Software Development Aspects of Integrating Linear Algebra Libraries



Marcel Koch, Tobias Ribizel, Pratik Nayak, Fritz Göbel, Gregor Olenik, and Terry Cojean

Abstract Many scientific discoveries are made through, or aided by, the use of simulation software. These sophisticated software applications are not built from the ground up, instead they rely on smaller parts for specific use cases, usually from domains unfamiliar to the application scientists. The software library Ginkgo is one of these building blocks to handle sparse numerical linear algebra on different platforms. By using Ginkgo, applications are able to ease the transition to modern systems, and speed up their simulations through faster numerical linear algebra routines. This paper discusses the challenges and benefits for application software in adopting Ginkgo. It will present examples from different domains, such as CFD, power grid simulation, as well as electro-cardiophysiology. For these cases, the impact of the integrations on the application code is discussed from a software engineering standpoint, and in particular, the approaches taken by Ginkgo and the applications to enable sustainable software development are highlighted.

M. Koch (✉) · P. Nayak · F. Göbel · G. Olenik · T. Cojean
Scientific Computing Center (SCC), KIT, Karlsruhe, Germany
e-mail: marcel.koch@kit.edu

P. Nayak
e-mail: pratik.nayak@kit.edu

F. Göbel
e-mail: fritz.goebel@kit.edu

G. Olenik
e-mail: gregor.olenik@kit.edu

T. Cojean
e-mail: terry.cojean@kit.edu

T. Ribizel
Computational Mathematics, TUM, Heilbronn, Germany
e-mail: tobias.ribizel@tum.de

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_3

1 Introduction

Integrating numerical linear algebra libraries has considerable impact for application software. They provide new functionality, or optimize existing ones. Typically, application developers have little expertise in numerical linear algebra. Thus, through the integrations, the application software can benefit from functionality developed by experts in this field. The integrations come at a cost, though. They require development time, which is usually scarce for application scientists.

Both application and library developers have to ensure that the development cost stays minimal. This paper considers efforts within the numerical linear algebra library Ginkgo, and selected applications, with integrations of it, taken to minimize development and maintenance effort. This paper will first give a brief overview of the related work in Sect. 2. Then, in Sect. 3 Ginkgo's main concepts regarding integrations are presented. A corresponding view on integration from the application side is discussed next in Sect. 4. Several case studies are presented in Sect. 5. These highlight the key points from the two previous sections. Finally, a summary of the main points is provided in Sect. 6.

2 Related Work

Many software libraries provide numerical linear algebra functionality. Some of the most versatile libraries are PETSc [3] and Trilinos [11]. Both offer features for linear algebra, but they also encompass more areas, such as non-linear solver, mesh handling, parallel communication, or discretization frameworks. As such, they are attractive to a variety of applications. The PETSc user manual, for example, has more than 4000 citations on Google Scholar and Trilinos close to 1500.[1] Most of these are application papers that usually present only the results of integration of one of these libraries, and don't discuss their approach to the integration in detail. Papers on the library themselves, on the other hand, e.g. PETSc's AMG implementation [14] or an overlapping Schwarz method in Trilinos [10], often focus on new algorithmic improvements, leaving out again details on the integration aspects.

In [8, 19] the history of important numerical software libraries is presented. Dongarra's Turing lecture [8] focuses on the developments of LAPACK and adjacent packages, e. g. BLAS, ScalLAPACK, or SLATE, while [19] discusses the mathematical software library HSL. Both articles show how libraries can adapt to emerging needs of the community to stay relevant. Since the articles present the rich history of longstanding libraries, little place is left for technical details on integration. Nevertheless, Dongarra also notes the funding and personnel cost for maintaining such long-standing efforts.

The performance impacts of the integrations in Sect. 5 have already been published. Ginkgo's batched solver, which provides most of the performance benefits for

[1] From https://scholar.google.com/ accessed on 2024.05.06.

PeleLM, are described in [1, 2]. The paper [20] shows the performance of Ginkgo's direct solver in the context of the ExaSGD application. Additionally, the summary paper [6] provides an overview of the performance gains of multiple applications from the Exascale Computing Project (ECP)[2] that use Ginkgo, including PeleLM and ExaSGD. The OpenFOAM results using OGL are available in [16]. All of these papers, again, provide little consideration to the integration effort. Only the papers on OGL are an exception since they discuss the design of OGL which impacts the integration.

3 The Ginkgo Library

Ginkgo is a software library for sparse numerical linear algebra. It started as part of the ECP with the goal of providing an efficient and portable library for heterogeneous hardware. In particular, Ginkgo runs on all major GPU architectures. The respective backends are written in their native language to gain the most performance, i.e. CUDA for NVIDIA GPUs, HIP for AMD GPUs, and SYCL for Intel GPUs. On the algorithmic side, the main focus lies on iterative and direct solvers for sparse linear systems. Together with preconditioners and reorderings, Ginkgo is capable of solving complex problems efficiently on a variety of different architectures. This section introduces the core concepts of Ginkgo and discusses how they facilitate the integration of Ginkgo into application software.

3.1 Concepts

At its core, Ginkgo is a modern C++ library that values performance, flexibility, and sustainability. The aforementioned hardware specific backends offer excellent performance through optimized low-level kernels, while the clear type hierarchy offers flexibility in terms of functionality for the users. Through modern C++ patterns (requiring the standard C++14), the library provides a sustainable design. Finally, exhaustive testing and benchmarking ensure the correctness of the library.

Kernel Backends

The library is split into a core library and the different backend libraries. Performance critical kernels are implemented in the backends, and the higher level algorithmic structure is in the core library. The separation enables Ginkgo to directly target the hardware in question. This allows Ginkgo to easily implement complex algorithms on an abstract level, while gaining performance through hardware-optimized kernels. Compatibility layers, such as Kokkos [4], typically add some overhead compared to native implementations as shown in [7], which Ginkgo circumvents.

[2] https://www.exascaleproject.org/.

The separation also benefits encapsulation within Ginkgo. Any dependencies on the vendor specific languages are restricted to their respective backend and not propagated through the whole library. An application thus only needs a standard C++ compiler to compile with Ginkgo, even if the application will use for example Ginkgo's CUDA kernels.

The backend design not only offers performant kernels, it also provides flexibility in terms of the used hardware. Ginkgo can be built with all backends enabled at the same time. This makes it not a compile-time choice, which hardware should be used and which not, but a runtime choice instead. Which backend is chosen only depends on the implementation of the `Executor` interface used at runtime. The following example shows how different executors are used, depending on a runtime parameter:

```
void app_run(std::shared_ptr<gko::Executor> exec, ...);
// ...
{
  auto exec = cuda ? gko::CudaExecutor::create(...)
                   : gko::OmpExecutor::create(...);
  app_run(exec, ...);
}
```

Any Ginkgo operation on `app_run` will then use the kernels belonging to the dynamic type of `exec`. Furthermore, as `app_run` only depends on the abstract `Executor` interface, it is not necessary to recompile this function when switching between executors.

Linear Operators

The Ginkgo library is mainly designed around the linear operator concept. The type `LinOp` represents an abstract interface for the mathematical notion of a linear operator. This encompasses of course matrices, but also preconditioners and linear solvers, although they satisfy linearity only up to the achieved solving precision. All Ginkgo types implementing these concepts are implementations of the abstract `LinOp` type. The primary requirement is that the implementations provide a mapping between vector spaces.

The abstract linear operator concept offers great flexibility for choosing different algorithms. Through polymorphism, different implementations of, for example, matrix formats can be used interchangeably in algorithms that only rely on the operator application. This is true for most Krylov methods. So, users can easily switch out the matrix format to be used in a Krylov solver, and pick the format that is most suitable for their specific application. However, the flexibility is not restricted to matrix formats. The Krylov (and direct) solvers themselves, as well as the preconditioners are also implementations of the `LinOp` interface. Thus, users can easily try out a wide range of functionality, for example switching between applying a matrix or its inverse approximated by a solver:

```
void operate(const gko::LinOp*, ...);
// ...
{
  if(invert){
    auto solver = gko::solver::Cg<>::build()->generate(mtx);
    operate(solver.get(), in, out);
  } else {
    operate(mtx.get(), in, out);
  }
}
```

Design Patterns

Ginkgo uses design patterns to simplify the code, convey intent, and provide safe code. A key pattern is the use of smart pointers. These enable behavior close to automatic garbage collection, while still having completely deterministic behavior. Through them, the lifetime and ownership of polymorphic objects is managed. Either an object only has a single owner, then a `std::unique_ptr` keeps the object alive as long as the pointer stays in scope. The following gives a short example using one of Ginkgo's matrix types:

```
{
  std::unique_ptr<gko::LinOp> mtx = gko::matrix::Csr::create(exec, ...);
  // operate on mtx
  // ...
} // after leaving the scope mtx is automatically deleted
```

Alternatively, an object might have multiple owners, e.g. a matrix that is owned by a solver and a preconditioner. In this case, a `std::shared_ptr` keeps the object alive while at least one owner stays in scope. Once all owners are out-of-scope, the object is finally deleted. This creates a semi-automatic garbage collector behavior that prevents common memory issues, such as leaks, or double frees, while using a simple API with deterministic behavior and minimal overhead.

Lastly, Ginkgo is designed to offer great compatibility with other libraries through its transparent API. Part of the transparent API is to provide access to the stored data of the previously mentioned data structures through raw pointers. In contrast to smart pointers, these don't modify the ownership, but this introspection of the data layout allows other libraries to operate directly on data owned by Ginkgo. Compatibility in the other direction is also supported, since most data structures can be created by passing in raw pointers to existing data, resulting in Ginkgo objects that don't take ownership of the data. Both options are presented in the following snippet:

```
// extract data (and number of non-zeros) from a Ginkgo CSR matrix
double* values = mtx->get_values();
gko::size_type nnz = mtx->get_num_stored_elements();

// lend data to Ginkgo
double* app_ptr = ...; // given from application
auto view = gko::make_array_view(exec, app_size, app_ptr);
```

3.2 Facilitating Integrations

The aforementioned design concepts provide distinct benefits when integrating Ginkgo. Besides offering high-performance kernels, the runtime selectable backends provide great flexibility when interacting with other realizations of platform portability. Many portability approaches are configurable only at compile-time. This includes portability layers like Kokkos. An application that uses such a compile-time approach has to make sure that all components use the same configured backend. Since the backend choice for Ginkgo is delayed until runtime, it's easy to ensure that it uses the correct backend.

Furthermore, Ginkgo's type hierarchy, especially the `LinOp` interface, offers a flexible and extensible API. Not only can existing Ginkgo algorithms be switched out easily, applications can also provide their own implementation of Ginkgo's interface and then use their types in Ginkgo's algorithms. For example, this allows users to implement an operator for matrix-free stencil applications. The matrix-free operator can then be used in Ginkgo's Krylov solvers. Thus, applications gain flexible access to algorithms through Ginkgo, with a simple extension interface to inject specialized operations.

Finally, Ginkgo's transparent API greatly helps the data exchange with application code. By exposing raw pointers, an application can easily pass Ginkgo data, such as vectors, to other components without them relying also on Ginkgo, while the ownership stays with Ginkgo. Additionally, data that is owned by the application can be lent to be used within Ginkgo. This helps the application to create clear boundaries between their internal components without requiring expensive copies.

4 Integration Approaches in Application Software

The previous section presented approaches within Ginkgo to ease the effort of integrating it into an application. Now, the application side of the integration is considered. In general, an application can follow two different approaches to the integration. First, the application can use tight coupling with Ginkgo. The other option is loose coupling. These approaches are not specific to Ginkgo, but rather common approaches in the design of software applications. They reflect how deeply intertwined the application becomes with a library.

The strongest intertwining is the tight coupling. In this case, the application uses the Ginkgo data structures directly and might even use them outside the linear solver setting. Loose coupling leads to less intertwining by relying on abstract interfaces instead of Ginkgo directly, which can be considered as an application of the facade design pattern as mentioned in [9]. Ginkgo supports both modes, and the examples in Sect. 5 show different variations on them. The OpenFOAM example uses tight coupling, and openCARP transitioned from tight coupling with PETSc to loose

coupling for both PETSc and Ginkgo. Both ExaSGD and PeleLM use loose coupling, although in different variations.

4.1 Tight Coupling

Ginkgo provides the basic data structures necessary for linear algebra. Both dense vectors and sparse matrices with different formats are available to use as the backbone of an application's linear algebra component. Additionally, Ginkgo offers a storage type for one-dimensional arrays. The array type is directly linked to the `Executor` concept, which allows allocating memory on different devices besides the CPU. Together with simple APIs for copying data between arrays of different executors, Ginkgo provides the means for arbitrary storage using different devices.

In contrast to other popular numerical linear algebra packages, Ginkgo's scope is limited to linear algebra. Applications might require other concepts that are not strictly part of linear algebra, mesh handling for example. Longstanding libraries such as PETSc or Trilinos provide these additional functionalities. Although Ginkgo is more focused than those, it might still be used as the sole linear algebra backend. Since Ginkgo allows for direct data exchange without overhead, it can easily be used in conjunction with other libraries that cover the missing areas.

The Example in Listing 1 shows how an application could implement an implicit Euler time stepper by using a tight coupling with Ginkgo. The first thing to note is that the application class stores Ginkgo objects directly. The `advance` function then uses these objects and calls methods directly on them. Additionally, as the function expects Ginkgo objects as parameters, the calling code must also use Ginkgo objects directly. Thus, the dependency of Ginkgo extends further than this class.

The example highlights some of the advantages and disadvantages of the tight coupling. As Ginkgo types are used directly, no additional type hierarchy on the application side is necessary. But the implementation on the application side now deeply depends on Ginkgo. Generally speaking, implementations using tight coupling have their lifetime tied to the integrated libraries. If the integrated library stops receiving updates, the application is stuck with potentially out-of-date code. Either the application would have to update the library or replace it. Both would be major undertakings for the application.

Another not yet mentioned benefit is the simpler dependency management. The application always depends on the integrated library, so no special handling has to be developed if a library is missing. This will usually simplify the application's build system.

```
using Dense = gko::matrix::Dense<>;

class ImplicitEuler{
public:
  void advance(Dense* u_n, Dense* dx){
    // assemble right-hand-side f_ based on u_n

    Ainv_->apply(f_, x_);

    u_n->add_scaled(dx, x_);
  }

private:
  std::unique_ptr<Dense> x_;
  std::unique_ptr<Dense> f_;
  std::unique_ptr<gko::LinOp> Ainv_;
}
```

Listing 1: A tight-coupling application code example that implements an Euler time-stepper using Ginkgo functionality.

4.2 Loose Coupling

Compared to tight coupling, loosely coupled integrations will use Ginkgo only through indirection. First, the application needs abstract interfaces for its linear algebra component. The interfaces represent the boundary between the application code and the external libraries, such as Ginkgo. Interface code is called with application data, and expects application data as output if applicable. But internally, implementations of the interface will use Ginkgo types and functions. This might incur some overhead due to copying. Although both sides of the interface use the same conceptual representation, e.g. a matrix in CSR format, the underlying data layouts might differ, which requires copies. On the other hand, if the underlying data layouts are the same, no copy should be necessary. Thus, if libraries can communicate the used data layouts through their APIs, then interoperability can be improved through the elimination of copies, as will be seen in the case studies from Sect. 5.1.

In the context of linear algebra, these interfaces usually involve vectors, matrices, and solvers. These interfaces can provide multiple purposes. The solver interface provides the solution to a linear system, and the vector and matrix interface can provide conversions between the application's native format and Ginkgo, or other features such as assembly. Applications might provide multiple implementations of an interface for Ginkgo. If they differentiate between Krylov solvers on a type level, for example, then multiple implementations of the solver interface might be necessary to cover different Ginkgo solvers.

The interfaces might also provide simpler access to complex functionality. A solver interface could offer a method to regenerate the solver with a new matrix. The Ginkgo implementation for this would contain multiple lines of code, with different

Ginkgo functionality used, because Ginkgo doesn't provide the regenerate functionality through a simple API. Thus, the interface can hide complexity in Ginkgo, making it easier to use for the application specific purposes.

It is not necessary to provide all the mentioned interfaces. If only the solution to a linear system is required, without additional matrix or vector algebra, then the solver interface would be enough. The conversions between application and Ginkgo data are then handled purely internally. Ultimately, it's up to the application if all three interfaces are necessary, or if a subset suffices.

The example in Listing 2 shows an implementation of a contrived abstract solver interface. In this case, the application only provides the solver interface, so the in- and outputs are defined as application types. During the constructor, an application matrix is provided. This is converted into a Ginkgo matrix using an explicit copy, since it is assumed that different data layouts are used. For the solution method, application vectors are provided, but no copy is performed. Now, the assumption is that both Ginkgo and application vectors have the same data layout. This is typically easier to guarantee since Ginkgo uses a contiguous data layout with strides, which is widely used, e.g. in the BLAS API, or for Kokkos Views.

The only Ginkgo specific type in the interface is the executor `exec`. If an application is already using different execution spaces, e.g. CPU execution, and GPU execution, then a mapping between the applications' notion of execution spaces and Ginkgo's executors has to be defined. In the special case of Kokkos' execution spaces, Ginkgo already provides these mapping. For applications that were not using GPUs before, this might consist in mapping `std::strings` to Ginkgo's executors.

Compared to the tight coupling, loose coupling exhibits advantages that support the applications' freedom of choice. As the dependency is hidden through an interface, it's easy to change, remove, or add external libraries implementations. Furthermore, the dependency is restricted to a clear set of implementation files. Another benefit of the interfaces is the restriction of the integrated library to only the necessary functionality. On the other hand, the interface introduces an additional class hierarchy to manage, as well as boilerplate code in some instances. Also, managing the dependencies in the build system is more complex.

5 Case Studies

In the following section, examples are brought up to highlight the improvements applications can gain from integrating Ginkgo, as well as the differences in their integration approaches. First, two ECP applications, ExaSGD and PeleLM, are discussed. Both use loose coupling to integrate Ginkgo. While ExaSGD benefits from the direct solvers in Ginkgo, PeleLM exploits the batched iterative solvers. The third application is openCARP, which transitioned from a tight coupling with PETSc to a loose coupling with both PETSc and Ginkgo. It benefits from the enabled GPU acceleration. Finally, the OGL library uses tight coupling to provide Ginkgo solvers

```
class GinkgoCG: public AbstractSolver{
  using Dense = gko::matrix::Dense<>;
  using Csr = gko::matrix::Csr<>;
  using Cg = gko::solver::Cg<>;

public:
  void solve(const AppVector& b, AppVector& x) override
  {
    // Create views on the application data.
    // If the application data uses host memory, then instances of exec_
    // needs to be replaced by exec_->get_master().
    auto gko_x = Dense::create(
                    exec_, gko::dim<2>{b.num_rows(), b.num_cols()},
                    gko::make_array_view(exec_, b.total_size(), b.data()),
                    b.stride());
    // This is the const-correct version of the call above.
    auto gko_b = Dense::create_const(
                    exec_, gko::dim<2>{b.num_rows(), b.num_cols()},
                    gko::make_const_array_view(exec_, b.total_size(),
                                               b.data()),
                    b.stride());
    solver_->apply(gko_b, gko_x);
    // since views are used, the AppVector x already contains the result
    // otherwise an extra copy is necessary
  }

  GinkgoCG(std::shared_ptr<const gko::Executor> exec, AppMatrix& A,
           int max_it, double eps)
    : AbstractSolver(), exec_(exec)
  {
    gko::matrix_data<> md(gko::dim<2>{A.num_rows(), A.num_cols()};
    // Fill md by extracting the triplet (row-index, col-index, value) for
    // each non-zero in A and pass it to
    // md.nonzeros.emplace_back(row, col, value)
    auto gko_A = Csr::create(exec);
    gko_A->read(md);
    solver_ = Cg::build()
                .with_criteria(gko::stop::Iteration::build()
                                 .with_max_iters(max_it),
                               gko::stop::ResidualNorm<>::build()
                                 .with_reduction_factor(eps))
                ->generate(std::move(gko_A));
  }

private:
  std::shared_ptr<const gko::Executor> exec_;
  std::unique_ptr<Cg> solver_;
};
```

Listing 2: A loose-coupling application integration example, implementing a GinkgoCG solver by following an abstract solver interface.

for OpenFOAM. Again, through these solvers, GPU acceleration is enabled for parts of computational fluid dynamics (CFD) simulations using OpenFOAM.

5.1 ECP Applications

Both the ExaSGD and PeleLM applications benefited greatly from their integration with Ginkgo. As the major findings have been presented in the papers [1, 2, 20], and additionally in the summary paper [6], the performance gain for the applications are not discussed here. Instead, the focus lies on the integration approaches. The summary paper also mentions integrations with MFEM, and XGC. These are not followed up on here, since XGC is very similar to PeleLM in how Ginkgo is used, and MFEM doesn't classify as a science application, since it is again a library that other applications will integrate.

The **ExaSGD** project simulates the power grid of the US in order to make projections for future developments. At its core, it solves non-linear optimization problems, which is done through HiOP [17]. The resulting linear systems tend to be highly ill-conditioned, requiring the use of a direct solver. In addition, these systems have little structure, which makes prominent direct solvers such as SuperLU less effective. A new direct solver has been co-developed within Ginkgo to address these issues. The authors provide a detailed overview in [18]. By integrating Ginkgo into HiOP, platform portable direct solvers became available, as discussed in [20].

As mentioned earlier, ExaSGD (through HiOP) integrates with Ginkgo through loose coupling. More specifically, it only provides a solver interface for Ginkgo to implement. Thus, the integration requires a few lines of code, in total about 470 lines. Together with a few outside changes, e.g. to the build system or code examples, the initial implementation was done in 700 line of code (LOC) changes.[3] This marks a relatively low burden on HiOP's code base.

The integration is very similar to the example provided in Sect. 4.2. In both cases, an application matrix is copied over to a Ginkgo matrix during initialization, while views on application vectors are used in the solver application phase. Additionally, more application-specific interfaces are provided. These concern updating the values of a matrix, without changing the structure, and recomputing the factorization. Since the direct solver classes in Ginkgo don't provide this functionality directly, the implementation complexity is hidden by the interface.

During the interface initialization, the Ginkgo objects can be customized. Since HiOP only requires a specific subset of Ginkgo, the customization is rather limited. Only five options are provided in a mapping from HiOP options to Ginkgo customization. Besides that, many Ginkgo settings are fixed. The solver is always an LU solver, possibly wrapped in GMRES depending on the options, and MC64 is always used as an apriori reordering of the system.

[3] https://github.com/LLNL/hiop/pull/433/.

The **PeleLM** application [15] is used for CFD simulation, specifically for reactive flow in the Low Mach number regime. The discretization approach requires solving many independent ODEs. These are solved using Sundials [12], which requires the solution of many small independent linear systems. To solve these efficiently, batched iterative solvers have been developed in Ginkgo. The papers [1, 2] offer more details on the performance aspects.

Although the PeleLM integration through Sundials shares many similarities to the HiOP integration, it had a vastly different starting point. In contrast to HiOP, Sundials is mainly a C code. This introduces additional friction, as many modern C++ design principles which Ginkgo uses are not compatible with C design patterns. Integrating Gingko using a C API would lead to struggles due to C++ features that are not part of C, mostly RAII-based object management and templating. To alleviate the differences, Sundials introduced a new C++ API instead. This API takes advantage of templating to let users decide which concrete Ginkgo type should be used in the implementation of the interface.

The extra work is reflected in the integration effort. About 8000 LOC additions and 2500 LOC deletions were necessary in the initial pull request[4] (PR), which included the new C++ API. Despite the large-scale API changes, the changes solely related to Ginkgo are in the same order as HiOP. The Ginkgo implementation of the linear solver C++ API requires about 350 LOC. In addition, the matrix interface is also implemented for Ginkgo, again, in about 350 LOC.

Compared to HiOP's implementations, the Sundials ones are more minimal. Besides the expected apply and solve method, the matrix and solver interfaces only provide conversions to the Sundials C API, as well as access to the underlying Ginkgo objects. Similar to HiOP the apply and solve methods work with views on the application data. However, in contrast to HiOP, no configuration options are available, either as methods or as constructor arguments. Instead, the Ginkgo interfaces are constructed from existing Ginkgo objects. This allows users of Sundials to use every possible Ginkgo configuration option, without adding a burden to the interface by translating these options into Sundials options. Thus, users profit from Ginkgo's flexibility, while the maintenance effort on Sundials is minimal.

5.2 openCARP

openCARP is a simulator for cardiac electrophysiology. It provides tools for a wide range of simulations with different physical models. The simulations are discretized with finite element discretization, which, in the end, leads to solving linear systems. Through integrating Ginkgo, openCARP is capable of offloading the linear systems to GPUs [5].

Through the integration of Ginkgo, openCARP transitioned from a tight coupling to a loose coupling. Before Ginkgo was used, openCARP solely relied on PETSc

[4] https://github.com/LLNL/sundials/pull/205.

for their linear algebra need. As a consequence, openCARP was tightly coupled with PETSc, using their data types throughout their codebase. Components that are not related to linear algebra, for example the physics module, contained hard dependencies on PETSc.[5] These dependencies had to be weakened for Ginkgo to be utilized.

The transition to a loose coupling introduced abstract interfaces for vectors, matrices, and solvers. This required a large-scale effort. About 8000 LOC were changed in the initial PR[6] due to the dependency on PETSc in many modules. Since openCARP doesn't have a native vector or matrix type like HiOP or Sundials, the interfaces provide a multitude of functionality to cover the previous usage. Most essential are the typical BLAS level 1 and 2 type operations. Additionally, methods to help the assembly are present. Still, even with these additions, the resulting interfaces are relatively small compared to the initial effort, only approximately 300–500 LOC per interface were necessary.

Now, the openCARP codebase has become more sustainable. The linear algebra backend implementations are properly separated from other modules, and their dependencies are restricted to the linear algebra module. It has become possible to easily add other implementations for this component. As both the PETSc and Ginkgo implementation have a manageable size, 1600 lines, and 1000 lines respectively, the expected effort for additional implementations would again be of moderate size.

5.3 OpenFOAM

OpenFOAM (Open Field Operation And Manipulation) is a C++ framework for the development of numerical solvers and pre-/post-processing utilities for continuum mechanics problems, especially computational fluid dynamics (CFD) [13]. While various efforts were made to provide GPU support to OpenFOAM no commonly accepted approach exists [16] to this date. One approach is to offload certain parts, e.g. the linear solver, to GPUs via dedicated plugins. These plugins are separate from OpenFOAM. Their respective libraries are loaded at runtime to provide access to the additional functionality. Thus, OpenFOAM interacts through loose coupling with the plugins, while the plugins themselves are implemented using tight coupling. The OpenFOAM Ginkgo Layer (OGL) [16][7] is one of the plugins, providing linear solvers from the Ginkgo library.

OGL functionality is split into three parts. First, it provides a mapping between OpenFOAM and Ginkgo data structures, especially for sparse matrices. For these Ginkgo data structures, OGL has to ensure that they persist through the whole simulation. Lastly, the Ginkgo solvers and preconditioners can be configured at runtime

[5] See for example the file `physics/electrics.cc` in the release `v11.0`.

[6] https://git.opencarp.org/openCARP/openCARP/-/merge_requests/79.

[7] https://github.com/hpsim/OGL/.

through OGL. All combined, OGL allows OpenFOAM to efficiently use the platform portable solvers of Ginkgo.

Since OGL is a standalone library, it needs a considerably larger effort than the previously discussed integrations. The library consists roughly of 7000 lines of C++ code. Currently, the largest part of that is handling the object persistency. This requires knowledge of the underlying object storage as defined in Ginkgo, which is directly exposed in Ginkgo API. Without tight coupling, more complex interfaces and hierarchies would be required to describe the storage dynamically for different libraries. In general, the type hierarchy is shallow, since very little abstraction is necessary.

6 Summary

Many papers discuss the functionality gain that library integrations offer to application software. Although these gains are the reason why the integration is considered in the first place, the consequences on the development effort for the application are rarely discussed. This paper offers a discussion on these kinds of impacts. By the example of the library Ginkgo and its integration into multiple applications, the effects on application software are highlighted.

While Ginkgo offered each application clear gains, e.g. in the form of a platform portable linear algebra component or specialized batched solvers, those came at a cost on the application side. The Ginkgo integration had to be developed and needed to be maintained, which occupies valuable application developer time. But under the right circumstances, the integration effort is minimized. The key factor here is the loose coupling through well-defined interfaces on the application side. Applications that pursued this approach could integrate Ginkgo in less than 500 LOC per interface, while otherwise a large effort was required by the application, either to switch to loose coupling or build the tight coupling from the ground up.

For the examples given here, loose coupling offered the best approach for application developers. However, the examples only represent a very limited subset. Further evaluation of different integration approaches is necessary to provide both library and application developers with concise guidelines. Another aspect to further evaluate is the extension to other library types. While this paper only considers linear algebra libraries, the integration of libraries with different focuses might lead to other conclusions.

Acknowledgements This work was supported by the “Impuls und Vernetzungsfond of the Helmholtz Association” under grant VH-NG-1241. This work was supported by the European High-Performance Computing Joint Undertaking EuroHPC under grant agreement No 955495 (MICROCARD). Further, it was supported by BMBF through the SCALEXA initiative and the ExaSim (16ME0676K) and PDExa (16ME0641) projects. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

1. Aggarwal, I., Kashi, A., Nayak, P., Balos, C.J., Woodward, C.S., Anzt, H.: Batched sparse iterative solvers for computational chemistry simulations on GPUs. In: 2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), pp. 35–43 (2021). 10/gn3xcg. https://ieeexplore.ieee.org/document/9652814
2. Aggarwal, I., Nayak, P., Kashi, A., Anzt, H.: Preconditioners for batched iterative linear solvers on GPUs. In: Doug, K., Al, G., Pophale, S., Liu, H., Parete-Koon, S. (eds.) Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation, Communications in Computer and Information Science, pp. 38–53. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-23606-8_3
3. Balay, S., Abhyankar, S., Adams, M.F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W.D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M.G., Kong, F., Kruger, S., May, D.A., McInnes, L.C., Mills, R.T., Mitchell, L., Munson, T., Roman, J.E., Rupp, K., Sanan, P., Sarich, J., Smith, B.F., Zampini, S., Zhang, H., Zhang, H., Zhang, J.: PETSc/TAO users manual. Tech. Rep. ANL-21/39-Revision 3.21, Argonne National Laboratory (2024). https://doi.org/10.2172/2205494
4. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
5. Cojean, T., Göbel, F., Houillon, M.: Porting the opencarp cardiac electrophysiology simulation framework to GPU architectures (2022)
6. Cojean, T., Nayak, P., Ribizel, T., Beams, N., Tsai, Y.H.M., Koch, M., Gobel, F., Grützmacher, T., Anzt, H.: Ginkgo—a math library designed to accelerate ECP science applications. Int. J. High Perform. Comput. Appl. (Forthcom.)
7. Deakin, T., McIntosh-Smith, S., Price, J., Poenaru, A., Atkinson, P., Popa, C., Salmon, J.: Performance portability across diverse computer architectures. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 1–13 (2019). https://doi.org/10.1109/P3HPC49587.2019.00006
8. Dongarra, J.J.: The evolution of mathematical software. Commun. ACM **65**(12), 66–72 (2022)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Professional (1994)
10. Heinlein, A., Klawonn, A., Rheinbach, O.: A parallel implementation of a two-level overlapping schwarz method with energy-minimizing coarse space based on trilinos. SIAM J. Sci. Comput. **38**(6), C713–C747 (2016)
11. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the trilinos project. ACM Trans. Math. Softw. **31**(3), 397–423 (2005)
12. Hindmarsh, A.C.: SUNDIALS: suite of nonlinear/differential/algebraic equation solvers. Tech. Rep. UCRL-JC-149711, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States) (2002). https://doi.org/10.1145/1089014.1089020. https://www.osti.gov/biblio/15002968. ISSN: 0098-3500 Issue: 3 Volume: 31
13. Jasak, H., Jemcov, A., Tukovic, Z., et al.: Openfoam: A c++ library for complex physics simulations. In: International Workshop on Coupled Methods in Numerical Dynamics, vol. 1000, pp. 1–20 (2007)
14. May, D.A., Sanan, P., Rupp, K., Knepley, M.G., Smith, B.F.: Extreme-scale multigrid components within PETSc. In: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '16. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2929908.2929913

15. Nonaka, A., Day, M.S., Bell, J.B.: A conservative, thermodynamically consistent numerical approach for low Mach number combustion. Part I: Single-level integration. Combust. Theory Model. **22**(1), 156–184 (2018). https://doi.org/10.1080/13647830.2017.1390610. Publisher: Taylor and Francis _eprint
16. Olenik, G., Koch, M., Boutanios, Z., Anzt, H.: Towards a platform-portable linear algebra backend for OpenFOAM. Meccanica (2024). https://doi.org/10.1007/s11012-024-01806-1
17. Petra, C.G., Chiang, N., Wang, J.: HiOp—User Guide. Tech. Rep. LLNL-SM-743591, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory (2018)
18. Ribizel, T., Anzt, H.: Parallel symbolic cholesky factorization. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, pp. 1721–1727. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624253
19. Scott, J.: HSL@60: a brief history of the HSL mathematical software library. Tech. rep., STFC Rutherford Appleton Laboratory (2023). https://doi.org/10.5286/stfctr.2023002. http://purl.org/net/epubs/work/54066639
20. Świrydowicz, K., Koukpaizan, N., Ribizel, T., Göbel, F., Abhyankar, S., Anzt, H., Peleš, S.: GPU-resident sparse direct linear solvers for alternating current optimal power flow analysis. Int. J. Electr. Power & Energy Syst. **155**, 109517 (2024)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

