Ginkgo and its Application in Solving Large Sparse Linear Systems

Elisei

June 29, 2025

Abstract

This paper explores the use of the Ginkgo high-performance linear algebra library for solving large sparse linear systems of the form $Ax = b$. We analyze key components within Ginkgo's Linear Operator framework, including various matrix formats (CSR, ELL, SELL-P) for efficient Sparse Matrix-Vector Multiplication (SpMV), factorization techniques, preconditioners (Jacobi, ILU, AMG), and iterative solvers (CG, GMRES, BiCGSTAB). Through performance analysis and code examples, we demonstrate how these components achieve performance portability across heterogeneous computing architectures, with particular emphasis on GPU acceleration. Our analysis shows that sophisticated preconditioners like ParILUT can provide significant performance improvements over standard approaches.

1 Introduction

Linear systems of the form $Ax = b$, where A is a matrix, (x) is the unknown vector, and (b) is the right-hand side vector, are ubiquitous in numerous scientific and engineering disciplines. They arise in simulations of physical phenomena, data analysis, optimization problems, and many other areas, including computational fluid dynamics, geosciences, and large-scale data analysis [1, Chapter 1], [2, Chapter 1], [3]. Solving these systems efficiently, especially when the matrix A is large and sparse, presents significant computational challenges [1, Chapter 2], [4]. The challenges with sparse matrices on modern hardware like GPUs include irregular memory accesses and lower arithmetic intensity, necessitating innovative algorithms and data structures [4].

The landscape of high-performance linear algebra has evolved significantly with changes in hardware. For decades, the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) standards were the cornerstones of scientific computing, providing optimized and portable interfaces for CPUs [2, Chapter 1]. The stated goal of BLAS was to create a standardized set of core routines to be used across all types of linear algebra software [5, p. 308]. However, the paradigm shifted dramatically with the rise of GPU computing. This trend was massively accelerated by the demands of deep learning,

where researchers noted that training large models was only practical on parallel hardware; for their groundbreaking AlexNet model, this process took "five to six days... on two GTX 580 3GB GPUs" [6, Section 3]. This necessity drove the development of vendor-specific libraries such as NVIDIA's cuSPARSE [7] and AMD's rocALUTION [8]. While highly optimized, these libraries introduced the problem of "vendor lock-in," forcing developers to maintain multiple codebases.

Long before the widespread adoption of GPUs, the challenge of creating portable and scalable scientific software for parallel CPU architectures was addressed by frameworks like PETSc (Portable, Extensible Toolkit for Scientific Computation) [9, Section 1]. PETSc provided a powerful, unified abstraction layer that became a standard for large-scale HPC applications. However, the modern challenge has shifted from pure CPU parallelism to managing heterogeneous systems. While established frameworks like PETSc are actively evolving to incorporate GPU support [10], this often involves retrofitting complex, existing designs. In contrast, modern C++ libraries like Ginkgo have emerged with performance portability across heterogeneous hardware as a primary design goal [11, Section 2]. Ginkgo provides a unified, high-level API that abstracts away hardware differences, allowing developers to write their code once and run it efficiently on various architectures. The library's design also extends to large-scale systems by supporting multi-node execution with MPI [12], aiming to provide maximum performance without compromising portability [11, 13].

2 Core Design Concepts in Ginkgo

At the heart of Ginkgo's design is the concept of a Linear Operator (`gko::LinOp`) [11, 14, 15]. This object-oriented design principle abstracts all functionality, including matrices, solvers, preconditioners, and factorizations, under a common interface. This approach allows users to easily combine and tailor different components to construct effective solution strategies [11, 14]. For example, any preconditioner can be combined with any compatible solver, because both are derived from the same LinOp base class.

Complementing the Linear Operator is the `Executor` concept. An executor specifies where the computations will be performed (e.g., on a multi-core CPU via `gko::OmpExecutor`, or an NVIDIA GPU via `gko::CudaExecutor`). This design separates the algorithm logic (the "what", defined by LinOp objects) from the hardware-specific implementation (the "where", managed by the Executor). This separation is fundamental to how Ginkgo achieves performance portability, as it allows the same high-level algorithm code to be executed on different hardware backends without modification.

3 Sparse Matrix-Vector Multiplication (SpMV) and Matrix Formats

Solving linear systems $Ax = b$ using iterative methods relies fundamentally on the repeated computation of the matrix-vector product Ax. When the matrix A is sparse, meaning it has a significantly larger number of zero elements than non-zero elements, this operation is referred to as Sparse Matrix-Vector Multiplication (SpMV). SpMV is often the most time-consuming kernel within iterative solvers, making its efficient implementation crucial for overall performance [1, Chapter 3], [4]. This is primarily because the operation is memory-bound: its performance is limited by memory bandwidth rather than computational speed due to its low arithmetic intensity and irregular memory access patterns.

The inherent sparsity of matrices necessitates specialized storage formats to avoid storing the large number of zero elements, thereby reducing memory consumption and enabling faster operations by only processing the non-zero entries. The choice of matrix format significantly impacts the performance of SpMV, as different formats exhibit varying memory access patterns and computational characteristics, which interact differently with underlying hardware architectures such as CPUs and GPUs [1, Chapter 2], [4].

A widely adopted format for storing general sparse matrices is the Compressed Sparse Row (CSR) format, implemented in Ginkgo as `gko::matrix::Csr`. In the CSR format, the sparse matrix is represented by three one-dimensional arrays:

1. `values`: Stores the non-zero elements of the matrix row by row.
2. `col_indices`: Stores the column index for each element in the `values` array.
3. `row_pointers`: Stores the index in the `values` (and `col_indices`) array where each row starts. The length of this array is the number of rows plus one, with the last element indicating the total number of non-zero elements.

For a concrete example of the CSR format, consider the following sparse matrix A:

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 4.0 \\ 0.0 & 0.0 & 5.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 \end{pmatrix}$$

This 4×4 matrix has 7 non-zero elements. Its representation in CSR format would be:

- `values`: $[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]$
- `col_indices`: $[0, 2, 1, 3, 2, 0, 3]$
- `row_pointers`: $[0, 2, 4, 5, 7]$

The SpMV operation $y = Ax$ using CSR can be expressed conceptually as:

$$y_i = \sum_{j=\text{row\_pointers}[i]}^{\text{row\_pointers}[i+1]-1} \text{values}[j] \times x[\text{col\_indices}[j]]$$

for each row i. This row-oriented nature makes CSR particularly well-suited for matrices where row-wise operations are dominant [1, Chapter 2].

Beyond CSR, Ginkgo supports several other sparse matrix formats, each with its own strengths and typical use cases [16]:

- **ELLPACK (ELL):** Efficient for matrices with a relatively constant number of non-zeros per row. It can provide excellent performance on vector processors and GPUs due to regular memory access patterns [1, Chapter 2].
- **Coordinate (COO):** A simple format storing a list of $(row, col, value)$ triplets for each non-zero element. It is easy to construct and is often used as an intermediate format before converting to more performance-oriented formats like CSR or SELL-P [1, Chapter 2].
- **SELL-P (Sliced ELLPACK):** A variation of ELLPACK designed to improve performance on many-core processors by grouping rows with similar sparsity patterns into "slices" [17]. Ginkgo's implementation of SELL-P aims to balance the memory access regularity of ELL with better handling of matrices with varying numbers of non-zeros per row [18, 19].

Choosing a Matrix Format: A Trade-off Analysis

The choice of a matrix format in Ginkgo is a critical performance decision involving a trade-off between memory overhead, memory access patterns, and suitability for the matrix's specific sparsity structure.

- **CSR (`gko::matrix::Csr`):** This is the most common, general-purpose format. Its primary strength is its compact storage for any arbitrary sparse matrix. However, its performance on GPUs can be limited by irregular memory accesses and potential load imbalance. **Use Case:** It is the default, robust choice when the matrix structure is unknown or highly irregular.
- **ELL (`gko::matrix::Ell`):** This format's strength lies in the perfectly regular, coalesced memory accesses it enables, which is ideal for maximizing performance on GPUs. Its major weakness is significant memory overhead if the number of non-zeros per row varies greatly. **Use Case:** Ideal for matrices originating from structured grids or where rows have a nearly constant number of non-zeros.
- **SELL-P (`gko::matrix::Sellp`):** This format is a compromise designed to improve upon both CSR and ELL for many-core architectures. By

grouping rows into "slices", SELL-P mitigates the load-balancing problem of CSR and reduces the memory overhead of ELL. **Use Case:** Often the best-performing format for a wide range of matrices on GPUs.

Ginkgo's ability to easily convert between these formats allows users to benchmark and select the optimal format for their specific problem and hardware.

4 Preconditioning

Preconditioning is a crucial technique used to transform an ill-conditioned linear system into an equivalent one that has more favorable spectral properties, thereby accelerating the convergence of iterative solvers [1, Chapter 9], [2, Section 6.4]. The goal is to find a preconditioner matrix M that is a good approximation of A^{-1} and is cheap to apply.

4.1 Incomplete Factorization Preconditioners (ILU, IC)

Matrix factorization, or decomposition, is a fundamental technique in linear algebra where a matrix A is expressed as a product of simpler matrices. For large sparse matrices, computing a complete factorization is often prohibitively expensive in terms of both computation and memory due to "fill-in" – the creation of non-zero elements in the factors where A had zeros [1, Chapter 10], [2, Section 6.5.3].

In the context of preconditioning, *incomplete* factorizations are employed. An incomplete factorization approximates the full factorization by computing the factors while deliberately discarding some or all of the fill-in elements [1, Chapter 10], [2, Section 6.5.3]. The resulting incomplete factors $\tilde{L}$ and $\tilde{U}$ maintain a level of sparsity, making the application of the preconditioner computationally feasible.

Ginkgo provides implementations of incomplete factorizations tailored for parallel execution, primarily used as preconditioners [16, 14]. Key examples include:

- `gko::factorization::ParIlu`: An incomplete LU factorization that typically implements ILU(0), where the sparsity pattern of the factors is fixed to that of the original matrix [16]. The challenge in parallelizing such a factorization lies in managing data dependencies, for which Ginkgo employs strategies like level-scheduling for the subsequent sparse triangular solves [20, 21].

- `gko::factorization::ParLut`: A parallel incomplete LU factorization with thresholding. Unlike fixed-sparsity ILU, 'ParLut' uses a dynamic strategy based on a dropping threshold, which can lead to a more effective preconditioner at the cost of potentially denser factors [2, 22].

- `gko::factorization::ParIc`: An Incomplete Cholesky factorization variant for symmetric positive definite matrices, approximating $A \approx \tilde{L}\tilde{L}^T$ [16].

The application of these preconditioners involves performing sparse triangular solves with the incomplete factors. Ginkgo provides highly optimized parallel sparse triangular solvers to perform these operations efficiently, employing techniques such as level-scheduling and graph coloring [20, 21]. The high performance of these kernels on the target device is crucial for the overall effectiveness of the preconditioned iterative method [18, 19].

4.2 Simple Iterative Preconditioners (Jacobi)

The Jacobi preconditioner is one of the simplest forms, where the preconditioner matrix M is chosen to be the diagonal part of the matrix A, i.e., $M = \text{diag}(A)$. Applying its inverse is a trivially parallel operation. However, its effectiveness is limited and it is most effective for diagonally dominant matrices [1, Chapter 4], [2, Section 6.2]. Ginkgo implements this in `gko::preconditioner::Jacobi` and also supports more powerful Block-Jacobi variants [16].

4.3 Algebraic Multigrid (AMG)

Multigrid methods are a class of highly efficient techniques that can serve as powerful preconditioners, capable of solving certain elliptic problems with a computational complexity that scales optimally, i.e., linearly with the number of unknowns [1, Chapter 13]. The fundamental principle is that simple relaxation schemes (smoothers), such as Jacobi or Gauss-Seidel, are effective at reducing high-frequency (oscillatory) components of the error but are very slow at damping low-frequency (smooth) components. Multigrid methods overcome this by transferring the problem to a hierarchy of coarser grids, where the smooth error components from a fine grid become oscillatory and can therefore be efficiently damped [1, Section 13.1]. There are two main families of multigrid methods: Geometric and Algebraic.

Geometric Multigrid (GMG) requires an explicit hierarchy of nested geometric grids, which must be supplied by the user. The operators that transfer information between grids—prolongation (coarse-to-fine) and restriction (fine-to-coarse)—are defined based on the geometry of the mesh (e.g., via linear interpolation). GMG is extremely effective for problems with regular, structured grids, but its application is difficult for problems with complex geometries or on unstructured meshes, where defining a meaningful grid hierarchy is a non-trivial task [23].

Algebraic Multigrid (AMG), in contrast, is a more versatile approach that does not require any geometric information. Instead, AMG constructs the entire multigrid hierarchy—the coarse "grids" (which are subsets of variables), the restriction operators R, the prolongation (or interpolation) operators P, and the coarse-grid operators A_H—purely from the entries of the matrix A [1, Section 13.6]. This makes AMG applicable to a much wider range of problems, including those arising from unstructured meshes, positioning it as a powerful algebraic solver.

The construction of an AMG preconditioner involves two main stages: a setup phase and a solution phase (the cycle).

Setup Phase: The goal of the setup phase is to automatically define the grid hierarchy and transfer operators.

1. **Coarsening:** The variables (nodes) of the current grid are partitioned into two disjoint sets: *C-points*, which will form the next coarser grid, and *F-points*, which remain on the fine grid. This selection is crucial. A common strategy is based on identifying "strong connections" in the matrix graph; typically, a_{ij} represents a strong connection if its magnitude is large relative to other entries in row i. The C-points are then chosen as a maximal independent set of nodes that are not strongly connected to each other [1, Section 13.6.3].

2. **Interpolation:** The prolongation operator P (or I_H^h), which interpolates a correction from the coarse grid (C-points) to the fine grid, is constructed. The value at an F-point i is defined as a weighted average of the values at its neighboring C-points j. The weights w_{ij} are typically based on the strength of the connection a_{ij}, ensuring that algebraically smooth error is interpolated accurately [1, Section 13.6.2].

3. **Restriction:** The restriction operator R (or I_h^H) maps residuals from the fine grid to the coarse grid. It is typically defined as the transpose of the interpolation operator, $R = P^T$.

4. **Coarse-Grid Operator:** The operator for the next level, A_H, is formed via the Galerkin projection: $A_H = RA_hP = P^TA_hP$. This process is applied recursively until the coarse-grid system is small enough to be solved directly [1, Section 13.4.1].

Solution Phase (V-cycle): Once the hierarchy is built, a multigrid cycle (e.g., a V-cycle) is used to solve the system.

1. **Pre-smoothing:** Apply a few iterations of a simple smoother (e.g., Gauss-Seidel) to the current system $A_hx_h = f_h$ to damp high-frequency errors.

2. **Coarse-Grid Correction:**

   - Compute the residual: $r_h = f_h - A_hx_h$.
   - Restrict the residual to the coarse grid: $r_H = Rr_h$.
   - Solve the coarse-grid problem $A_He_H = r_H$ for the error correction e_H. If this is the coarsest level, solve directly. Otherwise, solve recursively by applying another V-cycle.
   - Interpolate the correction back to the fine grid: $e_h = Pe_H$.
   - Update the solution: $x_h \leftarrow x_h + e_h$.

3. **Post-smoothing:** Apply a few more smoother iterations to the corrected solution to damp any high-frequency errors introduced by the interpolation.

Ginkgo provides support for this powerful technique through the `gko::preconditioner::Multigrid` class [16], allowing it to be flexibly combined with various Krylov solvers.

Use Cases: GMG is highly effective for problems where a natural geometric hierarchy of grids exists, such as PDEs on structured domains. AMG is the state-of-the-art choice for large, sparse systems arising from unstructured discretizations or problems where no geometric information is available, making it a more general-purpose and robust algebraic technique.

4.4 Choosing a Preconditioner: A Trade-off Analysis

The selection of a preconditioner involves a crucial trade-off between its effectiveness, its setup cost, and its application cost.

- **Jacobi:** Simplest and cheapest. Its setup cost is negligible and its application is extremely fast. However, it is often too weak to provide meaningful convergence acceleration. **Use Case:** A good first choice for simple, diagonally-dominant problems or as a smoother within a Multigrid method.
- **Incomplete Factorizations (ILU/IC):** Powerful and general-purpose. They often provide a high-quality approximation of the matrix inverse, leading to a dramatic reduction in solver iterations. Their main weakness is that the application phase requires solving sparse triangular systems, which is a known bottleneck for parallelism on GPUs [20, 21]. **Use Case:** A workhorse for a wide variety of systems arising from PDEs.
- **Algebraic Multigrid (AMG):** Often the most powerful and scalable preconditioner. For certain problems, AMG can be an optimal-complexity preconditioner, meaning the solution time scales linearly with the problem size. However, it has the highest setup cost and is the most complex to tune. **Use Case:** The state-of-the-art choice for large, sparse systems from discretized PDEs where simpler preconditioners fail to scale.

5 Solvers

Iterative methods are particularly well-suited for large sparse systems as they typically have lower memory requirements than direct methods and can be efficiently combined with preconditioners [1, Chapters 5-7], [2, Chapter 6]. Ginkgo provides a rich collection of iterative solvers, many of which are Krylov subspace methods, as part of its `gko::solver` module [16].

5.1 Conjugate Gradient (CG)

The Conjugate Gradient (CG) method is a foundational iterative solver, specifically designed for symmetric positive definite (SPD) linear systems [1, Section 6.7]. The algorithm iteratively builds a set of A-orthogonal search directions to find the solution.

Key properties of the CG method include [1, Section 6.7]:

- **Guaranteed Convergence for SPD:** For SPD matrices, CG is guaranteed to converge to the exact solution in at most n iterations (where n is the matrix size) in exact arithmetic.
- **Optimality:** CG yields the solution with the minimum error in the A-norm over the current Krylov subspace.
- **Minimum Residual for SPD:** For SPD systems, CG also minimizes the residual norm $||b - Ax_k||_2$ at each iteration.

Ginkgo provides an implementation of the Conjugate Gradient solver as `gko::solver::Cg` [16].

5.2 Generalized Minimal Residual (GMRES)

The Generalized Minimal Residual (GMRES) method is a widely used iterative solver applicable to general non-symmetric and indefinite linear systems [1, Section 6.5]. GMRES operates by finding the vector x_k within the Krylov subspace that minimizes the Euclidean norm of the residual, $||b - Ax_k||_2$.

Properties of GMRES include [1, Section 6.5], [2, Section 6.3]:

- **General Applicability:** It is applicable to any non-singular linear system.
- **Monotonic Residual Reduction:** The residual norm is guaranteed not to increase at each iteration.
- **Computational Cost and Memory:** A drawback of standard GMRES is that its cost and memory requirements grow with each iteration. To manage this, GMRES is often used with restarts, denoted as GMRES(m).

Ginkgo provides a high-performance implementation of GMRES as `gko::solver::Gmres` [16].

5.3 Biconjugate Gradient Stabilized (BiCGSTAB)

For linear systems where the coefficient matrix A is not symmetric positive definite (SPD), the Conjugate Gradient (CG) method is not applicable. This has led to the development of a family of related Krylov subspace methods for general non-symmetric systems. The Biconjugate Gradient Stabilized (BiCGSTAB) method is a particularly popular and effective member of this family, representing a refinement over its predecessors, CG and the Biconjugate Gradient (BiCG) method [1, Chapter 7].

The Conjugate Gradient (CG) Method

The CG method, shown in Algorithm 1, is a highly efficient iterative solver for SPD systems. Its effectiveness stems from generating a sequence of A-orthogonal (or "conjugate") search directions, which guarantees convergence in at most n iterations in exact arithmetic. This property relies on the symmetry of A, which ensures that the inner product (Ap, p) is positive and that the residuals r_j are orthogonal to each other [1, Section 6.7].

```
// Compute r_0 = b - Ax_0, p_0 = r_0
// For j = 0, 1, ... until convergence Do:
//      α_j = (r_j, r_j)/(Ap_j, p_j)
//      x_{j+1} = x_j + α_j p_j
//      r_{j+1} = r_j - α_j Ap_j
//      β_j = (r_{j+1}, r_{j+1})/(r_j, r_j)
//      p_{j+1} = r_{j+1} + β_j p_j
// EndDo
```

Listing 1: The Conjugate Gradient (CG) Algorithm [1, p. 200]

The Biconjugate Gradient (BiCG) Method

To handle non-symmetric systems, the BiCG method (Algorithm 2) was developed as a direct generalization of CG. Since A is not symmetric, a single sequence of orthogonal residuals cannot be maintained. BiCG circumvents this by generating two sequences of vectors: the residuals r_j based on A, and "shadow" residuals r_j^* based on the transpose A^T. It then enforces a biorthogonality condition, $(r_j, r_i^*) = 0$ for $i \neq j$, which requires an additional matrix-vector product with A^T at each step [1, Section 7.3.1].

```
// Compute r_0 = b - Ax_0. Choose r_0^* s.t. (r_0, r_0^*) != 0.
// Set p_0 = r_0, p_0^* = r_0^*.
// For j = 0, 1, ... until convergence Do:
//      alpha_j = (r_j, r_j^*)/(Ap_j, p_j^*)
//      x_{j+1} = x_j + alpha_j * p_j
//      r_{j+1} = r_j - alpha_j * Ap_j
//      r_{j+1}^* = r_j^* - alpha_j * A^T p_j^*
//      beta_j = (r_{j+1}, r_{j+1}^*)/(r_j, r_j^*)
//      p_{j+1} = r_{j+1} + beta_j * p_j
//      p_{j+1}^* = r_{j+1}^* + beta_j * p_j^*
// EndDo
```

Listing 2: The Biconjugate Gradient (BiCG) Algorithm [1, p. 235]

While BiCG successfully extends the framework of CG to general matrices, its practical performance is often hampered by irregular and erratic convergence. The residual norm can fluctuate wildly, and the algorithm can suffer from breakdowns if $(r_j, r_j^*) = 0$ or $(Ap_j, p_j^*) = 0$ [1, Section 7.3.1].

The BiCGSTAB Method

The Biconjugate Gradient Stabilized (BiCGSTAB) method was developed by van der Vorst to overcome the stability issues of BiCG and another transpose-free variant, CGS [24]. BiCGSTAB is a hybrid method that smooths the convergence of BiCG. At each iteration, after performing a BiCG-like step, it adds a "stabilizing" step that locally minimizes the residual, similar to a single step of GMRES.
This modification has two key benefits. First, it typically leads to much smoother and faster convergence than BiCG. Second, the algorithm is formulated to avoid any operations with the matrix transpose A^T. Instead, it requires two matrix-vector products with the original matrix A per iteration. This makes it more efficient than BiCG when A^T is unavailable or expensive to compute [1, Section 7.4.2]. Ginkgo includes a high-performance implementation of BiCGSTAB as `gko::solver::Bicgstab` [16].

```
// Compute r_0 = b - Ax_0. Choose r_0^*(e.g., r_0^* = r_0).
// Set p_0 = r_0.
// For j = 0, 1, ... until convergence Do:
//      alpha_j = (r_j, r_0^*)/(Ap_j, r_0^*)
//      s_j = r_j - alpha_j * Ap_j
//      // Stabilizing step
//      omega_j = (As_j, s_j)/(As_j, As_j)
//      x_{j+1} = x_j + alpha_j * p_j + omega_j * s_j
//      r_{j+1} = s_j - omega_j * As_j
//      // Update for next iteration
//      beta_j = [(r_{j+1}, r_0^*)/(r_j, r_0^*)] * [alpha_j/omega_j]
//      p_{j+1} = r_{j+1} + beta_j * (p_j - omega_j * Ap_j)
// EndDo
```

Listing 3: The Biconjugate Gradient Stabilized (BiCGSTAB) Algorithm [1, p. 247]

5.4 Code Implementation Example

The modularity of Ginkgo is best illustrated through its factory-based design. To solve a system, a user typically creates factories for the desired solver and preconditioner and then combines them. The official Ginkgo documentation provides a complete example of enhancing a Conjugate Gradient (CG) solver with a powerful Multigrid preconditioner, which is a common strategy for complex problems.
The code snippet in listing 4, adapted from this example, shows the core logic for setting up and using this advanced solver.

```
// Type aliases for convenience
using ValueType = double;
using vec = gko::matrix::Dense<ValueType>;
using mtx = gko::matrix::Csr<ValueType>;
using cg = gko::solver::Cg<ValueType>;
using mg = gko::solver::Multigrid;
using pgm = gko::multigrid::Pgm<ValueType>;
```

```
8
9  // exec is a smart pointer to a gko::Executor
10
11 // 1. Create a factory for the Multigrid preconditioner.
12 // Here, we configure a single cycle of an AMG variant called PGM.
13 std::shared_ptr<gko::LinOpFactory> multigrid_gen =
14 mg::build()
15 .with_mg_level(pgm::build().with_deterministic(true))
16 .with_criteria(gko::stop::Iteration::build().with_max_iters(1u))
17 .on(exec);
18
19 // 2. Create a factory for the Conjugate Gradient solver.
20 // Set stopping criteria: 100 iterations or 1e-8 residual reduction
       .
21 const gko::remove_complex<ValueType> tolerance = 1e-8;
22 auto solver_gen =
23 cg::build()
24 .with_criteria(gko::stop::Iteration::build().with_max_iters(100u),
25 gko::stop::ResidualNorm<ValueType>::build()
26 .with_reduction_factor(tolerance))
27 // 3. Combine the solver with the multigrid preconditioner.
28 .with_preconditioner(multigrid_gen)
29 .on(exec);
30
31 // 4. Generate the solver and preconditioner by passing the system
       matrix A.
32 auto solver = solver_gen->generate(A);
33
34 // 5. Add a logger to retrieve statistics like iteration count.
35 std::shared_ptr<const gko::log::Convergence<ValueType>> logger =
36 gko::log::Convergence<ValueType>::create();
37 solver->add_logger(logger);
38
39 // 6. Solve the system Ax = b.
40 solver->apply(b, x);
```

Listing 4: Core logic for creating a Multigrid-preconditioned solver in Ginkgo. Adapted from the official Ginkgo documentation.

This example demonstrates how Ginkgo's components can be composed to build a sophisticated solution strategy. The full example in the documentation also includes robust logic for selecting hardware executors (CPU, CUDA, HIP), reading data, and logging performance metrics such as generation and execution time. The factory pattern makes it trivial to swap the Multigrid preconditioner for another, such as ILU or Jacobi, to benchmark and find the optimal strategy for a given problem.

5.5 Performance Impact of Preconditioners

The choice of preconditioner has a dramatic effect on solver performance. This is clearly demonstrated in the performance results published by the Ginkgo authors in their main article [11].

Figure 1, reproduced from this article, shows the time-to-solution for a GMRES solver on two different GPU architectures when solving anisotropic flow

problems. The key comparison is between a standard ILU preconditioner from NVIDIA's cuSPARSE library and Ginkgo's advanced threshold-based parallel ILU, ParILUT [22].

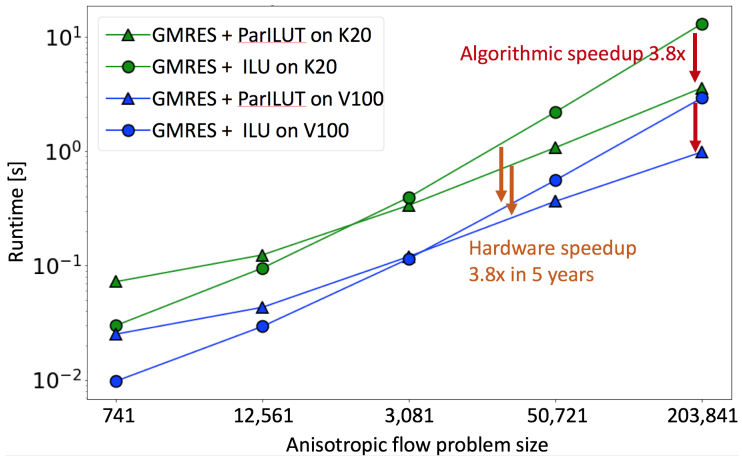

Figure 1: Time-to-solution comparison between a standard ILU preconditioner (from NVIDIA's cuSPARSE) and Ginkgo's advanced ParILUT for solving anisotropic flow problems. The y-axis represents time in seconds (logarithmic scale), while the x-axis lists different test matrices. Lower bars indicate better performance. Graph reproduced from Anzt et al. [11], Figure 9.

The analysis of fig. 1 reveals a key insight: for complex problems, a more sophisticated preconditioner can lead to significantly faster solution times, even if its setup cost is higher. The ParILUT preconditioner, which dynamically adapts the sparsity pattern of the incomplete factors, consistently outperforms the standard, static-pattern ILU from the vendor library. This demonstrates that investing in a higher-quality preconditioner is often crucial for achieving high performance. This result powerfully illustrates the trade-offs discussed earlier and highlights the value of the advanced algorithms implemented in Ginkgo.

6 Conclusion

In this paper, we have explored the capabilities of the Ginkgo high-performance linear algebra library for addressing the fundamental problem of solving linear systems of the form $Ax = b$. A central concept underpinning Ginkgo's design is the Linear Operator abstraction [11, 15], which provides a unified interface fostering modularity and flexibility.
We have delved into essential components within Ginkgo's framework. The efficiency of SpMV was highlighted as a core computational kernel, emphasizing the necessity of appropriate matrix formats like CSR, ELL, and SELL-P [1, Chapter 2], [4, 17]. Matrix factorizations, particularly incomplete factorizations

such as ILU, were discussed as crucial building blocks for constructing effective preconditioners [1, Chapter 10], [2, Section 6.5.3].
Preconditioning itself was identified as vital for accelerating the convergence of iterative solvers. As demonstrated with published performance data, the choice of preconditioner—from the simple Jacobi to the powerful ParILUT and AMG methods—is often the most critical factor in achieving an efficient solution [11].
Finally, we discussed key iterative solvers available in Ginkgo: CG for symmetric positive definite systems, and GMRES and BiCGSTAB for general non-symmetric systems [1, Chapters 6-7], [2, Section 6.3]. The performance of these Krylov subspace methods is greatly enhanced by the use of effective preconditioners. Ginkgo's factory-based design allows for the flexible combination of these components, enabling users to easily configure and test different solution strategies [16, 25].
In summary, Ginkgo provides a flexible, high-performance framework for tackling linear algebra problems. By offering optimized implementations of essential components, all unified under the Linear Operator abstraction, Ginkgo empowers users to achieve high performance on diverse computing platforms. Its focus on performance portability makes it an attractive library for scientific computing in high-performance environments [11, 13, 12].

References

[1] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[2] James W Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[3] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson Education, 2nd edition, 2007.

[4] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimizing sparse matrix-vector multiplication for GPUs. Technical Report LBNL-6214E, Lawrence Berkeley National Laboratory, 2007.

[5] Charles L Lawson, Richard J Hanson, David R Kincaid, and Fred T Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[7] NVIDIA Corporation. cuSPARSE Library User's Guide. `https://docs.nvidia.com/cuda/cusparse/index.html`, 2023. Accessed: June 28, 2025.

[8] Hartwig Anzt, Georg Stoyanov, and Jack Dongarra. The rocALUTION library-a multi-platform sparse linear algebra library. In *Proceedings of the 5th international workshop on performance modeling, benchmarking and simulation of high performance computer systems*, pages 1–8, 2015.

[9] Satish Balay, William D Gropp, Lois Curfman McInnes, and Barry F Smith. Efficient management of parallelism in object oriented numerical software libraries. In *Modern software tools for scientific computing*, pages 163–202. Birkhäuser, Boston, MA, 1997.

[10] PETSc Development Team. PETSc GPU Performance and Roadmap. `https://petsc.org/release/overview/gpu_roadmap/`, 2024. Accessed: June 28, 2025.

[11] Hartwig Anzt, Terry Cojean, Goran Flegar, Thomas Grützmacher, Pratik Loecher, Piotr Luszczek, Shreyas Nayak, Tobias Ribizel, Yu-Hsiang Tsai, Jan Trogemann, Fritz Goebel, and Enrique S. Quintana-Ortí. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Transactions on Mathematical Software (TOMS)*, 46(3):1–32, 2020.

[12] Shreyas Nayak, Piotr Luszczek, and Hartwig Anzt. A distributed memory Ginkgo library with CUDA-aware MPI. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 1–10. IEEE, 2020.

[13] Hartwig Anzt et al. Ginkgo: A high-performance linear algebra library for heterogeneous architectures. *International Journal of High Performance Computing Applications*, 34(4):473–489, 2020.

[14] Hartwig Anzt, Terry Cojean, Jack Dongarra, Goran Flegar, and Piotr Luszczek. Design and implementation of the Ginkgo numerical linear algebra library. Technical Report UT-EECS-18-820, University of Tennessee, Innovative Computing Laboratory, 2018.

[15] Hartwig Anzt, Terry Cojean, Jack Dongarra, Goran Flegar, and Piotr Luszczek. The linear operator concept in Ginkgo. Technical Report UT-EECS-18-821, University of Tennessee, Innovative Computing Laboratory, 2018.

[16] Hartwig Anzt, Terry Cojean, Jack Dongarra, Goran Flegar, and Piotr Luszczek. Ginkgo's software design. Technical Report UT-EECS-18-822, University of Tennessee, Innovative Computing Laboratory, 2018.

[17] Hartwig Anzt, Goran Flegar, Piotr Luszczek, and Jack Dongarra. SELL-C-σ and a new SpMV implementation for the Ginkgo linear algebra library. In *International Conference on High Performance Computing*, pages 397–416. Springer, 2015.

[18] Hartwig Anzt, Terry Cojean, Jack Dongarra, and Goran Flegar. Load-balancing-aware parallel sparse matrix-vector multiplication on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 645–654. IEEE, 2018.

[19] Hartwig Anzt, Terry Cojean, Jack Dongarra, and Goran Flegar. On the potential of the Ginkgo linear algebra library for graphics processing units. *Concurrency and Computation: Practice and Experience*, 31(17):e5034, 2019.

[20] Terry Cojean, Hartwig Anzt, and Jack Dongarra. Level-based parallelism for sparse triangular solution in Ginkgo. In *International Conference on Parallel Processing and Applied Mathematics*, pages 15–26. Springer, 2016.

[21] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Improving the performance of the sparse triangular solve on NVIDIA GPUs. In *International Conference on High Performance Computing*, pages 241–259. Springer, 2015.

[22] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. ParILUT—A Parallel Threshold ILU for GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 231–241. IEEE, 2019.

[23] Guoyuan Chen et al. A technical survey of sparse linear solvers in electronic design automation. *arXiv preprint arXiv:2504.11716*, 2024.

[24] Henk A. Van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[25] Hartwig Anzt et al. The Ginkgo Mathematics Library. `https://github.com/ginkgo-project/ginkgo`, 2017–2025. Accessed: June 6, 2025.