GEMM Working Note #1

devald@stanford.edu

02 June 2026

Abstract

These notes discuss developing a fast GEMM implementation in pure, safe Rust. I only focus on the no-transpose × no-transpose case.

This note discusses how the usual BLAS blocking hierarchy changes when matrices are already stored contiguously and column-major. In standard BLAS, blocking must manage both cache reuse and the awkwardness of arbitrary strides, often by packing panels before they reach the microkernel. In `lak`, the layout is already friendly to the kernel, so the routine can be organized around streaming full column panels instead. From there, I discuss cache blocking, register blocking, and why in principle, in `lak`, the usual `MC` blocking can mostly disappear while `NC`, `KC`, `MR`, and `NR` still guide the structure of the microkernel.[1]

The final implementation, single-threaded and on Apple Silicon, exceeds OpenBLAS armv8 for "short" (small m) matrices, but still performs well for larger matrices.

Let A be an $m \times k$ matrix. B is $k \times n$. The matrix product $AB = C$ yields an $m \times n$ matrix C. For $m = 3$, $k = 4$, $n = 3$, this gives

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ b_{30} & b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22}. \end{bmatrix} \tag{1}$$

The first column of C is given by

$$c_{:,0} = b_{00}a_{:,0} + b_{10}a_{:,1} + b_{20}a_{:,2} + b_{30}a_{:,3}, \tag{2}$$

and for general C column j,

$$c_{:,j} = b_{0,j}a_{:,0} + b_{1,j}a_{:,1} + b_{2,j}a_{:,2} + b_{3,j}a_{:,3}. \tag{3}$$

1 Noticing Reuse

`lak` is contiguous-only and column-major. So column slices are our best friend. Looking at all columns of C together,

$$\begin{aligned} c_{:,0} &= b_{00}\,\mathbf{a_{:,0}} + b_{10}\,\mathbf{a_{:,1}} + b_{20}\,\mathbf{a_{:,2}} + b_{30}\,\mathbf{a_{:,3}} \\ c_{:,1} &= b_{01}\,\mathbf{a_{:,0}} + b_{11}\,\mathbf{a_{:,1}} + b_{21}\,\mathbf{a_{:,2}} + b_{31}\,\mathbf{a_{:,3}} \\ c_{:,2} &= b_{02}\,\mathbf{a_{:,0}} + b_{12}\,\mathbf{a_{:,1}} + b_{22}\,\mathbf{a_{:,2}} + b_{32}\,\mathbf{a_{:,3}}, \end{aligned} \tag{4}$$

we see every column is formed from the same columns of A, bolded above. These columns are contiguous, and get reused. The scalar weights from B change between each column of C, though. This suggests a GEMM kernel should not compute one column of C at a time. Instead, it should hold several columns of C in registers, stream through the *shared* columns of A, and use the correct B scalars to update all the C columns before storing them back. This is the same as the GEMV/`faxpy` idea. Rather than repeatedly loading and storing the same output column of C, we keep a small panel (multiple columns) of C live, reuse the same column slices of A, and update multiple columns of C at once. We get more out of A data loaded, and we load and store columns of C less.

- Each active column of C is loaded once, updated many times, and stored once.
- Multiple columns of C are worked on at a time.
- The same entries of A are reused across several columns of C while kept in registers.

GEMM has enough reuse that the kernel can amortize memory traffic much more effectively than in GEMV. One consequence is that the routine is more confusing. There are multiple levels of loops and blocking and tail-handling. But as I'll show, `lak`'s contiguous-only case makes this routine a bit simpler.

1.1 Typical Cache Blocking

A GEMM microkernel should work with registers only. These registers are inside the CPU itself, and allow the use of SIMD operations. Once memory is in registers, work can be done very fast, and very often with good reuse as discussed above. However, once the CPU needs *new* data to be placed in registers, such as a new column of C to accumulate into, it needs to roughly go where the C matrix is stored in memory and bring the data it wants. If the data is stored far away, this becomes a brutal bottleneck that destroys arithmetic intensity. Large matrices, especially considering there are three matrices at play, together exceed the typical 32KB–128KB L1 cache size. To make better use of caches, BLAS divides the work into smaller "blocks" that can fit close to the CPU. Then, when the microkernel needs data from the current block, it can usually get that data from cache instead of going back to main memory.

$$\begin{array}{c} \\ \texttt{MC} \end{array} \overset{\overset{\texttt{NC}}{\downarrow}}{\begin{bmatrix} \boxed{\begin{matrix} c_{00} & c_{01} & c_{02} & \cdots \\ c_{10} & c_{11} & c_{12} & \cdots \\ c_{20} & c_{21} & c_{22} & \cdots \end{matrix}} & \begin{matrix} c_{0n} \\ c_{1n} \\ c_{2n} \end{matrix} \\ \begin{matrix} \vdots & \vdots & \vdots & \ddots \\ c_{m0} & c_{m1} & c_{m2} & \cdots \end{matrix} & \begin{matrix} \vdots \\ c_{mn} \end{matrix} \end{bmatrix}}$$

Typical BLAS implementations therefore update C one block at a time. This block has shape `MC` × `NC`.[2] The goal is not to compute a single entry of C and move on. The goal is to keep a useful region of C nearby, accumulate into its columns many times with good register reuse, and only then move to the next region.

From Equation 4, updating a block of C requires the matching rows of A and columns of B. So an `MC` × `NC` block of C depends on an `MC` $\times k$ row-panel of A and a $k\times$ `NC` column-panel of B. These are the pieces of A and B needed to finish that block of C. The inner dimension k can be large too, though. Hence the `MC` $\times k$ panel of A and the $k\times$ `NC` panel of B may still be too large to keep in cache. For this reason BLAS also blocks across k, walking across in smaller chunks of size `KC`. Each step uses an `MC` × `KC` panel of A and a `KC` × `NC` panel of B to apply one partial update to the same `MC` × `NC` block of C. After all `KC` chunks have been processed, that block of C is complete.

$$\begin{array}{c} \\ \texttt{MC} \end{array} \overset{\overset{\texttt{KC}}{\downarrow}\quad\overset{\texttt{KC}}{\downarrow}\quad\overset{\texttt{KC}}{\downarrow}}{\begin{bmatrix} \boxed{\begin{matrix} a_{00} & a_{01} & \cdots & a_{0k} \\ a_{10} & a_{11} & \cdots & a_{1k} \\ a_{20} & a_{21} & \cdots & a_{2k} \end{matrix}} & \\ \begin{matrix} \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \cdots & a_{mk} \end{matrix} & \end{bmatrix}} \overset{\overset{\texttt{NC}}{\downarrow}}{\begin{bmatrix} \boxed{\begin{matrix} b_{00} & b_{01} & b_{02} & \cdots \\ b_{10} & b_{11} & b_{12} & \cdots \end{matrix}} & \begin{matrix} b_{0n} \\ b_{1n} \end{matrix} \\ \boxed{\begin{matrix} \vdots & \vdots & \vdots & \ddots \\ b_{k-1,0} & b_{k-1,1} & b_{k-1,2} & \cdots \end{matrix}} & \begin{matrix} \vdots \\ b_{k-1,n} \end{matrix} \\ \begin{matrix} b_{k0} & b_{k1} & b_{k2} & \cdots \end{matrix} & b_{kn} \end{bmatrix}} = \overset{\overset{\texttt{NC}}{\downarrow}}{\begin{bmatrix} \boxed{\begin{matrix} c_{00} & c_{01} & c_{02} & \cdots \\ c_{10} & c_{11} & c_{12} & \cdots \\ c_{20} & c_{21} & c_{22} & \cdots \end{matrix}} & \begin{matrix} c_{0n} \\ c_{1n} \\ c_{2n} \end{matrix} \\ \begin{matrix} \vdots & \vdots & \vdots & \ddots \\ c_{m0} & c_{m1} & c_{m2} & \cdots \end{matrix} & \begin{matrix} \vdots \\ c_{mn} \end{matrix} \end{bmatrix}}$$

So the blocking has two jobs. `MC` and `NC` choose how much of C to keep close while it is being updated. `KC` chooses how much of the shared inner dimension to bring in at one time so the corresponding panels of A and B also stay cache-friendly. As we'll see, this makes `KC` also dictate how often we keep columns of C hot in SIMD registers before writing their accumulated results back to memory and loading the next chunk of columns of C, which, thanks to the blocking, aren't very far to access. The full GEMM is then built by repeating this process over all blocks of C.

1.2 Register Blocking

The microkernel works on columns of an `MC` × `NC` block of C at a time with SIMD registers. The number of columns of C in this block dictates how many times we can reuse a column of A in the corresponding `MC` × `KC` block. Ideally, we would want to use as many columns of C as possible. But there are only a finite number of registers in the CPU. Inside each `MC` × `NC` block of C, therefore, we *further* divide the work into smaller blocks of size `MR` × `NR`[3] to be able to not overdraft on our SIMD register count, which would make the compiler very sad. As a result, the block of A is further divided into panels `MR`-rows long, and the block of B is further divided into panels `NR`-columns wide.

2 LAK is easier

The number of loops and blockings to keep track of in typical BLAS makes my head spin. BLAS has all of this because it also deals with non-contiguous matrices, and packs the `MC/NC/KC` blocks of non-contiguous memory into corresponding blocks of contiguous memory before passing them into the microkernel so that it can fly.

In `lak`, everything is already contiguous, so `MC` loses its meaning. Note that it's only `MC`, because it's the only dimension where blocking would introduce non-unit strides – after every `MC`-long column, the compiler would have to jump $m -$ `MC` elements in memory to get to the next column in the block. So, `lak` will work on *full* column panels of C at a time – no `MC`. This will make the routine easier for me, and I hope/think it will still remain competitive, and possibly exceed BLAS for "short" matrices, where the overhead from BLAS blocking by `MC` doesn't pay off yet.

$$\begin{array}{c} \\ m \end{array} \overset{\overset{\texttt{NC}}{\downarrow}}{\begin{bmatrix} \boxed{\begin{matrix} c_{00} & c_{01} & c_{02} & \cdots \\ c_{10} & c_{11} & c_{12} & \cdots \\ c_{20} & c_{21} & c_{22} & \cdots \end{matrix}} & \begin{matrix} c_{0n} \\ c_{1n} \\ c_{2n} \end{matrix} \\ \begin{matrix} \vdots & \vdots & \vdots & \ddots \\ c_{m0} & c_{m1} & c_{m2} & \cdots \end{matrix} & \begin{matrix} \vdots \\ c_{mn} \end{matrix} \end{bmatrix}}$$

And anyway internally, the full column panels get "row-blocked" in a sense by working on `MR`-long SIMD registers at a time.

2.1 Edit after writing the routine

After writing the routine, I think the usual BLAS blocking names are a little misleading for this contiguous-only case. In packed BLAS, `MC`, `NC`, and `KC` describe cache-sized panels that are often copied into packed buffers before reaching the microkernel. In `lak`, the inputs are already contiguous, so there is no analogous packing step.

The routine instead streams through full column panels of C. One could think that for large matrices, even one full column of C and its working set from A and B could exceed cache. However, data only matters when you *touch* it. I could load in a MB worth of data into the microkernel, but the microkernel only works on `MR` × `NR` chunks at a time. So there's nothing ruining the memory hierarchy; effective row-blocking is already happening at the register level: each small chunk of C is loaded, accumulated through some amount of k, and then stored back before moving down the columns. The surrounding column panel does not need to be treated as a packed cache block; it is just the contiguous memory region the kernel walks through.[4]

With this design, `MC` essentially disappears. `NC` mostly controls how many columns are grouped in the outer loops, and in my implementation mainly affects loop structure and microkernel call overhead. `KC` remains useful, but it is better understood as controlling accumulation depth – how much of the inner dimension is consumed before an `MR` × `NR` chunk of C is written back.

[1] Edit* after optimizing GEMM for large ($n > 256$) matrices: This statement is valid only for "short" matrices where `MC` blocking overhead would reduce throughput. However, as matrices get larger, `MC` keeps working blocks involving `MC/NC/KC` in cache *between* different accumulations of `KC` blocks of A and B. This makes the updates after that involve the same re-used block of A wait less before letting the CPU fly.

[2] The `MC/NC` block naming convention originates from GotoBLAS, developed by Kazushige Goto. Here, `M` refers to row-blocking, `N` refers to column-blocking, and the `C` suffix stands for "cache".

[3] The `R` prefix is for "register," now.

[4] Refer to footnote 1. I also thought about this wrong. The purpose of blocking is to keep data in cache *between* successive `KC` block updates. For shorter matrices, not blocking by `MC` is faster. But there is a threshold where the `MC` variant becomes faster.