FUN, SAFE, MATH OPTIMIZATIONS

Martin Hořeňovský

PEX

FUN, SAFE, MATH OPTIMIZATIONS

Martin Hořeňovský

PEX

ABOUT THIS TALK

We will talk about fun and safe math optimizations

We will talk about fun and safe math optimizations

Or `-funsafe-math-optimizations` with GCC/Clang

We will talk about fun and safe math optimizations

Or `-funsafe-math-optimizations` with GCC/Clang

Who doesn't want fun and safe optimizations?

GCC functionality flags are prefixed with `-f`.

GCC functionality flags are prefixed with `-f`.

`-funsafe-math-optimizations` is actually `-f`**`unsafe-math-optimizations`**

unsafe optimizations are not fun

THE END

ABOUT THIS TALK

ABOUT THIS TALK

(FOR REAL THIS TIME)

We will talk about various flags that govern floating point optimizations in the compiler

We will talk about various flags that govern floating point optimizations in the compiler

And what their effect is on the generated code

I will also try to convince you not to use most of them

FLAGS - OVERVIEW

`-Ofast`

`-Ofast`

`-Ofast` seems unrelated, but it enables `-ffast-math`

`-ffast-math`

`-ffast-math`

Who doesn't want their math to be fast?

-ffast-math enables

- -fno-math-errno
- -funsafe-math-optimizations
- -ffinite-math-only
- -fno-rounding-math
- -fno-signaling-nans
- -fcx-limited-range
- -fexcess-precision=fast

-ffast-math enables

- -fno-math-errno
- -funsafe-math-optimizations
- -ffinite-math-only
- -fno-rounding-math
- -fno-signaling-nans
- -fcx-limited-range
- -fexcess-precision=fast

`-funsafe-math-optimizations`

Links in `crtfastmath.o`, which forces flush to zero for whole program.

Also enables

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

~~Links in `crtfastmath.o`, which forces flush to zero for whole program.~~

Also enables

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

That's a lot of flags from -ffast-math.

That's a lot of flags from -ffast-math.

Are you sure they won't break your code?

Don't use -ffast-math

Don't use -ffast-math

Use individual flags **after** you've read documentation and understand what they change.

Let's take a look at some of them

FLAGS - DETAILS

`-fno-math-errno`

`-fno-math-errno`

Skips setting `errno` for simple math functions

`-fno-math-errno`

Skips setting `errno` for simple math functions

This can help with vectorization

Example from my talk last month:

```
void do_sqrtf(std::span<float> data) {
    for (float& f : data) {
        f = sqrtf(f);
    }
}
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 4 lines of prologue

        vxorps  xmm1, xmm1, xmm1
.L6:
        vmovss  xmm0, DWORD PTR [rdi]
        vucomiss        xmm1, xmm0
        ja      .L8
        vsqrtss xmm0, xmm0, xmm0
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rdi, rax
        jne     .L6
.L8:
        mov     QWORD PTR [rsp+8], rax
        mov     QWORD PTR [rsp], rdi
        call    sqrtf
        mov     rdi, QWORD PTR [rsp]
        mov     rax, QWORD PTR [rsp+8]
        vxorps  xmm1, xmm1, xmm1
        vmovss  DWORD PTR [rdi], xmm0
        add     rdi, 4
        cmp     rax, rdi
        jne     .L6

        ; 4 lines of prologue
```

```
; GCC w/ -fno-math-errno
do_sqrtf(std::span<float, 18446744073709551615
        ; 16 lines of prologue

.L4:
        vsqrtps ymm0, YMMWORD PTR [rax]
        add     rax, 32
        vmovups YMMWORD PTR [rax-32], ymm0
        cmp     rax, rdx
        jne     .L4
        mov     rdx, r9
        and     rdx, -8
        and     r9d, 7
        lea     rax, [rcx+rdx*4]
        je      .L21
        vzeroupper

        ; loops for <32 elements
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 4 lines of prologue

        vxorps  xmm1, xmm1, xmm1
.L6:
        vmovss  xmm0, DWORD PTR [rdi]
        vucomiss        xmm1, xmm0
        ja      .L8
        vsqrtss xmm0, xmm0, xmm0
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rdi, rax
        jne     .L6
.L8:
        mov     QWORD PTR [rsp+8], rax
        mov     QWORD PTR [rsp], rdi
        call    sqrtf
        mov     rdi, QWORD PTR [rsp]
        mov     rax, QWORD PTR [rsp+8]
        vxorps  xmm1, xmm1, xmm1
        vmovss  DWORD PTR [rdi], xmm0
        add     rdi, 4
        cmp     rax, rdi
        jne     .L6

        ; 4 lines of prologue
```

```
; GCC w/ -fno-math-errno
do_sqrtf(std::span<float, 18446744073709551615
        ; 16 lines of prologue

.L4:
        vsqrtps ymm0, YMMWORD PTR [rax]
        add     rax, 32
        vmovups YMMWORD PTR [rax-32], ymm0
        cmp     rax, rdx
        jne     .L4
        mov     rdx, r9
        and     rdx, -8
        and     r9d, 7
        lea     rax, [rcx+rdx*4]
        je      .L21
        vzeroupper

        ; loops for <32 elements
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 4 lines of prologue

        vxorps  xmm1, xmm1, xmm1
.L6:
        vmovss  xmm0, DWORD PTR [rdi]
        vucomiss        xmm1, xmm0
        ja      .L8
        vsqrtss xmm0, xmm0, xmm0
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rdi, rax
        jne     .L6
.L8:
        mov     QWORD PTR [rsp+8], rax
        mov     QWORD PTR [rsp], rdi
        call    sqrtf
        mov     rdi, QWORD PTR [rsp]
        mov     rax, QWORD PTR [rsp+8]
        vxorps  xmm1, xmm1, xmm1
        vmovss  DWORD PTR [rdi], xmm0
        add     rdi, 4
        cmp     rax, rdi
        jne     .L6

        ; 4 lines of prologue
```

```
; GCC w/ -fno-math-errno
do_sqrtf(std::span<float, 18446744073709551615
        ; 16 lines of prologue

.L4:
        vsqrtps ymm0, YMMWORD PTR [rax]
        add     rax, 32
        vmovups YMMWORD PTR [rax-32], ymm0
        cmp     rax, rdx
        jne     .L4
        mov     rdx, r9
        and     rdx, -8
        and     r9d, 7
        lea     rax, [rcx+rdx*4]
        je      .L21
        vzeroupper

        ; loops for <32 elements
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 4 lines of prologue

        vxorps  xmm1, xmm1, xmm1
.L6:
        vmovss  xmm0, DWORD PTR [rdi]
        vucomiss        xmm1, xmm0
        ja      .L8
        vsqrtss xmm0, xmm0, xmm0
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rdi, rax
        jne     .L6
.L8:
        mov     QWORD PTR [rsp+8], rax
        mov     QWORD PTR [rsp], rdi
        call    sqrtf
        mov     rdi, QWORD PTR [rsp]
        mov     rax, QWORD PTR [rsp+8]
        vxorps  xmm1, xmm1, xmm1
        vmovss  DWORD PTR [rdi], xmm0
        add     rdi, 4
        cmp     rax, rdi
        jne     .L6

        ; 4 lines of prologue
```

```
; GCC w/ -fno-math-errno
do_sqrtf(std::span<float, 18446744073709551615
        ; 16 lines of prologue

.L4:
        vsqrtps ymm0, YMMWORD PTR [rax]
        add     rax, 32
        vmovups YMMWORD PTR [rax-32], ymm0
        cmp     rax, rdx
        jne     .L4
        mov     rdx, r9
        and     rdx, -8
        and     r9d, 7
        lea     rax, [rcx+rdx*4]
        je      .L21
        vzeroupper

        ; loops for <32 elements
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 4 lines of prologue

        vxorps  xmm1, xmm1, xmm1
.L6:
        vmovss  xmm0, DWORD PTR [rdi]
        vucomiss        xmm1, xmm0
        ja      .L8
        vsqrtss xmm0, xmm0, xmm0
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rdi, rax
        jne     .L6
.L8:
        mov     QWORD PTR [rsp+8], rax
        mov     QWORD PTR [rsp], rdi
        call    sqrtf
        mov     rdi, QWORD PTR [rsp]
        mov     rax, QWORD PTR [rsp+8]
        vxorps  xmm1, xmm1, xmm1
        vmovss  DWORD PTR [rdi], xmm0
        add     rdi, 4
        cmp     rax, rdi
        jne     .L6

        ; 4 lines of prologue
```

```
; GCC w/ -fno-math-errno
do_sqrtf(std::span<float, 18446744073709551615
        ; 16 lines of prologue

.L4:
        vsqrtps ymm0, YMMWORD PTR [rax]
        add     rax, 32
        vmovups YMMWORD PTR [rax-32], ymm0
        cmp     rax, rdx
        jne     .L4
        mov     rdx, r9
        and     rdx, -8
        and     r9d, 7
        lea     rax, [rcx+rdx*4]
        je      .L21
        vzeroupper

        ; loops for <32 elements
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 4 lines of prologue

        vxorps  xmm1, xmm1, xmm1
.L6:
        vmovss  xmm0, DWORD PTR [rdi]
        vucomiss        xmm1, xmm0
        ja      .L8
        vsqrtss xmm0, xmm0, xmm0
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rdi, rax
        jne     .L6
.L8:
        mov     QWORD PTR [rsp+8], rax
        mov     QWORD PTR [rsp], rdi
        call    sqrtf
        mov     rdi, QWORD PTR [rsp]
        mov     rax, QWORD PTR [rsp+8]
        vxorps  xmm1, xmm1, xmm1
        vmovss  DWORD PTR [rdi], xmm0
        add     rdi, 4
        cmp     rax, rdi
        jne     .L6

        ; 4 lines of prologue
```

```
; GCC w/ -fno-math-errno
do_sqrtf(std::span<float, 18446744073709551615
        ; 16 lines of prologue

.L4:
        vsqrtps ymm0, YMMWORD PTR [rax]
        add     rax, 32
        vmovups YMMWORD PTR [rax-32], ymm0
        cmp     rax, rdx
        jne     .L4
        mov     rdx, r9
        and     rdx, -8
        and     r9d, 7
        lea     rax, [rcx+rdx*4]
        je      .L21
        vzeroupper

        ; loops for <32 elements
```

I actually recommend using `-fno-math-errno`.

I actually recommend using `-fno-math-errno`.

But you have to ensure that the inputs are valid.

There is 1 more flag that I think is reasonable to enable by default

`-fno-trapping-math`

`-fno-trapping-math`

Assume no user-visible traps from floating point ops

`-fno-trapping-math`

Assume no user-visible traps from floating point ops

Allows reordering of some floating point computations

```
1 double arr[1024];
2
3 void foo(int n, double x, double y) {
4   for (int i = 0; i < n; ++i) {
5     if (arr[i] > 0.0)
6       arr[i] = x / y;
7   }
8 }
```

```
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;
  }
}
```

```
; GCC
foo(int, double, double):
        test    edi, edi
        jle     .L1
        movsx   rdi, edi
        xor     eax, eax
        pxor    xmm3, xmm3
.L5:
        movsd   xmm2, QWORD PTR arr[0+rax*8]
        comisd  xmm2, xmm3
        jbe     .L3
        movapd  xmm2, xmm0
        divsd   xmm2, xmm1
        movsd   QWORD PTR arr[0+rax*8], xmm2
.L3:
        add     rax, 1
        cmp     rax, rdi
        jne     .L5
.L1:
        ret
arr:
        .zero   8192
```

```
; GCC w/ -fno-trapping-math
foo(int, double, double):
        test    edi, edi
        jle     .L1
        divsd   xmm0, xmm1
        movsx   rdi, edi
        pxor    xmm2, xmm2
        xor     eax, eax
.L5:
        movsd   xmm1, QWORD PTR arr[0+rax*8]
        comisd  xmm1, xmm2
        jbe     .L3
        movsd   QWORD PTR arr[0+rax*8], xmm0
.L3:
        add     rax, 1
        cmp     rax, rdi
        jne     .L5
.L1:
        ret
arr:
        .zero   8192
```

```
; GCC
foo(int, double, double):
        test    edi, edi
        jle     .L1
        movsx   rdi, edi
        xor     eax, eax
        pxor    xmm3, xmm3
.L5:
        movsd   xmm2, QWORD PTR arr[0+rax*8]
        comisd  xmm2, xmm3
        jbe     .L3
        movapd  xmm2, xmm0
        divsd   xmm2, xmm1
        movsd   QWORD PTR arr[0+rax*8], xmm2
.L3:
        add     rax, 1
        cmp     rax, rdi
        jne     .L5
.L1:
        ret
arr:
        .zero   8192
```

```
; GCC w/ -fno-trapping-math
foo(int, double, double):
        test    edi, edi
        jle     .L1
        divsd   xmm0, xmm1
        movsx   rdi, edi
        pxor    xmm2, xmm2
        xor     eax, eax
.L5:
        movsd   xmm1, QWORD PTR arr[0+rax*8]
        comisd  xmm1, xmm2
        jbe     .L3
        movsd   QWORD PTR arr[0+rax*8], xmm0
.L3:
        add     rax, 1
        cmp     rax, rdi
        jne     .L5
.L1:
        ret
arr:
        .zero   8192
```

```
; GCC
foo(int, double, double):
        test    edi, edi
        jle     .L1
        movsx   rdi, edi
        xor     eax, eax
        pxor    xmm3, xmm3
.L5:
        movsd   xmm2, QWORD PTR arr[0+rax*8]
        comisd  xmm2, xmm3
        jbe     .L3
        movapd  xmm2, xmm0
        divsd   xmm2, xmm1
        movsd   QWORD PTR arr[0+rax*8], xmm2
.L3:
        add     rax, 1
        cmp     rax, rdi
        jne     .L5
.L1:
        ret
arr:
        .zero   8192
```

```
; GCC w/ -fno-trapping-math
foo(int, double, double):
        test    edi, edi
        jle     .L1
        divsd   xmm0, xmm1
        movsx   rdi, edi
        pxor    xmm2, xmm2
        xor     eax, eax
.L5:
        movsd   xmm1, QWORD PTR arr[0+rax*8]
        comisd  xmm1, xmm2
        jbe     .L3
        movsd   QWORD PTR arr[0+rax*8], xmm0
.L3:
        add     rax, 1
        cmp     rax, rdi
        jne     .L5
.L1:
        ret
arr:
        .zero   8192
```

Fun special case: C's floating point atomics are not allowed to expose FPE from compound assignment

```
float _Atomic a;

void foo(float x) {
  a += x;
}
```

```
; GCC
foo:
        sub     rsp, 72
        mov     eax, DWORD PTR a[rip]
        mov     DWORD PTR [rsp+28], eax
        fnstenv [rsp+32]
        fnclex
        stmxcsr DWORD PTR [rsp+12]
        mov     ecx, DWORD PTR [rsp+12]
        mov     edx, ecx
        and     edx, -64
        or      edx, 8064
        mov     DWORD PTR [rsp+12], edx
        ldmxcsr DWORD PTR [rsp+12]
.L4:
        movss   xmm1, DWORD PTR [rsp+28]
        mov     eax, DWORD PTR [rsp+28]
        addss   xmm1, xmm0
        movd    esi, xmm1
        lock cmpxchg    DWORD PTR a[rip], esi
        je      .L3
        mov     DWORD PTR [rsp+28], eax
        fnclex
        mov     DWORD PTR [rsp+12], edx
        ldmxcsr DWORD PTR [rsp+12]
        jmp     .L4
.L3:
        fnstsw  ax
        fldenv  [rsp+32]
        stmxcsr DWORD PTR [rsp+12]
        mov     edi, DWORD PTR [rsp+12]
```

```
; GCC
foo:
        sub     rsp, 72
        mov     eax, DWORD PTR a[rip]
        mov     DWORD PTR [rsp+28], eax
        fnstenv [rsp+32]
        fnclex
        stmxcsr DWORD PTR [rsp+12]
        mov     ecx, DWORD PTR [rsp+12]
        mov     edx, ecx
        and     edx, -64
        or      edx, 8064
        mov     DWORD PTR [rsp+12], edx
        ldmxcsr DWORD PTR [rsp+12]
.L4:
        movss   xmm1, DWORD PTR [rsp+28]
        mov     eax, DWORD PTR [rsp+28]
        addss   xmm1, xmm0
        movd    esi, xmm1
        lock cmpxchg    DWORD PTR a[rip], esi
        je      .L3
        mov     DWORD PTR [rsp+28], eax
        fnclex
        mov     DWORD PTR [rsp+12], edx
        ldmxcsr DWORD PTR [rsp+12]
        jmp     .L4
.L3:
        fnstsw  ax
        fldenv  [rsp+32]
        stmxcsr DWORD PTR [rsp+12]
        mov     edi, DWORD PTR [rsp+12]
```

```
; GCC w/ -fno-trapping-math
foo:
        mov     eax, DWORD PTR a[rip]
        movd    xmm1, eax
        mov     DWORD PTR [rsp-4], eax
.L4:
        addss   xmm1, xmm0
        mov     eax, DWORD PTR [rsp-4]
        movd    edx, xmm1
        lock cmpxchg    DWORD PTR a[rip], edx
        je      .L1
        mov     DWORD PTR [rsp-4], eax
        movss   xmm1, DWORD PTR [rsp-4]
        jmp     .L4
.L1:
        ret
a:
        .zero   4
```

That's all the flags that I think are useful and safe.

That's all the flags that I think are useful and safe.

Let's start going through the other ones.

`-fno-signed-zeros`

`-fno-signed-zeros`

Ignore the difference between positive and negative 0.

This flag is mostly safe, but not useful.

This flag is mostly safe, but not useful.

Or it is useful, but not safe.

In presence of signed zeros, it is not true that

$$x + 0 = x$$

That's basically all it does.

Now onto the more "fun and safe" flags.

`-ffinite-math-only`

`-ffinite-math-only`

Assume that floats cannot be NaN or ±Inf

With NaNs and Infs, these identities do not hold

$$x - x = 0$$

$$x = x$$

`finite-math-only` is usually enabled together with `no-signed-zeros`

`finite-math-only` is usually enabled together with `no-signed-zeros`

This removes "weird values" from consideration and enables vectorization in various cases

Example from my talk last month

```
float max(std::span<const float> input) {
    float ret = std::numeric_limits<float>::lowest();
    for (float f : input) {
        if (ret < f) {
            ret = f;
        }
    }
    return ret;
}
```

```
; GCC
max(std::span<float const, 18446744073709551615ul>):
        lea     rax, [rdi+rsi*4]
        vmovss  xmm0, DWORD PTR .LC0[rip]
        cmp     rdi, rax
        je      .L4
.L3:
        vmovss  xmm1, DWORD PTR [rdi]
        add     rdi, 4
        vmaxss  xmm0, xmm1, xmm0
        cmp     rdi, rax
        jne     .L3
        ret
.L4:
        ret
.LC0:
        .long   -8388609
```

```
; GCC
max(std::span<float const, 18446744073709551615ul>):
        lea     rax, [rdi+rsi*4]
        vmovss  xmm0, DWORD PTR .LC0[rip]
        cmp     rdi, rax
        je      .L4
.L3:
        vmovss  xmm1, DWORD PTR [rdi]
        add     rdi, 4
        vmaxss  xmm0, xmm1, xmm0
        cmp     rdi, rax
        jne     .L3
        ret
.L4:
        ret
.LC0:
        .long   -8388609
```

```
; GCC w/ -ffinite-math-only -fno-signed-zeros
max(std::span<float const, 18446744073709551
        ; prologue for different input sizes

.L4:
        vmaxps  ymm0, ymm0, YMMWORD PTR [rax]
        add     rax, 32
        cmp     rax, rdx
        jne     .L4
        vextractf128    xmm3, ymm0, 0x1
        vmaxps  xmm1, xmm3, xmm0
        vmovhlps        xmm2, xmm1, xmm1
        vmaxps  xmm2, xmm2, xmm1
        vshufps xmm1, xmm2, xmm2, 85
        vmaxps  xmm1, xmm1, xmm2
        test    sil, 7
        je      .L18
        and     rsi, -8
        vmaxps  xmm0, xmm0, xmm3
        lea     rax, [rdi+rsi*4]
        vzeroupper

        ; epilogue
```

What makes ffinite-math-only dangerous?

What makes ffinite-math-only dangerous?

You can no longer check for NaNs/Infs

What makes ffinite-math-only dangerous?

You can no longer check for NaNs/Infs

```
bool f1(float f) {
    return std::isnan(f);
}

bool f2(float f) {
    return std::isinf(f);
}
```

What makes ffinite-math-only dangerous?

You can no longer check for NaNs/Infs

```
bool f1(float f) {
    return std::isnan(f);
}

bool f2(float f) {
    return std::isinf(f);
}
```

```
; GCC w/ -ffinite-math-only
f1(float):
        xor     eax, eax
        ret

f2(float):
        xor     eax, eax
        ret
```

You might take neither true or false branch

You might take neither true or false branch

```
if (x > y) {
  foo();
} else {
  bar();
}
```

You might take neither true or false branch

```
if (x > y) {
  foo();
} else {
  bar();
}
```

```
if (x > y) {
  foo();
}
if (!(x > y)) {
  bar();
}
```

You might take neither true or false branch

```
if (x > y) {
  foo();
} else {
  bar();
}
```

```
if (x > y) {
  foo();
}
if (!(x > y)) {
  bar();
}
```

```
if (x > y) {
  foo();
}
if (x <= y) {
  bar();
}
```

Is this problem in practice?

Is this problem in practice?

```
float a[1024];
float b[1024];

void foo() {
  for (int i = 0; i < 1024; ++i) {
    if (b[i] > 42.0f) {
      a[i] = b[i] + 1.0f;
    } else {
      b[i] = a[i] + 1.0f;
    }
  }
}
```

```
; Clang
foo():
        xor     eax, eax
        lea     rcx, [rip + b]
        vbroadcastss    ymm0, dword ptr [rip + .LCPI0_0]
        lea     rdx, [rip + a]
        vbroadcastss    ymm1, dword ptr [rip + .LCPI0_1]
.LBB0_1:
        vmovups ymm2, ymmword ptr [rax + rcx]
        vcmpnltps       ymm3, ymm0, ymm2
        vaddps  ymm4, ymm1, ymmword ptr [rax + rdx]
        vmaskmovps      ymmword ptr [rax + rcx], ymm3, ymm4
        vcmpltps        ymm3, ymm0, ymm2
        vaddps  ymm2, ymm2, ymm1
        vmaskmovps      ymmword ptr [rax + rdx], ymm3, ymm2
        add     rax, 32
        cmp     rax, 4096
        jne     .LBB0_1
        vzeroupper
        ret
```

```
; Clang w/ -ffinite-math-only
foo():
        xor     eax, eax
        lea     rcx, [rip + b]
        vbroadcastss    ymm0, dword pt
        lea     rdx, [rip + a]
        vbroadcastss    ymm1, dword pt
.LBB0_1:
        vmovups ymm2, ymmword ptr [rax
        vcmpleps        ymm3, ymm2, ym
        vaddps  ymm4, ymm1, ymmword pt
        vmaskmovps      ymmword ptr [r
        vcmpltps        ymm3, ymm0, ym
        vaddps  ymm2, ymm2, ymm1
        vmaskmovps      ymmword ptr [r
        add     rax, 32
        cmp     rax, 4096
        jne     .LBB0_1
        vzeroupper
        ret
```

Let's talk about something even more fun

`-fassociative-math`

`-fassociative-math`

Assume that floating point operations are associative

Without associativity, these are not valid

$$X + Y - X = Y$$

$$X * Y + Y * Z = (X + Z) * Y$$

$$c * X + X = (c + 1) * X$$

$$\frac{c_1}{X} * c_2 = \frac{c_1 * c_2}{X}$$

Associativity enables lot of vectorization opportunities

```
#include <span>

float foo(std::span<const float> arr) {
  float sum = 0.0f;
  for (float f : arr) {
    sum += f;
  }
  return sum;
}
```

```
; GCC
foo(std::span<float const, 18446744073709551615ul>):
        ; 12 lines of prologue
.L4:
        vaddss  xmm0, xmm0, DWORD PTR [rax]
        add     rax, 32
        vaddss  xmm0, xmm0, DWORD PTR [rax-28]
        vaddss  xmm0, xmm0, DWORD PTR [rax-24]
        vaddss  xmm0, xmm0, DWORD PTR [rax-20]
        vaddss  xmm0, xmm0, DWORD PTR [rax-16]
        vaddss  xmm0, xmm0, DWORD PTR [rax-12]
        vaddss  xmm0, xmm0, DWORD PTR [rax-8]
        vaddss  xmm0, xmm0, DWORD PTR [rax-4]
        cmp     rax, rdx
        jne     .L4
        test    sil, 7
        je      .L1
        mov     rax, rsi
        and     rax, -8

        ; 30 lines of epilogue
```

```
; GCC
foo(std::span<float const, 18446744073709551615ul>):
        ; 12 lines of prologue
.L4:
        vaddss  xmm0, xmm0, DWORD PTR [rax]
        add     rax, 32
        vaddss  xmm0, xmm0, DWORD PTR [rax-28]
        vaddss  xmm0, xmm0, DWORD PTR [rax-24]
        vaddss  xmm0, xmm0, DWORD PTR [rax-20]
        vaddss  xmm0, xmm0, DWORD PTR [rax-16]
        vaddss  xmm0, xmm0, DWORD PTR [rax-12]
        vaddss  xmm0, xmm0, DWORD PTR [rax-8]
        vaddss  xmm0, xmm0, DWORD PTR [rax-4]
        cmp     rax, rdx
        jne     .L4
        test    sil, 7
        je      .L1
        mov     rax, rsi
        and     rax, -8

        ; 30 lines of epilogue
```

```
; GCC w/ -fassociative-math -fno-trapping-mat
foo(std::span<float const, 1844674407370955
        ; 12 lines of prologue
.L4:
        vaddps  ymm0, ymm0, YMMWORD PTR [rax]
        add     rax, 32
        cmp     rdx, rax
        jne     .L4
        vextractf128    xmm1, ymm0, 0x1
        vaddps  xmm1, xmm1, xmm0
        vmovhlps        xmm2, xmm1, xmm1
        vaddps  xmm2, xmm2, xmm1
        vshufps xmm0, xmm2, xmm2, 85
        vaddps  xmm0, xmm0, xmm2
        test    sil, 7
        je      .L18
        mov     rax, rsi
        and     rax, -8

        ; 30 lines of epilogue
```

Why is `fassociative-math` unsafe?

Why is `fassociative-math` unsafe?

Let's talk about Kahan Summation

Kahan Summation is an algorithm for summing up sequence of floats with reduced numerical error.

Kahan Summation is an algorithm for summing up sequence of floats with reduced numerical error.

It relies on precise ordering of operations

```
float kahan_sum(std::span<const float> input) {
    float sum = 0.f;
    float compensation = 0.f;

    for (float f : input) {
        auto const y = f - compensation;
        auto const t = sum + y;
        compensation = (t - sum) - y;
        sum = t;
    }

    return sum;
}
```

```
float kahan_sum(std::span<const float> input) {
    float sum = 0.f;
    float compensation = 0.f;

    for (float f : input) {
        auto const y = f - compensation;
        auto const t = sum + y;
        compensation = (t - sum) - y;
        sum = t;
    }

    return sum;
}
```

Mathematically, `compensation` is always 0

$$comp = (t - sum) - y$$
$$t = sum + y$$

$$comp = (sum + y - sum) - y$$
$$comp = (sum - sum) + (y - y)$$
$$comp = 0 + 0$$

Yes, the compiler will reason this through

Yes, the compiler will reason this through

```
; GCC
;
sum(std::span<float const, 18446744073709551615ul>):
        ; 15 lines of prologue
.L4:
        movups  xmm2, XMMWORD PTR [rax]
        add     rax, 16
        addps   xmm0, xmm2
        cmp     rdx, rax
        jne     .L4
        movaps  xmm1, xmm0
        mov     rax, rcx
        movhlps xmm1, xmm0
        and     rax, -4
        addps   xmm1, xmm0
        lea     rdx, [rdi+rax*4]
        movaps  xmm0, xmm1
        shufps  xmm0, xmm1, 85
        addps   xmm0, xmm1
        cmp     rcx, rax
        je      .L11
        ; 20 lines of epilogue
```

```
; GCC w/ -fassociative-math -fno-trapping-math
;        -fno-signed-zeros -ffinite-math-only
sum_kahan(std::span<float const, 184467440737095516
        ; 15 lines of prologue
.L4:
        movups  xmm2, XMMWORD PTR [rax]
        add     rax, 16
        addps   xmm0, xmm2
        cmp     rdx, rax
        jne     .L4
        movaps  xmm1, xmm0
        mov     rax, rcx
        movhlps xmm1, xmm0
        and     rax, -4
        addps   xmm1, xmm0
        lea     rdx, [rdi+rax*4]
        movaps  xmm0, xmm1
        shufps  xmm0, xmm1, 85
        addps   xmm0, xmm1
        cmp     rcx, rax
        je      .L11
        ; 20 lines of epilogue
```

Let's skip over other optimization flags from ffast-math

Let's skip over other optimization flags from ffast-math

And talk about different interesting flag

`-ffp-contract=style`

`-ffp-contract=style`

Setting of *floating point expressions contractions*

`-ffp-contract=style`

Setting of *floating point expressions contractions*

e.g. Fused Multiply Add for `a * b + c`

C++ allows FP contractions **in single expression**.

C++ allows FP contractions **in single expression**.

```
float fma_yes(float x, float y, float z) {
  return x*y + z;
}

float fma_no(float x, float y, float z) {
    float temp = x * y;
    return temp + z;
}
```

FMA can give different result because it skips rounding after multiply.

-ffp-contract=<value>

-ffp-contract=<value>

- on - follow language standard

-ffp-contract=<value>

- on - follow language standard
- off - disables contractions

-ffp-contract=<value>

- on - follow language standard
- off - disables contractions
- fast - go fast (e.g. contract across statements)

As of 2025 both GCC and Clang obey `ffp-contract`

As of 2025 both GCC and Clang obey `ffp-contract`

BUT

As of 2025 both GCC and Clang obey `ffp-contract`

BUT

they disagree on the default value

```
float fma_yes(float x, float y, float z) {
  return x*y + z;
}

float fma_no(float x, float y, float z) {
    float temp = x * y;
    return temp + z;
}
```

```c
float fma_yes(float x, float y, float z) {
  return x*y + z;
}

float fma_no(float x, float y, float z) {
    float temp = x * y;
    return temp + z;
}
```

```asm
; GCC
fma_yes(float, float, float):
        vfmadd132ss     xmm0, xmm2, xmm1
        ret
fma_no(float, float, float):
        vfmadd132ss     xmm0, xmm2, xmm1
        ret
```

```asm
; Clang
fma_yes(float, float, float):
        vfmadd213ss     xmm0, xmm1, xmm2
        ret

fma_no(float, float, float):
        vmulss  xmm0, xmm0, xmm1
        vaddss  xmm0, xmm0, xmm2
        ret
```

CONCLUSION

Don't use `Ofast` or `ffast-math`

Don't use `Ofast` or `ffast-math`

The will improve your performance, but also break the IEEE rules in many different ways

If you need the performance improvements from breaking IEEE, use the individual flags.

If you need the performance improvements from breaking IEEE, use the individual flags.

`-fno-math-errno` is almost always safe.

If you need the performance improvements from breaking IEEE, use the individual flags.

`-fno-math-errno` is almost always safe.

So is `-fno-trapping-math`

Other flags can carry more surprises, and you should not let them spread across multiple translation units.

Other flags can carry more surprises, and you should not let them spread across multiple translation units.

Or even functions.

Beware compiler defaults

THE END

Questions?