

Numerical stability and fast-math

Speeding up LHCb software through compilation optimization

Oscar Buon

July 18, 2023

Supervisor: Sebastien Ponce

Table of contents

- 1 Introduction
 - Floating-point arithmetic
 - IEEE 754
- 2 Correctness of some computations
 - Floats equality
 - Exception cases
- 3 Fast-math
 - Exception handling
 - Approximation
- 4 Results
 - Improvement
 - Differences in the counters

- 1 Introduction
 - Floating-point arithmetic
 - IEEE 754
- 2 Correctness of some computations
 - Floats equality
 - Exception cases
- 3 Fast-math
 - Exception handling
 - Approximation
- 4 Results
 - Improvement
 - Differences in the counters

Floating-point arithmetic

$$\begin{aligned}
 (0.75)_{10} &= (+7.5 \times 10^{-1})_{10} \\
 &= (+11 \times 2^{-2})_2 \\
 &= (\underbrace{+}_{\text{sign}} \underbrace{1.1}_{\text{mantissa}} \times 2^{\overbrace{-1}^{\text{exponent}}})_2 = (0.11)_2
 \end{aligned}$$

Non-nominal case

$$(0.8)_{10} \approx (1.1001100110011001101 \times 2^{-1})_2$$

The representation is an approximation, because the number of bits is limited.

IEEE 754

- 32 bits (~ 7.2 digits) and 64 bits (~ 15.9 digits) formats ;
- Special rules:
 - Subnormal numbers (numbers very close to 0) ;
 - Special values: $-\text{Inf}$, $+\text{Inf}$, NaN ;
 - Rounding rules ;
 - Exception handling ;
- We should always consider floats as approximations !

- 1 Introduction
 - Floating-point arithmetic
 - IEEE 754
- 2 Correctness of some computations
 - Floats equality
 - Exception cases
- 3 Fast-math
 - Exception handling
 - Approximation
- 4 Results
 - Improvement
 - Differences in the counters

Floats equality

- Never check for float equality.
- GCC flag `-Wfloat-equal`. Ex. in LHCb:
 - `0 == ip`
 - `0.5 == ip`
 - `lhs.m_energy == rhs.m_energy`

Exception cases

```
if (f(x) != 0.)  
    return 5./f(x);  
else  
    ...
```

- Maybe the two $f(x)$ will not be the same.
- Due to different optimizations.

```
float y = f(x);  
if (y != 0.)  
    return 5./y;  
else  
    ...
```

- Maybe $f(x)$ will still be calculated twice.
- If $f(x)$ is close to 0 then $5./y$ could be $+\text{Inf}$.

A good way is using `isfinite` after the computation.

```
float result = 5./f(x);  
if (std::isfinite(result))  
    return result;  
else  
    ...
```

There are also:

- The trapping system (registering a handler with *C* function `signal(handler)`);
- The signaling system via
`std::numeric_limits<T>::signaling_NaN`;

but they are old methods and are not recommended.

- 1 Introduction
 - Floating-point arithmetic
 - IEEE 754
- 2 Correctness of some computations
 - Floats equality
 - Exception cases
- 3 **Fast-math**
 - Exception handling
 - Approximation
- 4 Results
 - Improvement
 - Differences in the counters

Principle

- Set of flags.
- Making mathematically valid optimizations but not respecting Standard:
 - $a + b - a = b$.
 - With $a = 10^6$ and $b = 10^{-6}$,
Standard gives 0, fast-math gives 10^{-6} .
- Different results but not more wrong than without fast-math.

<https://godbolt.org/z/jqz8P8vjj>

no-math-errno

- Do not set `errno` after calling math functions that are executed with a single instruction, e.g., `sqrt`.

no-signaling-nans

- Compile code assuming that IEEE signaling NaNs may not generate user-visible traps during floating-point operations.
- Enables optimizations that may change the number of exceptions visible with signaling NaNs.
- Enabled by default.

no-trapping-math

- Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation.
- Implies no-signaling-nans.

finite-math-only

- Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or $\pm\text{Infs}$.
- Compiler can replace `isnan` by `false`. Actually it is an undefined behavior.
- Main source of issues from `fast-math`.

no-signed-zeros

- Allow optimizations for floating-point arithmetic that ignore the signedness of zero.
- Ex. $0.0+x$ or $0.0*x$.

associative-math

- Allow re-association of operands in series of floating-point operations.
- Can optimize $2.0 * x * 3.0$ in $6.0 * x \implies$ make some computations at compile time.
- May allow a better vectorization and use of *FMA*.
- Needs no-signed-zeros and no-trapping-math.

reciprocal-math

- Allow the reciprocal of a value to be used instead of dividing by the value if this enables optimizations.
- Ex. Replacing x/y by $x*(1/y)$.
- May decrease precision.

unsafe-math-optimizations

- Allow optimizations for floating-point arithmetic that
 - 1 assume that arguments and results are valid and
 - 2 may violate IEEE or ANSI standards.
- When used at link time, it may include libraries or startup files that change the default FPU control word or other similar optimizations.
- Can affect dynamically included libraries.
- Enables `-fno-signed-zeros`, `-fno-trapping-math`, `-fassociative-math` and `-freciprocal-math`.

- 1 Introduction
 - Floating-point arithmetic
 - IEEE 754
- 2 Correctness of some computations
 - Floats equality
 - Exception cases
- 3 Fast-math
 - Exception handling
 - Approximation
- 4 Results
 - Improvement
 - Differences in the counters

Improvement

Test: hlt2_pp_thor

Optimization	Improvement	Confidence interval (2σ)
Fast-math ¹	5.06%	$\pm 0.98\%$
Associative-math only	4.73%	$\pm 1.51\%$
Fast-math ¹ + LTO & PGO	11.02%	$\pm 0.98\%$

¹without finite-math-only and unsafe-math-optimizations

Time used by CPU for computing floats:

FP Arithmetic	10.9%
FP x87	0.0%
FP Scalar	6.7%
FP Vector	4.2%

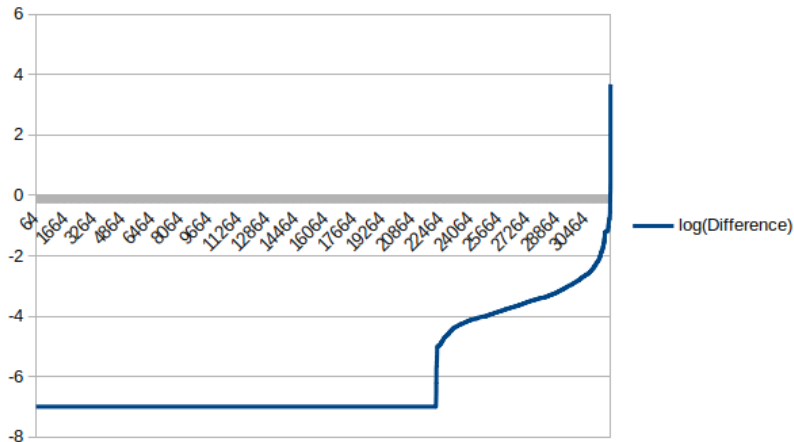
Figure: Reference

FP Arithmetic	9.5%
FP x87	0.0%
FP Scalar	5.9%
FP Vector	3.6%

Figure: Associative-math

Differences in the counters

Difference with the reference (all counters):



In a first time

- Enabling `-Wfloat-equal`.
- Enabling `fast-math` in one slot for instability checking.
 - Forget about `finite-math-only` and `unsafe-math-optimizations`.

In a second time

- Switch to fast-math for production.
 - Sometimes better precision (*FMA* and *double-precision-constant*).
 - Coding clear equations and letting compiler optimize them.
 - GPU already using similar principles.
 - Dropping features that are legacies.
 - Assuming floats are approximations.