Making a more idiomatic BLAS API

Deval Deliwala

Basic Linear Algebra Subprograms (BLAS) is a set of low-level routines that perform the most common linear algebra routines. These routines include vector addition, scalar multiplication, dot products, and most importantly, matrix multiplication.

They are the *de facto* standard low-level routines for linear-algebra libraries and were originally written in Fortran 77 in 1979. However, its API involved pointers for speed and is still standard today. Here's an example function call to the `GEMM` (General matrix-matrix vector multiply) routine in a C-written BLAS:

```c
void cblas_sgemm(const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE TransA,
                 const CBLAS_TRANSPOSE TransB, const CBLAS_INT M, const CBLAS_INT N,
                 const CBLAS_INT K, const float alpha, const float *A,
                 const CBLAS_INT lda, const float *B, const CBLAS_INT ldb,
                 const float beta, float *C, const CBLAS_INT ldc);
```

It has 14 arguments! Modern programming recommendations suggest functions to have no more than **7** arguments for codebase readability. NASA also suggests not using pointers at all!

The reason the BLAS API is standardized to this day is mostly because of *inertia* and prioritizing explicitness. It is verbose, but is modular, direct, and called from *many* higher-level linear algebra libraries like LAPACK that are also heavily used and have their own API-change inertia.

Put simply,

> *if it ain't broke, don't fix it.*

My first attempt at BLAS

I already made a BLAS in Rust that conforms to this pointer API. Using pointers required wrapping every function in `unsafe{ }` blocks. And because the routines were *already* unsafe, I also used unsafe AArch64 NEON SIMD intrinsics for speed.

You can see my original reddit post introducing it. The top comment said, *"seeing the entire implementations for the kernels wrapped in unsafe {} is rather spooky."*

`u/Shnatsel` was exactly right. At the time I retaliated saying reaching BLAS performance required unsafe code, to which he replied *"A handful of perfectly predictable branches per op should not be a problem"*. He is also Russian, so you know he's cracked.

I realized if I am trying to write a BLAS in Rust, a modern language unique for prioritizing memory-safety and writing idiomatic code, **it should be `safe`, and have a cleaner API**.

Designing the Safe API

Design Goals and Constraints

The standard BLAS interface operates on *descriptors*:
- vectors: (`x_ptr, n, incx`)
- matrices: (`A_ptr, m, n, lda`)
- scalars: (`alpha, beta`)

It *assumes* the user provided correct sizes `n` and strides `incx`. If not, BLAS will happily walk off the end of the buffer!

That means the caller is responsible for
- matching shapes to descriptors.
- ensuring buffers are large enough.
- avoiding overlapping mutable arguments in memory (aliasing).
- not mixing-up read-only vs output arguments.

If you get any of these wrong, you don't get a helpful error, you get *silent memory corruption.*

Rust wants:
- no out-of-bounds reads/writes
- no mutable aliasing (no two `&mut` views to overlapping memory)
- clear lifetimes (a view cannot outlive the slice it views)
- and ideally: APIs that are *hard to misuse*

Hence the job is:

> *encode the BLAS descriptors as safe Rust values whose constructor enforces the descriptor's validity.*

Vectors

A BLAS vector is not defined by contiguous memory. It is defined by a *walk* through memory:
- a starting position.
- a number of elements to walk through.
- and the stride (index increment) between successive elements.

This is why BLAS takes (`x, n, incx`) rather than just some data buffer `x`. The cleanest mental model is

```rust
struct VectorRef<'a, T> {
    data: &'a [T],    // backing storage
    n: usize,         // logical length
    inc: usize,       // stride in elements (BLAS: incx)
    offset: usize,    // starting index within data
}

struct VectorMut<'a, T> {
    data: &'a mut [T],
    n: usize,
    inc: usize,
    offset: usize,
}
```

The type system makes the roles impossible to confuse:
- `VectorRef` means "this routine may only *read* from it."
- `VectorMut` means "this routine may *mutate* it."

Already, correctness at the call site and aliasing safety via `&mut [T]` is ensured.

Preventing out-of-bounds

To ensure the descriptor does not walk out of bounds we have two options:

- check inside every kernel (slow and repetitive)
- check once when *building* a vector (at construction time)

Option 1 means having

```rust
// let x = VectorRef::new(data, n, incx, offset);

let length = n;
let length_to_walk = incx * n;

assert!(length >= length_to_walk, "vector data isn't long enough!");
```

for every vector inside every routine. This goes against DRY (Don't repeat yourself) principles of software development – it's repetitive. Option 2 is better.

Option 2 validates each vector within its own `VectorRef/Mut::new(..)` construction, which either validates the vector and produces a trusted object, or fails early. This means `::new(..)` returns a `Result<Self, BufferError>`.

The constructor thus ensures:
- Out-of-bounds prevention:
  - if a routine iterates `i = 0..n`, accessing `offset + i * incx`,
  - then every accessed index is valid in `data`.
- Meaningful strides
  - `incx != 0`, otherwise the vector is just a scalar.
- Start is in range
  - `offset < data.len()` unless `n == 0` (empty vector).

After this, routines are allowed to work with the vectors. This also improves performance. Compilers are intelligent enough today to mostly avoid bounds-checks entirely when raw indexing (`data[idx]`) if out-of-bounds checking was already explicitly validated.

So the cost model is:
- pay a small constant validation cost at vector construction,
- get a hot inner loop that is as tight as `CBLAS`,
- and get a cleaner `VectorRef/Mut` API instead of `(x, n, incx, stride)` engrossing the function call.

```rust
impl<'a, T> VectorRef<'a, T> {
    /// Constructor
    pub fn new (
        data    : &'a [T],
        n       : usize,
        stride  : usize,
        offset  : usize
    ) -> Result<Self, BufferError> {

        // empty vector
        if n == 0 {
            return Ok( Self {
                data,
                n,
                stride,
                offset
            })
        }

        if stride == 0 {
            return Err(BufferError::ZeroStride);
        }

        let required_length = (n - 1)
            .saturating_mul(stride)
            .saturating_add(offset)
            .saturating_add(1);
        let data_len = data.len();

        // out-of-bounds prevention
        if required_length > data_len {
            return Err(BufferError::OutOfBounds {
                required : required_length,
                len      : data_len
            });
        }

        Ok(Self { data, n, stride, offset })
    }

   /// Getters to access internal data, n, stride, and offset fields
   ...
}
```

Matrices

Matrices justify this type system further.

A plain `&[T]` with an associated `(m, n)` is not a BLAS matrix. Matrices must handle:
- leading dimension (`lda`) to describe the stride between columns (column-major) or rows (row-major)
- padding,
- sub-matrices inside a larger parent,
- and views into panels/tiles during optimized blocked algorithms.

So the matrix descriptor is planned as
- backing storage,
- shape (`m, n`),
- leading dimension `lda`,
- offset.

This is analogous to vectors. However, separating `MatrixRef` and `MatrixMut` is even more important than for vectors. The API again makes the roles explicit:

```rust
/// Immutable Matrix Type
#[derive(Debug, Copy, Clone)]
pub struct MatrixRef<'a, T> {
    data        : &'a [T],
    n_rows      : usize,
    n_cols      : usize,
    lda         : usize,
    offset      : usize
}


/// Mutable Matrix Type
#[derive(Debug)]
pub struct MatrixMut<'a, T> {
    data        : &'a mut [T],
    n_rows      : usize,
    n_cols      : usize,
    lda         : usize,
    offset      : usize
}
```

- `MatrixRef` is read-only, can be freely duplicated and passed around.
- `MatrixMut` is writable, and must be unique to avoid aliasing.

This type system ensures the following:
- Prevent accidental misuse
  - If a function takes `MatrixMut`, the caller can't pass an immutable slice by mistake.
- Enforce non-aliasing
  - `MatrixMut` originates from `&mut [T]`. Safe Rust makes it difficult to have two independent mutable borrows to the same backing buffer at once.
- Enable aggressive kernels
  - Easier for compilers to notice overlap prevention.

Preventing out-of-bounds

In column-major BLAS, the element `(i, j)` is at `offset + i + j*lda`. So `MatrixRef::new(..)` must guarantee:
- every `(i, j)` in bounds maps to an index inside `data`.

This is the matrix-analog of the vector reachability check. The matrix constructor will enforce:
- Valid Leading Dimension:
  - For column-major storage: `lda >= m`,
  - Otherwise a column may not have `m` entries.
- Start is in range
  - `offset < data.len()` unless the matrix is empty.
- End is in range
  - The farthest accessed element is the bottom of the last column:
    - `offset + (m - 1) + (n - 1)*lda` (when `m, n > 0`).
  - This must be `< data.len()`

```rust
impl<'a, T: Copy> MatrixRef<'a, T> {
    /// Constructor
    pub fn new (
        data    : &'a [T],
        n_rows  : usize,
        n_cols  : usize,
        lda     : usize,
        offset  : usize,
    ) -> Result<Self, BufferError> {

        // empty matrix
        if n_rows == 0 || n_cols == 0 {
            return Ok( Self {
                data,
                n_rows,
                n_cols,
                lda,
                offset
            })
        }

        /// empty column
        if lda == 0 {
            return Err(BufferError::ZeroStride);
        }

        if lda < n_rows {
            return Err(BufferError::InvalidLda { lda , n_rows });
        }

        let required_length = (n_cols - 1)
            .saturating_mul(lda)
            .saturating_add(n_rows)
            .saturating_add(offset);
        let data_len = data.len();

        /// ensure reachability
        if required_length > data_len {
            return Err(BufferError::OutOfBounds {
                required : required_length,
                len      : data_len
            });
        }

        Ok( Self { data, n_rows, n_cols, lda, offset })
    }

    /// Getters to access internal data, n_rows, n_cols, lda, and offset
    ...
}
```

Once these rules are validated, kernels can compute addresses with `i + j*lda` freely.

Modern `GEMM` API

The GEMM routine is

$$C \leftarrow \alpha AB + \beta C,$$

where the matrix `C` gets overwritten with the result – a `MatrixMut`. A and B are not mutated, so they are `MatrixRef`. Hence, this API calls `GEMM` via

```rust
let a = vec![...];
let b = vec![...];
let mut c = vec![...];

let aview = MatrixRef::new(&a, m, k, lda, 0)?;
let bview = MatrixRef::new(&b, k, n, ldb, 0)?;
let cview = MatrixMut::new(&mut c, m, n, ldc, 0)?;
sgemm(NoTrans, NoTrans, alpha, beta, aview, bview, cview);
```

The **7** argument suggestion is also satisfied! The first two arguments are `enum` variants that specify `GEMM` does not transpose `A` or `B`.

The caller's intent is also stated more clearly. By separating A, B, and C matrices and their own internal offsets, shapes, leading dimensions:

- "this buffer represents an `m` × `k` matrix with leading dimension `lda`"
- "start at offset 0"
- "and it is read-only (`Ref`) or writeable (`Mut`)"

And if buffers are too small or `lda` is invalid, the routine fails immediately, near the bug, with a *controlled error*. No memory-corruption.

The kernels are free to be fast because the views are validated descriptors:
- avoid bounds checks
- branch to fast paths (contiguous memory) when `lda == m` and `inc == 1`

> This system is an idiomatic API that is clean, strict, and typed.
>
> It feeds into an interior that can be unapologetically fast.
>
> CORAL

Wrappers around this API are written to conform to the standard API and LAPACK.