THE COMPILER IS SMARTER THAN YOU:

That's why it cannot optimize your code

Martin Hořeňovský

PEX

THE COMPILER IS SMARTER THAN YOU:

That's why it cannot optimize your code

Martin Hořeňovský

PEX

?!?

We will talk about the compiler missing "obvious" optimization opportunities …

We will talk about the compiler missing "obvious" optimization opportunities …

… and why they are not obvious

We will talk about the compiler missing "obvious" optimization opportunities …

… and why they are not obvious (nor allowed)

We will also talk about getting the desired optimizations to happen

THE AS-IF RULE

Executing a **well-formed** program

Executing a **well-formed** program must produce the same **observable** behaviour

Executing a **well-formed** program must produce the same **observable** behaviour, **AS IF** it was executed on the C++ abstract machine.

There are three exceptions to the as-if rule:

There are three exceptions to the as-if rule:

- copy elision - includes moves and destructors

There are three exceptions to the as-if rule:

- copy elision - includes moves and destructors
- new expression elision - even for user-provided new

There are three exceptions to the as-if rule:

- copy elision - includes moves and destructors
- new expression elision - even for user-provided new
- intermediate floating point point exceptions

There are three exceptions to the as-if rule:

- copy elision - includes moves and destructors
- new expression elision - even for user-provided new
- intermediate floating point point exceptions

None of these will be relevant to the talk

Keep the as-if rule in mind during the talk

TEST CONFIGURATION

All examples in this talk will be compiled with `-O3` and `-march=broadwell`

All examples in this talk will be compiled with `-O3` and `-march=broadwell`

We will be using GCC 14 and Clang 20 for testing

SQRTF IN A LOOP

Consider this simple C++ function:

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

Consider this simple C++ function:

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

What do we want to see in the ASM?

Consider this simple C++ function:

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

What do we want to see in the ASM?

`vsqrtps` to process floats 8 at a time.

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

What do we actually get?

```
; 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
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615
        ; 9 lines of prologue

        vxorps  xmm1, xmm1, xmm1
        jmp     .LBB0_2
.LBB0_3:
        vsqrtss xmm0, xmm0, xmm0
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        je      .LBB0_6
.LBB0_2:
        vmovss  xmm0, dword ptr [r14 + r15]
        vucomiss        xmm0, xmm1
        jae     .LBB0_3
        call    sqrtf@PLT
        vxorps  xmm1, xmm1, xmm1
        vmovss  dword ptr [r14 + r15], xmm0
        add     r15, 4
        cmp     rbx, r15
        jne     .LBB0_2

        ; 4 lines of epilogue
```

The compiler didn't vectorize the code becase it would not be equivalent if any input is not valid for `sqrtf`.

The compiler didn't vectorize the code becase it would not be equivalent if any input is not valid for `sqrtf`.

What if we know that the inputs are valid?

C++23 has `[[assume(expr)]]`

C++23 has [[assume(expr)]]

```
1 void do_sqrtf(std::span<float> data) {
2     for (float& f : data) {
3         [[assume(f >= 0)]];
4         f = sqrtf(f);
5     }
6 }
```

What do we get now?

What do we get now?

```
; 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
```

What do we get now?

```
; 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
```

```
1  ; clang
2  do_sqrtf(std::span<float, 18446744073709551615ul>
3          ; 18 lines to dispatch to the correct wid
4  .LBB0_11:
5          vsqrtps ymm0, ymmword ptr [rdi + 4*r8]
6          vsqrtps ymm1, ymmword ptr [rdi + 4*r8 + 3
7          vsqrtps ymm2, ymmword ptr [rdi + 4*r8 + 6
8          vsqrtps ymm3, ymmword ptr [rdi + 4*r8 + 9
9          vmovups ymmword ptr [rdi + 4*r8], ymm0
10         vmovups ymmword ptr [rdi + 4*r8 + 32], ym
11         vmovups ymmword ptr [rdi + 4*r8 + 64], ym
12         vmovups ymmword ptr [rdi + 4*r8 + 96], ym
13         add     r8, 32
14         cmp     rdx, r8
15         jne     .LBB0_11
16         cmp     rax, rdx
17         je      .LBB0_9
18         test    al, 28
19         je      .LBB0_14
20 .LBB0_6:
21         add     rcx, 28
22         and     rcx, rax
23         lea     r8, [rdi + 4*rcx]
24
25         ; loops for <32 elements
```

GCC completely ignores `[[assume]]`

GCC completely ignores `[[assume]]`

But it understands `__builtin_unreachable()`:

GCC completely ignores `[[assume]]`

But it understands `__builtin_unreachable()`:

```
1 void do_sqrtf(std::span<float> data) {
2     for (float& f : data) {
3         if (f < 0) __builtin_unreachable();
4         f = sqrtf(f);
5     }
6 }
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        lea     rax, [rdi+rsi*4]
        cmp     rdi, rax
        je      .L5
        vxorps  xmm1, xmm1, xmm1
.L3:
        vsqrtss xmm0, xmm1, DWORD PTR [rdi]
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rax, rdi
        jne     .L3
.L5:
        ret
```

```
; GCC
do_sqrtf(std::span<float, 18446744073709551615ul>):
        lea     rax, [rdi+rsi*4]
        cmp     rdi, rax
        je      .L5
        vxorps  xmm1, xmm1, xmm1
.L3:
        vsqrtss xmm0, xmm1, DWORD PTR [rdi]
        add     rdi, 4
        vmovss  DWORD PTR [rdi-4], xmm0
        cmp     rax, rdi
        jne     .L3
.L5:
        ret
```

```
; clang
do_sqrtf(std::span<float, 18446744073709551615ul>
        ; 18 lines to dispatch to the correct wid
.LBB0_11:
        vsqrtps ymm0, ymmword ptr [rdi + 4*r8]
        vsqrtps ymm1, ymmword ptr [rdi + 4*r8 + 3
        vsqrtps ymm2, ymmword ptr [rdi + 4*r8 + 6
        vsqrtps ymm3, ymmword ptr [rdi + 4*r8 + 9
        vmovups ymmword ptr [rdi + 4*r8], ymm0
        vmovups ymmword ptr [rdi + 4*r8 + 32], ym
        vmovups ymmword ptr [rdi + 4*r8 + 64], ym
        vmovups ymmword ptr [rdi + 4*r8 + 96], ym
        add     r8, 32
        cmp     rdx, r8
        jne     .LBB0_11
        cmp     rax, rdx
        je      .LBB0_9
        test    al, 28
        je      .LBB0_14
.LBB0_6:
        add     rcx, 28
        and     rcx, rax
        lea     r8, [rdi + 4*rcx]

        ; loops for <32 elements
```

What can we do to get GCC to vectorize the code?

What can we do to get GCC to vectorize the code?

Update to GCC 15, or

What can we do to get GCC to vectorize the code?

Update to GCC 15, or

`-ffast-math`

What can we do to get GCC to vectorize the code?

Update to GCC 15, or

~~`-ffast-math`~~ `-fno-math-errno`

```
do_sqrtf(std::span<float, 18446744073709551615ul>):
        ; 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
```

MAX ELEMENT FROM SPAN

Let's look at another simple C++ function:

```
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;
}
```

Let's look at another simple C++ function:

```
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;
}
```

What do we want to see in the ASM here?

Let's look at another simple C++ function:

```
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;
}
```

What do we want to see in the ASM here?

`vmaxps` to process floats 8 at a time.

What do we actually get?

What do we actually get?

```
1  ; GCC
2  max(std::span<float const, 18446744073709551615ul>):
3          lea     rax, [rdi+rsi*4]
4          vmovss  xmm0, DWORD PTR .LC0[rip]
5          cmp     rdi, rax
6          je      .L4
7  .L3:
8          vmovss  xmm1, DWORD PTR [rdi]
9          add     rdi, 4
10         vmaxss  xmm0, xmm1, xmm0
11         cmp     rdi, rax
12         jne     .L3
13         ret
14 .L4:
15         ret
16 .LC0:
17         .long   -8388609
```

What do we actually get?

```
; 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
```

```
; clang
.LCPI0_0:
        .long   0xff7fffff
max(std::span<float const, 1844674407370955
        test    rsi, rsi
        je      .LBB0_1
        shl     rsi, 2
        vmovss  xmm0, dword ptr [rip + .LCPI0
        xor     eax, eax
.LBB0_4:
        vmovss  xmm1, dword ptr [rdi + rax]
        vmaxss  xmm0, xmm1, xmm0
        add     rax, 4
        cmp     rsi, rax
        jne     .LBB0_4
        ret
.LBB0_1:
        vmovss  xmm0, dword ptr [rip + .LCPI0
        ret
```

`vmaxps` has different semantics from

```
if (ret < f) { ret = f; }
```

`vmaxps` has different semantics from

```
if (ret < f) { ret = f; }
```

- Different behaviour for NaNs

`vmaxps` has different semantics from

```
if (ret < f) { ret = f; }
```

- Different behaviour for NaNs
- Different behaviour for +/- 0

Can we get the compiler to vectorize the loop anyway?

- We can disregard NaNs: `-ffinite-math-only`
- We can disregard neg zero: `-fno-signed-zeros`

What do we get now?

What do we get now?

```
; GCC
max(std::span<float const, 18446744073709551615ul>):
        ; 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 do we get now?

```
; GCC
max(std::span<float const, 18446744073709551615ul>):
        ; 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
```

```
; clang
.LCPI0_0:
        .long   0xff7fffff
max(std::span<float const, 1844674407370955
        ; prologue for different input sizes

.LBB0_12:
        mov     rdx, rax
        and     rdx, rcx
        vbroadcastss    ymm0, dword ptr [rip
        xor     r8d, r8d
        vmovaps ymm1, ymm0
        vmovaps ymm2, ymm0
        vmovaps ymm3, ymm0
.LBB0_13:
        vmaxps  ymm0, ymm0, ymmword ptr [rdi
        vmaxps  ymm1, ymm1, ymmword ptr [rdi
        vmaxps  ymm2, ymm2, ymmword ptr [rdi
        vmaxps  ymm3, ymm3, ymmword ptr [rdi
        add     r8, 32
        cmp     rdx, r8
        jne     .LBB0_13

        ; epilogue
```

Can we do this without changing global floating point behaviour?

Can we do this without changing global floating point behaviour?

Sadly, no.

But we can get rid of `-fno-signed-zeros` (for GCC):

But we can get rid of `-fno-signed-zeros` (for GCC):

```
1 float max(std::span<const float> input) {
2     float ret = std::numeric_limits<float>::lowest();
3     for (float f : input) {
4         ret = std::fmax(ret, f);
5     }
6     return ret;
7 }
```

Floats are hard …

Floats are hard …

… let's talk about integers instead.

COMBINING INTEGER COMPARISONS

Consider this C(++) function:

```
struct pair {
  int16_t a, b;
};

bool eq(pair* lhs, pair* rhs) {
    return lhs->a == rhs->a && lhs->b == rhs->b;
}
```

Consider this C(++) function:

```
struct pair {
  int16_t a, b;
};

bool eq(pair* lhs, pair* rhs) {
    return lhs->a == rhs->a && lhs->b == rhs->b;
}
```

What do we want to see in the ASM here?

Consider this C(++) function:

```
struct pair {
  int16_t a, b;
};

bool eq(pair* lhs, pair* rhs) {
    return lhs->a == rhs->a && lhs->b == rhs->b;
}
```

What do we want to see in the ASM here?

4 byte `mov` from memory followed by 4 bytes `cmp`.

What do we actually get?

What do we actually get?

```
1  ; GCC
2  eq(pair*, pair*):
3          movzx   edx, WORD PTR [rsi]
4          xor     eax, eax
5          cmp     WORD PTR [rdi], dx
6          je      .L5
7          ret
8  .L5:
9          movzx   eax, WORD PTR [rsi+2]
10         cmp     WORD PTR [rdi+2], ax
11         sete    al
12         ret
```

What do we actually get?

```
1  ; GCC
2  eq(pair*, pair*):
3          movzx   edx, WORD PTR [rsi]
4          xor     eax, eax
5          cmp     WORD PTR [rdi], dx
6          je      .L5
7          ret
8  .L5:
9          movzx   eax, WORD PTR [rsi+2]
10         cmp     WORD PTR [rdi+2], ax
11         sete    al
12         ret
```

What do we actually get?

```
1  ; GCC
2  eq(pair*, pair*):
3          movzx   edx, WORD PTR [rsi]
4          xor     eax, eax
5          cmp     WORD PTR [rdi], dx
6          je      .L5
7          ret
8  .L5:
9          movzx   eax, WORD PTR [rsi+2]
10         cmp     WORD PTR [rdi+2], ax
11         sete    al
12         ret
```

```
1  ; clang
2  eq(pair*, pair*):
3          movzx   eax, word ptr [rdi]
4          cmp     ax, word ptr [rsi]
5          jne     .LBB0_1
6          movzx   eax, word ptr [rdi + 2]
7          cmp     ax, word ptr [rsi + 2]
8          sete    al
9          ret
10 .LBB0_1:
11         xor     eax, eax
12         ret
```

What do we actually get?

```
1  ; GCC
2  eq(pair*, pair*):
3          movzx   edx, WORD PTR [rsi]
4          xor     eax, eax
5          cmp     WORD PTR [rdi], dx
6          je      .L5
7          ret
8  .L5:
9          movzx   eax, WORD PTR [rsi+2]
10         cmp     WORD PTR [rdi+2], ax
11         sete    al
12         ret
```

```
1  ; clang
2  eq(pair*, pair*):
3          movzx   eax, word ptr [rdi]
4          cmp     ax, word ptr [rsi]
5          jne     .LBB0_1
6          movzx   eax, word ptr [rdi + 2]
7          cmp     ax, word ptr [rsi + 2]
8          sete    al
9          ret
10 .LBB0_1:
11         xor     eax, eax
12         ret
```

The compiler cannot issue wide load in general case:

The compiler cannot issue wide load in general case:

```
1  struct pair {
2    int16_t a, b;
3  };
4
5  bool eq(pair* lhs, pair* rhs) {
6      return lhs->a == rhs->a && lhs->b == rhs->b;
7  }
8
9  bool f() {
10     pair p1, p2;
11     p1.a = 1;
12     p2.a =2;
13     return eq(&p1, &p2);
14 }
```

The compiler cannot issue wide load in general case:

```
 1 struct pair {
 2   int16_t a, b;
 3 };
 4
 5 bool eq(pair* lhs, pair* rhs) {
 6     return lhs->a == rhs->a && lhs->b == rhs->b;
 7 }
 8
 9 bool f() {
10     pair p1, p2;
11     p1.a = 1;
12     p2.a =2;
13     return eq(&p1, &p2);
14 }
```

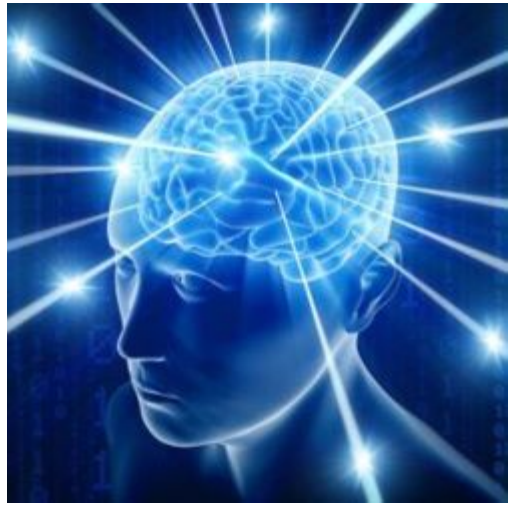
What if we want the compiler to use wide loads anyway?

The **AS-IF** rule governs the transformations of a **well-formed** program.

If it is ill-formed for `pair::b` to be unreadable, then the compiler is allowed to always load `pair::b` and optimize accordingly.

If we invoke the undefined behaviour ourselves, the compiler does not have to avoid it.

If we invoke the undefined behaviour ourselves, the compiler does not have to avoid it.



```
1 struct pair {
2   int16_t a, b;
3 };
4
5 bool eq(pair lhs, pair rhs) { <--- pass by value
6     return lhs.a == rhs.a && lhs.b == rhs.b;
7 }
```

```
1 struct pair {
2   int16_t a, b;
3 };
4
5 bool eq(pair lhs, pair rhs) { <--- pass by value
6     return lhs.a == rhs.a && lhs.b == rhs.b;
7 }
```

Calling `eq` with partially unreadable `pair` is now UB.

What do the compilers think about it now?

What do the compilers think about it now?

```
1  ; gcc
2  eq(pair, pair):
3          xor     eax, eax
4          cmp     di, si
5          je      .L5
6          ret
7  .L5:
8          sar     edi, 16
9          sar     esi, 16
10         cmp     di, si
11         sete    al
12         ret
```

What do the compilers think about it now?

```
; gcc
eq(pair, pair):
        xor     eax, eax
        cmp     di, si
        je      .L5
        ret
.L5:
        sar     edi, 16
        sar     esi, 16
        cmp     di, si
        sete    al
        ret
```

```
; clang
eq(pair, pair):
        cmp     edi, esi
        sete    al
        ret
```

You can also make reading b unconditional, under the same logic.

You can also make reading b unconditional, under the same logic.

```
struct pair {
  int16_t a, b;
};

bool eq(pair* lhs, pair* rhs) {
    return (lhs->a == rhs->a) & (lhs->b == rhs->b);
}
```

```
1  ; gcc
2  eq(pair*, pair*):
3          movzx   eax, WORD PTR [rsi]
4          cmp     WORD PTR [rdi], ax
5          movzx   ecx, WORD PTR [rsi+2]
6          sete    al
7          cmp     WORD PTR [rdi+2], cx
8          sete    dl
9          and     eax, edx
10         ret
```

```
; gcc
eq(pair*, pair*):
        movzx   eax, WORD PTR [rsi]
        cmp     WORD PTR [rdi], ax
        movzx   ecx, WORD PTR [rsi+2]
        sete    al
        cmp     WORD PTR [rdi+2], cx
        sete    dl
        and     eax, edx
        ret
```

```
; clang
eq(pair*, pair*):
        mov     eax, dword ptr [rsi]
        cmp     dword ptr [rdi], eax
        sete    al
        ret
```

To get GCC to optimize this, we have to be less subtle

To get GCC to optimize this, we have to be less subtle

```
struct pair {
  int16_t a, b;
};

bool eq(pair* lhs, pair* rhs) {
    return std::memcmp(lhs, rhs, sizeof(*lhs)) == 0;
}
```

To get GCC to optimize this, we have to be less subtle

```
struct pair {
  int16_t a, b;
};

bool eq(pair* lhs, pair* rhs) {
    return std::memcmp(lhs, rhs, sizeof(*lhs)) == 0;
}
```

This is error prone, but works

```
; gcc
eq(pair*, pair*):
        mov     eax, DWORD PTR [rsi]
        cmp     DWORD PTR [rdi], eax
        sete    al
        ret
```

```
; clang
eq(pair*, pair*):
        mov     eax, dword ptr [rdi]
        cmp     eax, dword ptr [rsi]
        sete    al
        ret
```

The **AS-IF** rule talks about original code

The **AS-IF** rule talks about original code

This means that the compiler can use the code before inlining to optimize the code after inlining.

```
1  struct pair {
2    int16_t a, b;
3  };
4
5  bool eq(pair lhs, pair rhs) {
6      return lhs.a == rhs.a && lhs.b == rhs.b;
7  }
8
9  bool foo(pair* lhs, pair* rhs) {
10     return eq(*lhs, *rhs);
11 }
```

```
1  ; clang
2  eq(pair, pair):
3          cmp     edi, esi
4          sete    al
5          ret
6
7  foo(pair*, pair*):
8          mov     eax, dword ptr [rdi]
9          cmp     eax, dword ptr [rsi]
10         sete    al
11         ret
```

But what about C++?

But what about C++?

Let's let the compiler implement the comparison

But what about C++?

Let's let the compiler implement the comparison

```
struct pair {
  int16_t a, b;
  friend bool operator==(pair, pair) = default;
};

bool foo(pair* lhs, pair* rhs) {
    return *lhs == *rhs;
}
```

```
; GCC
foo(pair*, pair*):
        movzx   eax, WORD PTR [rsi+2]
        cmp     WORD PTR [rdi+2], ax
        movzx   ecx, WORD PTR [rsi]
        sete    al
        cmp     WORD PTR [rdi], cx
        sete    dl
        and     eax, edx
        ret
```

```
; clang
foo(pair*, pair*):
        mov     eax, dword ptr [rdi]
        cmp     eax, dword ptr [rsi]
        sete    al
        ret
```

CONCLUSION

Compilers are sticklers for language rules minutiae

Compilers are sticklers for language rules minutiae

Missing optimizations are not always compiler defects

Compilers are sticklers for language rules minutiae

Missing optimizations are not always compiler defects

They might just not be valid without extra information

You can learn a lot from investigating missed optimizations

If in doubt, compare output from different compilers

If in doubt, compare output from different compilers

If they agree, there is a good chance that you are missing something

THE END

Questions?