Safe Interface Design for a Minimal BLAS-like Kernel Library

LAK Working Note #1

deval-d@stanford.edu

08 May 2026

Abstract

`lak` is a small BLAS-like kernel library being written in safe Rust. Its goal is not to reproduce the full BLAS interface, but to explore how clean a contiguous-memory API can get when you are not bound by historical constraints. The guiding question is whether a safe, idiomatic Rust interface can stay minimal and elegant, sacrificing nothing in the level-1 and level-2 BLAS, and nothing serious in the level-3 BLAS.

1 LAK

Basic Linear Algebra Subprograms (BLAS) is a standardized set of routines for common linear algebra operations: vector addition, scalar multiplication, dot products, matrix-vector products, and matrix multiplication. These routines are the most important building blocks in numerical computing. Libraries like BLIS, Intel's MKL, Apple Accelerate, and OpenBLAS have been refined over years to make these small primitives run as efficiently as possible across many different architectures.

`linalg-kernels`, or `lak`, is a small BLAS-like linear algebra library being written in safe Rust. It is not meant to be a BLAS implementation. Other mature Rust libraries for fast numerical linear algebra, such as `faer` or `nalgebra` also already exist. I am writing `lak` mainly to learn how these routines are designed and written to be fast. There is also a design question behind the project. If I do not try to support the entire historical interface, and instead focus on contiguous memory, simple views, and safe Rust wrappers, how clean can the code be while still matching perf for unit-stride BLAS? These working notes are my attempt to explain that process as I learn it.

2 Safe Interface Design

The standard BLAS interface operates on descriptors:

- vectors: $(x_{\text{ptr}}, n, \text{incx})$
- matrices: $(A_{\text{ptr}}, m, n, \text{lda})$
- scalars: (α, β)

It assumes the caller has provided consistent sizes, strides, and leading dimensions: n, incx, and lda. The caller is therefore responsible for matching shapes to descriptors, avoiding invalid aliasing between mutable arguments, and distinguishing read-only inputs from mutable outputs. If any of these assumptions are wrong, the failure mode is not usually a helpful error. It is undefined behavior (UB): out-of-bounds reads, out-of-bounds writes, or silent memory corruption. The tradeoff is that BLAS routines can expose a very small FFI-friendly low-level interface and let implementations operate directly on raw pointers, without carrying high-level shape information or performing bounds checks inside the inner kernels that make the routine fast. Because numerical linear algebra is everywhere, this tradeoff is fine. BLAS-based libraries make UB a non-issue for most users. Even so, this design is anti-Rust. I sacrifice the portability of the standard BLAS interface in exchange for one that is idiomatic Rust, and only in Rust.

2.1 Vectors and Matrices

`lak` only works with contiguous memory, so BLAS incx strides are unnecessary and n can be inferred directly from the length of the vector's buffer. Only a buffer is needed.

```rust
/// immutable vector type
#[derive(Clone, Copy, Debug)]
pub struct VecRef<'a, T> {
    buffer: &'a [T],
}

/// mutable vector type
#[derive(Debug)]
pub struct VecMut<'a, T> {
    buffer: &'a mut [T],
}
```

The `Ref`/`Mut` split makes read-only and mutable roles explicit. Matrices are similar, though (m, n) dimensions must be supplied explicitly, since the buffer alone does not encode the logical shape.

```rust
/// immutable matrix type
/// stored in column-major order
#[derive(Clone, Copy, Debug)]
pub struct MatRef<'a, T> {
    buffer: &'a [T],
    dimension: (usize, usize),
}

/// mutable matrix type
/// stored in column-major order
#[derive(Debug)]
pub struct MatMut<'a, T> {
    buffer: &'a mut [T],
    dimension: (usize, usize),
}
```

The `Mat`[1] constructors validate that the dimension and buffer are mutually consistent. The connecting thread across both types is that shape information lives in the wrapper, not in the caller's head. Every kernel is now allowed to assume:

- vector lengths are known from the slice length;
- matrix dimensions are checked when the view is constructed;
- immutable inputs and mutable outputs are separated by the type system;
- mutable aliases cannot be created through the safe interface;
- all storage is contiguous and column-major.

Invalid descriptors are ruled out before routines are called and inner kernels can remain safe:

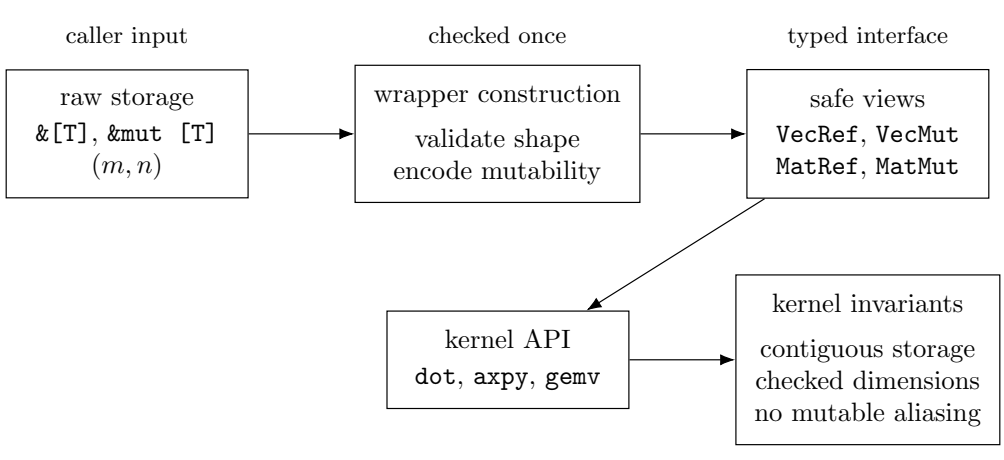

Figure 1: The wrapper types move descriptor validity into construction. Once a `VecRef`, `VecMut`, `MatRef`, or `MatMut` exists, kernels can rely on contiguity, checked shape, and Rust's aliasing rules as interface invariants.

2.2 Generic Kernels

Restricting `lak` to Rust also naturally simplifies how routines are named and implemented. The traditional BLAS interface has separate entry points for each scalar type. For example, single-precision matrix-vector multiplication is called through `sgemv`, while double-precision matrix-vector multiplication is called through `dgemv`. This is natural for a C or Fortran ABI, where the function name itself must encode the scalar type. In `lak`, the public interface does not need separate routines for `f32` and `f64`. A routine can be generic over the scalar type:

```rust
pub fn gemv<T>(
    alpha: T,
    beta: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    y: VecMut<'_, T>
)
where
    T: Float,
    ...
{
    ...
}
```

This removes a form of code duplication. Rust's monomorphization then produces machine code for each concrete scalar type, so the generic interface does not introduce the overhead normally associated with type erasure or dynamic dispatch.

The difficulty is that different scalar types do not have the same performance strategy. A double-precision value is twice as large as a single-precision value. This changes how many elements fit in a cache line, how many elements fit in a SIMD register, and how much matrix data fits in a fixed-size cache block (Figure 2). For example, a cache block that holds N single-precision values only holds $N/2$ double-precision values. Similarly, a 128-bit SIMD register holds four `f32` values but only two `f64` values. As a result, blocking constants, unroll factors, and register-level kernels cannot always be shared blindly between scalar types. A generic interface can expose one routine name, but the implementation still has to respect the different storage and register pressure of each type. One possible solution is to attach these constants to the scalar type itself. However, Rust does not yet allow generic associated constants such as `T::LANES` or `T::NC` to be used freely in const-generic positions, and these constants may be different depending on the routine. SIMD widths and type-specific blocking constants thus remain implementation details.

For the level-1 and level-2 BLAS, this is easily manageable. These routines are limited more by memory traffic than by floating-point throughput. The kernel spends much of its time streaming vectors or matrix columns from memory, so the main requirement is to preserve simple contiguous access and avoid unnecessary loads, stores, and bounds checks. The *same* constants tuned for `f64` can therefore often work well for `f32`. The level-3 BLAS requires more care. Matrix multiplication has enough arithmetic intensity that register blocking, cache blocking, packing layout, and SIMD width become central to performance.

Figure 2: The strategy depends on scalar width: double precision uses twice as many bytes per value, so the same cache or SIMD span holds half as many elements.

[1] `faer` has the same matrix naming convention.