Two GEMMs Walk Into a Compiler

GEMM Working Note #2

devald@stanford.edu

05 June 2026

Abstract

I did not take computer science in undergrad. For those of you who did, and who know a lot about compilers and optimization, this note may be difficult to read. I painfully learn about inlining and a bit about LLVM behavior while trying to understand why two structurally similar GEMM implementations ran at very different speeds. After many wrong guesses, and a lot of generated assembly, I learned that source-code equivalence is not codegen equivalence. This debugging rabbit hole eventually, and mostly by chance, led to a safe Rust GEMM that outpaced OpenBLAS on small-to-medium matrices and stayed competitive at larger sizes for both f32 and f64.

To keep track through the note:

Version	Structure	f32 $n = 512$
generic	separate `kernel_mrnr`	42.37 GFLOP/s
f32 rewrite	full path inside branch	40.31 GFLOP/s
forced inline	inline through microkernel	38.65 GFLOP/s
forced inline helper	separate helper, still forced inline	36 GFLOP/s
final	separate helper, no forced inline	**51.35** GFLOP/s
openblas		**55.89** GFLOP/s

1 Context

I described the intended GEMM microkernel structure in the previous working note. The outer routine uses only `NC/KC` blocking. It works on an `NC`-wide full column-panel of C, and updates that panel one `KC` block at a time. Inside that panel, each column of C is viewed as a collection of register-sized chunks, each `MR` elements long. The microkernel works on `NR` adjacent column chunks at a time, repeating this until the full `NC`-wide panel has been updated. For `lak`, I set `NR=4`. The full-panel path is then:

```
for each 4-column panel of C:
    load 4 C vectors
    for kk in KC:
        load one A vector
        load 4 B scalars
        FMA into 4 C vectors
    store 4 C vectors
```

From LAK Working Note #1, we know that level-3 routines need type-specific blocking constants. GEMM has enough arithmetic intensity that register blocking, cache blocking, and SIMD width actually matter. Still, as an experiment, I wanted to see how far a generic GEMM implementation could go. So I wrote one.

The generic microkernel had this shape:

```
pub(crate) fn microkernel<T>(...) {
    for j in (0..nc).step_by(NR) {
        let nr = (nc - j).min(NR);

        if nr == NR {
            kernel_mrnr(...);
        } else {
            // leftover columns via faxpy
            for jj in j..nc {
                faxpy(...);
            }
        }
    }
}

fn kernel_mrnr<T>(...) {
    // full MR x NR SIMD kernel
    // row tail also handled here
}
```

Benchmarking confirmed the problem with one generic `MR`. The best value for f32 was `MR=16`; the best value for f64 was `MR=8`. That was enough reason to split GEMM into explicit `sgemm` and `dgemm` routines. Each routine could now carry its own blocking constants.

So I wrote the f32 version. It was supposed to be the same algorithm. The outer blocking was the same. The `NR=4` update pattern was the same. The main difference is embarrassingly small: instead of keeping the full `MR` × `NR` path in a separate `kernel_mrnr` function, I put the whole thing directly inside the `if nr == NR` branch.

```
pub(crate) fn sgemm_nn_micro(...) {
    for j in (0..nc).step_by(NR_F32) {
        let nr = (nc - j).min(NR_F32);

        if nr == NR_F32 {
            // full MR x NR SIMD kernel directly here
            //
            // load c0/c1/c2/c3
            // for kk in 0..kc:
            //     load A vector
            //     load 4 B scalars
            //     FMA into 4 C vectors
            // store c0/c1/c2/c3
            //
            // row tail also directly here
        } else {
            // leftover columns via faxpy
            for jj in j..nc {
                faxpy(...);
            }
        }
    }
}
```

I do not know why I did this. It looked harmless. Separate function or not, it was the same algorithm. But this tiny change started the entire rabbit hole.

At $n = 512$, the generic version with a separate `kernel_mrnr` path reached about 42.37 GFLOP/s for f32 with `MR=16`. The f32-only rewrite used the same blocking constants and almost the same code, except that the full microkernel body was placed directly inside the branch. It dropped to 40.31 GFLOP/s, about 5% slower.

2 Now is where things get painful

I still wanted a common public `gemm<T>` function, to match the rest of the library. Internally, I wrote a trait dispatch. The associated `gemm` would call `sgemm` for f32 and `dgemm` for f64.

```
pub fn gemm<T>(...)
where
    T: GemmDispatch,
{
    T::gemm(...)
}
```

I did not know much about compiler behavior. The source code between the faster generic implementation and the slower `sgemm` looked almost the same. So I blamed the trait dispatch. It seemed plausible that the 5% slowdown came from hiding the specialized routine behind `T::gemm`.

2.1 Trait Dispatch Assembly

The generated assembly for the faster generic implementation with f32 arguments and $n = 512$ can be found here. It is a clean 578-line file with a specialized inner hot loop:

```
LBB4_26:
    fmov s30, w3
    fmul s27, s27, s30
    fmul s28, s28, s30
    ldr s31, [x30, #6144]
    fmul s29, s29, s30
    fmul s30, s31, s30
    fmla.4s v4, v25, v27[0]
    fmla.4s v7, v26, v27[0]
    ldp q8, q31, [x26]
    fmla.4s v22, v31, v27[0]
    fmla.4s v21, v8, v27[0]
    fmla.4s v5, v25, v28[0]
    fmla.4s v16, v26, v28[0]
    fmla.4s v20, v31, v28[0]
    fmla.4s v19, v8, v28[0]
    fmla.4s v3, v25, v29[0]
    fmla.4s v6, v26, v29[0]
    fmla.4s v18, v31, v29[0]
    fmla.4s v17, v8, v29[0]
    fmla.4s v1, v25, v30[0]
    fmla.4s v2, v26, v30[0]
    fmla.4s v24, v31, v30[0]
    add x15, x15, #1
    fmla.4s v23, v8, v30[0]
    add x26, x26, #2048
    add x14, x14, #512
    add x30, x30, #4
    cmp x28, x15
    ...
    stp q21, q22, [x11]
    stp q7, q4, [x11, #32]
    add x11, x4, x12
    stp q19, q20, [x11]
    stp q16, q5, [x11, #32]
    add x11, x21, x12
    stp q17, q18, [x11]
    stp q6, q3, [x11, #32]
    add x11, x17, x12
    stp q23, q24, [x11]
    add x2, x2, #64
    add x7, x7, #16
    stp q2, q1, [x11, #32]
    ...
```

The specific `sgemm` assembly looked very different. It showed almost nothing except a call to the trait implementation:

```
bl <f32 as lak::traits::GemmDispatch>::gemm
```

So my suspicion became: maybe the dispatch path lost optimizer visibility. Maybe the generic implementation, after monomorphization, showed LLVM the whole routine. Maybe the `sgemm` path hid too much behind a call.

This was the first wrong guess. I made a public `sgemm` function and changed the benchmark to call it directly. No trait dispatch.

```
pub fn sgemm(...) {
    sgemm_nn(...)
}
```

Performance stayed the same as the trait-dispatch version. The assembly changed, but not in the way I wanted. Instead of calling `<f32::GemmDispatch>::gemm`, it called `lak::l3::gemm::gemm::sgemm`.

The generic assembly still showed the whole routine. It had the loops, loads, stores, and `fmla` instructions visible in the output. It also did not show the tail paths. Somehow, for this benchmark, LLVM could see enough to specialize the generic path for $n = 512$. It could remove paths that would not run. It did not do that for the direct `sgemm` call.

3 I Learned What Inline Actually Does

It took longer than I would like to admit to learn about and try inlining. I first put

```
#[inline]
pub fn sgemm(...) {
    sgemm_nn(...)
}
```

above the public `sgemm` function. That changed the generated assembly from

```
bl lak::l3::gemm::gemm::sgemm
```

to

```
bl lak::l3::gemm::gemm_nn::sgemm_nn
```

The compiler could now now see one layer deeper.

The generic routine had no visible function calls in the assembly, so I tried to make `sgemm` look like that. I added `#[inline(always)]` all the way down:

```
#[inline(always)]
pub fn sgemm(...) { ... }

#[inline(always)]
pub(crate) fn sgemm_nn(...) { ... }

#[inline(always)]
pub(crate) fn sgemm_nn_micro(...) { ... }
```

The `always` does not ask politely. It forces the compiler to inline the function.

Guess what! `sgemm`'s performance ***got worse***. For f32 and $n = 512$, it dropped to 38.65 GFLOP/s on average. The generic routine was still around 42.37 GFLOP/s. The direct f32 routine was now almost 9% slower.

3.1 Post Inlining Assembly

The new `sgemm` assembly had no function calls. The inlining worked. But the file was *over 1700 lines long*.

The assembly included the full general routine: the full path, the leftover-column path, the row-tail path, and many bounds-check paths. The generic assembly, by contrast, looked specialized to the $n = 512$ case. It did not drag the whole possible GEMM routine into the output.

This was confusing. I thought showing LLVM more code would make it optimize more. That is what seemed to happen with the generic routine. But here, more inlining only made the generated function much larger, and the benchmark got worse.

4 My Wits End

At this point one visible difference remained. In the generic routine, the full `MR` × `NR` path lived in its own function. In the f32 routine, it lived directly inside `sgemm_nn_micro`. So I made the f32 routine match the generic shape. The full path became its own function again. The row tail stayed inside that function, just like before.

```
sgemm_nn_micro(...) {
    for j in ... {
        if nr == NR_F32 {
            skernel_mrnr(...)
        } else {
            leftover columns via faxpy
        }
    }
}

#[inline(always)]
fn skernel_mrnr(...) {
    full MR x NR SIMD kernel
    row tail also handled here
}
```

This matched the generic routine's structure. I still kept `#[inline(always)]` on the full-path microkernel, because the generic routine's assembly literally had the full hot loop, with the microkernel, inlined for LLVM to see and optimize. There's no reason for me not to inline here.

Guess what! `sgemm`'s performance ***got even worse***. It dropped to about 36 GFLOP/s. What is happening.

5 Victory at last

I removed `#[inline(always)]` above `skernel_mrnr` to see what would happen. That was it. The routine became faster than the generic version. For $n = 512$, direct `sgemm` reached **51.35 GFLOP/s**. OpenBLAS armv8 reached 55.89 GFLOP/s.

And what I hoped for in the previous GEMM working note finally happened. LAK did well for small-to-medium matrices being fully-safe. The OpenBLAS curve is strange, though, especially at $n = 64$, so I do not want to overstate it. But for this setup, LAK beat this OpenBLAS build on several small-to-medium square GEMMs. So for matrices roughly up to size 200×200, LAK can be comfortably used.

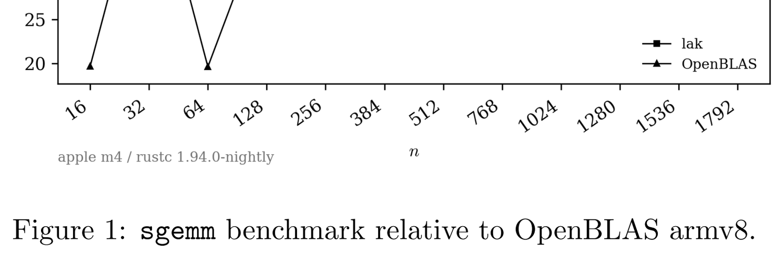

Figure 1: `sgemm` benchmark relative to OpenBLAS armv8.

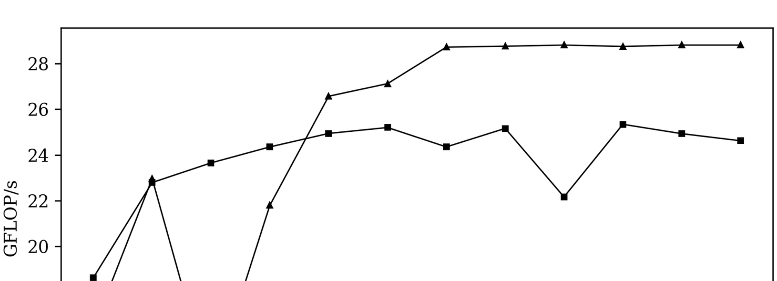

Figure 2: `dgemm` benchmark relative to OpenBLAS armv8.

The same structure worked for both f32 and f64 with tuned blocking constants and nontrivial α and β. The final `sgemm` structure became:

```
sgemm_nn_micro(...) {
    for j in (0..nc).step_by(NR_F32) {
        if nr == NR_F32 {
            skernel_mrnr(...);
        } else {
            leftover columns via faxpy
        }
    }
}

fn skernel_mrnr(...) {
    // full MR x NR SIMD kernel
    // row tail also handled here
}
```

6 Remarks

I still do not fully understand what happened. The final `sgemm` routine has the same visible structure as the generic routine, but it runs faster. Some light inlining helped expose the outer calls. Forcing the inner full-path microkernel to inline made things worse.

My best guess is that the function boundary was useful because it "localized" the full-path kernel and kept it separate from the full hot loop. Keeping the nonessentail tail inside the else branch directly was fine for some reason, though. When I forced the full-path kernel to inline, LLVM had one large routine containing the full path, leftover columns, row tails, and bounds checks. The hot loop was now surrounded by code that should not matter for $n = 512$, but still affected the generated function. More code gave the optimizer more to see, but also more to carry, schedule, and prove away. Leaving `skernel_mrnr` as its own function gave the compiler a smaller, and more cleanly separated problem. The outer loop only had to choose between the full `NR` path and the leftover path and the separation was more visible (this is still hand-wavy).