The Journal of Open Source Software

faer: A linear algebra library for the Rust programming language

Sarah Quiñones El Kazdadi [1]

1 Independent Researcher, France

DOI: 10.21105/joss.06099

Software

- Review
- Repository
- Archive

Editor: Jed Brown

Reviewers:

- @fclc
- @mhoemmen

Submitted: 14 November 2023
Published: 01 July 2026

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Summary

`faer` is a portable high performance dense linear algebra library written in Rust. The library offers a convenient high level API for performing matrix decompositions and solving linear systems. This API is built on top of a lower level API that gives the user more control over the memory allocation and multithreading settings.

Supported platforms include the ones supported by Rust. Explicit SIMD instructions are currently used for x86-64 and Aarch64 (NEON), with plans for SVE/SME and RVV optimizations once intrinsics for those are stabilized in Rust, possibly earlier than that if we allow usage of a Just-In-Time (JIT) backend[1].

The library provides a `Mat` type, allowing for quick and simple construction and manipulation of matrices, as well as lightweight view types `MatRef` and `MatMut` for building memory views over existing data.

These views are currently used to represent different kinds of matrices, such as generic rectangular matrices, symmetric/Hermitian/triangular (where only half of the matrix is stored) square matrices. In the future, we plan to make use of the robust Rust type-system to better express the properties of those matrices, and prevent accidental misuse of the library's API.

Multiple scalar types are supported, and the library code is generic over the data type. Native floating point types `f32`, `f64`[2], `c32`, and `c64` are supported out of the box, as well as any user-defined types that satisfy the requested interface, such as extended precision real numbers (double-double or multi-precision floats), complex numbers using the aforementioned types as the base element, dual/hyper-dual numbers[3].

Statement of need

Rust was chosen as the language for the library since it allows full control over the memory layout of data and exposes low level CPU intrinsics for SIMD[4] computations. Additionally, its memory safety features make it a perfect candidate for writing efficient and parallel code, since the compiler statically checks for errors that are common in other low level languages, such as data races and fatal use-after-free errors.

[1] Inline assembly is not entirely appropriate for our use case since it's hard to make it generic enough for all the operations and types that we wish to support.

[2] IEEE 754-2008, with no implicit `fusedMultiplyAdd` contractions and with slight differences around NaN handling. See the float semantics tracking issue for more information.

[3] These support at least for the simpler matrix decompositions (Cholesky, LU, QR). It's not clear yet how to handle iterative algorithms like the SVD and Eigendecomposition.

[4] Single instruction, multiple data operations that CPUs can use to parallelize data processing at the instruction level.

Rust also allows compatibility with the C ABI, allowing for simple interoperability with C, and most other languages by extension. Once a design has been properly fleshed out, we plan to expose a C API, along with bindings to other languages (C, C++, Python and Julia bindings are currently planned).

Aside from `faer`, the Rust ecosystem lacks high performance matrix factorization libraries that aren't C library wrappers, which presents a distribution challenge and can impede generic programming.

Features

`faer` supports vectorization of arbitrary SIMD user types by deinterleaving Array of Structures (AoS) data to Structure of Arrays (SoA) in registers on the fly. This helps keep the API simple in that it doesn't need to accomodate SoA storage in memory, and is often also benefitial from a memory locality point of view.

The library generically implements algorithms for matrix multiplication, based on the approach of Van Zee & van de Geijn (2015). For native types, `faer` uses SIMD explicitly depending on the detected CPU features, to dispatch instructions to several precompiled variants for operations that can make use of these features. An interesting alternative would be to compile the code JIT, which could improve compilation times and reduce binary size. But there are also possible downsides that have to be weighed against these advantages, such as increasing the startup time to optimize and assemble the code, as well as the gap in maturity between ahead-of-time compilation (currently backed by LLVM), and JIT compilation, for which the Rust ecosystem is still developing. The library then uses matrix multiplication as a building block to implement commonly used matrix decompositions, based on state of the art algorithms in order to guarantee numerical robustness:
- Cholesky (LLT, LDLT and Bunch-Kaufman LDLT),
- QR (with and without column pivoting),
- LU (with partial and full pivoting),
- SVD (with or without singular vectors, thin or full), and
- eigenvalue decomposition (with or without eigenvectors).

For algorithms that are memory-bound and don't make much use of matrix multiplication, `faer` uses optimized fused kernels[5]. This can immensely improve the performance of QR decomposition with column pivoting, LU decomposition with full pivoting, as well as reduction to condensed form to prepare matrices for SVD or eigenvalue decomposition, as described by Van Zee et al. (2012).

State of the art algorithms are used for each decomposition, allowing performance that matches or even surpasses other low level libraries such as OpenBLAS (Wang et al. (2013)), LAPACK (Anderson et al., 1999), and Eigen (Guennebaud et al., 2010).

To achieve high performance parallelism, `faer` uses the Rayon library (Rayon developers, 2015) as a backend, and has shown to be competitive with other frameworks such as OpenMP (Chandra et al., 2001) and Intel Thread Building Blocks (Pheatt, 2008).

Performance

Here we present the benchmarks for a representative subset of operations that showcase our improvements over the current state of the art.

The benchmarks were run on an 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz with 12 threads. Eigen is compiled with the `-fopenmp` flag to enable parallelism.

[5]For example, computing Ax and $A^\top y$ with a single pass over A, rather than two.

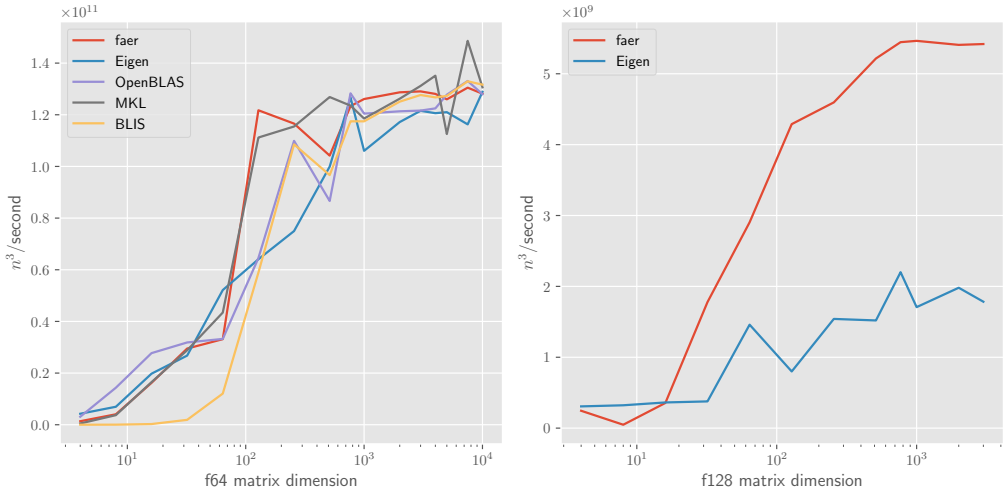

Figure 1: n^3 over run time of matrix multiplication. Higher is better

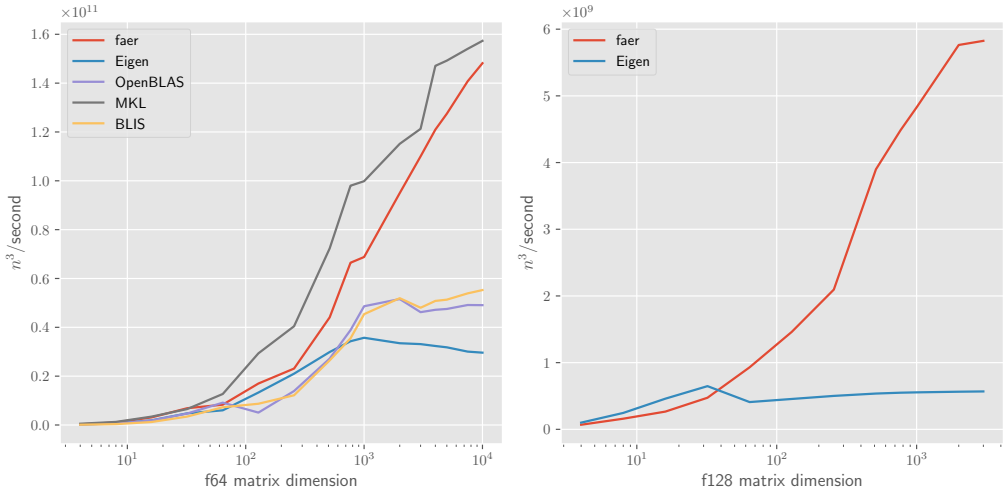

Figure 2: n^3 over run time of QR decomposition. Higher is better

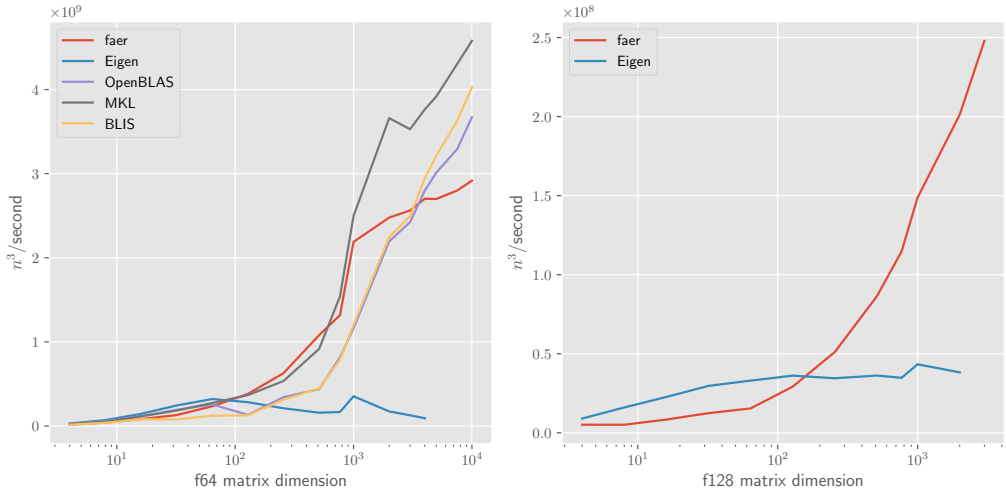

Figure 3: n^3 over run time of eigenvalue decomposition. Higher is better

Future work

We have so far focused mainly on dense matrix algorithms, which will eventually form the foundation of supernodal sparse decompositions. Sparse algorithm implementations will be showcased in a future paper.

References

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., & Sorensen, D. (1999). *LAPACK users' guide* (Third). Society for Industrial and Applied Mathematics. ISBN: 0-89871-447-8

Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., & McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann.

Guennebaud, G., Jacob, B., & others. (2010). *Eigen v3*. http://eigen.tuxfamily.org

Pheatt, C. (2008). Intel® threading building blocks. *J. Comput. Sci. Coll.*, *23*(4), 298. https://doi.org/10.1016/b978-0-12-803761-4.00011-3

Rayon developers. (2015). *Rayon*. https://github.com/rayon-rs/rayon

Van Zee, F. G., & van de Geijn, R. A. (2015). BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, *41*(3), 14:1–14:33. https://doi.org/10.1145/2764454

Van Zee, F. G., van de Geijn, R. A., Quintana-Ortí, G., & Elizondo, G. J. (2012). Families of algorithms for reducing a matrix to condensed form. *ACM Transactions on Mathematical Software*, *39*(1). https://doi.org/10.1145/2382585.2382587

Wang, Q., Zhang, X., Zhang, Y., & Yi, Q. (2013). AUGEM: Automatically generate high performance dense linear algebra kernels on X86 CPUs. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/2503210.2503219