Designing the Triangular Matrix-Vector Multiply

LAK Working Note #4

devald@stanford.edu

24 May 2026

Abstract

The triangular matrix-vector multiply, `TRMV`, breaks the clean SIMD structure that appeared in `GEMV`. There is no separate output vector. The routine performs $x \leftarrow Ax$, overwriting x as it moves through the matrix. This in-place constraint, together with the triangular structure of A, makes traversal order part of the algorithm itself: entries of x must only be overwritten after their final use. For the no-transpose cases, this leads to a new fused level-2 microkernel that handles the rectangular region away from the diagonal. The transpose cases are simpler. Each entry is naturally formed by a dot product over a triangular column segment, so the level-1 `DOT` kernel is all that is needed, just as in transpose `GEMV`.

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

1 Introduction

Naturally, at first, I asked what the point of this routine was. `GEMV` is already quite fast and performs a *general* matrix-vector multiply. Even if `GEMV` wastes work by reading the zero half of a triangular matrix, it was not obvious to me that `TRMV` deserved its own implementation yet. If the routine was just a slightly cheaper `GEMV`, I would rather move on and become prepared for level-3 `GEMM`. However, there is a crucial difference. `GEMV` performs

$$y \leftarrow \alpha Ax + \beta y, \tag{1}$$

placing the result of αAx into an output vector y. `TRMV`, on the other hand, performs

$$x \leftarrow Ax, \tag{2}$$

overwriting x as the routine progresses. So if someone really wanted to perform a `TRMV`, even if content with sacrificing some speed, they could not simply use `GEMV` in-place. They would need to compute into a temporary output vector and then perform a level-1 `COPY` back into x. `TRMV` is designed this way for a reason. I don't know what that reason is yet, but `lak` should have it.

2 Lower No-Transpose `TRMV`

The lower no-transpose triangular matrix-vector multiply computes

$$x \leftarrow Lx, \tag{3}$$

where L is a lower triangular matrix. This looks close to the no-transpose `GEMV` routine from the previous note. The difference is that only the diagonal and subdiagonal entries of L are logical. In column j, the useful segment begins at L_{jj} and runs downward. Consequently, column j only contributes to the suffix $x_{j:n}$. There is no separate y to accumulate into. The same entries of x that are being overwritten are also the entries needed by later column updates. Hence, we have to update x without wrecking future updates that still need the *original* values of x.

2.1 Right to Left

If there were a separate output vector, the implementation could walk left-to-right and accumulate each scaled column into the output. But in `TRMV`, writing into x_j too early destroys the value that still has to be used as a scalar weight for future column updates. This is resolved by sweeping across L from right to left instead. When we reach column j, all entries to the right have already been handled. To see this explicitly, note what a lower no-transpose `TRMV` does:

$$\begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} L_{00}x_0 \\ L_{10}x_0 + L_{11}x_1 \end{pmatrix}. \tag{4}$$

Hence, x is updated as

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \leftarrow \begin{pmatrix} L_{00}x_0 \\ L_{10}x_0 + L_{11}x_1 \end{pmatrix}. \tag{5}$$

If we walked left-to-right normally, then x_0 would first be replaced with $L_{00}x_0$. Then that *modified* value of x_0 would be used when calculating $L_{10}x_0 + L_{11}x_1$, or in reality, $L_{10}(L_{00}x_0) + L_{11}x_1$, destroying the operation. Walking right-to-left allows us to first modify x_1 as $L_{10}x_0 + L_{11}x_1$, with the correct x_0 and x_1, and then update x_0 afterwards as $L_{00}x_0$. At that point, x_1 is not used again. Once the sweep direction is fixed, each column can be handled almost exactly like an `AXPY`; the only special case is the diagonal element. For column j, we save x_j. Then we use x_j as the scalar in an `AXPY` on the column suffix $L_{j+1:n,\,j}$ and the vector suffix $x_{j+1:n}$:

$$x_{j+1..n} \mathrel{+}= x_j L_{j+1..n,\,j}. \tag{6}$$

Again, this is just an `AXPY`. After x_j has done its job, we can finally overwrite it with the diagonal update

$$x_j \leftarrow L_{jj}x_j. \tag{7}$$

The procedure is more obvious from the dependency order than from the code itself. It is visualized in Figure 1.

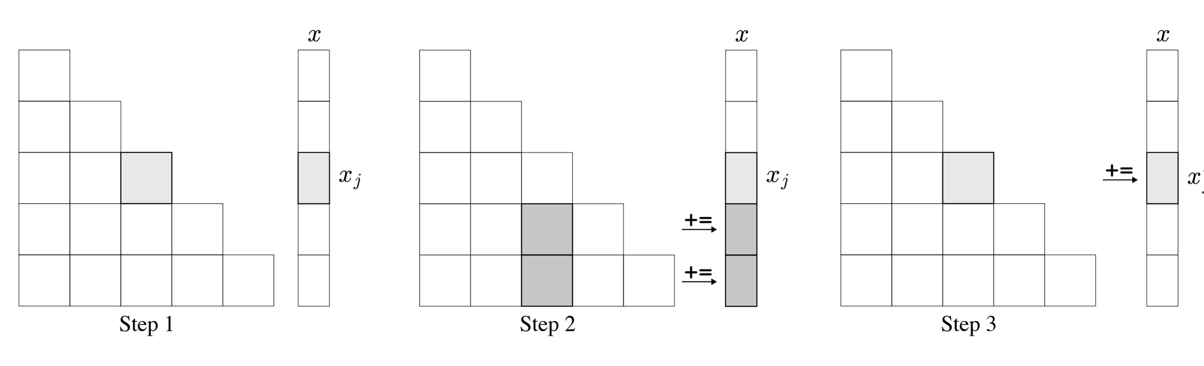

Figure 1: Naive lower no-transpose `TRMV`. Columns are processed from right to left. For each column j, the original value of x_j is saved, used as the scalar in an `AXPY` into the already-safe suffix of x, and only then overwritten by the diagonal update.

This gives a correct implementation. It is also close to the design goal of `lak`: the inner column update is just a level-1 `AXPY`. However, the same problem from no-transpose `GEMV` appears again. If every column update is performed separately, then the same suffix of x is repeatedly loaded and stored, creating a ~2× slowdown relative to OpenBLAS.

2.2 Fused Re-design

The useful observation is that the triangular structure is local to the diagonal. Once a small column panel is selected, everything below its diagonal block is just a rectangular update into a suffix of x. This is the same fusion idea used for no-transpose `GEMV`: the destination slice is loaded once, updated by several columns while live in registers, and stored once. The difference is where the kernel is allowed to run. In `GEMV`, the fused kernel can run over full column panels because every entry of A is logical. In lower no-transpose `TRMV`, the fused kernel can only be applied to the full rectangular block below the active diagonal region. This is visualized in Figure 2.

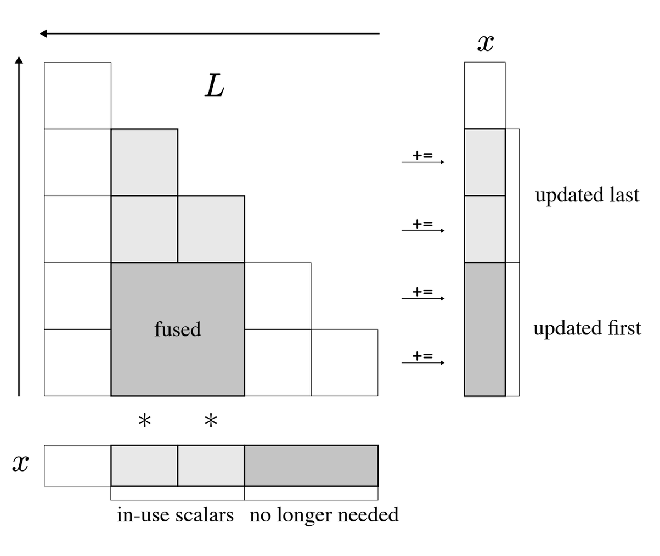

Figure 2: Lower no-transpose `TRMV` block update. The rectangular region below the active diagonal block is updated first with a fused four-column `AXPY`; only afterwards is the diagonal block handled from right to left. This order preserves the original scalar values of x until their final use.

For a four-column block $j..j+4$, the entries below the diagonal block, $x_{j+4..n}$, are safe to update first: none of them will be needed as scalar weights for the current block. The kernel performs a fused four-column update into this suffix, keeping the destination slice live in registers and storing it once. Only after this rectangular update is complete do we return to the small triangular block and update $x_j, \dots, x_{j+3}$ from right to left. This preserves correctness and recoveres the fused operations needed to reduce loads and stores of elements of x. The upper no-transpose routine is the mirror image. The rectangular fused panel lies above the diagonal block, and correctness now requires a left-to-right sweep. Still, the fused `AXPY` occurs before the triangular block is handled.

The fused kernel, "`FTRMV`", is shown in the appendix. It is very similar to `FAXPY` from the previous note, but there are two differences. First, because the input and output vector are the same object, the scalar weights from x are copied out before the mutable suffix is updated. The kernel then splats those saved scalars into SIMD registers and applies four FMAs to the destination slice. Second, because the matrix is triangular, the kernel does not accept full column panels like `FAXPY`. Instead, it accepts four column slices below the active diagonal block, four saved scalar values from x, and the mutable suffix of x to be updated.

2.3 Benchmarks

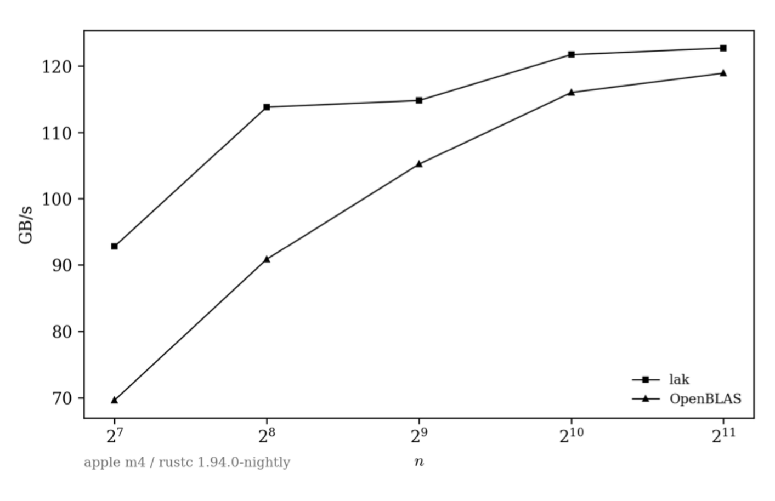

Figure 3: Single-threaded lower no-transpose `STRMV_LN` bandwidth benchmark relative to OpenBLAS armv8.

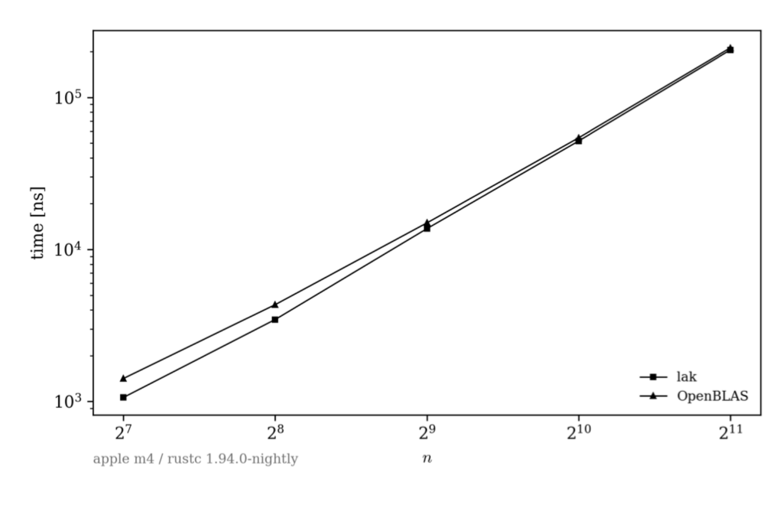

Figure 4: Single-threaded lower no-transpose `STRMV_LN` time benchmark relative to OpenBLAS armv8.

For smaller matrices, `lak` has a fairly large lead over OpenBLAS. For a matrix of size 256×256, `lak` `STRMV_LN` sustains an average effective memory bandwidth of 113.8 GB/s, whereas OpenBLAS sustains 90.87 GB/s. Consequently, `lak` finishes the routine in 3.458 μs, corresponding to $\sim$18.94 GFLOP/s, whereas OpenBLAS takes 4.332 μs, corresponding to $\sim$15.12 GFLOP/s. For larger matrix sizes, OpenBLAS closes the gap.

3 Transpose `TRMV`

The transpose routines have a different shape. For lower triangular L, the transpose multiply computes

$$x \leftarrow L^T x. \tag{8}$$

The j-th entry of x is formed using the triangular column segment beginning at the diagonal:

$$x_j \leftarrow \operatorname{dot}(L_{j..n,\,j}, x_{j..n}). \tag{9}$$

The routine is naturally expressed as a loop over contiguous dot products. Unlike lower no-transpose `TRMV`, it does not repeatedly update a long suffix of x over many different columns. Each step produces one scalar entry. This is the same structural simplification that made transpose `GEMV` clean. In the no-transpose case, fusion matters because many column updates repeatedly touch the same output vector. In the transpose case, the output traffic is already small. Each entry of x is produced by one dot product, so there is no comparable vector of repeated stores to remove.

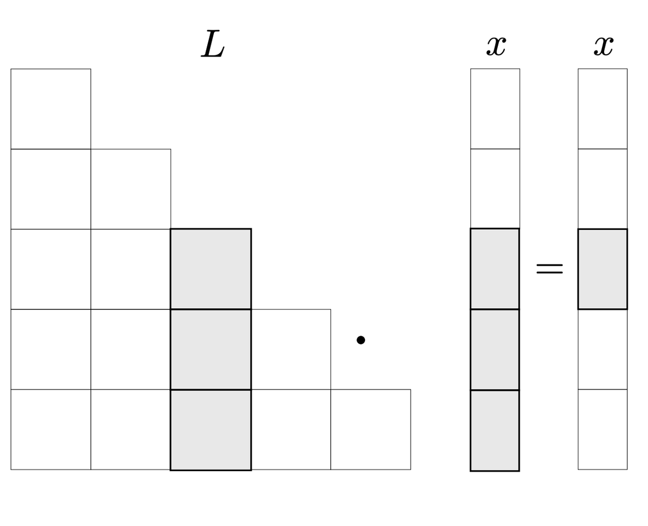

Figure 5: Lower transpose `TRMV`. Moving left to right, x_j is updated as the dot product of the column suffix of L beginning at the diagonal with the matching suffix of x.

The performance is therefore mostly the performance of the level-1 `DOT` kernel applied to contiguous triangular column segments. A lower transpose implementation can be written elegantly as

```rust
pub(crate) fn trmv_lt<T>(
    a: MatRef<'_, T>,
    mut x: VecMut<'_, T>,
)
where
    T: Copy
        + AddAssign
        + Mul<Output = T>
        + Fma
        + SimdElement,

    Simd<T, N_ROWS_PER_CHUNK>: SimdFloat<Scalar=T>
        + AddAssign
        + Mul<Output = Simd<T, N_ROWS_PER_CHUNK>>
        + Fma,
{
    let (n_rows, n_cols) = a.dimension();
    assert_eq!(n_rows, n_cols, "matrix a must be square nxn to be triangular");
    assert_length_eq_n_cols!(a, x);

    let n = n_rows;
    let a_slice = a.as_slice();
    let x_slice = x.as_slice_mut();

    for j in 0..n {
        let a_col = VecRef::new(&a_slice[j * n + j..(j + 1) * n]);
        let x_col = VecRef::new(&x_slice[j..n]);

        // only a level-1 dot!
        x_slice[j] = dot(a_col, x_col);
    }
}
```

and maintain performance competitive with OpenBLAS. The upper transpose routine is analogous, but the direction changes to preserve correctness. For upper transpose, the routine walks right-to-left and dots each column segment from the top of the column through the diagonal. In both cases, no new fused level-2 microkernel is needed.

3.1 Benchmarks

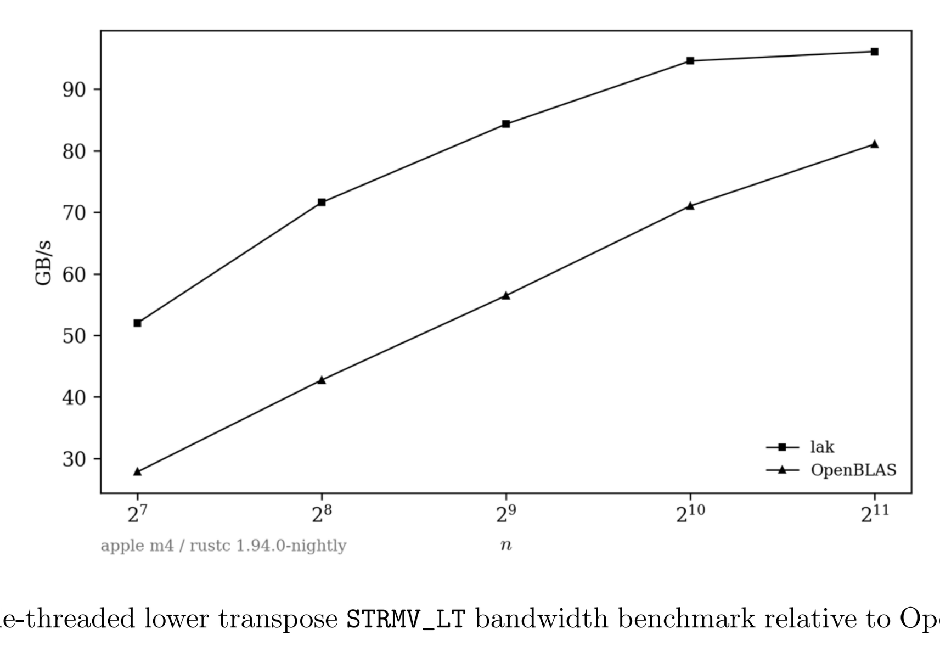

Figure 6: Single-threaded lower transpose `STRMV_LT` bandwidth benchmark relative to OpenBLAS armv8.

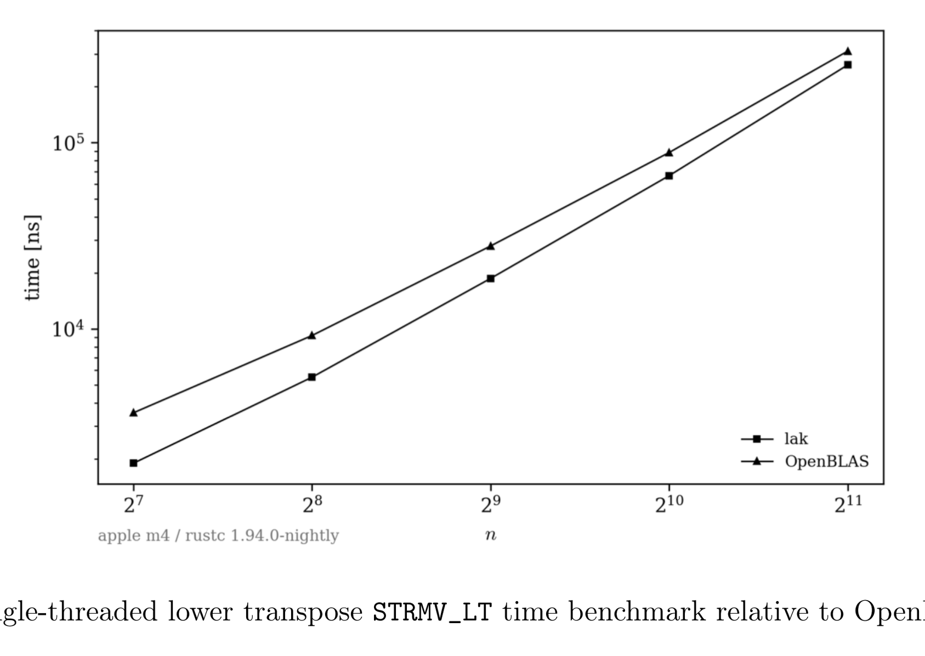

Figure 7: Single-threaded lower transpose `STRMV_LT` time benchmark relative to OpenBLAS armv8.

Here, `lak` again holds a fairly large lead over OpenBLAS for smaller matrices. For a matrix size of 256×256, `lak` sustains 71.59 GB/s and finishes in 5.499 μs on average. OpenBLAS sustains 42.75 GB/s and finishes in 9.208 μs on average, roughly 1.7× slower. The same gap is present in the upper-triangular routines, for both no-transpose and transpose variants. This gap similarly diminishes as matrices get larger.

The important split is therefore not lower versus upper, but no-transpose versus transpose. The no-transpose routines need a small fused level-2 kernel to control repeated updates into x, while the transpose routines fall back cleanly to contiguous level-1 dot products. And again, like in the previous notes, because `TRMV` is memory-bound, benchmarks have the same shape for double precision, too.

4 Appendix

The fused rectangular kernel used by no-transpose `TRMV` is shown below.

```rust
// ftrmv.rs

use std::ops::{Add, AddAssign, Mul};
use std::simd::{Simd, SimdElement};
use std::simd::num::SimdFloat;

use crate::traits::Fma;

pub(crate) const N_ROWS_PER_CHUNK: usize = 16;

/// a kernel for handling full rectangular
/// column panels in trmv no-transpose routines
///
/// "fuses" four column ops together to reduce
/// loads and stores of x
#[allow(clippy::too_many_arguments)]
pub(crate) fn ftrmv_n<T>(
    col0: &[T],
    col1: &[T],
    col2: &[T],
    col3: &[T],
    x0: T,
    x1: T,
    x2: T,
    x3: T,
    x: &mut [T],
)
where
    T: SimdElement
        + AddAssign<T>
        + Mul<Output=T>
        + Add<Output=T>
        + Copy
        + Fma,

    Simd<T, N_ROWS_PER_CHUNK>: SimdFloat<Scalar=T>
        + AddAssign
        + Mul<Output = Simd<T, N_ROWS_PER_CHUNK>>
        + Fma,
{
    debug_assert!(
        col0.len() == col1.len() &&
        col1.len() == col2.len() &&
        col2.len() == col3.len(),
        "column vector lengths must be equal"
    );

    let n_rows = col1.len();
    let n_row_chunks = n_rows / N_ROWS_PER_CHUNK;

    let x0v = Simd::<T, N_ROWS_PER_CHUNK>::splat(x0);
    let x1v = Simd::<T, N_ROWS_PER_CHUNK>::splat(x1);
    let x2v = Simd::<T, N_ROWS_PER_CHUNK>::splat(x2);
    let x3v = Simd::<T, N_ROWS_PER_CHUNK>::splat(x3);

    for row_chunk in 0..n_row_chunks {
        let i = row_chunk * N_ROWS_PER_CHUNK;

        let c1 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
            &col0[i..i + N_ROWS_PER_CHUNK],
        );
        let c2 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
            &col1[i..i + N_ROWS_PER_CHUNK],
        );
        let c3 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
            &col2[i..i + N_ROWS_PER_CHUNK],
        );
        let c4 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
            &col3[i..i + N_ROWS_PER_CHUNK],
        );

        let xchunk = &mut x[i..i + N_ROWS_PER_CHUNK];
        let mut xv = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(xchunk);

        xv = c1.fma(x0v, xv);
        xv = c2.fma(x1v, xv);
        xv = c3.fma(x2v, xv);
        xv = c4.fma(x3v, xv);

        xv.copy_to_slice(xchunk);
    }

    let tail_beg = n_row_chunks * N_ROWS_PER_CHUNK;
    for i in tail_beg..n_rows {
        x[i] += {
            col0[i] * x0 +
            col1[i] * x1 +
            col2[i] * x2 +
            col3[i] * x3
        };
    }
}
```