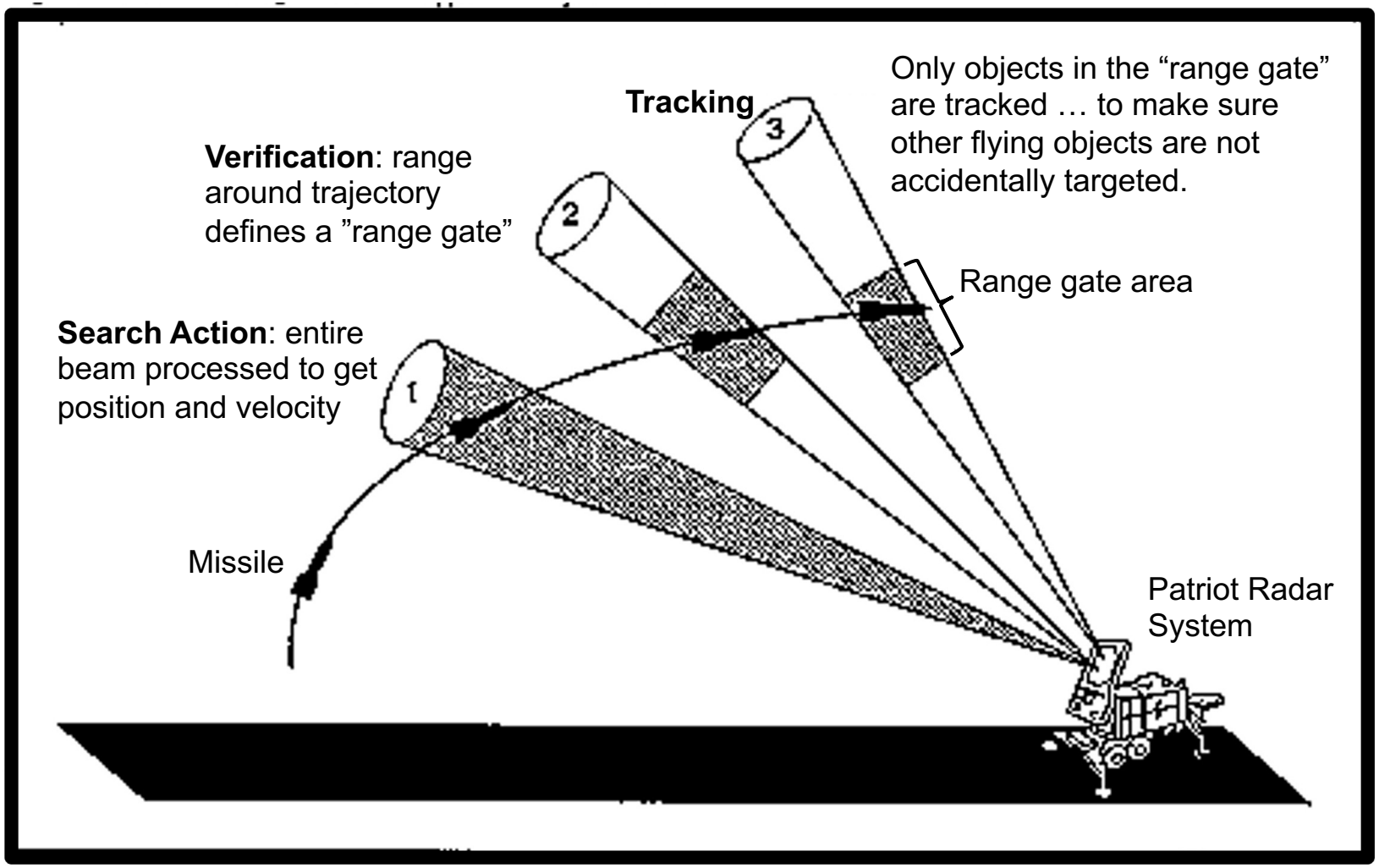
Patriot Missile system

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

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

Patriot missile system: how it works

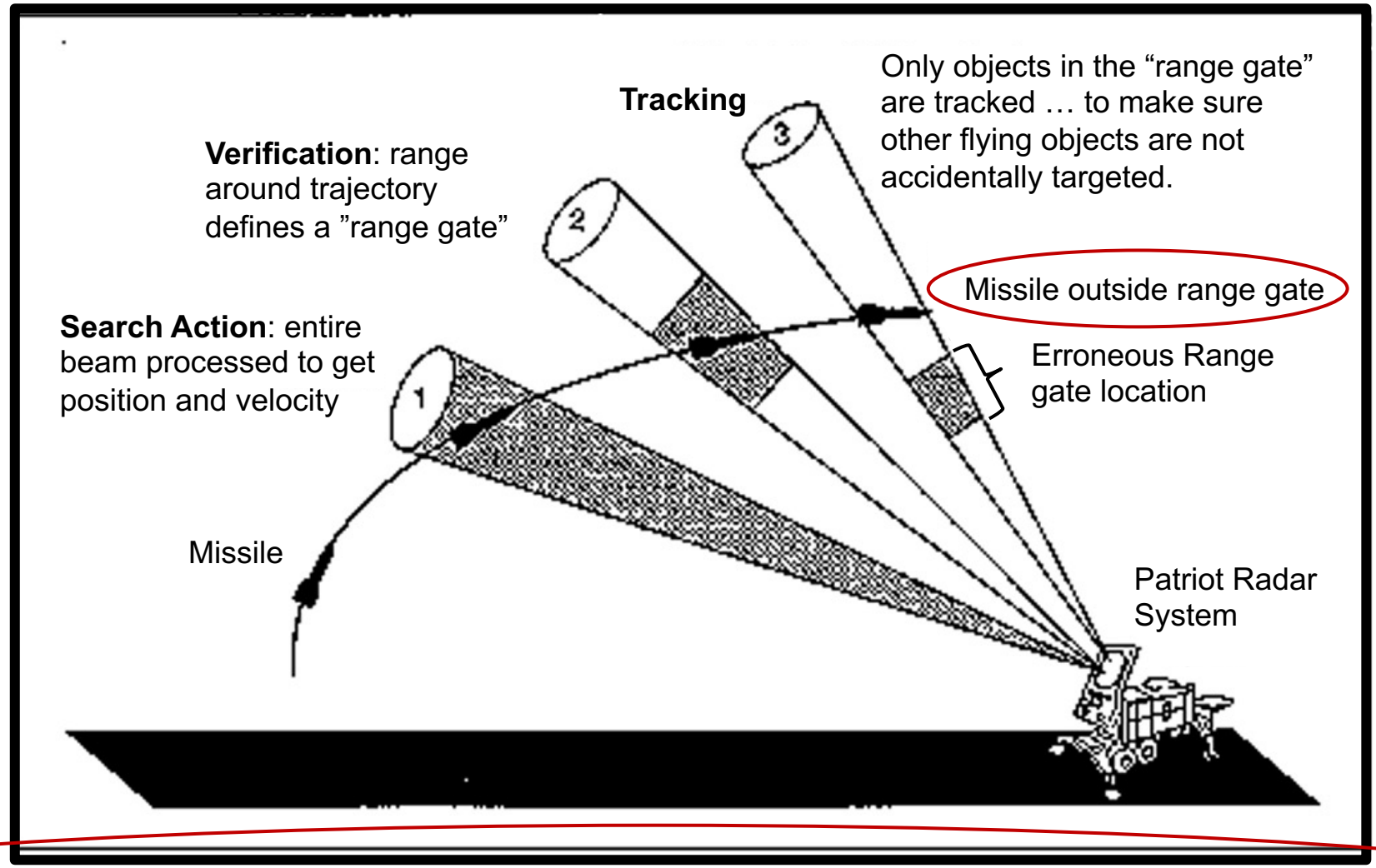
Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



24 bit clock counter defines time. Range calculations defined by real arithmetic, so convert to real (i.e. floating point) numbers.

Patriot missile system: Disaster Strikes

Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



Multiplication of clock-ticks (int) by the float representation of 0.01 led to an error of 0.3433 seconds after 100 hours of operation which, when you are trying to hit a missile moving at mach 5, corresponds to an error of 687 meters

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set

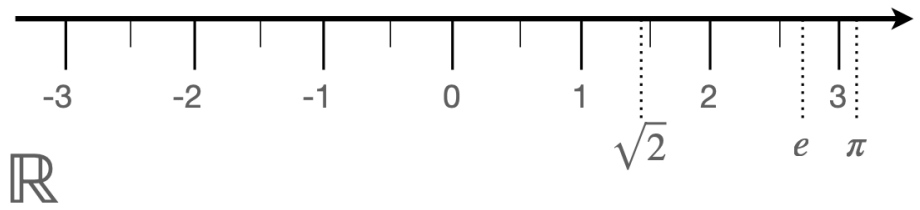
A quick introduction to floating point numbers

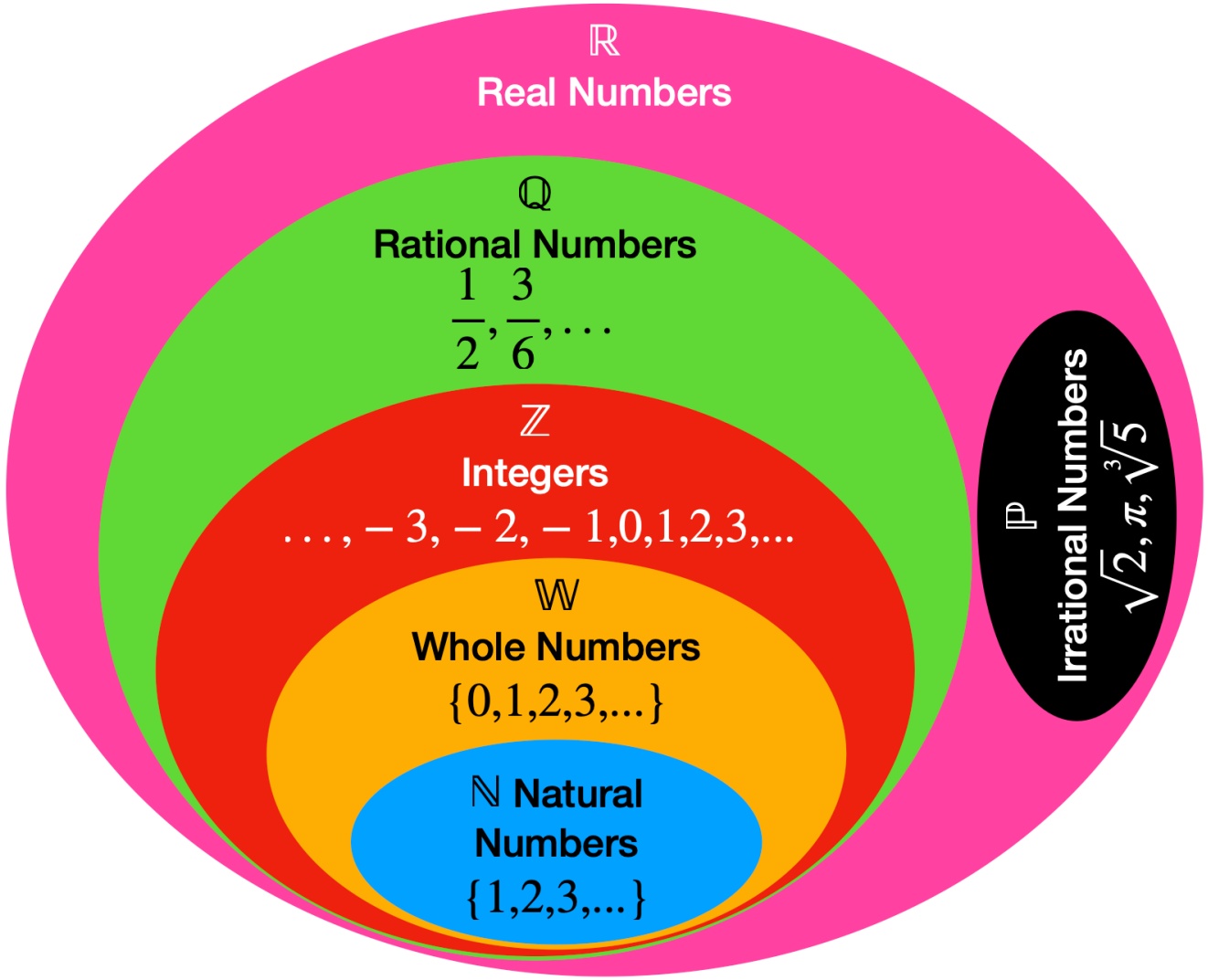
PRINCETON UNIVERSITY

iris hep

Real Numbers

in mathematics





Real numbers can be thought of as all points on a line called the number line or real line, where the points corresponding to integers are equally spaced

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Scientific Notation

real number representation

significand exponent

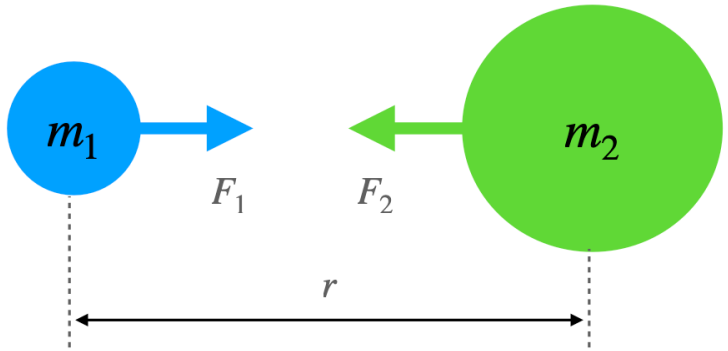
$$G \approx 6.674 \times 10^{-11} m^3 \cdot kg^{-1} \cdot s^{-2}$$

radix

The above expression means

$$0.00000000006674 \text{ or } 6.674e-11$$

the exponent shows by how far to "float" the decimal point



$$F_1 = F_2 = G\frac{m_1 \times m_2}{r^2}$$

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

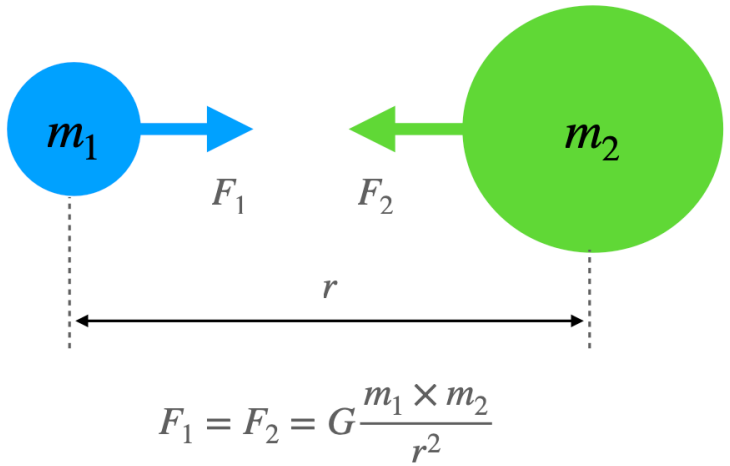
Scientific Notation

real number representation

A real number can be represented by

$$x = (-1)^s \sum_{i=0}^{\infty} d_i b^{-i} \cdot b^e,$$

where $s \in \{0,1\}$, $b \geq 2$, $i \in \{0,1,2,...\}$, $d_i \in \{0,...,b-1\}$
and $d_0 > 0$ when $x \neq 0$, b and e are integers



For example: $G \approx 6.674 \times 10^{-11} m^3 \cdot kg^{-1} \cdot s^{-2}$

$$(6 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2} + 4 \times 10^{-3}) \times 10^{-11}$$

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Sets of Floating-point Data

finite computer representation

- Floating-point number system is defined by the four natural numbers:
  - $b \geq 2$, the radix, (2 or 10)
  - $p \geq 1$, the precision (the number of of digits in the significand)
  - e_{max}, the largest possible exponent
  - e_{min}, the smallest possible exponent, (shall be $1 - e_{max}$ for all formats)
- Notation:

$$F(b, p, e_{min}, e_{max})$$

In computing, the binary (base-2), octal (base-8) and hexadecimal (base-16) bases are most commonly used.

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Floating-point Number Systems

a finite subset of $\mathbb{R}$

The set $F(b, p, e_{min}, e_{max})$ of real numbers represented by this system consists of all floating-point numbers of the form:

$$(-1)^s \sum_{i=0}^{p-1} d_i b^{-i} \cdot b^e,$$

$s \in \{0,1\}$, $d_i \in \{0,...,b-1\}$ for all i, $e \in \{e_{min}, \ldots, e_{max}\}$.

represented in radix b:

$$\pm d_0 . d_1 \ldots d_{p-1} \times b^e,$$

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Floating-point Number Systems

Representations of the decimal number 0.1 (with $b = 10$)

$1.0 \times 10^{-1}, 0.1 \times 10^{0}, 0.01 \times 10^{1}, \ldots$

Different representations due to choice of exponent

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Normalized Representation

Normalized number:

$$\pm d_0 . d_1 \ldots d_{p-1} \times b^e, d_0 \neq 0$$

The normalized representation is unique and therefore preferred

The number 0, as well as all numbers smaller than $b^{e_{min}}$, have no normalized representation

Set of normalized numbers:

$$F^*(b, p, e_{min}, e_{max})$$

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Quiz:

How many floating point numbers do the systems $F^*(b, p, e_{min}, e_{max})$ and $F(b, p, e_{min}, e_{max})$ contain?

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Quiz:

How many floating point numbers do the systems $F^*(b, p, e_{min}, e_{max})$ and $F(b, p, e_{min}, e_{max})$ contain?

For each exponent, $F^*(b, p, e_{min}, e_{max})$ has $b - 1$ possibilities for the first digit, and b possibilities for the remaining $p - 1$digits

The size of $F^*(b, p, e_{min}, e_{max})$ is therefore

$$2(e_{max} - e_{min} + 1)(b - 1)b^{p-1},$$

if we take the two possible signs into account.

$F(b, p, e_{min}, e_{max})$ has extra nonnegative numbers of the form

$$0.d_1 \ldots d_{p-1} 2^{e_{min}},$$

and there are b^{p-1}. Adding the non-positive ones and subtracting 1 for counting 0 twice, we get

$2b^{p-1} - 1$ extra numbers.

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

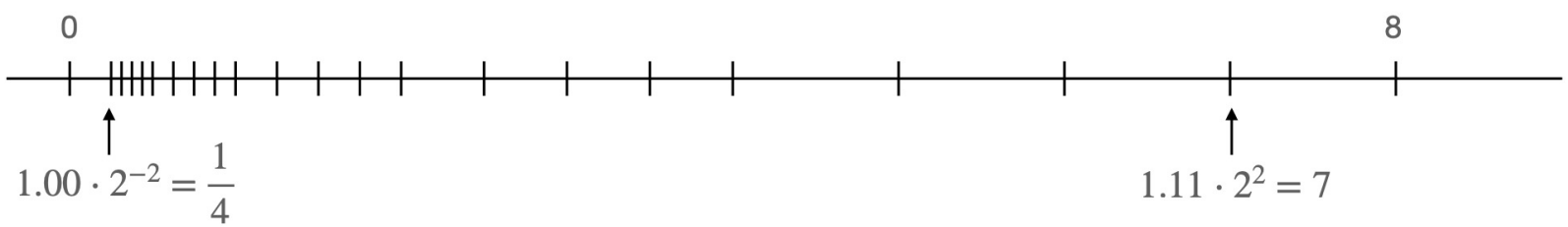
PRINCETON UNIVERSITY

iris hep

Normalized Representation

$F^*(2,3,-2,2)$

$d_0 . d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



F*(Radix, Precision, emin, emax)

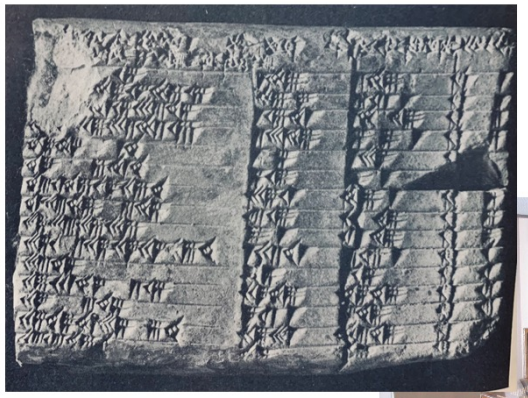
Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Floating-point Arithmetic Timeline

"…the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!"
https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/ieee-computer.pdf



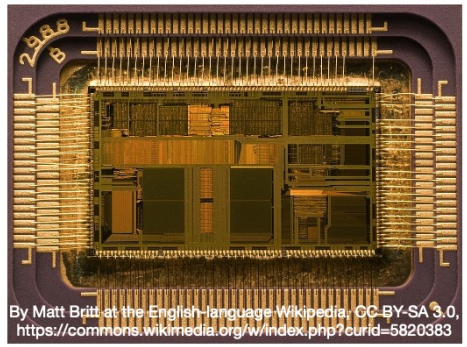
Babylonians worked with floating-point sexagesimal numbers

- Average calculation speed: addition – 0.8 seconds, multiplication – 3 seconds[1]
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root[1]

By Venusianer, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3632073

Intel 80486

the first tightly-pipelined[c] x86 design as well as the first x86 chip to include more than one million transistors. It offered a large on-chip cache and an integrated floating-point unit.



By Matt Britt at the English-language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5820383

The floating point unit fully conforms to the ANSI/ IEEE standard 754-1985 for floating point arithmetic.

"new kinds of computational demands might eventually encompass new kinds of standards, particularly for fields like artificial intelligence, machine vision and speech recognition, and machine learning. Some of these fields obtain greater accuracy by processing more data faster rather than by computing with more precision – rather different constraints from those for traditional scientific computing."

1750 B.C.

1941 Zeus Z3

1960's a mainframe era

1964 IBM 360
CDC 6600
DEC 10

1970's boom of minicomputers
DEC PDP-11

IEEE p745

1976 1977 Intel began to design a floating-point co-processor for its i8086/8 and i432 microprocessors

IEEE 754-1985

1985

1989

IEEE 754-1987

IEEE 754-2008

IEEE 754-2019 binary and decimal floating-point arithmetic

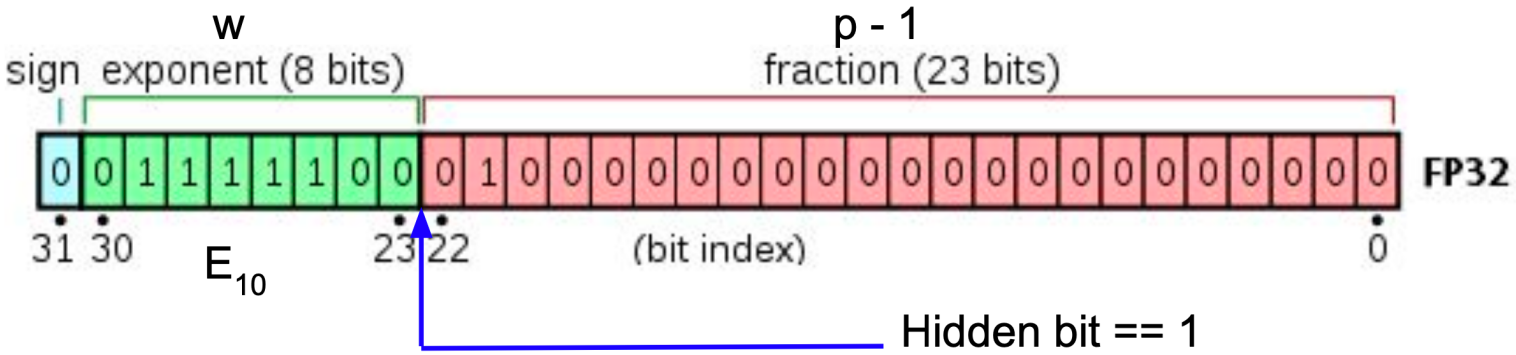
you are here

IEEE 754-2029 subjected to review at least every 10 years

- each hardware manufacturer had its own type of floating point
- different machines from the same manufacturer might have different types of floating point
- when floating point was not supported in the hardware, the different compilers emulated different floating point types

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

IEEE 754 Floating Point Numbers



IEEE Name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

Source: Wahid Redjeb, ESC'22

PRINCETON UNIVERSITY

iris hep

Exceptions

- The IEEE floating point standard defines several exceptions that occur when the result of a floating point operation is unclear or undesirable. Exceptions can be ignored, in which case some default action is taken, such as returning a special value. When trapping is enabled for an exception, an error is signaled whenever that exception occurs.
- Possible floating point exceptions:
  - **Underflow**: The result of an operation is too small to be represented as a normalized float in its format. If trapping is enabled, the floating-point-underflow condition is signaled. Otherwise, the operation results in a denormalized float or zero.
  - **Overflow**: The result of an operation is too large to be represented as a float in its format. If trapping is enabled, the floating-point-overflow exception is signaled. Otherwise, the operation results in the appropriate infinity.
  - **Divide-by-zero**: A float is divided by zero. If trapping is enabled, the divide-by-zero condition is signaled. Otherwise, the appropriate infinity is returned.
  - **Invalid**: The result of an operation is ill-defined, such as (0.0/0.0). If trapping is enabled, the floating-point-invalid condition is signaled. Otherwise, a quiet NaN is returned.
  - **Inexact**: The result of a floating point operation is not exact, i.e. the result was rounded. If trapping is enabled, the floating-point-inexact condition is signaled. Otherwise, the rounded result is returned.
- Trapping of these exceptions can be enabled through compiler flags, but be aware that the resulting code will run slower.

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Special values and related operations/exceptions

TABLE D-2 IEEE 754 Special Values

Exponent	Fraction	Represents
$e = e_{min} - 1$	$f = 0$	±0
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$
$e_{min} \leq e \leq e_{max}$	--	$1.f \times 2^e$
$e = e_{max} + 1$	$f = 0$	$\pm\infty$
$e = e_{max} + 1$	$f \neq 0$	NaN

TABLE D-3 Operations That Produce a NaN

Operation	NaN Produced By
+	$\infty + (-\infty)$
×	$0 \times \infty$
/	$0/0$, ∞/∞
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$ (when $x < 0$)

TABLE D-4 Exceptions in IEEE 754*

Exception	Result when traps disabled	Argument to trap handler
overflow	$\pm\infty$ or $\pm x_{max}$	round($x2^{-\alpha}$)
underflow	0, $\pm 2^{e_{min}}$ or denormal	round($x2^{\alpha}$)
divide by zero	$\pm\infty$	operands
invalid	NaN	operands
inexact	round(x)	round(x)

Walking Through Floating-point Numbers

- 0x0000000000000000 — +zero
- 0x0000000000000001 — smallest subnormal
- . . .
- 0x000fffffffffffff — largest subnormal
- 0x0010000000000000 — smallest normal
- . . .
- 0x001fffffffffffff
- 0x0020000000000000 — 2 X smallest normal

For the subnormal numbers (also called ”denormal”) the hidden bit is 0

Walking Through Floating-point Numbers

- 0x0020000000000000 — 2 X smallest normal
- . . .
- 0x7fefffffffffffff — largest normal
- 0x7ff0000000000000 — +infinity
- 0x7ff0000000000001 — NaN
- . . .
- 0x7fffffffffffffff — NaN
- 0x8000000000000000 — -zero

Not a Number

Walking Through Floating-point Numbers

- 0x8000000000000000 — -zero
- 0x8000000000000001 — "smallest" negative subnormal
- . . .
- 0x800fffffffffffff — "largest" negative subnormal
- 0x8010000000000000 — "smallest" negative normal
- . . .
- 0xfff0000000000000 — -infinity

The properties of operations over IEEE 754 floating point numbers

PRINCETON UNIVERSITY

iris hep

Correctly Rounded Arithmetic

- The IEEE standard requires that the result of addition, subtraction, multiplication and division be **exactly rounded**
  - Exactly rounded means the results are calculated exactly and then rounded. For example: assuming $p = 23$, $x = (1.00..00)_2 \times 2^0$ and $z = (1.00..01)_2 \times 2^{-25}$, then $x - z$ is

$$
\begin{array}{rl}
 & (\ 1.00000000000000000000000|\ )_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|01000000000000000000000001\ )_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|10111111111111111111111111\ )_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111|01111111111111111111110\ )_2 \times 2^{-1} \\
\text{Round to} & \\
\text{Nearest :} & (\ 1.11111111111111111111111\ )_2 \times 2^{-1}
\end{array}
$$

  - Compute the result exactly is very expensive if the operands differ greatly in size
  - The result of two or more arithmetic operations are NOT exactly rounded
- How is **correctly rounded** arithmetic implemented?
  - Using two additional guard bits plus one sticky bit guarantees that the result will be the same as computed using exactly rounded [Goldberg 1990]. The above example can be done as

$$
\begin{array}{rl}
 & (\ 1.00000000000000000000000|\ \ \ )_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|011\ )_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|101\ )_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111|01\ )_2 \times 2^{-1} \\
\text{Round to Nearest :} & (\ 1.11111111111111111111111\ )_2 \times 2^{-1}
\end{array}
$$

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

iris hep

Correctly Rounded Arithmetic

- The IEEE standard requires that the result of addition, subtraction, multiplication and division be **exactly rounded**
  - Exactly rounded means the results are calculated exactly and then rounded. For example: assuming $p = 23$, $x = (1.00..00)_2 \times 2^0$ and $z = (1.00..01)_2 \times 2^{-25}$, then $x - z$ is

$$
\begin{array}{rl}
 & (\ 1.00000000000000000000000 \mid\ )_2 \times 2^0 \\
- & (\ 0.00000000000000000000000 \mid 01000000000000000000000001\ )_2 \times 2^0 \\
= & (\ 0.11111111111111111111111 \mid 10111111111111111111111\ )_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111 \mid 01111111111111111111110\ )_2 \times 2^{-1} \\
\text{Round to} & \\
\text{Nearest :} & (\ 1.11111111111111111111111\ )_2 \times 2^{-1}
\end{array}
$$

  - Compute the result exactly is very expensive if the operands differ greatly in size
  - The result of two or more arithmetic operations are NOT exactly rounded

The standard also requires exact rounding for:
- Square root
- Remainder
- Fused multiply/Add
- Conversion between integer and floating point

But NOT:
- Conversion between decimal and floating point.

- How is **correctly rounded** arithmetic implemented?
  - Using two additional guard bits plus one sticky bit guarantees that the result will be the same as computed using exactly rounded [Goldberg 1990]. The above example can be done as

$$
\begin{array}{rl}
 & (\ 1.00000000000000000000000 \mid\ \ )_2 \times 2^0 \\
- & (\ 0.00000000000000000000000 \mid 011\ )_2 \times 2^0 \\
= & (\ 0.11111111111111111111111 \mid 101\ )_2 \times 2^0 \\
\text{Normalize :} & (\ 1.11111111111111111111111 \mid 01\ )_2 \times 2^{-1} \\
\text{Round to Nearest :} & (\ 1.11111111111111111111111\ )_2 \times 2^{-1}
\end{array}
$$

PRINCETON UNIVERSITY

iris hep

Rounding

absolute and relative rounding error

- A positive REAL number in the *normalized range* ($x_{min} \leq x \leq x_{max}$) can be represented as

$$(x)_2 = (1 \,.\, b_1 b_2 \ldots b_{p-1} \ldots) \times 2^e,$$

where $x_{min}(=2^{e_{min}})$ and $x_{max}(= (1 - 2^{-p})2^{e_{max}+1})$ are the smallest and largest normalized floating point numbers. (Subscript 2 for binary representation is omitted since now.)

- The nearest floating point number less than or equal to x is

$$x_- = (1.b_1 b_2 \ldots b_{p-1}) \times 2^e$$

- The nearest floating point number larger than x is

$$x_+ = (1.b_1 b_2 \ldots b_{p-1} + 0\,.\,00\ldots1) \times 2^e$$

- The gap between x_+ and x_-, called **unit in the last place** (ulp) is

$$2^{-(p-1)}2^e$$

- The **absolute rounding error** is

$$abserr(x) = \left| round(x) - x \right| < 2^{-(p-1)}2^e = \text{ulp}$$

- The **relative rounding error** is

$$relerr(x) = \frac{|round(x) - x|}{|x|} < \frac{2^{-(p-1)}2^e}{2^e} = 2^{-(p-1)} = \varepsilon$$

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Relative Errors and Ulps

- Rounding error is inherent in floating-point computation
- How do we measure this error.
  - Consider radix 10 floating-point format (decimal) numbers with three digits.
    - If the result of a floating-point computation is 3.12×10^{-2}, and the answer when computed to infinite precision is .0314, it is clear that this is in error by 2 units in the last place.
    - if the real number .0314159 is represented as 3.14×10^{-2}, then it is in error by .159 units in the last place.
    - We define "unit in the last place" by the acronym "ulp"
  - Continuing with radix 10 numbers with three digits
    - If the result using real arithmetic is 3.14159 is approximated with floating point at 3.14, we an define the *relative error* as the difference between the real and the floating point results divided by the real result …. (3.14159 = 3.14)/3.14159 = 0.0005.
- When a result from an operation is carried out to produce the correct result (as determined by real arithmetic) and rounded to the nearest floating point number, the error can be no larger than 0.5 ulp

IEEE 754 rounding modes

The 4 rounding modes in IEEE 754

Default rounding mode

rounding mode	C name
to nearest	FE_TONEAREST
toward zero	FE_TOWARDZERO
to +infinity	FE_UPWARD
to -infinity	FE_DOWNWARD

Two versions …
- Nearest, on a tie, round to even
- Nearest, on a tie, away from zero

Three directed roundings

C

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = fegetround( );

// establish the desired rounding mode
fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
fesetround(originalRounding);
```

C++

```
#include <cfenv>
#pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = std::fegetround( );

// establish the desired rounding mode
std::fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
std::fesetround(originalRounding);
```

Exercise

- We provide two programs, one in C and one in C++, that demonstrate how to change rounding modes.
  - roundC.c. and roundCpp.cc

- Experiment with these programs just to confirm that you know how the rounding modes work and to verify that you can manipulate them.

- If you have time, use them in other programs you might have, to see if you can see any impact from the different rounding modes.

```
rounding mode     C name
-------------------------------
to nearest        FE_TONEAREST
toward zero       FE_TOWARDZERO
to +infinity      FE_UPWARD
to -infinity      FE_DOWNWARD
```

You must be careful how you manage rounding…

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.

Condition numbers

- Given a `x` number you want to compute `f(x) = y`
  - But there is the rounding in place
    - Most of the time you have ẍ `= x + Δx`
    - `f(x + Δx) = y + Δy`
- We can compute the following number

$$C = \frac{|\frac{\Delta y}{y}|}{|\frac{\Delta x}{x}|} = \frac{|x \cdot f'(x)|}{|f(x)|}$$

Example:

$$ln(x) \quad \text{with} \quad x = 1$$

$$C = \frac{1}{lnx} \longrightarrow \infty$$

- Small condition number means:
  - Small `Δx` produces small `Δy`
  - **Well Conditioned**
- Big condition number means:
  - Small `Δx` produces big `Δy`
  - **Ill conditioned**

Source: Wahid Redjeb, ESC'22

When moving beyond basic operations, however, you may not get exactly rounded results

- Floating point numbers are:
  - Commutative: $A * B = B * A$
  - NOT Associative: $A * (C * B) \neq (A * C) * B$
  - NOT Distributive: $A*(B+C) \neq A*B + A*C$

- For example, consider addition. To add a pair of numbers, you must shift the smaller number of the pair to use the same exponent then combine them.
  - This can cause you to loose digits from the smaller number. Hence it is dangerous to sum numbers that are of greatly different magnitudes.
  - When summing a sequence of numbers, you can get different results depending on the order of the operations … i.e. floating point operations are not associative

Exercise: Summation with floating point arithmetic

- We have provided a C program called sum.c

- In the program, we generate a sequence of floating point numbers (all greater than zero). Don't look at how we create that sequence … treat the sequence generator as a black box (in other words, just work on the sequence, don't use knowledge of how it was generated).

- Write code to sum the sequence of numbers. You can compare your result to our estimate of the correct result.
  - Only use float types (it's cheating to use double … at least to start with).
  - Using what you know about floating point arithmetic, is there anything you can think of doing to improve the quality of your sum?

Exercise: Summation with floating point arithmetic

- Possible solutions to try.
  - Use a double for the accumulator. Does that make any difference?

- Look at the way the sequence is generated and who the ”correct value’ was produced.
  - Is it a problem if we add two numbers that have widely different magnitudes?
  - What if you sort the array before the summation?

Adding floating point numbers

- Lets keep things simple and work with F*(10, 3, -2, 2)

- Find the sum … $1.23 \times 10^{1} + 3.11 \times 10^{-1}$
  - Align smaller number to the exponent of the larger number

    0.0311×10^{1}
  - Add the two aligned numbers …..

$$\begin{array}{llllll} 1 & . & 2 & 3 & & \times 10^{1} \\ 0 & . & 0 & 3 & 1 & 1 \quad \times 10^{1} \\ 1 & . & 2 & 6 & 1 & 1 \quad \times 10^{1} \end{array}$$

  - Round to nearesgt (the default rounding in IEEE 754).

$$1 \quad . \quad 2 \quad 6 \quad \times 10^{1}$$

  - Adding numbers with greatly different magnitudes causes loss of the low order bits from the exact result.

F*(Radix, Precision, emin, emax)

The Summation problem: Numerical Analysis to the rescue

- The branch of mathematics that studies the consequences of computer arithmetic is called numerical analysis. (and yes, it is fundamentally boring … and don't tell Professor Kahan that I said that)..

- Kahan (and collaborators … there are many forms of this algorithm) came up with a method to sum sequences of numbers without loosing so much precision.

*Professor William Kahan of UC Berkeley is the father of modern floating point arithmetic (IEEE-754)

Exercise: The Kahan Summation Algorithm

- Implement this algorithm in your summation program. Does it improve the sum.
- Spend some time playing with the code to understand EXACTLY how it works.

```
Input: a sequence of N values, x[i]for i=1,N

   correction = 0.0
   sum = 0.0

   for i = 1 to N:

      xcor = x[i] – correction

      tmpSum =  sum + xcor

      correction = (tmpSum – sum) – xcor

      sum = tmpSum
   }
Output: sum
```

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.

PRINCETON UNIVERSITY

iris hep

Cancellation

- Cancellation occurs when we subtract two almost equal numbers
- The consequence is the error could be much larger than the machine epsilon
- For example, consider two numbers

$$x = 3.141592653589793 \text{ (16-digit approximation to } \pi)$$

$$y = 3.141592653585682 \text{ (12-digit approximation to } \pi)$$

Their difference is

$$z = x - y = 0.000000000004111 = 4.111\text{x}10^{-12}$$

In a C program, if we store x, y in single precision and display z in single precision, the difference is

0.000000e+00 Complete loss of accuracy

If we store x, y in double precision and display z in double precision, the difference is

4.110933815582030e-12 Partial loss of accuracy

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Solution to the quadratic equation.

- Consider a quadratic equation.

$$ax^2 + bx + c = 0$$

- Using real arithmetic the solution is.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Take a look at the program quad.c and run it for a few values of a, b, and c.
- Try the case:

a = .0005, b =. 100 and c = 0.005

- Can you explain the result?

Solution to the quadratic equation.

- Consider a quadratic equation.

$$ax^2 + bx + c = 0$$

- Using real arithmetic the solution is.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Two sources of cancelation

Can you algebraically transform the above equation to reduce the impact of the cancelation?

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.
With arbitrary precision, there is no loss of accuracy when subtracting real numbers	Subtracting two numbers of similar size cancels higher order bits

Floating point arithmetic and your compiler

PRINCETON UNIVERSITY

iris hep

Optimization levels and related options

- GCC has a rich optimization pipeline that is controlled by approximately a hundred command line options
- The default is to not optimize. You can specify this optimization level on the command line as -O0. It is often used when developing and debugging a project. This means it is usually accompanied with the command line switch -g so that debug information is emitted. As no optimizations take place, no information is lost because of it. No variables are optimized away, the compiler only inlines functions with special attributes that require it, and so on. As a consequence, the debugger can almost always find everything it searches for in the running program and report on its state very well. On the other hand, the resulting code is big and slow
- The most common optimization level for release builds is -O2 which attempts to optimize the code aggressively but avoids large compile times and excessive code growth.
- Optimization level -O3 instructs GCC to simply optimize as much as possible, even if the resulting code might be considerably bigger and the compilation can take longer.
- Note that neither -O2 nor -O3 imply anything about the precision and semantics of floating-point operations. Even at the optimization level -O3 GCC implements math functions so that they strictly follow the respective IEEE and/or ISO rules. This often means that the compiled programs run markedly slower than necessary if such strict adherence is not required. The command line switch -ffast-math is a common way to relax rules governing floating-point operations.
- The most aggressive optimization level is `-Ofast` which does imply `-ffast-math` along with a few options that disregard strict standard compliance. In GCC 11 this level also means the optimizers may introduce data races when moving memory stores which may not be safe for multithreaded applications. Additionally, the Fortran compiler can take advantage of associativity of math operations even across parentheses and convert big memory allocations on the heap to allocations on stack. The last mentioned transformation may cause the code to violate maximum stack size allowed by **`ulimit`** which is then reported to the user as a segmentation fault.

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Compilers options!

- **There are many compiler options which affect floating point results!**
- Some of them can be enabled/disabled by other options!
- Different compilers might have different options!

- gcc default mode is “Strict IEEE 754 mode”
- `-O1, -O2, -O3, -Ofast , -ffast-math, -funsafe-math-optimizations`

https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Optimize-Options.html

Tool for inspecting assembly code
http://gcc.godbolt.org

Source: Wahid Redjeb, ESC’22

PRINCETON UNIVERSITY

iris hep

Optimization level recommendation

- Usually we(*) recommend using -O2, because at this level the compiler makes balanced size and speed trade-offs when building a general-purpose operating system
- However, we suggest using -O3 if you know that your project is compute-intensive and is either small or an important part of your actual workload
- Moreover, if the compiled code contains performance-critical floating-point operations, we strongly advise that you investigate whether -ffast-math or any of the fine-grained options it implies can be safely used

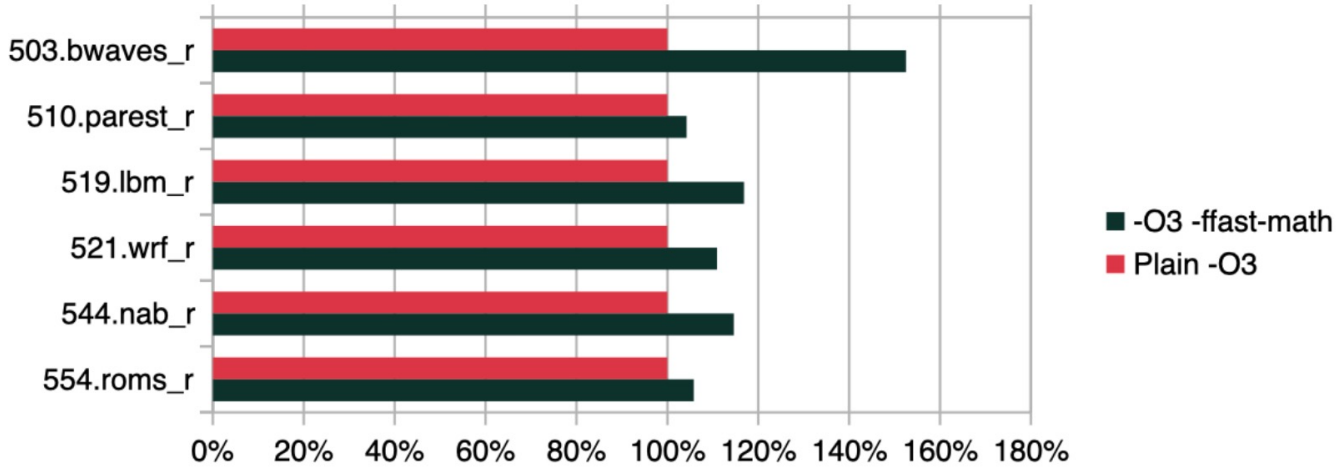


FIGURE 18: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED FLOATING-POINT BENCHMARKS BUILT WITH GCC 11.2 AND -O3 -MARCH=NATIVE, WITHOUT AND WITH -FFAST-MATH

- (*) https://documentation.suse.com/sbp/server-linux/single-html/SBP-GCC-11/

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

… so what should you do with this information.

“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

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.)

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 (this wont work with libraries that require a particular rounding mode).
  3. Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results vary.

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?

When you don't know accuracy …

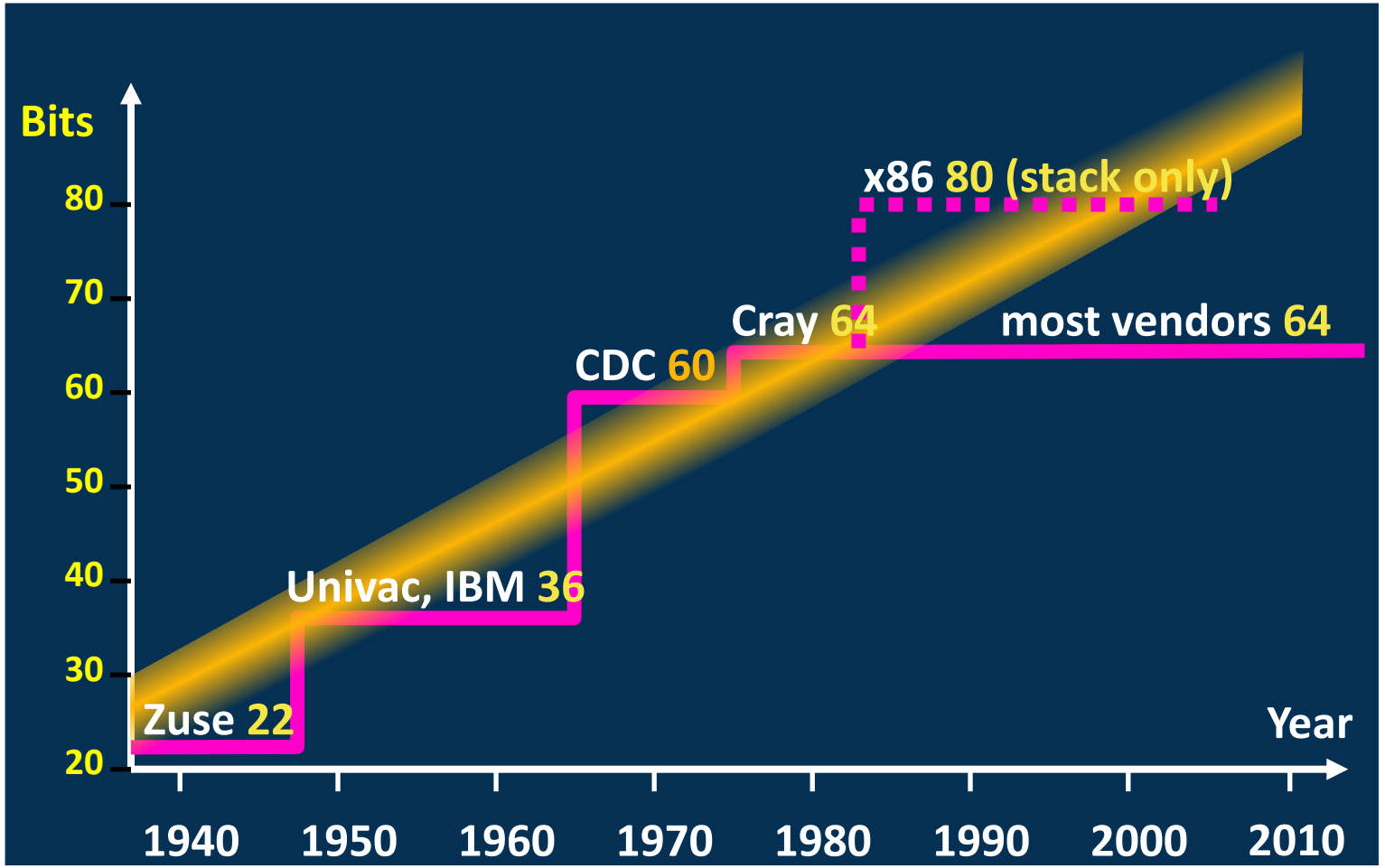
Sleipner Oil Rig Collapse (8/23/91). Loss: **$700 million.**

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.

… so let's just use so many bits in our floating point numbers that we NEVER have any problems.

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

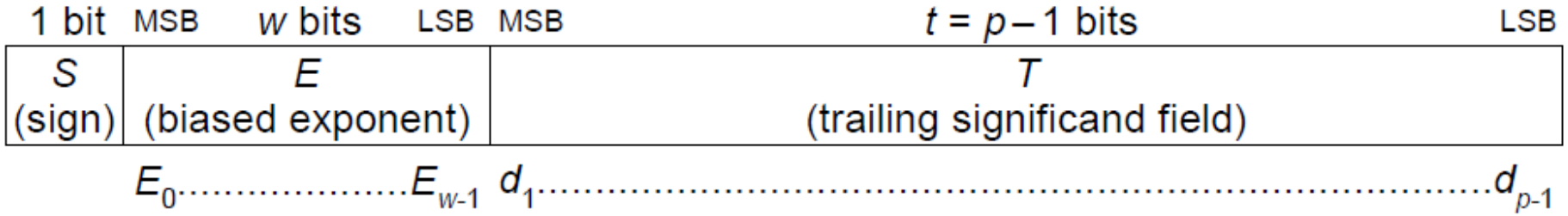


Is 64 bits enough? Is it too much? We're *guessing*.

Quad Precision

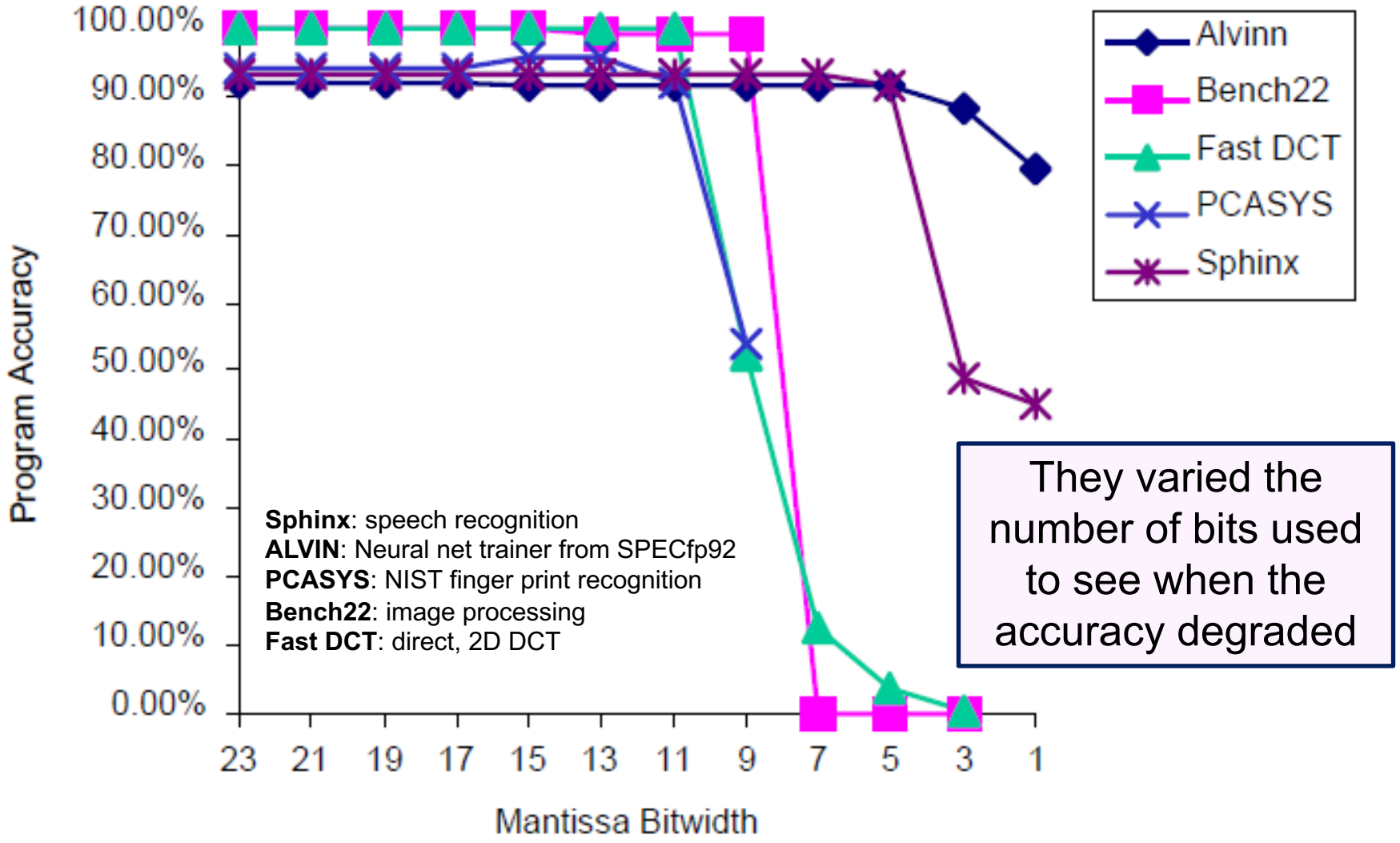
- IEEE 754™ defines a range of formats including quad (128)

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



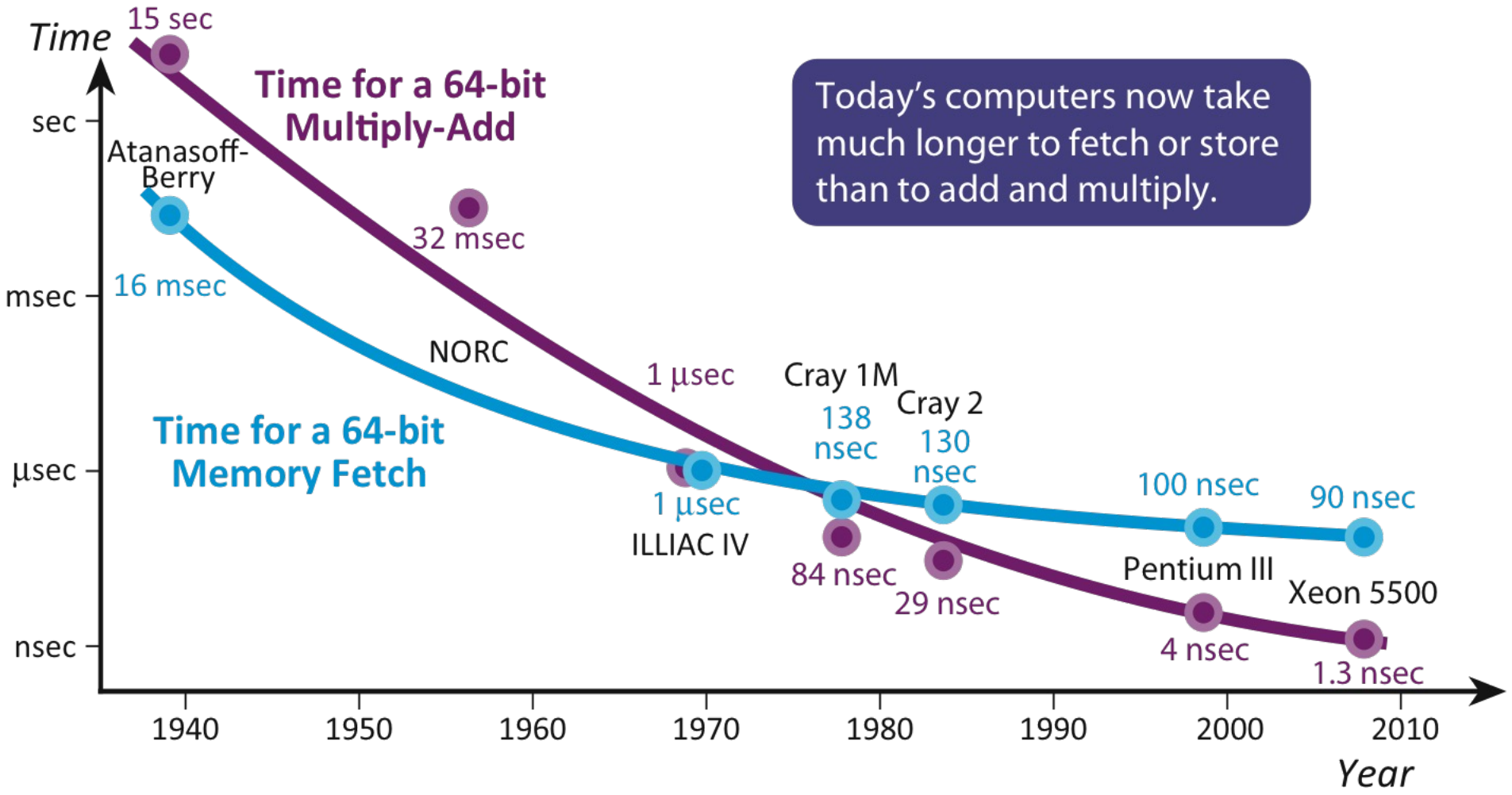
- There are pathological cases where you lose all the precision in an answer, but the more common case is that you lose only half the digits.
- Hence, for 32 or 64 bit input data, quad precision (113 significant bits) is probably adequate to make most computations safe (Kahan 2011).

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,

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



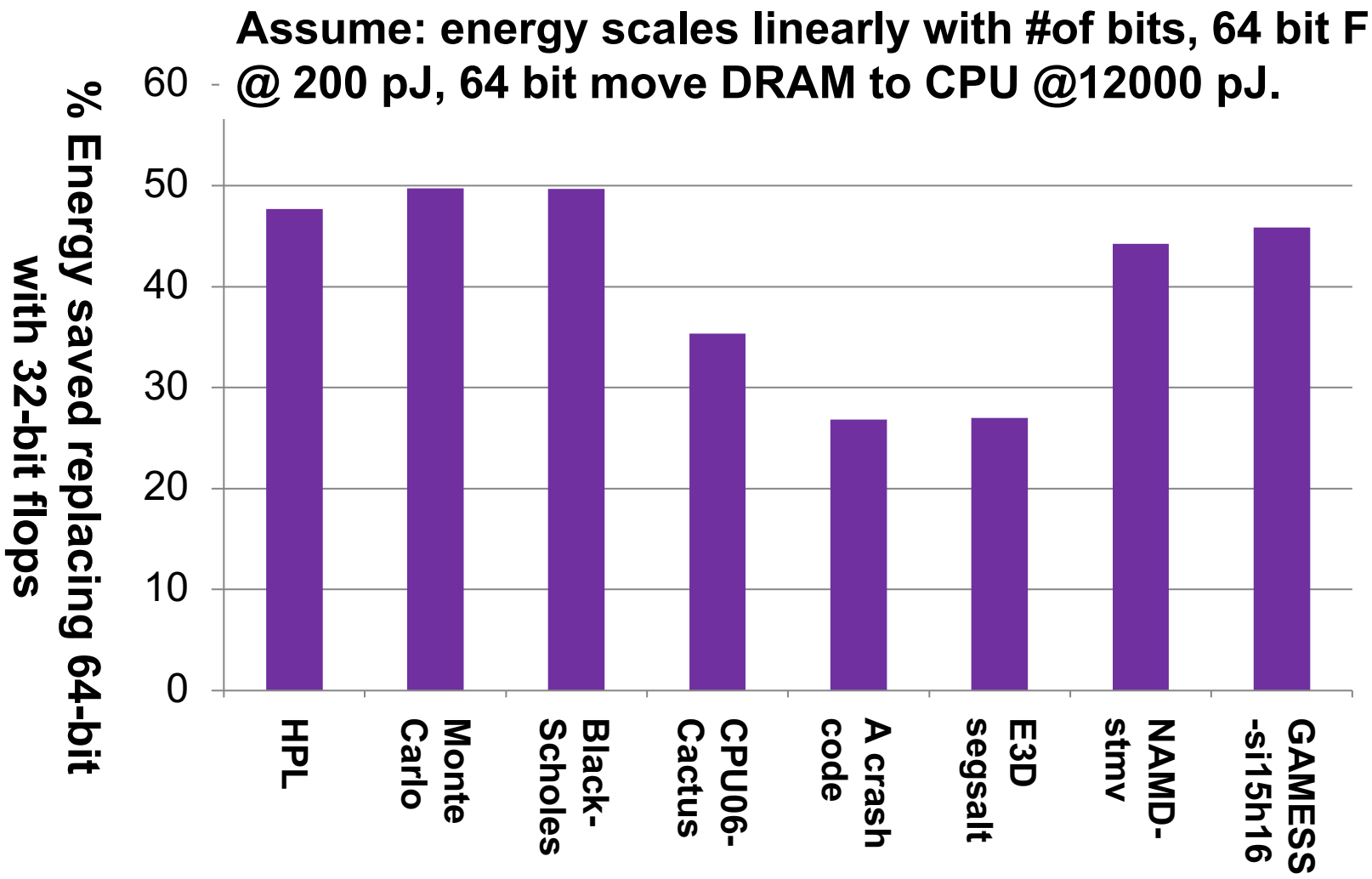
Energy implications of floating point numbers: 32 bit vs. 64 bit numbers

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

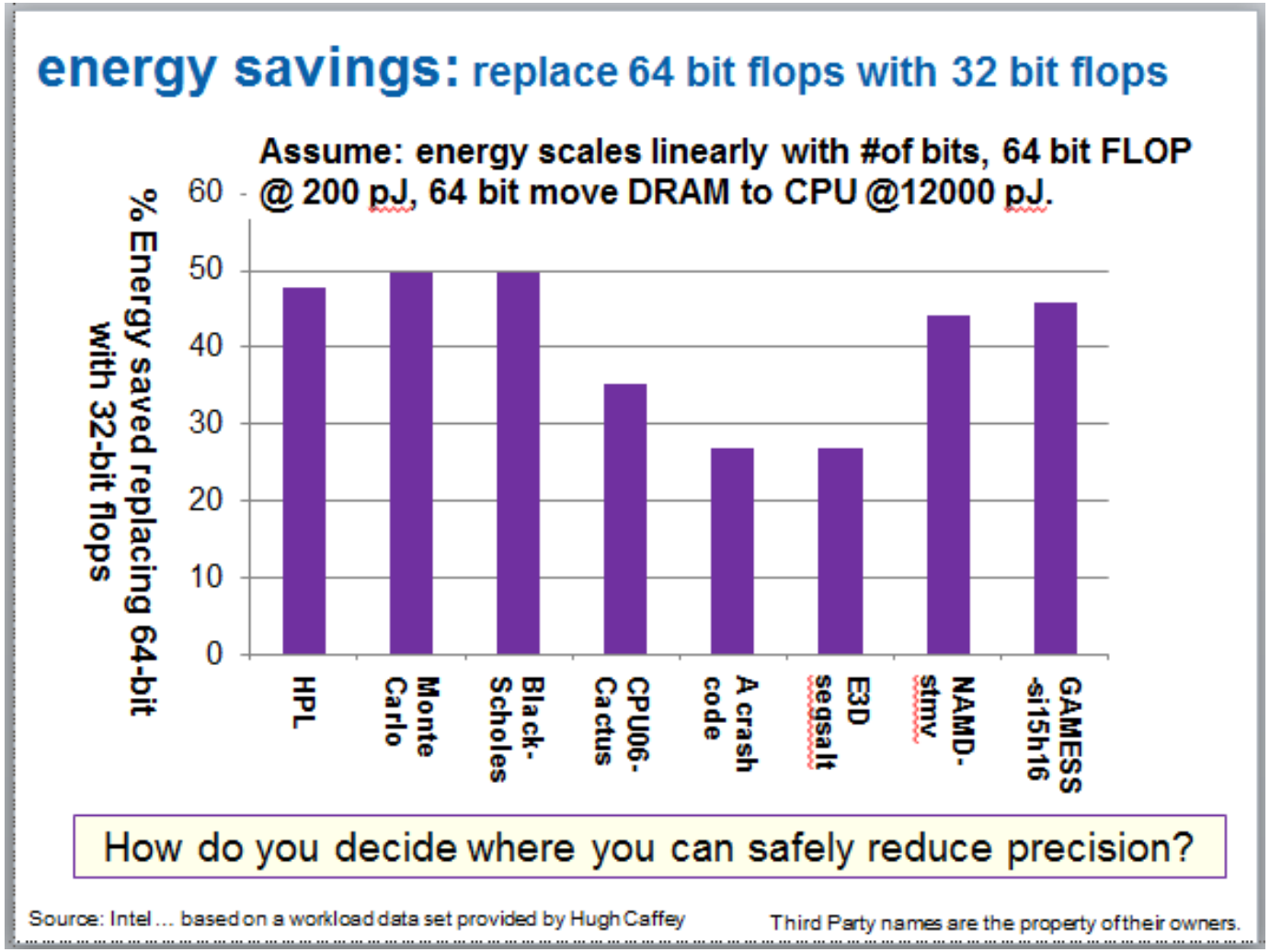


How do you decide where you can safely reduce precision?

Source: Intel … based on a workload data set provided by Hugh Caffey (2010)

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.



… or give up on floating point numbers and use a safe arithmetic system instead.

Interval Arithmetic

Interval Numbers

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

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

- Representing real numbers:
  - A single floating point number

    $1/3 \approx 0.333333$

  - An interval that bounds the real number

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

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

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

… or return to slide rules

(an elegant weapon for a more civilized age)

Image source: Presbrey advertising agency for International Business Machines, 1951

Sleipner Oil Rig Collapse: The slide-rule wins!!!

It was recognized that finding and correcting the flaws in the computer analysis and design routines was going to be a major task. Further, with the income from the lost production of the gas field being valued at perhaps $1 million a day, it was evident that a replacement structure needed to be designed and built in the shortest possible time.

A decision was made to proceed with the design using the pre-computer, slide-rule era techniques that had been used for the first Condeep platforms designed 20 years previously. By the time the new computer results were available, all of the structure had been designed by hand and most of the structure had been built. On April 29, 1993 the new concrete gravity base structure was successfully mated with the deck and Sleipner was ready to be towed to sea (See photo on title page).

The failure of the Sleipner base structure, which involved a total economic loss of about $700 million, was probably the most expensive shear failure ever. The accident, the subsequent investigations, and the successful redesign offer several lessons for structural engineers. No matter how complex the structure or how sophisticated the computer software it is always possible to obtain most of the important design parameters by relatively simple hand calculations. Such calculations should always be done, both to check the computer results and to improve the engineers' understanding of the critical design issues. In this respect it is important to note that the design errors in Sleipner were not detected by the extensive and very formal quality assurance procedures that were employed.

OK, lets wrap this up and conclude.

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.
With arbitrary precision, there is no loss of accuracy when subtracting real numbers	Subtracting two numbers of similar size cancels higher order bits

Aditional sources of numerical inaccuracy

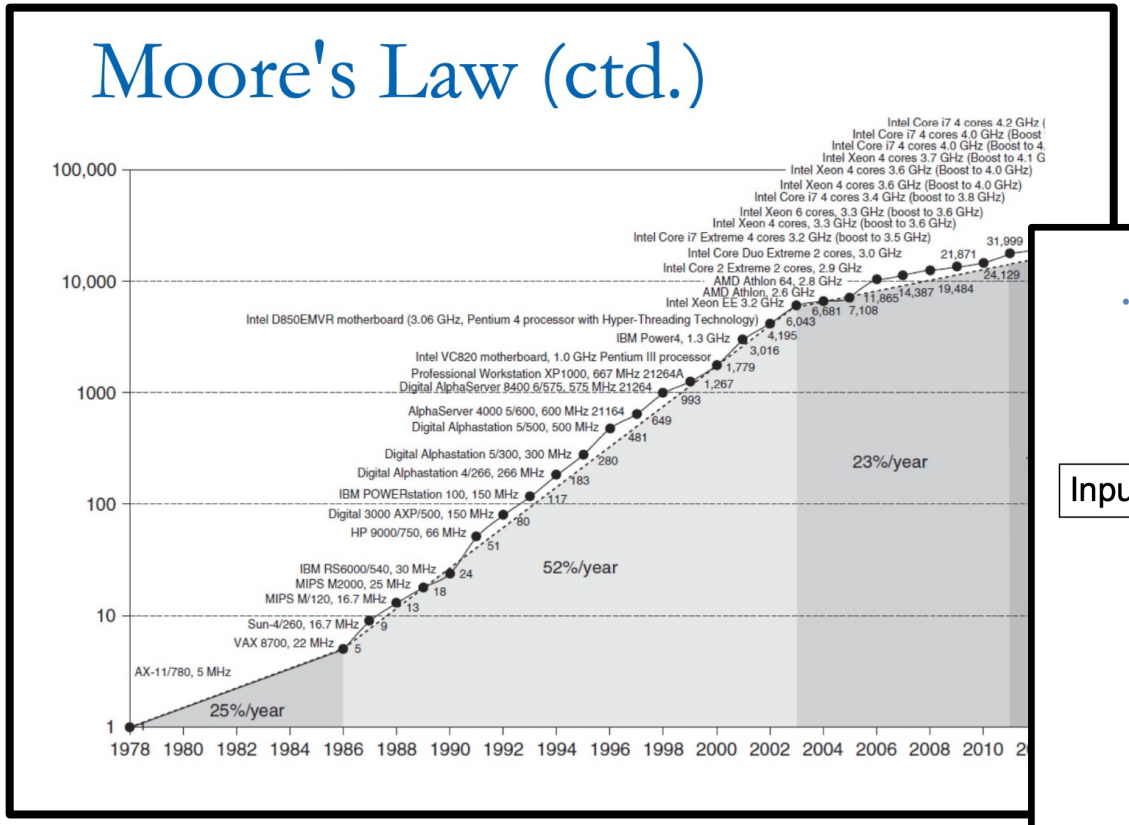
- Floating Point errors
  - Cumulative rounding (“creeping crud”)
  - Operations on values of very different magnitudes (catastrophic accuracy destruction; like, $10^{16} + 3.14$ gives 10^{16}.)
  - I/O; conversion of decimal to binary numbers and back
- Programmer-caused errors
  - Naïve algorithms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

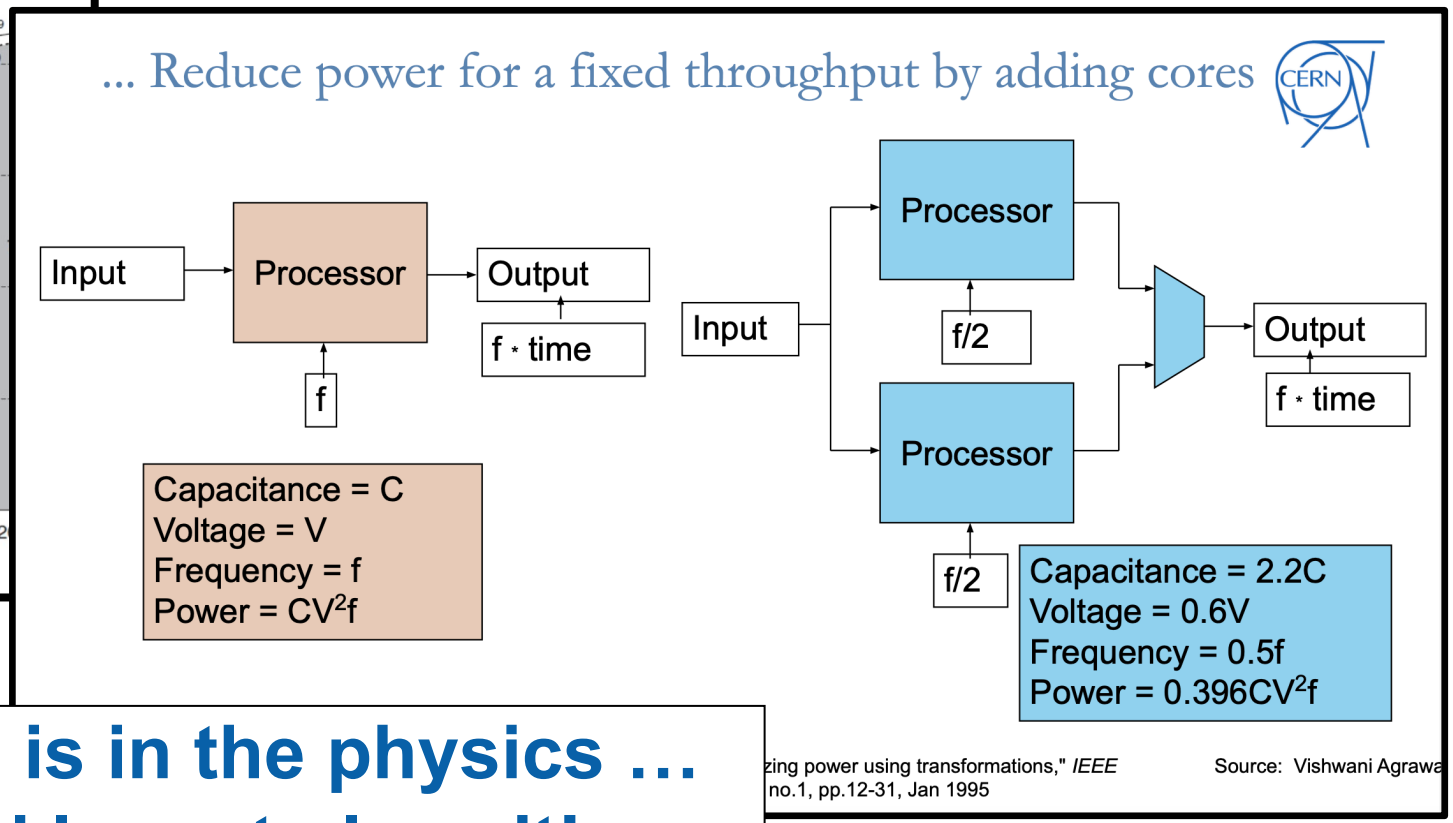
Conclusion

- Floating point arithmetic usually works and you can “almost always” be comfortable using it.
- There are well known problems … which we covered today.
- What we did not cover is numerical analysis. Floating point arithmetic is mathematically rigorous. You can prove theorems and develop rigorous error bounds. It is very powerful.
- Unfortunately, almost nobody is learning this mathematics these days. As scientists using computers in your research, be careful and don’t be shy about testing the fidelity of your computations.
  - Modify rounding modes as an easy way to see if round-off errors are a problem.
  - Recognize that unless you impose an order of association, every order is equally valid. If your answers change as the number of threads changes, that is valuable information suggesting in ill-conditioned problem. Anyone who suggests the need for bitwise identical results from a parallel code should be harshly criticized/punished.

… wait, before we go, I have one more topic to discuss.

Hardware Today is Fundamentally Parallel





zing power using transformations," *IEEE* no.1, pp.12-31, Jan 1995

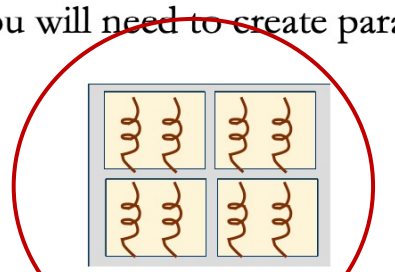
Source: Vishwani Agrawa

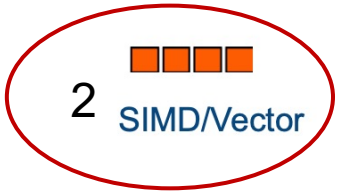
The push for parallelism is in the physics … stop complaining and learn to love it!

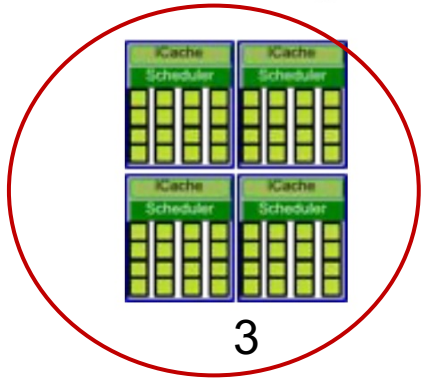
It's actually not so bad … there are only four different hardware platforms you need to learn how to program

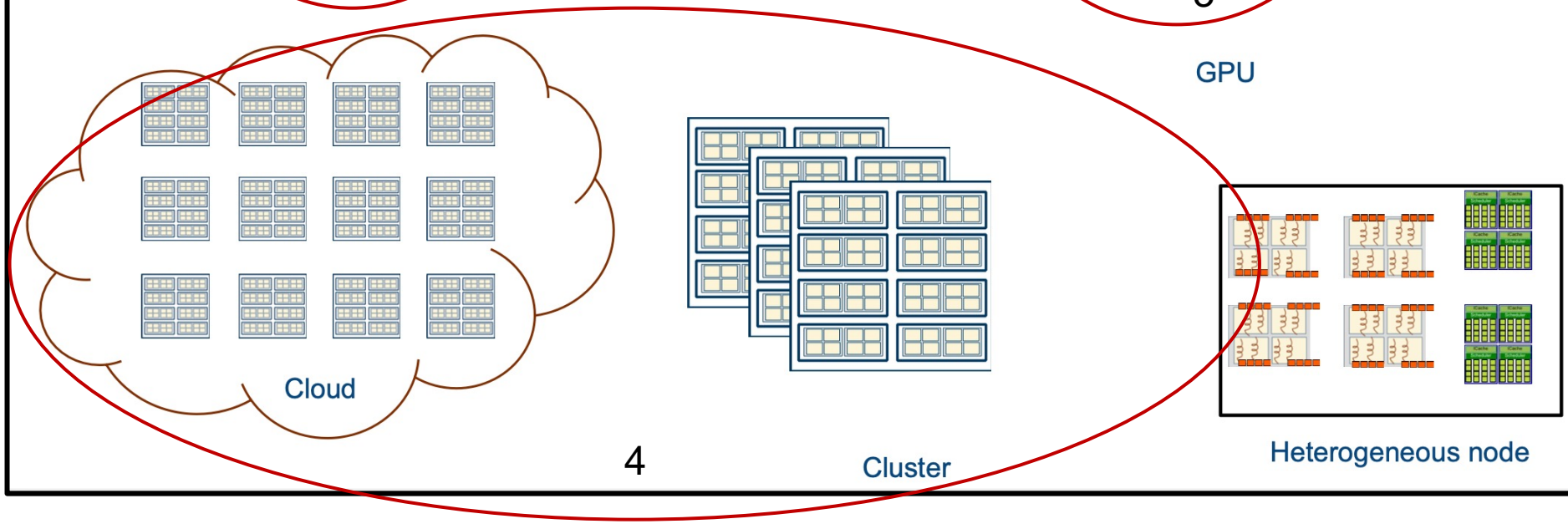
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.





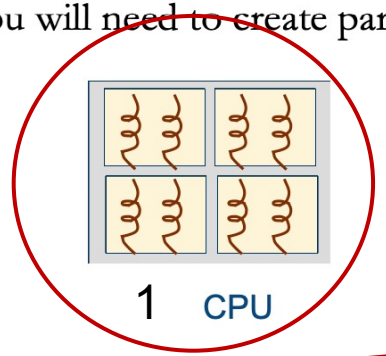


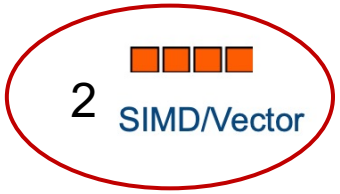


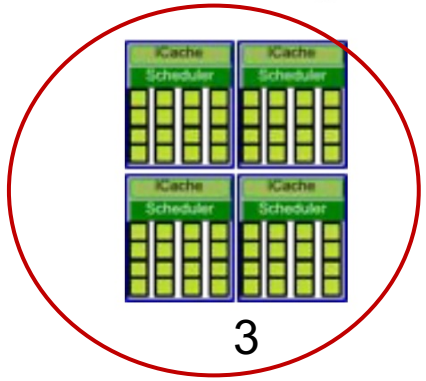
It's actually not so bad … there are only four different hardware platforms you need to learn how to program

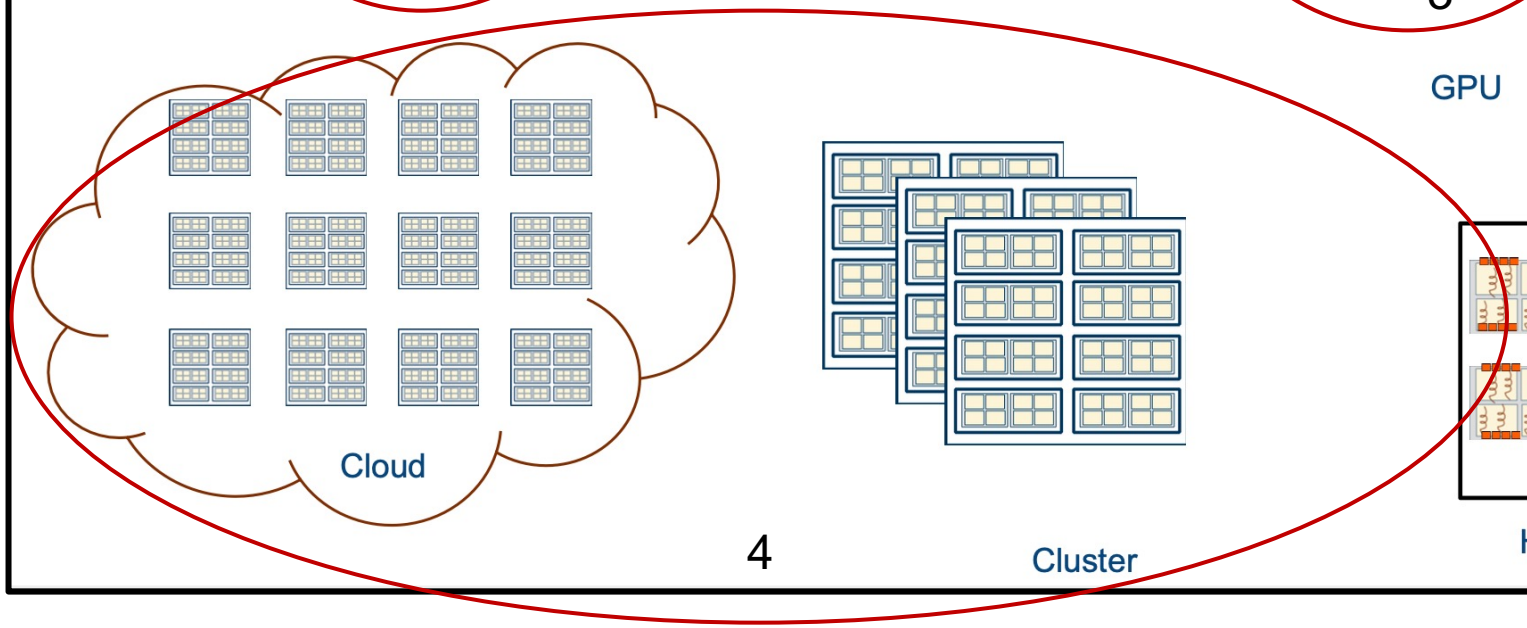
For hardware … parallelism is the path to performance

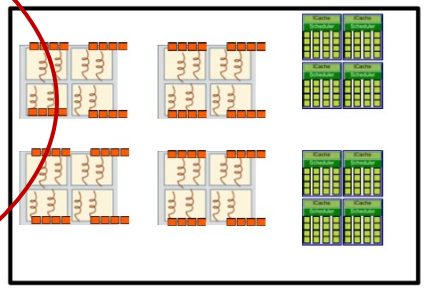
All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.







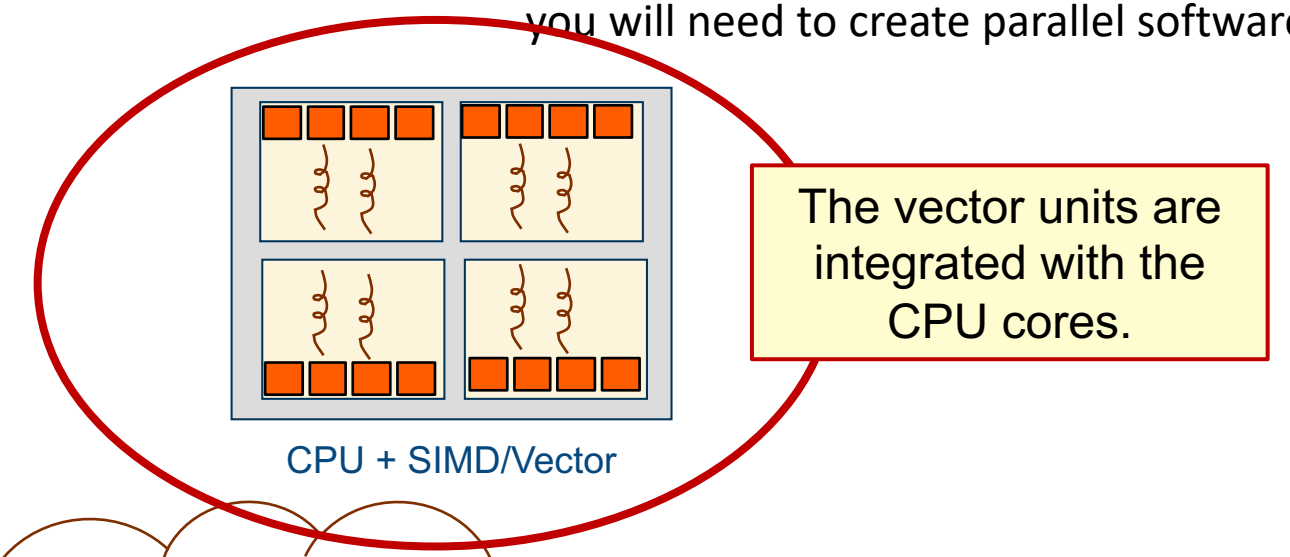




Let's briefly talk about programming a vector unit

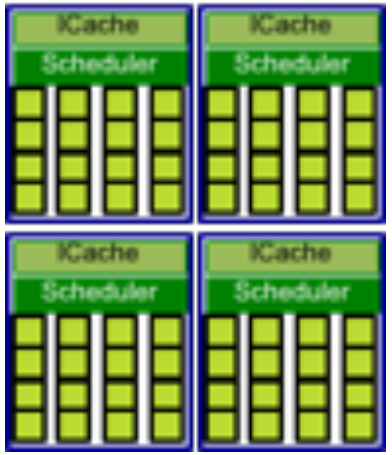
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

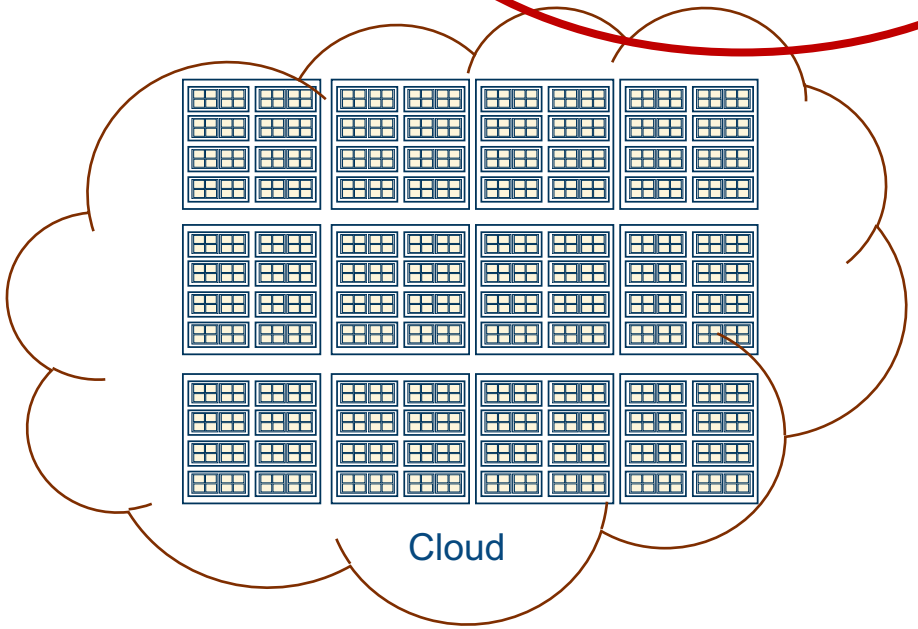


CPU + SIMD/Vector

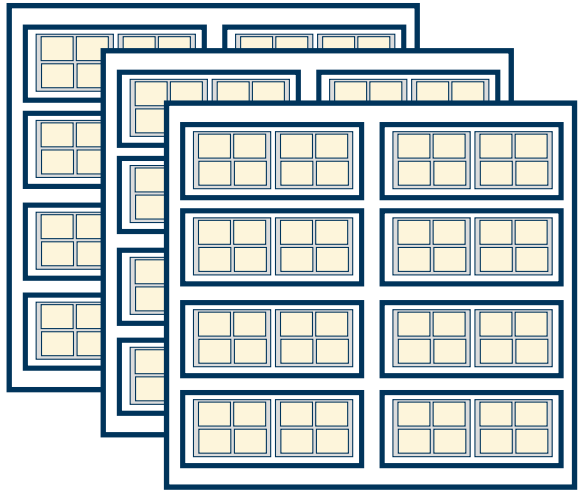
The vector units are integrated with the CPU cores.



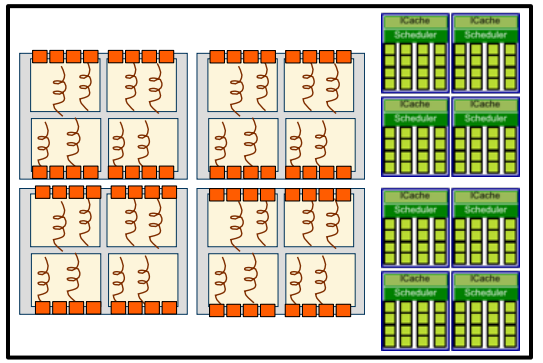
GPU



Cloud



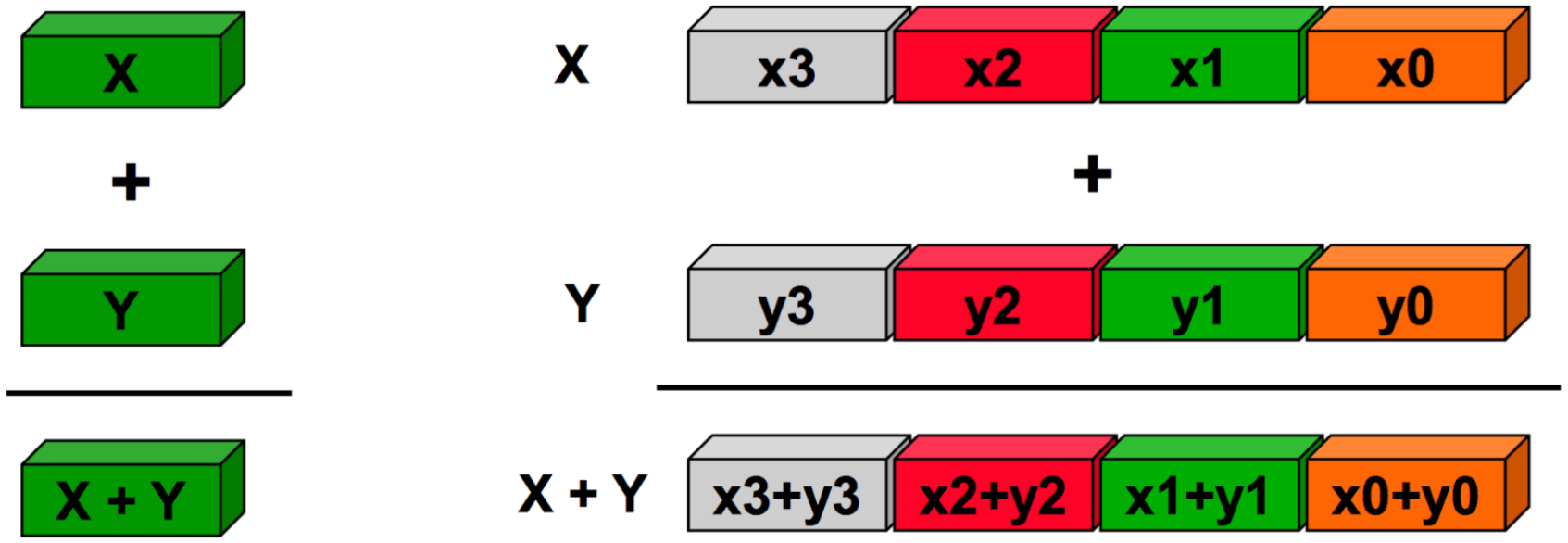
Cluster



Heterogeneous node

Single Instruction Multiple Data

- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing (Intel)
  - with SSE / SSE2
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

An alternative way to think of vector operations

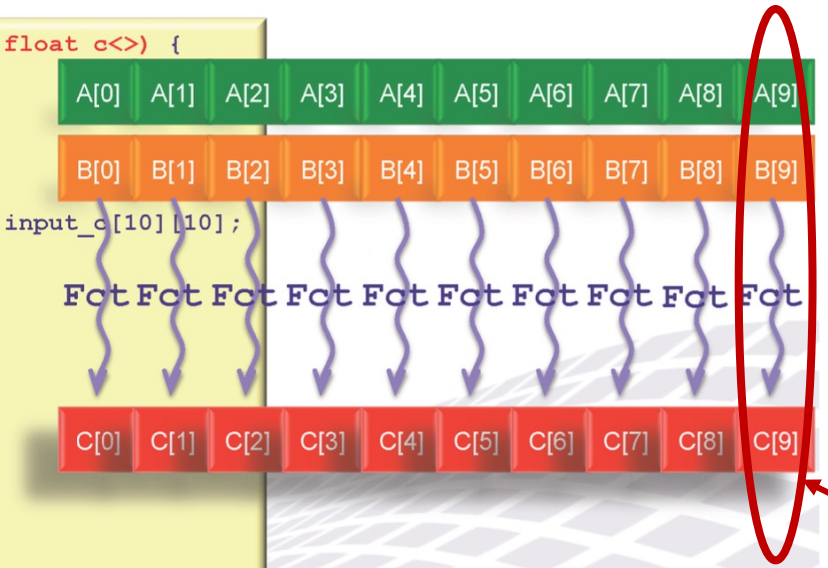
What is Stream Computing?

CAPS

- A similar computation is performed on a collection of data (*stream*)
  - There is no data dependence between the computation on different stream elements
- Stream programming is well suited to GPU *and vector-cpu!*

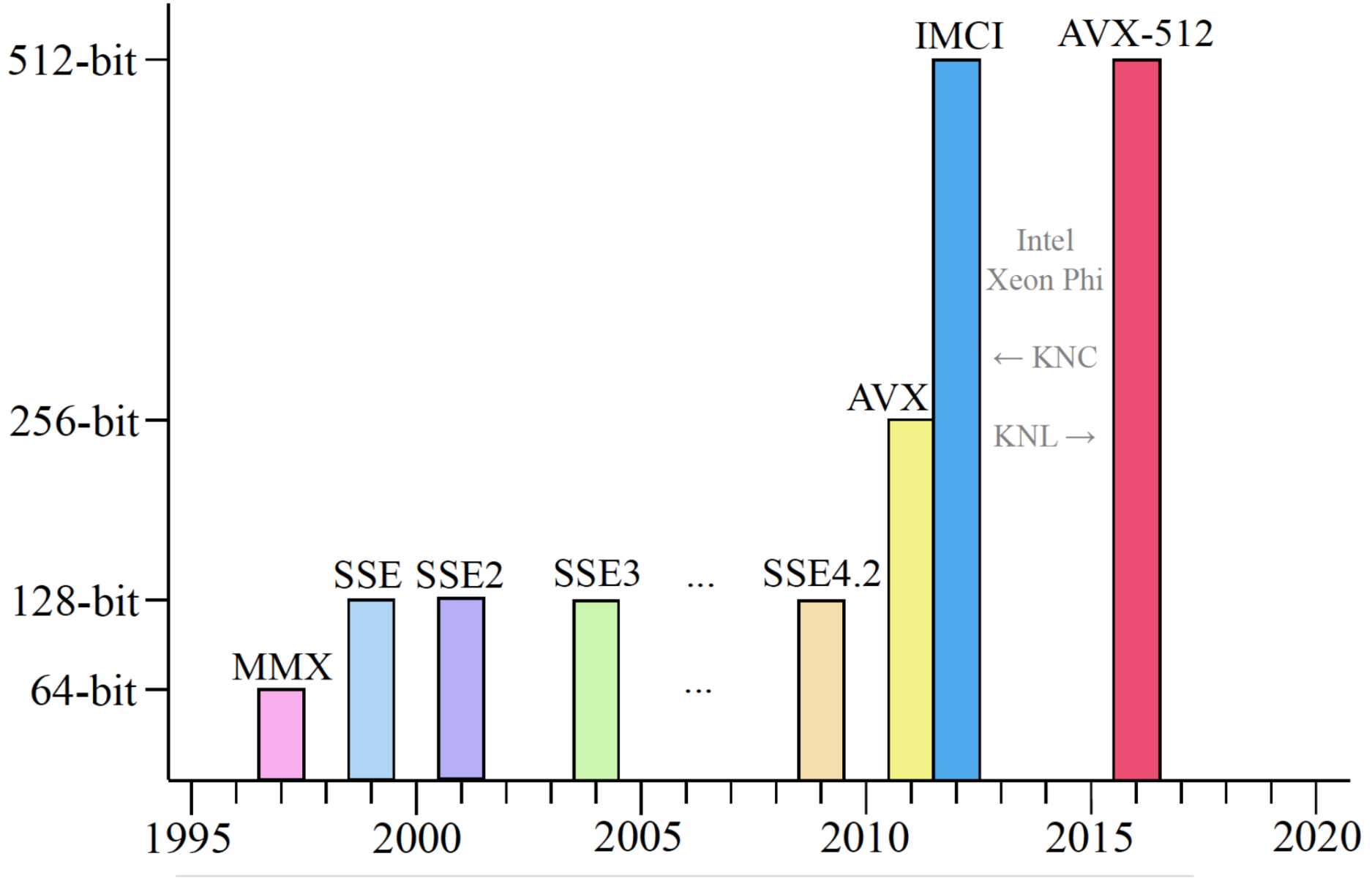
```
kernel void Fct(float a<>, float b<>, out float c<>) {
   c = a + b;
 }
int main(int argc, char** argv) {
   int i, j;
   float a<10, 10>, b<10, 10>, c<10, 10>;
   float input_a[10][10],input_b[10][10], input_c[10][10];
  for(i=0; i<10; i++) {
     for(j=0; j<10; j++) {
      input_a[i][j] = (float) i;
      input_b[i][j] = (float) j;
     }
  }
 streamRead(a, input_a);
 streamRead(b, input_b);
 Fct(a, b, c);
 streamWrite(c, input_c);
 ...
}
```

Brook+ example



For N streams, we think of the vector unit as having N *vector-lanes* (in this case, 10)

Different Vector Instruction sets (x86 only)



Writing vectorized code

- You can write “assembly-like” code to use the vector units.
- But almost nobody (outside of performance fanatics … mostly working at Intel) does this. They get the compiler to generate the code for them.
- Compiler driven vectorization happens in two ways …
  - Super-word parallelization
  - Loop vectorization

Single Instruction Multiple Data

- SLP (Superword Level Parallelism)
  - Direct mapping to underling SIMD machine instruction
  - Usually implemented using array/vector notation
- Loop Vectorization
  - Transform a loop into N streams (N=SIMD-width)
  - Compiler assisted or implemented in a "vector-library"
- Loop vectorization is more efficient than SLP
  - Transform your problem in a long loop over simple quantities

Single Instruction Multiple Data

- SLP (Superword Level Parallelism)
  - Direct mapping to underling SIMD machine instruction
  - Usually implemented using array/vector notation
- Loop Vectorization
  - Transform a loop into N streams (N=SIMD-width)
  - Compiler assisted or implemented in a "vector-library"
- Loop vectorization is more efficient than SLP
  - Transform your problem in a long loop over simple quantities

Even though you are going let the compiler vectorize code for you … it helps if you understand EXACTLY how it does this.

Transform a loop into N streams …

- The first thing the compiler must do is to create a number of streams corresponding to the number of ”vector lanes”
- Example … consider a simple loop that adds two vectors.

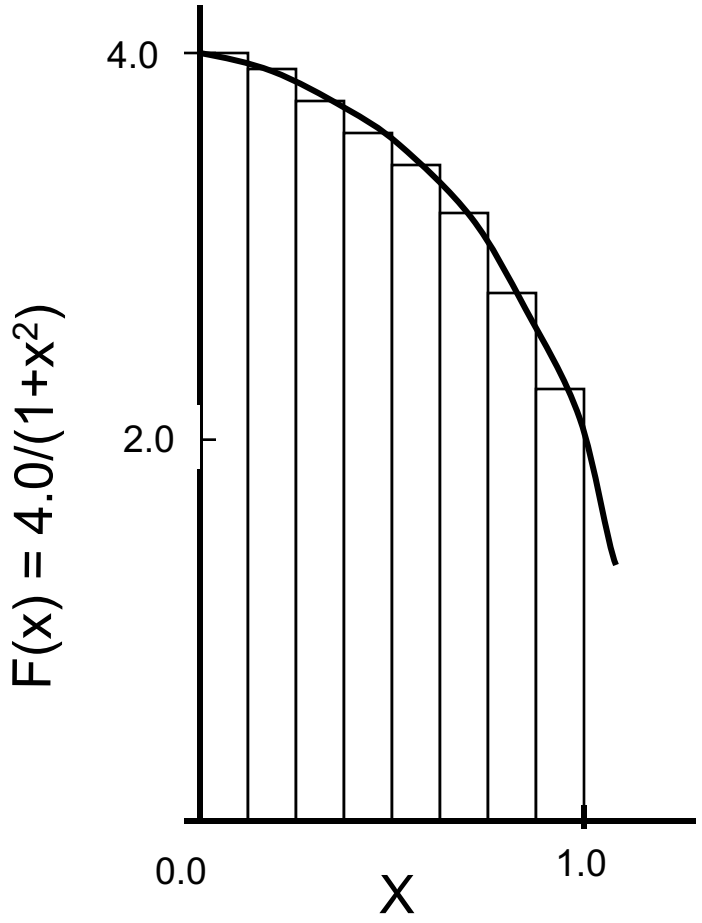
```
for (int i=0; i<N;i++)
      C[i] = A[i] + B[i];
```

- Assume the vectors are a type float (4 bytes, or 32 bits)and we have a SIMD/Vector unit of width 128 bits. That means 128/32 = 4 streams.
- To unroll a loop, we block the loop so each iteration of the new loop body handles 4 iterations of the original loop (for now, let’s keep things simple and assume four evenly divides N).

```
for (int i=0; i<N/4;i+4){
      C[i]   = A[i]   + B[i];
      C[i+1] = A[i+1] + B[i+1];
      C[i+2] = A[i+2] + B[i+2];
      C[i+3] = A[i+3] + B[i+3];
}
```

An Interesting Problem to Play With

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of N rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x = \Delta x \sum_{i=0}^{N} F(x_i) \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{           double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (int i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

See esc23/hands-on/flop/pi.c

Exercise: Unroll the pi program

- Make a copy of the pi.c program. You want to save the original one so you can compare to it later.
- Unroll the loop in the pi program four-fold.
- Note: I use OpenMP to use its simple timer … omp_get_wtime() so you will need to compile the program as:

```
gcc –fopenmp –O0 pi.c
```

- Start with optimization level 0. This tells the compiler do NOT do anything fancy (such as autovectorization).
- Once you have the unrolled program running, so how it changes with other optimization levels.
- How does the performance compare to the original program?

Pi unrolled

```
  sum=0.0;

  for (int i=0;i< num_steps; i=i+4){
      x0  = (i+0.5f)*step;
      x1  = (i+1.0f)*step;
      x2  = (i+1.5f)*step;
      x3  = (i+2.0f)*step;
      sum = sum + 4.0f*(1.0f/(1.0f+x0*x0) + 1.0f/(1.0f+x1*x1)  +
                        1.0f/(1.0f+x2*x2)  + 1.0f/(1.0f+x3*x3));
  }
```

```
$ gcc -O0 -fopenmp -o pi_unroll pi_unroll.c

$ ./pi_unroll

 Base (double)     8388608 steps 3.141593: ave=0.027333 min=0.021463 max=0.060666 secs
 Base (float)      8388608 steps 2.991987: ave=0.021593 min=0.020670 max=0.022706 secs
 Unroll 4 (float)  8388608 steps 3.139504: ave=0.014700 min=0.014274 max=0.015405 secs
 Unroll 4 (double) 8388608 steps 3.141593: ave=0.015834 min=0.014840 max=0.016305 secs
```

Replace the unrolled loop body with the vector intrinsics

- Vector intrinsics use an assembly code style … explicit vector registers and low-level register-to-register instructions.

- Use the include file … #include<intrinsics.h>. Tell the compiler to enable the native vector instruction sets with –mach=nativer

- Common instructions:

```
__m128 var0;   //var is a variable of type 128 bit vector register
__m128 var1  _mm_setr_ps(a, b, c, d);  // pack 4 single precision values into a vector register
__m128 var2    = _mm_load1_ps(&val); // pack 4 SP values with the value pointed to by val
__m128 var3 = _mm_add_ps(var1,var2); // add two packed SP registers and put the result in var3
__m128 var3 = _mm_mul_ps(var1,var2); // multiply two packed SP registers and put the result in var3
__m128 var3 = _mm_div_ps(var1,var2); // divide two packed SP registers and put the result in var3
_mm_store_ps(&varrauy[0],var3); // move 4 values in a packed SP (var3) into 4 elements of varray
```

- You can chain the vector instructions:

```
__m128 var4 = _mm_mul_sp(_mm_vadd_ps(v1,v2),v3);
```

For more info … https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Replace the unrolled loop body with the vector intrinsics

```
step = 1.0/(float) num_steps;
__m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
__m128 one    = _mm_load1_ps(&scalar_one);
__m128 four   = _mm_load1_ps(&scalar_four);
__m128 vstep  = _mm_load1_ps(&step);
__m128 sum    = _mm_load1_ps(&scalar_zero);
__m128 xvec;
__m128 denom;
__m128 eye;

for (i=0;i< num_steps; i=i+4){ // assume num_steps%4 = 0

   ival  = (float)i;
   eye   = _mm_load1_ps(&ival);
   xvec  = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
   denom = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
   sum   = _mm_add_ps(_mm_div_ps(four,denom),sum);
}
_mm_store_ps(&vsum[0],sum);

pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
```

This is hard and not portable between vector instruction sets

- Most of us let the compiler do the vectorization for us. The following table lists the most important compiler switches.

-O2	vectorize qnd other optimizations, but try to keep code size down and keep compile-time down
-O3	vectorize and other aggressive optimizations, don't worry about how long it takes or code size
-Ofast	vectorize VERY aggressively relaxing floating point standards if needed
`-fopt-info-vec`	`Generate a detailed report about vectorization`
`-march=native`	Enable native vector instruction sets

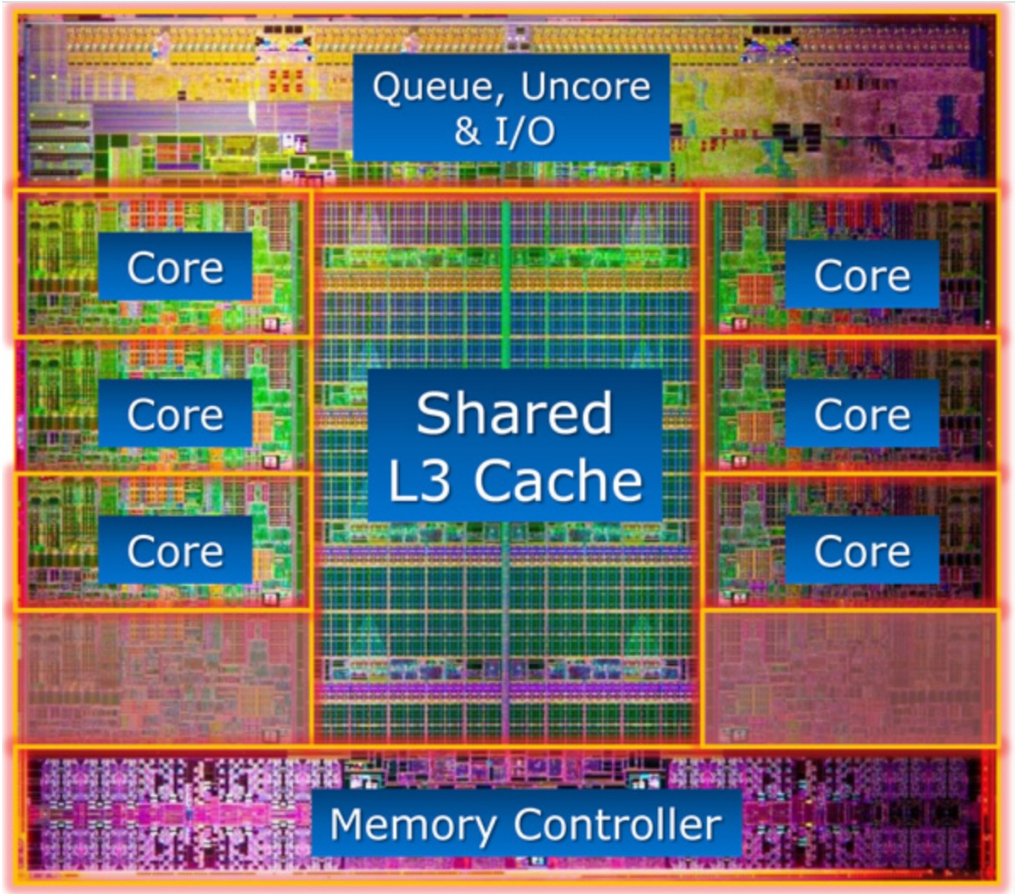
- `There is much more we could cover in a complete discussion of compiler vectorization. Our goal here was to explain how it works using a simple example and give you some basic options to get you started.`

Why not just ignore the vector/SIMD units?

- If you are not using the vector unit, you are ignoring much of the available performance (even though you are paying for it … in dollars/euros and power).
- For example, consider an Intel i7 CPU (SandyBridge … an old CPU, 2010) with SSE vector instructions.
  - 6 cores, 3.2 Ghz, 2-wide hyperthreading, 4-wide Single Precision (SP) Vector unit, 2-wide scale SP scalar*.
    - 6*3.2*2*2 = 76.8 SP Gigaflops Scalar
    - 6*3.2*2*4 = 153.6 SP Gigaflops Vector

Most of the performance comes from the vector unit.

How do you access that performance?



https://www.anandtech.com/show/5091/intel-core-i7-3960x-sandy-bridge-e-review-keeping-the-high-end-alive

Peak perf estimates based on: "Debunking the 100X CPU vs. GPU myth", Lee et al, https://dl.acm.org/doi/10.1145/1815961.1816021

Conclusion

- Floating point arithmetic … we typically pretend its real and we take whatever performance we can get.

- Hopefully you now see that this can be dangerous. Be careful and watch out for problems.
  - Check condition numbers when they are provided by libraries.
  - Try different rounding modes and make sure answers are stable
  - Run loops backwards (when you can) and make sure answers are stable
  - Think about algorithms and watch for round-off error accumulations and cancelation problems
- When all else fails, find a good numerical analyst to help
- And finally … if you have a nice CPU to use, make the most of it. Make your loops friendly to vectorize and turn on vectorization compiler flags

Back-up slides

Exercise: The Kahan Summation Algorithm

- Using the properties of floating point arithmetic, algorithms that reduce round-off errors can be designed.
- A famous one is the Kahan Summation Algorithm. Here it is in pseudo-code

```
Input: a sequence of N values, x[i] i=1,N

   correction = 0.0
   sum = 0.0

   for i = 1 to N:

      xcor = x[i] – correction


      tmpSum =  sum + xcor

      correction = (tmpSum – sum) – xcor

      sum = tmpSum
   }
Output: sum
```

apply the correction term to x[i] since it's a small number and we'd loose precision if we corrected sum (which is much larger, so we'd loose low order bits in the operation)

sum is big, but xcor is small. → Low order digits of xcor are lost

(tmpSum-sum) recovers xcor, but without the bits lost in computation of tmpSum.
Xcor has those bits. Subtracting xcor recovers those lost bits. But xcor is later than (tmpSum-sum) so you must apply the correction by subtracting it.

Rounding - Catastrophic Cancellation - Quadratic Equation

$$ax^2 + bx + c = 0 \longrightarrow$$

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$= -\frac{b}{2a}(1 \mp \sqrt{1 - \frac{4ac}{b^2}})$$

- Let's rewrite the solutions
- Let's define □ = 4ac / b^2

$$x_+ = -\frac{b}{2a}(1 - \sqrt{1 - \delta})$$

- When b^2 >> 4ac → □ << 1
  - $(1 - \sqrt{1 - \delta})$ contains a possible cancellation!
- We can remove one cancellation rationalizing the expression
  - Multiply by $1 + \sqrt{1 - \delta}$ numerator and denominator
- Now no catastrophic cancellation can occur!

$$x_+ = -\frac{b}{2a}\left(\frac{1 - (1 - \delta)}{1 + \sqrt{1 - \delta}}\right)$$
$$= -\frac{2c}{b}\left(\frac{1}{1 + \sqrt{1 - \delta}}\right)$$

Source: Wahid Redjeb, ESC'22

Workload	Description	Accuracy Measurement
Sphinx III	CMU's speech recognition program based on fully continuous hidden Markov models. The input set is taken from the DARPA evaluation test set which consists of spoken sentences from the Wall Street Journal.	Accuracy is estimated by dividing the number of words recognized correctly over the total number of words in the input set.
ALVINN	Taken from SPECfp92. A neural network trainer using backpropagation. Designed to take input sensory data from a video camera and a laser range finder and guide a vehicle on the road.	The input set consists of 50 road scenes and the accuracy is measured as the number of correct travel direction decisions made by the network.
PCASYS	A pattern-level finger print classification program developed at NIST. The program classifies images of fingerprints into six pattern-level classes using a probabilistic neural network.	The input set consists of 50 different finger print images and the classification result is measured as percentage error in putting the image in the wrong class. The accuracy of the recognition is simply (1 - percentage error).
Bench22	An image processing benchmark which warps a random image, and then compares the warped image with the original one.	Percentage deviation from the original outputs are used as a measure of accuracy.
Fast DCT	A direct implementation of both 2-dimensional forward Discrete Cosine Transform (DCT) and inverse DCT of blocks of 8x8 pixels.	100 random blocks of 8x8 pixels are transformed by forward DCT and then recovered by inverse DCT. Accuracy measured as percentage of correctly recovered pixels.

Notes to support the "how many bits do we need" slide