A Generic Implementation of General Matrix-Vector Multiplication

LAK Working Note #3

devald@stanford.edu

19 May 2026

Abstract

The general matrix-vector multiply, GEMV, is in Level-2 BLAS. In `lak`, it's where matrix structure first tests the design goal of the library: keeping the code safe, generic, and elegant while still maintaining competitive performance. The no-transpose case can't be written as a naive sequence of AXPY calls without repeatedly loading and storing the same output entries of y. `lak` uses a "fused" AXPY kernel, following the same idea used in ulmBLAS, and applies it over contiguous column panels made available by the library's contiguous layout restriction. The transpose case is deliberately simpler: since each column of A produces one entry of y, there is no output-vector traffic to remove, and just using the level-1 DOT kernel is enough. These two paths keep GEMV's implementation elegant while still matching OpenBLAS-level performance for both `f32` and `f64` with a single generic implementation.

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

1 The Routine

The general matrix-vector multiply, GEMV, computes

$$y \leftarrow \alpha\,\mathrm{op}(A)x + \beta y, \tag{1}$$

where α and β are scalars, and $\mathrm{op}(A)$ denotes either A or A^T. This gives two variants: the no-transpose case, which forms Ax, and the transpose case, which forms A^Tx. The stored matrix buffer is the same in both cases. The variant only changes how the kernel walks through that buffer. It either accumulates full columns of A into y, or takes dot products between rows of A and the input vector. The goal of `lak` level-2 is to express these matrix-vector routines in terms of the optimized contiguous level-1 kernels whenever possible without sacrificing much performance. This preserves the work already done at level-1 while avoiding duplicated low-level code.

2 No-transpose GEMV

The no-transpose variant computes

$$y \leftarrow \alpha Ax + \beta y. \tag{2}$$

2.1 Naive

For a column-major matrix, the product αAx has the useful form:

$$\alpha \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \alpha x_1 \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix} + \alpha x_2 \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{pmatrix} + \cdots + \alpha x_n \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{nn} \end{pmatrix} \tag{3}$$

$$= \alpha x_1 A_{:,1} + \alpha x_2 A_{:,2} + \cdots + \alpha x_n A_{:,n}.$$

No-transpose GEMV is a weighted accumulation over the columns of A. Each already-contiguous column $A_{:,j}$ contributes to the output with weight αx_j. The naive no-transpose implementation can be written as a loop over columns, where each column update is just a level-1 AXPY:

$$y \leftarrow \alpha x + y. \tag{4}$$

With $x \equiv A_{:,j}$ and $\alpha \equiv \alpha x_j$, each step performs

$$y \leftarrow \alpha x_j A_{:,j} + y. \tag{5}$$

The full routine is then:

```
pub fn gemv_n<T>(
    alpha: T,
    beta: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    mut y: VecMut<'_, T>,
)
where
    T: Copy
        + Clone
        + AddAssign
        + Mul<Output=T>,
{
    // verifies consistent dimensions
    assert_length_eq_n_cols!(a, x);
    assert_length_eq_n_rows!(a, y);

    let x_slice = x.as_slice();

    // y <- beta y; level-1
    scal(beta, y.reborrow());

    // iterate over columns
    // y <- alpha x_j A_{:,j} + y
    for (j, &xj) in x_slice.iter().enumerate() {
        let aj = a.col(j);

        axpy(alpha * xj, aj, y.reborrow());
    }
}
```

Despite its simplicity, this is a reasonable baseline. The inner loop is not a fresh matrix-vector kernel written from scratch; it is the optimized level-1 AXPY kernel, reused once per column. This implementation is roughly 2× slower than OpenBLAS, but it gets there with almost no new code. Even the inner AXPY kernel uses no SIMD, as discussed in the previous note!

The important point is that the arithmetic is not the problem. GEMV is still fundamentally memory-bound. Each column update performs useful work over contiguous data. The problem is that y is forced through the memory hierarchy once per column. For large matrices, this means the implementation is spending much of its time *repeatedly* reading and rewriting the accumulator rather than discovering new data from A.

2.2 Optimized

The ∼2× slowdown in the naive version comes primarily from the repeated reading and overwriting of y for every column of A in the AXPY loop. The key thing to observe is, from Equation 3, is that the final y is the result of *many* accumulated weighted column vectors of A. This means we can reduce the amount of loads and stores of y by "fusing" the AXPY to operate on *many* column vectors of A at a time, shown in Figure 1. I learned this from working through ulmBLAS, and reading through this paper. The naive loop fixes one column of A and sweeps all of y; the fused loop fixes a small block of rows of y and applies several columns of A before letting that block leave registers.

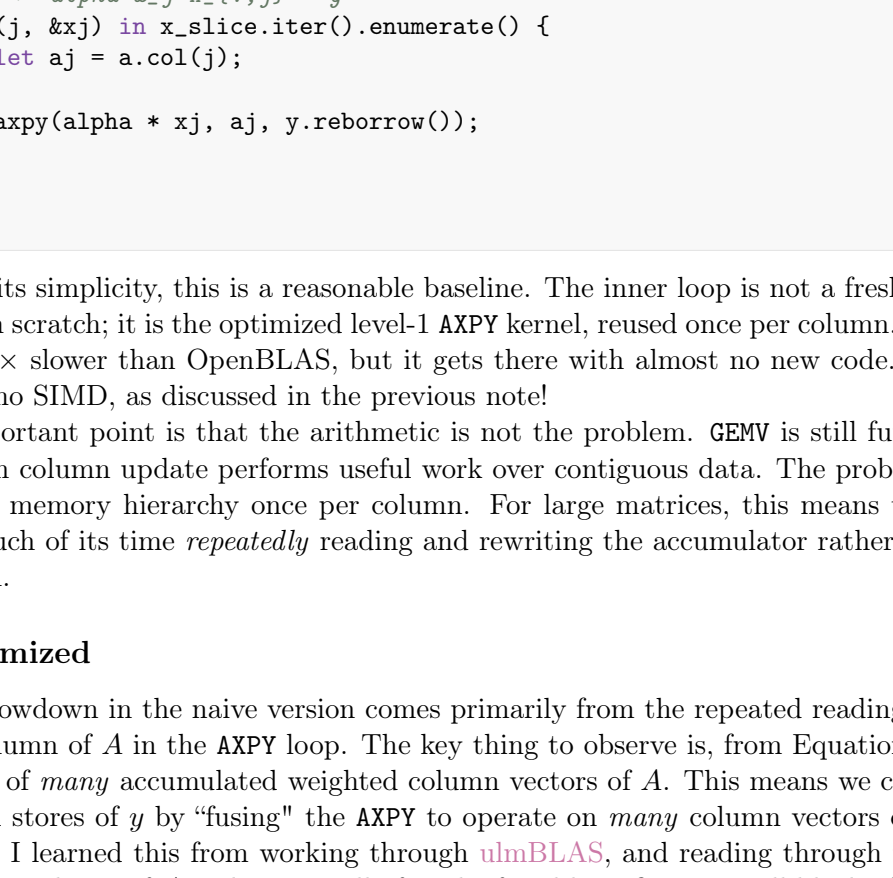

Figure 1: `lak` no-transpose GEMV implementation. The active four-column panel of A is multiplied by the matching entries of x, and the resulting row chunk is accumulated into y. The zoomed side shows the fused AXPY, "FAXPY", update $y_{i..i+64} \leftarrow y_{i..i+64} + A_{i..i+64,\,j..j+4}\,x_{j..j+4}$.

For a contiguous-only library, the "FAXPY" implementation remains fairly elegant too.

Algorithm 1 FAXPY panel kernel for no-transpose GEMV

Require: Matrix $A \in \mathbb{F}^{m\times n}$ stored column-major, vectors $x \in \mathbb{F}^n$ and $y \in \mathbb{F}^m$, scalar $\alpha \in \mathbb{F}$
Require: Row block size m_r, column panel size n_r
Ensure: $y \leftarrow y + \alpha Ax$

- $M \leftarrow \lfloor m/m_r \rfloor m_r$ ▷ *largest SIMD-covered row prefix*
- $N \leftarrow \lfloor n/n_r \rfloor n_r$ ▷ *largest full column-panel prefix*
- **for** $j = 0, n_r, 2n_r, \ldots, N - n_r$ **do** ▷ *$n_r = 4$ here*
  - $\chi_0 \leftarrow \alpha x_j$
  - $\chi_1 \leftarrow \alpha x_{j+1}$
  - $\chi_2 \leftarrow \alpha x_{j+2}$
  - $\chi_3 \leftarrow \alpha x_{j+3}$
  - **for** $i = 0, m_r, 2m_r, \ldots, M - m_r$ **do**
    - $\mathbf{y}_i \leftarrow y_{i:i+m_r}$
    - $\mathbf{a}_0 \leftarrow A_{i..i+m_r,\,j}$
    - $\mathbf{a}_1 \leftarrow A_{i..i+m_r,\,j+1}$
    - $\mathbf{a}_2 \leftarrow A_{i..i+m_r,\,j+2}$
    - $\mathbf{a}_3 \leftarrow A_{i..i+m_r,\,j+3}$
    - $\mathbf{y}_i \leftarrow \mathrm{fma}(\chi_0, \mathbf{a}_0, \mathbf{y}_i)$
    - $\mathbf{y}_i \leftarrow \mathrm{fma}(\chi_1, \mathbf{a}_1, \mathbf{y}_i)$
    - $\mathbf{y}_i \leftarrow \mathrm{fma}(\chi_2, \mathbf{a}_2, \mathbf{y}_i)$
    - $\mathbf{y}_i \leftarrow \mathrm{fma}(\chi_3, \mathbf{a}_3, \mathbf{y}_i)$
    - $y_{i:i+m_r} \leftarrow \mathbf{y}_i$
  - **end for**
- **end for**
- ▷ *Handle leftover columns over the SIMD-covered row prefix*
- **for** $j = N, N+1, \ldots, n-1$ **do**
  - $y_{0..M} \leftarrow y_{0..M} + \alpha x_j A_{0..M,\,j}$
- **end for**
- ▷ *Handle leftover rows for all columns*
- **if** $M < m$ **then**
  - **for** $j = 0, 1, \ldots, n-1$ **do**
    - $y_{M..m} \leftarrow y_{M..m} + \alpha x_j A_{M..m,\,j}$
  - **end for**
- **end if**

The fused kernel groups the columns of A into narrow column panels of width n_r. For each panel, it forms

$$y \leftarrow y + \alpha x_j A_{:,j} + \alpha x_{j+1} A_{:,j+1} + \cdots + \alpha x_{j+n_r-1} A_{:,j+n_r-1}.$$

This is the same update as applying n_r separate AXPY operations, but with one important difference: the corresponding block of y is loaded once, updated by all n_r columns while it is still live in registers, and then written back once. That is the point of the fusion. A sequence of ordinary AXPY calls repeatedly streams through y, so each column contribution reloads and rewrites the same output vector. The fused kernel amortizes that traffic across n_r columns. Ignoring tails, the load/store traffic to y is reduced by roughly a factor of n_r. With `nr = 4` for example, this changes four separate passes over the same part of y into one pass. The kernel still streams the same four columns of A, but it pays the read-modify-write cost for y once instead of four times. This matters because y is the only vector that is both read and written. Unlike the columns of A, which are only streamed as inputs, stores to y require cache ownership and eventually have to be written back through the cache hierarchy.

Within each column panel, the kernel then partitions the rows into chunks of length m_r. For example, with $m_r = 64$, each active column is viewed as

$$A_{0:64,j}, \quad A_{64:128,j}, \quad A_{128:192,j}, \quad \ldots$$

and the same row range is taken from each of the n_r columns in the panel. At the start of the panel, the corresponding scalar weights

$$\alpha x_j,\ \alpha x_{j+1},\ \ldots,\ \alpha x_{j+n_r-1}$$

are broadcast once into SIMD registers. Those broadcast values are kept live and reused across every m_r-row chunk in the panel. Thus each row chunk only has to load the active slices of A and the current slice of y; the x weights for the panel are already sitting in registers. Each row chunk then performs a small register accumulation into a local vector holding $y_{i:i+m_r}$:

$$y_{i:i+m_r} \leftarrow y_{i:i+m_r} + \alpha x_j A_{i:i+m_r,j} + \cdots + \alpha x_{j+n_r-1} A_{i:i+m_r,j+n_r-1}.$$

The leftover cases are deliberately simple. Any remaining columns that do not fill a complete n_r-wide panel are handled by ordinary AXPY over the SIMD-covered row prefix. Any remaining rows that do not fill a complete m_r-wide row chunk are then handled by ordinary AXPY over the row tail. The main path stays specialized for the regular blocked case, while the edge cases are delegated to the simpler level-1 AXPY routine. The final optimized GEMV no-transpose implementation is just

```
pub(crate) NC: usize = 128;

pub(crate) fn gemv_n<T>(
    ...

    // y <- beta y
    scal(beta, y.reborrow());

    // iterating over NC-wide panels of A
    for (cols, a_panel) in a.col_panels(NC) {
        // x scalars relevant for col panel weights
        let x_panel = VecRef::new(x.slice(cols));

        faxpy(alpha, a_panel, x_panel, y.reborrow());
    }
}
```

The outer GEMV loop first batches the matrix into larger column panels of width N_C. This is a macro-kernel blocking parameter, not the fused width n_r itself. This does not really do anything, though. Internally FAXPY divides the NC-wide column panels to smaller n_r wide chunks, so this does nothing to reduce y traffic. It does reduce some loop/function/slice setup overhead, but it's not a bottlneck at all.[1] I found $n_r = 4$ and $m_r = 64$ to work best. These values are not universal. They are an apple-M4 compromise between keeping enough independent work in registers, avoiding register pressure, and keeping the inner loop simple enough for the compiler. Increasing n_r reduces y traffic further, but also asks the kernel to keep more column data and scalars live in registers at once. Here are the benchmarks relative to OpenBLAS:

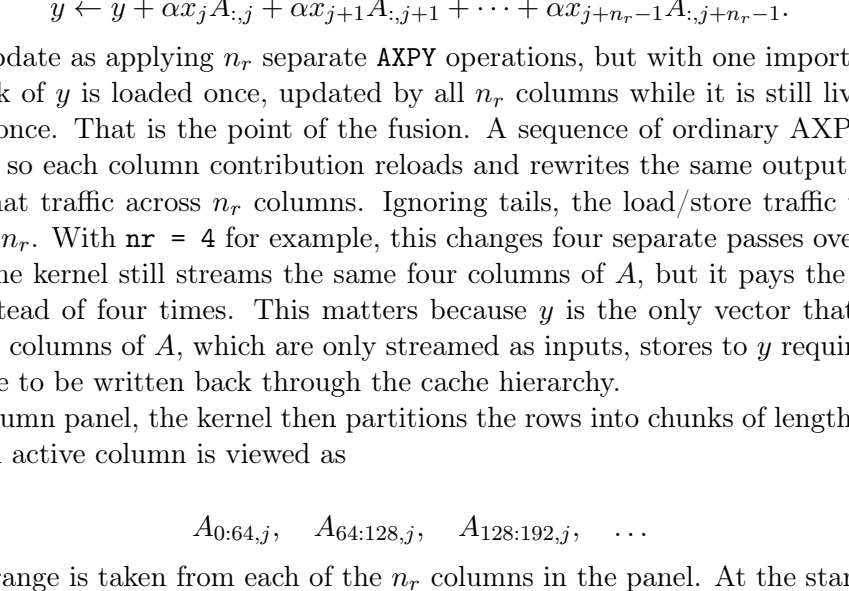

Figure 2: Single-threaded no-transpose SGEMV bandwidth benchmark relative to OpenBLAS armv8.

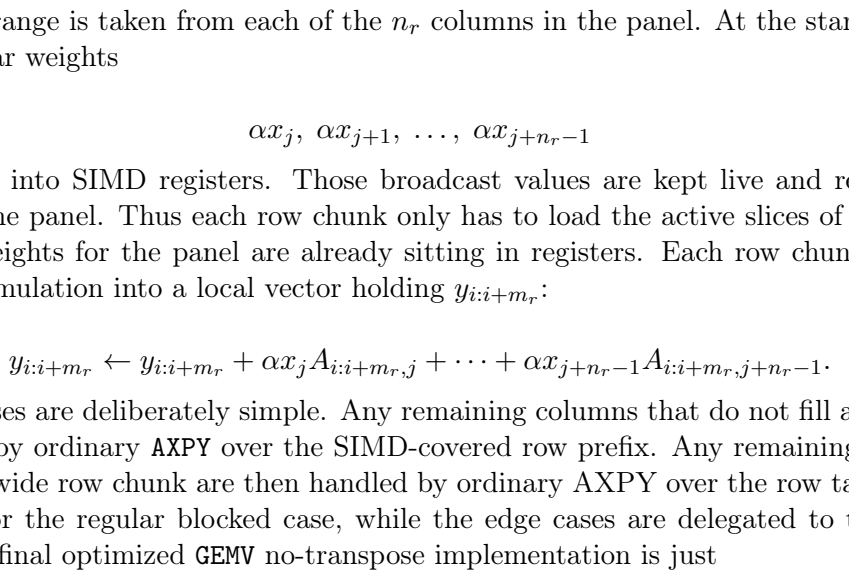

Figure 3: Single-threaded no-transpose SGEMV time benchmark relative to OpenBLAS armv8.

For a matrix of size 2048 × 2048, OpenBLAS sustains, on average, an effective memory-bandwidth of 22.29GB/s. `lak` sustains 22.9GB/s. Similarly, OpenBLAS, on average, takes 376.4μs, whereas `lak` takes 366.4μs. The full code for the FAXPY kernel is shown in the appendix.

3 Transpose GEMV

The transpose variant computes

$$y \leftarrow \alpha A^T x + \beta y. \tag{6}$$

For a column-major matrix, this case is very different from no-transpose GEMV. Instead of accumulating each column of A into the full output vector, each column of A contributes to exactly one entry of y:

$$\begin{aligned} y_j &\leftarrow \alpha A_{:,j}^T x + \beta y_j \\ &= \alpha\,\mathrm{dot}(A_{:,j}, x) + \beta y_j. \end{aligned} \tag{7}$$

Thus transpose GEMV is a loop over columns of A, where each column update is just a level-1 DOT followed by a scalar update to one entry of y. Unlike the no-transpose case, there is no repeated streaming through the entire output vector. Each element of y is read and written once.

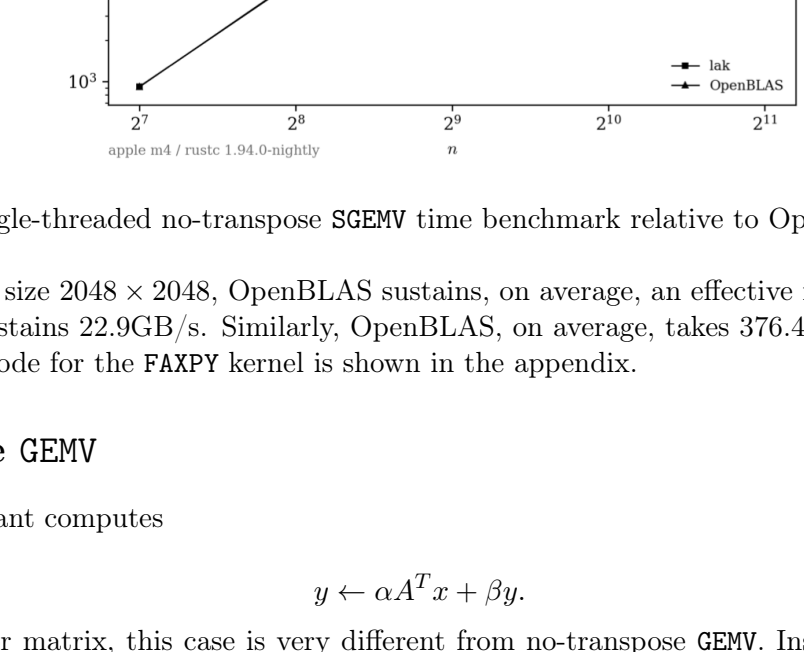

Figure 4: `lak` transpose GEMV implementation. Each output entry y_j is updated by taking the dot product of x with the corresponding column $A_{:,j}$, then accumulating the result into y_j.

This means the naive-looking implementation is already roughly optimal. Since the matrix is column-major, each $A_{:,j}$ is contiguous, so the inner operation can reuse the optimized level-1 DOT kernel directly:

```
pub fn gemv_t<T>(
    alpha: T,
    beta: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    mut y: VecMut<'_, T>,
)
where
    T: Copy
        + AddAssign
        + Mul<Output = T>
        + MulAssign,
{
    // verifies consistent dimensions
    assert_length_eq_n_rows!(a, x);
    assert_length_eq_n_cols!(a, y);

    // y <- beta y
    scal(beta, y.reborrow());

    let y_slice = y.as_slice_mut();

    // y_j <- y_j + alpha dot(A_{:,j}, x)
    for j in 0..a.n_cols() {
        let aj = a.col(j);
        y_slice[j] += alpha * dot(aj, x);
    }
}
```

There is nothing to fuse here. Each dot product only updates a single scalar y_j. The output traffic is already minimal – one read and one write of each entry of y. This reverses the optimization problem. Each output entry is a scalar accumulator, so the question is simply whether the input column and x can be used efficiently by the DOT kernel. Hence the performance of transpose GEMV is the performance of the level-1 DOT kernel. `lak`'s DOT kernel is already optimized, so this simple implementation is enough to reach competitive performance.

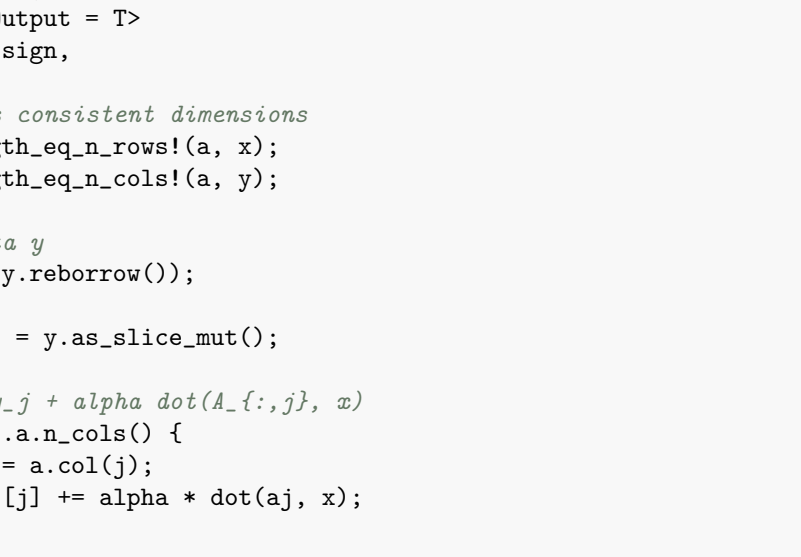

Figure 5: Single-threaded transpose SGEMV bandwidth benchmark relative to OpenBLAS armv8.

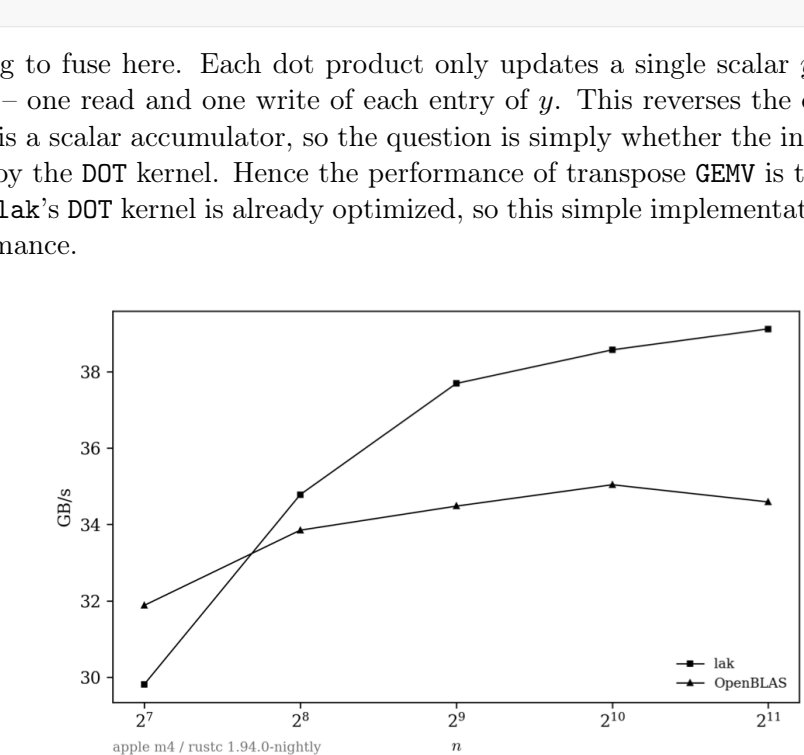

Figure 6: Single-threaded transpose SGEMV time benchmark relative to OpenBLAS armv8.

For small matrices, OpenBLAS is slightly faster. At $n = 128$, OpenBLAS takes 2.103 μs on average, while `lak` takes 2.249 μs, a difference of about 0.15 μs. Fixed overheads and dispatch details matter, I guess. For larger matrices, however, the simple DOT-based implementation is unusually strong. At $n = 2048$, OpenBLAS takes 487.4 μs on average, while `lak` takes 429.4 μs.

And again, since the level-2 BLAS is also memory-bound, the benchmarks have the same shape relative to OpenBLAS for double precision GEMV – both no-transpose and transpose, too.

4 Appendix

The inner FAXPY kernel, described in Algorithm 1, is the bulk of no-transpose GEMV:

```
// faxpy.rs

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

use crate::l1::axpy;
use crate::types::{MatRef, VecMut, VecRef};
use crate::traits::Fma;

pub const N_ROWS_PER_CHUNK: usize = 64;
pub(crate) const N_COLS_PER_CHUNK: usize = 4;

/// a "fused" axpy / mini no-transpose gemv panel:
pub fn faxpy<T>(
    alpha: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    mut y: VecMut<'_, T>,
)
where
    T: SimdElement
        + AddAssign<T>
        + Mul<Output=T>
        + Copy
        + Fma,

    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();
    let n_row_chunks = n_rows / N_ROWS_PER_CHUNK;
    let n_col_chunks = n_cols / N_COLS_PER_CHUNK;

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

    for col_chunk in 0..n_col_chunks {
        let j = col_chunk * N_COLS_PER_CHUNK;

        let x1 = Simd::<T, N_ROWS_PER_CHUNK>::splat(alpha * x_slice[j]);
        let x2 = Simd::<T, N_ROWS_PER_CHUNK>::splat(alpha * x_slice[j + 1]);
        let x3 = Simd::<T, N_ROWS_PER_CHUNK>::splat(alpha * x_slice[j + 2]);
        let x4 = Simd::<T, N_ROWS_PER_CHUNK>::splat(alpha * x_slice[j + 3]);

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

            let y_beg = i;
            let y_end = i + N_ROWS_PER_CHUNK;

            let col1_beg = j * n_rows + i;
            let col2_beg = (j + 1) * n_rows + i;
            let col3_beg = (j + 2) * n_rows + i;
            let col4_beg = (j + 3) * n_rows + i;

            let c1 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
                &a_slice[col1_beg..col1_beg + N_ROWS_PER_CHUNK],
            );
            let c2 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
                &a_slice[col2_beg..col2_beg + N_ROWS_PER_CHUNK],
            );
            let c3 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
                &a_slice[col3_beg..col3_beg + N_ROWS_PER_CHUNK],
            );
            let c4 = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(
                &a_slice[col4_beg..col4_beg + N_ROWS_PER_CHUNK],
            );

            let ychunk = &mut y_slice[y_beg..y_end];
            let mut yv = Simd::<T, N_ROWS_PER_CHUNK>::from_slice(ychunk);

            yv = x1.fma(c1, yv);
            yv = x2.fma(c2, yv);
            yv = x3.fma(c3, yv);
            yv = x4.fma(c4, yv);

            yv.copy_to_slice(ychunk);
        }
    }

    let row_tail_beg = n_row_chunks * N_ROWS_PER_CHUNK;
    let col_tail_beg = n_col_chunks * N_COLS_PER_CHUNK;

    // doing axpy on leftover columns
    for j in col_tail_beg..n_cols {
        let a_vec = VecRef::new(&a_slice[j * n_rows..j * n_rows + row_tail_beg]);
        let y_vec = VecMut::new(&mut y_slice[..row_tail_beg]);

        axpy(alpha * x_slice[j], a_vec, y_vec);
    }

    // doing axpy on leftover rows
    if row_tail_beg < n_rows {
        for j in 0..n_cols {
            let xalpha = alpha * x_slice[j];
            let a_tail = &a_slice[j * n_rows + row_tail_beg..(j + 1) * n_rows];
            let y_tail = &mut y_slice[row_tail_beg..n_rows];

            let a_vec = VecRef::new(a_tail);
            let y_vec = VecMut::new(y_tail);

            axpy(xalpha, a_vec, y_vec);
        }
    }
}
```

And finally, the no-transpose and transpose GEMV callers:

```
// gemv.rs

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

use crate::traits::Fma;
use crate::types::{MatRef, VecMut, VecRef, Transpose};
use crate::{assert_length_eq_n_cols, assert_length_eq_n_rows};

use crate::l1::{
    scal::scal,
    dot::dot,
    dot::LANES,
};
use crate::fused::faxpy::{faxpy, N_ROWS_PER_CHUNK};

pub(crate) const NC: usize = 128;

/// no transpose gemv
pub(crate) fn gemv_n<T>(
    alpha: T,
    beta: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    mut y: VecMut<'_, T>,
)
where
    T: AddAssign
        + Mul<Output=T>
        + MulAssign
        + SimdElement
        + Fma,

    Simd<T, N_ROWS_PER_CHUNK>: SimdFloat<Scalar=T>
        + AddAssign
        + Mul<Output=Simd<T, N_ROWS_PER_CHUNK>>
        + Fma,
{
    assert_length_eq_n_cols!(a, x);
    assert_length_eq_n_rows!(a, y);

    scal(beta, y.reborrow());

    for (cols, a_panel) in a.col_panels(NC) {
        let x_panel = VecRef::new(x.slice(cols));
        faxpy(alpha, a_panel, x_panel, y.reborrow());
    }
}

/// transpose gemv
pub(crate) fn gemv_t<T>(
    alpha: T,
    beta: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    mut y: VecMut<'_, T>,
)
where
    T: Copy
        + Mul<Output=T>
        + MulAssign
        + SimdElement
        + Default
        + AddAssign
        + Fma
        + Add<Output=T>,

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

    scal(beta, y.reborrow());

    let y_slice = y.as_slice_mut();

    for j in 0..a.n_cols() {
        let aj = a.col(j);
        y_slice[j] += alpha * dot(aj, x);
    }
}


/// general matrix-vector multiplication
///
/// y <- alpha * A * x + beta * y
///
/// args:
/// * trans: [Transpose] - whether to use A or A^T
/// * alpha: T - scales A * x
/// * beta: T - scales y
/// * a: [MatRef] - matrix A
/// * x: [VecRef] - vector x
/// * y: [VecMut] - vector y
pub fn gemv<T>(
    trans: Transpose,
    alpha: T,
    beta: T,
    a: MatRef<'_, T>,
    x: VecRef<'_, T>,
    y: VecMut<'_, T>,
)
where
    T: AddAssign
        + Mul<Output=T>
        + MulAssign
        + SimdElement
        + Fma
        + Default
        + Add<Output=T>,

    Simd<T, N_ROWS_PER_CHUNK>: SimdFloat<Scalar=T>
        + AddAssign
        + Mul<Output=Simd<T, N_ROWS_PER_CHUNK>>
        + Fma,

    Simd<T, LANES>: SimdFloat<Scalar=T>
        + Fma
        + AddAssign,
{
    match trans {
        Transpose::NoTranspose => gemv_n(alpha, beta, a, x, y),
        Transpose::Transpose   => gemv_t(alpha, beta, a, x, y),
    }
}
```

The public GEMV function is just a pattern `match`.

[1] Making $N_C = n_r$ makes the routine run ∼1GFLOP/s slower.