The Pleasure and Pain Of Cross Platform CPU Vector Code In Rust

Richard Neale
mathsDOTearth

May 31, 2025

Introduction

RISC-V HPC Library Performance Optimisation

For my MSc dissertation at EPCC I am looking at RISC-V HPC Library Performance Optimisation. RISC-V technology is developing rapidly and in the near future HPC will be able to take advantage of these developments by optimising common libraries to take advantage of RISC-V RVV vector extensions to provide a wider choice of platforms for research and development.

Github

mathsDOTearth

If you wish to follow along with the the code I wrote for this talk, you can find it at:

https://github.com/mathsDOTearth/SIMDRustTalk2025

Project Prep

Rust In The Beginning

While doing project preparation work in 2024, I initially looked at using Rust as the core language to provide code examples for my dissertation. This did not proceed due to initial lack of a stable version of Rust on the available systems. The dissertation went ahead using C examples.

A Year Later

RISC-V Developments

A year later armed with a Banana Pi F3 and a Milk-V Jupiter both running Fedora 42 I felt it was time to give Rust another try. Having better software support and RISC-V CPUs supporting RVV 1.0, I planned to write a simple test program implementing an xGEMM like function with AVX, NEON and RVV versions.

Step 1

Scalar and AVX2

Using my trusty AMD 5700x Linux PC, I wrote the initial Scalar and AVX2 code. xGEMM, for this example is implemented with xDOT.

This initial step was surprisingly painless!

To gain access to AVX2 instructions I added:

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

Functions containing AVX2 instructions are `unsafe`.

Finally we need to as a `.cargo/config.toml` file:

```
[build]
rustflags = ["-C", "target-cpu=native"]
```

Rayon is used for multi-threading.

First Run

Timings and Tolerance

The test program performs a GEMM operation on two 1024 x 1024 arrays filled with random numbers.

Because AVX looses precision, when we compare the scalar and SIMD calculated results we apply a tolerance of 1e-3 for f32 and le-8 for f64.

The tests are run with `cargo run -r`

AMD 5700x AVX 2 Results

Benchmarking Timing and Speedup

Run	Timing	Speedup
f32 scalar	693.41ms	-
f32 scalar Rayon	88.22ms	7.86x
f32 vector	95.57ms	7.26x
f32 vector Rayon	13.43ms	51.65x
f64 scalar	703.61ms	-
f64 scalar Rayon	84.40ms	8.34x
f64 vector	210.81ms	3.34x
f64 vector Rayon	25.06ms	28.08x

Number of Rayon Threads: 16

Step 2

ARM NEON

Next ARM NEON is implemented on an Oracle ARM Cloud computer. To gain access to ARM NEON instructions we use:

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

To tell the Rust compiler which machine specific code to build we now add in `target_arch` statements.

For the x86_64 code we prepend code with:

```
#[cfg(target_arch = "x86_64")]
```

For the ARM code we prepend with:

```
#[cfg(target_arch = "aarch64")]
```

ARM Neoverse N1 NEON Results

Benchmarking Timing and Speedup

Run	Timing	Speedup
f32 scalar	1.27s	-
f32 scalar Rayon	323.74ms	3.93x
f32 vector	461.54ms	2.75x
f32 vector Rayon	48.87ms	26.00x
f64 scalar	1.29s	-
f64 scalar Rayon	337.20ms	3.83x
f64 vector	940.43ms	1.37x
f64 vector Rayon	121.25ms	10.64x

Number of Rayon Threads: 4

Step 3

x64 AVX512

Next we add AVX512. This is where we meet our first quirks. AVX512F support is only available in the rustc nightly build, so to use these features we have to set rustc to be that version:

```
rustup override set nightly
```

We also need to add the following lines at the top of the code to take advantage of these experimental features:

```
#![cfg_attr(target_arch = "x86_64",
feature(avx512_target_feature))]
#![cfg_attr(target_arch = "x86_64",
feature(stdarch_x86_avx512))]
```

Step 3 - STOP PRESS

x64 AVX512 - STOP PRESS

As of `rustc 1.87.0 (17067e9ac 2025-05-09)` we no longer need to use the `rustc` nightly build, to access the features of AVX512 used in this example, so back to stable:

```
rustup override set stable
```

We also need to remove the following lines at the top of the code:

```
#![cfg_attr(target_arch = "x86_64",
feature(avx512_target_feature))]
#![cfg_attr(target_arch = "x86_64",
feature(stdarch_x86_avx512))]
```

Step 3

x64 AVX512 (cont)

The second quirk is that our XEON CPU that supports AVX512 also supports AVX2 so we need to use a logical `not` to access AVX2 features as being not AVX512. To identify which AVX version we do this:

For x86_64 AVX2 we now use:

```
#[cfg(all(target_arch = "x86_64", not(target_feature =
"avx512f"))]
```

For x86_64 AVX512 we use:

```
#[cfg(all(target_arch = "x86_64", target_feature =
"avx512f"))]
```

The `all` keyword is needed in this case to enable these differing features on the same CPU family.

Intel XEON 8170

Benchmarking Timing and Speedup

Run	Timing	Speedup
f32 scalar	1.16s	-
f32 scalar Rayon	65.30ms	17.81x
f32 vector	126.89ms	9.16x
f32 vector Rayon	7.22ms	161.08x
f64 scalar	1.17s	-
f64 scalar Rayon	79.32ms	14.80x
f64 vector	246.97ms	4.75x
f64 vector Rayon	13.78ms	85.15x

Number of Rayon Threads: 52

The Pain

RISC-V RVV 1.0

This talk is titled The Pleasure and Pain of Developing Cross Platform CPU Vector Code in Rust. And you may be forgiven for wondering where that title came from considering the ease with which AVX2, AVX512 and NEON were implemented.

Let me introduce RISC-V RVV 1.0 in to the mix...

Step 4

RISC-V RVV 1.0

The initial issue is that the support for RISC-V seems to not include intrinsics for the RVV instructions, so for try one we will use `asm!` to embed RISC-V assembly language instructions.

Step 5

Giving Up

I gave up, running low on time I decided in the end to implement the xDOT function for RVV 1.0 in C. I am currently targeting a single RISC-V architecture so I have to admit that I have maybe not written the most flexible code, but it works for my needs.

Using C rather than Rust defeats the point of the original plan and the Rust code is a little cludgy now with the need for RISC-V sections vs everything else.

Step 5

Adding C to the Build

To add the C code we need to create a `build.rs` file to give the C compiler flags, then add in `cc` as a `[build-dependency]` with in our `Cargo.toml` file.

Using C rather than Rust defeats the point of the original plan and the Rust code is a little cludgy now with the need for RISC-V sections vs everything else.

Step 5

Adding C to the Code

Using `#[cfg(target_arch = "riscv64")]` and `#[cfg(not(target_arch = "riscv64"))]`, the code for calling the xGEMM functions is split up. From the RISC-V build, the xDOT functions are called via a `unsafe extern "C"` function call to the functions written in the `vector_dot.c` file.

This code builds and runs with the stable `rustc` compiler.

Spacemit K1

Benchmarking Timing and Speedup

Run	Timing	Speedup
f32 scalar	16.67s	-
f32 scalar Rayon	1.17s	14.22x
f32 vector	2.67s	6.24x
f32 vector Rayon	474.80ms	35.10x
f64 scalar	7.87s	-
f64 scalar Rayon	1.21s	6.51x
f64 vector	3.81s	2.07x
f64 vector Rayon	1.07s	7.34x

Number of Rayon Threads: 8

Conclusion

And in the end...

CPU vector instruction sets offer great opportunities for optimisation of certain workloads within Rust. AVX512 really does stand out above and beyond the other instruction sets I have used (although I have yet to play with ARM SVE2).

Sadly, for my dissertation, RISC-V RVV 1.0 needs work to get full support, and the RISC-V CPUs I have access to are not high performance devices; but the potential is there and in a few years I am sure they will make big strides.

Any Questions?