Level 1 BLAS

Deval Deliwala

Table of Contents

BLAS is divided into three levels:
- Level 1 - vector-vector routines.
- Level 2 - matrix-vector routines.
- Level 3 - matrix-matrix routines.

Each level is progressively more difficult to optimize than the previous. With modern CPUs, Level 1 and 2 are mostly memory-bound. And Level 3 is mostly compute-bound.

Consequently, optimizing Level 1 is straightforward and leans heavily on LLVM and `portable-simd`. `portable-simd` is a `nightly` SIMD interface in the Rust standard library that maps cleanly onto modern vector instructions across architectures.

Previously, I designed a clean API for my BLAS implementation in Rust. It contains `VectorRef` and `VectorMut` types that internally handle vector buffers, strides, and offsets cleanly. The separation of `Ref`/`Mut` types also intuitively allow function calls to be impossible to confuse:

- `VectorRef` means “this routine may only *read* from it.”
- `VectorMut` means “this routine may *write* into it.”

Optimizing the Dot Product

The `sdot` routine calculates the dot product of two vectors:

$$\vec{x} \cdot \vec{y} = \sum_{i=0}^{n-1} x_i y_i.$$

in single precision `f32`s, where n is the length of $\vec{x}$ and $\vec{y}$.

This routine does not mutate or overwrite any vector. It only outputs the calculated `f32` product. So I use `VectorRef`s.

Naive implementation

Contiguous memory means the vector is tightly packed. The next element is at the next index.

For example, consider $x = \binom{0}{1}$. It could be represented as follows:

- **contiguous**:
  - `let x = vec![0, 1];`
  - increment `incx = 1`
- **not contiguous**:
  - `let x = vec![0, 0, 1];`
  - increment `incx = 2`, accessing every other element.

When memory is contiguous, it can all be brought to the CPU together in cachelines. This yields a *much* faster execution time. Consequently I have a *fast* path for when vectors are contiguous (`inc == 1`) and a *slow* path otherwise.

```rust
use crate::types::VectorRef;

/// Computes the dot product of two [VectorRef]s.
#[inline]
pub fn sdot (
    x: VectorRef<'_, f32>,
    y: VectorRef<'_, f32>,
) -> f32 {
    let xn = x.n();
    let yn = y.n();

    if xn != yn {
        panic!("x y vector dimensions must match!");
    }

    // empty vector
    if xn == 0 {
        return 0.0;
    }

    let mut acc_sum = 0.0;

    // fast path
    if let (Some(xs), Some(ys)) = (x.contiguous_slice(), y.contiguous_slice()) {
        for (xk, yk) in xs.iter().zip(ys.iter()) {
            acc_sum += xk * yk;
        }

        return acc_sum;
    }

    // slow path
    let incx = x.stride();
    let incy = y.stride();

    let ix = x.offset();
    let iy = y.offset();

    let xs = x.as_slice();
    let ys = y.as_slice();
    let xs_iter = xs[ix..].iter().step_by(incx).take(xn);
    let ys_iter = ys[iy..].iter().step_by(incy).take(yn);

    for (&xk, &yk) in xs_iter.zip(ys_iter) {
        acc_sum += xk * yk
    }

    acc_sum
}
```

Naive Benchmark

When vectors x and y contain 1024 elements,This routine runs in **750 nanoseconds** on average, which is already extremely fast. On modern CPUs, LLVM can often vectorize patterns like

```
acc_sum += xk * yk
```

into SIMD instructions automatically when the access pattern is simple and contiguous. The work that has gone into making modern compilers like LLVM intelligent enough to do this is incredible.

I’ll show this explicitly in the Assembly section for SAXPY.

Optimized Implementation

However, I can make it faster by writing the SIMD myself. I use `portable-simd`, which “compile to the best available SIMD instructions” for all modern computer architectures. On AArch64 (including Apple M4) architectures, the SIMD interface is called “NEON”.

The algorithm is the same. And SIMD only really works with BLAS when vectors are contiguous, so the slow path stays *exactly* the same.

The rough procedure for working with SIMD in the fast path goes as follows:

```
// define chunk size at compile time
const LANES: usize = <some value>;

// decompose vector into chunks of size LANES
// and the leftover tail of length < LANES
let (chunks, tail) = vector.as_chunks::<LANES>();

for chunk in chunks
    // convert to SIMD vector
    let simd_chunk = Simd::from_array(chunk);
    <do some stuff>
end

// leftover tail scalar path
for value in tail
    <do some stuff>
end
```

I hope my code is readable enough to understand this:

```rust
//! Level 1 [`?DOT`](https://www.netlib.org/lapack/explore-html/d1/dcc/group__dot.
html)
//! routine in single precision.
//!
//! \\[
//! \sum\_{i=0}^{n-1} x_i \\, y_i
//! \\]
//!
//! # Author
//! Deval Deliwala


use std::simd::Simd;
use std::simd::num::SimdFloat;
use crate::types::VectorRef;
use crate::debug_assert_n_eq;


/// Takes the dot product over logical elements in [VectorRef]
/// `x` and `y`.
///
/// Arguments:
/// * `x`: [VectorRef] - over [f32]
/// * `y`: [VectorRef] - over [f32]
///
/// Returns:
/// - [f32] dot product.
#[inline]
pub fn sdot (
    x: VectorRef<'_, f32>,
    y: VectorRef<'_, f32>,
) -> f32 {
    // ensures x and y have same length `n`
    debug_assert_n_eq!(x, y);

    let n = x.n();
    if n == 0 {
        return 0.0;
    }

    // fast path
    if let (Some(xs), Some(ys)) = (x.contiguous_slice(), y.contiguous_slice()) {
        const LANES: usize = 32;
        let mut acc = Simd::<f32, LANES>::splat(0.0);

        let (xv, xt) = xs.as_chunks::<LANES>();
        let (yv, yt) = ys.as_chunks::<LANES>();

        // fixed chunk sizes
        for (&xc, &yc) in xv.iter().zip(yv.iter()) {
            let xm = Simd::from_array(xc);
            let ym = Simd::from_array(yc);

            acc += xm * ym;
        }

        // leftover tail
        let mut acc_tail = 0.0;
        for (xf, yf) in xt.iter().zip(yt.iter()) {
            acc_tail += xf * yf;
        }

        return acc.reduce_sum() + acc_tail;
    }

    // slow path
    let mut acc = 0.0;
    let incx = x.stride();
    let incy = y.stride();
    let ix = x.offset();
    let iy = y.offset();

    let xs = x.as_slice();
    let ys = y.as_slice();

    let xs_it = xs[ix..].iter().step_by(incx).take(n);
    let ys_it = ys[iy..].iter().step_by(incy).take(n);

    for (&xv, &yv) in xs_it.zip(ys_it) {
        acc += xv * yv;
    }

    acc
}
```

The only tricky part is learning the `portable-simd` syntax and tuning the `LANES` vector length. I have an Apple M4 Pro, whose NEON vector registers are 128-bit wide, i.e. 4 `f32`s per vector operation. This means SIMD can apply the same arithmetic to four `f32`s in parallel.

Specifically,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} * \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \\ x_4 y_4 \end{pmatrix},$$

applies the same instruction across four lanes at once. Based on this, setting `LANES = 4` would be reasonable. This would separate x and y into `chunks` of 4 `f32`s at a time, which is perfect for Apple M4’s 128-bit registers.

However, within the hot loop, there is still per-iteration overhead: loop control, bounds/tail handling, and moving chunks in and out of SIMD values. Increasing to `LANES = 32` batches more work per iteration, so the loop runs 8× fewer iterations than `LANES = 4`. This is because 32 `f32`s get processed per iteration, instead of just 4.

For example, let $\vec{x}$ and $\vec{y}$ have length 1024.

$$\begin{aligned} \text{if LANES} = 4 \; &\longrightarrow 256 \text{ chunks of } x, y \\ \text{if LANES} = 32 &\longrightarrow 32 \quad \text{chunks of } x, y \\ &\longrightarrow 8 \times \text{ less iterations} \end{aligned}$$

Despite vector registers only storing 4 `f32`s at a time, processing 8 registers (`LANES = 32`) at a time is more efficient than matching native register width, as I show below.

Optimized Benchmark

When vectors x and y contain 1024 elements, this routine runs in

$$\begin{aligned} \text{LANES} &= 4\text{:} \;\; 416\text{ns} \\ \text{LANES} &= 16\text{: } 166\text{ns} \\ \text{LANES} &= 32\text{: } 125\text{ns}. \end{aligned}$$

These are all extremely fast. But setting `LANES = 32` is fastest and **10.328× faster** than the naive scalar loop implementation.

Optimizing Vector Addition

The `axpy` routine performs

$$y \leftarrow \alpha x + y,$$

i.e. **A** lpha **X P** lus **Y**, and y gets overwritten with the solution. Hence, for single-precision `saxpy`, I use `VectorRef` for x and a mutable `VectorMut` for y containing `f32`s.

The procedure is similar to the dot product. However, the SIMD-optimized results are very different.

Naive Implementation

```rust
use crate::types::{VectorMut, VectorRef};

/// Updates [VectorMut] `y` by adding `alpha * x` [VectorRef]
#[inline]
pub fn saxpy (
    alpha : f32,
    x     : VectorRef<'_, f32>,
    mut y : VectorMut<'_, f32> // gets overwritten
) {
    let xn = x.n();
    let yn = y.n();

    if xn != yn {
        panic!("x y vector length must match!");
    }

    // no op
    if xn == 0 || alpha == 0.0 {
        return;
    }

    // fast path
    if let (Some(xs), Some(ys)) = (x.contiguous_slice(), y.contiguous_slice_mut())
{
        for (xk, yk) in xs.iter().zip(ys.iter_mut()) {
            *yk += alpha * *xk;
        }

        return;
    }

    let ix = x.offset();
    let iy = y.offset();
    let incx = x.stride();
    let incy = y.stride();

    let xs = x.as_slice();
    let ys = y.as_slice_mut();
    let xs_it = xs[ix..].iter().step_by(incx).take(xn);
    let ys_it = ys[iy..].iter_mut().step_by(incy).take(yn);

    for (&xv, yv) in xs_it.zip(ys_it) {
        *yv += alpha * xv;
    }
}
```

I hope this code is understandable just by reading through it. It is very clean and elegant by directly iterating through every x_k and y_k in x and y, and overwriting $y_k \leftarrow \alpha x_k + y_k$ in the process:

```rust
for (xk, yk) in xs.iter().zip(ys.iter_mut()) {
    *yk += alpha * *xk;
}
```

After going through the SIMD-optimized implementation, this example really shows how impressive LLVM is. I will show the naive benchmarks after the optimized section below.

Optimized Implementation

The SIMD-procedure is roughly the same as with the dot product `sdot` routine.

```rust
//! Level 1 [`?AXPY`](https://www.netlib.org/lapack/explore-html/d5/d4b/group__axpy.
html)
//! routine in single precision.
//!
//! \\[
//! y \leftarrow \alpha x + y
//! \\]
//!
//! # Author
//! Deval Deliwala


use std::simd::Simd;
use crate::debug_assert_n_eq;
use crate::types::{VectorRef, VectorMut};


/// Updates [VectorMut] `y` by adding `alpha` * `x` [VectorRef]
///
/// Arguments:
/// * `alpha`: [f32] - scalar multiplier for `x`
/// * `x`: [VectorRef] - struct over [f32]
/// * `y`: [VectorMut] - struct over [f32]
///
/// Returns:
/// Nothing. `y.data` is overwritten.
#[inline]
pub fn saxpy (
    alpha : f32,
    x     : VectorRef<'_, f32>,
    mut y : VectorMut<'_, f32>,
) {
    debug_assert_n_eq!(x, y);

    let n = x.n();
    let incx = x.stride();
    let incy = y.stride();

    if n == 0 || alpha == 0.0 {
        return;
    }

    // fast path
    if let (Some(xs), Some(ys)) = (x.contiguous_slice(), y.contiguous_slice_mut())
{
        const LANES: usize = 32;
        let a = Simd::<f32, LANES>::splat(alpha);

        let (xv, xr) = xs.as_chunks::<LANES>();
        let (yv, yr) = ys.as_chunks_mut::<LANES>();

        // chunks
        for (xc, yc) in xv.iter().zip(yv.iter_mut()) {
            let xvec = Simd::from_array(*xc);
            let yvec = Simd::from_array(*yc);

            let out  = a * xvec + yvec;

            *yc = out.to_array();
        }

        // scalar remainder tail
        for (xt, yt) in xr.iter().zip(yr.iter_mut()) {
            *yt += alpha * *xt;
        }

        return;
    }

    // slow path
    let ix = x.offset();
    let iy = y.offset();

    let xs = x.as_slice();
    let ys = y.as_slice_mut();

    let xs_it = xs[ix..].iter().step_by(incx).take(n);
    let ys_it = ys[iy..].iter_mut().step_by(incy).take(n);

    for (&xv, yv) in xs_it.zip(ys_it) {
        *yv += alpha * xv;
    }
}
```

The only difference is overwriting y’s `VectorMut` in the process, which is accomplished via

```
*yc = out.to_array();
```

in the SIMD fast path.

Benchmarks

Here are the benchmarks (again on Apple M4):

For vectors x and y of length 1024, the naive implementation takes 83 nanoseconds on average. The optimized implementation takes 83 nanoseconds on average.

The two routines run at the *exact same speed*. This is because LLVM recognizes the saxpy pattern and emits NEON vector multiply + add in the fast path:

```
*yt += alpha * *xt;
```

and automatically lowers it into NEON SIMD instructions when the data is contiguous. The naive implementation’s code is 40 lines less, much more readable, but is just as fast because of how well engineered LLVM is and the memory-bound nature of Level 1 BLAS.

Analyzing Assembly

The naive and SIMD implementations of `saxpy` run at the exact same speed. This is not accidental, and it is not because the SIMD version is ineffective. It is because LLVM lowers both implementations into the same class of NEON vectorized loops on Apple M4.

The important observation is not that the Rust code looks similar, but that the generated machine code has the same **structure** in the contiguous fast path. Once both implementations reach this point, performance is dominated by memory bandwidth, not by arithmetic.

What to look for in the assembly

For this comparison, only three instruction patterns matter:

- NEON vector loads and stores: `ldp q..`, `stp q..`
- NEON arithmetic on four `f32`s at once: `fmul.4s`, `fadd.4s`
- Scalar remainder loops: `ldr s..`, `fmul`, `fadd`, `str s..`

When these appear in a tight loop with no intervening calls or branches to panic paths, the loop is fully vectorized and bounds checks have been hoisted out.

Naive saxpy autovectorizes

The core of the naive implementation is:

```rust
for (xk, yk) in xs.iter().zip(ys.iter_mut()) {
    *yk += alpha * *xk;
}
```

LLVM recognizes this as a SAXPY pattern and emits a NEON loop in the contiguous fast path. The following block from this naive `saxpy` implementation is the hot loop:

```asm
LBB0_31:
    ldp q1, q2, [x12, #-32]
    ldp q3, q4, [x12], #64
    fmul.4s v1, v1, v0[0]
    fmul.4s v2, v2, v0[0]
    fmul.4s v3, v3, v0[0]
    fmul.4s v4, v4, v0[0]
    ldp q5, q6, [x13, #-32]
    ldp q7, q16, [x13]
    fadd.4s v1, v5, v1
    fadd.4s v2, v6, v2
    fadd.4s v3, v7, v3
    fadd.4s v4, v16, v4
    stp q1, q2, [x13, #-32]
    stp q3, q4, [x13], #64
    subs x14, x14, #16
    b.ne LBB0_31
```

This loop performs the exact saxpy operation in vector form. It is crucial to notice no bounds checks:

```
core::slice::index::slice_index_fail
```

occurring within this loop. This is because we wrote the `for` loop as:

```rust
for (xc, yc) in xv.iter().zip(yv.iter_mut()) {
  ...
}
```

directly iterating over chunks `xc, yc` in `xv, yv` so the compiler knows these chunks exist within `xv, yv`. If I instead wrote

```rust
for (idx1, idx2) in (0..n).zip(0..n) {
  let xc = xv[idx1];
  let yc = yv[idx2];
  ...
}
```

without explicitly defining `let n = xv.len()` beforehand, it would have been much more likely for bounds checks to appear and slow down the routine.

These are the opcodes that help read the assembly above:

- Vector loads from `x`:
  - `ldp q1, q2, ...` and `ldp q3, q4, ...` are **load-pair** instructions. Each `q` register is 128 bits, so loading `q1..q4` brings in multiple `f32`s from memory in one iteration.
- Multiply by the scalar `alpha`:
  - `fmul.4s v1, v1, v0[0]` means “multiply 4 lanes of 32-bit floats.” The `.4s` indicates 4 parallel `f32` lanes inside a 128-bit NEON vector. The `v0[0]` operand means the scalar `alpha` is taken from lane 0 and broadcast across all lanes.
- Add the corresponding `y` vectors:
  - `ldp q5, q6, ...` and `ldp q7, q16, ...` load the `y` data.
  - `fadd.4s v1, v5, v1` is the saxpy update: `v1 <- v5 + v1`. The same pattern repeats for `v2/v6`, `v3/v7`, and `v4/v16`.
- Store the updated `y` back to memory:
  - `stp q1, q2, ...` and `stp q3, q4, ...` are **store-pair** instructions. These write the updated NEON vectors back to the `y` buffer.
- Loop structure:
  - `subs x14, x14, #16` decrements the loop counter and sets flags.
  - `b.ne LBB0_31` branches back while the counter is nonzero.

The naive implementation also contains a scalar remainder loop for the final few elements:

```asm
LBB0_15:
    ldr s1, [x9], #4
    fmul s1, s0, s1
    ldr s2, [x8]
    fadd s1, s2, s1
    str s1, [x8], #4
    subs x10, x10, #1
    b.ne LBB0_15
```

This is the same update performed one element at a time:

- `ldr s1, [x9], #4` loads one 32-bit float and advances the pointer by 4 bytes.
- `fmul s1, s0, s1` multiplies by `alpha` (held in `s0`).
- `ldr s2, ...` and `fadd ...` add `y`.
- `str s1, ...` stores back to `y`.

SIMD saxpy lowers to the same NEON structure

The `portable-simd` implementation expresses vectorization explicitly in Rust, but LLVM lowers it into a NEON loop with the same shape.

The hot loop from the optimized `saxpy` implementation is:

```asm
LBB2_12:
    ldp q1, q2, [x16]
    ldp q3, q4, [x16, #32]
    ldp q5, q6, [x16, #64]
    ldp q7, q16, [x16, #96]
    ldp q22, q21, [x17]
    ldp q20, q19, [x17, #32]
    ldp q18, q17, [x17, #64]
    ldp q24, q23, [x17, #96]
    fmul.4s v1, v1, v0[0]
    fmul.4s v2, v2, v0[0]
    fmul.4s v3, v3, v0[0]
    fmul.4s v4, v4, v0[0]
    fmul.4s v5, v5, v0[0]
    fmul.4s v6, v6, v0[0]
    fmul.4s v7, v7, v0[0]
    fmul.4s v16, v16, v0[0]
    fadd.4s v1, v1, v22
    fadd.4s v2, v2, v21
    fadd.4s v3, v3, v20
    fadd.4s v4, v4, v19
    fadd.4s v5, v5, v18
    fadd.4s v6, v6, v17
    fadd.4s v7, v7, v24
    fadd.4s v16, v16, v23
    stp q1, q2, [x17]
    stp q3, q4, [x17, #32]
    stp q5, q6, [x17, #64]
    stp q7, q16, [x17, #96]
    add x16, x16, #128
    add x17, x17, #128
    subs x14, x14, #1
    b.ne LBB2_12
```

The same reading rules apply:

- the `ldp q..` lines at the top load NEON vectors from `x` and `y`,
- the `fmul.4s ... v0[0]` lines apply the scalar `alpha` to each vector,
- the `fadd.4s ...` lines accumulate into `y`,
- the `stp q..` lines store the results,
- and `b.ne` forms the loop backedge.

The difference is only that this version is unrolled more aggressively (more `q` registers are processed per iteration). The core operation is identical.

Why both implementations are equally fast

In SAXPY, each element requires:

- one load from `x`,
- one load from `y`,
- one store to `y`,
- one multiply and one add.

This is a very small amount of computation relative to the amount of memory traffic. Once LLVM has produced a NEON loop for the contiguous case, both implementations are limited by how fast data can be streamed through cache and memory, not by how the arithmetic is expressed in Rust. This is what *memory-bound* means!

As a result, the naive iterator-based implementation and the handwritten SIMD implementation converge to the same machine-level structure and achieve the same performance.

This is a concrete example of how strong LLVM’s autovectorization is for Level 1 BLAS routines: clean, idiomatic code can compile into the same NEON kernels as explicitly vectorized code when the access patterns are simple and contiguous.