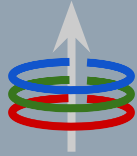


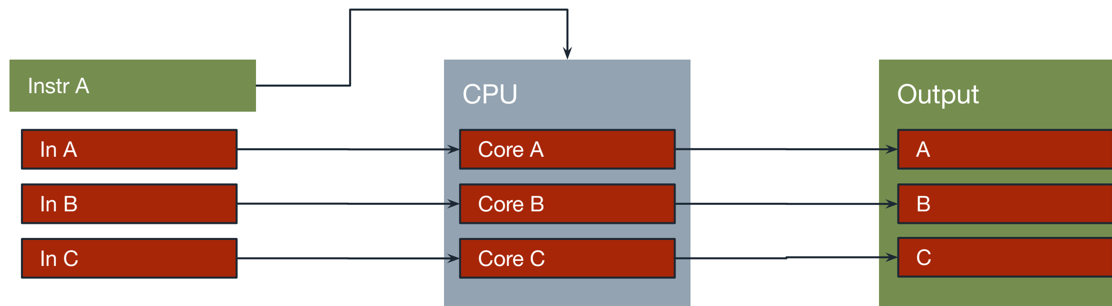
CoE 164

Computing Platforms

12b: Rust SIMD

SIMD

In a **single-instruction multiple-data (SIMD)** processor, multiple processors are loaded with the same instructions, but working on different data units. They are usually used to process smaller inputs to build a larger output.

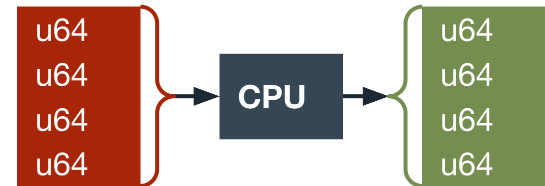


SIMD VECTORIZATION

Modern CPUs employ **vector operations**, which are instructions that process four (4) to eight (8) data in a single cycle.

Vector operations are instruction set extensions that are specific to the CPU and manufacturer.

SIMD VECTORIZATION



Scalar

- One input and output each clock cycle
- Same operations have to be executed in successions

Vector

- Four to eight input and output each clock cycle
- Four to eight same operations can be executed at the same time

SIMD VECTORIZATION

Intel has been at the forefront of adding the following SIMD extensions to their 64-bit processors.

- Streaming SIMD Extensions (SSE 4.2) - 2008 and later
- Advanced Vector Instructions (AVX 2) - 2013 and later

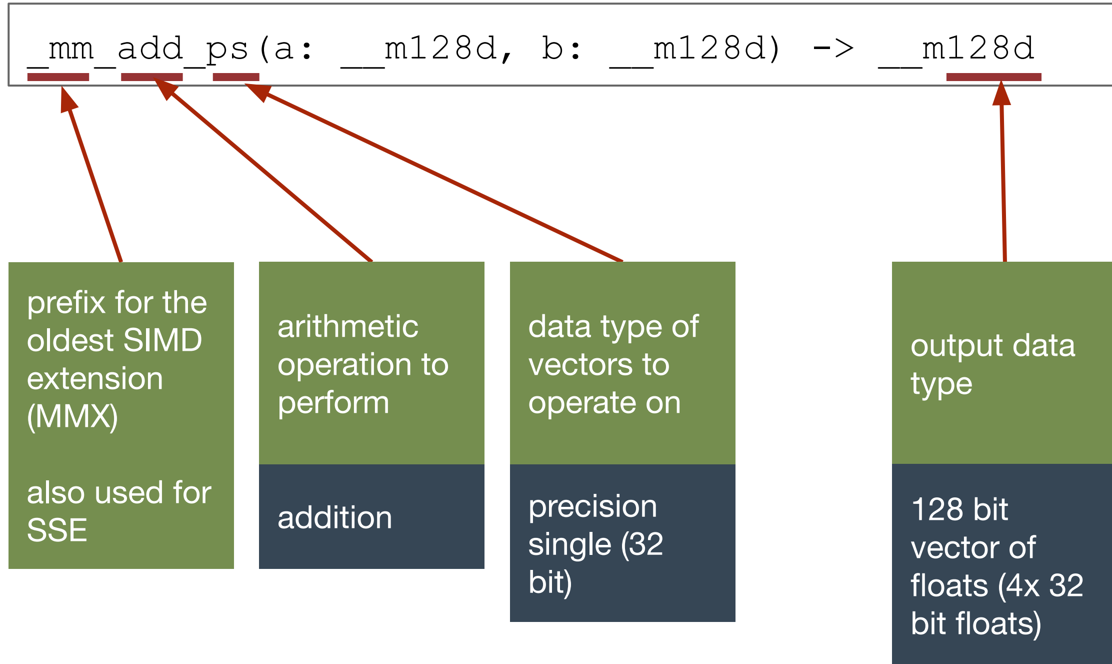
SIMD is platform-dependent!

SIMD INTRINSICS

Operating on vectors requires knowledge of **intrinsics**, or the functions specifically implemented by the CPU manufacturer.

SSE and AVX intrinsics are named based on the operation itself and the data type they are operating on.

SIMD INTRINSICS



SIMD VECTORIZATION

Taking advantage of SIMD vectorization generally follows the three steps:

1. Format data so that it becomes an SIMD vector.
2. Operate on the SIMD vectors.
3. Recover data from the SIMD vectors.

SIMD: LOAD

The `load` intrinsic converts an array or vector of elements into an SIMD vector. The appropriate intrinsic for the data type of the array should be used.

Note that the arguments require conversion of arrays or vectors into its raw and unsafe pointers.

```
// assign 32-bit float 0.3 to all four lanes
let a = _mm_load1_ps(0.3);

// load (unaligned) 4x32 bit floats
let b = _mm_loadu_ps([0.0, 1.0, 2.0, 3.0].as_ptr() as *const _);
```

Example

SIMD: OPERATIONS

SSE and AVX supports various operations on the vectors. Note that both operands should have the same data type.

```
// a = x + (y * z)
let a = _mm_add_ps(x, __mm_mul_ps(y, z));

// b = x & (y | z)
let b = _mm_and_si128(x, __mm_or_si128(y, z));
```

Example

SIMD: OPERATIONS

```
_mm_add_ps(a: __m128d, b: __m128d) -> __m128d
```

Arithmetic	Boolean	Data Types	Lane Lengths
add	and	u8/i8	m128
adds	or	u16/i16	m256
sub	xor	i32	m512
subs	sll	i64	m128i/d/bh
mul	srl	i128	m256i/d/bh
div	sra	ps	m512i/d/bh
		pd	

More operations are available!

SIMD: STORE

The `store` intrinsic converts an SIMD vector into an array. The destination array should be mutable and has the appropriate size for the intrinsic to work.

Note that the arguments require conversion of arrays or vectors into its raw and unsafe pointers.

```
// save x into (unaligned) v
let mut v = [0f32; 4];
_mm_storeu_ps(v.as_mut_ptr() as *mut _, x);

// v now has the values of x
```

Example

SIMD: FEATURE DETECTION

Since SIMD programs are platform-dependent, we should check whether the computer that runs the Rust program supports SSE or AVX using the feature detection macro. Otherwise, it is a good idea to execute a fallback function that *does not* use SIMD intrinsics.

Note that the SIMD intrinsics are unsafe, so execution of functions that use them should be enclosed in an `unsafe` block.

```
#[cfg(any(target_arch="x86_64"))]
{
    if is_x86_feature_detected!("avx2") {
        return unsafe { fcn_avx2() };
    }
}
```

Example

SIMD: FEATURE DETECTION

Example

```
#[cfg(any(target_arch="x86", target_arch="x86_64"))]
{
    if is_x86_feature_detected!("avx2") {
        println!("AVX2 detected!");
        return unsafe { fcn_avx2(dirs) };
    }
    else if is_x86_feature_detected!("sse4.2") {
        return unsafe { fcn_sse42(dirs) };
    }
}

fcn_fallback(dirs);
```

SIMD: FEATURE DETECTION

Functions containing SIMD intrinsics should be prefixed with an `unsafe` keyword. `cfg` and `target_feature` attributes should also be set to note which architecture and instruction set extension the function is for.

All SIMD intrinsics are in the `std::arch` package. The appropriate package for the CPU address size (32- or 64-bit) should be imported.

```
#[cfg(any(target_arch="x86_64"))]
#[target_feature(enable="sse4.2")]
unsafe fn fcn_sse42(a: &mut f32, b: &mut [f32]) {
    #[cfg(target_arch = "x86_64")]
    use std::arch::x86_64::*;
}
```

Example

SIMD: PORTABILITY

Rust currently has an experimental portable SIMD library to deal with the feature detection and juggling of intrinsics. Common CPUs can now be made to use SIMD extensions using a platform-agnostic library.

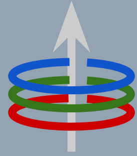
The code from the `portable-simd` crate below shows the basic usage of the said library.

```
#![feature(portable_simd)]
use std::simd::f32x4;
fn main() {
    let a = f32x4::splat(10.0);
    let b = f32x4::from_array([1.0, 2.0, 3.0, 4.0]);
    println!("{:?}", a + b);
}
```

Example

RESOURCES

- The Rust Book
- Intel Intrinsics Reference
- SIMD Tutorial from Utrecht University



CoE 164

Computing Platforms

12b: Rust SIMD