FLOATING POINT IS NOT REAL

Matthieu Lefebvre
Princeton University

Wednesday, 25 July 2018

Second Computational and Data Science school for HEP (CoDaS-HEP)

SETUP – EXAMPLES / EXERCISES

- Prerequisites:
  - Recent C++ compiler (at least C++11 compliant)
  - Eventually CMake

- git clone https://github.com/mpbl/codas_fpa/

- README.md for instructions

OUTLINE

- Disasters
- Reminder on integers
- Floating point numbers
- IEEE 754
  - Rounding modes, exceptions, underflow, …
- Improving FPA accuracy
  - Kahan algorithm, FMA
- Computing Faster
  - Fast Math, reduced precision, mixed precision
- Concurrency
- Conclusion
- References

DISASTERS DUE TO MACHINE REPRESENTATION

Patriot Missile Failure
Rounding errors
1991, Gulf War. Failed to track and intercept an incoming Iraqi Scud missile. Inaccurate calculation of the time since boot due to computer arithmetic errors

Explosion of the Ariane 5
Overflow
1996, Kourou, French Guiana
software error in the inertial reference system
Storing 64 bits FP into 16 bits integers

http://www-users.math.umn.edu/~arnold/disasters/

A WORD ABOUT INTEGERS

```
template <typename Integer>
void world_population() {
    //
https://en.wikipedia.org/wiki/List_of_continents_by_population
    // in 2010
    std::cout << "Sizeof(Integer) : " << sizeof(Integer) <<
std::endl;

    std::map<std::string, Integer> continents = {
            {"africa", 1'044'107'001}, {"americas", 943'952'001},
            {"asia",   4'169'860'001}, {"europe",   735'395'001},
            {"oceania",   36'411'001}};

    Integer total = 0;
    for (auto &continent : continents) {
        total += continent.second;
    }
    std::cout <<  "Total world population : " << total << std::endl;
}

world_population<uint32_t>();
```

✎ 01_integer/integer_overflow

- How many people, worldwide?
- Does it makes sense?
- What might be the problem?
- Why data are from 2010 and not 2016?

REMINDER: INTEGERS

➔ `int` is not integer

- Representation uses a limited number of bits
  - Positive numbers are just represented using their binary form
  - Negative numbers often use two's complement

- Properties of arithmetic types can be queried using std::numeric_limits (C++)
  - On my machine (and probably on yours)
    - 2^{32} < int = int32_t <= 2^{31}-1 // 1 bit is used to store the sign
    - 0 < unsigned int = uint32_t <= 2^{32}

WHY USING FLOATING POINT NUMBERS

- Representing numbers that would be too large or too small to be represented as integers
  - 1.4e−45 to 3.4e38

- Representing numbers that are not representable as integers

- Of course, floating points representations are also subject to use only a limited number of bits.

DESIRABLE PROPERTIES

- Speed
- Accuracy:
  - “Correct” results
- Range:
  - Large and small numbers
- Portability:
  - Run on different machines, giving the same answer
- Ease of implementation and use
  - Needs to feel natural, at least to the user

Handbook of Floating Point Arithmetic

REAL TO FLOATING POINTS

- A number is represented exactly by:

$$\textit{Significand} \times \textit{base}^{\textit{exponent}}$$

- By instance:

$$3.1415 = 31415 \times 10^{-4}$$

Significand:
- Mantissa
- Coefficient

Base:
- Radix

- **Stored in memory using a limited number of bits:**

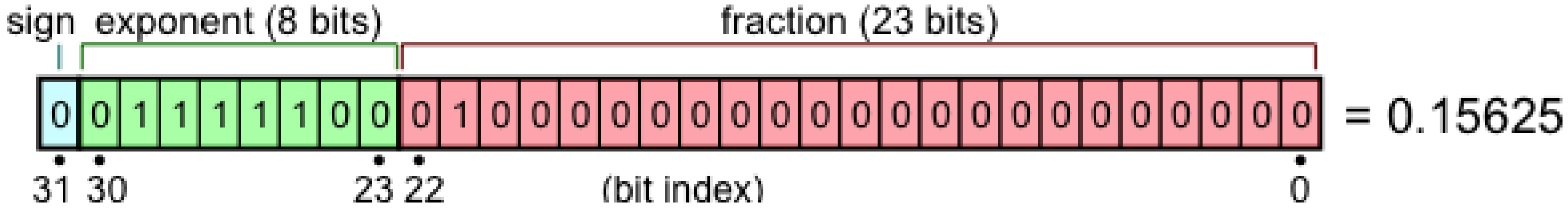
8 bits

16 bits (**half** precision)

32 bits (**single** precision)

64 bits (**double** precision)

IEEE 754 REPRESENTATION OF SINGLE PRECISION FP



Exponent	Fraction == 0	Fraction != 0	Equation
All Zeros	0, -0	Subnormal value (Fraction starts with an implicit 0)	$(-1)^{sign} \times 2^{-126} \times 0.fraction$
All Ones	$\pm\infty$	NaN	
Otherwise	Normalized value (Fraction starts with an implicit 1)		$(-1)^{sign} \times 2^{exponent - 127} \times 1.fraction$

https://en.wikipedia.org/wiki/Single-precision_floating-point_format

ROUNDING MODES

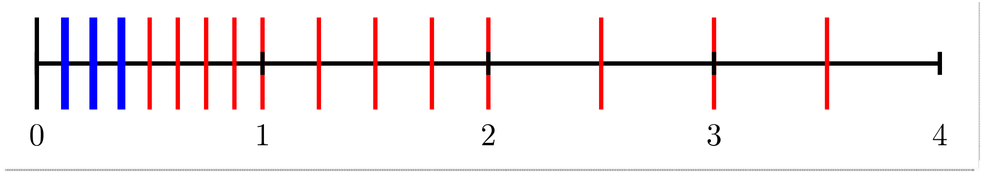
- **Roundings to nearest**
  - **Round to nearest, ties to even** [Default Mode]– rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit; this is the default for binary floating-point and the recommended default for decimal.
  - **Round to nearest, ties away from zero** – rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers); this is intended as an option for decimal floating point.
- **Directed roundings**
  - **Round toward 0** – directed rounding towards zero (also known as **truncation**).
  - **Round toward +∞** – directed rounding towards positive infinity (also known as rounding up or **ceiling**).
  - **Round toward −∞** – directed rounding towards negative infinity (also known as rounding down or **floor**).

FLOATING POINT EXCEPTIONS

- The IEEE standard defines several FP exceptions
  - Can be ignored ➔ Default action is taken
  - Can be trapped ➔ Error is signaled
- **Underflow**: Too small to be represented as a normalized float in its format.
  - If ignored, the operation results in a denormalized float or zero.
- **Overflow**: Too large to be represented as a float in its format.
  - If ignored, the operation results in the appropriate infinity.
- **Divide-by-zero**: Float is divided by zero.
  - If ignored, the appropriate infinity is returned.
- **Invalid**: Ill-defined operation, such as (0.0/ 0.0).
  - If ignored, a quiet NaN is returned.
- **Inexact**: The result of a floating point operation is not exact, i.e. the result was rounded.
  - If ignored, the rounded result is returned

GRADUAL UNDERFLOW (SUBNORMALS)

- Subnormals (or denormals) are FP smaller than the smallest normalized FP: they have leading zeros in the significand
  - For single precision they represent the range 10^{-38} to 10^{-45}



- Subnormals guarantee that additions never underflow
  - Any other operation producing a subnormal will raise a underflow exception if also inexact
- Hardware is not always able to deal with subnormals
  - Software assist is required: SLOW
  - To get correct results even the software algorithms need to be specialized
- It is possible to tell the hardware to flush-to-zero (ftz) subnormals
  - It will raise underflow and inexact exceptions

IMPROVED ACCURACY: KAHAN SUMMATION ALGORITHM

```
function KahanSum(input)
    var sum = 0.0
    var c = 0.0 // A running compensation for lost low-order bits.
    for i = 1 to input.length do
        var y = input[i] - c // So far, so good: c is zero.
        // Alas, sum is big, y small, so low-order digits of y are lost.
        var t = sum + y
        // (t - sum) cancels the high-order part of y;
        // subtracting y recovers negative (low part of y)
        // Algebraically, c should always be zero.
        // Beware overly-aggressive optimizing compilers!
        c = (t - sum) - y
        sum = t
    return sum
```

✎ patriot/patriot.cpp (V. Innnocente)

https://en.wikipedia.org/wiki/Kahan_summation_algorithm

IMPROVED ACCURARY

- Kahan Summation Algorithm does not work for "ill-conditioned" sums
  - In particular in an element is larger than the sum
- Other summation algorithms
  - Fast2Sum (Dekker), 2Sum (Knuth et al.), …
- Products also have specific algorithms for accurate computations:
  - Dekker, …
- Algorithms for computing means, variances, …

☞Handbook of Floating-Point Arithmetic

FUSED MULTIPLY-ACCUMULATE (FMA)

- Or Fused Multiply-Add (FMA) : $a \times b + c$
- **Multiplier–Accumulator** (MAC) hardware unit
- Performed with a single rounding (IEEE 754-2008) (instead of 2 for one multiplication followed by an addition)
- A fast FMA can speed up and improve the accuracy of many computations that involve the accumulation of products:
  - Dot product
  - Matrix multiplication
  - Polynomial evaluation (e.g., with Horner's rule)
  - Newton's method for evaluating functions.
  - Convolutions and artificial neural networks

https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation

ROUND-OFF ERROR ANALYSIS

- Inverse analysis
  - based on the " Wilkinson principle": the computed solution is assumed to be the exact solution of a nearby problem provides error bounds for the computed results
- Interval arithmetic
  - The result of an operation between two intervals contains all values that can be obtained by performing this operation on elements from each interval.
    - guaranteed bounds for each computed result
    - the error may be overestimated
    - specific algorithms
- Probabilistic approach
  - uses a random rounding mode
  - estimates the number of exact significant digits of any computed result

http://www.math.twcu.ac.jp/~conf/FJWNC2015/doc/Jezequel.pdf

COST OF OPERATIONS (IN CPU CYCLES)

Operator	Instruction	AVX FP32	AVX FP64
+, -	ADD, SUB	3	3
==, !=	COMISS, CMP	2,3	2,3
cast fp32 <-> fp64	CVT	4	4
|, &, ^	AND, OR	1	1
*	MUL	5	5
/, sqrt	DIV, SQRT	21-29	21-45
1.f/□, 1.f/sqrt□	RCP, RSQRT	7	
=	MOV	1,4…	1,4…

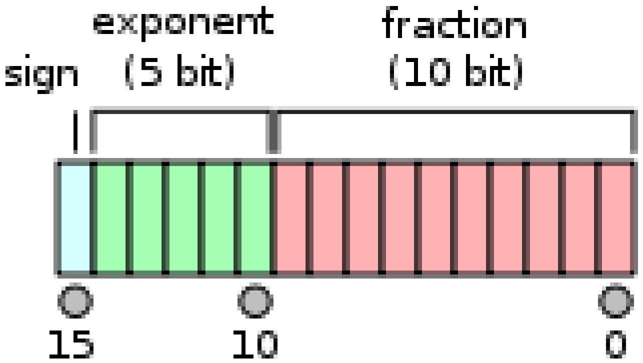
FAST MATH

- man gcc /-ffast-math -- Sets the options:
- -fno-math-errno
  - Do not set "errno" after calling math functions that are executed with a single instruction
- -funsafe-math-optimizations :
  - assume that arguments and results are valid.
- -ffinite-math-only
  - Allow re-association of operands in series of floating-point operations.
  - ☞ Patriot example ?
- -fno-rounding-math
  - Disable transformations and optimizations that assume default floating-point rounding behavior.
- -fno-signaling-nans
  - Do not assuming that IEEE signaling NaNs may generate user-visible traps during floating-point operations. (default)
- -fcx-limited-range: range check for complex division.

SPEEDING MATH UP

- Avoid or factorize-out division and sqrt
  - if possible compile with "–Ofast" or "-ffast-math"
- Prefer linear algebra to trigonometric functions
- Cache quantities often used
  - No free lunch: at best trading memory for cpu
- Choose precision to match required accuracy
  - Square and square-root decrease precision
  - Catastrophic precision-loss in the subtraction of almost-equal large numbers

HALF PRECISION

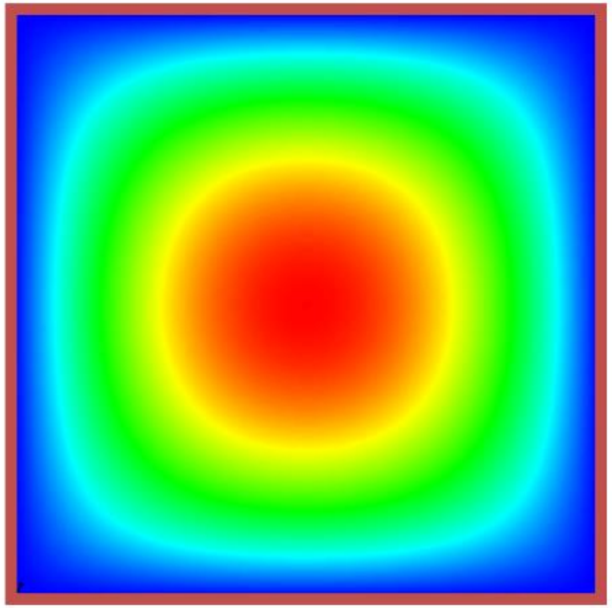


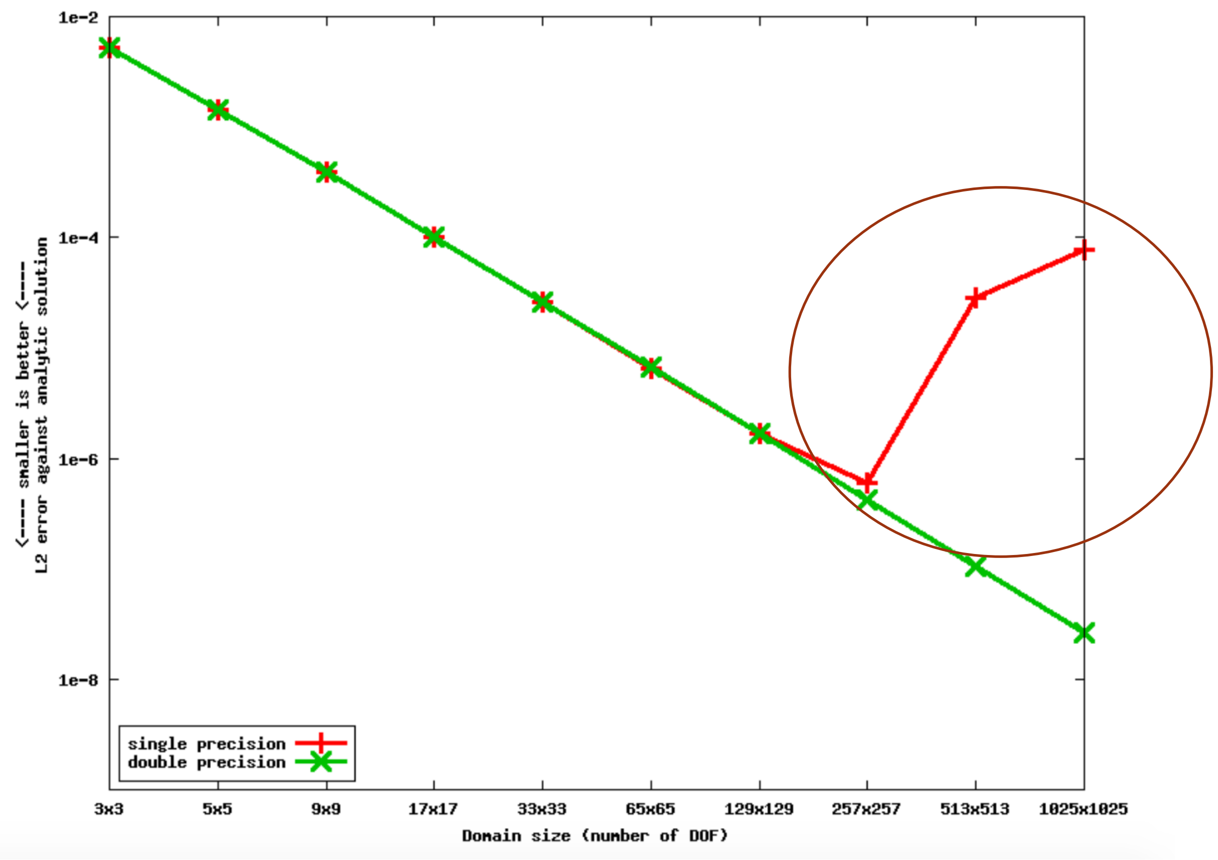
- Getting popular for some machine learning application
  - NVIDIA P100 can perform FP16 arithmetic at twice the throughput of FP32.
- Large number of parameters and the generally modest accuracy required for the final output – is this image a cat? or is this a fraudulent application?
- Training can be successful with floating point half precision (16 bits) or with fixed point or integers (as low as 8 bits in some cases).
- **Don't use it blindly in your codes: Check first!**

http://www.theregister.co.uk/2016/11/10/short_wide_deep_but_not_high/

SINGLE VS. DOUBLE PRECISION

- For some problem it does matter
- Poisson Equation $-\Delta u = f$
  - Finite elements





Strzodka et al.

http://www.nvidia.com/content/nvision2008/tech_presentations/NVIDIA_Research_Summit/NVISION08-Mixed_Precision_Methods_on_GPUs.pdf

Strzodka et al.
http://www.nvidia.com/content/nvision2008/tech_presentations/NVIDIA_Research_Summit/NVISION08-Mixed_Precision_Methods_on_GPUs.pdf

MIXED-PRECISION

- Exploit the speed of low precision and obtain a result of high accuracy
  - **Compute** in **high** precision (cheap)
  - **Solve** in **low** precision (fast)
  - **Correct** in **high** precision (cheap)
  - Iterate until convergence in high precision

$$d_k = b - Ax_k$$
$$Ac_k = d_k$$
$$x_{k+1} = x_k + c_k$$
$$k = k+1$$

- Now also half-precision in single precision codes
  - https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/

CONCURRENCY

- Concurrency makes it worse!
  - Operations a shared variables (e.g. reduction)
  - Concurrency implies unknown orders for operation
  - Inherent to concurrency; does not depend on the parallel model

- Worst as the degree of parallelism increases
  - For instance, on GPU codes using atomics

CONCLUSION

- Should you worry about the accuracy of every LoC you write?
- Study your problem /algorithm to understand what level of precision is required / acceptable
  - Usually the answer is already known by your community
- Verify your results / programs
  - Convergence tests, statistical tests, analytical solutions, …
- Check for performance bottlenecks
  - ☞ Other CoDaS' talks

REFERENCES

- *Optimal floating point computation: Accuracy, Precision, Speed in scientific computing.* Innocente. 2012
- *Handbook of Floating-Point Arithmetic*. Mueller et al. 2010
- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Goldberg. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html