

Ginkgo - A math library designed to accelerate Exascale Computing Project science applications

Terry Cojean¹ , Pratik Nayak¹ , Tobias Ribizel¹ ,
Natalie Beams² , Yu-Hsiang Mike Tsai¹, Marcel Koch¹,
Fritz Göbel¹, Thomas Grützmacher¹  and Hartwig Anzt^{2,3} 

The International Journal of High
Performance Computing Applications
2024, Vol. 0(0) 1–17
© The Author(s) 2024



Article reuse guidelines:

sagepub.com/journals-permissions
DOI: 10.1177/10943420241268323
journals.sagepub.com/home/hpc



Abstract

Large-scale simulations require efficient computation across the entire computing hierarchy. A challenge of the Exascale Computing Project (ECP) was to reconcile highly heterogeneous hardware with the myriad of applications that were required to run on these supercomputers. Mathematical software forms the backbone of almost all scientific applications, providing efficient abstractions and operations that are crucial to harness the performance of computing systems. GINKGO is one such mathematical software library, nurtured by ECP, providing high-performance, user-friendly, and performance portable interfaces for applications in ECP and beyond. In this paper, we elaborate on GINKGO's philosophy of high-performance software that is sustainable, reproducible, and easy to use. We showcase the wide feature set of solvers and preconditioners available in GINKGO and the central concepts involved in their design. We elaborate on four different ECP software integrations: MFEM, PeleLM + SUNDIALS, XGC, and ExaSGD that use GINKGO to accelerate their science runs. Performance studies of different problems from these applications highlight the effectiveness of GINKGO and the benefits incurred by these ECP applications.

Keywords

Ginkgo, integration, mixed precision, algebraic multigrid, batched solvers, LU factorization, combustion simulations, plasma simulations, power grid simulations

1. Introduction

GINKGO is a high-performance math library aiming at accelerating science. Developed as a software technology product of the US Exascale Computing Project, GINKGO's design acknowledges portability as a central design principle, adheres to high standards for correctness and sustainability, and guides functionality and interface development by the needs of the ECP applications. In this paper, we revisit the design choices made in GINKGO's development, discuss how algorithmic development is guided by application needs, and present some of the major application integrations that enabled better science by providing customized high-performance computational linear algebra functionality. We begin with a general presentation of the GINKGO math library in Section 2 before describing how GINKGO's mixed precision algebraic multigrid (AMG) allows for fast execution of finite element simulations using the MFEM library in Section 4. In Section 5 and Section 6 we discuss the application need for batched iterative solvers, the design of this functionality in

the GINKGO library, and the performance improvement GINKGO's batched iterative functionality brings to combustion simulations and plasma simulations, respectively. In Section 7 we demonstrate how an application-specific sparse direct solver development enables GINKGO to accelerate power grid simulations. We conclude in Section 8 with a summary of the lessons learned during the collaborative development of functionality for the acceleration of scientific applications.

¹Department of SCC, Karlsruhe Institute of Technology, Eggenstein-Leopoldshafen, Germany

²Innovative Computing Laboratory, University of Tennessee at Knoxville, Knoxville, TN, USA

³Technical University of Munich - Campus Heilbronn, Heilbronn, Germany.

Corresponding author:

Hartwig Anzt, Technical University of Munich - Campus Heilbronn, Heilbronn, Germany.

Email: hartwig.anzt@tum.de

2. The Ginkgo math library

GINKGO is a software library developed under the US Exascale Computing Project (ECP) that focuses on the efficient handling of sparse linear systems on GPUs. GINKGO is implemented in modern C++ to accommodate a large number of scientific application codes. The software features multiple backends in hardware-native languages: CUDA for NVIDIA GPUs, HIP for AMD GPUs, and DPC++/SYCL for Intel GPUs. GINKGO contains a set of iterative solvers, including Krylov solvers, algebraic multigrid (AMG), and parallel preconditioners that serve as a valuable toolbox for application codes; see [Figure 1](#) for an overview of functionality supported on different hardware.

FUNCTIONALITY		OMP	CUDA	HIP	DPC++
Basic	SpMV	✓	✓	✓	✓
	SpMM	✓	✓	✓	✓
	SpGeMM	✓	✓	✓	✓
	BICG	✓	✓	✓	✓
	BICGSTAB	✓	✓	✓	✓
	CG	✓	✓	✓	✓
Krylov solvers	CGS	✓	✓	✓	✓
	GCR	✓	✓	✓	✓
	GMRES	✓	✓	✓	✓
	FCG	✓	✓	✓	✓
	FGMRES	✓	✓	✓	✓
	IR	✓	✓	✓	✓
	IDR	✓	✓	✓	✓
	Block-Jacobi	✓	✓	✓	✓
	ILU/IC	✓	✓	✓	✓
	Parallel ILU/IC	✓	✓	✓	✓
Preconditioners	Parallel ILUT/ICT	✓	✓	✓	✓
	ISAI	✓	✓	✓	✓
	Batched BiCGSTAB	✓	✓	✓	✓
	Batched CG	✓	✓	✓	✓
Batched	Batched GMRES	✓	✓	✓	✓
	Batched ILU	✓	✓	✓	✓
	Batched ISAI	✓	✓	✓	✓
	Batched Block-Jacobi	✓	✓	✓	✓
	AMG preconditioner	✓	✓	✓	✓
	AMG solver	✓	✓	✓	✓
AMG	Parallel Graph Match	✓	✓	✓	✓
	Sparse direct	✓	✓	✓	✓
	Symbolic Cholesky	✓	✓	✓	✓
Sparse direct	Numeric Cholesky	✓	✓	✓	✓
	Symbolic LU	✓	✓	✓	✓
	Numeric LU	✓	✓	✓	✓
	Sparse TRSV	✓	✓	✓	✓
	On-Device Matrix Assembly	✓	✓	✓	✓
	MC64/RCM reordering	✓	✓	✓	✓
Utilities	Wrapping user data	✓	✓	✓	✓
	Logging	✓	✓	✓	✓
	PAPI counters	✓	✓	✓	✓
		✓	✓	✓	✓



 MPI Support
  Single-GPU Support

Figure 1. GINKGO functionality and support on different backends.

At every stage of development, GINKGO emphasizes *portability*, *scalability* of performance, *productivity* of its users, and *sustainability* of its codebase. We will now briefly discuss how these concepts have shaped GINKGO.

2.1. Portability

GINKGO was designed at the start of the US Exascale Computing Project (ECP) and acknowledged ECP’s plan to deploy supercomputers featuring hardware accelerators from different vendors. For this reason, the portability of functionality is a central design principle of the GINKGO ecosystem. While there exist different levels of portability across architectures, GINKGO aims to provide not only functional portability but also performance portability (Cojean et al. (2022)). To achieve this, instead of relying on a portability layer like Kokkos¹, RAJA² or SYCL³ (Godoy et al. (2023)), GINKGO uses a backend model that lifts portability to the software design level. The idea is to rely on a set of kernels implemented using vendor-native programming models, for each hardware target (Cojean et al. (2022)). These kernels are then used to implement the high-level algorithms. This is reflected in [Figure 2](#) visualizing the backend model used in the GINKGO library design⁴.

At the start of the ECP project, GINKGO was mostly focused on CPU multicore functionality and NVIDIA GPUs, reflecting the state of the existing exascale machines. Due to the GINKGO design, portability to other vendor ecosystems was quickly achieved. The GINKGO HIP backend targeting AMD GPUs was completed with release 1.2.0 (July 2020). The 1.4.0 minor release (Aug 2021) brought most of the GINKGO functionality to the oneAPI ecosystem, enabling Intel-GPU and CPU execution, excluding preconditioners. The full oneAPI and SYCL support were completed in release 1.5.0. GINKGO currently features backends for GPUs from AMD, NVIDIA, and Intel, and an OpenMP backend for efficient algorithm execution on multicore CPUs. Additionally, there is a “reference” backend that contains sequential CPU implementations of the kernels. The reference backend serves two purposes: to check the correctness of the abstract algorithm implementations and to enable rigorous unit tests of the highly-tuned parallel backends for multicore and GPU architectures. The user controls the algorithm execution on the distinct backends by creating an “executor” that determines on which hardware the code is to be executed. The kernel selection is then orchestrated by runtime polymorphism. This enables user productivity as changing the executor’s target in a single place suffices to move the complete execution to another hardware.

2.2. Performance and scalability

As an ECP software product, GINKGO has the clear goal of scaling to hundreds and thousands of GPUs. While the

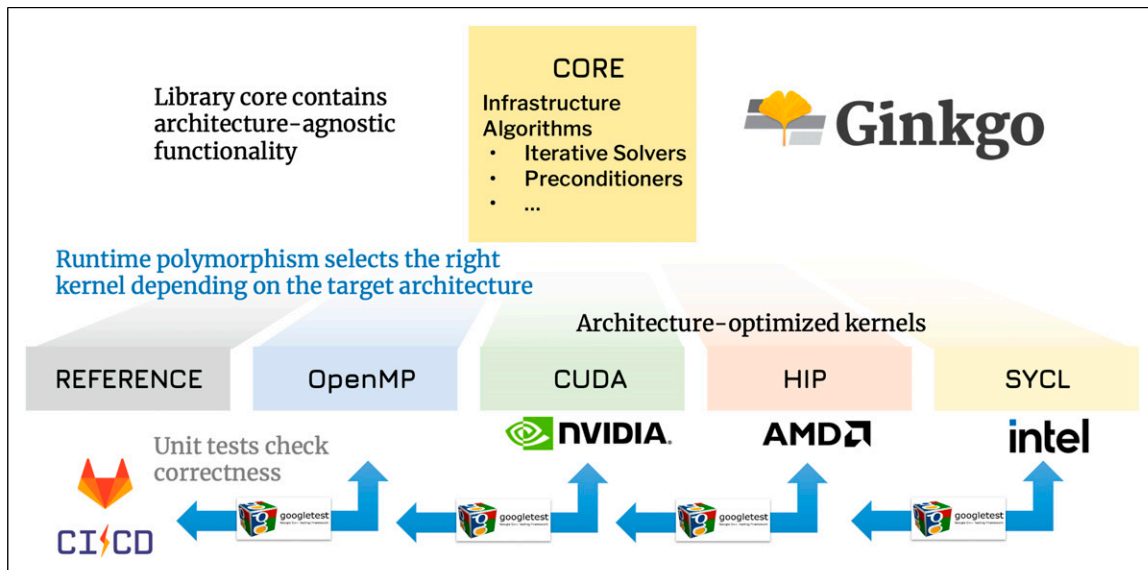


Figure 2. Overview of the GINKGO library design using the backend model for platform portability Cojean et al. (2022). High-level algorithms are contained in the library core and composed of algorithm-specific kernels coded for the different hardware backends.

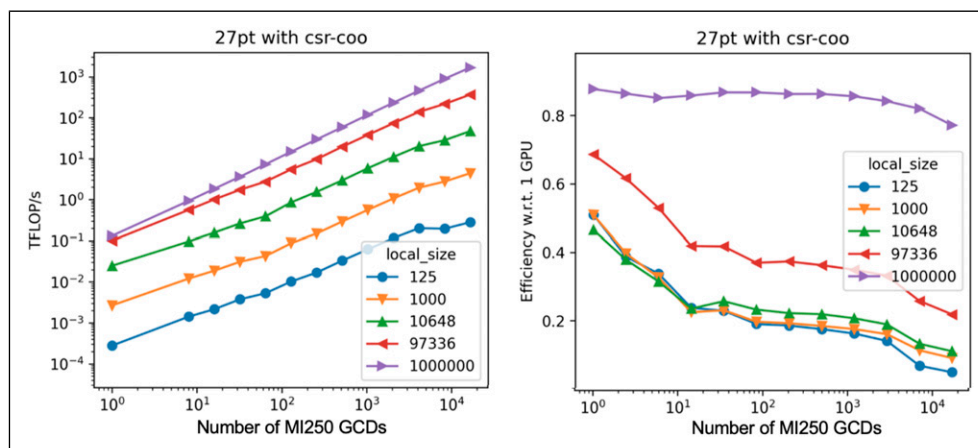


Figure 3. Weak scaling test for GINKGO's SpMV functionality on the Frontier supercomputer. Test matrix: 3D Laplace discretized with a 27-pt stencil. The size indicates the number of rows in the local matrix. Left: Performance in GFLOP/s; Right: Parallel efficiency using 1 GCD as reference.

backend model allows for hardware portability, multi-GPU and multi-node parallelization are enabled by using MPI on top of the backend model. The efficiency of this approach is demonstrated in Figures 3 and 4 visualizing weak and strong scalability of GINKGO's functionality on the Frontier supercomputer. Figure 3 (left) reveals that the performance on a single GPU is very poor if the local problem is of small size, but reaches about 100 GFLOP/s when the local problem has a size of 1 million rows. For this setting, on 16,000 GCDs (2,000 nodes of Frontier), GINKGO exceeds 1 PFLOP/s, which indicates almost perfect weak scalability. For the smaller problems, the communication becomes the bottleneck, and the parallel

efficiency drops significantly (see the right-hand side in Figure 3). The same effect can be observed in the strong scaling experiments in Figure 4: for all solvers, the runtime decreases linearly only as long as the local problem remains large. The inferior strong scalability characteristics of GMRES compared to CG and BiCGStab are expected given that GMRES is based on long recurrences and has to orthogonalize against all Krylov vectors of the restart cycle. In contrast, CG and BiCGStab only orthogonalize against the previous Krylov basis vector. The orthogonalization involves a significant amount of communication and synchronization, penalizing parallel efficiency.

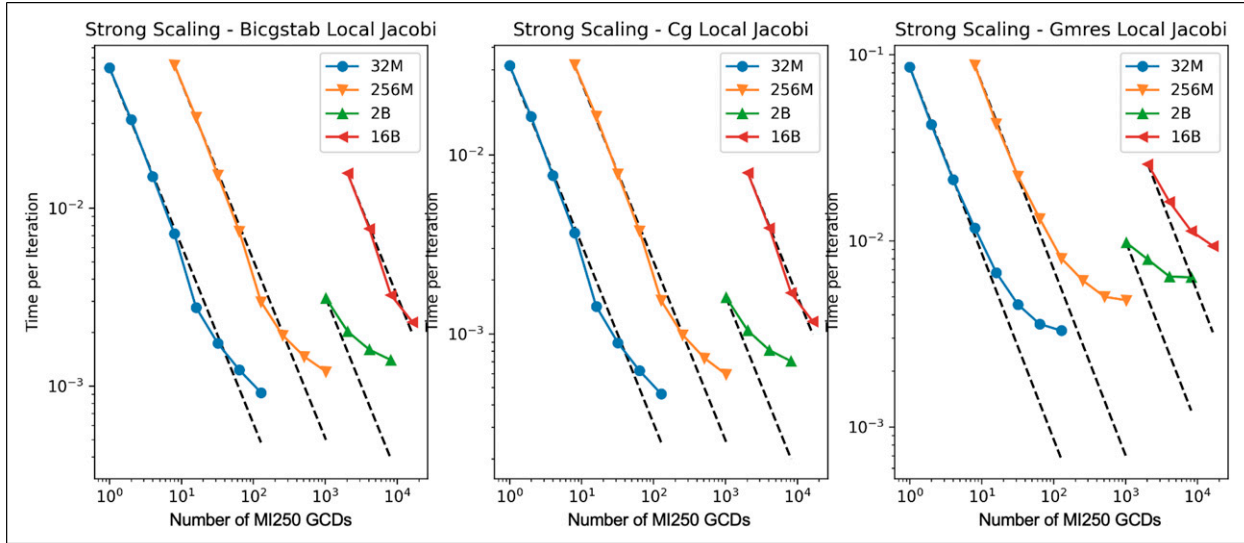


Figure 4. Strong scaling test for GINKGO’s iterative solver functionality for 3D Laplace problems of different sizes discretized with a 27-pt stencil on the Frontier supercomputer. The size indicates the number of rows in the global matrix. GMRES uses Classical Gram-Schmidt with reorthogonalization and a restart of 50.

2.3. Productivity

To enhance the productivity of domain scientists, GINKGO exposes an application programming interface (API) that allows users to easily combine different components for the iterative solution of linear systems: solvers, matrix formats, preconditioners, etc. The API enables running distinct iterative solvers and enhancing the solvers with different types of preconditioners. A preconditioner can be a matrix, an abstract operator, or even another solver. Furthermore, the system matrix does not need to be stored explicitly in memory but can be available only as a function that is applied to a vector to compute a matrix-vector product (matrix-free). The objective of providing a clean and easy-to-use interface mandates that all these special cases are uniformly realized in the API.

A concept that accommodates productivity and flexibility for the user and extensibility for the library developers is the abstraction of all solver-, matrix-, and preconditioner functionality as *linear operators* (Anzt et al. (2022)):

1. The major operation that an iterative solver performs on the system matrix A is the multiplication with a vector (realized as a Matrix-Vector product, or MV). This operation can be viewed as the application of the induced linear operator $L_A: z \mapsto Az$. In some cases, multiplication with the transpose is also needed, which is yet another application of a linear operator $L_{A^T}: z \mapsto A^T z$.
2. The solver itself solves a system $Ax = b$, which is the application of the linear operator $S_A: b \mapsto A^{-1}b (= x)$. Here, the term “solver” is not used to denote a function f that takes A and b as inputs and produces x ,

but instead a function with the system matrix A already fixed (i.e., $S_A = f(A, \cdot)$).

3. The application of the preconditioner M , as in $v = M^{-1}u$, can be viewed as the application of the linear operator $P_M: u \mapsto M^{-1}u (= v)$.

We note that in the context of numerical computations with finite precision arithmetic, the term “linear operator” should be understood loosely (Anzt et al. (2022)). In fact, none of the functionality strictly satisfy the linearity definition of the linear operator: $L(ax + \beta y) = aL(x) + \beta L(y)$, where a, β are scalars and x, y denote vectors. Instead, they are just approximations of the linear operators that satisfy $L(ax + \beta y) = aL(x) + \beta L(y) + E$, where the error term $E = E(L, a, \beta, x, y)$ is the result of rounding errors introduced by storing non-representable values in floating-point format, rounding errors introduced by finite-precision floating-point arithmetic, instability and inaccuracy of the method used to apply the linear operator to a vector, and inexact operator application, e.g. only few iterations of an iterative linear solver.

Acknowledging the inconsistency with the mathematical concept, the linear operator abstraction in GINKGO enables a high level of productivity for both the domain scientists using GINKGO and the GINKGO software developers. For example, using the *LinOp* abstraction, a user of the GINKGO library can easily call an iterative solver from the GINKGO library, combine it with different preconditioners available in GINKGO, or even pass a user-defined preconditioner as an opaque linear operator to the GINKGO iterative solver. At the same time, from the developer perspective, an iterative solver can be implemented via references to other *LinOps* that represent

the system matrix and the preconditioner (Anzt et al. (2022)). The solver does not have to be aware of the type of the matrix or the preconditioner — it is sufficient to know that they are both conformal with the *LinOp* interface. This means that the same implementation of the solver can be configured to integrate various preconditioners and matrices. Furthermore, the linear operator abstraction can also be used to compose “cascaded” solvers where the preconditioner can be replaced by another, less accurate solver, or even to create matrix-free methods by supplying a specialized operator as the system matrix, without explicitly storing the matrix.

2.3.1. Inspecting Ginkgo’s internals: The Ginkgo loggers

In addition to a general interface designed to allow flexibility in configuring and using GINKGO functionality, GINKGO provides users with easy access to internal execution data such as iterative solver convergence, iterative residuals, algorithm runtime, memory consumption, etc. These metrics can be relevant for the library user in the optimization of algorithm selection, algorithm configuration, or execution policy. The concept GINKGO employs for this purpose is the GINKGO logger, which allows the recording of information about GINKGO’s execution and exposes it to the user. For ease of use, the event logging tools provide different output formats and allow the usage of multiple loggers at once. As with the rest of GINKGO, this tool is designed to be controllable, extensible, and as lightweight as possible. To offer support for all those capacities, the Logger infrastructure follows the visitor and observer design patterns (Gamma et al. (1994)). This design implies a minimal overhead to the function execution (Anzt et al. (2022)).

The most popular already-available logger concepts in GINKGO are:

- the *Stream* logger, which logs the events to a stream (e.g., file, standard output, etc.);
- the *Record* logger, which stores the events in a structure that has a history of all received events that the user can retrieve at any moment;
- the *Convergence* logger is a simple mechanism that stores the relative residual norm and number of iterations of the solver on convergence;
- the *PerformanceHint* logger tracks common performance issues in user-code, such as common cross-executor copies between the same pointers;
- the *ProfilerHook* logger allows to annotate GINKGO’s internal functionality execution on several

common profilers: TAU, VTune, NSightSystems (NVTX) and rocPROF(ROCTX); and

- the *PAPI SDE* logger uses the PAPI Software Defined Events backend Jagode et al. (2019) in order to enable access to GINKGO’s internal information through the PAPI interface and tools.

In particular, the PAPI SDE logger brings convenience to the user as it allows the combined collection of GINKGO SDEs and software-defined events exposed by the scientific application. Further, it allows for the straightforward mapping of the software-side information to the hardware-side information such as clock speed, processor load, occupancy, etc, that PAPI provides. The PAPI-SDE integration is optionally available through GINKGO’s spack package.

2.4. Sustainability

GINKGO is available as open source code on Github⁵ under a permissive BSD 3-clause license. Overall, GINKGO has a heavy focus on sustainability and correctness, and all kernels implemented in any of the backends are complemented with unit tests checking the correctness against the reference backend implementation. GINKGO uses a sustainable software development lifecycle requiring two code reviews for any code addition and a Continuous Integration/Continuous Benchmarking framework for the automatic validation of compilability and correctness of the code Anzt et al. (2019). The CI framework executes pipelines on more than 30 different hardware/software configurations including server-grade GPUs from different vendors and features code quality checks like Sonarqube, sanitizers, cuda-memcheck, clang-tidy, and others. GINKGO has comprehensive unit and functional test coverage. GINKGO is part of the extreme-scale Scientific Software Development Kit (xSDK^{Vert}) and the Extreme Scale Scientific Software Stack (E4S^{**}). GINKGO is available as a Spack package and smoke tests are run through the spack CI. A well-defined release schedule anticipates a release every 6 months. The sustainability of the GINKGO library is also visible in the extensibility of the backend model: new backends can be added without changes to the existing functionality, and backends can be eliminated if the hardware is discontinued. The only core parts of GINKGO that need to be adapted to do so are the executor and operation classes, as well as related utilities.

GINKGO bundles a comprehensive benchmarking suite and compares against vendor solutions when available. In addition, all GINKGO internal data can be inspected and

exported thanks to a logger system providing hooks at critical code points; see Section 2.3.1.

3. Related work

The Exascale computing project supported and nurtured many mathematical software libraries, enabling the development of new algorithms and their high-performance implementations. There exist many state-of-the-art software libraries that provide similar functionality to GINKGO. PETSc (PET (2021)) and Trilinos (Tri (2021)) provide sparse linear solvers and preconditioners, and have been used in various applications. For sparse-direct solvers, libraries such as SuperLU (Li and Demmel (2003)), MUMPS (Amestoy et al. (2001)), CHOLMOD (Chen et al. (2008)) etc. Contain high-performing implementations for CPU and distributed systems, as well as GPUs (NVIDIA/AMD) with SuperLU. MAGMA (Dongarra et al. (2014)) and most LAPACK (Blackford and Dongarra (1991)) implementations provide implementations of dense direct solvers for batched routines, with MAGMA focusing on GPUs.

Within ECP, GINKGO was a part of the software technologies stack, particularly a part of the mathematical software stack, aiming to provide high-performance numerical linear algebra functionality, which usually forms the computational bottleneck in most scientific applications. GINKGO's novelty lies in its software design providing flexibility and extensibility, the utilization of vendor-provided programming models for optimal performance on each backend, and application-friendly utilities. Put together, this helps reduce the overhead of integrating GINKGO's linear solver functionality for users while enabling high performance for each hardware backend.

GINKGO's design has been elaborated in Anzt et al. (2022), where we discuss the software infrastructure, including detailed performance results for the building blocks (SpMV) and the various solvers available in GINKGO. Each following section in this paper considers applications

from ECP, which were accelerated using GINKGO functionality. We focus on novel algorithmic and implementation developments in these sections and provide concrete cases where the applications obtained speedups in their real-world runs. Therefore, each section contains relevant papers that provide more detailed information about these developments.

5. Mixed precision algebraic multigrid accelerating MFEM simulations

We will now turn our attention to examples of GINKGO's impact on ECP applications. In this section, we begin with how the development of a mixed precision algebraic multigrid method has enabled faster finite element simulations with the MFEM library. A more detailed discussion and result presentation is available in Tsai et al. (2022, 2023) and Tsai (2024).

Algebraic multigrid (AMG) is a popular choice for solving or preconditioning linear problems originating from finite element methods. Geometric multigrid (GMG) uses information about the underlying geometric mesh directly, but an AMG solver is constructed directly from the sparse system matrix. Multigrid methods build a hierarchy of consecutively coarser grids and compute error correction terms on the coarser grids to improve the solution on finer grids. Specifically, the residual is restricted from a fine grid to a coarser grid, where we obtain an error correction that is prolonged back to the finer grid to update the solution approximation. These correction computations frequently entail a few iterations of an iterative method, called a "smoother" because it acts to smooth the high-frequency errors on the scale of that grid; the coarsest grid, which is much smaller than the original problem, may use a direct solver or an iterative method.

AMG's concept of building a sequence of consecutively coarser grids and computing error correction terms on the coarser grids (representing problem approximations on a coarser grid) motivates research on the use of

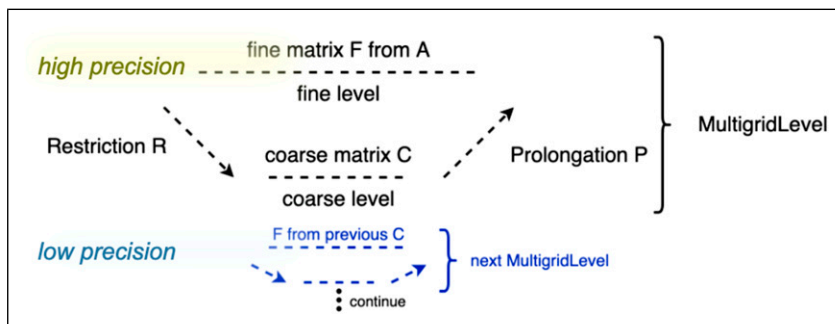


Figure 5. Mixed precision AMG handling only the first multigrid level in high precision and subsequent levels in lower precision.

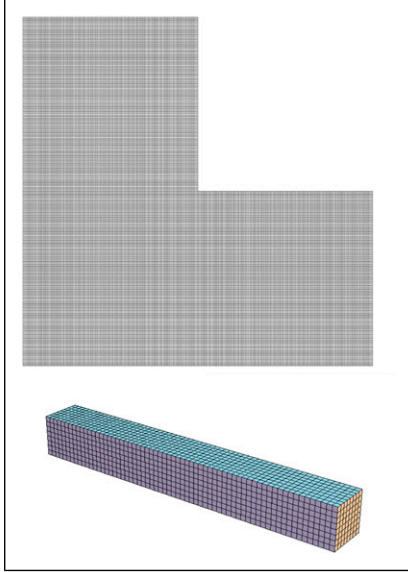


Figure 6. Meshes used for MFEM diffusion experiments. Top: L-shape mesh with seven levels of uniform refinement (49,152 elements); Bottom: Beam mesh with 3 levels of uniform refinement (4,096 elements).

low precision for solving the coarse grid problem representations. This idea is visualized in Figure 5. The hope is that the faster execution of the computations in lower precision accelerates the overall multigrid cycle without negatively impacting the multigrid quality. It is important to note that AMG is generally memory bound and that thus potential runtime saving comes from the reduced data transfer volume, not from faster arithmetic.

GINKGO features an AMG implementation that can be competitive with the AmgX library Naumov et al. (2015) developed by NVIDIA Tsai et al. (2022). GINKGO falls short in terms of supporting the same variety of coarsening strategies, currently only supporting parallel graph match (PGM). At the same time, GINKGO’s AMG allows for more flexibility in terms of choosing the precision formats for the distinct multigrid levels individually, with the conversion between the formats handled on-the-fly in the restriction and prolongation operations Tsai et al. (2022).

MFEM Anderson et al. (2021); is a popular open-source finite element library with broad support for high-order meshes and basis functions, among many other features. MFEM’s “example 1” demonstrates the solution of a standard diffusion problem $-\nabla \cdot (c \nabla u) = 1$, where c is a given coefficient. MFEM provides many sample meshes for testing; we select two, which are shown in Figure 6. For the “L-shape” mesh (top), a constant coefficient of $c = 1$ is used, while the “beam” mesh (bottom) uses a piecewise constant coefficient with a jump from 1 to 0.1 at the midpoint of the length of the beam. All our tests use standard tensor-product basis functions on the Legendre-

Table 1. Characteristics for the selected MFEM discretizations.

Problem	Size	Nonzero elements
Beam (-o2 -l3)	37,281	21,67,425
Beam (-o3 -l3)	120,625	14,070,001
Beam (-o4 -l3)	279,873	57,251,713
Beam (-o3 -l4)	924,385	111,573,601
L-shape (-o3 -l7)	443,905	11,066,881
L-shape (-o3 -l8)	1,772,545	44,252,161
L-shape (-o4 -l7)	788,481	28,323,841
L-shape (-o4 -l8)	3,149,825	113,270,785

Gauss-Lobatto nodes, MFEM’s default choices for quadrature points based on the order of basis functions, and homogeneous Dirichlet boundary conditions.

To evaluate the performance of the mixed precision AMG we developed in GINKGO, and to compare against NVIDIA’s AmgX, we use AMG as a preconditioner within MFEM’s CG solver, with one V-cycle application for each iteration of CG. Intending to provide a fair comparison, we match the parameter settings as closely as possible for AmgX and GINKGO as we detail in Tsai et al. (2022). We built on MFEM’s existing GINKGO wrappers to use GINKGO AMG within MFEM, and AmgX preconditioning support is provided through MFEM’s AmgXSolver class. For more meaningful insight, in the following performance evaluation, we consider different settings modulating the *orders* of basis functions ($-o$) and *levels* of mesh refinement ($-l$). We end up with a set of 8 test problems with their characteristics listed in Table 1. Increasing the order of basis functions increases both the problem size and the dependencies between unknowns; increasing the refinement of the mesh increases the problem size while retaining sparsity. We set the stopping criterion to a maximum of 300 iterations and implicit relative residual norm reduction of 10^{-12} . We here revisit a subset of the results and the discussion we present in Tsai et al. (2022) and refer the reader to the original paper for more details.

Initially, we analyze the runtime of a single CG iteration with the AMG V-cycle preconditioner, without considering the full iterative solver and convergence. In Figure 7 we compare the runtime for the different AMG solvers (“NVIDIA AmgX (DP)”, “GINKGO AMG (DP)” and “GINKGO AMG (MP)”) for the beam mesh (top) and the L-shape mesh (bottom). A first observation is that for all discretizations, GINKGO’s AMG is competitive or outperforms NVIDIA’s AmgX in the runtime-per-iteration metric. A second observation is that GINKGO’s mixed precision (MP) AMG outperforms the double precision (DP) variant.

We next consider the quality of the AMG preconditioner in terms of the CG convergence. For this, in Table 2, we list the iteration counts and overall runtime for the different CG/preconditioner configurations. We first focus on four



Figure 7. Runtime of one AMG-preconditioned CG iteration on the V100 GPU for MFEM's example I for the beam mesh (top) and the L-shape mesh (bottom).

Table 2. MFEM **beam** (top) and **L-shape** (bottom) examples using MFEM's AMG-preconditioned CG solver. GINKGO's AMG is executed in IEEE double precision (DP) and mixed precision mode (MP) using IEEE single precision on the subsequent levels. Target architecture is the NVIDIA V100 GPU.

Problem	NVIDIA AmgX (DP)		GINKGO AMG (DP)		GINKGO AMG (MP)	
	Runtime [ms]	#iter	Runtime [ms]	#iter	Runtime [ms]	#iter
Beam						
-o 2 -l 3	20.71	15	20.27	15	19.96	15
-o 3 -l 3	52.94	20	39.93	21	39.56	21
-o 4 -l 3	155.47	26	128.69	27	120.41	27
-o 3 -l 4	329.68	29	294.68	29	270.39	29
L-shape						
-o 3 -l 7	242.27	93	178.02	93	170.08	94
-o 3 -l 8	1211.38	180	1033.96	173	943.27	177
-o 4 -l 8	3452.91	251	3044.24	236	2722.63	237
-o 4 -l 7	551.99	129	407.27	122	366.99	120

discretizations for the **beam** geometry. The iteration counts of the AMG-preconditioned CG solver are generally consistent, and using GINKGO's AMG in mixed-precision mode does not, in this case, increase the CG iteration count above the double precision setting. The CG preconditioned with NVIDIA's AmgX preconditioner sometimes needs one less iteration to converge, but as we have seen, the AmgX preconditioner application is more expensive per iteration than GINKGO's AMG preconditioner: see Figure 7 (top), which shows the average execution time per one CG iteration, with GINKGO's AMG being approximately 20%–40% faster than NVIDIA's AmgX for the three larger problems.

This performance combined with nearly identical iteration counts results in GINKGO's AMG consistently outperforming NVIDIA's AmgX for this test case – with the mixed-precision configuration (double precision for the finest level and single precision for the coarse levels) increasing the performance advantage.

Compared to the **beam** geometry, the **L-shape** geometry is numerically more challenging due to its re-entrant corner. We use the same experiment settings and report the results in the bottom part of Table 2. Here, the trend of the AmgX-preconditioned CG requiring fewer iterations is reversed, as in this case, GINKGO's AMG enables faster convergence.

Combined with the already-observed faster preconditioner application per iteration (see bottom of Figure 7), GINKGO’s AMG offers attractive runtime savings over AmgX for all discretizations of the **L-shape** geometry. The runtime savings increase when using GINKGO’s AMG in mixed-precision mode. For example, for the “-o 4 -l 7” discretization, preconditioning CG with GINKGO’s mixed-precision AMG allows us to solve the problem 1.5× faster than when using NVIDIA’s AmgX; see Table 2 (bottom).

We conclude that the mixed precision AMG offers attractive runtime savings over the double precision AMG for finite element simulations. This holds also for other problems as it is demonstrated in Tsai et al. (2022). The seamless usage of GINKGO functionality in MFEM applications ensures high productivity when accelerating MFEM-based fluid flow simulations with GINKGO’s solver and preconditioner functionality.

6. Batched iterative solvers in Pele combustion simulations

The PeleLM application code (Nonaka et al. (2018)) is designed to simulate reactive flow in the Low Mach number regime, and these flow simulations are dynamic and evolve in both time and space. Like other hydrodynamics simulation frameworks, the PeleLM operator splits the reactions from the hydrodynamics and thus requires the solution of many

independent chemical reaction systems of ordinary differential equations (ODEs). PeleLM uses the SUNDIALS (Hindmarsh (2002)) software library to solve for the reaction ODEs in each cell. The resulting linearized systems all share the same sparsity pattern, and thousands of independent systems with the same sparsity pattern have to be solved in parallel. Batched functionality designed for the data-parallel processing of many small problems forms a natural candidate for this scenario. While there exists batched functionality for dense problems, including linear system solvers, sparse problems have remained unexplored by math library developers. This is because sparse problems of small size can be handled with existing dense direct solver functionality by treating the sparse system as a dense problem.

Investigating the properties of the linearized systems in PeleLM simulations, however, reveals that they could benefit from batched sparse solvers: they are of small or moderate dimension, are sparse, typically have low condition numbers which allows for fast convergence, and no exact solution is required – a rough solution approximation for the linear systems is sufficient for the convergence of the ODE solver.

In response to the potential benefits for the Pele hydrodynamics simulations, we developed batched iterative solvers in GINKGO. The conceptual idea is to launch a set of GPU thread blocks with each thread block handling the iterative solution of one linear system of the batch, see Figure 8. It is important to acknowledge that the iteration process is asynchronous and the different matrix characteristics allow for some iteration processes to reach the required solution

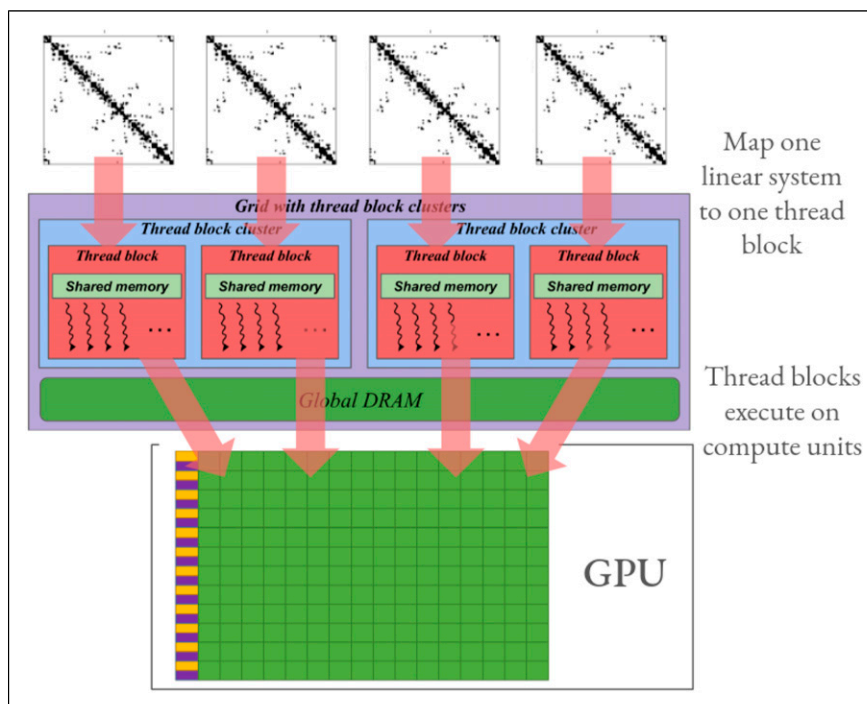


Figure 8. Mapping batched linear system solutions to a GPU.

accuracy and thus terminate earlier than others. While a detailed discussion about the design and implementation of GINKGO’s batched iterative solvers can be found in [Nayak \(2023\)](#) and [Aggarwal et al. \(2021\)](#), we highlight the performance benefits the functionality brings to the simulation codes here.

To showcase the integration and benefits of GINKGO’s batched solvers with Pele, we use the PelePhysics [NREL \(2023c\)](#) repository, which contains physics databases and implementation harnesses that are used within other Pele codes such as PeleC [NREL \(2023a\)](#) and PeleLMeX [NREL \(2023b\)](#). It allows the user to efficiently manage the different chemistry and transport models, which allows for state-of-the-art simulation. A central component of the PelePhysics code is the evolution of the reactive flow in time. PelePhysics relies on the CVODE module from the SUNDIALS suite, which implements several time integrators. In many cases, these reactive flow simulations are numerically stiff, which requires an implicit time-stepping scheme to enforce stability. Assembling these reaction mechanisms so that they can be integrated over time with an implicit time-stepping scheme gives us linear systems that need to be solved at each time step.

To demonstrate the performance advantages GINKGO’s batched iterative solvers render over existing strategies, we compare three options that are available for the solution of the batch of linear systems in PelePhysics through SUNDIALS-CVODE: MAGMA’s batched direct solver [Haidar et al. \(2015\)](#), GINKGO’s **BatchBicgstab** solver based on short recurrences and preconditioned with a scalar Jacobi, and GINKGO’s **BatchGmres** solver based on long recurrences (longer than the iteration count) and also enhanced with a scalar Jacobi preconditioner. We run our experiments on 2 nodes of Frontier (16 GCDs of MI250X) and compare the runtimes of the non-linear solve in the PelePhysics test case.

We focus on two reaction mechanisms, the dodecane_lu mechanism, giving us batched linear systems with matrix size (54×54) each, and the dodecane_lu_qss mechanism with matrices of size (35×35). The matrix characteristics listed in [Table 3](#) reveal that these matrices – though handled by GINKGO’s batched iterative solvers with the CSR format – are relatively dense, which may suggest that a dense direct solver could be a method of choice. However, the performance results in [Figure 9](#) prove the superiority of

Table 3. Key matrix and sparsity characteristics of the benchmark reaction mechanisms extracted from PeleLM chemistry simulation runs.

Problem	Size	Non-zeros (A)	Non-zeros (L + U)
dodecane_lu_qss	35	1042 (85%)	1164 (95%)
dodecane_lu	54	2332 (80%)	2754 (94%)

the batched iterative solvers. Here, we visualize the performance of GINKGO’s batched iterative solvers as speedup over MAGMA’s dense direct solver for the complete non-linear loop within the PeleLM application. For the smaller mechanism (top), **BatchGmres** performs slightly better than **BatchBicgstab**, but both the batched iterative solvers outperform the batched LU solver from MAGMA. For the larger mechanism (bottom), we see that the **BatchBicgstab** is the best choice with an average speedup of around 2x over MAGMA.

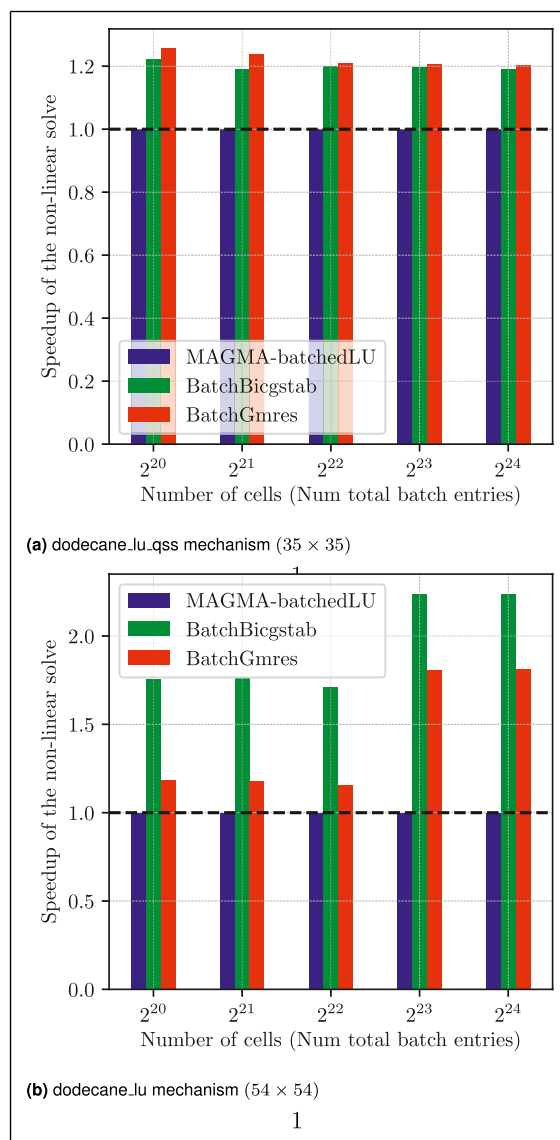


Figure 9. Speedup for the entire non-linear solver with GINKGO’s batched solvers over MAGMA’s batched dense direct solver for 2 mechanisms on 16 GCDs of MI250X on Frontier.

7. Batched iterative solvers in XGC plasma simulations

The batched iterative solvers in GINKGO were specifically designed and deployed to accelerate hydrodynamics simulations. With the functionality providing attractive runtime benefits to the solution of batched sparse linear systems, a different science application realized that the batched iterative solvers fit their requirements as well.

xgc is a 5D full-function gyrokinetic particle-in-cell (PIC) application code (Hager et al. (2016); Ku et al. (2009)) that numerically simulates fusion edge plasmas. For accurate high-resolution and high-fidelity, these simulations must scale to tens of thousands of nodes, each containing multiple GPUs. In the XGC simulations, a nonlinear collision operator is required to model edge plasmas accurately. Therefore, xgc employs a nonlinear Fokker-Planck-Landau operator in the 2D guiding-center velocity space for multiple particle species. Coulomb collisions between particles in the plasma have been identified as a bottleneck in xgc. An implicit time integration method is employed, and the Picard method is used for the nonlinear solver. At each configuration space grid node, we must solve the nonlinear operator on the 2D velocity space grid to evolve the different species through time. In each cell of the velocity grid, we must first solve for the collisions involving a linear solution. As the collision operator is based on a stencil, each linear solve in each cell shares the same sparsity pattern. Hence, to compute the evolution of these species, multiple independent linear systems need to be solved, all sharing a sparsity pattern. As is detailed in Nayak (2023), the batched iterative solver functionality developed in GINKGO is a perfect candidate for this task, efficiently reducing the time to solve for the overall simulation.

The collision kernel of the xgc simulation code is based on MPI for multiple CPU nodes and uses Kokkos (Carter Edwards et al. (2014)) to offload to GPUs as well as utilize OpenMP for intra-node CPU parallelism.

The collision operator in XGC is a fully non-linear multi-species Fokker-Planck-Landau operator. The velocity space is discretized for each mesh vertex with a velocity grid. Using a 2D nine-point stencil produces matrices of the $O(1000)$ rows on each mesh vertex with nine nonzeros per row, sharing a sparsity pattern across all mesh vertices. An outer Picard iteration is used to resolve the non-linear operator, and within each Picard iteration, we need to solve many independent linear systems. Due to the typically larger nature of the matrices, traditional dense direct solvers are unsuitable. XGC used the LAPACK banded solver, `dgbsv`, which assigned one linear system to one CPU core and parallelized the available linear systems over the number of CPU cores available (64 for both Perlmutter and Frontier). Within a stand-alone proxy app integrated with GINKGO's batched solvers, as shown in our previous work Kashi et al. (2022, 2023), the batched iterative

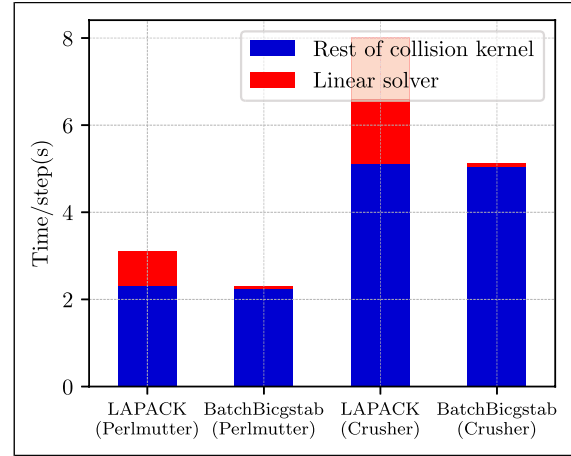


Figure 10. LAPACK v/s GINKGO batched BiCGStab solving the DIII-D Tokamak Electromagnetic test case on 32 nodes of NERSC Perlmutter with 128 NVIDIA A100 GPUs (left) and 32 nodes of OLCF Crusher with 128 AMD MI250X (right).

solvers can render significant performance benefits for the XGC application.

Figure 10 presents a runtime comparison between LAPACK's band LU solver and GINKGO's batched iterative solver for the DIII-D Tokamak Electromagnetic test case. The test case contains 432,000 mesh vertices, with each vertex needing to solve for two species. The velocity grid is of size (33×39) leading to matrices of size (1278×1278) , each with 9 nonzeros per row as before in the proxy app. In total, there are 22.4 million particles per species per GPU. This leads to 864,000 linear solver calls for every call to the collision operator solve. For more details on matrix properties and batched solver implementation, we refer the reader to our previous work in Kashi et al. (2023). The test case runs for 20 time steps, with the collisions being calculated every other step. This is a full-simulation test that compares the execution on the OLCF Frontier TDS (Crusher) supercomputer (right) and the NERSC Perlmutter supercomputer (left). We observe that using the **BatchBicgstab** solver reduces the linear solver time by about 90% on both systems. We acknowledge that the linear solver only accounts for part of the collision operator cost. Nevertheless, as a plug-in replacement for LAPACK's direct solver, GINKGO's **BatchBicgstab** succeeds in serving as a faster, more flexible, and platform-portable alternative.

8. Sparse direct solvers in ExaSGD simulations

With increased energy needs, more variable generation added to the grid, less predictable weather patterns, and ever-increasing cyber threats, the computational cost of power grid simulations is increasing rapidly. AC Optimal Power Flow (ACOPF) analysis O'Neill et al. (2012); Frank et al. (2012) is

Table 4. Characteristics of the three test networks specifying the number of buses, generators, and lines. The specifics of the linear system for each of these networks are given in terms of the matrix size (N) and number of non-zeros (nnz).

Grid	Buses	Generators	Lines	$N(K_k)$	nnz (K_k) (M)
Northeast US	25 K	4.8 K	32.3 K	108 K	1.19
East US	70 K	10.4 K	88.2 K	296 K	3.20
West & East US	82 K	13.4 K	104.1 K	340 K	3.73

an important tool for simulating near- and long-term power grid use. With growing energy needs an increasing amount of variable energy sources added to the power grid, these simulations have become computationally more demanding, pushing the limits of existing tools Świrydowicz et al. (2024). While new architectures like high-performance GPUs provide the computational power to handle the computational complexity, mathematical and computational methods typically used for economic dispatch are designed for CPU systems and perform poorly on GPUs. This is particularly true when solving the underlying linear systems Świrydowicz et al. (2022), which typically make up more than half of the overall computational cost. The system characteristics, specifically the sparsity pattern and the extremely high condition numbers, especially in later iterations of the non-linear optimization solver, make GPU-based iterative methods unsuitable for the solution process. Even robust preconditioners often fall short in a comparison against CPU-based direct solvers employing the combination of matrix factorization and forward/backward substitution. While there have been several efforts to develop GPU-accelerated sparse linear solvers that are effective for computations in the power systems domain, including electromagnetic transient Dinkelbach et al. (2021); Razik et al. (2019) and power flow simulations D'orto et al. (2021), there is far less reporting on linear solvers suitable for ACOPT analysis due to the challenging properties of linear systems there. The use of dense direct methods has been evaluated Rakai and Rosehart (2014); Abhyankar et al. (2021), but the computational complexity of dense direct solvers growing $O(N^3)$ and the memory complexity growing $O(N^2)$ with the linear system size N makes this strategy infeasible for larger grid models.

In this section, we present the sparse direct solver functionality we developed in GINKGO to accelerate the HiOp optimization engine Petra et al. (2018) used in the power grid simulations. The section combines content and in Ribizel and Anzt (2023) and Świrydowicz et al. (2024), on the sparse Cholesky factorization and the rendered performance boost to the HiOp optimization engine, respectively. For additional details, we refer the interested reader to these publications.

The structure of linear systems arising in the interior-point optimization algorithm used for ACOPT analysis is very sparse and does not profit from the commonly used supervariable agglomeration performance-wise, making state-of-the-art supernodal direct solvers inefficient. Instead,

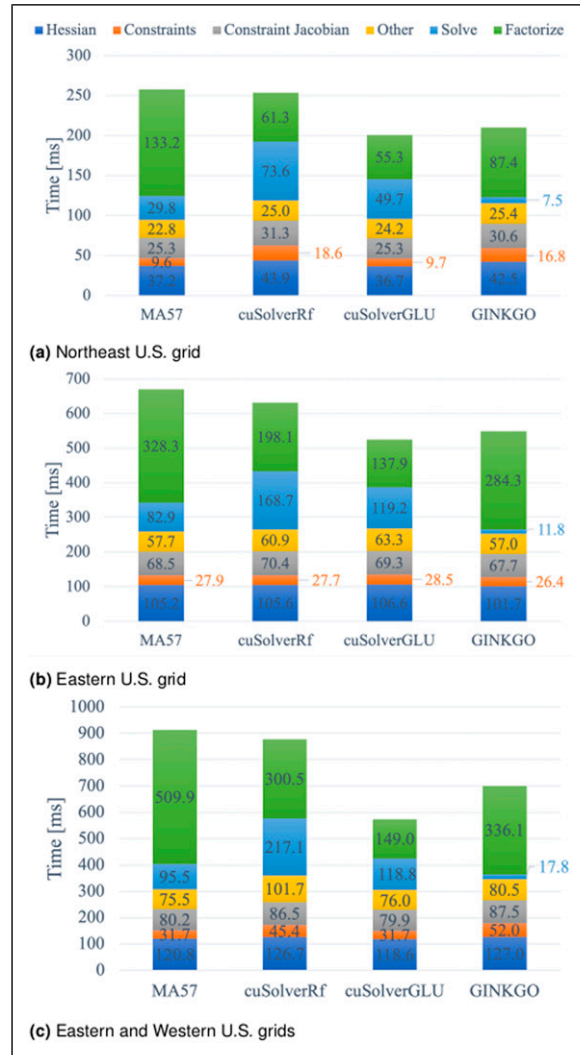


Figure 11. Comparison of the average computational cost per optimization solver step when different linear solvers are used for ACOPT on OLCF Summit with a breakdown in terms of most expensive operations Świrydowicz et al. (2024). The cost of the first step, which is executed on CPU, is accounted for in the averages.

GINKGO provides a lightweight LU and Cholesky factorization and associated triangular solvers to enable the efficient solution of generic sparse systems without the use of supernode detection and other structural properties.

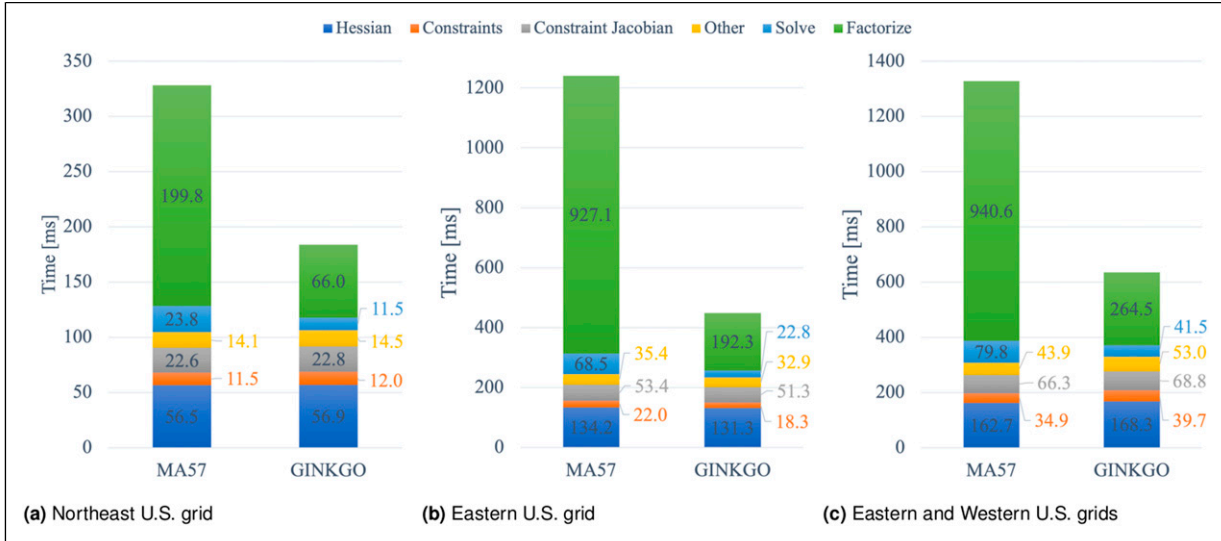


Figure 12. Comparison of the average computational cost per optimization solver step when different linear solvers are used for ACOPF on OLCF Crusher with a breakdown in terms of most expensive operations Świrydowicz et al. (2024). The cost of the first step, which is executed on CPU, is accounted for in the averages.

The direct solver framework relies on a CPU-GPU hybrid symbolic factorization for problems with symmetric sparsity pattern Ribizel and Anzt (2023) and a CPU symbolic factorization for general unsymmetric problems. The symbolic factors are then used in the numerical phase to compute the values of the lower and upper triangular factors in a warp-per-row parallelization with lightweight scheduling based on the dependency DAG encoded in the lower factor. The lower and upper factors are finally used to compute the solution to the linear system using forward- and backward substitution. Since the system matrices become increasingly ill-conditioned during the optimizer iterations, and sparse direct solvers commonly suffer from large storage requirements due to fill-in, GINKGO uses the stability-enhancing MC64 reordering Duff and Koster (2001) in combination with the fill-in reducing AMD reordering Amestoy et al. (2004) to preprocess the system.

For the performance evaluation of the developed GINKGO sparse direct solver, we use it as a direct solver in the HiOp optimization solver. For the experimental evaluation, we consider three power grid problems that are listed along with some key characteristics in Table 4. We compare GINKGO’s sparse direct solver against the state-of-the-art open source sparse direct solvers MA57, cuSolverRf, and cuSolverGLU.

MA57 Duff (2004) is commonly used in commercial and open-source tools when solving alternating current optimal power flow (ACOPF) and similar optimization problems. When using MA57, symbolic factorization is performed only once for all systems with the same sparsity pattern. The profiling results for the solution of the KKT system for these three networks with MA57 linear solver show that solving

the linear system takes up more than 60% of the overall ACOPF cost on an IBM Power 9 central processing unit (CPU). CuSolverRf is NVIDIA’s sparse direct solver, cuSolverGLU is an undocumented but publicly released alternative Świrydowicz et al. (2024). Both routines are bound to NVIDIA architectures and cannot execute on AMD GPUs.

We acknowledge that the overall cost of ACOPF simulations derives as an interplay of sparse matrix factorization cost, triangular solver cost, refactorization cost, and the number of optimization steps. We here only focus on the overall simulation runtime and refer the interested reader to the original paper Świrydowicz et al. (2024) for additional information.

In Figures 11 and 12 we report the comparison and detailed performance breakdown of the HiOp optimization solver step into its components on the Summit and Crusher supercomputers. The averages were obtained by normalizing the total computational time for each component by the number of optimization steps to allow comparison between the approaches, and these averages account for the cost of the first factorization performed on CPU Świrydowicz et al. (2024).

We first focus on the evaluation of the benchmark results obtained on the Summit supercomputer equipped with NVIDIA V100 GPUs. MA57 is executed on the IBM Power9 CPU architecture of Summit. The results in Figure 11 reveal that the GPU solvers outperform the MA57 baseline implementation for all test cases. CuSolverRf is 10 – 30% slower overall than cuSolverGLU. In terms of overall ACOPF compute time, using cuSolverGLU and GINKGO on V100 graphical

processing unit (GPU) leads to 1.3 – 1.4× and 1.05 – 1.3× faster solution, respectively, compared to the CPU baseline with MA57. The runtime breakdown reveals that the performance advantage of the `cuSolverGLU` is primarily driven by a faster factorization (which is for the combined Eastern and Western U.S. grid about 3.4× faster than MA57). The faster factorization compensates for the slower triangular solves: The `cuSolverGLU` triangular solves are about 40% slower than the CPU counterpart. For the `GINKGO` GPU solver, the story is different: though still faster than the MA57 code, the speedup achieved with the `GINKGO` factorization is smaller. On the other hand, the triangular solve in `GINKGO` is faster, mainly because it does not call iterative refinement. We note that it is impossible to combine the `cuSolverGLU` factorization with the `GINKGO` sparse triangular solves as the `cuSolverGLU` does not provide access to the triangular factors Świrydowicz et al. (2024).

We now turn to the performance results in Figure 12 on the Oak Ridge Leadership Computing Facility (OLCF) Crusher system featuring AMD MI250 GPUs. Overall, ACOPF is overall 1.8 – 2.4× faster when using the `GINKGO` linear solver on the AMD MI250X GPU instead of MA57 on the AMD EPYC 7A53 CPU⁶. The runtime breakdown reveals that the performance superiority comes from both a faster factorization (3 – 4.8× speedup) and faster triangular solves (1.9 – 3× speedup). Comparing `GINKGO`'s linear solver (triangular solve and factorization) performance on the MI250X GPU and the NVIDIA V100 GPU, we notice that executing `GINKGO` on the newer MI250 GPU is 20% – 40% faster than on the NVIDIA V100 GPU Świrydowicz et al. (2024).

This demonstrates that the application-specific sparse direct solver development for GPUs is successful in terms of runtime saving when executing power grid simulations on GPU-accelerated supercomputers. In a cross-institutional focus effort, specialized GPU-based direct solvers were designed that meet the requirements of ACOPF simulations. `GINKGO` now contains production-ready sparse direct solvers that allow users to run high-performance ACOPF simulations on GPU systems and to easily migrate applications between systems featuring GPUs from different vendors.

9. Summary

The `GINKGO` library accelerates several science applications on the US Department of Energy's supercomputers. Choosing performance portability as a key software design strategy set `GINKGO` on a path to success that allowed for execution on upcoming hardware architecture earlier than portability layers have implemented support. On an interpersonal level, the scientific application users benefited significantly from close communication with the development team and a willingness to develop functionality geared toward their applications. The batched iterative solvers are

an example of functionality that had previously received no attention from the computational linear algebra community, and only in-depth discussions with application scientists revealed the potential of this functionality for domain scientists. The sustainable software development cycle featuring rigorous unit testing and code quality checks has proven to be extremely valuable in the reduction of software bugs. Finally, it is important to see that money does not write software, but individuals do, and individuals have their own life plans, dreams, and needs. Recognizing and acknowledging every individual who contributes to a scientific software stack, participates in collaborative research, or contributes as an external advisor is key to creating a productive and trustful environment. We consider it unlikely that these findings are all specific and exclusive to `GINKGO`'s development and integration activities and encourage software projects to learn from the design and strategies we described in this paper.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the funding from 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.

ORCID iDs

Terry Cojean  <https://orcid.org/0000-0002-1560-921X>
 Pratik Nayak  <https://orcid.org/0000-0002-7961-1159>
 Tobias Ribizel  <https://orcid.org/0000-0003-3023-1849>
 Natalie Beams  <https://orcid.org/0000-0001-6060-4082>
 Thomas Grützmacher  <https://orcid.org/0000-0001-9346-2981>
 Hartwig Anzt  <https://orcid.org/0000-0003-2177-952X>

Notes

1. <https://kokkos.github.io>
2. <https://raja.readthedocs.io>
3. <https://www.khronos.org/sycl/>
4. `GINKGO` uses DPC++/SYCL as one of its backends, which is a portability layer in itself.
5. <https://github.com/ginkgo-project/ginkgo>
6. <https://e4s-project.github.io>

References

- Abhyankar S, Peles S, Rutherford R, et al. (2021) Evaluation of AC optimal power flow on graphical processing units 2021 IEEE Power & Energy Society General Meeting

- (PESGM), Washington, DC, USA, 26-29 July 2021, 01–05. DOI: [10.1109/PESGM46819.2021](https://doi.org/10.1109/PESGM46819.2021).
- Aggarwal I, Kashi A, Nayak P, et al. (2021) Batched sparse iterative solvers for computational chemistry simulations on GPUs 2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), St. Louis, MN, USA, 19-19 November 2021, 35–43. <https://ieeexplore.ieee.org/document/9652814>.
- Amestoy PR, Duff IS, L'Excellent JY, et al. (2001) MUMPS: a general purpose distributed memory sparse solver. In: Sørevik T, Manne F, Gebremedhin AH, et al. (eds) *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Berlin, Heidelberg: Springer, 121–130. DOI: [10.1007/3-540-70734-4_16](https://doi.org/10.1007/3-540-70734-4_16).
- Amestoy PR, Davis TA and Duff IS (2004) Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software* 30(3): 381–388. DOI: [10.1145/1024074.1024081](https://doi.org/10.1145/1024074.1024081).
- Anderson R, Andrej J, Barker A, et al. (2021) MFEM: a modular finite element methods library. *Computers & Mathematics with Applications* 81: 42–74. DOI: [10.1016/j.camwa.2020.06.009](https://doi.org/10.1016/j.camwa.2020.06.009).
- Anzt H, Chen YC, Cojean T, et al (2019) *Towards continuous benchmarking: an automated performance evaluation framework for high performance software* Proceedings of the Platform for Advanced Scientific Computing Conference, Zurich, Switzerland, June 12-14, 2019, 1–11.
- Anzt H, Cojean T, Flegar G, et al. (2022) Ginkgo: a modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software* 48(1): 1–33.
- Blackford S and Dongarra J (1991) *LAPACK Working Note 41 Installation Guide for LAPACK*. Knoxville, TN: Department of Computer Science, University of Tennessee.
- Carter Edwards H, Trott CR and Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74(12): 3202–3216. DOI: [10.1016/j.jpdc.2014.07.003](https://doi.org/10.1016/j.jpdc.2014.07.003).
- Chen Y, Davis TA, Hager WW, et al. (2008) Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software* 35(3): 1–14. DOI: [10.1145/1391989](https://doi.org/10.1145/1391989).
- Cojean T, Tsai YHM and Anzt H (2022) Ginkgo—a math library designed for platform portability. *Parallel Computing* 111: 102902.
- Dinkelbach J, Schumacher L, Razik L, et al. (2021) Factorisation path based refactorisation for high-performance LU decomposition in real-time power system simulation. *Energies* 14(23): 7989.
- Dongarra J, Gates M, Haidar A, et al. (2014) Accelerating numerical dense linear algebra calculations with GPUs. In: Kindratenko V (ed) *Numerical Computations with GPUs*. Cham: Springer International Publishing, 3–28. DOI: [10.1007/978-3-319-06548-9_1](https://doi.org/10.1007/978-3-319-06548-9_1).
- Duff IS (2004) MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software* 30(2): 118–144.
- Duff IS and Koster J (2001) On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications* 22(4): 973–996. DOI: [10.1137/S0895479899358443](https://doi.org/10.1137/S0895479899358443).
- D'orto M, Sjöblom S, Chien LS, et al. (2021) Comparing different approaches for solving large scale power-flow problems with the Newton-Raphson method. *IEEE Access* 9: 56604–56615.
- Frank S and Rebennack S (2012) A primer on optimal power flow: theory, formulation, and practical examples. In: *Technical Report 14*. Golden, CO: Colorado School of Mines.
- Gamma E, Helm R, Johnson R, et al. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition. Boston: Addison-Wesley Professional.
- Godoy WF, Valero-Lara P, Dettling TE, et al. (2023) *Evaluating Performance and Portability of High-Level Programming Models: Julia, Python/Numba, and Kokkos on Exascale Nodes*. Knoxville, TN: Department of Computer Science, University of Tennessee. DOI: [10.48550/arXiv.2303.06195](https://doi.org/10.48550/arXiv.2303.06195).
- Hager R, Yoon ES, Ku S, et al. (2016) A fully non-linear multi-species Fokker–Planck–Landau collision operator for simulation of fusion plasma. *Journal of Computational Physics* 315: 644–660. DOI: [10.1016/j.jcp.2016.03.064](https://doi.org/10.1016/j.jcp.2016.03.064) <https://www.sciencedirect.com/science/article/pii/S0021999116300298>
- Haidar A, Dong TT, Tomov S, et al. (2015) A framework for batched and GPU-resident factorization algorithms applied to block householder transformations. In: Kunkel JM and Ludwig T (eds) *High Performance Computing, Lecture Notes in Computer Science*. Cham: Springer International Publishing, 31–47. DOI: [10.1007/978-3-319-20119-1_3](https://doi.org/10.1007/978-3-319-20119-1_3).
- Hindmarsh AC (2002) SUNDIALS: suite of nonlinear/differential/algebraic equation solvers. Technical Report UCRL-JC-149711, Livermore, CA (United States): Lawrence Livermore National Lab. (LLNL). <https://www.osti.gov/biblio/15002968>.
- Jagode H, Danalis A, Anzt H, et al. (2019) Papi software-defined events for in-depth performance analysis. *The International Journal of High Performance Computing Applications* 33(6): 1113–1127.
- Kashi A, Nayak P, Kulkarni D, et al. (2022) Batched sparse iterative solvers on GPU for the collision operator for fusion plasma simulations. In: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 30 May 2022 - 03 June 2022. pp. 157–167. DOI: [10.1109/IPDPS53621.2022.00024](https://doi.org/10.1109/IPDPS53621.2022.00024).
- Kashi A, Nayak P, Kulkarni D, et al. (2023) Integrating batched sparse iterative solvers for the collision operator in fusion plasma simulations on GPUs. *Journal of Parallel and Distributed Computing* 178. DOI: [10.1016/j.jpdc.2023.03.012](https://doi.org/10.1016/j.jpdc.2023.03.012) <https://www.sciencedirect.com/science/article/pii/S0743731523000540>

- Ku S, Chang CS and Diamond PH (2009) Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion* 49(11): 115021. DOI: [10.1088/0029-5515/49/11/115021](https://doi.org/10.1088/0029-5515/49/11/115021).
- Li XS and Demmel JW (2003) SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software* 29(2): 110–140. DOI: [10.1145/779359.779361](https://doi.org/10.1145/779359.779361).
- Naumov M, Arsaev M, Castonguay P, et al. (2015) AmgX: a library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing* 37(5): S602–S626.
- Nayak PV (2023) *Synchronization-free algorithms for exascale and beyond : a study of asynchronous and batched iterative methods*. Karlsruhe: Karlsruhe Institute of Technology. PhD Thesis. DOI: [10.5445/IR/1000165437](https://doi.org/10.5445/IR/1000165437).
- Nonaka A, Day MS and Bell JB (2018) A conservative, thermodynamically consistent numerical approach for low Mach number combustion. Part I: single-level integration. *Combustion Theory and Modelling* 22(1): 156–184.
- NREL (2023a) PeleC: an adaptive mesh refinement solver for compressible reacting flows. <https://github.com/AMReX-Combustion/PeleC>. Original-date:2018-10-19T18:28:51Z.
- NREL (2023b) *PeleLMex: an adaptive mesh hydrodynamics simulation code for low Mach number reacting flows without level sub-cycling*. Golden, CO: NREL.
- NREL (2023c) PelePhysics: a collection of physics databases and implementation code for use with the Pele suite of codes. <https://github.com/AMReX-Combustion/PelePhysics>. Original-date:2018-10-18T22:58:09Z.
- O'Neill RP, Castillo A and Cain MB (2012) The IV formulation and linear approximations of the AC optimal power flow problem (OPF Paper 2). <https://www.ferc.gov/industries/electric/industry/market-planning/opf-papers/acopf-2-iv-linearization.pdf>.
- Petra CG, Chiang N and Wang J (2018) HiOp – user guide. In: *Technical Report LLNL-SM-743591, Center for Applied Scientific Computing*. Livermore, CA: Lawrence Livermore National Laboratory.
- PETSc/Tao: Home page (2021) PETSc/Tao: home page. <https://www.mcs.anl.gov/petsc/index.html>.
- Rakai L and Rosehart W (2014) *GPU-accelerated solutions to optimal power flow problems 2014* 47th Hawaii International Conference on System Sciences, Waikoloa, Hawaii, USA, Jan 06 - Jan 09, 2014. IEEE, 2511–2516.
- Razik L, Schumacher L, Monti A, et al. (2019) A comparative analysis of LU decomposition methods for power system simulations 2019 IEEE Milan PowerTech. Milan, Italy, 23–27 June 2019, IEEE, 1–6.
- Ribizel T and Anzt H (2023) Parallel symbolic cholesky factorization *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23*. New York, NY, USA: Association for Computing Machinery, 1721–1727. DOI: [10.1145/3624062](https://doi.org/10.1145/3624062).
- Świrydowicz K, Darve E, Jones W, et al. (2022) Linear solvers for power grid optimization problems: a review of GPU-accelerated linear solvers. *Parallel Computing* 111: 102870.
- Świrydowicz K, Koukpaizan N, Ribizel T, et al. (2024) Gpu-resident sparse direct linear solvers for alternating current optimal power flow analysis. *International Journal of Electrical Power & Energy Systems* 155: 109517. <https://www.sciencedirect.com/science/article/pii/S0142061523005744>
- Trilinos software (2021) Trilinos software. <https://trilinos.github.io/>.
- Tsai YH (2024) *Portable mixed precision algebraic multigrid on high performance GPUs*. Karlsruhe: Karlsruhe Institute of Technology. PhD Thesis. DOI: [10.5445/IR/1000168914](https://doi.org/10.5445/IR/1000168914).
- Tsai YHM, Beams N and Anzt H (2022) Mixed precision algebraic multigrid on gpus *International Conference on Parallel Processing and Applied Mathematics*. Berlin: Springer, 113–125.
- Tsai YHM, Beams N and Anzt H (2023) Three-precision algebraic multigrid on gpus. *Future Generation Computer Systems* 149: 280–293.

Author biographies

Terry Cojean is a research scientist and deputy group leader at the Karlsruhe Institute of Technology. He obtained his PhD in Computer Science from the University of Bordeaux. His interests include task-based runtime systems, efficient scheduling strategies, software sustainability, software portability, and GPU computing. He is a core developer of the Ginkgo software.

Pratik Nayak is a research scientist at Karlsruhe Institute of Technology. He obtained his PhD in Computer Science from Karlsruhe Institute of Technology, Germany and his Masters from TU Delft, Netherlands. His research interests include High performance computing, asynchronous methods, distributed algorithms and numerical linear algebra. He is also interested in sustainable software development and is a core developer of the Ginkgo software.

Tobias Ribizel is a research scientist and PhD student at Karlsruhe Institute of Technology. He obtained his Masters in Mathematics and Computer Science from Karlsruhe Institute of Technology. His interests include graph algorithms, sparse algorithms, direct solvers and efficient GPU algorithms. He is also interested in research software engineering and is a core developer of the Ginkgo software.

Natalie Beams is a research assistant professor in the Innovative Computing Laboratory at the University of Tennessee.

She holds a Ph.D. in Theoretical and Applied Mechanics from the University of Illinois at Urbana-Champaign. Her research interests include numerical methods for PDEs, scientific computing, and high-performance computing.

Yu-Hsiang Tsai is a research scientist at Karlsruhe Institute of Technology. He obtained his PhD in Computer Science from Karlsruhe Institute of Technology, Germany. His research interests include HPC, portability, multigrid methods and mixed-precision algorithms. He is a core developer of the Ginkgo software.

Marcel Koch is a research scientist at the Karlsruhe Institute of Technology. He obtained his Ph.D. in Mathematics from the University of Münster. He is interested in scientific computing, high-performance computing, and research software engineering. He is also a member of the core developer team of Ginkgo.

Fritz Goebel is a research scientist and PhD student at Karlsruhe Institute of Technology. He obtained his Masters in Mathematics from Karlsruhe Institute of Technology. His interests include distributed preconditioners, direct solvers and mixed-precision solvers. He is a developer of the Ginkgo software.

Thomas Grützmacher is a Ph.D. student at the Karlsruhe Institute of Technology and a researcher at the Technical University of Munich. He received a bachelor's degree from the KIT in 2015 with an emphasis on software development for mobile computing and IoT. In 2018 he completed his Master's studies with an emphasis on High-Performance Computing and unconventional precision formats. Thomas Grützmacher's research focus is on designing a modular precision ecosystem. He also is among the core developer team of the Ginkgo open-source library.

Hartwig Anzt is the Chair of Computational Mathematics at the TUM School of Computation, Information and Technology of the Technical University of Munich (TUM) Campus Heilbronn. He also holds a Research Associate Professor position at the Innovative Computing Lab (ICL) at the University of Tennessee (UTK). Hartwig Anzt holds a PhD in applied mathematics from the Karlsruhe Institute of Technology (KIT) and specializes in iterative methods and preconditioning techniques for the next generation hardware architectures. He also has a long track record of high-quality development. He is author of the MAGMA-sparse open source software package and managing lead of the Ginkgo math software library.