Optimizations enabled by -ffast-math

This blog post describes the optimizations enabled by `-ffast-math` when compiling C or C++ code with GCC 11 for x86_64 Linux (other languages/operating systems/CPU architectures may enable slightly different optimizations).

`-ffast-math`

Most of the "fast math" optimizations can be enabled/disabled individually, and `-ffast-math` enables all of them:[1]

- `-ffinite-math-only`
- `-fno-signed-zeros`
- `-fno-trapping-math`
- `-fassociative-math`
- `-fno-math-errno`
- `-freciprocal-math`
- `-funsafe-math-optimizations`
- `-fcx-limited-range`

When compiling standard C (that is, when using `-std=c99` etc. instead of the default "GNU C" dialect), `-ffast-math` also enables `-ffp-contract=fast`, allowing the compiler to combine multiplication and addition instructions with an `FMA` instruction (godbolt). C++ and GNU C are not affected as `-ffp-contract=fast` is already the default for them.

Linking using `gcc` with `-ffast-math` does one additional thing – it makes the CPU treat subnormal numbers as 0.0.

Treating subnormal numbers as 0.0

The floating-point format has a special representation for values that are close to 0.0. These "subnormal" numbers (also called "denormals") are very costly[2] in some cases because the CPU handles subnormal results using microcode exceptions.

The x86_64 CPU has a feature to treat subnormal input as 0.0 and flush subnormal results to 0.0, eliminating this performance penalty. This can be enabled by

```
#define MXCSR_DAZ (1<<6)    /* Enable denormals are zero mode */
#define MXCSR_FTZ (1<<15)   /* Enable flush to zero mode */

unsigned int mxcsr = __builtin_ia32_stmxcsr();
mxcsr |= MXCSR_DAZ | MXCSR_FTZ;
__builtin_ia32_ldmxcsr(mxcsr);
```

Linking using `gcc` with `-ffast-math` makes it disable subnormal numbers for the application by adding this code in a global constructor that runs before `main`.

`-ffinite-math-only` and `-fno-signed-zeros`

Many optimizations are prevented by properties of the floating-point values `NaN`, `Inf`, and `-0.0`. For example:

- `x+0.0` cannot be optimized to `x` because that is not true when `x` is `-0.0`.
- `x-x` cannot be optimized to `0.0` because that is not true when `x` is `NaN` or `Inf`.
- `x*0.0` cannot be optimized to `0.0` because that is not true when `x` is `NaN` or `Inf`.
- The compiler cannot transform

  ```
  if (x > y) {
    do_something();
  } else {
    do_something_else();
  }
  ```

  to the form

  ```
  if (x <= y) {
    do_something_else();
  } else {
    do_something();
  }
  ```

(which is a useful optimization when simplifying control flow and ensuring that the common case is handled without taking branches) because that is not true when `x` or `y` is `NaN`.

`-ffinite-math-only` and `-fno-signed-zeros` tell the compiler that no calculation will produce `NaN`, `Inf`, or `-0.0`, so the compiler can then do the kind of optimization described above.

Note: The program may behave in strange ways (such as not evaluating either the true or false part of an `if`-statement) if calculations produce `Inf`, `NaN`, or `-0.0` when these flags are used.

`-fno-trapping-math`

It is possible to enable trapping of floating-point exceptions by using the GNU libc function `feenableexcept` to generate the signal `SIGFPE` when any floating-point instruction overflow, underflow, generates `NaN`, etc. For example, the function `compute` below does some calculations that overflow to `Inf`, which makes the program terminate with `SIGFPE` when `FE_OVERFLOW` is enabled.

```
// Compile as "gcc example.c -D_GNU_SOURCE -O2 -lm"
#include <stdio.h>
#include <fenv.h>

void compute(void) {
  float f = 2.0;
  for (int i = 0; i < 7; ++i) {
    f = f * f;
    printf("%d: f = %f\n", i, f);
  }
}

int main(void) {
  compute();

  printf("\nWith overflow exceptions:\n");
  feenableexcept(FE_OVERFLOW);
  compute();
```

```
  return 0;
}
```

This means that the compiler cannot schedule floating-point instructions to execute speculatively, as the speculated instruction could then generate a signal for cases where the original program did not. For example, the calculation `x/y` is constant in the loop below, but the compiler cannot hoist it out of the loop because that could cause the function to execute `x/y` for cases where all elements of `arr` are larger than `0.0` and could therefore crash for cases where the original program did not (godbolt).

```
double arr[1024];

void foo(int n, double x, double y) {
  for (int i = 0; i < n; ++i) {
    if (arr[i] > 0.0)
      arr[i] = x / y;
  }
}
```

One other fun special case is C floating-point atomics. The C standard requires that floating-point exceptions are discarded when doing compound assignment (see "compound assignment" in the C standard), so the compiler must insert extra code unless trapping-math is disabled (godbolt).

Passing `-fno-trapping-math` tells the compiler that the program will not enable floating-point exceptions, and the compiler can then do these optimizations.

`-fassociative-math`

`-fassociative-math` allows re-association of operands in series of floating-point operations (as well as a few more general reordering optimizations). Most of the optimizations need `-fno-trapping-math`, `-ffinite-math-only`, and `-fno-signed-zeros` too.

Some examples of `-fassociative-math` optimizations:

Original	Optimized
`(X + Y) - X`	`Y`

Original	Optimized
`(X * Z) + (Y * Z)`	`(X + Y) * Z`
`(X * C) + X`	`X * (C + 1.0)` when `C` is a constant
`(C1 / X) * C2`	`(C1 * C2) / X` when `C1` and `C2` are constants
`(C1 - X) < C2`	`(C1 - C2) > X` when `C1` and `C2` are constants

Re-association is especially useful for vectorization. Consider, for example, the loop (godbolt)

```
float a[1024];

float foo(void) {
  float sum = 0.0f;
  for (int i = 0; i < 1024; ++i) {
    sum += a[i];
  }
  return sum;
}
```

All additions to `sum` are made serially, so the calculations cannot normally be vectorized. But `-fassociative-math` permits the compiler to change the order to

```
sum = (a[0] + a[4] + ... + a[1020]) + (a[1] + a[5] + ... + a[1021
```

and can compile the loop as if it was written as

```
float a[1024];

float foo(void) {
  float sum0 = sum1 = sum2 = sum3 = 0.0f;
  for (int i = 0; i < 1024; i += 4) {
    sum0 += a[i    ];
    sum1 += a[i + 1];
    sum2 += a[i + 2];
    sum3 += a[i + 3];
  }
```

```
  return sum0 + sum1 + sum2 + sum3;
}
```

which is easy to vectorize.

`-fno-math-errno`

The C mathematical functions may set `errno` if called with invalid input. This possible side effect means that the compiler must call the libc function instead of using instructions that can calculate the result directly.

The compiler can, in some cases, mitigate the problem by not calling the function when it knows the operation will succeed. For example, (godbolt)

```
double foo(double x) {
  return sqrt(x);
}
```

is compiled to code using the `sqrtsd` instruction when the input is in range and only calls `sqrt` for input that will return `NaN`

```
foo:
        pxor    xmm1, xmm1
        ucomisd xmm1, xmm0
        ja      .L10
        sqrtsd  xmm0, xmm0
        ret
.L10:
        jmp     sqrt
```

This eliminates most of the overhead (as the comparison/branch is essentially free when predicted correctly), but this extra branch and function call makes life harder for other optimizations (such as vectorization).

`-fno-math-errno` makes GCC optimize all math functions as if they do not set `errno` (that is, the compiler does not need to call the libc functions if the architecture has suitable instructions).

Non-math functions

One surprising effect of `-fno-math-errno` is that it makes GCC believe that memory-allocating libc functions (such as `malloc` and `strdup`) do not set `errno`. This can be seen in the function below where `-fno-math-errno` makes GCC optimize away the call to `perror` (godbolt)

```
void *foo(size_t size) {
  errno = 0;
  void *p = malloc(size);
  if (p == NULL) {
    if (errno)
      perror("error");
    exit(1);
  }
  return p;
}
```

This is reported as GCC bug 88576.

`-freciprocal-math`

`-freciprocal-math` allows the compiler to compute `x/y` as `x*(1/y)`. This is useful for code of the form (godbolt)

```
float length = sqrtf(x*x + y*y + z*z);
x = x / length;
y = y / length;
z = z / length;
```

where the compiler now can generate the code as if it was written as

```
float t = 1.0f / sqrtf(x*x + y*y + z*z);
x = x * t;
y = y * t;
z = z * t;
```

This optimization generates more instructions, but the resulting code is, in general, better as multiplication is faster than division and can execute on more ports in the CPU.

`-funsafe-math-optimizations`

`-funsafe-math-optimizations` enables various "mathematically correct" optimizations that may change the result because of how floating-point numbers work. Some examples of this are

Original	Optimized
`sqrt(x)*sqrt(x)`	`x`
`sqrt(x)*sqrt(y)`	`sqrt(x*y)`
`exp(x)*exp(y)`	`exp(x+y)`
`x/exp(y)`	`x*exp(-y)`
`x*pow(x,c)`	`pow(x,c+1)`
`pow(x,0.5)`	`sqrt(x)`
`(int)log(d)`	`ilog(d)`
`sin(x)/cos(x)`	`tan(x)`

Note: Many of these optimizations need additional flags, such as `-ffinite-math-only` and `-fno-math-errno`, in order to trigger.

`-funsafe-math-optimizations` also enables

- `-fno-signed-zeros`
- `-fno-trapping-math`
- `-fassociative-math`
- `-freciprocal-math`

as well as `-ffp-contract=fast` (for standard C) and treating subnormal numbers as 0.0 in the same way as described for `-ffast-math`.

`-fcx-limited-range`

The mathematical formulas for multiplying and dividing complex numbers are

$$(a+ib)\times(c+id)=(ac-bd)+i(bc+ad)$$

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}$$

but these does not work well for floating-point values.

One problem is that the calculations may turn overflowed values into `NaN` instead of `Inf`, so the implementation needs to adjust. Multiplication ends up with code similar to

```
double complex
mul(double a, double b, double c, double d) {
  double ac, bd, ad, bc, x, y;
  double complex res;

  ac = a * c;
  bd = b * d;
  ad = a * d;
  bc = b * c;

  x = ac - bd;
  y = ad + bc;

  if (isnan(x) && isnan(y)) {
    /* Recover infinities that computed as NaN + iNaN.  */
    _Bool recalc = 0;
    if (isinf(a) || isinf(b)) {
      /* z is infinite.  "Box" the infinity and change NaNs
       * in the other factor to 0.  */
      a = copysign(isinf(a) ? 1.0 : 0.0, a);
      b = copysign(isinf(b) ? 1.0 : 0.0, b);
      if (isnan(c)) c = copysign(0.0, c);
      if (isnan(d)) d = copysign(0.0, d);
      recalc = 1;
    }
    if (isinf(c) || isinf(d)) {
      /* w is infinite.  "Box" the infinity and change NaNs
       * in the other factor to 0.  */
      c = copysign(isinf(c) ? 1.0 : 0.0, c);
      d = copysign(isinf(d) ? 1.0 : 0.0, d);
      if (isnan(a)) a = copysign(0.0, a);
      if (isnan(b)) b = copysign(0.0, b);
```

```
      recalc = 1;
    }
    if (!recalc
        && (isinf(ac) || isinf(bd)
            || isinf(ad) || isinf(bc))) {
      /* Recover infinities from overflow by changing NaNs
       * to 0.  */
      if (isnan(a)) a = copysign(0.0, a);
      if (isnan(b)) b = copysign(0.0, b);
      if (isnan(c)) c = copysign(0.0, c);
      if (isnan(d)) d = copysign(0.0, d);
      recalc = 1;
    }
    if (recalc) {
      x = INFINITY * (a * c - b * d);
      y = INFINITY * (a * d + b * c);
    }
  }

  __real__ res = x;
  __imag__ res = y;
  return res;
}
```

One other problem is that the calculations can overflow even when the result of the operation is in range. This is especially problematic for division, so the implementation of division needs to add even more extra code (see the C standard for more details).

`-fcx-limited_range` makes the compiler use the usual mathematical formulas for complex multiplication/division.

---

1. It also enables `-fno-signaling-nans`, `-fno-rounding-math`, and `-fexcess-precision=fast`, which are enabled by default when compiling C or C++ code for x86_64 Linux, so I will not describe them in this blog post. ↩

2. For example, Agner Fog's microarchitecture document says that Broadwell has a penalty of approximately 124 clock cycles when an operation on normal numbers gives a subnormal result. ↩

Written on October 19, 2021