RESEARCH-ARTICLE

pyGinkgo: A Sparse Linear Algebra Operator Framework for Python

Published: 08 September 2025

Citation in BibTeX format

ICPP '25: 54th International Conference on Parallel Processing
September 8 - 11, 2025
CA, San Diego, USA

ICPP '25: Proceedings of the 54th International Conference on Parallel Processing (September 2025)
https://doi.org/10.1145/3754598.3754648
ISBN: 9798400720741

pyGinkgo: A Sparse Linear Algebra Operator Framework for Python

Keshvi Tuteja
Karlsruhe Institute of Technology
Karlsruhe, Germany
keshvi.tuteja@kit.edu

Gregor Olenik
Technical University of Munich
Heilbronn, Germany
gregor.olenik@tum.de

Roman Mishchuk
Technical University of Munich
Heilbronn, Germany
roman.mishchuk@tum.de

Yu-Hsiang Tsai
Technical University of Munich
Heilbronn, Germany
yu-hsiang.tsai@tum.de

Markus Götz
Helmholtz AI
Karlsruhe, Germany
Karlsruhe Institute of Technology
Karlsruhe, Germany
markus.goetz@kit.edu

Achim Streit
Karlsruhe Institute of Technology
Karlsruhe, Germany
achim.streit@kit.edu

Hartwig Anzt
Technical University of Munich
Heilbronn, Germany
University of Tennessee
Knoxville, USA
hartwig.anzt@tum.de

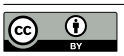
Charlotte Debus
Karlsruhe Institute of Technology
Kalsruhe, Germany
charlotte.debus@kit.edu

Abstract

Sparse linear algebra is a cornerstone of many scientific computing and machine learning applications. Python has become a popular choice for these applications due to its simplicity and ease of use. Yet high-performance sparse kernels in Python remain limited in functionality, especially on modern CPU and GPU architectures. We present pyGinkgo, a lightweight and Pythonic interface to the Ginkgo library, offering high-performance sparse linear algebra support with platform portability across CUDA, HIP, and OpenMP backends. pyGinkgo bridges the gap between high-performance C++ backends and Python usability by exposing Ginkgo's capabilities via Pybind11 and a NumPy and PyTorch compatible interface. We benchmark pyGinkgo's performance against state-of-the-art Python libraries including SciPy, CuPy, PyTorch and TensorFlow. Results across hardware from different vendors demonstrate that pyGinkgo consistently outperforms existing Python tools in both Sparse Matrix Vector (SpMV) product and iterative solver performance, while maintaining performance parity with native Ginkgo C++ code. Our work positions pyGinkgo as a compelling backend for sparse machine learning models and scientific workflows.

Keywords

Sparse Linear Algebra, Performance Optimization, Software Engineering, Parallel Algorithms, Python, Sparse Matrix Vector Multiplication



ICPP '25, San Diego, CA, USA

ACM ISBN 979-8-4007-2074-1/25/09
https://doi.org/10.1145/3754598.3754648

ACM Reference Format:
Keshvi Tuteja, Gregor Olenik, Roman Mishchuk, Yu-Hsiang Tsai, Markus Götz, Achim Streit, Hartwig Anzt, and Charlotte Debus. 2025. pyGinkgo: A Sparse Linear Algebra Operator Framework for Python. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3754598.3754648

1 Introduction

Sparse linear algebra plays a crucial role in many scientific and engineering applications, including fluid dynamics, heat transfer problems, and data mining. Frameworks and libraries used for high-performance sparse linear algebra are typically very advanced and offer a highly optimized stack of computational kernels [3, 18]. However, these are mostly implemented in low-level languages like C++ and Fortran.

While using low-level languages has obvious performance benefits, Python has emerged as a popular choice for scientific computing and data science applications due to its ease of use, readability, and extensive ecosystem of libraries. Its dominance is especially visible in the machine learning community. In contrast to its strengths in dense linear algebra and general usability, Python's ecosystem for scalable and efficient sparse computations remains relatively underdeveloped. For example, SciPy [35] and NumPy [21], rely on frameworks like BLAS [19] [1] [20] and LAPACK [6] for dense linear algebra operations, but the functionality and hardware support for sparse linear algebra remains limited.

The need for sparse linear algebra libraries in Python is currently becoming even more imminent, given the intensive developments towards sparse neural networks [23]. These sparse models, e.g., when most elements of the weight matrices tend to zero, rely heavily on core operations like sparse matrix-vector and matrix-matrix

products. Even though the prevailing workloads, such as Transformers [41], are using dense representations, there are a growing number of methods that leverage sparse formulation, including, but not limited to, spiking and graph neural networks as well as the weight matrices after the gradient-descent based optimization [17, 38, 42].

Ginkgo [8] is a C++ based high-performance linear algebra library with a special focus on sparse linear systems. To bridge the performance gap between dense and sparse computations in the Python world, we present pyGinkgo – Python bindings for the Ginkgo library. pyGinkgo utilizes Pybind11 [24] to access Ginkgo's C++ kernels from Python side, which provides a Pythonic API as well as interoperability with NumPy and PyTorch, while introducing only a minimal performance overhead. This design enables Python users to leverage Ginkgo's advanced capabilities for performing sparse computations, offering significant potential for improving the performance in python-based scientific computing and machine learning workloads.

Our key contributions through this work are as follows:

- pyGinkgo, a sparse linear algebra library for Python, providing access to Ginkgo's powerful iterative solvers and SpMV kernels.
- A thorough performance evaluation of state-of-the-art sparse linear algebra frameworks in Python, focusing on matrix-vector multiplication (SpMV) and iterative solvers. The comparison spans both CPU and GPU platforms, including hardware from NVIDIA and AMD.
- An empirical estimation of the engineering overhead introduced by Pybind11 [24] when comparing pyGinkgo to the native C++ implementation of Ginkgo, demonstrating marginal performance loss.

The implementation of pyGinkgo is publicly available at https://github.com/Helmholtz-AI-Energy/pyGinkgo.

2 Related Work and State-of-the-Art

Over the years, several efforts have been made to enable sparse linear algebra capabilities in Python. Early developments in this area mainly relied on low-level backends originally designed for C and C++ environments. Some notable examples implemented in low-level languages are Ginkgo, libsrb [26], cuSPARSE [28], MAGMA [10] and Kokkos Kernels [39]. Among these resources, we selected Ginkgo due to its combination of modern design, performance portability across multiple hardware, and rich support for iterative solvers and preconditioners.

Building on these low-level foundations, a number of high-level Python libraries have emerged to improve usability and integration within the Python ecosystem. These libraries either wrap existing low-level backend or offer native sparse functionality. Some notable examples are SLEPc [22], PyTrilinos [36], FEniCS [5], PyAMG [14] and PySparse [37]. These libraries provide varying degrees of sparse matrix support, often targeting specialized applications such as finite element analysis, eigenvalue problems and algebraic multigrid methods. However, most of these libraries either lack GPU acceleration or compatibility with modern hardware, fast implementations, or a Pythonic interface. As a consequence, some of these libraries have become obsolete or are no longer actively maintained.

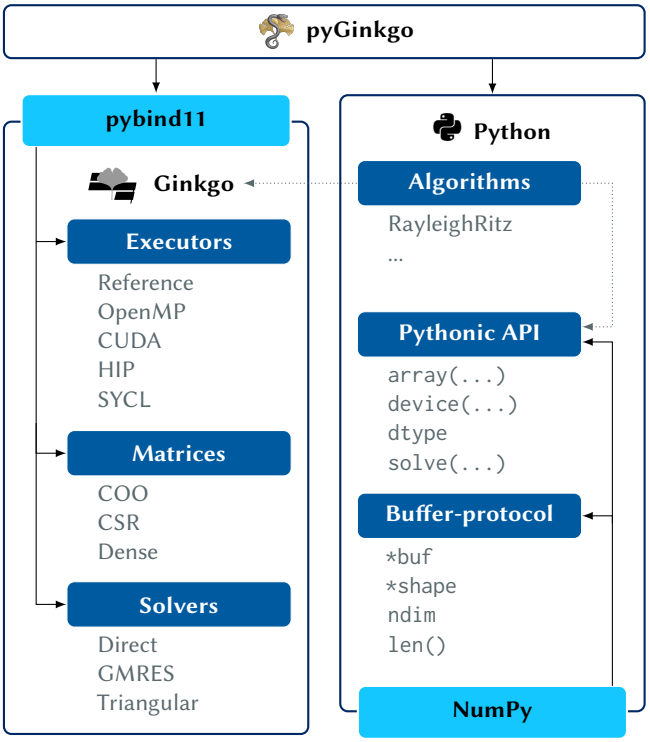


Figure 1: Overview of pyGinkgo

More recently, major Python frameworks such as PyTorch [34], TensorFlow [2], SciPy, and CuPy [33] have progressively incorporated support for sparse linear algebra operations. PyTorch and TensorFlow provide support for sparse matrices and a limited set of sparse operations and its integration with neural networks is restricted. Both of these libraries lack advanced features like iterative solvers and preconditioners. TensorFlow only supports the Coordinate (COO) matrix format. Moreover, computations at double precision in PyTorch and TensorFlow are rather inefficient, making them unsuitable for many high-performance computing tasks. Our own results show that their SpMV kernel is not optimized (see section Section 6). In comparison, CuPy and SciPy offer decent support. They provide functionality for various sparse matrix formats as well as various direct and iterative solvers. CuPy leverages several CUDA Toolkit libraries such as cuBLAS [31], cuRAND [30], cuSOLVER [29], cuSPARSE, cuFFT [32], cuDNN [16], and NCCL [27] to provide functionality similar to SciPy. Although these contemporary libraries offer broad dense computation capabilities, most lack comprehensive support for sparse matrices, iterative solvers, or efficient parallelism on both CPUs and GPUs. This highlights the need for an optimized, portable and extensible sparse linear algebra library like pyGinkgo.

3 Overview

pyGinkgo is built as a wrapper around Ginkgo. Its design is rooted in three main building blocks:

- The C++ Core: Ginkgo's high-performance algorithms and data structures are implemented in C++ and it is the central building block for pyGinkgo.
- Python Bindings: Pybind11 is used to create a thin wrapper around Ginkgo enabling the support for all the accelerator backends already provided by Ginkgo
- A user-friendly Python Application Programming Interface (API) that provides basic interoperability with commonly used Python packages.

3.1 Principal Architecture of pyGinkgo

The pyGinkgo library consists of two principal parts:

(1) The C++ bindings using pybind11 [24] that generate functions and classes which are accessible from Python.
(2) Further implementations in pure Python, which are responsible for a seamless integration into the existing Python ecosystem by providing complex dispatching based on datatypes or implementing complex mathematical algorithms. This includes integration with NumPy and SciPy and PyTorch interoperability to make pyGinkgo more idiomatic to Python.

Figure 1 shows the general structure of the pyGinkgo library and its main features. In addition to exposing functionality provided by Ginkgo, pyGinkgo also includes several algorithms and methods implemented purely in Python. Currently, we provide implementations of the Rayleigh–Ritz method, with ongoing development to expand this further. We also provide methods like `array()` and `device()` to make the API more intuitive. By design, pyGinkgo tries to implement a Pythonic interface. For example, operations on matrices and vectors resemble NumPy idioms, and solver pipelines can be constructed with minimal boilerplate code. This balance between leveraging Ginkgo's performance and Python's simplicity is central to pyGinkgo's design philosophy. Note, that pyGinkgo is still under active development. We plan to add support for various other matrix formats and algorithms that Ginkgo offers.

3.2 Ginkgo as a Computational Engine

Ginkgo is a high-performance linear algebra library with a focus on sparse linear systems. It provides high-performance algorithms and data structures implemented in C++ [7, 8]. One of its primary design choices is platform-portable performance [9, 11, 40], ensuring that computations run optimally across a diverse range of hardware architectures from major vendors. To accomplish this, Ginkgo implements its computational kernels using vendor-native programming models, such as CUDA for NVIDIA GPUs, HIP for AMD GPUs, and SYCL for Intel GPUs, while also leveraging OpenMP annotated C++ code for efficient multi-threaded execution on CPUs. This approach enables Ginkgo to take full advantage of hardware-specific optimizations while maintaining a unified and consistent interface for users. By providing a common C++ API, Ginkgo abstracts away the complexity of hardware-specific implementations, allowing developers to focus on their applications without having to modify their code for different computing platforms. While Ginkgo is implemented in C++ for high performance, many users prefer Python due to its ease of use, extensive ecosystem, and strong support for numerical computing through libraries like NumPy and SciPy. For example, libraries such as TensorFlow, PyTorch, and JAX [15] also offer integration with NumPy and SciPy. These libraries have gained widespread popularity because they successfully combine a Pythonic interface with high-performance C/C++ backends. These libraries follow a common design philosophy: expose minimal and idiomatic APIs to the user while deferring performance-critical operations to lower-level code. pyGinkgo is built on the same underlying design principles. However, while these libraries have focused primarily on dense operations, pyGinkgo addresses this gap. This would be especially beneficial for the applications that require efficient sparse matrix computations and GPU-accelerated operations, as Ginkgo's optimized kernels for CUDA, HIP, and SYCL could be directly utilized from Python.

3.3 Pybind11 for Python Bindings

Since Ginkgo implements a C++ API to access its algorithms and data structures, a way to interact with the C++ API from Python code is required. When selecting a tool to create Python bindings for the Ginkgo C++ codebase, we had the following key requirements:

- Minimal performance overhead
- Pythonic API and interoperability with NumPy
- Modern C++ support
- Simple build integration

After evaluating several options, we found that pybind11 was the best fit for our needs, especially when compared to alternatives like SWIG [12], Boost.Python [4], Cython [13], and ctypes. Pybind11's modern, lightweight, and header-only design enables the generation of high-performance Pythonic APIs with minimal boilerplate, avoiding the complexity of intermediate layers. Unlike SWIG, which supports multiple languages at the cost of increased overhead and less intuitive bindings, pybind11 focuses exclusively on seamless C++/Python interoperability, providing direct and natural mappings for C++ classes, templates, and STL containers. We also considered Cython, which is known for its ease of use and rapid development, especially for calling simple C functions or writing performance-critical Python code. However, Cython falls short when dealing with more advanced C++ features, such as class hierarchies, templates, and lifetime management of native objects. Moreover, Pybind11 includes valuable features like NumPy support, keyword arguments, and smart pointer management. Although Boost.Python offers similarly powerful functionality, but it comes at the price of heavier dependencies, slower compilation, and more complex setups. Since we are focusing on performance-critical workloads, pybind11 was the superior choice. As demonstrated in section Section 6.3, pyGinkgo is quite light-weight in terms of relative performance difference with respect to Ginkgo's native implementation.

3.4 Algorithms in Pure Python

Ginkgo is built entirely in C++ and relies heavily on C++ features to enable modular and efficient interactions between its components. However, experimenting with new ideas or customizing algorithms within a C++ codebase can pose a significant barrier for users who are unfamiliar with advanced C++ or are more comfortable with Python. With pyGinkgo, users can now explore new methods and integrate Ginkgo into their applications directly from Python,

making it much easier to adopt high-performance linear algebra in diverse scientific workflows.

To ensure pyGinkgo feels like a native Python library rather than a thin C++ wrapper, additional care is needed beyond the use of pybind11. For example, Python has no direct equivalent to function overloading, which we discuss more in Section 5.1, and instead accepts types indiscriminately as arguments to a function if no explicit type checking is performed. Thus, naturally, Python APIs are different compared to C++ APIs, often with a single entry point function that implements complex dispatching mechanisms based on the arguments passed. For an integration into the Python ecosystem, pyGinkgo implements comparable entry point functions, such as `as_tensor`, `solve`, or `device` in PyTorch or CuPy, that require a dispatching mechanism to the functions provided by pybind11. As proof of concept, we implemented Rayleigh-Ritz method on the Python side, which is not natively supported by Ginkgo yet. This demonstrates the extensibility of our framework, through which users can construct complex algorithms composed of existing linear algebra operations exposed using pyGinkgo. For instance, Rayleigh–Ritz relies on repeated applications of operations such as sparse matrix–vector products and other operations which are available as Ginkgo operators. This allows users to focus on algorithm design without worrying about low-level GPU or CPU parallelization details. Hence, prototyping of new solvers or algorithmic variants is possible, while still leveraging the performance and portability of the Ginkgo backend.

3.5 pyGinkgo API

In this section, we discuss how to use the pyGinkgo API to solve a sparse linear system of the form $Ax = b$ as shown in Listing 1. In principle, pyGinkgo provides two ways to create a linear solver: first, via specific solver bindings shown in Listing 1 and second, via a generic solver interface called the `config solver`.

Initially, a CUDA device (`pg.device('CUDA')`) is instantiated to enable GPU-based computations. The sparse matrix in Compressed Sparse Row (CSR) format is then loaded using the `read` function. The input matrix is read from a Matrix Market (MTX) file named `m1.mtx`. This is demonstrated in lines 4-7. The `device` function is an abstraction for the `Executor` on Ginkgo's side. It determines the device on which data is stored and the computations are performed. As consistent with Ginkgo, pyGinkgo currently supports CUDA, HIP, and OpenMP executors (discussed in Section 4.1).

The right-hand side vector b and the solution vector x are initialized with `double` (`np.float64`) for numerical precision. pyGinkgo currently supports half, single and double-precision values and index types (discussed in Section 5.1).

Vectors can be instantiated directly using `pyGinkgo.as_tensor` by specifying the executor, shape, and fill value, as demonstrated for b and x in lines 9-14. Alternatively, a NumPy array can be created and copied into a pyGinkgo tensor (discussed in Section 5.2).

After instantiating the necessary data structures, `mtx`, `x`, and `b`, an `Ilu` preconditioner is instantiated. In the example, generalized minimal residual method (GMRES), a Krylov subspace method is instantiated with a Krylov dimension of 30. The solver is configured to stop based on a maximum of 1000 iterations or a relative residual reduction factor of 10^{-6}. This is specified in lines 16-22. The `apply` method is then called. The function returns a logger, which provides diagnostic information about convergence and iteration progress, and the solution vector, which overwrites the initial guess.

In addition to providing the direct bindings to Ginkgo's solver classes, pyGinkgo also offers a configuration-based interface. When the `solve` method is called on pyGinkgo's side, a dictionary that is based on the arguments that are passed is created at the python backend. An example dictionary is as seen in lines 1-16 in Listing 2. This dictionary is then used to call Ginkgo's `config_solve` method. In the example, GMRES is instantiated with a Krylov dimension of 30 and a Jacobi preconditioner with a block size of 1. The solver is configured to stop based on a maximum of 1000 iterations or a relative residual reduction factor of 10^{-6}.

```
1  import pyGinkgo as pg
2  import numpy as np
3
4  fn = 'm1.mtx'
5  dev = pg.device("cuda")
6  mtx = pg.read(device=dev, path=fn, dtype="double", format
       ="Csr")
7  n_rows = mtx.size[0]
8
9  b = pg.as_tensor(
10   device=dev, dim=(n_rows,1), dtype="double", fill=1.0
11 )
12 x = pg.as_tensor(
13   device=dev, dim=(n_rows,1), dtype="double", fill=0.0
14 )
15
16 # Create ILU preconditioner
17 preconditioner = pg.preconditioner.Ilu(dev, mtx)
18
19 #Setup GMRES solver
20 solver = pg.solver.gmres(dev, mtx, preconditioner,
21     max_iters=1000, krylov_dim=30, reduction_factor=1e-06
22 )
23
24 #Apply
25 logger, result = solver.apply(b, x)
```

Listing 1: Python example demonstrating the use of pyGinkgo to solve a sparse linear system using the GMRES iterative solver with an ILU preconditioner using the direct solver bindings.

```
1  args = {
2      "type": "solver::Gmres",
3      "krylov_dim": 30,
4      "preconditioner": {
5          "type": "preconditioner::Jacobi",
6          "max_block_size": 1
7      },
8      "criteria": [
9          {"type": "Iteration", "max_iters": 1000},
10         {
11           "type": "ResidualNorm",
12           "reduction_factor": 1e-6,
13           "baseline": "rhs_norm"
14         }
15     ],
16 }
```

Listing 2: An example dictionary created at Python backend to be passed to Ginkgo's solver kernels demonstrating the use of pyGinkgo to solve a sparse linear system using the GMRES iterative solver with a Jacobi preconditioner on a CUDA executor using the config solver.

4 Implementation Details and Advanced Concepts

4.1 Device Access via Executor

pyGinkgo provides bindings for Ginkgo's executor class that determines where the underlying data is stored and where computations take place. Executors play a crucial role in managing memory and controlling execution across different hardware backends. These are also responsible for allocation/de-allocation of memory on device, synchronizing computations between host and device as well as copying data between multiple executors. pyGinkgo relies on pybind11's support for smart pointers allowing Python to share ownership with C++ in a safe way. As already discussed in section 3.2, pyGinkgo currently implements bindings to the CUDA, HIP, Omp, and reference executors offered by Ginkgo. For integration of pyGinkgo into the existing ecosystem, pyGinkgo refers to the executor as `device`. Additionally, pyGinkgo provides a `pyGinkgo.device(name, id=0)` factory function to create an instance of an executor. Functions typically accept an executor instance as an argument, and data structures provide a getter to retrieve an assigned executor. Executor objects are among the first components to be defined when writing code with pyGinkgo. It is important to note that a program can utilize multiple executors simultaneously, allowing users to perform different operations and store data on specific devices as needed. Using multiple executors may have additional overhead in copying the data from one device to another.

To better illustrate some of the pybind11 concepts, the bindings for the executor base class and the CUDA executor are shown in the Listing 3. The binding to a C++ class is declared by pybind's `class_<T>(module, "name")`. In the given syntax, `T` are a set of template arguments, where typical arguments are the corresponding class for which bindings are generated, any base classes of the class, and the holder type. For more information on further template parameters, we refer to the pybind11 documentation[1]. Specifying information about the base class allows to then pass any child classes as arguments, which accepts the base class as a parameter to generated functions. The holder type defines whether the underlying instance can be accessed as a reference or via a smart pointer, an important aspect regarding the fact that Ginkgo handles objects with smart pointers. This behavior is enforced by implementing constructors as protected, thus forcing the instantiation of concrete objects via static create functions, which, in turn, always return a smart pointer to the created object.

Furthermore, the function argument `module` defines the Python module in which the corresponding Python class should be registered, and `name` defines the name of the class on the Python side. In this example, it is `CUDA`.

```
py::class_<gko::CudaExecutor,
    gko::detail::ExecutorBase<gko::CudaExecutor>,
    std::shared_ptr<gko::CudaExecutor>
>(module, /*name = */"CUDA")
    .def(py::init([](int dev_id,
           std::shared_ptr<gko::Executor> master,
           std::shared_ptr<gko::CudaAllocatorBase> alloc,
           std::shared_ptr<gko::cuda_stream> stream
       ) {
           return gko::CudaExecutor::create(dev_id,
               master, alloc, stream->get());
       }));
```

Listing 3: Exemplary implementation of the CudaExecutor binding.

[1]https://pybind11.readthedocs.io/en/stable/advanced/classes.html

4.2 Ginkgo's Linear Operator Abstraction

The `LinOp` (Linear Operator) class is a core abstraction in Ginkgo, serving as the base type for nearly all data structures, including matrices, solvers, and preconditioners. Each concrete implementation of `LinOp` defines an `apply` method, which specifies how the operator acts on input vectors. Under this abstraction lies a variety of operations: a matrix `LinOp` performs a SpMV, a solver `LinOp` applies a linear system solver to a right-hand side vector (as seen in Listing 1), and a preconditioner `LinOp` applies a transformation to accelerate convergence. With this abstraction, every object that models a linear operation can be applied using the same method call. This unified interface fosters a high degree of composability, enabling users to construct sophisticated solver pipelines by combining different linear operators. `pyGinkgo` preserves this design philosophy, extending the `LinOp` abstraction and its capabilities to Python users.

5 Generic Ginkgo Solver Entry Points

Ginkgo provides a generic entry point for its solver via configuration parameters in JSON, YAML, or custom format. This generic solver entry point, internally also referred to as config-solver, can be instantiated as any other LinOp via a LinOpFactory and invoked through the LinOp's apply function, as discussed in Section 4.2. With this method, users of Ginkgo gain access to all available solvers and specialized preconditioners without having to write code for each use case. Additionally, this also allows the selection of solvers and preconditioner combinations at run-time, thus giving users the flexibility to experiment with different setups by simply modifying a configuration without having to recompile the application.

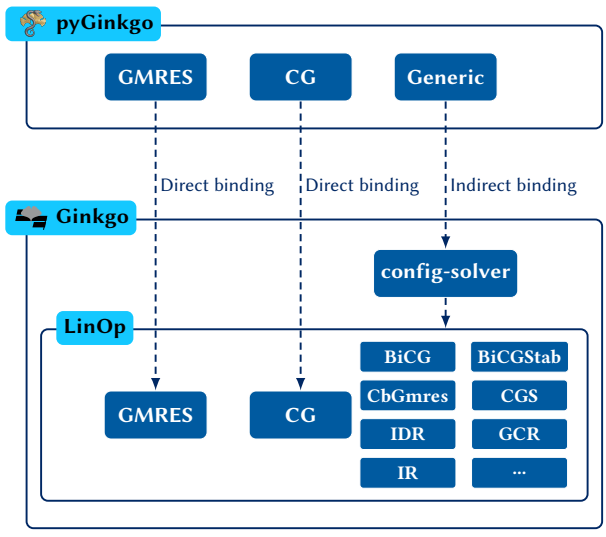


Figure 2: Overview of Solver Bindings

pyGinkgo leverages this approach to gain access to a rich set of solvers, factorizations, and preconditioners without the need to explicitly bind them. Additionally, this reduces the development pressure on the pyGinkgo side: whenever Ginkgo introduces a new functionality, which can be accessed via the generic solver entry point, pyGinkgo users only need to update to a recent version of Ginkgo without requiring explicit bindings. While the typical use case is to read the configuration from a file, pyGinkgo also implements a wrapper for the generic entry point that generates the configuration parameters in JSON format from Python's dictionary directly as also shown earlier in Listing 2, without depending on any temporary configuration files on disk.

A drawback of the configuration file approach is that the configuration files themselves can become difficult to implement and to verify, since currently no JSON schema for validation is available. Additionally, for developers, interacting with the data structures directly instead of writing JSON configurations can be preferable because of features like auto-completion or access to doc-strings of the classes and functions through Python's `help()` and `dir()` functions. Thus, pyGinkgo provides access to Ginkgo's solvers via the config-solver interface. Additionally, it also provides explicit bindings to commonly used solvers like GMRES, the direct solver, and triangular solvers, and to preconditioners like IC and ILU, which is illustrated in Figure 2. We plan to expand this set further.

5.1 Available Value and Index Types

To implement datatypes and algorithms in Ginkgo that support lower, higher, and mixed precision, Ginkgo relies heavily on C++ templates. However, the concept of templated types and the corresponding potential overloading of functions is not directly applicable to Python. For example, the functions `funcxx(int a)` and `funcxx(float a)` can have completely different implementations, whereas in Python, only a single `funcxx(a)` can exist. This issue stems from the language differences, and is resolved in practice via pre-instantiation of all possible template parameter combinations that the Python side might require. In pyGinkgo, when creating bindings to overloaded functions, the functions are disambiguated by appending the typename to the function, resulting in `funcxx_int(a)` and `funcxx_float(a)`.

This raises the issue of abstracting away from these parameters on the bindings side and simplifying the access for the users, as is done e.g. in NumPy or PyTorch. They implement both raw and abstracted away functions on C/C++ side. Our current approach consists of directly exposing instantiated C++ templates inside of `pyGinkgo.pyGinkgoBindings` module and then using them within the more abstracted Python API inside of `pyGinkgo` module. Thus, a function named `funcxx(a)` is implemented in the pyGinkgo module, which simply dispatches to either `funcxx_int(a)` or `funcxx_float(a)` based on the type of a.

In comparison to NumPy or PyTorch source code, Ginkgo has more templated parameters, which results in a very large number of instantiations. Thus, the decision to introduce a second layer of abstraction on the Python side originates from the fact that implementing it in C++ would have provided little to no performance benefits while drastically increasing the complexity and amount of code. Additionally, it allows for easier implementation of more sophisticated user algorithms on the Python side by utilizing direct access to the exposed C++ API.

Table 1: Overview of the available data and index types

Size (bytes)	Value Type	Index Type
2	`half`	
4	`float`	`int32`
8	`double`	`int64`

Currently, pyGinkgo supports three value and two index types, which are listed in Table 1.

5.2 Buffer-protocol and Interoperability with the Python Ecosystem

Python's buffer protocol [2] defines a standard for the direct access to an object's memory, without requiring a copy. This is particularly useful when integrating Python with low-level, high-performance code written in languages like C or C++. By exposing a direct memory buffer, the protocol enables efficient data sharing between Python and external libraries, improving both performance and interoperability.

NumPy supports the buffer protocol, allowing its arrays to be passed to pyGinkgo without duplication. This significantly enhances efficiency, as Python can act as a lightweight wrapper over low-level implementations.

As a result, NumPy objects can be passed to Ginkgo on the C++ side in the following ways:

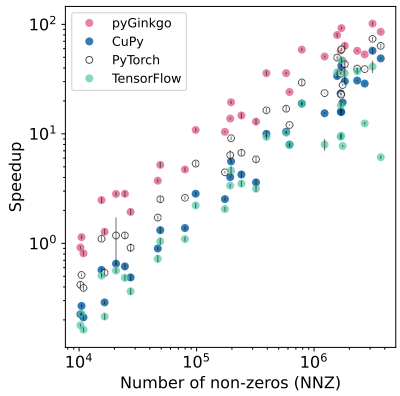
```
x = np.random.rand(dim).astype(np.float32)
x1 = pg.matrix.dense_float(executor, x) # via binding
x2 = pg.as_tensor(x, device=executor) # via dispatch
```

Interoperability is a key requirement for modern scientific software, particularly in Python, where users often construct complex workflows by combining multiple specialized libraries. A flexible interface that integrates naturally with tools like NumPy, SciPy, or PyTorch enables users to experiment and deploy algorithms without being locked into a rigid ecosystem. Like other major libraries, pyGinkgo offers very low computational overhead which makes interoperability feasible and efficient.

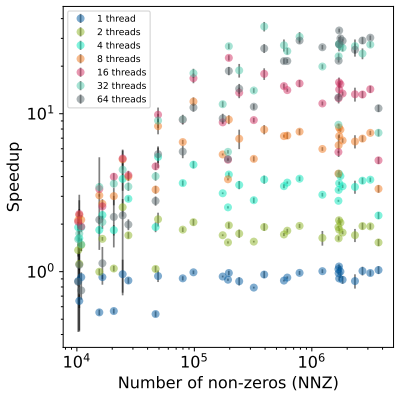
6 Performance Results and Analysis

To demonstrate the practical benefits of pyGinkgo, we benchmarked the performance of SpMV and solver kernels across widely used Python libraries, including PyTorch, SciPy, TensorFlow, and CuPy, on both CPU and GPU platforms. Benchmarks for SpMV kernels were conducted using a dataset of 30 sparse matrices from the SuiteSparse Matrix Collection [25]. Benchmarks for solver kernels were conducted using 40 sparse matrices. These test matrices cover a wide range of dimensions and densities, with dimensions up to 10^6 and densities below 1% in all cases except for five with a density greater than 1%. Since machine learning workloads primarily rely on SpMV in low precision, the SpMV benchmarks were conducted using single-precision. In contrast, most scientific computing workflows that require solvers demand double precision for accuracy,

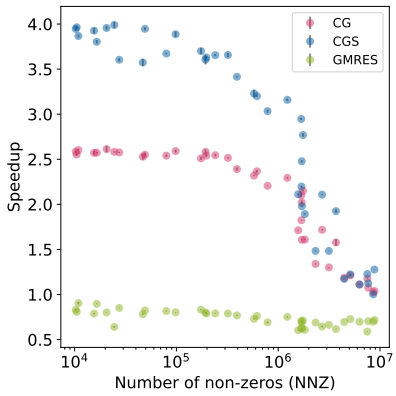
[2] https://docs.python.org/3/c-api/buffer.html



(a) SpMV on NVIDIA A100 GPU: Variation of speedup relative to SciPy with number of non-zeros

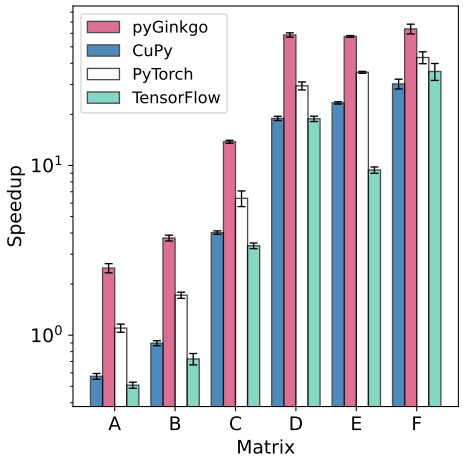


(b) SpMV on Intel Xeon Platinum 8368 CPU: Variation of speedup relative to SciPy with number of non-zeros

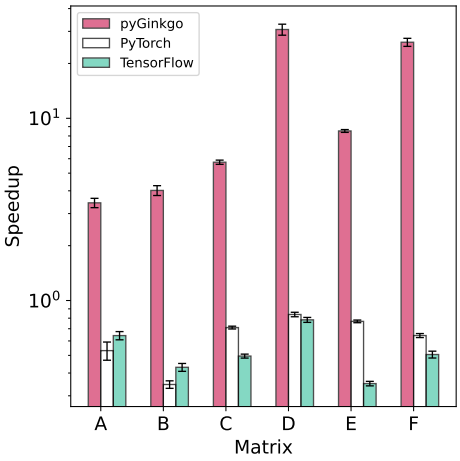


(c) Solver on NVIDIA A100: Variation of speedup relative to CuPy with number of non-zeros for 1000 solver iterations

Figure 3: SpMV and solver performance on different architectures



(a) SpMV on NVIDIA A100 GPU



(b) SpMV on Intel Xeon Platinum 8368 CPU

Figure 4: Variation of Speedup relative to SciPy for representative matrices on GPU and CPU

so the solver benchmarks were performed using double-precision. Benchmarks were conducted for both CSR and COO matrix formats.

We present the results for the best-performing matrix formats for each library. The benchmarks were executed on the HoreKa supercomputer, featuring Intel Xeon Platinum 8368 CPUs and NVIDIA A100 GPUs. A single CPU node on HoreKa consists of 2 sockets with 76 physical cores each. We assessed single-core and multi-core performance for CPUs, alongside GPU execution performance. We also compared pyGinkgo's performance with that of Ginkgo's on a set of 45 sparse matrices from the SuiteSparse Matrix Collection. This was done on NVIDIA A100 and AMD Instinct MI100 accelerators for both CSR and COO matrix formats.

All experiments presented in this paper were conducted using Python 3.11. For the NVIDIA backend, we used CUDA 12.2, loaded via the environment module `devel/cuda/12.2`. For the AMD backend, we used ROCm 6.2.2 loaded via the environment module `toolkit/rocm/6.2.2`. These module configurations were applied consistently during benchmarking to ensure a fair comparison across hardware platforms

6.1 SpMV

To evaluate pyGinkgo's performance for SpMV, we considered operations of the form $x = Ab$, where A is a sparse matrix from the dataset and b is initialized as a vector of random values. This setup was kept consistent across all the libraries.

6.1.1 GPU:. Figure 3a shows the SpMV speedups achieved by pyGinkgo, PyTorch, TensorFlow, and CuPy on an NVIDIA A100 GPU, relative to the baseline performance of SciPy running on a single CPU core. The matrices are ordered by increasing non-zero count.

Table 2: Test matrices and relevant attributes

Matrix	Dimension	NNZ	Name
A	25,503	1.55e+04	bcsstm37
B	46,772	4.68e+04	bcsstm39
C	25,187	1.93e+05	mult_dcop_01
D	131,072	7.86e+05	delaunay_n17
E	41,092	1.68e+06	av41092
F	32,1671	1.83e+06	ASIC320ks

Among all libraries, pyGinkgo consistently outperforms the alternatives, exhibiting near-linear scaling with respect to the number of non-zeroes (NNZ).

For the evaluated matrices, pyGinkgo reaches a peak performance of approximately 150 GFLOP/s, followed by PyTorch at 110 GFLOP/s, CuPy at 85 GFLOP/s, and TensorFlow at 50 GFLOP/s. While PyTorch achieves relatively high peak performance, it is still approximately twice as slow as pyGinkgo across most test cases. CuPy is generally 3–4 times slower, and TensorFlow exhibits the largest performance gap, being 2–14 times slower than pyGinkgo, depending on the matrix.

Figure 4a illustrates the speedup of selected matrices spanning several orders of magnitude in terms of NNZ. The relevant attributes of these representative matrices are listed in Table 2. Across all libraries, a clear trend of increasing speedup with increasing NNZ can be observed.

With pyGinkgo, we are able to run SpMV computations not only on NVIDIA GPUs but also on AMD GPUs, enabling a broader performance analysis across architectures. The performance comparison between NVIDIA A100 and AMD Instinct MI100 accelerators for SpMV is presented in Figure 5a. The plot compares both CSR and COO matrix formats. We observe that performance on NVIDIA A100 is slightly better than that on AMD GPU, particularly for matrices with larger NNZ.

6.1.2 CPU:. On a single CPU thread, we observe that SciPy performs better than all the other libraries, although its performance does not scale well with increasing number of threads. pyGinkgo, on the other hand, scales well with increasing number of threads, its speedup relative to SciPy across multiple threads is shown in Figure 3b. pyGinkgo's execution time for the SpMV kernel proves to be at least 7-35 times faster than SciPy for matrices with higher NNZ as we scale to 32 threads. Compared to PyTorch and TensorFlow, pyGinkgo is 10-60 and 30-90 times faster, respectively. As also shown for GPU earlier, the speedups across various libraries for the selected matrices is presented in Figure 4b. We observe that it is highly beneficial to use pyGinkgo for large matrices (D and F). For the matrix E, the speedup drops across all libraries because of its higher density. Comparing Figure 4a and Figure 4b, we observe that it is more efficient to use CPU instead of GPU for matrices with low NNZ (A and B).

6.2 Solvers

To evaluate pyGinkgo's performance for solvers, we considered linear systems of the form $Ax = b$, where A is a sparse matrix from the dataset, b is initialized as a vector of ones, and the initial guess x is a vector of zeros. This setup was kept consistent across CuPy, SciPy and pyGinkgo. We conducted a similar evaluation for solver kernels as we did for SpMV, using a benchmark set of 40 sparse matrices.

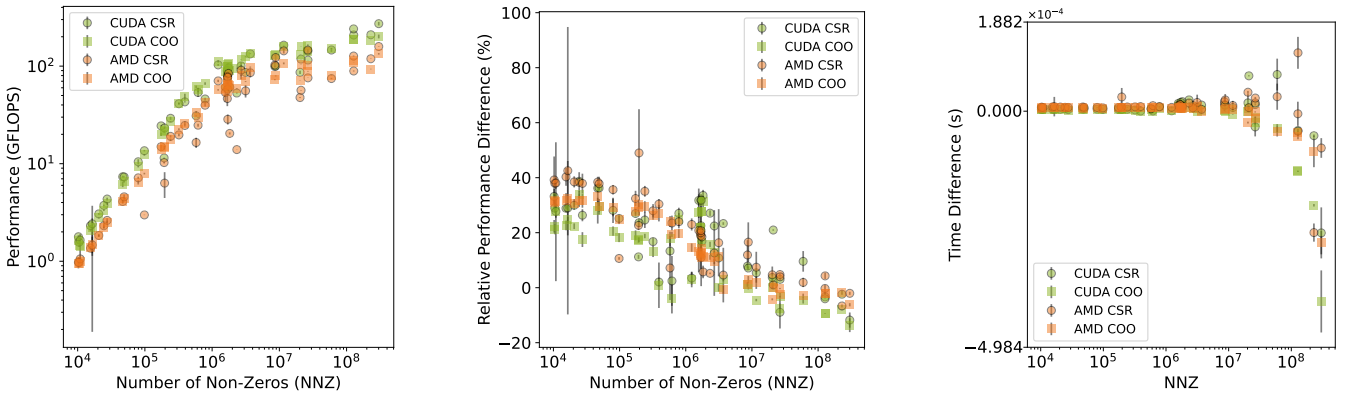
6.2.1 GPU:. While PyTorch and TensorFlow offer little support for sparse linear algebra, primarily supporting direct solvers, they do not provide implementations of iterative solvers. Consequently, our comparison in this section focuses on CuPy and pyGinkgo.

CuPy supports a limited set of iterative Krylov subspace solvers, specifically the Conjugate Gradient (CG), the conjugate gradient squared method (CGS), generalized minimal residual (GMRES), least squares with QR factorization (LSQR), least squares minimization of residuals (LSMR) and the Minimum Residual (MINRES) methods. Since there is no native support for preconditioners on CuPy's side, this analysis was done without a preconditioner. It is worth noting, however, that pyGinkgo exposes a rich set of both iterative solvers and preconditioners. Our benchmark experiments were done for CG, CGS and GMRES for both CuPy and pyGinkgo, performing 1000 solver iterations for each case. Many of the SuiteSparse matrices we tested were ill-conditioned, and as a result, several of them did not converge, especially in the absence of a preconditioner. Therefore, it was more meaningful to evaluate performance based on time per iteration rather than total time to convergence. However, time per iteration alone does not reflect the total time required to reach convergence.

pyGinkgo's speedup relative to CuPy for all three solvers is presented in Figure Figure 3c. CGS consistently achieves the highest speedup, particularly for matrices with fewer nonzeros, reaching up to 4 times than that of CuPy. CG offers moderate speedup of around 2.5 times across a wide range of NNZ values. We observe that the speedup for pyGinkgo decreases noticeably with the number of nonzeros, This result suggests possible overheads or optimization thresholds on CuPy's side.

Although pyGinkgo generally outperforms CuPy for CG and CGS solvers, CuPy shows slightly better performance for GMRES. Here, the restart factor for GMRES was chosen to be 30.

The GMRES implementations in CuPy and Ginkgo differ in several important ways. First, CuPy solves the least-squares problem associated with the Hessenberg matrix on the CPU, whereas Ginkgo performs the entire solution process on the GPU. Although GPU-based computation generally offers higher throughput, solving very small Hessenberg systems may actually be faster on the CPU due to low parallelization from dependence requirement in the triangular solver. Second, Ginkgo updates the Hessenberg matrix by using Givens rotations, while CuPy employs an orthonormal projection approach. Third, Ginkgo checks the residual norm after every update to the Hessenberg matrix, while CuPy only performs the check after the complete Hessenberg matrix is constructed, resulting in $restart - 1$ additional checks in Ginkgo. Additionally, Ginkgo reuses the computed Givens rotations to efficiently update the residual norm. These differences in strategy, particularly the more frequent residual checks and the full GPU implementation may contribute to CuPy's slightly faster performance compared to pyGinkgo in GMRES especially when using a fixed number of iterations rather than convergence-based stopping.



(a) pyGinkgo's SpMV performance with number of non-zeros

(b) Relative Performance difference of pyGinkgo versus Ginkgo

(c) Time difference of pyGinkgo compared to Ginkgo in seconds

Figure 5: Performance difference and time difference analysis relative to native Ginkgo implementation for SpMV on NVIDIA A100 and AMD Instinct MI100 accelerators for CSR and COO matrix formats

6.2.2 CPU:. We also executed these benchmarks on CPU to compare it with SciPy. pyGinkgo is around 3 - 8 times faster than Scipy in the case of CG for the same systems of matrices that we considered for CuPy. We obtained similar results for CGS and GMRES as well.

6.3 pyGinkgo versus Ginkgo

In addition to benchmarking against Python-based libraries, we also benchmarked pyGinkgo against Ginkgo's native SpMV to assess the overhead of the Python bindings. Results for the GPU backends are presented in Figure 5b and Figure 5c. We evaluated the performance and the corresponding time difference of pyGinkgo relative to the native Ginkgo implementation using the following metrics.

The relative performance difference is defined as:

$$P_{\text{overhead}} = \frac{P_{\text{Ginkgo}} - P_{\text{pyGinkgo}}}{P_{\text{Ginkgo}}} \times 100$$

where P_{overhead} represents the relative performance difference between pyGinkgo and Ginkgo, P_{Ginkgo} is the performance of Ginkgo in GFLOPS, and P_{pyGinkgo} is the performance of pyGinkgo in GFLOPS for the corresponding matrix format and hardware backend.

The time difference is computed as:

$$T_{\text{overhead}} = T_{\text{pyGinkgo}} - T_{\text{Ginkgo}}$$

where T_{overhead} is the additional execution time in seconds introduced by pyGinkgo, with T_{pyGinkgo} and T_{Ginkgo} denoting the SpMV execution times of pyGinkgo and Ginkgo, respectively.

6.3.1 NVIDIA GPU:. Results indicate that for NVIDIA GPUs, the performance overhead of pyGinkgo is around 30% for matrices with lower NNZ. It substantially decreases as number of non-zeros increase. For both matrix formats, overheads drop from approximately 25 – 35% at lower NNZ to below 10% for larger NNZ (NNZ $> 10^7$) for most cases. However, the time difference, as presented in Figure 5c, mostly remains in the order of $10^{-7} - 10^{-5}$ seconds for all test matrices, which is negligible and unlikely to be of practical concern, especially for matrices with low NNZ. In the cases for matrices with larger NNZ, the time difference can sometimes be below zero due to variability from system noise, which prevents perfectly consistent timing results. Although we followed the same benchmarking procedure on both the Python and C++ sides, the timing mechanisms differ: we use `steady_clock` from the C++ standard library and the time module in Python, both after explicit GPU synchronization. These differences in timer implementation and the effects of synchronization introduce some measurement uncertainty. Nevertheless, as shown in Figure 5b and Figure 5c, the results consistently indicate that pyGinkgo adds negligible overhead for large-scale problems on NVIDIA GPUs.

6.3.2 AMD GPU:. On AMD GPUs, the performance overhead introduced by pyGinkgo is slightly higher as compared to NVIDIA GPUs, particularly for the CSR format. At smaller matrix sizes, the percentage overhead can exceed 40% for a few cases, with notable fluctuations across both CSR and COO formats. The absolute time difference remains relatively low - typically within the range of $10^{-6} - 10^{-4}$ seconds for all test matrices. Similar to the observation on NVIDIA GPUs, the time difference below zero indicates that the machine noise, differences in timer implementation and the effects of synchronization lead to that difference. These results suggest that pyGinkgo incurs low overhead on AMD hardware as well, especially for matrices with larger NNZ.

7 Conclusion and Outlook

In this work, we brought pyGinkgo, a Python wrapper for Ginkgo, providing high-performance sparse matrix operations and linear solvers on CPUs and GPUs. Our benchmarks show that pyGinkgo significantly outperforms existing Python libraries such as SciPy, CuPy, TensorFlow, and PyTorch for sparse matrix-vector multiplication. For iterative solvers, pyGinkgo demonstrates performance comparable to that of CuPy, further validating its efficiency for workloads with sparse linear solvers. This makes pyGinkgo a strong

candidate for serving as the computational backbone of sparse scientific computing and machine learning workloads.

Until now, Python users have had limited access to optimized sparse operations across modern architectures, often sacrificing performance for simplicity. pyGinkgo changes this by combining Ginkgo's state-of-the-art, hardware-accelerated backend with a lightweight, NumPy-compatible, and Pythonic interface. Users are now able to use SpMV kernels, advanced iterative solvers, and preconditioners without writing C++ code. We are actively extending the library with new algorithmic capabilities, and we are also implementing iterative methods like the Rayleigh-Ritz algorithm entirely on the Python side to support advanced eigensolvers. Our framework's flexibility also allows users to implement custom algorithms easily by combining existing linear algebra operations accessible via pyGinkgo.

On the Ginkgo side, future work includes the integration of a convolution kernel, which would allow Ginkgo and pyGinkgo to support key operations required in image processing and convolutional neural networks. With continuous developments, pyGinkgo interface will become even more Pythonic and user-friendly, enhancing usability for the machine learning and scientific computing communities. In essence, pyGinkgo makes high-performance sparse computing accessible to the Python community. It opens the door for scientists, engineers, and machine learning practitioners to write high-level Python code while still benefiting from the raw speed and scalability of Ginkgo.

Acknowledgments

This work was performed on the HoreKa supercomputer and the NHR@KIT Future Technologies Partition testbed, funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Research, Technology and Space. This work is supported by the Helmholtz AI platform grant. The authors would like to thank the Federal Ministry of Education and Research and the state governments (www.nhr-verein.de/unsere-partner) for supporting this work/project as part of the joint funding of National High Performance Computing (NHR). We are also grateful to Dr. Yoshifumi Nakamura and Dr. Hitoshi Murai from RIKEN for their meaningful discussions and valuable insights. The authors acknowledge the use of AI-based tools for language editing and grammar improvement during the preparation of this manuscript.

References

[1] 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151. doi:10.1145/567806.567807

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://doi.org/10.48550/arXiv.1603.04467 Software available from tensorflow.org.

[3] Ahmad Abdelfattah, Natalie Beams, Robert Carson, Pieter Ghysels, Tzanio Kolev, Thomas Stitt, Arturo Vargas, Stanimire Tomov, and Jack Dongarra. 2024. MAGMA: Enabling exascale performance with accelerated BLAS and LAPACK for diverse GPU architectures. *The International Journal of High Performance Computing Applications* (2024-06 2024). doi:10.1177/10943420241261960

[4] David Abrahams and Ralf W Grosse-Kunstleve. 2003. Building hybrid systems with Boost. Python. *C/C++ Users Journal* 21, LBNL-53142 (2003). https://www.boost.org/doc/libs/1_87_0/libs/python/doc/html/index.html

[5] Martin S. Alnaes, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Software* 40 (2014). doi:10.1145/2566630

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[7] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. 2020. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software* 5, 52 (2020), 2260. doi:10.21105/joss.02260

[8] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. 2022. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Trans. Math. Software* 48, 1 (Feb. 2022), 2:1–2:33. doi:10.1145/3480935

[9] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. 2020. Load-Balancing Sparse Matrix Vector Product Kernels on GPUs. *ACM Trans. Parallel Comput.* 7, 1, Article 2 (March 2020), 26 pages. doi:10.1145/3380930

[10] Hartwig Anzt, William Sawyer, Stanimire Tomov, Piotr Luszczek, Ichitaro Yamazaki, and Jack Dongarra. 2014. Optimizing Krylov Subspace Solvers on Graphics Processing Units. (05-2014 2014).

[11] Hartwig Anzt, Yuhsiang M. Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. 2020. Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 26–38. doi:10.1109/PMBS51919.2020.00009

[12] David M Beazley et al. 1996. Swig: An easy to use tool for integrating scripting languages with C and C++.. In *Tcl/Tk Workshop*, Vol. 43. 74. https://www.swig.org/

[13] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39. doi:10.1109/MCSE.2010.118

[14] Nathan Bell, Luke N. Olson, Jacob Schroder, and Ben Southworth. 2023. PyAMG: Algebraic Multigrid Solvers in Python. *Journal of Open Source Software* 8, 87 (2023), 5495. doi:10.21105/joss.05495

[15] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/jax-ml/jax

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs.NE] https://arxiv.org/abs/1410.0759

[17] Maxwell D. Collins and Pushmeet Kohli. 2014. Memory Bounded Deep Convolutional Networks. arXiv:1412.1442 [cs.CV] https://arxiv.org/abs/1412.1442

[18] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.

[19] J. J. Dongarra, Jermey Du Cruz, Sven Hammarling, and I. S. Duff. 1990. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 18–28. doi:10.1145/77626.77627

[20] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. doi:10.1145/77626.79170

[21] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[22] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. 2005. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Softw.* 31, 3 (Sept. 2005), 351–362. doi:10.1145/1089014.1089019

[23] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. arXiv:2102.00554 [cs.LG] https://doi.org/10.48550/arXiv.2102.00554

[24] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11–Seamless operability between C++ 11 and Python. *URL: https://github. com/pybind/pybind11* (2017).

[25] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019), 1244. doi:10.21105/joss.01244

[26] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. 2010. Utilizing Recursive Storage in Sparse Matrix-Vector Multiplication - Preliminary Considerations. (2010), 300–305.

[27] NVIDIA Corporation. [n. d.]. *NVIDIA Collective Communications Library (NCCL).* NVIDIA. https://developer.nvidia.com/nccl

[28] NVIDIA Corporation. 2022. *cuSPARSE - GPU library APIs for sparse computation.* NVIDIA. https://docs.nvidia.com/cuda/archive/12.6.1/pdf/CUSPARSE_Library.pdf

[29] NVIDIA Corporation. 2023. *cuSOLVER -Direct Linear Solvers on NVIDIA GPUs.* NVIDIA. https://docs.nvidia.com/cuda/archive/12.6.1/pdf/CUSOLVER_Library.pdf

[30] NVIDIA Corporation. 2024. *cuRAND - Random Number Generation on NVIDIA GPUs.* NVIDIA. https://docs.nvidia.com/cuda/archive/12.6.1/pdf/CURAND_Library.pdf

[31] NVIDIA Corporation. 2025. *cuBLAS - Basic Linear Algebra on NVIDIA GPUs.* NVIDIA. https://docs.nvidia.com/cuda/archive/12.6.1/pdf/CUBLAS_Library.pdf

[32] NVIDIA Corporation. 2025. *cuFFT - GPU-accelerated Fast Fourier Transform (FFT) implementations.* NVIDIA. https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

[33] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman. 2017. CuPy : A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS).* https://api.semanticscholar.org/CorpusID:41278748

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://doi.org/10.48550/arXiv.1912.01703

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830. https://doi.org/10.48550/arXiv.1201.0490

[36] M. Sala, W. Spotz, and M. Heroux. 2008. PyTrilinos: High-Performance Distributed-Memory Solvers for Python. *ACM Transactions on Mathematical Software (TOMS)* 34 (March 2008). Issue 2. doi:10.1145/1326548.1326549

[37] SourceForge [n. d.]. *PySparse: A Fast Sparse Matrix Library for Python.* SourceForge. https://pysparse.sourceforge.net/contents.html Version 1.0.2.

[38] Suraj Srinivas, Akshayvarun Subramanya, and R. Venkatesh Babu. 2016. Training Sparse Neural Networks. arXiv:1611.06694 [cs.CV] https://arxiv.org/abs/1611.06694

[39] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. doi:10.1109/TPDS.2021.3097283

[40] Yuhsiang M. Tsai, Terry Cojean, and Hartwig Anzt. 2020. Sparse Linear Algebra on AMD and NVIDIA GPUs – The Race Is On. In *High Performance Computing*, Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 309–327. doi:10.1007/978-3-030-50743-5_16

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[42] Yulong Yan, Haoming Chu, Yi Jin, Yuxiang Huan, Zhuo Zou, and Lirong Zheng. 2022. Backpropagation With Sparsity Regularization for Spiking Neural Network Learning. *Frontiers in Neuroscience* 16 (2022). doi:10.3389/fnins.2022.760298