Synthesizing Numerical Linear Algebra using Julia

Sophie Xuan[1,4], Evelyne Ringoot[1,5], Rabab Alomairy[1,2,6], Felipe Tome[1,3,7], Julian Samaroo[1,8], and Alan Edelman[1,9]

[1]Computer Science & Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA.
[2]College of Computer Science and Engineering, University of Jeddah, KSA.
[3]São Carlos School of Engineering, University of São Paulo, BR.
[4] *sopnxuan@mit.edu* [5]*eringoot@mit.edu*
[6]*rabab.alomairy@mit.edu* [7]*felipe0@mit.edu* [8]*jsamaroo@mit.edu* [9]*edelman@mit.edu*

***Abstract*—This paper presents work in progress to implement generic Julia functions for dense numerical linear algebra that support various data types and hardware (CPU and GPU) with a single API. This implementation transcends the traditional type-specific and hardware-specific LAPACK/BLAS libraries, by leveraging the Julia programming language and its LLVM-based compilation process, without sacrificing performance. This work aims to demonstrate the potential of the Julia language to advance the field of high-performance computing by providing future-proof, efficient, and generic alternatives to legacy libraries.**

I. Introduction

The performance of Basic Linear Algebra Subroutines (BLAS) is crucial for many scientific computations [4]. The LAPACK/BLAS modules encapsulate fundamental dense linear algebra algorithms and are frequently used as benchmarks for the performance of hardware, software engineering optimizations, and programming languages. Historically, LAPACK/BLAS subroutines have been implemented in Fortran, optimized for vectorization, multicore machines, and GPUs [9], [1]. These classic algorithms have been continuously re-adapted with each new hardware evolution to maintain optimal performance.

Recently, there has been a significant effort to provide native implementations of linear algebra operations in C++ and Python, moving away from FORTRAN. With C++26, all BLAS operations, including vector, matrix-vector, and matrix-matrix operations, have become part of the language standard [5]. In addition, NVIDIA [7] has provided comprehensive math libraries for the Python ecosystem, further enhancing the performance capabilities and ease of use for developers working with advanced linear algebra operations.

This work presents a contribution to the effort to make advanced linear algebra more accessible to the general audience and providing them access to the capabilities of High-Performance Computing. We propose implementing the LAPACK/BLAS subroutines in Julia, providing hardware- and data type-agnostic scalable performance. Julia language's multiple-dispatch and type inference enable LLVM to generate optimized machine code for performance over various argument types.[2] The integration of advanced optimization features such as specialization, interoperability, loop unrolling, and vectorization enable achieving performance close to native code [8]. These features permit the development of a single generic API for various hardware and data types, which reduces development time and provides composability: new data types can be supported by the existing BLAS implementations without modifications, making the library future-proof. We focus on the BLAS/LAPACK library as it is the most widely recognized linear algebra library, but our work could equally be applied to novel alternatives to BLAS e.g., BLIS. [10]

The original contribution of this work consists of:

- Demonstrating efficient LAPACK/BLAS performance in Julia for medium to large problem sizes.
- Demonstrating that a single generic API supports several hardwares and data types.
- Providing open-source implementations.

II. Methods

To demonstrate portability and performance, we implemented the **larfb** and **unmqr** functions, which are now publicly available. **unmqr** applies an orthogonal matrix Q formed by a the block-householder factorization Y of a QR-factorization to a general rectangular matrix A:

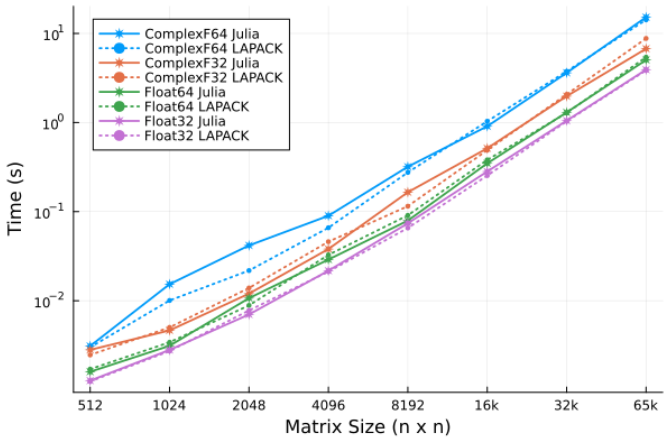
$$A + AYTY^T = AQ \quad \text{and} \quad A + YTY^TA = QA$$

with T being the scalar factors of elementary reflectors. The **larfb**) function utilized by **unmqr** performs the individual projections.

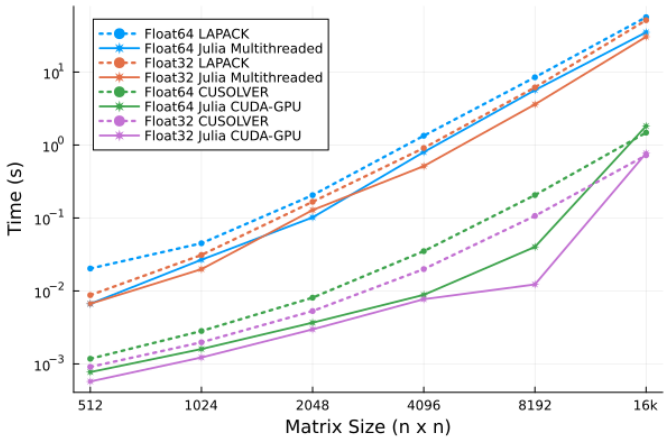
We implement the LAPACK/BLAS functions as described in [4] by leveraging Julia's abstract array interface [6], which supports arbitrary element types (including 64-bit, 32-bit, 16-bit, complex, and real) and can dispatch on both CPU and GPU architectures with minimal changes to the API. Our approach utilizes Julia's metaprogramming capabilities to generate and compile efficient code dynamically at runtime, enabling flexibility and performance across architectures [3].

III. Performance

In order to demonstrate the performance of Julia BLAS/LAPACK implementations, we benchmarked the **larfb** and **unmqr** LAPACK functions across various data types (section A) and hardware types (section B) using Julia (v1.10.4) and oneMKL (v2020.0.166).



(a) Performance across data types of **larfb** function on Intel Ice Lake processor (Table I).



(b) Performance across hardware of **unmqr** function on Intel Cascade Lake CPU and V100 GPU (Table I).

Fig. 1: Performance comparison of execution time of LAPACK/CUSOLVER (dashed) and multithreaded Julia/JuliaGPU implementations (solid) in function of the size n for input matrices of size $n \times n$. For each data type and hardware type, the performance of the Julia function closely matches that of the LAPACK/CUSOLVER functions.

A. Performance across data types

Figure 1a shows the running time performance of the Julia Unified API **larfb** function (solid lines) for various data types, compared to the LAPACK functions (dashed lines), as a function of input data size. The figure indicates that for each data type, the performance of the Julia **larfb** function closely matches that of the LAPACK functions. The latter require separate implementations for each data type.

B. Performance Across Hardware

Figure 1b presents the running time performance of the Julia **unmqr** function (solid lines) in comparison to the LAPACK/CUSOLVER function s(dashed lines). This comparison regards CPU (orange and blue lines) and GPU (purple and green lines) performance for Float64 and Float32. The results demonstrate that, for both hardware types and data types, Julia implementations closely match the performance of LAPACK functions on the CPU and CUSOLVER functions on the GPU, with Julia being slightly faster. Historically, separate implementations were required for CPU and GPU, but the Julia implementation provides an equally performant single-API solution.

IV. Conclusion

We have demonstrated the capability of the Julia language to serve as a general-purpose numerical linear algebra library with the **larfb** and **unmqr** functions as a proof of concept. Both provide a hardware- and data type- agnostic implementation without sacrificing performance. This report describes a work-in-progress to develop a Julia library aiming to implement BLAS/LAPACK functions.

Acknowledgments

This material is based upon work supported by NSF (grant OAC-1835443, SII-2029670, ECCS-2029670, OAC-2103804, PHY-2021825), ARPA-E, U.S. DoE (Award DE-AR0001211, DE-AR0001222), RCN, Equinor (project 308817), AFRL, DAF-AI (Cooperative Agreement FA8750-19-2-1000), FAPESP (grant #2022/07810-7). The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or the FAPESP. The authors would also like to thank KAUST for the support and resources provided. Additionally, authors extend their appreciation to the KAUST Ibn Rushd Fellowship, the Belgian American Educational Foundation.

Vendor and Family	**Intel Ice Lake**	**Intel CascadeLake**
Model	6330	6248
Socket(s) and cores per Socket	2 sockets with 28 cores each	2 sockets with 20 cores each
Clock Speed	2 GHz	2.5GHz
DDR4 memory size	1 TB	384GB
L3 Cache size	84 MiB	27.5MiB
GPU type	NVIDIA A100	NVIDIA V100

TABLE I: Hardware specifications

References

[1] Abdelfattah Ahmad and all. MAGMA: Enabling exascale performance with accelerated BLAS and LAPACK for diverse GPU architectures. *The International Journal of High Performance Computing Applications*, 0(0):10943420241261960, 0.

[2] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, and Viral B. et al. Shah. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.*, oct 2018.

[3] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral Shah, and Alan Edelman. Array Operators Using Multiple Dispatch: A design methodology for array implementations in dynamic languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, page 56–61, New York, NY, USA, 2014. Association for Computing Machinery.

[4] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[5] cppreference. C++26: Basic linear algebra algorithm, 2024.

[6] Simon Danisch, Tim Besard, Valentin Churavy, and et al. GPUArrays.

[7] Leo Fang and et al. NVIDIA Math Libraries for the Python Ecosystem. https://github.com/NVIDIA/nvmath-python, 2024.

[8] Mosè Giordano, Milan Klöwer, and Valentin Churavy. Productivity Meets Performance: Julia on A64FX. In *2022 IEEE International Conference on Cluster Computing*, pages 549–555. IEEE, 2022.

[9] Daniel Reed, Dennis Gannon, and Jack Dongarra. Reinventing High Performance Computing: Challenges and Opportunities, 2022.

[10] Field G. Van Zee and Robert A. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.*, 41(3), jun 2015.