Towards Useful Fast-Math

...

Andy Kaylor

andy_kaylor@yahoo.com

LLVM Developers' Meeting, October 2024

Fast-Math Considered Harmful

“-ffast-math

Might allow some programs designed to not be too dependent on IEEE behavior for floating-point to run faster, or die trying.” (gcc 3.4.6 documentation)

“Ofast [means] ‘make it faster and wronger please’” (Jon Chesterton, discussion of deprecating -Ofast)

“Beware of fast-math” (Simon Byrne, https://simonbyrne.github.io/notes/fastmath/)

Problems with fast-math

- **Fast-math will almost certainly change your numeric results**
  - **(A * B) * C → A * (B * C)**
- Fast-math may optimize away explicit checks for NaN or infinity
  - if (std::isnan(x)) → if (false)
- Fast-math may turn off support for denormal values
  - FTZ/DAZ are set during program initialization
- Fast-math can lead to **complete** loss of precision
  - (A - B) + Epsilon → (A + Epsilon) - B

Problems with fast-math

- Fast-math will almost certainly change your numeric results
  - (A * B) * C → A * (B * C)
- **Fast-math may optimize away explicit checks for NaN or infinity**
  - **if (std::isnan(x)) → if (false)**
- Fast-math may turn off support for denormal values
  - FTZ/DAZ are set during program initialization
- Fast-math can lead to **complete** loss of precision
  - (A - B) + Epsilon → (A + Epsilon) - B

Problems with fast-math

- Fast-math will almost certainly change your numeric results
  - (A * B) * C → A * (B * C)
- Fast-math may optimize away explicit checks for NaN or infinity
  - if (std::isnan(x)) → if (false)
- **Fast-math may turn off support for denormal values**
  - **FTZ/DAZ are set during program initialization**
- Fast-math can lead to **complete** loss of precision
  - (A - B) + Epsilon → (A + Epsilon) - B

Problems with fast-math

- Fast-math will almost certainly change your numeric results
  - (A * B) * C → A * (B * C)
- Fast-math may optimize away explicit checks for NaN or infinity
  - if (std::isnan(x)) → if (false)
- Fast-math may turn off support for denormal values
  - FTZ/DAZ are set during program initialization
- **Fast-math can lead to complete loss of precision**
  - **(A - B) + Epsilon → (A + Epsilon) - B**

A Worst Case Example

```
float foo() {
  float A = 1.0f;
  float C = 1.0f;

  // Find the smallest value A = 2^k for which (A + 1 - A) != 1
  do {
    A *= 2.0f;
    C = A + 1.0f - A;
  } while (C == 1.0f);

  return A;
}
```

A Worst Case Example

```
define dso_local noundef nofpclass(nan inf) float @_Z3foov() {
entry:
  unreachable
}
```

So why would anyone use this?

- Better optimization through algebra
  - (A * B) * C → A * (B *C)
- Vectorization
  - Vector operations often require reassociation
- Eliminate restrictions/handling for special cases
  - X - X → 0.0

So why would anyone use this?

- **Better optimization through algebra**
  - **(A * B) * C → A * (B *C)**
- Vectorization
  - V
- Elimin
  - X

```
// Reassociation will allow hoisting X * Y
for (i = 0; i < n; ++i)
  A[i] = X * B[i] * Y;
```

So why would anyone use this?

- Better optimization through algebra
  - (A * B) * C → A * (B *C)
- **Vectorization**
  - **Vector operations often require reassociation**
- Elim
  -

```
// This requires reassociation to vectorize
for (i = 0; i < n; ++i)
  sum += x[i];
```

So why would anyone use this?

- Better optimization through algebra
  - (A * B) * C → A * (B *C)
- Vectorization
  - Vector operations often require reassociation
- **Eliminate restrictions/handling for special cases**
  - **X - X → 0.0**

Fast-math, unsafe math, and fp-model

- GCC defines two broad options for fast-math: `-ffast-math` and `-funsafe-math-optimizations`
  - Which one sounds riskier?
  - `-ffast-math` is more aggressive – it assumes no NaN or infinite values will be seen
- In clang, we offer `-ffp-model` to broadly control floating-point semantics
  - `-ffp-model=fast` is roughly equivalent to `-funsafe-math-optimizations`
  - `-ffp-model=aggressive` is roughly equivalent to `-ffast-math`
- Many other options let you control individual semantics
  - `-f[no-]math-errno`
  - `-f[no-]honor-nans, -f[no-]honor-infinities`
  - `-ffp-contract=[on|off|fast]`
  - `-f[no-]associative-math`
  - `-f[no-]reciprocal-math`

Pragmas for local control

#pragma float_control([push|pop])

#pragma float_control(precise, [on|off])

#pragma clang fp reassociate([on|off])

#pragma clang fp reciprocal([on|off])

#pragma STDC FP_CONTRACT [ON|OFF|DEFAULT]

A general approach

1. Compile everything with `-ffp-model=fast`
2. Test for acceptable results
3. Disable fast-math for a subset of files until tests pass
4. Re-enable fast-math and use pragmas to isolate sensitive functions
5. Move pragmas into local scopes in the sensitive functions

What else are can the compiler do?

I propose a feature to allow the user to interactively disable fast-math optimizations

Conceptually similar to opt-bisect, but more user-oriented

Would require new infrastructure to request permission to perform a fast-math transformation

```
allowFastMath("X / (X * Y) --> 1.0 / Y", &I, Op1);
```

Output would inform users what was happening

> "Allowing fast-math (1): 'X / (X * Y) --> 1.0 / Y' at src.cpp, line 26"

Typical fast-math optimization

```
// X / (X * Y) --> 1.0 / Y
// Reassociate to (X / X -> 1.0) is legal when NaNs are not allowed.
// We can ignore the possibility that X is infinity because INF/INF is NaN.
Value *X, *Y;
if (I.hasNoNaNs() && I.hasAllowReassoc() &&
    match(Op1, m_c_FMul(m_Specific(Op0), m_Value(Y)))) {
  replaceOperand(I, 0, ConstantFP::get(I.getType(), 1.0));
  replaceOperand(I, 1, Y);
  return &I;
}
```

Proposed fast-math optimization

```
// X / (X * Y) --> 1.0 / Y
// Reassociate to (X / X -> 1.0) is legal when NaNs are not allowed.
// We can ignore the possibility that X is infinity because INF/INF is NaN.
Value *X, *Y;
if (I.hasNoNaNs() && I.hasAllowReassoc() && Op1->hasAllowReassoc()&&
    match(Op1, m_c_FMul(m_Specific(Op0), m_Value(Y)))) {
  if (allowFastMath("X / (X * Y) --> 1.0 / Y", Op1, &I)) {
    replaceOperand(I, 0, ConstantFP::get(I.getType(), 1.0));
    replaceOperand(I, 1, Y);
    return &I;
  }
}
```

```
Allowing fast-math (1): 'X / (X * Y) --> 1.0 / Y' at src.cpp, line 26
  %mul = fmul fast double %x, %y
  %div = fdiv fast double %x, %mul
```