Meccanica
https://doi.org/10.1007/s11012-024-01806-1

RESEARCH

Towards a platform-portable linear algebra backend for OpenFOAM

Gregor Olenik · Marcel Koch · Ziad Boutanios · Hartwig Anzt

Received: 29 February 2024 / Accepted: 16 April 2024

Abstract Graphics processing unit accelerators have become a widespread technology on modern high performance computing clusters for increasing the performance of scientific computing algorithms. Despite early efforts to adopt linear solvers that utilize graphics processing units for OpenFOAM, to this date no widely accepted approach has gotten traction. In recent years, the number of different vendors providing graphics processing units accelerators has grown, and as of the writing of this paper, no commonly accepted, unified approach to leverage accelerators exists. This makes platform-portable solutions to increase the efficiency of graphics processing units offloading techniques desirable, and an important research topic. In this work, we investigate a platform-portable solution using the Ginkgo sparse linear algebra library.

Keywords Computational fluid dynamics · OpenFOAM · Ginkgo · GPU offloading · Linear solver · Platform-portability

1 Introduction

To serve the ever-increasing demand for computing power, general purpose graphics processing units (GPUs) have become an integral part of most high performance computing (HPC) leadership systems. In most cases, the GPUs are of the discrete server type that are attached to the nodes as co-processors, and provide the lion's share of the compute performance. Here, the GPU plays the role of an accelerator. In some HPC systems, the host central processing units (CPUs) are left with the task of orchestrating the communication. As a consequence, CPUs with less compute power than what was present in the CPU-only HPC systems of the past can be used. Initially, NVIDIA was the sole vendor that developed GPUs for general-purpose computing, complemented with a general programming framework, named CUDA. Nowadays, leadership HPC systems like Aurora and Frontier are equipped with GPUs from Intel and

All the authors equally contributed to this work.

G. Olenik (✉) · M. Koch · Z. Boutanios
Scientific Computing Center, Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
e-mail: gregor.olenik@kit.edu

M. Koch
e-mail: marcel.koch@kit.edu

Z. Boutanios
e-mail: ziad.boutanios@kit.edu

H. Anzt
School of Computation, Information and Technology, Technical University Munich (TUM), 80333 Munich, Bavaria, Germany
e-mail: hartwig.anzt@tum.de

H. Anzt
Innovative Computing Lab, University of Tennessee (UTK), Knoxville, TN 37996, USA

Published online: 12 June 2024

Table 1 Selected GPU linear algebra solutions for OpenFOAM

Library	Approach	Parallelization model
OGL[1] https://github.com/hpsim/OGL	Plugin	Ginkgo, OMP, CUDA, HIP, SYCL
YoctoFOAM[2] https://gitlab.hpc.cineca.it/openfoam/yoctoFOAM-herd/yocto foam-core	Fork	CUDA
OpenFOAM_HMM[3] https://github.com/ROCm/OpenFOAM_HMM	Portability	OpenMP
PETSc4FOAM[4] https://develop.openfoam.com/modules/external-solver	Plugin	PETSc
RapidCFD[5] https://github.com/Atizar/RapidCFD-dev	Fork	CUDA
PARALUTION[6] https://www.paralution.com	Plugin	Paralution, OpenCL, OMP, CUDA
foamExtend[7] https://sourceforge.net/projects/foam-extend/	Fork	CUDA

AMD respectively. This poses not only a challenge to redesign scientific software such that it can utilize modern HPC systems effectively, but also ideally in a platform-portable way so that the software can run on all available systems.

For computational fluid dynamics (CFD), three approaches are possible. The first one is to write computational kernels in a GPU native computing language such as CUDA [1], HIP [2], or SYCL [3]. The second consists of leveraging portability solutions like OpenMP [4] or Kokkos [5, 6] in the original code stack. The third approach is to offload the computationally expensive parts of an existing code to the GPUs using specialized linear algebra libraries such as PETSc [7] or Ginkgo [8]. The computationally expensive parts generally consist of the solution of the linear transport equation. In particular, the solution of Poisson equations such as the pressure equation is an important factor. It is solved in algorithms for incompressible flow solvers of the SIMPLE [9] and PISO [10] families, the latter being also applicable to compressible flow. Solvers based on such algorithms are commonly referred to as pressure-based solvers in the CFD community. One CFD toolkit that makes use of pressure-based solvers is OpenFOAM [11], widely used in academia and industry, and this paper deals exclusively with the subject of GPU computations in OpenFOAM. To avoid confusion, throughout the manuscript, OpenFOAMsolvers like icoFoam or pimpleFoam will be referred to as solver applications, whereas matrix solvers like PCG or GAMG will be referred to as linear solvers.

Several past projects addressed the need to offload OpenFOAM's linear algebra computations to GPUs. Early work on GPU computing for OpenFOAMinclude cufflink [12], the speedit version of OpenFOAM [13], and using PETSc as linear solver backend via the PETSc4FOAM plugin [14, 15]. Others have also used NVIDIA's AMGx [16–18], and the Ginkgo library for offloading [19]. A list of the important active projects is given in Table 1, and they follow one of the three general approaches mentioned above.

The first approach of rewriting computational kernels in a native GPU language is referred to as Fork. The forking approach comes at the cost of high development effort that needs to be repeated for a multitude of computational kernels, and is hardware vendor-dependent when it uses a specific GPU programming language. We note that HIP and OpenCL [20] make it possible to compile code for both NVIDIA and AMD GPUs, but this approach is not widely used. However, the forking approach introduces the least runtime overhead in terms of interfacing external libraries and modifying data structures.

The second approach of leveraging portability layers in the original code stack, and compiling directly for the GPU, face the challenge of adapting to the particular code features. Take OpenMP for example, with several algorithms implemented in OpenFOAMbeing essentially sequential in nature. For preconditioners like IC or ILU a simple annotation-based offloading strategy won't guarantee sufficient performance on GPUs. Thus, to run efficiently on GPUs different algorithms are necessary for optimal performance. This approach is listed under Portability, but it clearly involves forking the original code as well.

The third approach of offloading the computationally-expensive parts of the simulation is referred to as Plugin. The plugin approach requires writing an external plugin in a mainstream high-level language like C++, and incurs the overhead of offloading after

the matrix assembly. It can also require unnecessary communication between the host and the GPU device compared to the forking approach. The plugin approach is generally most promising if sufficient time is spent in the linear solver itself. Here, the offloading target could be the Poisson pressure equation only, while rapidly converging transport equations like the momentum equation are solved on the CPU. Another advantage of the plugin approach is that it makes available a set of third-party, highly tuned, and extensively tested GPU solvers, eliminating the associated burdens of development and maintenance.

Several of the mentioned projects, however, seemed to be discontinued or in most cases support only NVIDIA GPUs. Thus, this work focuses on implementing a plugin named the OpenFOAM Ginkgo Layer (OGL) [19, 21] to provide access to Ginkgo [22] acting as a linear solver backend. Ginkgo is a free and open-source library, which is under active development and has platform portability as one of its central design goals.

The paper is divided into three main sections. Firstly, the general constraints and limitations of the plugin approach used for OGL are discussed. Secondly, the details of how OGL allows for efficient use of the Ginkgo linear algebra library are discussed. Lastly, performance results on three different HPC machines equipped with Intel, AMD, and NVIDIA GPUs are presented. The performance results demonstrate that using Ginkgo as a linear solver backend via OGL allows offloading for all relevant GPU hardware commercially available today.

2 Constraints and limitations of the plugin offloading approach

In this section, the general considerations on the constraints and limitations of the plugin approach are presented.

The plugin approach as outlined in Sect. 1 is limited to the scope of the solve method of the `scalarField` or `vectorField` instance. Thus, only the solution of the linear system can be offloaded. Offloading the complete process of matrix assembly and solving the linear system is currently not supported by this approach. Naturally, this limits the attainable speedup for solving a transport equation, which can be expressed by Amdahl's law as $S = \frac{1}{1-p}$, where the acceleratable proportion is given by $p = t_{\mathrm{LS}}/\left(t_{\mathrm{LS}} + t_{\mathrm{MA}}\right)$, with t_{LS} being the time required to solve the linear system and t_{MA} the time required for matrix assembly. Hence, achieving a speedup of a factor 10, for example, solely by accelerating the linear solver, is only possible if 90% of the runtime is spent in the linear solvers.

As mentioned above, the linear solvers are the key part of the offloading approach via plugins. To discuss how OpenFOAM's default linear solvers can be replaced by solvers provided by a third-party library, the call graph to a linear solver is outlined first.

Every solver application (e.g. icoFoam or pimpleFoam) calls for every transport equation the linear solver either via a free function call of type `Foam::solve(fvMatrix<Type> &fvm)` or as member function call like `fvMatrix<Type>.solve(...)` from its corresponding matrix class, e.g. `fvScalarMatrix`, within its time loop. The former, however, just dispatches to the latter via `fvm.solve()`. Through further dispatching calls, references to the corresponding solution vector[1] and right-hand side[2] are obtained. Next, the solver call is forwarded either to a segregated or coupled solver. In the following, only segregated solvers are considered, since the majority of the compute resources are spent for the segregated pressure solve step in the simulation cases considered herein.

During this process, an object of type `lduMatrixSolver` is constructed, which besides some sanity checks also looks up the `constructorPtrTable` for a solver specified by the solver keyword in the corresponding section of the `fvSolution` file. The `constructorPtrTable` is a typical entry point for plugins like OGL or PETSc4FOAM. When the aforementioned plugins are loaded at startup time, for example by adding `libs ("libOGL.so");` to the `controlDict`, the plugin inserts pointers to the constructors of the solver classes provided by the plugin. One simple way to insert the pointers is via the `defineTypeNameAndDebug` macros, as shown in Lst. 1 for *GKOCG* as a solver for symmetric matrices.

[1] Typically named psi.

[2] Typically named source.

Listing 1 Usage example for creating run time selectable solver for plugins via the defineTypeNameAndDebug macro and addsymMatrixConstructorToTable.

```
defineTypeNameAndDebug(GKOCG, 0);

lduMatrix::solver::addsymMatrixConstructorToTable<GKOCG>
    addGKOCGSymMatrixConstructorToTable_;
```

After constructing an instance of the specified solver using the pointer from the `constructorPtrTable`, a member function named `solve` is called. Hence, the third-party linear solver classes are required to implement a member function with the signature shown in Lst. 2

Listing 2 Required function signature for linear solver call implementations.

```
virtual solverPerformance solve(
    scalarField &psi,
    const scalarField &source,
    const direction cmpt = 0
) const
```

Here the `scalarField &psi` refers to the solution vector and `source` to the right-hand side. The `scalarField psi` is an out parameter, i. e. the solution of the linear system can be written directly into psi. Properties of the solution process, such as residual norms and the number of iterations, are returned via the `solverPerformance` data structure. After solving the linear system, the instance of the solver class is deallocated and reconstructed on subsequent solve calls. This is an important point for an efficient plugin implementation, since intermediate computations need to be stored via the object registry to avoid costly recomputation.

3 Implementation details of the OGL offloading approach

This section discusses the considered offloading approach using the OGL library to make Ginkgo's solver available for OpenFOAM. Furthermore, it addresses some of the challenges mentioned in Sec. 2 and discusses how to limit the effect of the aforementioned constraints. This includes interfacing with Ginkgo to guarantee platform portability, a discussion on device persistent data structures to reduce the amount of data transfer, and efficient implementation of the stopping criterion to avoid unnecessary evaluations of the scaled L1 norm.

3.1 Offloading procedure

After constructing the solver class from the plugin, the following central steps are performed:

1. At first, the OpenFOAM system matrix is converted from LDU format to an appropriate format for the GPU. For OGL this can be either the coordinate list (COO), compressed sparse row (CSR), or the Ellpack (ELL) sparse matrix format.
2. Since the sparsity pattern of the system matrix will usually remain constant over the course of a simulation, the row and column indices of the system matrix on the GPU are preserved and only the values are updated. For this, a mapping between the LDU matrix entries and the COO/CSR matrix entries is required.
3. Furthermore, the right-hand side and the solution vector are copied to the device. While the solu-

tion vector is copied back to OpenFOAM after the solution process, it is also re-used as the initial guess for the next time step.
4. Construction of the solver and preconditioner objects from the Ginkgo library.
5. Call the corresponding linear solver and pass the solution vector back to OpenFOAM.

3.2 Overview of the Gingko linear algebra library

The offloading approach presented here relies on several important aspects of the Ginkgo[3] library [22]. Ginkgo is a math library for linear algebra written in modern C++. Its main focus lies on sparse linear algebra for CPU multicore and GPU architectures by implementing hardware-specific kernels in their native languages, i. e. CUDA (for NVIDIA GPUs), HIP (for AMD GPUs), OpenMP (for general-purpose multicore processors, such as those from Intel, AMD, or ARM), and SYCL for (Intel GPUs). It supports a variety of high performance linear algebra solvers and preconditioners suitable for CFD simulations. Furthermore, Ginkgo is an attractive candidate as a backend for OpenFOAM since it is open-source and comes with a permissive BSD 3-clause license, is part of the extreme-scale Scientific Software Development Kit (xSDK) [23], and has already been integrated as a backend into other simulation libraries like deal.ii [24], mfem [25], and openCarp [26, 27]. This implies not only that users can find Ginkgo pre-installed on large machines supporting xSDK, but also that they can use the Ginkgo technology in commercial OpenFOAM projects. The following key data structures are used within OGL:

- Ginkgo container types like `gko::array` and `gko::vector`, to handle memory transfer between OpenFOAM's host-based data and the GPU-based data. Here, the `gko::vector` type is used to represent distributed vectors for MPI parallelized runs.
- Ginkgo executor classes for an abstraction of the memory location and available device type.
- Linear solver and preconditioner classes from Ginkgo as an efficient and GPU-focused implementation of algorithms for preconditioning and solving linear systems.

[3] Available at https://github.com/ginkgo-project/ginkgo.

3.3 Device persistent data structures

A key principle for increasing performance with offloading is to reduce communication between the offloading target and the host to a minimum. To solve a linear equation

$$Ax = b \tag{1}$$

on a dedicated accelerator device, the amount of memory that needs to be transferred to the device is given by

$$m_{\text{mat}} = \left(s_{\text{scalar}} + 2s_{\text{label}}\right) n_{\text{NNZ}} \tag{2}$$

$$m_{\text{vec}} = 3 n_{\text{DOF}} s_{\text{scalar}} \tag{3}$$

where m_{mat} and m_{vec} are the memory transferred in bytes for the matrix and vectors, respectively. Furthermore, s_{scalar} and s_{label} are the sizes of a scalar value and the single precision label type respectively. The number of non-zero entries in the system matrix is given by n_{NNZ} and the number of cells by n_{DOF}. Note that Eq. 3 considers three transfers of scalar arrays of length n_{DOF} since initially x and b are copied from the host to the device and, after solving the linear equation, the solution vector x needs to be copied back to the host. However, for typical CFD applications, the linear equations Eq. 1 are solved multiple times throughout a simulation, for example when advancing in time for transient simulations or within outer loops for steady-state simulations. Therefore, if the computational grid is fixed throughout a simulation, the amount of memory transferred between host and device can be reduced by only updating the previously copied matrix on the device and reusing the solution vector as an initial guess in the subsequent linear solver application.

$$m_{\text{mat}} = s_{\text{scalar}} n_{\text{NNZ}} \tag{4}$$

$$m_{\text{vec}} = 2 n_{\text{DOF}} s_{\text{scalar}}. \tag{5}$$

For double precision scalar values of 8 bytes and 4 byte label sizes, considering a typical ratio of $\frac{n_{\text{NNZ}}}{n_{\text{DOF}}} = 7$ [4] the ratio between copying all arrays every time and reusing existing memory on the device is approximately 0.52. Thus, almost half of the data transfer can be avoided by reusing existing sparsity patterns and performing updates of the system matrix. Further

[4] Based on a 3D stencil, where each cell has six neighbors.

optimizations can be achieved by employing compression algorithms on the transferred data or exploiting existing symmetry when possible. However, these approaches are not within the scope of this work.

Reusing the system matrix and vectors on the GPU device on subsequent linear solver calls needs the implementation of additional functionalities into OpenFOAM. Since the `lduMatrixSolver` object is deallocated after the linear solver call has been completed, any intermediate computations and data structures on the GPU would also be deallocated if no further measures are taken. The principal idea to avoid costly recomputation is to use OpenFOAM's object registry to store smart pointers to Ginkgo data types that contain data on the accelerator. As long as the smart pointer is kept in the object registry and the reference count does not decrease to zero, the data on the accelerator is not deallocated and can be retrieved on subsequent solver calls. The full workflow for a device persisting Ginkgo matrix is shown in Fig. 1. When the constructor of a device persisting class is called, the constructor checks whether an object with the requested name is already registered in the object registry. If that is not the case, the device persisting class performs the initialization of the managed class. Otherwise, it will perform a less memory-intensive update, reusing the persisting data.

To explain 1 in more detail, consider a typical control flow for a simulation with multiple linear solves. During its first encounter, the persistent matrix does not exist, and thus the "no" branch in Fig. 1 will be taken. The persistent system matrix will then be initialized in the following two steps. First, a mapping between OpenFOAM LDU matrix coefficients and matrix coefficients in row-major order on the device is computed. From that the distributed Ginkgo matrix can be initialized and all required data structures are stored in the object registry. Here, we use the object registry of the mesh instance, thus as long as the mesh instance exists the same sparsity pattern and any corresponding mappings can be reused without any re-computation and data transfer between host and device. The persistent matrix can now be used to solve the linear system with Ginkgo solvers. In the subsequent linear solve encounters, the object now exists, so the "yes" branch is chosen. Since the object registry contains the required objects, they can be retrieved and updates of the matrix coefficients can be performed using the stored LDU-COO mapping.

To facilitate this workflow of storing and retrieving data via the object registry automatically, a

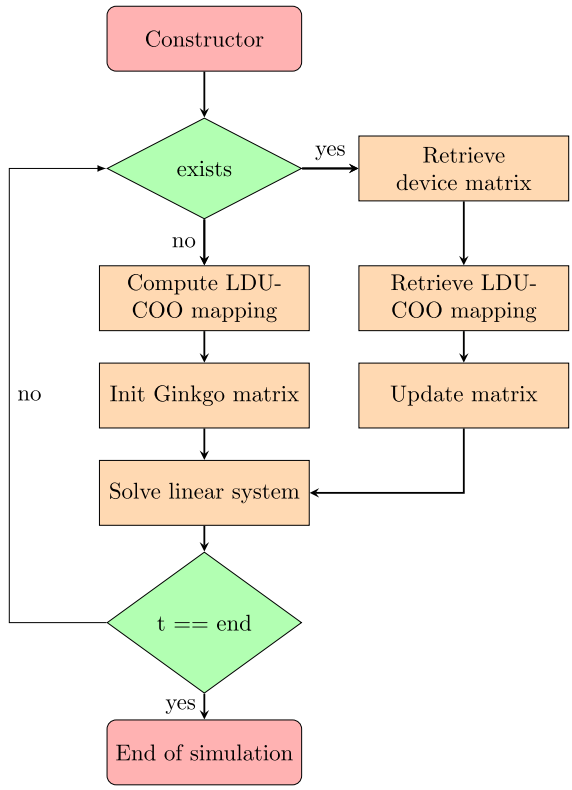

Fig. 1 Procedure for initializing or reusing device persisting system matrices

base class named `PersistentBase` was implemented[5]. A pseudo code implementation is given in Lst. 3. The base class requires two template arguments, `T` which defines the underlying type for which `std::shared_ptr<texttttT>` is stored in the object registry and `initFunctor` which is a functor that provides a `init|` and a `update` function. When the constructor of `PersistentBase` is called, it searches in the given object registry whether an object with the specified name already exists. If an object with that name was found and no explicit update is requested via the `update` parameter, the shared pointer is retrieved from the object registry. In contrast, if an explicit update is requested, the update function from the functor is called using the pointer to the original object. Finally, if no object with the given name is found in the object registry, initialization of the persistent object is performed via the functors init function.

[5] https://github.com/hpsim/OGL/tree/dev/DevicePersistent/Base.

Listing 3 Pseudo code showing the implementation of the PersistentBase class.

```
template<class T, class InitFunctor>
class PersistentBase {
...
public:
  PersistentBase(const word name, const objectRegistry &db,
                 const InitFunctor f, const bool update)
    : name_(name), db_(db),
      from_registry_(db.foundObject<regIOobject>(name)),
      init_functor_(f) {
        if (from_registry_) {
          if (update) {
            persistent_object_ =
              db_.lookupObjectRef<DevicePersistentBase<T>>(name)
                .get_ptr();
              f.update(persistent_object_);
            } else {
              persistent_ =
                db_.
                  lookupObjectRef<DevicePersistentBase<T>>(name)
                  .get_ptr();
          }
        } else {
          auto po = new DevicePersistentBase<T>(
              IOobject(...), f.init()
          );
          persistent_ = po->get_ptr();
        }
    }
};
```

This allows the base class to check whether the managed object is accessible via the object registry and initialize or update it if required. A simple use case for the device persistent class is the `PersistentVector<T>` class that derives from `PersistentBase<GkoVector<T, VectorInitFunctor<T>>`. Hence, it stores a shared pointer to a Ginkgo distributed vector for elements of type `T` and provides a function for initialization and updating the managed vector. Listing 4 shows the constructor call for the persistent vector class. Here, the first parameter `psi.begin()` gives access to data which needs to be copied to the accelerator device, the field name parameter is used as a key in the object registry, the `db_` parameter is a reference to the mesh object registry which stores the shared pointer, `this->get_exec()` provides access to the executor responsible for data access, and `updateInitGuess` defines whether the data needs to updated on the next constructor call

Table 2 Overview of GPU and CPU hardware of the utilized systems

System	CPU (total number of cores)	GPU
NLA	2 AMD EPYC 7302 (32)	8 AMD MI100
HoreKa	2 Intel Xeon Platinum 8368 (76)	4 NVIDIA A100-40 GPU
SuperMUC-NG	2 Intel Xeon Platinum 8480+ (112)	4 Intel Data Center GPU Max 1550

Listing 4 Simplified constructor call for a PersistentVector

```
PersistentVector<scalar> dist_x{
    psi.begin(), // original host data address
    this->fieldName(), // name to search for in object registry
    db_, // here the mesh object registry
    this->get_exec(), // executor handling where data is allocated
    updateInitGuess, // bool whether update is needed, default false
};
```

3.4 Solver generation and execution

Listing 5 gives a minimal code example of how a preconditioned solver, here a preconditioned CG solver is created. In the first step, a solver factory is created, which allows to optionally set parameters like the stopping criterion or the preconditioner. Within OGL this is handled for the individual solvers in the `Solver` directories and can be accessed via the `create_dist_solver` method, shown in Lst. 6.

In the next step, the actual solver instance is created via the `generate` call and the distributed matrix object called `dist_A_v`. Next, the linear system is solved using the solver `apply` method with the distributed RHS vector named `dist_b_v` and the distributed initial guess vector, `dist_x_v`. Finally, the call to `dist_x.copy_back()` copies the result data back to OpenFOAM via the pointer that initially served for copying the input data.

Listing 5 Example code creating a Ginkgo solver.

```
using cg = gko::solver::Cg<scalar>;

auto cg = cg::build()
    .with_criteria(stoppingCriterionVec_)
    .with_generated_preconditioner(precond)
    .on(exec);
```

Listing 6 Example code executing a Ginkgo solver from OpenFOAM

```
// Get the solver factory
auto solver_gen = this->create_dist_solver(
       this->get_exec_handler().get_device_exec(), dist_A_v, dist_x_v,
       dist_b_v, verbose_, dist_A.get_export(), precond);
// Create the actual solver instance
auto solver = solver_gen->generate(dist_A_v);
// Solve the linear system
solver->apply(dist_b_v, dist_x_v);
// Hand the solution back to OpenFOAM
dist_x_v.copy_back();
```

3.5 Efficient stopping criterion implementation

One potentially large overhead, especially for Krylov solvers, can be the evaluation of the stopping criterion in every iteration. In OpenFOAM, the computation of the stopping criterion requires computing a scaled L1 norm, [14]. If this L1 norm calculation relies on OpenFOAM's implementation on the host side, it requires copying the full residual norm vector back to the host for every solver iteration. Thus, a dedicated GPU version of the scaled L1 norm calculation is needed to guarantee equivalent convergence behavior without introducing the overhead of repeated device-host data transfers. Additionally, the cost of evaluating the residual norm should be kept minimal. If one knows that a minimum number of iterations is needed to solve a certain linear problem, evaluating the residual norm before the minimum number of iterations is reached is only relevant for monitoring purposes. Furthermore, the number of times evaluating the residual norm should be kept minimal. This can be achieved by reducing the frequency at which the residual norm is evaluated. This can lead to additional iterations of the linear solver until the stopping criterion is evaluated. However, the overall computation time can benefit if the additional iterations are faster than the residual norm calculation. The optimal frequency at which the residual norms should be evaluated depends on two factors:

1. The time needed for the evaluation of the residual norm;
2. The cost of the solver iterations.

To compute the optimal frequency f with which the residual norm should be calculated, we use a simple cost model based on the number of iterations needed to solve the linear equation in a previous solve step i. e. $n = n^{t-1}$ and the ratio of the computational costs to perform a single iteration of the linear solver and to evaluate the scaled L1 norm.

3.6 Build procedure and usage example

Ginkgo and OGL use CMake as a build file generator. Furthermore, OGL's build procedure can handle the download, configure, build and installation steps of Ginkgo to ensure the correct version is used. If a sufficiently recent CMake version is installed on the target system, the provided CMake presets can be used to configure, build and install OGL. For a release build with Ninja, for a system without GPUs, the following commands can be used

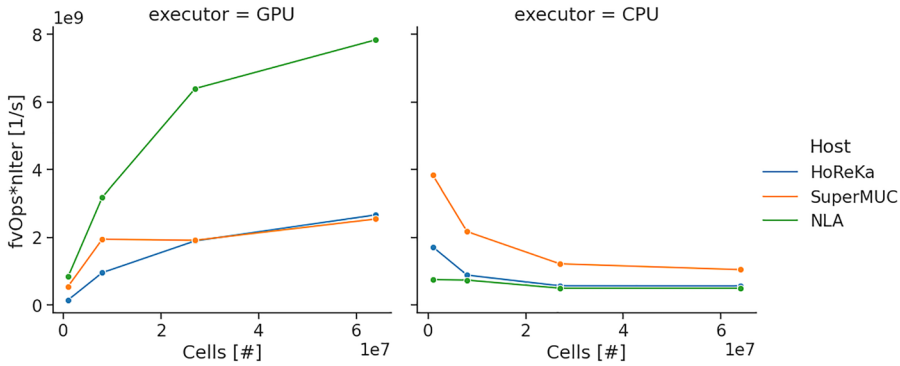

Fig. 2 GPU (left) and CPU (right) linear pressure solver performance over the grid resolution of the three tested HPC machines on a single node.

```
> cmake --preset ninja-cpuonly-release
> cmake --build
        --preset ninja-cpuonly-release
        --target install
```

Besides the `cpuonly` preset other presets are available for CUDA, AMD, and Intel devices. After a successful build several shared objects, e.g. `libOGL.so` and `libGinkgo.so` are copied to the `|FOAM_USER_LIBBIN` directory. To load the shared object at application startup one has to add `libs ("libOGL.so");` to the controlDict file. Then Gingko solver can be selected by adapting the `fvSolution` file. The following shows an example using CG with a block Jacobi preconditioner:

```
p
    {
        solver          GKOCG;
        preconditioner  BJ;
        tolerance       1e-05;
        relTol          0.01;
        executor        cuda;
    }
```

Where the `executor cuda` specifies to run on a NVIDIA device.

4 Results

For a performance evaluation of Ginkgo's GPU solvers within OpenFOAM, the lidDrivenCavity3D[6] was selected. The case was previously used as a benchmark test case by several authors, e.g. [13, 14]. The presented test case uses the original, uniform, cubic grid and evaluates results for various grid resolutions ranging from 100^3 to 500^3 while keeping the CFL number constant. Experiments where conducted using the ESI OpenFOAM version 2306 and OGL version 0.5.3.

Table 3 Maximum achieved performance in fvOps for the GPU accelerated and non-accelerated CPU runs

System	CPU, total	CPU, per Core	GPU	per GPU
NLA	7.49E+08	2.34E+07	7.84E+09	9.80E+08
HoreKa	1.71E+09	2.26E+07	2.67E+09	6.66E+08
SuperMUC	3.83E+09	3.42E+07	2.54E+09	6.36E+08

In the experimental evaluation, we use three hardware systems, referred to as NLA, HoreKa, SuperMUC-NG. The NLA machine is non-public machine composed of two AMD EPYC 7302 16 Core CPUs and eight AMD MI100 GPUs. The second machine we use is the HoreKa cluster[7], where each node hosts two Intel Xeon Platinum 8368 CPUs and four NVIDIA A100-40 GPUs. The third system we use in the experimental evaluation is the SuperMUC-NG Phase 2[8]. SuperMUC-NG features two Intel Sapphire Rapids Xeon Platinum 8480+ CPUs and four Intel Ponte Vecchio Data Center Max 1550 GPUs on each node. Table 2 summarizes the hardware of the tested HPC systems.

[6] https://develop.openfoam.com/committees/hpc.git.

[7] https://www.scc.kit.edu/dienste/HoreKa.php.

[8] https://doku.lrz.de/hardware-of-supermuc-ng-phase-2-222891050.html.

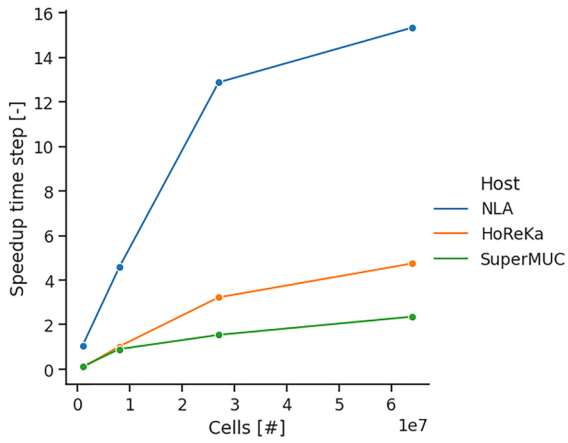

Fig. 3 Resulting speedup of an average timestep for the GPU accelerated cases over the grid resolution for the tested HPC nodes

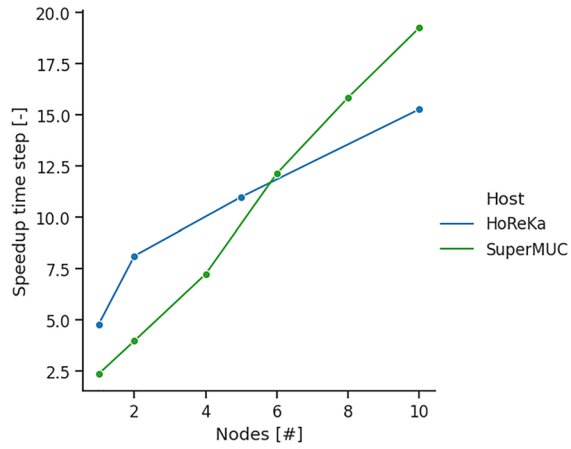

Fig. 4 Strong scaling results of the lidDrivenCavity3D case

The MPI-parallel OpenFOAM runs without GPU acceleration employ as many MPI ranks as physical cores are available on the respective machines, i.e. 32 on the NLA machine, 76 on the HoreKa cluster, and 112 on the SuperMUC-NG cluster. For the OpenFOAM runs using either Ginkgo's CUDA, HIP, or SYCL backend, the computational domain is decomposed into twice as many subdomains as GPUs available. In our experience, this results in a good balance between the computational cost of assembling the system matrix on the CPU and solving the linear systems on the GPUs. The *simple* scheme was taken as a decomposition method.

4.1 Single node performance results

First, the single-node performance was investigated. Here the LidDrivenCavity3D case was run for mesh configurations containing 1, 8, 27, and 64 million cells.

Figure 2 shows the achieved performance of the linear pressure solver on a single node over different mesh resolutions. As a performance metric, fvOps of the linear pressure solver (PCG) are evaluated, with $\text{fvOps} = n_{\text{Cells}} n_{\text{Iter,p}} / t_{\text{solver}}$, where n_{Cells} is the total number of grid cells, $n_{\text{Iter,p}}$ the number of required linear solver iterations till convergence, and $/t_{\text{solver}}$ the required time to solution. Here, a larger number of fvOps indicate better performance. The left figure shows the performance of the GPU-accelerated run and the right figure the performance of the non-accelerated runs on the CPUs. In general, for the GPUs an increase of the performance with increasing problem size can be observed. Across the tested grid sizes, the NLA system which is equipped with eight MI100 GPUs, shows the largest performance compared to the other systems. For smaller grid sizes of up to eight million cells, the Intel Max 1550 shows favorable performance compared to the NVIDIA A100. For the larger cases with more than 8M cells, the Intel Max 1550 and the A100 results are comparable.

The CPU performance decreases with increasing problem size as shown in the right of Fig. 2. This can be explained by a decrease in the cache utilization for large problem sizes. The CPU performance of the SuperMUC system exceeds the CPU performance of the other systems across the tested problem sizes. The maximum achieved performance for each system is summarized in Table 3. It can be seen that the performance, especially when normalized by the number of CPU cores and GPU devices, is comparable. The highest CPU performance per core was measured on the Intel CPUs of the SuperMUC-NG cluster. This is most likely a result of the significantly larger memory bandwidth of the Sapphire Rapids architecture. The highest per GPU performance was measured for the AMD GPUs on the NLA system.

The single node benchmarks are conducted in a range of cells per CPU core that is not necessarily optimal for the CPU caches, thus one might expect even better maximum performance per CPU core when fewer grid cells per CPU core are utilized. This

can be achieved by scaling simulations across multiple compute nodes. However, with the increasing adoption of GPU accelerators, HPC clusters tend to provide fewer compute nodes in total. This could in consequence require running simulations with more grid cells per CPU core than considered optimal today.

Figure 3 shows the achieved speedup by offloading on the tested systems over the grid size. Here, the speedup is computed as $S = t_{\text{GPU}}/t_{\text{CPU,SN}}$, where t_{GPU} is the average time required to solve a time step with GPU acceleration and $t_{\text{CPU, SN}}$ the average time required without GPU acceleration using all available CPU cores of that node. The resulting speedup includes all overhead costs of the offloading procedure, including the time required to update the GPU device matrix and copying data between the host and the accelerator.

The speedup generally increases with increasing problem size. This is a result of the increasing performance of the GPU and the decreasing CPU performance for increasing problem sizes, c.f. Fig. 2. The largest speedup of approximately a factor of 15 was observed on the NLA machine, which is equipped with the largest number of GPU devices and the lowest number of CPU cores. The machines with four GPUs and more CPU cores reach a speedup of approximately four on the HoreKa and two on the SuperMUC-NG machines, respectively.

4.2 Multi node performance results

In the next experiment, we investigate strong scaling behavior on up to 10 compute nodes and 40 GPUs on the HoreKa and the SuperMUC-NG cluster. For this, a case with 125M cells was selected and decomposed into 8 subdomains per node for accelerated simulations. The simulation without acceleration, but utilizing all CPU cores, was taken as a reference to compute the speedup, which is shown in Fig. 4.

For both tested systems, performance increases with the number of compute nodes. However, on the SuperMUC-NG machine a super linear speedup can be observed when using more than 4 nodes. This seems to be a consequence of the favorable GPU performance profile of the SuperMUC-NG machine for smaller grid resolutions, as indicated in Fig. 2.

5 Conclusion and Outlook

A platform-portable offloading approach was presented and tested on three different systems equipped with accelerators from Intel, AMD, and NVIDIA. For all tested cases, a speedup for sufficiently large grid sizes can be observed. For two of the three tested machines, strong scaling studies on up to 40 GPUs were performed, demonstrating good strong scalability. Additionally, methods to reduce the communication overhead by employing GPU device persistent data structures and reducing the frequency of the stopping criterion evaluation were discussed to increase the overall performance. Additionally, to the work presented here, several improvements are currently under investigation. This includes the implementation and evaluation of a distributed Multigrid solver, the impact of GPU based preconditioner, as well as advanced repartitioning schemes to support different decompositions between CPU and GPU.

Acknowledgements This work was supported by the German Federal Ministry of Education and Research (grant number 16ME0676K). This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). Furthermore, the authors acknowledge the organizing committee of the 18th OpenFOAM Workshop for the support provided.

Author contributions All authors contributed to the study's conception and design. Material preparation, data collection, and analysis were performed by Gregor Olenik. Necessary implementations in the investigated software were performed by Gregor Olenik and Marcel Koch. The first draft of the manuscript was written by Gregor Olenik and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript

Funding Open Access funding enabled and organized by Projekt DEAL. This work was supported by the German Federal Ministry of Education and Research (grant number 16ME0676K).

Data availability Not applicable.

Code availability The invesigated code is available at https://github.com/hpsim/OGL.

Declarations

Conflict of interest Not applicable.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

Materials availability Not applicable

References

1. NVIDIA: CUDA C++ programming guide (2023)
2. AMD: ROCm-developer-tools/HIP. ROCm Developer Tools (2023)
3. Khronos: SYCL—C++ single-source heterogeneous programming for acceleration offload (2014)
4. Dagum L, Menon R (1998) Openmp: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
5. Trott CR, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman DS, Ibanez D, Liber N, Madsen J, Miles J, Poliakoff D, Powell A, Rajamanickam S, Simberg M, Sunderland D, Turcksin B, Wilke J (2022) Kokkos 3: programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst 33(4):805–817. https://doi.org/10.1109/TPDS.2021.3097283
6. Edwards HC, Trott CR, Sunderland D (2014) Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. 74(12):3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003Domain-Specific Languages and High-Level Frameworks for High-Performance Computing
7. ...Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, Buschelman K, Constantinescu EM, Dalcin L, Dener A, Eijkhout V, Faibussowitsch J, Gropp WD, Hapla V, Isaac T, Jolivet P, Karpeev D, Kaushik D, Knepley MG, Kong F, Kruger S, May DA, McInnes LC, Mills RT, Mitchell L, Munson T, Roman JE, Rupp K, Sanan P, Sarich J, Smith BF, Zampini S, Zhang H, Zhang H, Zhang J (1999) PETSc Web page. https://petsc.org/ (2024). https://petsc.org/
8. Anzt H, Cojean T, Flegar G, Göbel F, Grützmacher T, Nayak P, Ribizel T, Tsai YM, Quintana-Ortí ES (2020) Ginkgo: a modern linear operator algebra framework for high performance computing. arXiv preprint arXiv:2006.16852
9. Patankar SV, Spalding DB (1972) A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. Int J Heat Mass Transf 15(10):1787–1806. https://doi.org/10.1016/0017-9310(72)90054-3
10. Issa RI (1986) Solution of the implicitly discretised fluid flow equations by operator-splitting. J Comput Phys 62(1):40–65. https://doi.org/10.1016/0021-9991(86)90099-9
11. Weller HG, Tabor G, Jasak H, Fureby C (1998) A tensorial approach to computational continuum mechanics using object-oriented techniques. Comput. Phys. 12(6):620–631. https://doi.org/10.1063/1.168744
12. Combest DP, Day J (2011) Cufflink: a library for linking numerical methods based on CUDA C/C++ with OpenFOAM, 2011
13. Tomczak T, Zadarnowska K, Koza Z, Matyka M, Mirosław Ł (2012) Complete PISO and SIMPLE solvers on graphics processing units. arXiv preprint arXiv:1207.1571
14. Bná S, Spisso I, Olesen M, Rossi G (2020) PETSc-4FOAM: a library to plug-in PETSc into the OpenFOAM framework. PRACE White Paper
15. Zampini S, Bnà S, Valentini M, Spisso I (2020) GPU-accelerated OpenFOAM simulations using PETSc-4FOAM. In: 8th ESI-OpenFOAM conference
16. Piscaglia F, Ghioldi F (2023) GPU acceleration of CFD simulations in OpenFOAM. Aerospace 10(9):792
17. Rathnayake T (2016) Integrating OpenFOAM and GPUs using AMGx. PhD thesis
18. Rathnayake T, Jayasena S, Narayana M (2017) OpenFOAM on GPUs using AMGx. In: Proceedings of the 25th high performance computing symposium, pp 1–12
19. Olenik G, Kashi A, Nayak P, Göbel F, Ribizel T, Cojean T, Tsai Y-H, Koch M, Georgiou V, Anzt H Improving linear solver performance by offloading computations to GPGPUS with GINKGO. In: 17th OpenFOAM workshop
20. Munshi A (2009) The OpenCL specification. In: 2009 IEEE hot chips 21 symposium (HCS). IEEE, pp 1–314
21. Olenik G, Cojean T, Göbel F, Grützmacher T, Kashi A, Nayak P, Ribizel T, Mike Tsai Y, Anzt H (2021) Accelerating OpenFOAM simulations with GPUs using Ginkgo. In: 9th OpenFOAM conference
22. Anzt H, Cojean T, Flegar G, Göbel F, Grützmacher T, Nayak P, Ribizel T, Tsai Y, Quintana-Ortí ES (2022) Ginkgo: a modern linear operator algebra framework for high performance computing. ACM Trans Math Softw 48(1):1–33. https://doi.org/10.1145/3480935
23. Bartlett R, Demeshko I, Gamblin T, Hammond G, Heroux M, Johnson J, Klinvex A, Li X, McInnes LC, Moulton JD et al (2017) xsdk foundations: toward an extreme-scale scientific software development kit. arXiv preprint arXiv:1702.08425
24. Arndt D, Bangerth W, Davydov D, Heister T, Heltai L, Kronbichler M, Maier M, Pelteret J-P, Turcksin B, Wells D (2021) The deal.II finite element library: design,

features, and insights. Comput Math Appl 81:407–422. https://doi.org/10.1016/j.camwa.2020.02.022
25. Anderson R, Andrej J, Barker A, Bramwell J, Camier J-S, Cerveny J, Dobrev V, Dudouit Y, Fisher A, Kolev T, Pazner W, Stowell M, Tomov V, Akkerman I, Dahm J, Medina D, Zampini S (2021) MFEM: a modular finite element methods library. Comput Math Appl 81:42–74. https://doi.org/10.1016/j.camwa.2020.06.009
26. Plank* G, Loewe* A, Neic* A, Augustin C, Huang Y-LC, Gsell M, Karabelas E, Nothstein M, Sánchez J, Prassl A, Seemann* G, Vigmond* E, (2021) The openCARP simulation environment for cardiac electrophysiology. Comput Methods Prog Biomed 208:106223. https://doi.org/10.1016/j.cmpb.2021.106223
27. openCARP consortium, Augustin C, Boyle PM, Loechner V, Colin R, Huppé A, Gsell M, Houillon M, Huang Y-.C, Hustad KG, Karabelas E, Loewe A, Myklebust L, Neic A, Nothstein M, Plank G, Prassl A, Sánchez, J., Seemann, G., Stary, T., Thangamani, A., Tippmann, N., Trevisan Jost, T., Vigmond, E., Wülfers, E.M., Linder, M.: openCARP. https://doi.org/10.35097/1979 . https://git.opencarp.org/openCARP/openCARP