# Beginner's Guide To SIMD

Hello and welcome to our SIMD basics guide!

Because SIMD is a subject that many programmers haven't worked with before, we thought that it's best to outline some terms and other basics for you to get started with.

## Quick Background

**SIMD** stands for *Single Instruction, Multiple Data*. In other words, SIMD is when the CPU performs a single action on more than one logical piece of data at the same time. Instead of adding two registers that each contain one `f32` value and getting an `f32` as the result, you might add two registers that each contain `f32x4` (128 bits of data) and then you get an `f32x4` as the output.

This might seem a tiny bit weird at first, but there's a good reason for it. Back in the day, as CPUs got faster and faster, eventually they got so fast that the CPU would just melt itself. The heat management (heat sinks, fans, etc) simply couldn't keep up with how much electricity was going through the metal. Two main strategies were developed to help get around the limits of physics.

- One of them you're probably familiar with: Multi-core processors. By giving a processor more than one core, each core can do its own work, and because they're physically distant (at least on the CPU's scale) the heat can still be managed. Unfortunately, not all tasks can just be split up across cores in an efficient way.
- The second strategy is SIMD. If you can't make the register go any faster, you can still make the register *wider*. This lets you process more data at a time, which is *almost* as good as just having a faster CPU. As with multi-core programming, SIMD doesn't fit every kind of task, so you have to know when it will improve your program.

## Terms

SIMD has a few special vocabulary terms you should know:

- **Vector:** A SIMD value is called a vector. This shouldn't be confused with the `Vec<T>` type. A SIMD vector has a fixed size, known at compile time. All of the elements within the vector are of the same type. This makes vectors *similar to* arrays. One difference is that a vector is generally aligned to its *entire* size (eg: 16 bytes, 32 bytes, etc), not just the size of an individual element. Sometimes vector data is called "packed" data.

- **Vectorize**: An operation that uses SIMD instructions to operate over a vector is often referred to as "vectorized".

- **Autovectorization**: Also known as *implicit vectorization*. This is when a compiler can automatically recognize a situation where scalar instructions may be replaced with SIMD instructions, and use those instead.

- **Scalar:** "Scalar" in mathematical contexts refers to values that can be represented as a single element, mostly numbers like 6, 3.14, or -2. It can also be used to describe "scalar operations" that use strictly scalar values, like addition. This term is mostly used to differentiate between vectorized operations that use SIMD instructions and scalar operations that don't.

- **Lane:** A single element position within a vector is called a lane. If you have `N` lanes available then they're numbered from `0` to `N-1` when referring to them, again like an array. The biggest difference between an array element and a vector lane is that in general is *relatively costly* to access an individual lane value. On most architectures, the vector has to be pushed out of the SIMD register onto the stack, then an individual lane is accessed while it's on the stack (and possibly the stack value is read back into a register). For this reason, when working with SIMD you should avoid reading or writing the value of an individual lane during hot loops.

- **Bit Widths:** When talking about SIMD, the bit widths used are the bit size of the vectors involved, *not* the individual elements. So "128-bit SIMD" has 128-bit vectors, and that might be `f32x4`, `i32x4`, `i16x8`, or other variations. While 128-bit SIMD is the most common, there's also 64-bit, 256-bit, and even 512-bit on the newest CPUs.
- **Vector Register:** The extra-wide registers that are used for SIMD operations are commonly called vector registers, though you may also see "SIMD registers", vendor names for specific features, or even "floating-point register" as it is common for the same registers to be used with both scalar and vectorized floating-point operations.
- **Vertical:** When an operation is "vertical", each lane processes individually without regard to the other lanes in the same vector. For example, a "vertical add" between two vectors would add lane 0 in `a` with lane 0 in `b`, with the total in lane 0 of `out`, and then the same thing for lanes 1, 2, etc. Most SIMD operations are vertical operations, so if your problem is a vertical problem then you can probably solve it with SIMD.
- **Reducing/Reduce:** When an operation is "reducing" (functions named `reduce_*`), the lanes within a single vector are merged using some operation such as addition, returning the merged value as a scalar. For instance, a reducing add would return the sum of all the lanes' values.
- **Target Feature:** Rust calls a CPU architecture extension a `target_feature`. Proper SIMD requires various CPU extensions to be enabled (details below). Don't confuse this with `feature`, which is a Cargo crate concept.

## Target Features

When using SIMD, you should be familiar with the CPU feature set that you're targeting.

On `arm` and `aarch64` it's fairly simple. There's just one CPU feature that controls if SIMD is available: `neon` (or "NEON", all caps, as the ARM docs often put it). Neon registers can be used as 64-bit or 128-bit. When doing 128-bit operations it just uses two 64-bit registers as a single 128-bit register.

> *By default, the `aarch64`, `arm`, and `thumb` Rust targets generally do not enable `neon` unless it's in the target string.*

On `x86` and `x86_64` it's slightly more complicated. The SIMD support is split into many levels:

- 128-bit: `sse`, `sse2`, `sse3`, `ssse3` (not a typo!), `sse4.1`, `sse4.2`, `sse4a` (AMD only)
- 256-bit (mostly): `avx`, `avx2`, `fma`
- 512-bit (mostly): a *wide* range of `avx512` variations

The list notes the bit widths available at each feature level, though the operations of the more advanced features can generally be used with the smaller register sizes as well. For example, new operations introduced in `avx` generally have a 128-bit form as well as a 256-bit form. This means that even if you only do 128-bit work you can still benefit from the later feature levels.

> *By default, the `i686` and `x86_64` Rust targets enable `sse` and `sse2`.*

### Selecting Additional Target Features

If you want to enable support for a target feature within your build, generally you should use a target-feature setting within you `RUSTFLAGS` setting.

If you know that you're targeting a specific CPU you can instead use the target-cpu flag and the compiler will enable the correct set of features for that CPU.

The Steam Hardware Survey is one of the few places with data on how common various CPU features are. The dataset is limited to "the kinds of computers owned by people who play computer games", so the info only covers `x86` / `x86_64`, and it also probably skews to slightly higher quality computers than average. Still, we can see that the `sse` levels have very high support, `avx` and `avx2` are quite common as well, and the `avx-512` family is still so early in adoption you can barely find it in consumer grade stuff.

## Running a program compiled for a CPU feature level that the CPU doesn't support is automatic undefined behavior.

This means that if you build your program with `avx` support enabled and run it on a CPU without `avx` support, it's **instantly** undefined behavior.

Even without an `unsafe` block in sight.

This is no bug in Rust, or soundness hole in the type system. You just plain can't make a CPU do what it doesn't know how to do.

This is why the various Rust targets *don't* enable many CPU feature flags by default: requiring a more advanced CPU makes the final binary *less* portable.

So please select an appropriate CPU feature level when building your programs.

## Size, Alignment, and Unsafe Code

Most of the portable SIMD API is designed to allow the user to gloss over the details of different architectures and avoid using unsafe code. However, there are plenty of reasons to want to use unsafe code with these SIMD types, such as using an intrinsic function from `core::arch` to further accelerate particularly specialized SIMD operations on a given platform, while still using the portable API elsewhere. For these cases, there are some rules to keep in mind.

Fortunately, most SIMD types have a fairly predictable size. `i32x4` is bit-equivalent to `[i32; 4]` and so can be bitcast to it, e.g. using `mem::transmute`, though the API usually offers a safe cast you can use instead.

However, this is not the same as alignment. Computer architectures generally prefer aligned accesses, especially when moving data between memory and vector registers, and while some support specialized operations that can bend the rules to help with this, unaligned access is still typically slow, or even undefined behavior. In addition, different architectures can require different alignments when interacting with their native SIMD types. For this reason, any `#[repr(simd)]` type has a non-portable alignment. If it is necessary to directly interact with the alignment of these types, it should be via `mem::align_of`.