Ginkgo: A high performance numerical linear algebra library

Hartwig Anzt[1, 2], **Terry Cojean**[1], **Yen-Chen Chen**[4], **Goran Flegar**[3], **Fritz Göbel**[1], **Thomas Grützmacher**[1], **Pratik Nayak**[1], **Tobias Ribizel**[1], and **Yu-Hsiang Tsai**[1]

1 Karlsruhe Institute of Technology **2** Innovative Computing Laboratory, University of Tennessee, Knoxville **3** University of Jaume I **4** The University of Tokyo

DOI: 10.21105/joss.02260

Software

- Review
- Repository
- Archive

Editor: Matthew Sottile

Reviewers:

- @adam-m-jcbs
- @zeroset

Submitted: 20 May 2020
Published: 31 August 2020

Summary

Ginkgo is a production-ready sparse linear algebra library for high performance computing on GPU-centric architectures with a high level of performance portability and focuses on software sustainability.

The library focuses on solving sparse linear systems and accommodates a large variety of matrix formats, state-of-the-art iterative (Krylov) solvers and preconditioners, which make the library suitable for a variety of scientific applications. Ginkgo supports many architectures such as multi-threaded CPU, NVIDIA GPUs, and AMD GPUs. The heavy use of modern C++ features simplifies the addition of new executor paradigms and algorithmic functionality without introducing significant performance overhead.

Solving linear systems is usually one of the most computationally and memory intensive aspects of any application. Hence there has been a significant amount of effort in this direction with software libraries such as UMFPACK (Davis, 2004) and CHOLMOD (Chen, Davis, Hager, & Rajamanickam, 2008) for solving linear systems with direct methods and PETSc (Balay et al., 2020), Trilinos ("The Trilinos Project Website," 2020), Eigen (Guennebaud, Jacob, & others, 2010) and many more to solve linear systems with iterative methods. With Ginkgo, we aim to ensure high performance while not compromising portability. Hence, we provide very efficient low level kernels optimized for different architectures and separate these kernels from the algorithms thereby ensuring extensibility and ease of use.

Ginkgo is also a part of the xSDK effort (Bartlett et al., 2017) and available as a Spack (Gamblin et al., 2015) package. xSDK aims to provide infrastructure for and interoperability between a collection of related and complementary software elements to foster rapid and efficient development of scientific applications using High Performance Computing. Within this effort, we provide interoperability with application libraries such as `deal.ii` (Arndt et al., 2019) and `mfem` (Anderson et al., 2020). Ginkgo provides wrappers within these two libraries so that they can take advantage of the features of Ginkgo.

Features

As sparse linear algebra is one of the main focus of Ginkgo, we provide a variety of sparse matrix formats such as COO, CSR, ELL, HYBRID and SELLP along with highly tuned Sparse Matrix Vector product (SpMV) kernels (Anzt et al., 2020). The SpMV kernel is a key building block of virtually all iterative solvers and typically accounts for a significant fraction of the application runtime. Additionally, we also provide high performance conversion routines between the different formats enhancing their flexibility.

Ginkgo provides multiple iterative solvers such as the Krylov subspace methods: Conjugate Gradient (CG), Flexible Conjugate Gradient (FCG), Bi-Conjugate Gradient (BiCG) and its stabilized version (Bi-CGSTAB), Generalized Minimal Residual Method (GMRES) and more generic methods such as Iterative Refinement (IR), which forms the basis of many relaxation methods. Ginkgo also features support for direct and iterative triangular solves within incomplete factorization preconditioners.

Ginkgo features some state-of-the-art general-purpose preconditioners such as the Block Jacobi preconditioner with support for a version which reduces pressure on the memory bandwidth by dynamically adapting the memory precision to the numerical requirements. This approach (Anzt et al., 2019a) has been shown to be very efficient for problems with a block structure. Ginkgo also features highly-parallel incomplete factorization preconditioners such as the ParILU and the ParILUT preconditioners (Anzt et al., 2019b) and Approximate Inverse preconditioners such as the Incomplete Sparse Approximate Inverse preconditioner (Anzt, Huckle, Bräckle, & Dongarra, 2018). A detailed feature overview including design and implementation of all algorithms in Ginkgo with performance results is elaborated in the Ginkgo paper (Anzt et al., 2020).

Software extensibility and sustainability.

Ginkgo is extensible in terms of linear algebra solvers, preconditioners and matrix formats. Basing on modern C++ (C++11 standard), various language features such as data abstraction, generic programming and automatic memory management are leveraged to enhance the performance of the library while still maintaining ease of use and maintenance.

The Ginkgo library is constructed around two principal design concepts. The first one is the class and object-oriented design based in part on linear operators which aims to provide an easy to use interface, common for all the devices and linear algebra objects. This allows users to easily cascade solvers, preconditioners or matrix formats and tailor solvers for their needs in a seamless fashion. The second main design concept consists of the low level device specific kernels. These low level kernels are optimized for the specific device and make use of C++ features such as templates to generate high-performance kernels for a wide variety of parameters.

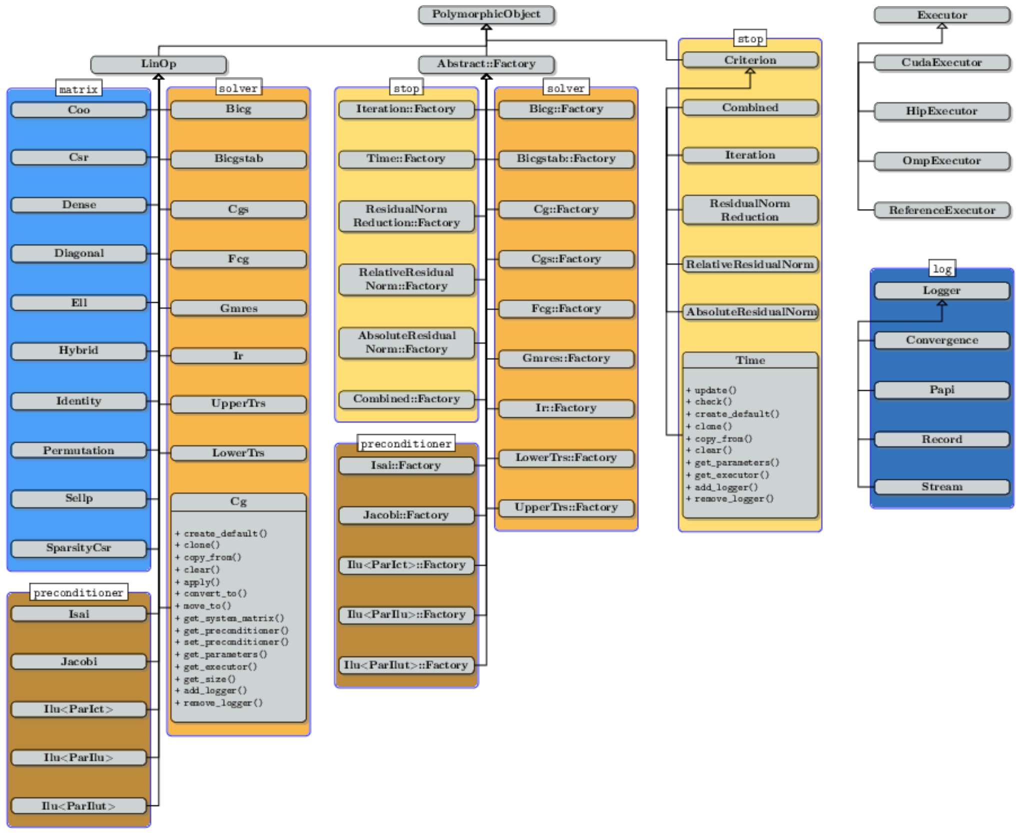

Figure 1: Core architecture of Ginkgo. All solvers, preconditioners and matrix formats are accessible through the same LinOp interface.

Ginkgo adopts a rigorous approach to testing and maintenance. Using continuous integration tools such as Gitlab-CI and Github-Actions, we make sure that the library builds on a range of compilers and environments. To verify that our code quality is of the highest standards we use the Sonarcloud platform ("Sonarcloud - a source code analyzer." 2020), a code analyzer to analyze and report any code smells. To ensure correctness of the code, we use the Google Test library ("Googletest - google testing and mocking framework." 2020) to rigorously test each of the device specific kernels and the core framework of the library.

Performance and Benchmarking

The Ginkgo software is tailored for High Performance Computing and provides high performance implementations on modern manycore architectures. In particular, Ginkgo is competitive with hardware vendor libraries such as hipSPARSE and cuSPARSE (Tsai, Cojean, & Anzt, 2020). The two following figures highlight this fact by comparing the performance of Ginkgo's Hybrid and CSR SpMV kernels against the counterparts from the vendor libraries AMD hipSPARSE and NVIDIA cuSPARSE for all matrices available in the Suite Sparse Matrix Collection.

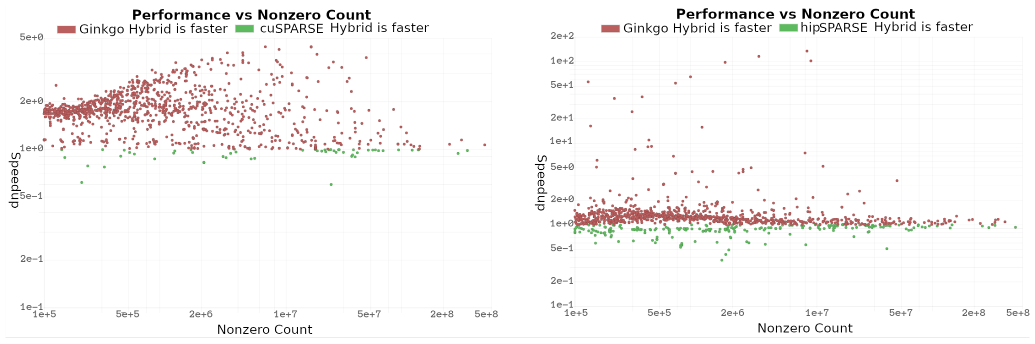

Figure 2: Ginkgo Hybrid spmv provides better performance than (left) cuSPARSE and (right) hipSPARSE

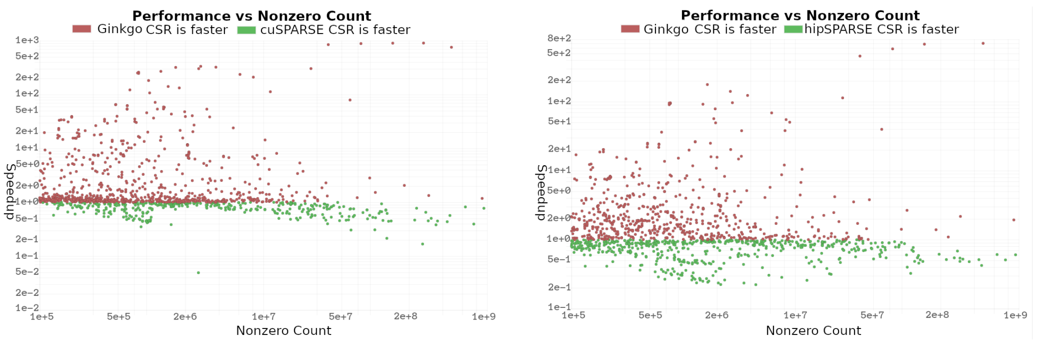

Figure 3: Ginkgo CSR spmv performance is competitive against (left) cuSPARSE and (right) hipSPARSE

Ginkgo provides comprehensive logging facilities both in-house and with interfaces to external libraries such as PAPI (Terpstra, Jagode, You, & Dongarra, 2010). This allows for detailed analysis of the kernels while reducing the intellectual overhead of optimizing the applications.

To enhance reproducibility from a performance perspective, the Ginkgo project aims at providing Continuous Benchmarking on top of its extensive Continuous Integration setup (Anzt, Chen, et al., 2019). To this end, we provide the performance results of our kernel implementations in an open source git repository. A unique feature of Ginkgo is the availability of an interactive webtool, the Ginkgo Performance explorer ("Ginkgo performance explorer," 2020), which can plot results from the aforementioned data repository. Additionally, we have also put in some effort in making benchmarking easier, within the Ginkgo repository using the `rapidjson` ("RapidJSON - a fast JSON parser/generator for c++," 2020) and `gflags` ("gflags - a c++ library that implements commandline flags processing." 2020) libraries to run and generate benchmarking results for a variety of Ginkgo features.

Acknowledgements

Research and software development in Ginkgo received support from the Helmholtz association (Impuls und Vernetzungsfond VH-NG-1241), and the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Cerveny, J., Dobrev, V., et al. (2020). MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*. doi:10.1016/j.camwa.2020.06.009

Anzt, H., Chen, Y.-C., Cojean, T., Dongarra, J., Flegar, G., Nayak, P., Quintana-Ortí, E. S., et al. (2019). Towards continuous benchmarking: An automated performance evaluation framework for high performance software. In *Proceedings of the platform for advanced scientific computing conference*, PASC '19. New York, NY, USA: Association for Computing Machinery. doi:10.1145/3324989.3325719

Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T., et al. (2020). Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *arXiv e-prints*, arXiv:2006.16852. Retrieved from http://arxiv.org/abs/2006.16852

Anzt, H., Cojean, T., Yen-Chen, C., Dongarra, J., Flegar, G., Nayak, P., Tomov, S., et al. (2020). Load-balancing sparse matrix vector product kernels on GPUs. *ACM Trans. Parallel Comput.*, *7*(1). doi:10.1145/3380930

Anzt, H., Dongarra, J., Flegar, G., Higham, N. J., & Quintana-Ortí, E. S. (2019a). Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience*, *31*(6), e4460. doi:10.1002/cpe.4460

Anzt, H., Huckle, T. K., Bräckle, J., & Dongarra, J. (2018). Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Computing*, *71*, 1–22. doi:10.1016/j.parco.2017.10.003

Anzt, H., Ribizel, T., Flegar, G., Chow, E., & Dongarra, J. (2019b). ParILUT - a parallel threshold ILU for GPUs. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 231–241. doi:10.1109/IPDPS.2019.00033

Arndt, D., Bangerth, W., Clevenger, T. C., Davydov, D., Fehling, M., Garcia-Sanchez, D., Harper, G., et al. (2019). The `deal.II` library, version 9.1. *Journal of Numerical Mathematics*. doi:10.1515/jnma-2019-0064

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., et al. (2020). *PETSc users manual* (No. ANL-95/11 - Revision 3.13). Argonne National Laboratory. Retrieved from https://www.mcs.anl.gov/petsc

Bartlett, R., Demeshko, I., Gamblin, T., Hammond, G., Heroux, M., Johnson, J., Klinvex, A., et al. (2017). XSDK foundations: Toward an extreme-scale scientific software development kit. *Supercomputing Frontiers and Innovations*, *4*(1). doi:10.14529/jsfi170104

Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, *35*(3). doi:10.1145/1391989.1391995

Davis, T. A. (2004). Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, *30*(2), 196–199. doi:10.1145/992200.992206

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., & Futral, S. (2015). The spack package manager: Bringing order to HPC software chaos. In *SC '15: Proceedings of the international conference for high performance computing, networking, storage and analysis* (pp. 1–12).

gflags - a c++ library that implements commandline flags processing. (2020).*GitHub repository*. GitHub. Retrieved from https://github.com/gflags/gflags

Ginkgo performance explorer. (2020). Retrieved from https://ginkgo-project.github.io/gpe/

Googletest - google testing and mocking framework. (2020). *GitHub repository*. GitHub. Retrieved from https://github.com/google/googletest

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

RapidJSON - a fast JSON parser/generator for c++. (2020). *GitHub repository*. GitHub. Retrieved from https://github.com/Tencent/rapidjson

Sonarcloud - a source code analyzer. (2020). *GitHub repository*. GitHub. Retrieved from https://sonarcloud.io/

Terpstra, D., Jagode, H., You, H., & Dongarra, J. (2010). Collecting performance data with PAPI-C. In M. S. Müller, M. M. Resch, A. Schulz, & W. E. Nagel (Eds.), *Tools for high performance computing 2009* (pp. 157–173). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-11261-4_11

The Trilinos Project Website. (2020). Retrieved from https://trilinos.github.io

Tsai, Y. M., Cojean, T., & Anzt, H. (2020). Sparse linear algebra on amd and nvidia gpus – the race is on. In P. Sadayappan, B. L. Chamberlain, G. Juckeland, & H. Ltaief (Eds.), *High performance computing* (pp. 309–327). Cham: Springer International Publishing. doi:10.1007/978-3-030-50743-5_16