SIMD instructions with Rust on Android

Zürich Rust Meetup

Guillaume Endignoux

Thursday 8th June, 2023

https://github.com/gendx
https://gendignoux.com
Opinions presented here are my own, not my (past present or future) employer's.

What are SIMD instructions?

Arithmetic in a CPU

CPU $\approx$ registers (memory) + operations (hard-wired circuits).

- XOR (8-bit)

```
lhs:  0100 0100
rhs:  1101 0010
   =  1001 0110
```

- Addition (8-bit)

```
      lhs:    0100 0100
      rhs: +  1101 0010
    carry: +  1000 000.
            =  0001 0110
```

Arithmetic in a CPU

Over time, registers get wider...

- XOR (16-bit)

```
lhs: 0100 0100 1010 1111
rhs: 1101 0010 0010 1000
  = 1001 0110 1000 0111
```

- Addition (16-bit)

```
     lhs:   0100 0100 1010 1111
     rhs: + 1101 0010 0010 1000
   carry: + 1000 0000 0101 000.
          = 0001 0110 1101 0111
```

Arithmetic in a CPU

...and wider.

- XOR (32-bit)

```
lhs: 44af 260c
rhs: d228 1a35
  = 9687 3c39
```

- Addition (32-bit)

```
    lhs:   44af 260c
    rhs: + d228 1a35
  carry: + 0010 101.
         = 16d7 4041
```

Arithmetic in a CPU

...and wider.

- XOR (64-bit)

```
lhs: 44af 260c 28cc 7af0
rhs: d228 1a35 04c3 f815
  =  9687 3c39 2
```

- Addition (64-bit)

```
lhs:     44af 260c 28cc 7af0
rhs: +   d228 1a35 04c3 f815
         1010 0111 110.
         4041 2d90 7305
```

How wide can we go?

Beyond 64 bits?

- CPU registers can be (and became) wider.
- Arithmetic on $\geq$ 128 bits isn't really useful.

Single Instruction Multiple Data

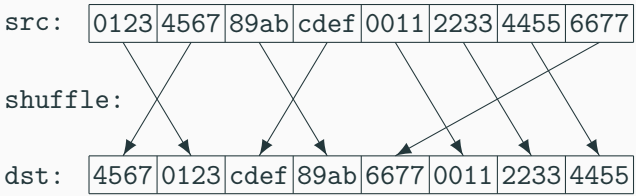
⇒ SIMD (*Single Instruction Multiple Data*): divide a wide register into an array of *lanes* to operate on, e.g. `u128 = [u32; 4]`.

- Addition (`u32x4`)

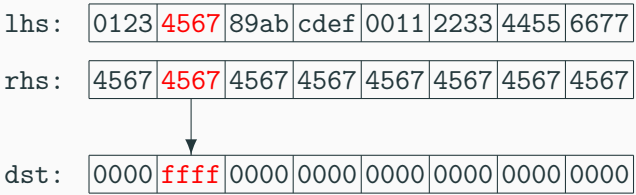
`lhs:`		`44af 260c`	`28cc 7af0`	`2b45 0e65`	`53c0 cbf0`
`rhs:`	`+`	`d228 1a35`	`04c3 f815`	`00ce 191b`	`6250 c7a4`
`carry:`	`+`	`0010 101.`	`0111 110.`	`0110 101.`	`0101 110.`
	`=`	`16d7 4041`	`2d90 7305`	`2c13 2780`	`b611 9394`

More exotic instructions

- Shuffle:



- Comparison:



Application: Loop vectorization

Several items at once = faster loop.

```
fn add_slices(x: &mut [u32], y: &[u32]) {
    for i in 0..x.len() {
        x[i] += y[i];
    }
}
```

```
fn add_slices_simd(x: &mut [u32], y: &[u32]) {
    // Add 4 elements at a time.
    [x[0],x[1],x[2],x[3]] += [y[0],y[1],y[2],y[3]];
    [x[4],x[5],x[6],x[7]] += [y[4],y[5],y[6],y[7]];
    ...
}
```

Application: Special instructions

Many instructions at once = faster.

```rust
fn shuffle(x: &mut [u32; 4]) {
    // Many loads and stores.
    let a = x[1];
    let b = x[2];
    let c = x[0];
    let d = x[3];
    x[0] = a;
    x[1] = b;
    x[2] = c;
    x[3] = d;
}
```

```rust
fn shuffle_simd(x: &mut [u32; 4]) {
    // One instruction.
    __shuffle__(x, [1, 2, 0, 3])
}
```

Application: Special instructions

A whole loop at once!

```rust
fn poly_mul(a: u64, b: u64) -> u128 {
    // Polynomial multiplication.
    let mut tmp: u128 = b as u128;
    let mut result: u128 = 0;
    // Loop many times!
    for i in 0..64 {
        if a & (1 << i) != 0 {
            result ^= tmp;
        }
        tmp <<= 1;
    }
    result
}
```

```rust
fn poly_mul_simd(a: u64, b: u64) -> u128 {
    // One instruction.
    __pmul__(a, b)
}
```

Lots of potential

SIMD-optimized algorithms are everywhere:

- Rust's `HashMap` implementation, using the *Swiss Tables* algorithm.[1]
- *SIMD-friendly algorithms for substring searching*,[2] used by the `memchr` crate.
- *Parsing gigabytes of JSON per second*.[3]

[1] https://abseil.io/blog/20180927-swisstables
[2] http://0x80.pl/articles/simd-strfind.html
[3] https://github.com/simdjson/simdjson

Challenge #1: Calling SIMD from Rust

Example

```
fn main() {
    let x: [u32; 4] = [0x11111111, 0x22222222, 0x44444444, 0x88888888];
    let y: [u32; 4] = [0x99999999, 0xaaaaaaaa, 0xcccccccc, 0xffffffff];
    let z = add_u32x4(x, y);

    println!("x = {x:08x?}");
    println!("y = {y:08x?}");
    println!("z = {z:08x?}");
}

fn add_u32x4(x: [u32; 4], y: [u32; 4]) -> [u32; 4] {
    todo!("Use SIMD instructions")
}
```

```
x = [11111111, 22222222, 44444444, 88888888]
y = [99999999, aaaaaaaa, cccccccc, ffffffff]
```

Inline assembly

```
fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        asm!(
            "paddd {x} {y}",
            x = inout(reg) x,
            y = in(reg) y,
        );
        x
    }
}
```

Inline assembly was recently stabilized.[a] Let's try it!

- First challenge: find the instruction's name.

[a] https://doc.rust-lang.org/reference/inline-assembly.html

A little type error

```
fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        asm!(
            "paddd {x} {y}",
            x = inout(reg) x,
            y = in(reg) y,
        );
        x
    }
}
```

```
error: cannot use value of type `[u32; 4]` for
↪  inline assembly
  --> src/main.rs:21:16
   |
21 | x = inout(reg) x,
   |                ^
   |
   = note: only integers, floats, SIMD vectors,
   ↪  pointers and function pointers can be
   ↪  used as arguments for inline assembly
```

Let's transmute to u128

```
fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        let xx: u128 = transmute(x);
        let yy: u128 = transmute(y);
        asm!(
            "paddd {x} {y}",
            x = inout(reg) xx,
            y = in(reg) yy,
        );
        transmute(xx)
    }
}
```

```
error: type `u128` cannot be used with this
↪  register class
  --> src/main.rs:23:16
   |
23 | x = inout(reg) xx,
   |                ^^
   |
   = note: register class `reg` supports these
   ↪  types: i16, i32, i64, f32, f64
```

Intel's docs mention `xmm`. Let's try with `xmm_reg` perhaps?

```
fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        let xx: u128 = transmute(x);
        let yy: u128 = transmute(y);
        asm!(
            "paddd {x} {y}",
            x = inout(xmm_reg) xx,
            y = in(xmm_reg) yy,
        );
        transmute(xx)
    }
}
```

```
error: type `u128` cannot be used with this
↪  register class
  --> src/main.rs:23:20
   |
23 | x = inout(xmm_reg) xx,
   |                    ^^
   |
   = note: register class `xmm_reg` supports
   ↪  these types: i32, i64, f32, f64, i8x16,
   ↪  i16x8, i32x4, i64x2, f32x4, f64x2
```

Ok, let's try the new `i32x4` type

```rust
#![feature(portable_simd)]
use std::simd::i32x4;

fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        let xx: i32x4 = transmute(x);
        let yy: i32x4 = transmute(y);
        asm!(
            "paddd {x} {y}",
            x = inout(xmm_reg) xx,
            y = in(xmm_reg) yy,
        );
        transmute(xx)
    }
}
```

```
error[E0384]: cannot assign twice to immutable
↪  variable `xx`
  --> src/main.rs:21:1
   |
19 |   let xx: i32x4 = transmute(x);
   |       --
   |       |
   |       first assignment to `xx`
   |       help: consider making this binding
   ↪  mutable: `mut xx`
20 |   let yy: i32x4 = transmute(y);
21 | / asm!(
22 | |     "paddd {x} {y}",
23 | |     x = inout(xmm_reg) xx,
24 | |     y = in(xmm_reg) yy,
25 | | );
   | |_^ cannot assign twice to immutable
   ↪  variable
```

Better with `mut` :) But now a syntax error?!

```rust
#![feature(portable_simd)]
use std::simd::i32x4;

fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        let mut xx: i32x4 = transmute(x);
        let yy: i32x4 = transmute(y);
        asm!(
            "paddd {x} {y}",
            x = inout(xmm_reg) xx,
            y = in(xmm_reg) yy,
        );
        transmute(xx)
    }
}
```

```
error: unexpected token in argument list
  --> src/main.rs:22:2
   |
22 | "paddd {x} {y}",
   |  ^
   |
note: instantiated into assembly here
  --> <inline asm>:2:13
   |
2  |     paddd xmm1 xmm0
   |               ^
```

Ok, a missing comma

```
#![feature(portable_simd)]
use std::simd::i32x4;

fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        let mut xx: i32x4 = transmute(x);
        let yy: i32x4 = transmute(y);
        asm!(
            "paddd {x}, {y}",
            x = inout(xmm_reg) xx,
            y = in(xmm_reg) yy,
        );
        transmute(xx)
    }
}
```

```
x = [11111111, 22222222, 44444444, 88888888]
y = [99999999, aaaaaaaa, cccccccc, ffffffff]
z = [aaaaaaaa, cccccccc, 11111110, 88888887]
```

What about the options?!

```
#![feature(portable_simd)]
use std::simd::i32x4;

fn add_u32x4(x: [u32; 4], y: [u32; 4]) ->
↪  [u32; 4] {
    unsafe {
        let mut xx: i32x4 = transmute(x);
        let yy: i32x4 = transmute(y);
        asm!(
            "paddd {x}, {y}",
            x = inout(xmm_reg) xx,
            y = in(xmm_reg) yy,
            options(pure, nomem, nostack),
        );
        transmute(xx)
    }
}
```

```
x = [11111111, 22222222, 44444444, 88888888]
y = [99999999, aaaaaaaa, cccccccc, ffffffff]
z = [aaaaaaaa, cccccccc, 11111110, 88888887]
```

Language ergonomics: SIMD intrinsics

Intrinsics = functions wrapping the low-level instructions with Rust types.

- Defined in `core::arch::*`.
- Naming modeled after vendor manuals.[4]

[4] `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index`, `https://developer.arm.com/architectures/instruction-sets/intrinsics/`

Language ergonomics: SIMD intrinsics

```rust
use std::arch::x86_64::{__m128i, _mm_add_epi32};

fn add_u32x4(x: [u32; 4], y: [u32; 4]) -> [u32; 4] {
    unsafe {
        // Transmute to vendor-specific type __m128i.
        let xx: __m128i = transmute(x);
        let yy: __m128i = transmute(y);
        // Addition by lanes of 32 bits.
        let zz: __m128i = _mm_add_epi32(xx, yy);
        transmute(zz)
    }
}
```

Challenge #2: Writing cross-platform algorithms

A fragmented ecosystem

Are these the same instructions?

- `_mm_clmulepi64_si128` (Intel)
- `vmull_p64` (ARM64)
- `pclmulqdq` (Intel)
- `pmull` (ARM64)

A fragmented ecosystem

Levenshtein distance?

- `_mm_clmulepi64_si128` (Intel)
- `vmull_p64` (ARM64)

A fragmented ecosystem

Checking the Rust docs?

- `_mm_clmulepi64_si128` (Intel)
- `vmull_p64` (ARM64)

Function core::arch::x86_64::_mm_clmulepi64_si128

```
pub unsafe fn _mm_clmulepi64_si128(
    a: __m128i,
    b: __m128i,
    const IMM8: i32
) -> __m128i
```

Available on **(x86 or x86-64) and target feature `pclmulqdq` and x86-64** only.

[–] Performs a carry-less multiplication of two 64-bit polynomials over the finite field GF(2^k).

The immediate byte is used for determining which halves of `a` and `b` should be used. Immediate bits other than 0 and 4 are ignored.

Intel's documentation

Function core::arch::aarch64::vmull_p64

```
pub unsafe fn vmull_p64(a: u64, b: u64) -> u128
```

Available on **AArch64 and target feature `neon`, `aes`** only.

[–] Polynomial multiply long

Arm's documentation

A fragmented ecosystem

Checking the vendor manuals?

Synopsis

```
__m128i _mm_clmulepi64_si128 (__m128i a, __m128i b, const int imm8)
#include <wmmintrin.h>
Instruction: pclmulqdq xmm, xmm, imm8
CPUID Flags: PCLMULQDQ
```

Description

Perform a carry-less multiplication of two 64-bit integers, selected from `a` and `b` according to `imm8`, and store the results in `dst`.

Operation

```
IF (imm8[0] == 0)
        TEMP1 := a[63:0]
ELSE
        TEMP1 := a[127:64]
FI
IF (imm8[4] == 0)
        TEMP2 := b[63:0]
ELSE
        TEMP2 := b[127:64]
FI
FOR i := 0 to 63
        TEMP[i] := (TEMP1[0] and TEMP2[i])
        FOR j := 1 to i
                TEMP[i] := TEMP[i] XOR (TEMP1[j] AND TEMP2[i-j])
        ENDFOR
        dst[i] := TEMP[i]
ENDFOR
FOR i := 64 to 127
        TEMP[i] := 0
        FOR j := (i - 63) to 63
                TEMP[i] := TEMP[i] XOR (TEMP1[j] AND TEMP2[i-j])
        ENDFOR
        dst[i] := TEMP[i]
ENDFOR
dst[127] := 0
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize)    operand1 = Vpart[n, part];
bits(datasize)    operand2 = Vpart[m, part];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d] = result;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

Testing with ARM on Android

$\Rightarrow$ Testing and benchmarking on real hardware is essential.

Testing with ARM on Android

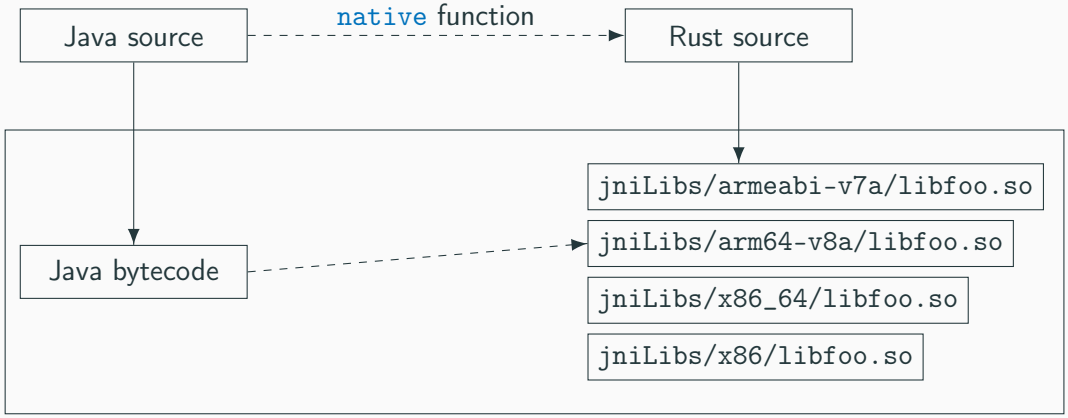
$\Rightarrow$ Testing and benchmarking on real hardware is essential.

My plan:

- Compile Rust code for Android.
- Make sure the SIMD instructions are supported on my device.
- Tests and benchmarks.

Challenge #3: Using Rust on Android (userspace)

Native code in an Android app



Compiling Rust code for an Android app

Summary of the steps:[5]

- Install the `*-linux-android` targets with `rustup`.
- Install a suitable[6] Android NDK (Native Development Kit), and configure it in the linker and the `$PATH`.
- Configure a `.so` output in `Cargo.toml`.
- No stdout, no `println!` → use the `jni` crate and write to `android.util.Log`.

[5] Details in https://gendignoux.com/blog/2022/10/24/rust-library-android.html

[6] Compatible versions depend on your Rust compiler version:
https://blog.rust-lang.org/2023/01/09/android-ndk-update-r25.html

Hello World (simplified)

```
#[no_mangle]
pub unsafe extern "C" fn Java_com_example_myrustapplication_NativeLibrary_nativeRun(
    env: JNIEnv, _: JClass,
) {
    log(env, "MyRustSimdApplication",
        &format!("Your CPU architecture is {}", get_arch_name()));
}

fn get_arch_name() -> &'static str {
    #[cfg(target_arch = "arm")]
    return "arm";
    #[cfg(target_arch = "aarch64")]
    return "aarch64";
    ...
}
```

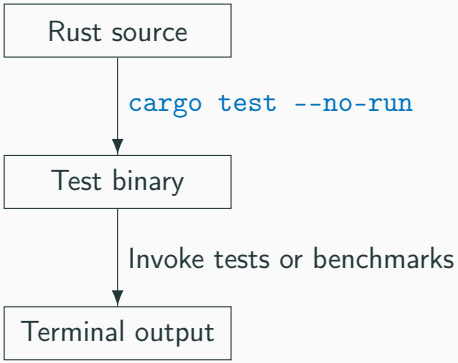
```
D MyRustSimdApplication: Your CPU architecture is aarch64
```

Running unit tests?

`cargo test` and `cargo bench` are convenient. Can we make them work on Android?

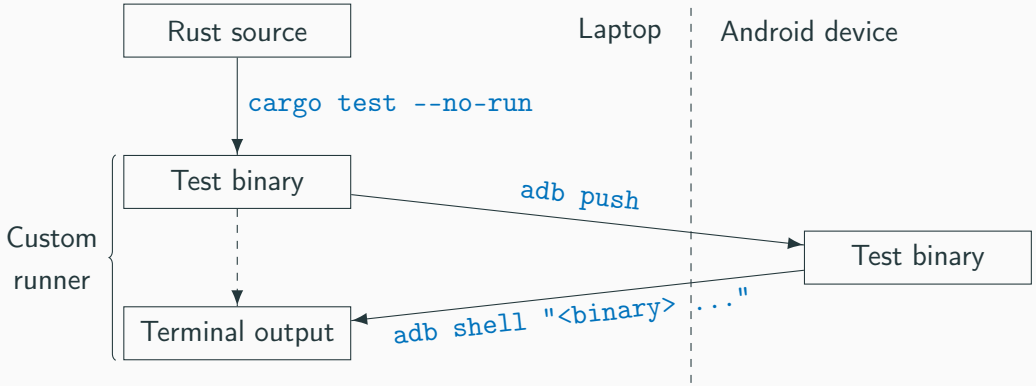
Running unit tests?

`cargo test` and `cargo bench` are convenient. Can we make them work on Android?



Running unit tests?

`cargo test` and `cargo bench` are convenient. Can we make them work on Android?



Custom Cargo runner

- Declare a custom runner in `.cargo/config`:

```
[target.aarch64-linux-android]
runner = "/home/dev/android-runner.sh"
...
```

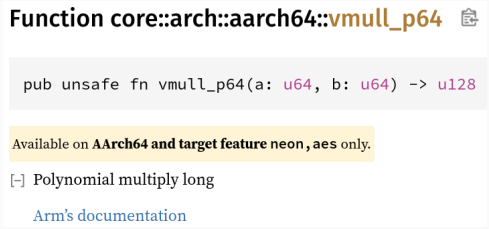
- Simplified runner script:

```
#!/bin/bash
BINARY_PATH="$1"
shift
adb push "${BINARY_PATH}" "/data/local/tmp/binary"
adb shell "chmod 755 /data/local/tmp/binary"
# Run the test binary, forwarding the remaining parameters, so that benchmarks, test
↪  filtering, etc. work.
adb shell "/data/local/tmp/binary $@"
```

Challenge #4: Detecting supported SIMD instructions

Detecting supported instructions

SIMD instructions are useful... but not available everywhere!



⇒ Each CPU model supports its own *target features*.

⇒ Running unsupported instructions will crash the application ("illegal instruction").

Detecting supported instructions

Rust offers two methods to detect features on the current CPU:[7]

- Assume at compile time via `RUSTFLAGS`, e.g. `-C target-feature=+foo`. This crashes the program ("illegal instruction") if run on an unsupported CPU.
- Use the `is_<arch>_feature_detected!` macro at runtime.

[7] https://doc.rust-lang.org/std/arch/index.html#cpu-feature-detection

Detecting supported instructions

Macro std::arch::is_aarch64_feature_detected

```
macro_rules! is_aarch64_feature_detected {
    ("neon") => { ... };
    ("pmull") => { ... };
    ("fp") => { ... };
    ("fp16") => { ... };
    ("sve") => { ... };
    ("crc") => { ... };
    ("lse") => { ... };
    ("lse2") => { ... };
```

```
Detected 3 enabled features:
    [asimd, fp, neon]
Detected 39 disabled features:
    [aes, bf16, bti, crc, dit, dotprod, dpb, dpb2, f32mm, f64mm, fcma, fhm, flagm, fp16,
    ↪  frintts, i8mm, jsconv, lse, lse2, mte, paca, pacg, pmull, rand, ...]
```

Detecting supported instructions

The `vmull_p64` instruction doesn't seem supported... let's YOLO it?

```rust
pub fn pmul_cheat(a: u64, b: u64) -> (u128, &'static str) {
    // FIXME: Here we cheat and omit to detect the "aes" feature.
    if is_aarch64_feature_detected!("neon") {
        return unsafe { pmul_aarch64_neon(a, b) };
    }
    pmul_nosimd(a, b)  // Fallback implementation without any SIMD instruction.
}

#[cfg(target_arch = "aarch64")]
#[target_feature(enable = "neon", enable = "aes")]
unsafe fn pmul_aarch64_neon(a: u64, b: u64) -> (u128, &'static str) {
    (vmull_p64(a, b), "simd implementation")
}
```

Detecting supported instructions

Behind the scenes of `is_<arch>_feature_detected`:

- Intel: special `cpuid` instruction. Just works :)
- ARM: also a special instruction, but only works in kernel mode...

Detecting supported instructions

Behind the scenes of `is_<arch>_feature_detected`:

- Intel: special `cpuid` instruction. Just works :)
- ARM: also a special instruction, but only works in kernel mode...

On Linux, 3 methods implemented by the `std-detect` crate:

- `getauxval()` function,
- `/proc/self/aux` file,[8]
- `/proc/cpuinfo` file.

⇒ Manually using these methods finds more features than what Rust detects!

[8] Not readable for Android apps in release mode.

Detecting supported instructions

Problem: Android wasn't recognized as a Linux-like OS.[9]

```
} else if #[cfg(all(target_os = "linux", feature = "libc"))] {
    #[path = "os/linux/mod.rs"]
    mod os;
...
} else {
    #[path = "os/other.rs"]
    mod os;
}
```

[9] `target_os` is set to `"android"` here: https://github.com/rust-lang/rust/blob/1.65.0/compiler/rustc_target/src/spec/android_base.rs#L5.

Patching the Rust compiler

My first compiler patch.[10]

```
-} else if #[cfg(all(target_os = "linux", feature = "libc"))] {
+} else if #[cfg(all(any(target_os = "linux", target_os = "android"), feature = "libc"))] {
```

⇒ 4 more features

```
Detected 7 enabled features:
    [aes, asimd, crc, fp, neon, pmull, sha2]
Detected 35 disabled features:
    [bf16, bti, dit, dotprod, dpb, dpb2, f32mm, f64mm, fcma, fhm, flagm, fp16, frintts,
    ↪  i8mm, jsconv, lse, lse2, mte, paca, pacg, rand, rcpc, rcpc2, ...]
```

[10] https://github.com/rust-lang/stdarch/pull/1351

Intermezzo: Building a local Rust compiler to test my patch

Building a patched Rust compiler

- Follow the basic instructions:[11]

```
./x.py setup
./x.py build --stage 0
```

- The NDK is everywhere (here in `config.toml`):

```
[target.aarch64-linux-android]
android-ndk = ".../android-sdk/ndk/25.1.8937393/toolchains/llvm/prebuilt/linux-x86_64"
...
```

[11] https://rustc-dev-guide.rust-lang.org/building/how-to-build-and-run.html

Building a patched Rust compiler

- Build a stage 1 compiler (≈ `std::*`) for all Android targets + **your host target**

```
./x.py build --stage 1 \
    --target x86_64-unknown-linux-gnu \
    --target aarch64-linux-android \
    ...
```

- Add the new toolchain to `rustup`

```
rustup toolchain link stage1 build/x86_64-unknown-linux-gnu/stage1
...
cargo +stage1 build
```

Building `rustc` in 2022: What resources?

Disk usage ≈ 10 GB

- `stage0-rustc/` ≈ 4 GB
- each `stageN-std/` (one per target) ≈ 500 MB
- 45% in the `incremental/` folders

Runtime on my laptop:

CPUs	RAM	stage 0	stage 1
1	2 GB	3:46	31:23
2	2 GB	3:12	20:44
4	3 GB	2:45	15:09
8	5 GB	2:44	13:08

Challenge #5: Performance pitfalls in practice

Are SIMD intrinsics a zero-cost abstraction?

The promise of SIMD intrinsics:

- Benefit from Rust's type-checking.
- No need to hand-roll assembly to get the best performance.

Are SIMD intrinsics a zero-cost abstraction?

```rust
use std::arch::aarch64::*;

fn foo(/* ... */) {
    bar(&mut x, &k);
    bar(&mut y, &k);
}

#[inline(always)]
fn bar(block: &mut uint8x16_t, key:
↪  &uint8x16_t) {
    unsafe {
        let zero = vdupq_n_u8(0);
        let x = vaeseq_u8(block.0, zero);
        let y = vaesmcq_u8(x);
        block.0 = veorq_u8(y, key.0);
    }
}
```

Case study:

- Top-level function `foo()` → intermediate functions → intrinsics.
- Goal: inline everything inside `foo()` (hot path).
- Goal: don't re-write `foo()` for the fallback implementation (only change the low-level primitives).

Are SIMD intrinsics a zero-cost abstraction?

```rust
use std::arch::aarch64::*;

fn foo(/* ... */) {
    bar(&mut x, &k);
    bar(&mut y, &k);
}

#[inline(always)]
fn bar(block: &mut uint8x16_t, key:
↪  &uint8x16_t) {
    unsafe {
        let zero = vdupq_n_u8(0);
        let x = vaeseq_u8(block.0, zero);
        let y = vaesmcq_u8(x);
        block.0 = veorq_u8(y, key.0);
    }
}
```

```
foo:
        ...
        stp     q0, q1, [sp, #48]
        bl      core_arch__arm_shared__crypto__vaeseq_u8
        add     x0, sp, #256
        add     x1, sp, #112
        bl      core_arch__arm_shared__crypto__vaesmcq_u8
        adrp    x8, .LCPI2_0
        ...

; SIMD intrinsics, not inlined!
core_arch__arm_shared__crypto__vaesmcq_u8:
        ldr     q0, [x1]
        aesmc   v0.16b, v0.16b  ; SIMD instruction
        str     q0, [x0]
        ret

core_arch__arm_shared__crypto__vaeseq_u8:
        ldr     q0, [x1]
        ldr     q1, [x2]
        aese    v0.16b, v1.16b
        str     q0, [x0]
        ret
```

Are SIMD intrinsics a zero-cost abstraction?

```rust
use std::arch::aarch64::*;

fn foo(/* ... */) {
    bar(&mut x, &k);
    bar(&mut y, &k);
}

#[inline(always)]
fn bar(block: &mut uint8x16_t, key:
↪  &uint8x16_t) {
    unsafe {
        let zero = vdupq_n_u8(0);
        let x = vaeseq_u8(block.0, zero);
        let y = vaesmcq_u8(x);
        block.0 = veorq_u8(y, key.0);
    }
}
```

Inlined and **5x** faster[a] with

`RUSTFLAGS='-C target-feature=+aes'`

```
foo:
        ldp     q0, q2, [x1]
        movi    v1.2d, #0000000000000000
        adrp    x8, .LCPI0_0
        adrp    x9, .LCPI0_1
        mov     v3.16b, v0.16b
        aese    v3.16b, v1.16b  ; vaeseq_u8
        aesmc   v3.16b, v3.16b  ; vaesmcq_u8
        ldr     q4, [x8, :lo12:.LCPI0_0]
        mov     v6.16b, v2.16b
        ldr     q5, [x9, :lo12:.LCPI0_1]
        aese    v6.16b, v1.16b
        ...
```

[a] Inlining is an essential optimization, see *Can You Trust a Compiler to Optimize Your Code?* by matklad.

When target features get in the way of inlining...

Intrinsics definition (simplified) in the standard library:

```
#[inline]
#[target_feature(enable = "aes")]
pub unsafe fn vaeseq_u8(data: uint8x16_t, key: uint8x16_t) -> uint8x16_t {
    vaeseq_u8_(data, key)
}
```

Actual definition: defer to LLVM.

```
#[allow(improper_ctypes)]
extern "unadjusted" {
    #[cfg_attr(target_arch = "aarch64", link_name = "llvm.aarch64.crypto.aese")]
    fn vaeseq_u8_(data: uint8x16_t, key: uint8x16_t) -> uint8x16_t;
}
```

When target features get in the way of inlining...

Intrinsics definition (simplified) in the standard library:

```
#[inline]
#[target_feature(enable = "aes")]
pub unsafe fn vaeseq_u8(data: uint8x16_t, key: uint8x16_t) -> uint8x16_t {
    vaeseq_u8_(data, key)
}
```

- `target_feature(enable = "foo")` annotates the function in a special way.
- Functions with distinct target features cannot be inlined in each other.[12]
- `-C target-feature=+foo` annotates all functions with feature `foo`.

[12] `rust-lang/rust/issues/54353`, `rust-lang/rust/issues/53069`

Be mindful of function calls

```rust
#[target_feature(enable = "aes")]
unsafe fn foo_simd(/* ... */) {
    loop { bar_simd(/* ... */); }
}

// "always" = as long as the caller has
↪  "aes" enabled.
#[inline(always)]
#[target_feature(enable = "aes")]
unsafe fn bar_simd(/* ... */) {
    let zero = vdupq_n_u8(0);
    let x = vaeseq_u8(block.0, zero);
    let y = vaesmcq_u8(x);
    block.0 = veorq_u8(y, key.0);
}
```

Choose your poison:

- avoid function calls in the hot path,
- re-write all the code twice (with and without SIMD),
- compile twice and play with the linker,[a]
- play with macros to generate the code twice.[b]

[a] https://gendignoux.com/blog/2023/01/05/rust-arm-simd-android.html

[b] https://crates.io/crates/multiversion

Conclusion

Conclusion

Intrinsics are not perfect yet, but much better ergonomics than assembly.

- Portable SIMD effort: https://github.com/rust-lang/portable-simd.

Resources:

- Slides: https://gendignoux.com/research/
- Details on my blog: https://gendignoux.com/blog/tags.html#android
- Code: https://github.com/gendx/android-rust-library

Special thanks:

- Rust maintainers: blazingly-fast code review.
- https://godbolt.org