Fast-math flags: a bag of issues and a handful of solutions

Intel Corporation
EURO LLVM 2026, Dublin
April 2026

Mikołaj Piróg

Roadmap

- What (and why) are fast-math flags?
- Semantics of fast-math flags
- Respecting the semantics
- Outlook for the future
- Existing efforts

What are fast-math flags

- A way for users to allow certain transformations to appear
- Realized on IR level as flags attached to floating-point operations

```
define float @before_contract(float %a, float %b, float  %c) {
entry:
  %mul = fmul contract float %b, %a
  %add = fadd contract float %mul, %c
  ret float %add
}

define float @after_contract(float %a, float %b, float  %c) {
entry:
  %res = call contract float @llvm.fma.f32(float %a, float %b, float %c)
  ret float %res
}
```

List of fast-math flags

Value-based flags:

- **nnan** – no NANs
- **ninf** – no infinity
- **nsz** – no signed zeros
- Modeled in Alive2

Rewrite-based flags:

- **arcp** – allow reciprocal
- **contract** – contract operations
- **afn** – approximate functions
- **reassoc** – reassociation
- Not well-defined – hard to model, handling by LLVM can be contradictory at times

fast meta flag – a union of all seven flags

arcp flag

`a / b -> a * (1 / b)`

- Well-defined semantics

contract flag

`a * b + c -> fma(a, b, c)`

- Most commonly understood as allowing the appearance of FMA (and as such defined in the LangRef)
- Such understanding is limiting - why not further transformations?

afn flag and library issues

- Doesn’t say how approximate the functions are
- Precision of functions ultimately depend on library
- Library we are compiling with is not necessarily the one that will be present during execution, further complicating things
- Various libraries offer different accuracies; the trend is to improve it. Comprehensive comparison Accuracy of Mathematical Functions in Single, Double, Double Extended, and Quadruple Precision

reassoc flag

`a * b + a * c -> a * (b + c)`

- Most widely misused flag
- Also most widely defined: ”Allow algebraically equivalent transformations …”

fast meta-flag

- Simply a union of all flags
- Convenient for frontends, IR users
- Doesn’t make sense for optimizations to check for **fast** meta-flag

Respecting the semantics

Respecting the semantics of fast-math flags means:

- Checking required flags for given transformation
- Checking flags on appropriate nodes taking part in the transformation
- Properly setting flags for newly created nodes

Nothing on sqrt

DAGCombiner.cpp:visitFDIV

```
define float @source(float %f) local_unnamed_addr {
  %sqrt = tail call float @llvm.sqrt.f32(float %f)
  %div = fdiv arcp float 1.000000e+00, %sqrt
  ret float %div
}

define float @target(float %f) local_unnamed_addr {
  %rsqrt = tail call float @llvm.rsqrt.f32(float %f)
  ret float %rsqrt
}
```

This transformation shouldn’t happen because **sqrt** doesn’t have any fast-math flags

Reassoc use

InstCombineMulDivRem.cpp:visitFdiv

```
define float @source(float %x) {
  %sin = call reassoc float @llvm.sin.f32(float %x)
  %cos = call reassoc float @llvm.cos.f32(float %x)
  %result = fdiv reassoc float %sin, %cos
  ret float %result
}

define float @target(float %x) {
  %tan = call reassoc float @llvm.tan.f32(float %x)
  %result = fdiv reassoc float 1.000000e+00, %tan
  ret float %result
}
```

This rewrite goes much beyond what typical is understood as reassociation

Wrong flag propagation

X86ISelLowering.cpp:combineFneg

```
define double @source(double %x, double %y) {
  %mul = fmul contract nsz afn double %x, %y
  %neg = fneg contract afn double %mul
  ret double %neg
}

; After x86isel:
; ....
;   %3:fr64x = nofpexcept VFNMSUB213SDZr %1:fr64x(tied-def 0), %0:fr64x, killed
    %2:fr64x, implicit $mxcsr
; ...
```

Newly created node should have fast-math flags set

IsFast usage

InstCombineMulDivRem.cpp:visitFMul

```
  // ...
  // log2(X * 0.5) * Y = log2(X) * Y - Y
  if (I.isFast()) {
    IntrinsicInst *Log2 = nullptr;
    if (match(Op0, m_OneUse(m_Intrinsic<Intrinsic::log2>(
           m_OneUse(m_FMul(m_Value(X), m_SpecificFP(0.5)))))) {
      Log2 = cast<IntrinsicInst>(Op0);
      Y = Op1;
    }
// ...
```

Checking fast here is too restrictive

The future

Updating the flags

- Extend definition of contract
- Joshua Cranmer PR:187340 to extend the reach of contract beyond FMA
- Split reassoc into reassoc-fundamental and reassoc-all
- Consider augmenting afn with precision information
- Think about the flags as ordered by precision-loss, not speed benefits
- In general flags shouldn’t do too much
- It’s not that easy to simply add flags to the IR

Agreeing on semantics and formal verification

- Come up with better definitions of the flags
- Already there to some extent - Alive2 support for value-based flags
- Could we extend Alive2 to cover rewrite flags?
- What about flag propagation?
- Agreeing on semantics as a prerequisite

Why is it all worthwhile?

- To be truly useful, semantics of each flag need to well-defined and allow for fine-level control
- Wrong flag propagation inhibits optimizations
- Fuzzy semantics are not encouraging to use
- It’s also about correctness

Existing and previous efforts

(This list is no-exhaustive)

- Joshua Cranmer:
  - PR187340 - update semantics of contract
  - PR99557 - adjust documentation of some fast-math flags
  - Contract RFC
- Paperchalice:
  - PR185582 - remove uses of global ”no NANs” options from x86
  - PR163504 - remove uses of global ”no NANs” options from DAGCombiner
  - ... and many more
- Andy Kaylor: PR105746 - honor pragma with -ffp-contract=fast option, deprecate -ffp-contract=fast-honor-pragmas
- A few by me:
  - PR167252 - stop relying on global contract flag in x86 backend
  - PR167595 - honor rewrite semantics in DAGCombiner visitFDIV
  - PR174291 - honor rewrite semantics of fast-math flags in InstCombineMulDivRem


intel

- Joshua's changes to introduce more bits for fast-math flags: link
- Paperchalice's PR:191190 with similar idea
- C standard doesn't force all functions to be correctly rounded (diverging from IEEE754). This is a source of headaches
- How to deal with constant folding? Currently constant folding is different based on the library. Have APFloat support for correctly rounded functions?
- Value based flags don't really have flag propagation - they could appear after some rewrite
- There were minor issues about correctly propagating the flags from IR to LLVM internals and auto-upgrade
- Various interfaces (matchers) should be aware about fast-math flags (already being done)
- Is MLIR doing things differently? If we changes flags in LLVM, what will happen in MLIR?

- C standard offers a notion of contraction within an expression
- In Fortran expression in brackets should be evaluated as is
- IEEE754 has a notion of ”value changing optimizations” (10.4)