Lean Design of Optimized Level-1 BLAS

LAK Working Note #2

devald@stanford.edu

13 May 2026

Abstract

The level-1 BLAS consists of $O(n)$ vector-vector routines with low arithmetic intensity. In `lak`, these kernels fall into two implementation cases. Elementwise routines expose independent work through simple scalar loops over contiguous slices, so LLVM often automatically widens hot loops to use SIMD intrinsics. Reduction routines are different: their scalar form introduces a loop-carried accumulator, which hides the available parallelism behind serial recurrence. These routines need to be redesigned with independent partial sums and an explicit SIMD accumulator. This gives a nice level-1 design. I keep ordinary loops ordinary when the compiler can easily see parallelizable structure, and use manual SIMD only when the naive form of the routine would otherwise serialize the hot loop.

Level-1 is simple. This note does not introduce anything new or unique to `lak`. It's meant to make clear the reasoning behind common level-1 design choices, including the ones used in `lak`.

The compiler discussed is `rustc 1.94.0-nightly`. Other compilers may not have similar behavior.

1 Dependency Chains

The level-1 BLAS routines operate on vectors. Examples include scaling a vector, adding one vector to another, copying data, taking a dot product, and computing vector norms. These routines have $O(n)$ work, but they do not all give the same optimization problem to the compiler. The important distinction in this note is whether the "naive" scalar loop has a *loop-carried dependence*. A loop-carried dependence happens when one iteration of the loop needs a value produced by an earlier iteration. When no such dependence exists, the iterations can be treated as independent pieces of work. When this dependence does exist, the loop looks sequential unless the implementation rewrites it to expose more parallelism.

The `AXPY` routine computes

$$y_i \leftarrow \alpha x_i + y_i. \tag{1}$$

Each index i updates a different element of y. The calculation of y_7 does not depend on the result of the computation of y_6. This is the elementwise structure shown on the left of Figure 1. Once the compiler sees a clean loop over contiguous slices, the elementwise structure is explicit. On a modern target, LLVM turns that scalar loop into SIMD instructions automatically, even though the source code never even mentions SIMD. This will be covered more in sections 2 and 3.

The dot product has a different shape. It computes

$$x \cdot y = \sum_{i=0}^{n-1} x_i y_i. \tag{2}$$

A direct scalar implementation introduces a running sum:

```
pub fn dot<T>(
    x: VecRef<'_, T>,
    y: VecRef<'_, T>,
) -> T
where
    T: Default
        + Copy
        + Fma
        + Mul<Output=T>
        + Add<Output=T>,
{
    // ensures buffers have equal length
    assert_length_eq!(x, y);

    // access stored &[T] buffer;
    let xdata = x.as_slice();
    let ydata = y.as_slice();

    // running sum
    let mut sum = T::default();
    for (&xi, &yi) in xdata.iter().zip(ydata.iter()) {
        // dependency chain
        // sum += xi * yi
        sum = xi.fma(yi, sum);
    }

    sum
}
```

The inner loop doesn't compute an independent output for each index. Each iteration reads the current value of `sum` and produces the next value of `sum`. The value produced at iteration i is an input to iteration $i+1$. This is the dependency chain shown on the right side of Figure 1.

This does not mean the dot product is unvectorize-able. The products $x_i y_i$ are independent. What is serial in the naive loop is the single accumulated `sum` state. To use SIMD well, the implementation can change the *shape* of the computation: instead of one running sum, it uses several independent partial sums and combines them at the end, outside of the inner loop. This rewrite is the main subject of section 3.

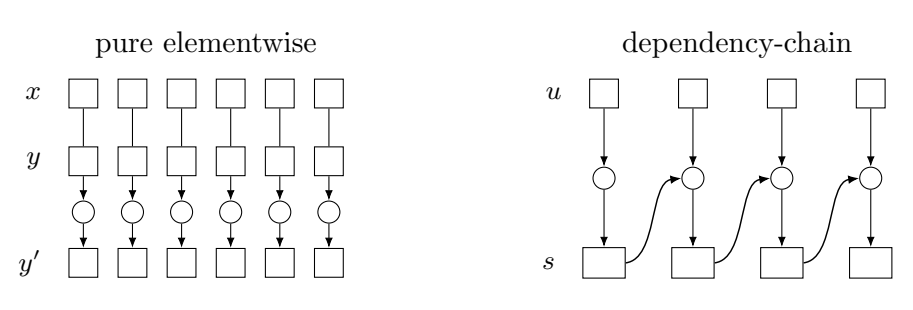

Figure 1: Classification of the level-1 kernels. Elementwise routines expose independent work directly. Dependency-chain routines carry state between iterations, hiding available parallelism behind scalar recurrence.

This gives the organizing principle for level-1. For elementwise kernels, expose a clean contiguous loop and let the compiler emit SIMD when it can. For reduction kernels, remove the single scalar recurrence from the hot loop using independent accumulators. I use `AXPY` and `DOT` as representative cases. `AXPY` shows when a scalar loop is enough, while `DOT` shows when it's not, and the routine itself has to be reshaped.

2 AXPY

$$y_i \leftarrow \alpha x_i + y_i \tag{3}$$

"Alpha x plus y," or `AXPY`, performs a scaled vector addition. It is a purely elementwise operation. In `lak`, the contiguous-only vector interface gives the compiler a simple loop over two slices, with one independent update per index. In this memory-bound setting, that scalar source is enough for LLVM to generate a competitive SIMD loop. The implementation can therefore remain minimal:

```
pub fn axpy<T>(
    a: T,
    x: VecRef<'_, T>,
    mut y: VecMut<'_, T>,
)
where
    T: Copy
    + AddAssign
    + Mul<Output=T>
    + Fma,
{
    assert_length_eq!(x, y);

    let x_slice = x.as_slice();
    let y_slice = y.as_slice_mut();

    // no simd needed, already fast
    for (&xi, yi) in x_slice.iter().zip(y_slice.iter_mut()) {
        // *yi += a * xi + *yi
        *yi = a.fma(xi, *yi);
    }
}
```

`*yi = a.fma(xi, *yi)` is the entire inner loop. There is no explicit vectorization here. The single-threaded, single-precision `SAXPY` benchmarks below test this implementation for vector lengths up to $2^{13} = 8192$.

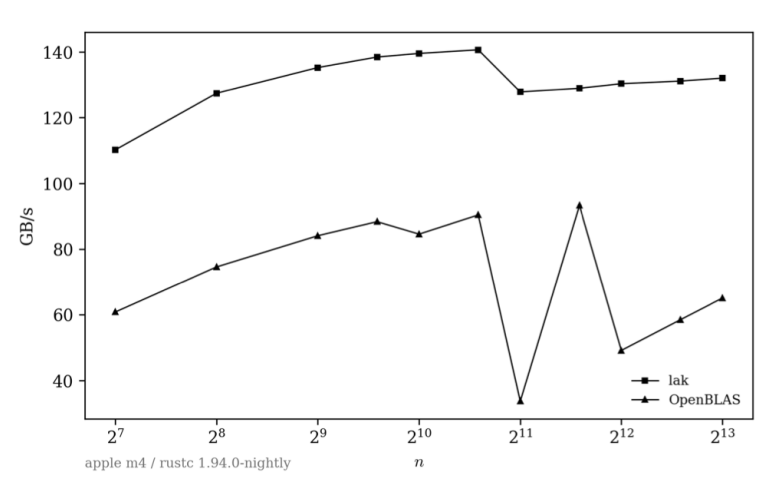

Figure 2: Single-threaded `SAXPY` memory bandwidth benchmark relative to OpenBLAS armv8.

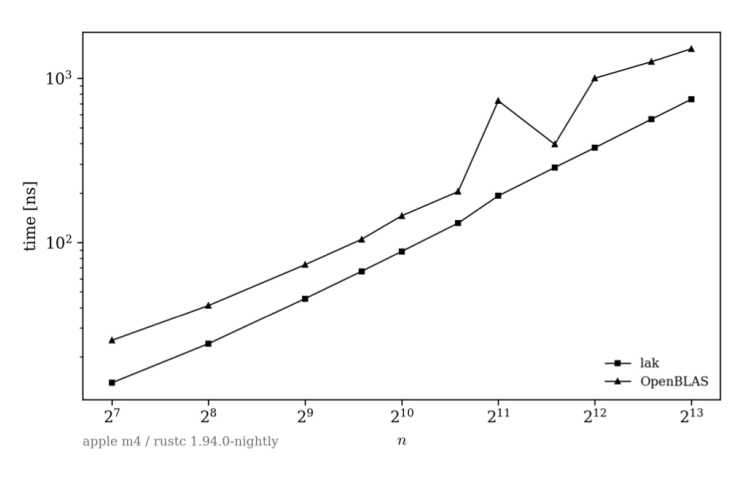

Figure 3: Single-threaded `SAXPY` time benchmark relative to OpenBLAS armv8. The y-axis is log-scaled.

In the bandwidth plot, higher is better. In the time plot, lower is better, and the y-axis is log-scaled. At $n = 8192$, `lak` finishes the routine in 754.7ns on average. OpenBLAS armv8 takes 1.541μs on average on the same single-threaded Apple M4 benchmark.

The compiler-generated assembly for single precision shows that the scalar source loop was automatically vectorized into a modestly unrolled SIMD loop:

```
LBB5_7:
    ldp q1, q2, [x9, #-32]
    ldp q3, q4, [x9], #64
    ldp q5, q6, [x10, #-32]
    ldp q7, q16, [x10]
    fmla.4s v5, v0, v1
    fmla.4s v6, v0, v2
    fmla.4s v7, v0, v3
    fmla.4s v16, v0, v4
    stp q5, q6, [x10, #-32]
    stp q7, q16, [x10], #64
    subs x11, x11, #16
    b.ne LBB5_7
    cmp x1, x8
    b.eq LBB5_15
    tst x1, #0xc
    b.eq LBB5_13
```

Each loop iteration loads 16 `f32` values from x into four SIMD registers, `q1` through `q4`. It also loads 16 `f32` values from y into `q5`, `q6`, `q7`, and `q16`. The core operation is:

```
fmla.4s v5,  v0, v1
fmla.4s v6,  v0, v2
fmla.4s v7,  v0, v3
fmla.4s v16, v0, v4
```

The vector `v0` is `alpha` broadcast across all four lanes. The updated 16 floats are then stored back into y. The loop is not scalar, but it is also not aggressively unrolled. For a memory-bound `AXPY`, this is fine. The compiler has widened the loop enough to make arithmetic throughput a secondary concern, while the limiting cost remains the load and store memory traffic.

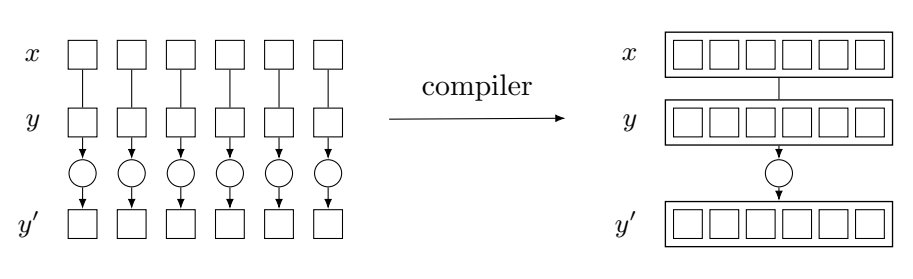

Figure 4: Independent elementwise iterations are automatically widened into a single SIMD operation.

3 DOT

The dot product

$$\vec{x} \cdot \vec{y}, \tag{4}$$

has a different implementation shape because its naive form carries a running sum.

```
pub fn dot<T>(
    x: VecRef<'_, T>,
    y: VecRef<'_, T>,
) -> T
where
    T: Default
        + Copy
        + Fma
        + Mul<Output=T>
        + Add<Output=T>,
{
    assert_length_eq!(x, y);

    let xdata = x.as_slice();
    let ydata = y.as_slice();

    // running sum
    let mut sum = T::default();
    for (&xi, &yi) in xdata.iter().zip(ydata.iter()) {
        // dependency chain
        // sum += xi * yi
        sum = xi.fma(yi, sum);
    }

    sum
}
```

The generated assembly for this routine shows the following inner loop:

```
LBB5_1:
        ldr s1, [x0], #4
        ldr s2, [x2], #4
        fmadd s0, s1, s2, s0
        subs x8, x8, #1
        b.ne LBB5_1
```

This loop uses `fmadd`, a scalar `f32` FMA. It is not the `fmla` instruction shown in the `AXPY` assembly above, which is a vector FMA. The loop performs one scalar FMA per iteration. The accumulator lives in `s0`, and every iteration reads and writes that same register. LLVM therefore preserves the scalar recurrence rather than widening the loop into SIMD instructions as it did for `AXPY`. This version takes about 6μs for vectors of length $n = 8192$, while OpenBLAS armv8 takes about 1μs on the same benchmark.

The fix is to replace the single long recurrence with several independent partial recurrences. Each SIMD lane carries its own partial sum through the hot loop, and those partial sums are reduced only after the main loop is finished. This requires manual SIMD.

```
pub(crate) const LANES: usize = 32;

pub fn dot<T>(
    x: VecRef<'_, T>,
    y: VecRef<'_, T>,
) -> T
where
    T: SimdElement
        + Copy
        + Default
        + AddAssign
        + Mul<Output=T>
        + Add<Output=T>
        + Fma,

    Simd<T, LANES>: SimdFloat<Scalar=T>
        + AddAssign
        + Fma,
{
    assert_length_eq!(x, y);

    let x_slice = x.as_slice();
    let y_slice = y.as_slice();

    // partitioning buffers into fixed LANES lengths
    // with a leftover tail
    let (x_chunks, x_tail) = x_slice.as_chunks::<LANES>();
    let (y_chunks, y_tail) = y_slice.as_chunks::<LANES>();

    // inner loop

    // independent accumulators in each LANE
    let mut accumulator = Simd::<T, LANES>::splat(T::default());
    for (&x_chunk, &y_chunk) in x_chunks.iter().zip(y_chunks.iter()) {
        let x_vec = Simd::from_array(x_chunk);
        let y_vec = Simd::from_array(y_chunk);

        accumulator = x_vec.fma(y_vec, accumulator);
    }

    // scalar tail leftover
    let mut sum = T::default();
    for (&xt, &yt) in x_tail.iter().zip(y_tail.iter()) {
        sum = xt.fma(yt, sum);
    }

    // final horizontal sum across all LANES
    // outside the inner loop
    accumulator.reduce_sum() + sum
}
```

The generics get a little ugly, but the idea is simple. A chunk size `LANES` is fixed at compile time. The vectors x and y are split into full chunks of length `LANES`, plus a tail of length less than `LANES`. In the main loop, each chunk is loaded into a SIMD vector, and the SIMD `fma` updates the accumulator vector. Each lane of that accumulator carries its own partial sum. I found `LANES=32` to be fastest for this benchmark.

After the SIMD loop, the leftover tail is handled with a scalar loop. Then the SIMD accumulator is horizontally reduced and added to the scalar tail sum to produce the final dot product $\vec{x} \cdot \vec{y}$. The hot loop has been changed from one scalar recurrence into several independent recurrences. The parallelism was already present in the dot product, but the naive version hid it behind a single running sum. This process is visualized in Figure 5. The generated assembly for the improved `DOT` inner loop is shown below:

```
LBB5_4:
    ldp q17, q16, [x9, #64]
    ldp q19, q18, [x9, #32]
    ldp q20, q21, [x9]
    ldp q23, q22, [x10, #32]
    ldp q24, q25, [x10]
    fmla.4s v7, v25, v21
    fmla.4s v6, v24, v20
    fmla.4s v5, v23, v19
    ldp q20, q19, [x10, #64]
    fmla.4s v4, v22, v18
    fmla.4s v3, v20, v17
    fmla.4s v2, v19, v16
    ldp q17, q16, [x9, #96]
    ldp q19, q18, [x10, #96]
    fmla.4s v1, v19, v17
    fmla.4s v0, v18, v16
    add x10, x10, #128
    add x9, x9, #128
    subs x8, x8, #1
    b.ne LBB5_4
```

This is the manually vectorized dot-product loop. Unlike the naive implementation, it keeps eight independent SIMD accumulators, `v0` through `v7`. Each accumulator carries a different partial sum. One loop iteration processes 32 elements of x and 32 elements of y: eight SIMD registers, with four `f32` lanes per register. The loop therefore performs eight `fmla.4s` instructions per iteration, instead of one scalar `fmadd` on one pair of elements.

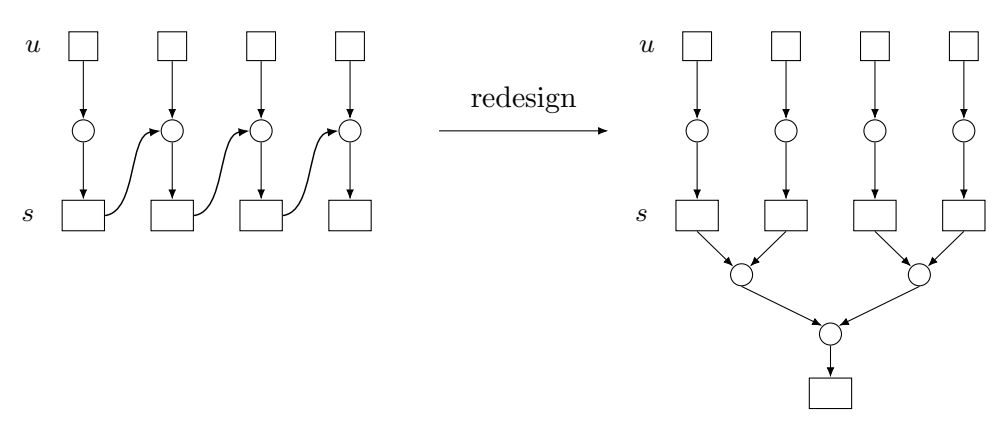

Figure 5: A running accumulator creates a loop-carried dependency. Splitting the work across independent accumulators exposes parallelism inside the loop; the partial sums are reduced afterward.

The important point is not that the improved loop uses manual SIMD. It's that it changes the dependency structure. There are still dependencies, because each accumulator is carried across loop iterations. But there are now eight independent vector accumulation chains instead of one scalar running sum. That gives the CPU more independent work to schedule while FMA results are still waiting on latency.

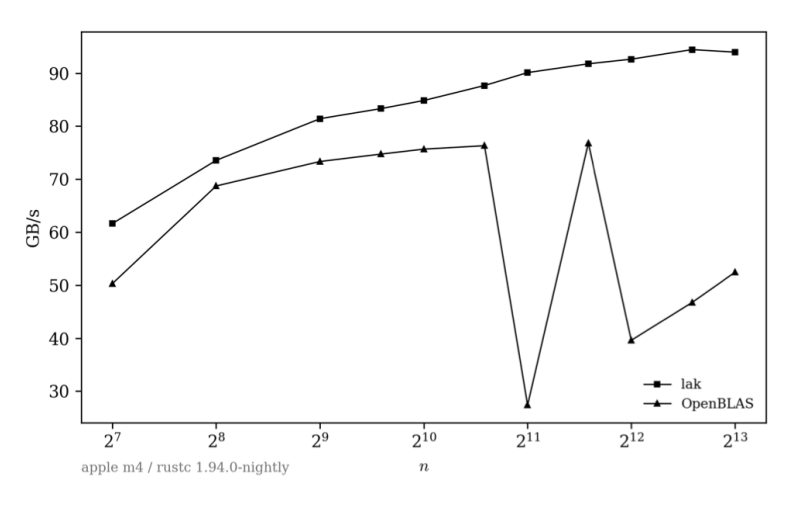

Figure 6: Single-threaded `SDOT` memory-bandwidth benchmark relative to OpenBLAS armv8.

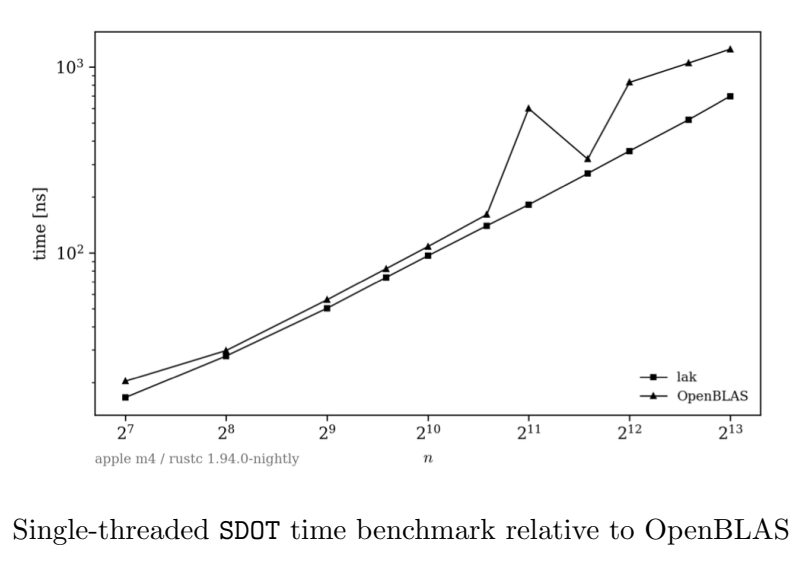

Figure 7: Single-threaded `SDOT` time benchmark relative to OpenBLAS armv8.

The level-1 BLAS does not need every routine to be written as a hand-vectorized SIMD kernel. For elementwise routines like `AXPY`, the scalar loop already exposes independent work, and LLVM generates the SIMD loop on its own. For dependency-chain routines like `DOT`, the scalar loop hides the available parallelism behind a single recurrence. These routines need a different implementation shape: independent partial sums in the hot loop, and one final reduction. This is the main design principle used in `lak`'s level-1 kernels. Because level-1 is also memory-bound, the benchmark figures above have the same shape for double precision `DAXPY` and `DDOT`, too. For much, much larger vectors where DRAM traffic becomes important, `lak` does dip *slightly* below OpenBLAS armv8.