

Level 1 BLAS

Deval Deliwalा

Table of Contents

- Level 1 BLAS
 - Optimizing the Dot Product
 - Naive implementation
 - Optimized implementation
 - Optimizing Vector Addition
 - Naive Implementation
 - Optimized Implementation
 - LLVM is very impressive
 - Analyzing Assembly
 - What to look for in the assembly
 - Naive saxy autopvectorizes
 - SIMD saxy lowers to the same NEON structure
 - Why both implementations are equally fast

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 natively 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:

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

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 $x = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$. It could be represented as follows:

- contiguous:
 - let $x = \text{vec}\{[0, 1]\};$
 - increment $\text{incx} = 1$
- not contiguous:
 - let $x = \text{vec}\{[0, 0, 1]\};$
 - increment $\text{incx} = 2$, 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 ($\text{inc} == 1$) and a slow path otherwise.

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

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.

As I'll show in the [Assembly](#) section at the end for SAXPY, these instructions compile down to SIMD instructions for multiplying and adding.

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:

```
/// 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 Deliwalा

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` are SIMD vectors.
/// 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::splat(0.0);
        let (xv, xt) = xs.as_chunks::<LANES>();
        let (yv, yt) = ys.as_chunks::<LANES>();
        acc += xv * yt;
        for (xk, yk) in xv.iter().zip(yt.iter()) {
            acc += xk * yk;
        }
        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_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 f32s per vector operation. This means SIMD can apply the same arithmetic to four f32s 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 f32s 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 8x fewer iterations than `LANES = 4`. This is because 32 f32s get processed per iteration, instead of just 4.

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

$$\text{if } \text{LANES} = 4 \rightarrow 256 \text{ chunks of } x, y$$

$$\text{if } \text{LANES} = 32 \rightarrow 32 \text{ chunks of } x, y$$

$$\rightarrow 8 \times \text{less iterations}$$

Despite vector registers only storing 4 f32s at a time, processing 8 registers (`LANES = 32`) at a time is more efficient than matching native register width.

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

$$\text{LANES} = 4: 416\text{ns}$$

$$\text{LANES} = 16: 166\text{ns}$$

$$\text{LANES} = 32: 125\text{ns}.$$

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

Optimizing Vector Addition

The `saxy` routine performs

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

A $\alpha \vec{x} + \vec{y}$ is \vec{y} , and y gets overwritten with the solution. Hence, we use a `VectorRef` for x and a mutable `VectorMut` for y .

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

Naive Implementation

```
use crate::types::VectorRef;
// Updates [VectorMut] `y` by adding `alpha * x` [VectorRef];
#[inline]
pub fn saxy (
    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()) {
        const LANES: usize = 32;
        let mut acc = Simd::f32::splat(0.0);
        let (xv, xt) = xs.as_chunks::<LANES>();
        let (yv, yt) = ys.as_chunks::<LANES>();
        acc += xv * yt;
        for (xk, yk) in xv.iter().zip(yt.iter()) {
            acc += xk * yk;
        }
        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_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
}
```

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.

```
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'll show the benchmarks at the end for SAXPY, these instructions compile down to SIMD instructions for multiplying and adding.

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.

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

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.

As I'll show in the [Assembly](#) section at the end for SAXPY, these instructions compile down to SIMD instructions for multiplying and adding.

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.

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

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.

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.

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

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.

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.

```
use crate::types::VectorRef;
// Computes the dot product of two [VectorRef]s.
#[inline]
pub
```