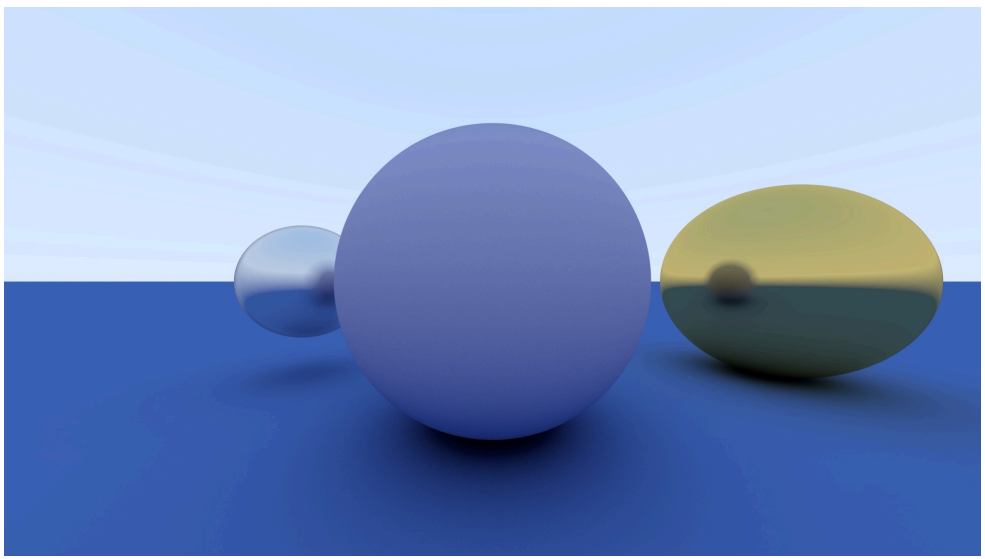
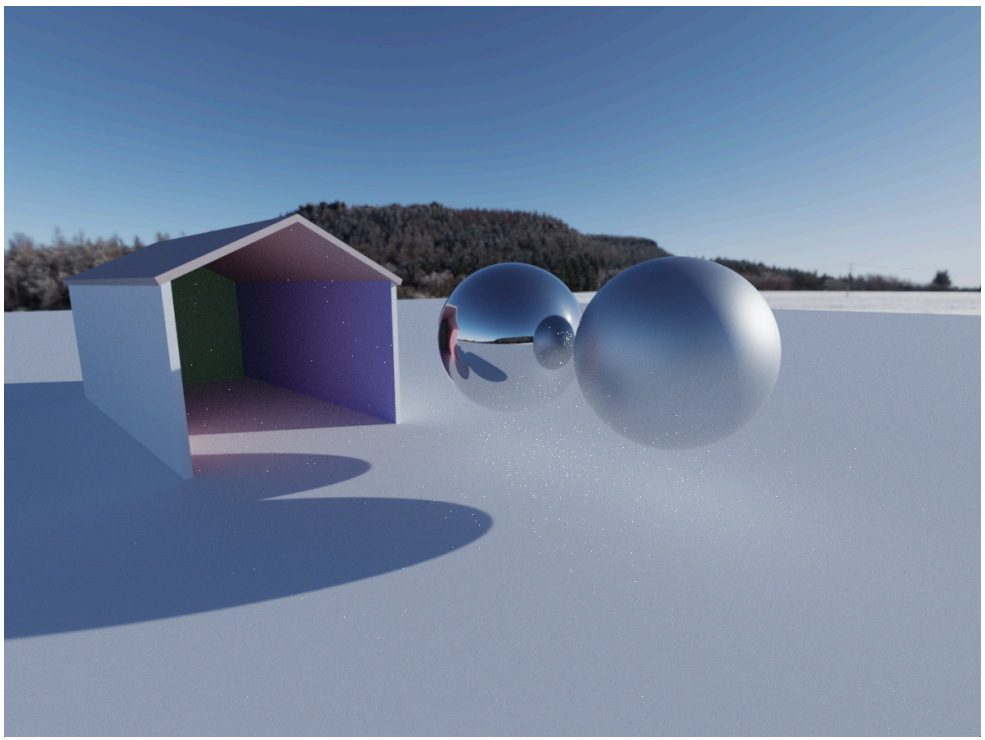
Välkomna tillbaka!





SIMD

Single Instruction Multiple Data

SIMD vs SISD

SISD - Single Instruction *Single* Data

- `ADDQ`, `SUBQ`, ...
- Enskilda värden, ett per register (max. 64 bitar).

$$x_1 + y_1 = z_1$$
$$x_2 + y_2 = z_2$$
$$x_3 + y_3 = z_3$$
$$x_4 + y_4 = z_4$$

SIMD - Single Instruction *Multiple* Data

- Instruktioner opererar på flera värden samtidigt
- Register med minst 128 bitar

$$x_1 \; x_2 \; x_3 \; x_4 + y_1 \; y_2 \; y_3 \; y_4 = z_1 \; z_2 \; z_3 \; z_4$$

Register i x86-64

- `rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsp`, `rsi`, `rdi`, `r8-15` (64-bit)
- SSE: `xmm0-15` (128-bit), 1999
- AVX: `ymm0-15` (256-bit), 2011
- AVX-512: `zmm0-31` (512-bit), 2016

SSE - Streaming SIMD Extensions

Exempel på instruktioner (addition, i C):

- 4 + 4 32-bitar flyttal: `_mm_add_ps(...)`
- 2 + 2 64-bitar flyttal: `_mm_add_pd(...)`
- 4 + 4 32-bitar heltal: `_mm_add_epi32(...)` eller `_mm_add_epu32(...)`
- 16 + 16 8-bitar heltal: `_mm_add_epi8(...)` eller `_mm_add_epu8(...)`
- Alla funktioner finns dokumenterade här.

Loads och stores måste ske på minnesadresser som är en multipel av datastorleken.

Vad händer annars?

```
.global main
main:
  movaps (%rsp), %xmm0
  ret
```

```
$ gcc simd.S && ./a.out
Segmentation fault         (core dumped) ./a.out
```

Alignment

Eller ännu värre!

```
struct Vec4 {
    /* Ingen alignment! */
    float x, y, z, w;
};

bool cmp_simd(Vec4 a, Vec4 b) {
    __m128 xmm_a = _mm_load_ps(&a.x);
    __m128 xmm_b = _mm_load_ps(&b.x);
    __m128 eq = _mm_cmpeq_ps(xmm_a, xmm_b);
    return _mm_movemask_ps(eq) == 0     ;
}
```

```
cmp_simd(Vec4, Vec4):
    movaps  %xmm4, -24(%rsp)
    movq    -16(%rsp), %rdx
    movq    %xmm0, -24(%rsp)
    movq    %rdx, -16(%rsp)
    movdqa  -24(%rsp), %xmm4
    movaps  %xmm5, -24(%rsp)
    movq    -16(%rsp), %rdx
    movq    %xmm2, -24(%rsp)
    punpcklqdq      %xmm1, %xmm4
    movq    %rdx, -16(%rsp)
    movdqa  -24(%rsp), %xmm5
    movdqa  %xmm4, %xmm1
    punpcklqdq      %xmm3, %xmm5
    cmpeqps %xmm5, %xmm1
    movmskps        %xmm1, %eax
    cmpl    $15, %eax
    sete    %al
    ret
```

SSE - Exempel i C


```c
#include <stdio.h>
#include <xmmintrin.h>

int main(int argc, char **argv)
{
    _Alignas(__m128i) int xs[4] = { 1, 2, 3, 4 };
    _Alignas(__m128i) int ys[4] = { 2, 4, 6, 8 };
    _Alignas(__m128i) int zs[4];

    __m128i x = _mm_load_si128((__m128i*) xs);
    __m128i y = _mm_load_si128((__m128i*) ys);
    __m128i z = _mm_add_epi32(x, y);

    _mm_store_si128((__m128i*) zs, z);
    for (int i = 0; i < 4; i++)
            printf("%d\n", zs[i]);
}
```

```
$ ./add
3
6
9
12
```

SSE - Shuffle


```
__m128i _mm_shuffle_epi32 (__m128i a, int imm8)
```

```
DEFINE SELECT4(src, control) {
  CASE(control[1:0]) OF
  0:  tmp[31:0] := src[31:0]
  1:  tmp[31:0] := src[63:32]
  2:  tmp[31:0] := src[95:64]
  3:  tmp[31:0] := src[127:96]
  ESAC
  RETURN tmp[31:0]
}
dst[31:0] := SELECT4(a[127:0], imm8[1:0])
dst[63:32] := SELECT4(a[127:0], imm8[3:2])
dst[95:64] := SELECT4(a[127:0], imm8[5:4])
dst[127:96] := SELECT4(a[127:0], imm8[7:6])
```

SSE - Shuffle


```c
#include <stdio.h>
#include <xmmintrin.h>

int main(int argc, char **argv)
{
    __m128i x = _mm_set_epi32(1, 2, 3, 4);
    x = _mm_shuffle_epi32(x, _MM_SHUFFLE(3, 2, 1, 0));

    _Alignas(__m128i) int xs[4];
    _mm_store_si128((__m128i*) xs, x);

    for (int i = 0; i < 4; i++)
        printf("%d\n", xs[i]);
}
```

```
$ ./shuffle
4
3
2
1
```

Exempel: Skalärprodukt

SIMD lämpar sig väl för vektoroperationer, t.ex. skalärprodukt:

```
_Alignas(__m128) float us[4] = { 1, 2, 3 };
_Alignas(__m128) float vs[4] = { 2, 4, 6 };
float p;
```

Exempel: Skalärprodukt

- Hur många instruktioner?
- Om vi har fyra par vektorer?

```
__m128 u = _mm_load_ps(us);
__m128 v = _mm_load_ps(vs);
__m128 w = _mm_mul_ps(u, v);
w = _mm_hadd_ps(w, w);
w = _mm_hadd_ps(w, w);
p = _mm_cvtss_f32(w);
```

Exempel: Skalärprodukt

- Föregående exempel var ett exempel på en *horisontell* operation.
- Vi kan förbättra detta genom att göra det *vertikalt*:

```
_Alignas(__m128) float us[4][4] = { { 1, 2, 3, 3, },
                                    { 7, 2, 2, 9, },
                                    { 5, 9, 6, 7, }, };
_Alignas(__m128) float vs[4][4] = { { 2, 4, 6, 5, },
                                    { 1, 3, 9, 9, },
                                    { 9, 8, 1, 1, }, };
_Alignas(__m128) float ps[4];
```

Exempel: Skalärprodukt

- Hur många instruktioner har vi i detta fall?

```
__m128 ux = _mm_load_ps(us[0]);
__m128 vx = _mm_load_ps(vs[0]);
__m128 uy = _mm_load_ps(us[1]);
__m128 vy = _mm_load_ps(vs[1]);
__m128 uz = _mm_load_ps(us[2]);
__m128 vz = _mm_load_ps(vs[2]);
__m128 px = _mm_mul_ps(ux, vx);
__m128 py = _mm_mul_ps(uy, vy);
__m128 pz = _mm_mul_ps(uz, vz);
__m128 p = _mm_add_ps(px, py);
p = _mm_add_ps(p, pz);
_mm_store_ps(ps, p);
```

SIMD intrinsics i rust? 😱


```rust
use std::arch::x86_64::*;

fn main() {
    unsafe {
        let a = _mm_set_ps(0.0, 1.0, 2.0, 3.0);
        let b = _mm_set_ps(4.0, 5.0, 6.0, 7.0);

        let mut r = _mm_mul_ps(a, b);
        r = _mm_hadd_ps(r, r);
        r = _mm_hadd_ps(r, r);
        r = _mm_cvtss_f32(r);

        println!("{}", r);
    }
}
```

```
$ rustc simd.rs && ./simd
38
```

Finns även `#![feature(portable_simd)]`, dock experimentell. Se rust docs.

Kompilatortransformeringar

Kompilatorer gör transformationer som kan vara svåra att känna till innan man kikar på maskinkoden.

```c
int ub(int *ptr) {
    int a = *ptr;

    if (ptr == NULL)
        return 1;

    return a;
}
```

```
ub:
    movl    (%rdi), %eax
    ret
```

Short-circuiting

I C och många andra språk är `&&` och `||` *short-circuiting*. Högerledet utvärderas bara om vänsterledet är **sant** (`&&`) eller **falskt** (`||`).

```
int f(int *ptr) {
    if (ptr && *ptr == 0)
        return 1;
    else
        return 0;
}
```

```
f(int*):
    testq   %rdi, %rdi
    je      .LBB0_3
    cmpl    $0, (%rdi)
    je      .LBB0_2
.LBB0_3:
    xorl    %eax, %eax
    retq
.LBB0_2:
    movl    $1, %eax
    retq
```

Short-circuiting


```
struct Vec4 {
    float x, y, z, w;
};

bool cmp_sisd(Vec4 a, Vec4 b) {
    return  a.x == b.x &&
            a.y == b.y &&
            a.z == b.z &&
            a.w == b.w ;
}
```

```
cmp_sisd(Vec4, Vec4):
    ucomiss %xmm2, %xmm0
    jne     .LBB0_4
    jp      .LBB0_4
    cmpeqps %xmm2, %xmm0
    pextrb  $4, %xmm0, %eax
    testb   $1, %al
    je      .LBB0_4
    ucomiss %xmm3, %xmm1
    jne     .LBB0_4
    jp      .LBB0_4
    cmpeqps %xmm3, %xmm1
    pextrb  $4, %xmm1, %eax
    andb    $1, %al
    retq
.LBB0_4:
    xorl    %eax, %eax
    andb    $1, %al
    retq
```

Short-circuiting

Om vi vet att vi inte behöver short-circuiting, och vår kompilator inte optimerar det automatiskt, så kan vi göra en snabbare implementation med SIMD.

```
struct Vec4 {
    alignas(__m128) /* Viktigt! */
    float x, y, z, w;
};

bool cmp_simd(Vec4 a, Vec4 b) {
    __m128 xmm_a = _mm_load_ps(&a.x);
    __m128 xmm_b = _mm_load_ps(&b.x);
    __m128 eq = _mm_cmpeq_ps(xmm_a, xmm_b);
    return _mm_movemask_ps(eq) == 15;
}
```

```
cmp_simd(Vec4, Vec4):
    movaps  72(%rsp), %xmm0
    cmpeqps 8(%rsp), %xmm0
    movmskps  %xmm0, %eax
    cmpb    $15, %al
    sete    %al
    retq
```

Läxa

1. Använd SIMD för att optimera antingen
   - er raytracer, eller
   - någon annan algoritm (t.ex. Mandelbrotmängden, matrismultiplikation...).
2. Använd något brenchmarkingbibliotek för att jämföra lösningar med och utan SIMD. För att få meningsfulla resultat kan ni prova att kompilera utan optimeringar.
3. Presentera resultaten i er `README`.
4. Lägg kod och `README` i ett repo med namnet `<kth-id>-simd`. Om ni väljer att jobba vidare på raytracern, lägg endast en `README` med resultaten i det nya repot, och beskriv vilka ändringar ni gjort.

Övning: Romerska siffor

- Skriv en funktion för att omvandla tal representerade som romerska siffror till heltal.
- Exempel: `f("iv") = 4`, `f("xxii") = 22`, `f("mcdxl") = 1440`.

$$I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000.$$