GALÆXI: Solving complex compressible flows with high-order discontinuous Galerkin methods on accelerator-based systems

Marius Kurz[a,b,1], Daniel Kempf[a,1,*], Marcel Blind[a], Patrick Kopper[a], Philipp Offenhäuser[c], Anna Schwarz[a], Spencer Starr[a], Jens Keim[a], Andrea Beck[a]

[a]Institute of Aerodynamics and Gas Dynamics, University of Stuttgart, Pfaffenwaldring 21, 70569 Stuttgart, Germany
[b]Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands
[c]Hewlett Packard Enterprise (HPE), Herrenberger Straße 140, 71034 Böblingen, Germany

Abstract

This work presents GALÆXI as a novel, energy-efficient flow solver for the simulation of compressible flows on unstructured hexahedral meshes leveraging the parallel computing power of modern Graphics Processing Units (GPUs). GALÆXI implements the high-order Discontinuous Galerkin Spectral Element Method (DGSEM) using shock capturing with a finite-volume subcell approach to ensure the stability of the high-order scheme near shocks. This work provides details on the general code design, the parallelization strategy, and the implementation approach for the compute kernels with a focus on the element local mappings between volume and surface data due to the unstructured mesh. The scheme is implemented using a pure distributed memory parallelization based on a domain decomposition, where each GPU handles a distinct region of the computational domain. On each GPU, the computations are assigned to different compute streams which allows to antedate the computation of quantities required for communication while performing local computations from other streams to hide the communication latency. This parallelization strategy allows for maximizing the use of available computational resources. This results in excellent strong scaling properties of GALÆXI up to 1024 GPUs if each GPU is assigned a minimum of one million degrees of freedom. To verify its implementation, a convergence study is performed that recovers the theoretical order of convergence of the implemented numerical schemes. Moreover, the solver is validated using both the incompressible and compressible formulation of the Taylor–Green-Vortex at a Mach number of 0.1 and 1.25, respectively. A mesh convergence study shows that the results converge to the high-fidelity reference solution and that the results match the original CPU implementation. Finally, GALÆXI is applied to a large-scale wall-resolved large eddy simulation of a linear cascade of the NASA Rotor 37. Here, the supersonic region and shocks at the leading edge are captured accurately and robustly by the implemented shock-capturing approach. It is demonstrated that GALÆXI requires less than half of the energy to carry out this simulation in comparison to the reference CPU implementation. This renders GALÆXI as a potent tool for accurate and efficient simulations of compressible flows in the realm of exascale computing and the associated new HPC architectures.

Keywords: Discontinuous Galerkin, High-Performance Computing, GPUs, Accelerators, Turbulence, Compressible Flow

1. Introduction

The computational sciences have become an essential driver for understanding the dynamics of complex, nonlinear systems ranging from the dynamics of earth's climate [1] to obtaining information about a patient's characteristic blood flow to derive personalized approaches in medical therapy [2]. While these successes also rely on significant breakthroughs in the development of numerical methods and physical models, a major portion can be ascribed to the exponential increase in available computing power, which has allowed simulating increasingly large and complex problems over the last decades. However, the corresponding process of shrinking transistors from generation to generation has become increasingly challenging and the resulting gains in performance have diminished in recent years [3]. As a consequence, the community is moving towards accelerator chips, which do not serve as a one-size-fits-all hardware like traditional CPUs. These chips are specialized to yield better performance and efficiency for specific tasks, such as workloads in artificial intelligence, video encoding or cryptography. This also shows in the field of high-performance computing (HPC), where nine out of the ten fastest supercomputers listed in the most recent TOP500 [4] from November 2023 employ some form of accelerator. In the most recent GREEN500 [5] list, which focuses on sustainability in terms

---

*Corresponding author

Email addresses: marius.kurz@iag.uni-stuttgart.de (Marius Kurz), daniel.kempf@iag.uni-stuttgart.de (Daniel Kempf), marcel.blind@iag.uni-stuttgart.de (Marcel Blind), patrick.kopper@iag.uni-stuttgart.de (Patrick Kopper), philipp.offenhaeuser@hpe.com (Philipp Offenhäuser), anna.schwarz@iag.uni-stuttgart.de (Anna Schwarz), spencer.starr@iag.uni-stuttgart.de (Spencer Starr), jens.keim@iag.uni-stuttgart.de (Jens Keim), andrea.beck@iag.uni-stuttgart.de (Andrea Beck)

[1]M. Kurz and D. Kempf contributed equally and cordially agree to share first authorship.

of energy invested per computation, all of the ten most efficient HPC systems employ GPU accelerators.

However, such accelerators generally differ considerably from general-purpose CPUs in terms of hardware design, as well as the working principle. As a consequence, using accelerators oftentimes not only requires rewriting and redesigning large portions of existing code to make efficient use of such hardware, but also might change which numerical algorithms are most efficient for a specific task. This poses significant challenges for legacy HPC codes due to the considerable effort required to migrate the existing codebase to hardware accelerators. This issue is particularly pervasive in the field of Computational Fluid Dynamics (CFD), where scale-resolving simulations of turbulent flows generally require significant HPC resources. Here, modern high-order discretization methods such as Discontinuous Galerkin (DG) and Flux Reconstruction (FR) schemes have become popular due to their computational efficiency for such multi-scale problems and their excellent scaling properties on HPC systems.

Over the last two decades, these advantages have led to the development of a rich landscape of high-order HPC solvers for scale-resolving CFD simulations. Here, spectral element discretizations have seen particular interest and have been implemented for instance in Nektar++ [6] and HORSES3D [7], which can solve both the compressible and incompressible Navier–Stokes equations (NSE), and the DG solver ExaDG [8], which solves the incompressible NSE. While these solvers are written in traditional HPC languages like C++ and Fortran, the solver Trixi.jl [9] implements multiple DG discretizations in Julia to combine the advantages of high computational performance with the flexibility and ease of implementation provided by modern Python-style languages. In addition to DG-based approaches, also other high-order methods have been widely adopted using for instance weighted essentially non-oscillatory (WENO) finite volume (FV) schemes as implemented in UCNS3D [10] or high-order finite difference methods with WENO-type shock capturing utilized in the STREAmS [11, 12] solver.

The need to adapt the existing and established code bases to the new GPU-focused HPC architectures has already been considered in the CFD community. One of the most established high-order codes for incompressible and weakly compressible flow is Nek5000 [13]. The first effort to port Nek5000 to accelerators was reported in 2015 [14], where a barebones version of the code was adapted for GPUs using the OpenACC library. Full GPU support was then offered by its successor NekRS [15], which is based on the Open Concurrent Computing Abstraction (OCCA) [16]. Similarly, Neko [17] was implemented from scratch using modern, object-orientated Fortran and abstraction layers to support multiple hardware backends. While the previous codes focus mainly on incompressible and weakly compressible flows, pyFR [18] also solves the compressible NSE on unstructured meshes using the FR approach. Moreover, it is written in Python and relies on code generation to support multiple computing backends including accelerators and provides excellent scaling properties on HPC systems. Similarly, the deal.II [19] and MFEM [20] libraries provide DG discretizations to solve the compressible NSE and both have added GPU support in recent years.

In this work, we present GALÆXI[2] as a GPU-accelerated solver for hyperbolic-parabolic conservation laws with special emphasis on compressible flows. The numerical simulation of compressible flows is highly relevant for a large number of problems, e.g., from the aviation industry or aeroacoustics. GALÆXI builds on the well-established FLEXI solver [21] and inherits the majority of its features and its extensive pre- and post-processing suite that is designed for large-scale applications. Hence, GALÆXI implements multiple flavors of the Discontinuous Galerkin Spectral Element Method (DGSEM) and can handle fully unstructured, hexahedral, curved, high-order meshes to account for complex geometries. Moreover, multiple stabilization techniques are implemented to ensure the stability of the scheme in underresolved simulations and in the vicinity of shocks using shock capturing schemes based on localized finite-volume (FV) subcell approaches. We consider this focus on compressible, transonic flow through the combination of high-order accuracy and provable stability near shocks together with the efficient implementation of this hybrid discretization on GPU systems as one of the unique features of GALÆXI that sets it apart from existing solvers. The user interface of GALÆXI is deliberately kept compatible to FLEXI, such that GALÆXI can serve as a drop-in replacement to run existing simulation setups on GPU systems without modifications.

This work contributes the following aspects to the challenging but necessary steps for the transition to exascale HPC architectures in CFD. It provides insights into the suitability of DGSEM for GPU acceleration and quantifies the gains in performance and efficiency that can be expected for explicit, high-order DG methods when moving from traditional CPUs to GPUs. It also provides practical guidelines on how existing codebases can be ported to GPU hardware and proposes parallelization concepts for achieving parallel efficiency on HPC hardware with high-order schemes. Furthermore, the savings in the context of energy-to-solution are discussed in particular and can serve as a point of reference in terms of energy efficiency.

This work is organized as follows. First, Section 2 introduces the governing equations and the numerical methods implemented in GALÆXI. Based on this, Section 3 provides details on the parallelization strategy and the implementation of the compute kernels. The resulting performance and scaling abilities of GALÆXI are presented and discussed in Section 4. The implementation of the numerical scheme is verified in Section 5, demonstrating the theoretical convergence rates of the numerical methods and accurate results for the incompressible and compressible formulations of the Taylor–Green-Vortex (TGV) test case. To demonstrate the applicability to applications of relevance, GALÆXI is employed in Section 6 to compute a large-scale, wall-resolved LES of the NASA Rotor 37 [22] test case. Section 7 summarizes the major results of the paper and provides an outlook on further developments.

---

[2]`https://github.com/flexi-framework/galaexi/`

2. Numerical Methods

GALÆXI is implemented as a general solution framework for hyperbolic-parabolic conservation equations, similar to the FLEXI framework, but exhibits a particular focus on the compressible Navier–Stokes equations (NSE), which are introduced in Section 2.1. The DGSEM with its temporal and spatial high-order accurate discretization will be introduced in Section 2.2, followed by the compatible sub-cell shock capturing scheme in Section 2.3.

2.1. Governing Equations

GALÆXI is used to solve the compressible NSE, which describe the evolution of the conserved variables $U(x, t) = (\rho, \rho u, \rho e)^T$ which are comprised of the density, momentum, and energy density, respectively, at each position in space x and time t. The NSE can be derived by enforcing the conservation of mass, momentum, and energy across an infinitesimal control volume. This yields the evolution equations of the conserved variables in differential form as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho u) = 0, \qquad (1)$$

$$\frac{\partial \rho u}{\partial t} + \nabla \cdot (\rho u \otimes u + pI - \tau) = \mathbf{0}, \qquad (2)$$

$$\frac{\partial \rho e}{\partial t} + \nabla \cdot (u (\rho e + p) - \tau \cdot u + q) = 0, \qquad (3)$$

where p denotes the static pressure, I the identity matrix, and $\mathbf{0}$ the zero vector. Assuming a Newtonian fluid and Fourier's law of thermal conduction yields the stress tensor τ and heat flux q as

$$\tau = \mu \left(\nabla u + \nabla u^T - \frac{2}{3} (\nabla \cdot u) I \right), \qquad (4)$$

$$q = -\lambda \nabla T. \qquad (5)$$

Here, μ denotes the dynamic viscosity of the fluid and λ denotes its heat conductivity. Both are material properties of the specific fluid and depend in the general case on the fluid's local state. Hence, both quantities cannot be considered constant in the general case. In this work, we assume the viscosity to follow Sutherland's law [23], which postulates a dependency of the viscosity on the temperature of the form

$$\mu(T) = \mu_{ref} \frac{1.4042 \, (T/T_{ref})^{3/2}}{T/T_{ref} + 0.4042}, \qquad (6)$$

where μ_{ref} is the viscosity at the reference temperature T_{ref}. Based on this, the thermal conductivity can be computed as

$$\lambda = \frac{\gamma R}{\gamma - 1} \frac{\mu}{\text{Pr}}, \qquad (7)$$

with γ as the ratio of specific heats, R denoting the specific gas constant, and Pr as the dimensionless Prandtl number, which is assumed in the following to be constant with Pr = 0.71.
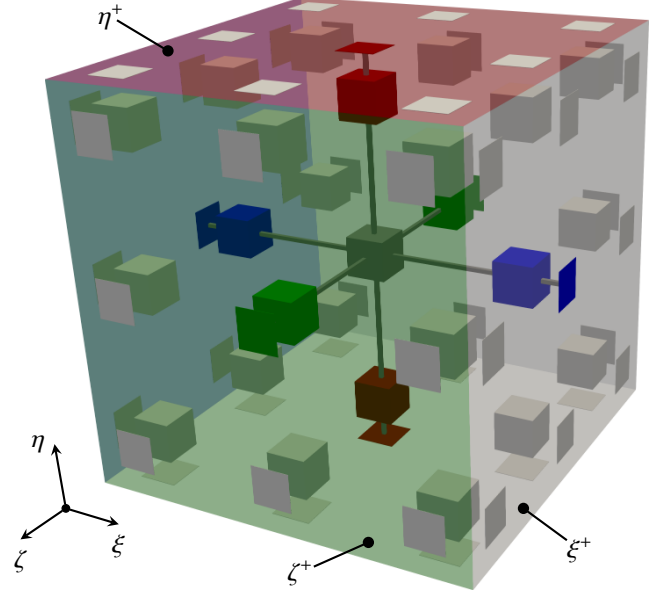

Figure 1: Perspective sketch of a single DG element in the reference space using Legendre-Gauss interpolation points with $N = 2$. Gray cubes indicate the interpolation points within the element, while the gray squares indicate interpolation points on the six local faces called $\xi^\pm, \eta^\pm, \zeta^\pm$. The linewise operations of the tensor product ansatz are indicated for the center interpolation point, where the operations along the coordinates $\boldsymbol{\xi} = (\xi, \eta, \zeta)$ are highlighted in blue, red and green, respectively.

Lastly, the equation-of-state (EOS) closes the NSE by providing a relationship between the conserved variables and the pressure. For a perfect gas, this can be written as

$$p = (\gamma - 1) \left(\rho e - \frac{\rho}{2} \, u \cdot u \right), \quad \text{or} \qquad (8)$$

$$T = \frac{p}{\rho R}. \qquad (9)$$

Equations (8) and (9) thus allow the computation of the primitive, i.e. non-conserved, variables $U^{prim} = (\rho, u, p, T)^T$ from the state U.

2.2. Discontinuous Galerkin Spectral Element Method

In the following section, the DGSEM will be derived for the compressible NSE, which can be written in flux formulation as

$$\frac{\partial U}{\partial t} + \nabla_x \cdot F(U, \nabla_x U) = \mathbf{0}, \qquad (10)$$

where $F(U, \nabla_x U)$ encapsulates both the convective and viscous fluxes. Each of the main construction steps of the DGSEM will be discussed. However, a more in-depth derivation of the DGSEM and its implementation is provided by Krais et al. [21].

Mapping the Equations

For the DGSEM, the domain Ω is subdivided into a set of non-overlapping, curvilinear, hexahedral elements. Each physical element is then mapped from the physical space $x = (x, y, z)^T$ to the reference element $E \in [-1, 1]^3$ in computational space $\boldsymbol{\xi} = (\xi, \eta, \zeta)^T$ using a transfinite polynomial mapping $\boldsymbol{\xi} = \chi(x)$.

3

The reference element is shown for $N = 2$ in Fig. 1. The Jacobian $\mathcal{J}$ of this mapping follows as the determinant of the Jacobian matrix $\nabla_\xi \chi$, where ∇_ξ denotes the del operator in the computational coordinates. The transformation of the governing equations into the computational space requires the contravariant basis vectors $\mathcal{J}a^i$, with $i = 1, 2, 3$, which follow in the curl form as

$$\mathcal{J}a_n^i = -\hat{x}_i \cdot \nabla_\xi \times (x_l \nabla_\xi x_m), \quad (n, m, l) \text{ cyclic}, \tag{11}$$

where $\hat{x}_i$ is the unit vector in the i-th Cartesian direction. Using the basis vectors and the Jacobian, the transformed equations in the reference element follow as

$$\mathcal{J}\frac{\partial U}{\partial t} + \nabla_\xi \cdot \mathcal{F}^i = \mathbf{0}, \tag{12}$$

where $\mathcal{F}^i$ denotes the contravariant fluxes given by

$$\mathcal{F}^i = \mathcal{J}a^i \cdot F. \tag{13}$$

To construct the DGSEM, Eq. (12) is formulated in the weak form, which will be derived in the next paragraph.

Weak formulation

To derive its weak form, Eq. (12) is projected onto a set of test functions $\psi(\xi)$, spanning a polynomial subspace, using the inner product, which yields

$$\int_E \mathcal{J}\frac{\partial U}{\partial t}\psi \, d\xi + \underbrace{\oint_{\partial E} \psi \, (\mathcal{F} \cdot \mathcal{N})^* \, dS}_{\text{Surface Integral}} - \underbrace{\int_E \mathcal{F} \cdot \nabla_\xi \psi \, d\xi}_{\text{Volume Integral}} = \mathbf{0}. \tag{14}$$

Here, the surface integral incorporates the contribution of the fluxes across the element faces while the volume integral considers only the degrees of freedom within the element. Because adjacent elements share a common face and no continuity across elements has been imposed, the solution is generally discontinuous across the element faces. Consequently, the solution and hence the fluxes on the element faces are non-unique. Therefore, numerical flux functions are used to compute a unique numerical flux across element boundaries, which is denoted by the asterisk $(\cdot)^*$.

Solution Representation

Within each element, the solution is represented by high-order Lagrange polynomials with any desired order resulting in arbitrary high-order accuracy of the DGSEM as demonstrated in Section 5.1. The j-th one-dimensional Lagrange polynomial of degree N is defined as

$$\ell_i^N(x) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{x_i - x}{x_i - x_j}, \tag{15}$$

with respect to a set of interpolation points $\{x_j\}_{j=0}^N$. In practice, either Legendre–Gauss (GL) or Legendre–Gauss–Lobatto (LGL)

nodes are used as interpolation points. The superscript is subsequently dropped to keep the notation concise. Lagrange polynomials fulfill the Kronecker delta property given by

$$\ell_i(x_j) = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases} \tag{16}$$

A tensor-product ansatz is used to construct a three-dimensional basis of the polynomial subspace $\mathbb{P}^N$ from the one-dimensional Lagrange polynomials. This yields the approximation of the solution in the computational space as

$$U(\boldsymbol{\xi}, t) \approx \sum_{i,j,k=0}^{N} \hat{U}_{ijk}(t)\, \ell_i(\xi)\, \ell_j(\eta)\, \ell_k(\zeta). \tag{17}$$

Semi-discrete form

Evaluating the integrals using the Gauss-type quadrature associated with the chosen set of interpolation points, i.e. collocation of interpolation and integration points, yields the semi-discrete form of the DG operator that can be written for each point $i, j, k \in [0, N]$ as

$$\frac{\partial \hat{U}_{ijk}}{\partial t} = -\overbrace{\frac{1}{\mathcal{J}_{ijk}}}^{\text{ApplyJac}} \Bigg[\sum_{\alpha=0}^{N} \mathcal{F}_{\alpha jk}^1 \hat{D}_{i\alpha} + \Big(\overbrace{[f^*\hat{s}]_{jk}^{\xi^+}}^{\text{FillFlux}} \hat{\ell}_i^+ + \overbrace{[f^*\hat{s}]_{jk}^{\xi^-}}^{\text{FillFlux}} \hat{\ell}_i^- \Big)$$
$$+ \sum_{\beta=0}^{N} \mathcal{F}_{i\beta k}^2 \hat{D}_{j\beta} + \Big([f^*\hat{s}]_{ik}^{\eta^+} \hat{\ell}_j^+ + [f^*\hat{s}]_{ik}^{\eta^-} \hat{\ell}_j^- \Big)$$
$$+ \underbrace{\sum_{\gamma=0}^{N} \mathcal{F}_{ij\gamma}^3 \hat{D}_{k\gamma}}_{\text{VolInt}} + \underbrace{\Big([f^*\hat{s}]_{ij}^{\zeta^+} \hat{\ell}_k^+ + [f^*\hat{s}]_{ij}^{\zeta^-} \hat{\ell}_k^- \Big)}_{\text{SurfInt}} \Bigg]. \tag{18}$$

This notation follows Kopriva [24], where

$$\hat{\ell}_i^{\pm} = \frac{\ell_i(\pm 1)}{\omega_i} \quad \text{and} \quad \hat{D}_{ij} = -\frac{\omega_i}{\omega_j} \left.\frac{d\ell_i(\xi)}{d\xi}\right|_{\xi=\xi_j} \tag{19}$$

are one-dimensional building blocks that entail the numerical quadrature weights ω_i and are precomputed during initialization to improve the overall performance of the implementation. Moreover, $f^* = f^*(\tilde{U}^L, \tilde{U}^R)$ denotes the unique flux at the faces based on the solution on the surface of the left and right element, respectively, and $\hat{s}$ denotes the surface element, which is the norm of the non-normalized physical unit vector as discussed in more detail by Krais et al. [21]. The monospaced namings in Eq. (18) refer to the routines in the numerical implementation which are summarized in Table 1 and are also detailed in Appendix A.

At this point, we would like to briefly discuss the influence of the unstructured mesh topology. First of all, the unstructured neighbor relations only influence the surface-related operations and only direct neighbors are considered. Here, the relative orientation between the adjacent elements and their faces must be taken into account. This is taken into account by corresponding mappings in the PROLONGTOFACE and SURFINT routines.

4

Routine	Vol/Surf	DOF-local	Lift_*	Operations	Explanation
ConsToPrim	Surf, Vol	YES	NO	$O(N^{2,3})$	Computes primitive variables U^{prim} from state U.
VolInt	Vol	NO	YES	$O(N^4)$	Evaluates volume fluxes $\mathcal{F}$ and multiplies with $\hat{D}$.
ProlongToFace	Vol→Surf	NO	YES	$O(N^2)$	Evaluates solution at element faces $U^{L/R}$ to compute f^*.
FillFlux	Surf	YES	YES	$O(N^2)$	Computes common flux f^* on faces with Riemann solver.
SurfInt	Vol←Surf	NO	YES	$O(N^2)$	Computes surface integral with f^* and $\hat{\ell}^{\pm}$.
ApplyJac	Vol	YES	YES	$O(N^3)$	Applies Jacobian $\mathcal{J}$ to $\hat{U}_t$.

Table 1: Individual operations required to evaluate the three-dimensional DG operator with details on whether the routine acts on volume data or surface data. Moreover, it is indicated whether performed operations are DOF-local, i.e. are performed independently for each specific DOF, if they have to be re-applied during the computation of the gradients, which is indicated by the prefix Lift_*, and their computational complexity in terms of N. A more detailed discussion of these routines can be found in Appendix A.

Time integration

The semi-discrete form Eq. (18) is integrated in time using an appropriate integration scheme. GALÆXI offers a variety of different explicit high-order Runge–Kutta-type schemes in a low-storage formulation to reduce the memory consumption. In the following, a fourth-order Runge–Kutta scheme with 5 stages [25] is used for the validation and verification results in Section 5, while a scheme with 14 stages [26] is used for the large-scale application in Section 6. The latter scheme is chosen since it exhibits an optimized stability region for convection-dominated problems and allows for larger time steps.

Nonlinear Stability

The semi-discrete form of the DG discretization in Eq. (18) is derived by means of numerical quadrature rules. However, since the integrands, i.e. the fluxes of the compressible NSE, are non-polynomial, they cannot be integrated exactly by the applied quadrature rules. The resulting integration errors manifest as aliasing that can cause simulations to crash, especially in the underresolved regime. For DG, multiple mitigation strategies have been devised ranging from overintegration [27, 28], also referred to as polynomial dealiasing, to filtering procedures that strive to counteract the accumulation of energy in the highest solution modes [29, 30]. In this work, we rely on the split-flux formulation introduced by Gassner et al. [31] to construct a non-linear stable DG scheme. This approach is based on the strong formulation of the governing equations, which can be obtained through a second integration-by-parts of Eq. (14). The discretized form can be cast into the same algorithmic form as Eq. (18) with only minor modifications in the formulation of the fluxes [21]. Here, the fluxes of the NSE are replaced by split-form two-point fluxes that are equivalent on an analytical level but can be used to enforce additional constraints such as entropy consistency in the discretized formulation. In this work, a kinetic-energy-preserving split-flux formulation proposed by Pirozzoli [32] is applied. It is important to stress that the evaluation of two-point fluxes increases the computational cost considerably.

Second-Order Equations

For the NSE, the gradients of the primitive variables $\nabla_x U^{prim}$ are required to evaluate the viscous fluxes. While several approaches exist in the literature, GALÆXI follows the *BR1* method

by Bassi and Rebay [33]. Here, so-called *lifted* gradients g are introduced that should fulfill

$$g - \frac{1}{\mathcal{J}} \nabla_x U^{prim} = 0. \tag{20}$$

This equation is then solved for g by deriving the weak form of Eq. (20) and applying the DGSEM as is done for the NSE themselves. This yields an additional set of equations that is structurally similar to Eq. (18) but using the lifting fluxes instead of the fluxes of the NSE as detailed in Krais et al. [21]. Hence, the computation of the gradients corresponds to increasing the set of unknowns by an additional $(n_{dim} \times n_{lift})$ variables, where n_{dim} corresponds to the number of spatial dimensions and n_{lift} to the number of primitive variables for which the gradients should be computed. This also means that each of the operations indicated in Eq. (18), i.e. ApplyJac, SurfInt, VolInt, FillFlux, has to be executed again for the lifting procedure in each spatial direction.

Computational Complexity

Table 1 also provides the estimated number of operations, i.e. the computational complexity, for the different steps of the three-dimensional DG discretization. The given numbers describe the asymptotic behavior of the individual operations without considering details such as the computational complexity of the flux computation and compiler optimizations. Most importantly, the computational effort to compute the volume integral scales one order higher in terms of N than all other operations, i.e. it scales with $O(N^4)$ instead of $O(N^3)$ or even $O(N^2)$. Hence, the volume integral becomes the dominant operation for increasing N. However, the computations carried out for the volume integral are purely element local, highly dense and can be computed very efficiently on various types of hardware. In practice, this increase in efficiency was observed to partly compensate for the additional operations required with increasing N [34]. Moreover, the communication stencil between elements is small, since only surface fluxes with direct neighbor elements have to be exchanged. Consequently, the high cost of the volume integral and the small communication stencil allow for hiding the communication latencies very efficiently in parallel computations.

2.3. Shock Capturing

GALÆXI is designed for the simulation of compressible flows which can entail discontinuities in the form of shocks. However, the application of high-order discretizations near discontinuities or strong gradients in the solution produces spurious oscillations and can cause the numerical scheme to become unstable. As a consequence, a wide variety of different shock identification and capturing methods are proposed in the literature that all strive to stabilize high-order discretizations near shocks and provide stable and accurate simulations of compressible flows. The common objective of those methods is to retain the high-order accuracy of the baseline scheme in smooth regions while identifying and handling so-called troubled cells within the domain during the simulation. Already Cockburn and Shu [35] used limiters to stabilize DG methods near shocks, which was in parts inspired by the total variation diminishing/bounded properties of FV schemes. Based on this, a myriad of different limiting techniques have been proposed in the literature [36, 37, 38]. Other common approaches introduce some form of artificial viscosity near the shock region [39, 40, 41] or use high-order filtering techniques [42]. Another approach is to employ a hybrid discretization, where the high-order DG scheme is stabilized in the vicinity of the troubled region with a low-order FV scheme. For this, the DG element is subdivided into multiple FV subcells as indicated in Fig. 2. This low-order scheme can then either be solved directly within the troubled elements and coupled to the surrounding DG elements using the common Riemann fluxes [43] or can also be used as a regularizing limiter [44]. In the following, we employ the blending approach by Hennemann et al. [45], who proposed to compute a convex blending of both discretization operators. This approach has also been demonstrated to yield a sensible turbulence model if tuned correctly [46]. Within each element both the high-order DG operator $\mathcal{R}^{\mathrm{DG}}(\hat{U})$ and the compatible low-order FV scheme $\mathcal{R}^{\mathrm{FV}}(\hat{U})$ are evaluated. The convex blending of both schemes then yields

$$\frac{\partial \hat{U}}{\partial t} = (1 - \alpha)\,\mathcal{R}^{\mathrm{DG}}(\hat{U}) + \alpha\,\mathcal{R}^{\mathrm{FV}}(\hat{U}), \tag{21}$$

where the blending factor $\alpha \in [0, 1]$ can be computed either via an *a priori* or *a posteriori* strategy for each individual DG element [47]. In this work, the *a priori* indicator by Hennemann et al. [45] is used, for which the blending approach becomes an operator local to each individual element. Clearly, the standard DG scheme can be recovered for $\alpha = 0$, while $\alpha = 1$ yields a pure FV discretization. It is important to stress that only the contributions of the operators within the element have to be blended since the outer surface fluxes are identical for the DG and FV formulation.

3. Parallelization Strategy on Accelerators

GALÆXI is the endeavor to extend the flow solver FLEXI [21] towards accelerator-based HPC systems. Here, GALÆXI follows three distinct design principles. First, the general data structure and parallelization strategy of FLEXI for unstructured
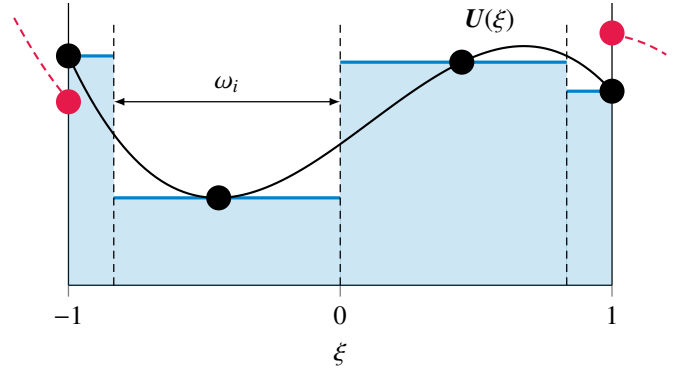

Figure 2: Sketch of the sub-cell shock capturing scheme. The DG polynomial using LGL points and a polynomial degree of $N = 3$ is shown in black with the interpolation points indicated as dots and the integral mean solution within the subcells is shown in blue. The solution in the neighboring DG elements is indicated in red.

geometries should be retained. Second, we strive to retain the majority of the codebase and the associated features of the original implementation. Lastly, GALÆXI is designed such that all routines called during the time-stepping are executed on the accelerator without the need to transfer data to and from the CPU. Device code and compute kernels are only required for routines that are called during time-stepping and thus have to be computed on the accelerator. In contrast, initialization and non-frequently performed analyzing routines are still computed on the CPU, since they are less time-critical and CPUs are better suited towards unstructured workloads. Both GALÆXI and FLEXI are implemented in modern Fortran 2008. The device code for the accelerators in GALÆXI is currently implemented using CUDA Fortran, but the integration of other compute backends is under development.

The design and implementation of GALÆXI is detailed in the following sections using a hierarchical top-down approach. First, the high-level distribution of work across different compute devices and the employed communication scheme between them is detailed in Section 3.1. Based on this, Section 3.2 provides details on how communication and compute kernels are arranged and scheduled on a single GPU. Lastly, the general implementation paradigms for the individual compute kernels are detailed in Section 3.3. Obviously, performance optimizations have to be performed across all of these three levels and changes on one level affect the suitability and performance of the others. While these individual levels are inherently interlinked, we chose this partitioning in the following to provide a more structured overview of the design principles and methods applied in GALÆXI.

3.1. Inter-GPU Parallelization

The parallelization strategy between GPUs in GALÆXI is largely inherited from FLEXI, which employs a pure distributed memory approach using MPI. Before going into the specific implementation details of GALÆXI, the original MPI parallelization strategy of FLEXI is briefly presented. Here, each computational rank is assigned a subdomain of roughly the same
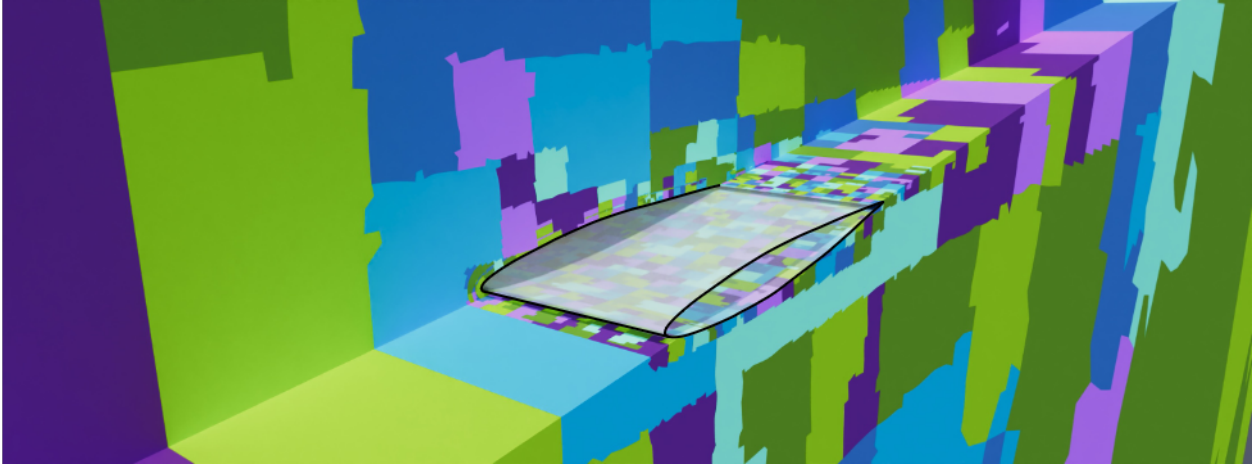
Figure 3: Domain decomposition for a generic airfoil simulation with large spanwise extension. The domain is cut such that the airfoil (transparent surface) including the boundary layer part is visible. Patches of different colors represent individual MPI domains that are processed by different ranks. This figure is an example of a fine granularity, e.g. in the CPU case. In the GPU case, larger MPI domains occur.

amount of elements as shown in Fig. 3. In the DG context, elements are only coupled via their surface fluxes. Thus, only the surface information across the MPI borders has to be exchanged between individual MPI ranks during the computation. Moreover, FLEXI sorts this face information such that the data exchanged between two MPI partners is contiguous in memory and that the sorting is known a priori on both faces. This allows to exchange solely the data itself without any additional sorting information. The overall communication effort is thus proportional to the number of faces at the MPI boundaries, which are referred to in the following simply as *MPI faces*. To minimize the amount of communication, i.e. the number of MPI faces in the domain, FLEXI distributes the domain along a pre-computed space-filling curve. This ensures that the resulting subdomains remain reasonably compact for any amount of subdomains while minimizing partitioning effort. During the simulation, communication is generally asynchronous and non-blocking, which means that communicated data is computed and sent at the earliest possible opportunity. The communication barrier that checks whether the data has been received is positioned at the latest possible instant before the data is actually required for further computations. This allows to effectively hide the communication latency by performing local operations during the data exchange. For this, operations on MPI faces are prioritized over inner faces to use all operations performed on inner faces for latency hiding.

GALÆXI follows the same general approach for parallelizing across multiple GPUs. Here, each GPU on a node is associated with a distinct CPU core while respecting the memory topology to optimize performance.[3] Moreover, CUDA-aware[4] implementations of MPI are used, which improve the performance of MPI communication. By providing Remote Direct Memory Access (RDMA) and host offloading, these allow di-

rect access to the local memory of different GPUs on the same node and transmission of MPI messages directly from the GPU to the network adapter without the assistance of the CPU or the main memory. The following key differences emerge between the CPU and GPU implementations. First, the domain size on a single GPU is larger than for the CPU case. This is a result of the much higher computational power a GPU provides compared to a single CPU core. A GPU requires significantly more workload to run at capacity and to exploit the full degree of its parallelism. In practice, this means that the computational domain per rank increases if GPUs are used. Since the subdomains are compact, an increase in size means that the volume increases much faster than the MPI surface, i.e. that inner work becomes more dominant in comparison to the required communication and at the same time the amount of data to be communicated decreases. In consequence, the performance of the interconnect becomes less dominant than in the CPU case. Second, the GPU implementation has to consider the asynchronicity between the GPU device and the host. While an operation is launched by the host at a specific position in the code sequence, the GPU schedules and executes the operation independently of the work performed by the host in the meantime. Hence, additional synchronization between the host and the device is necessary to ensure data consistency. This entails for instance ensuring that a buffer that is about to be sent via MPI has already been filled with the required information by the GPU. This introduces additional overhead. However, due to the asynchronous operation, CPU and device operations can again be overlapped, which results in an additional level of parallelism on an intra-GPU level and is addressed in Section 3.2.

3.2. Intra-GPU Parallelization

As already discussed in the previous paragraphs, device kernels are launched within host code. However, the GPU schedules and executes the launched kernels asynchronously and can also execute multiple kernels concurrently to maximize its utilization. It is important to consider these properties to maxi-

---

[3]This means, for instance, to associate the CPU core and the GPU such that both reside within the same non-uniform memory access (NUMA) domain.

[4]For further information about CUDA-aware MPI see: `https://docs.open-mpi.org/en/v5.0.x/tuning-apps/networking/cuda.html`
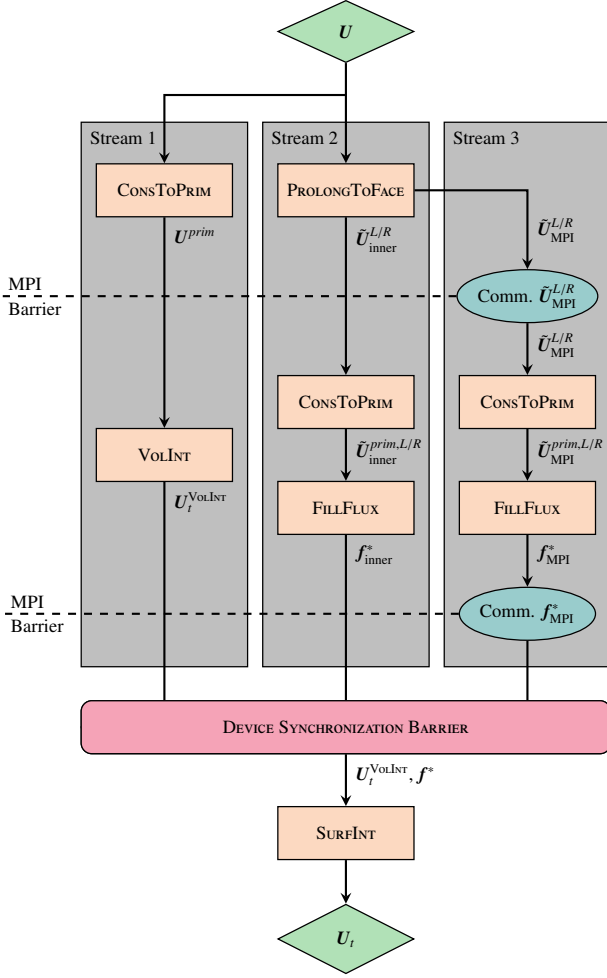
Figure 4: Flowchart of GALÆXI for a single evaluation of the convective DG operator using streams. Some routines comprise several individual compute kernels instead of single, monolithic device kernels. These are summarized here to keep the flowchart concise. Moreover, the lifting procedure to compute the gradients is omitted here for readability.

mize the achieved performance on the device. GALÆXI relies on so-called *streams* to manage the concurrency and scheduling of operations on the GPU. Within the GPU context, streams are similar to execution pipelines. Kernels within each stream are executed serially, i.e. the next kernel within a stream pipeline is only executed once all preceding kernels within this distinct pipeline have finished execution. However, kernels from different streams can run concurrently on the GPU to maximize utilization. This can improve the overall performance either when running small kernels that cannot fully utilize the GPU or by hiding the overhead associated with starting a kernel on the device. Another benefit is that streams allow the mitigation of the *tail effect*, which describes the negative performance impact of the last partial wave of computations in a kernel. This effect stems from the last thread blocks of a kernel call which will generally not fill the whole GPU, leading to a significant portion of the GPU idling when the last wave of computations are performed. By using streams, the idling resources can execute kernels from different streams that are known to be independent of the current computation, which improves GPU utiliza-

tion and thus the overall performance.

A key factor when using streams is to ensure correct results independent of how the individual kernels are scheduled. Therefore, another level of synchronization between the streams is required to mitigate race conditions. In GALÆXI, the different operations of the convective DG operator, summarized in Table 1, are assigned to individual streams depending on their interdependence. This means that if one kernel requires a previous kernel to be completed, both are assigned to the same stream to be executed sequentially. In contrast, operations that are independent of each other get assigned to different streams. In GALÆXI, three streams are employed to account for the available concurrency:

- **Stream 1** (priority low): Operations within DG elements.

- **Stream 2** (priority mid): Operations on inner faces.

- **Stream 3** (priority top): Operations on MPI faces.

Here, each stream is assigned a priority which incentivizes the GPU to preempt and postpone the execution of low-priority kernels in favor of high-priority ones. In GALÆXI, Stream 3 containing the MPI faces is assigned the highest priority to ensure that data that has to be communicated is always computed at the earliest possible instant to ensure optimal latency hiding. The flowchart for the evaluation of the DG operator using these streams is shown in Fig. 4. The local operations within the DG element and the operations at the element faces can be performed independently in their streams until the computation of SurfInt, where the surface fluxes f^* as well as the contributions of the VolInt denoted U_t^{VolInt} are required. Hence, an explicit synchronization barrier is employed to wait until all previous operations in all streams have completed. Then, the surface contributions can be added to the volume integral to yield the final U_t.

Special care must be taken when extending the operator towards multiple GPUs, which requires MPI communication. Here, it has to be ensured that all kernels *within* the GPU are assigned correctly to individual streams and that the MPI communication *between* GPUs is effectively hidden by the local work. For this, the work associated with MPI faces, i.e. Stream 3, is ensured to be computed with the highest priority, such that the communication can be initialized as soon as possible. The work queued in the other streams is then used to hide both the local overhead of tail effects on the GPU and the latency of the MPI communication. In practical application, the host idles at the MPI barrier until the communication is finished, but the GPU is kept busy with the work from Stream 1 and Stream 2 to retain the overall efficiency. Effectively, the communication of the solution at the MPI faces $\tilde{U}_{\text{MPI}}^{L/R}$ is hidden by ConsToPrim on Stream 1. The communication of the resulting fluxes across the MPI faces f_{MPI}^* is hidden by VolInt in Stream 1 and ConsToPrim and FillFlux in Stream 2.

3.3. Kernel Implementation

The goal for the implementation of the compute kernels is to maximize the utilization of the available parallel resources
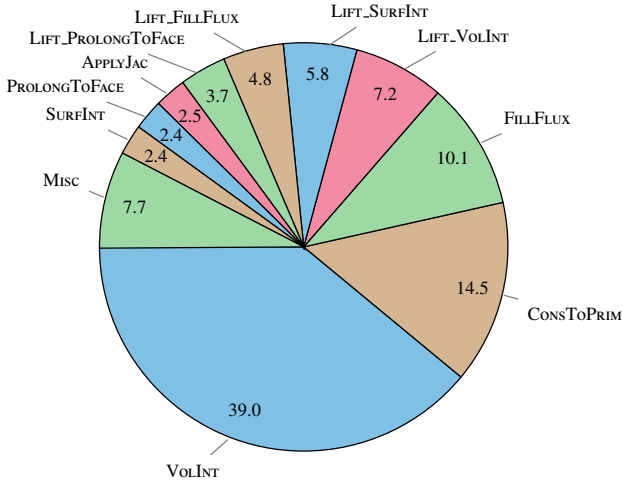
Figure 5: Portion of compute time in percent for individual routines on HAWK-AI with $N = 7$, split-form DG and 8.9×10^5 DOF on a single GPU. Routines associated with the computation of the gradients via the lifting method are prefixed with "Lift_". Various small routines associated with performing the actual time integration, i.e. updating U, are summarized under Misc.

provided by the device which roughly translates to keeping as many threads as possible busy. However, oftentimes the number of concurrent threads is limited by the number of registers required by each thread and the amount of shared memory. Futhermore, the effective performance is limited by the available memory bandwidth, which might not be sufficient to keep all threads busy.

Since the specifics of these limitations and their importance depend heavily on the specific hardware, GALÆXI approaches the problem from a more general perspective. Here, it is assumed that addressing these GPU-specific limitations in general and improving the overall performance for some generic GPU architecture also yields sensible improvements across all specific GPU models. While this approach may not achieve the optimum performance for each specific type of hardware, it has demonstrated to yield a suitable baseline for more in-depth and hardware-specific optimization.

Device code is typically based on a *kernel*, which is the code each individual thread executes. The overall number of threads and their grouping are specified in the *launch configuration*. In some sense, the launch configuration entails an implicit tightly nested loop, while the loop body, i.e. the actual computation, is implemented in the kernel. The optimal launch configuration is oftentimes highly hardware-specific and can improve (or impair) the overall performance significantly. Along the same lines as discussed above, our code relies on sensible initial guesses for all of these kernels, which gave reasonable results. Further improvements are planned through the application of more sophisticated tuning approaches, for instance the *kernel tuner* toolkit [48], which allows automatized optimization of the launch configuration for specific hardware.

The complete list of operations of the DG operator is detailed in Table 1 and the computing time of the kernels associated with these operations is summarized in Fig. 5. Naturally, operations that are DOF-local are the easiest to imple-

---

Algorithm 1 Wrapper for ConsToPrim_Point on CPU

1: **function** ConsToPrim_CPU(N, n_{Elems}, U)
2:    **for** $n \leftarrow 1$ to n_{Elems} **do**   ▷ loop over elements
3:       **for** $i, j, k \leftarrow 0$ to N **do**   ▷ loop within element
4:         $U^{prim}_{ijk,n} \leftarrow$ ConsToPrim_Point($U_{ijk,n}$)
5:       **end for**
6:    **end for**
7:    **return** U^{prim}
8: **end function**

Algorithm 2 Wrapper for ConsToPrim_Point on GPU

1: **function** ConsToPrim_GPU(N, n_{Elems}, U)
2:    $n_{DOF} \leftarrow (N + 1)^3 n_{Elems}$ ▷ number of DOF in array
3:    $U^{prim} \leftarrow$ ConsToPrim_Kernel<<config>>(n_{DOF}, U)
4:    **return** U^{prim}
5: **end function**
6:
7: **kernel** ConsToPrim_Kernel (n_{DOF}, U)
8:    $i \leftarrow$(blockID-1)*blockDim+threadID   ▷ own index
9:    **if** $i \leq n_{DOF}$ **then**
10:      $U^{Prim}_i \leftarrow$ ConsToPrim_Point(U_i)
11:      **return** U^{prim}_i
12:    **end if**
13: **end kernel**

---

ment for different hardware. Hence, the following paragraph first introduces how kernels are designed for DOF-wise operations in GALÆXI before moving to the much more intricate task of kernels that map data between the volume and faces of DG elements.

Pointwise operations

For pointwise operations, a large number of identical computations have to be performed with no interdependence between individual DOFs. Such a computation becomes embarrassingly parallel and straightforward to distribute. The following paragraph details how such computations are implemented in GALÆXI using the ConsToPrim operation as an example. This operation computes the primitive variables $U^{prim} = (\rho, \boldsymbol{u}, p, T)^T$ based on the vector of conservative variables U using the EOS defined in Eqs. (8) and (9). For this, an elemental ConsToPrim_Point routine is implemented that performs the computation for a single DOF. This elemental routine is the building block of the main computation and is agnostic to the underlying hardware. GALÆXI then uses different wrappers for this elemental function. These wrappers distribute the overall work depending on the specific type of computational hardware used. If CPUs are used, the design of the wrapper becomes straightforward as shown in Algorithm 1. A single CPU core just calls the ConsToPrim_Point routine for each DOF within each element of its domain using a tightly nested loop. The GPU wrapper shown in Algorithm 2 is based on the CUDA programming model and consists of two individual components. First, the *kernel* that implements the actual compute operation of an individual GPU thread. The second component is a function that

9

calls the kernel and provides the *launch configuration* config. The launch configuration determines how many threads will be started to execute the kernel and how the individual threads are grouped into thread blocks. In this specific case, each thread of the GPU performs the computation for a single DOF in the domain. For this, each thread determines in line 8 of Algorithm 2 its own globally unique thread ID i. This thread ID incorporates the ID of the current block (blockID), the size of each block (blockDim), and its thread number within the block (threadID), which are all available for each thread during runtime. The thread then performs the computation for this i-th DOF. Note that the high-dimensional structure of the array becomes irrelevant in this case and can be "flattened" to a one-dimensional array containing n_{DOF} entries.

More advanced techniques can be used to optimize those wrappers for different hardware. This includes for instance vectorization, such that either the vector units of a CPU or real vector accelerators can perform the operations performed in ConsToPrim on several entries of U simultaneously. Similarly, optimization such as loop unrolling or shared memory parallelization are straightforward to implement. For GPU usage, the wrapper can be adapted to distribute multiple DOFs to each thread and optimize the launch configuration, depending on the hardware specifics. The same building block approach can also be applied to support other backends such as HIP, ROCm, OpenMP or OpenACC while only maintaining a single version of the equation-specific code.

Volume↔ Surface Operations

The optimization potential of the pointwise operations discussed above is mostly independent of the core algorithms themselves. In contrast, the most challenging routines for GPU porting and parallelization in the DG context are routines that map data between the faces and the volume. Due to the highly local nature of the DG method, the transfer of data from within the element to its faces and vice versa is required in only a few operations. Thus, the original FLEXI code opted to store the data on the element faces and within the elements in different arrays from which it is retrieved based on precomputed mappings. However, revisiting Table 1 reveals that two specific operations in the DG operator access both volume and surface data: ProlongToFace and SurfInt.[5] The former evaluates the polynomial solution from the interior points at the element faces and stores it in a face-based array ($U \rightarrow \tilde{U}^{L/R}$), while the latter computes the integral of the fluxes on the element faces and adds their contribution to the volume ($f^* \rightarrow U_t$). In both cases, an interpolation point in the volume is linked to several points on the surface and vice versa, as shown in Fig. 1. Special care must be taken to exploit the full potential for parallelization of the task on a GPU while avoiding race conditions and costly synchronizations among individual threads. In the following, this is illustrated for the SurfInt routine.

---

[5]As shown in Table 1, the volume integral is also not a point-local operation due to the application of the differentiation matrix along the lines indicated in Fig. 1. However, the operations are retained to the interpolation points within the volume of the DG element, i.e. no exchange of information between the volume and the faces is required.

Algorithm 3 CPU implementation of the SurfInt operation

1: **function** SurfInt($f^*, U_t, \hat{\ell}^+, \hat{\ell}^-$)
2:     **for** $s \leftarrow 1$ to n_{Faces} **do**
3:         **if** isPrimary **then**
4:             $f^{*,\text{tmp}}$, locFace $\leftarrow$ FaceMapping(s, isPrimary, f^*_s)
5:             $U_t \leftarrow$ DoSurfInt(locFace, U_t, $f^{*,\text{tmp}}$, $\hat{\ell}^+$, $\hat{\ell}^-$)
6:         **end if**
7:         **if** isReplica **then**
8:             $f^{*,\text{tmp}}$, locFace $\leftarrow$ FaceMapping(s, isPrimary, $-f^*_s$)
9:             $U_t \leftarrow$ DoSurfInt(locFace, U_t, $f^{*,\text{tmp}}$, $\hat{\ell}^+$, $\hat{\ell}^-$)
10:         **end if**
11:     **end for**
12:     **return** U_t
13: **end function**
14:
15: **function** DoSurfInt(locFace, U_t, f^*_{pq}, $\hat{\ell}^+$, $\hat{\ell}^-$)
16:     **switch** (locFace)
17:         **case** ξ^-
18:             **for** $i, j, k \leftarrow 0$ to N **do**
19:                 $U_{t,ijk} \leftarrow U_{t,ijk} + f^*_{jk}\, \hat{\ell}^-_i$
20:             **end for**
21:         **case** ...
22:         **case** ζ^+
23:             **for** $i, j, k \leftarrow 0$ to N **do**
24:                 $U_{t,ijk} \leftarrow U_{t,ijk} + f^*_{ij}\, \hat{\ell}^+_k$
25:             **end for**
26: **end function**

Algorithm 4 GPU kernel for the SurfInt operation

1: **kernel** SurfInt_Kernel (N, n_{Elems}, f^*, U_t)
2:     $i \leftarrow$ (blockID-1)*blockDim+threadID
3:     $n_{DOF} \leftarrow (N + 1)^3 n_{Elems}$ ▷ number of volume DOF
4:     **if** $i \leq n_{DOF}$ **then**
5:         **for** locFace $\in \{\xi^-, \eta^-, \zeta^-, \xi^+, \eta^+, \zeta^+\}$ **do**
6:             $p, q, s, \hat{\ell}^\pm_k$, isPrimary $\leftarrow$ FaceMapping(i, locFace)
7:             **if** isPrimary **then**
8:                 $U_{t,i} = U_{t,i} + f^*_{pq,s}\, \hat{\ell}^\pm_k$
9:             **else**
10:                 $U_{t,i} = U_{t,i} - f^*_{pq,s}\, \hat{\ell}^\pm_k$
11:             **end if**
12:         **end for**
13:         **return** $U_{t,i}$
14:     **end if**
15: **end kernel**

In the original CPU version, Algorithm 3, the SurfInt routine loops over all faces on the current rank. For each face, it obtains the orientation of the face with respect to the volume. The orientation of the face of a hexahedral DG element depends on which of its six local faces it refers to. The contribution of this face is then added to all DOFs within the element.

This operation is hard to parallelize for GPU hardware since all 6 local faces add their contribution to each individual DOF within the element. Writing to the same entries in an array multiple times can yield race conditions if the individual threads are

not properly synchronized–but synchronizing threads is costly. In GALÆXI, the sequence of operations is thus altered for the GPU implementation in comparison to the original CPU implementation. The developed algorithm, Algorithm 4, runs as follows. First, each GPU thread is assigned a single DOF within an element. Due to the tensor product structure of the DGSEM, this results in only a single DOF per face influencing the solution as indicated in Fig. 1. The thread then loops over all six faces (locFace) of the element. For each face, it identifies the face index s within the flux array and the corresponding DOF on the face specified by the indices p, q. The face whose normal vector is used to compute the Riemann flux is determined by the flag isPrimary, while for the adjacent element (isReplica face) the sign of the flux contribution has to be flipped to account for the fact that its outward facing normal vector points in the opposite direction. Additionally, the correct integration weight $\hat{\omega}$ is identified to add the flux contribution of this locFace to the respective DOF. While this requires multiple threads to access the same face data multiple times, it avoids race conditions between threads without the need of explicit synchronization, since only a single thread writes to a specific entry in the U_t array. Lastly, transforming the fluxes from the face-local to the element-local coordinate system requires some form of mapping. Since GALÆXI is an unstructured solver, the algorithm also needs to account for the case where coordinate systems of neighboring elements are rotated with respect to each other. The combination results in mappings which are non-trivial to obtain. However, the required mappings are hardware-agnostic and not relevant for the efficiency of the GPU kernel. In consequence, these specifics are condensed into a single call to a subroutine FaceMapping to keep the algorithm concise. More details on the face connectivity can be found in Krais et al. [21].

At this point, it is important to revisit the required compute time of the different DG operations as shown in Fig. 5. It is evident that the majority of the computational work can be attributed to the operations VolInt, Lift_VolInt, and ConsToPrim, which are operations local to each DG element that can be scheduled independently of any communication. This has three crucial implications. First, only a small number of routines require the majority of the compute time, which yields distinct targets for more sophisticated optimization. Second, these routines do not require any communication, which again highlights the beneficial ratio of local work to required communication of DG schemes. Third, the overhead introduced by the unstructured mesh is negligible, since the additional work is mainly limited to the routines mapping from the faces to the volumes, i.e. SurfInt and ProlongToFace, which take only around 15 % of the overall compute time.

3.4. Summary of the Parallelization Strategy

This section provides details on the parallelization concept of GALÆXI on three different levels. First, the parallelization of the workload between GPUs was introduced. Here, GALÆXI subdivides the domain into subdomains with roughly the same number of elements, which are then assigned to the individual GPUs and communication across the boundaries of neighboring subdomains is performed using CUDA-aware MPI.

Second, the individual compute kernels within the GPU are scheduled using streams to improve the overall utilization of the GPU. Operations associated with the MPI communication are assigned to the stream with the highest priority to allow the GPU to antedate the execution of these kernels to initiate the communication at the earliest possible point in time. Third, the design concepts of the kernels were introduced using the ConsToPrim operations as an example for pointwise operations and the SurfInt to detail the more intricate case of kernels that have to map from the elements' volume to their faces and vice versa. The resulting performance of the kernels demonstrates that the overhead of the unstructured mesh is negligible. A detailed discussion of the resulting parallel performance of GALÆXI across multiple GPUs is provided in the following paragraphs.

4. Performance Evaluation

In the following section, the performance and the scaling abilities of GALÆXI are demonstrated. First, Section 4.1 introduces the details of the applied systems, i.e. HAWK-AI and JUWELS Booster. Section 4.2 then derives the performance metrics that are used to evaluate the performance. With these in place, Section 4.3 provides details on the code's memory consumption while the results of the scaling tests are discussed in Section 4.4.

4.1. Hardware Architecture

The performance of GALÆXI and FLEXI is investigated for two different systems. First, the JUWELS Booster installed at the Jülich Supercomputing Centre (JSC) and second, the HAWK and HAWK-AI systems at the High-Performance Computing Center Stuttgart (HLRS).

The JUWELS Booster module entails a total of 936 two-socket nodes. Each node provides two AMD EPYC 7402 processors with 24 cores per socket and a total of 512 GiB of DDR4-3200 main memory per node. Each node comprises 4 NVIDIA A100 GPUs with 40 GiB memory interconeccted using NVlink, where each GPU is connected to its own network adapter and the individual nodes are integrated using a Mellanox HDR200 InfiniBand interconnect with 200 Gbit/s per adapter in a DragonFly+ topology.

The HAWK supercomputer at HLRS is based on an HPE Apollo 9000 with 5632 dual-socket nodes. Each node is equipped with two AMD EPYC 7742 CPUs, which yield 128 CPU cores per node. Each node comprises 256 GiB of main memory and the nodes are connected using a Mellanox HDR200 InfiniBand interconnect in a 9D-hypercube topology. The HAWK-AI partition of HAWK is based on an HPE Apollo 6500 Gen10 Plus with 24 nodes, where each node is equipped with two 64-core AMD EPYC 7702 processors, 8 NVIDIA A100 GPUs interconeccted using NVlink, and 1 TiB of main memory. 20 nodes employ A100 GPUs with 40 GiB memory and 4 nodes entail A100 GPUs in the 80 GiB version. The nodes of HAWK-AI are fully integrated into the main HAWK partition using an Inifiniband interconnect in a Fat-Tree topology, such that nodes from both systems can be used within a single compute job.

The HAWK-AI partition was designed to integrate AI and big data capabilities into traditional HPC jobs but is also capable of running and scaling GPU-accelerated HPC applications on its own.

4.2. Performance Metrics

In the following, we focus on two distinct metrics to quantify and compare the performance of GALÆXI and FLEXI on different hardware, which rely on the time-to-solution and the energy-to-solution paradigms, respectively. For this, we use the performance index (PID), which is a commonly used metric to quantify the performance of DG codes [21, 49] and is defined as

$$\text{PID} = \frac{\text{Walltime} \times \text{\#Ranks}}{\text{\#RK-stages} \times \text{\#DOF}}. \tag{22}$$

The PID describes the walltime required by a single rank to advance a single DOF for one stage of the explicit Runge–Kutta time-stepping. Hence, the PID is independent of the number of timesteps performed, the number of DOF used in the simulation and the number of ranks employed, where a rank refers either to a CPU core or a whole GPU as discussed in Section 3. While this provides a good measure of efficiency for code performance comparison on either CPU or GPU systems, the usefulness of this definition is limited when comparing GPU and CPU codes with each other. Here, a whole GPU would be compared to a single CPU core with a vastly different compute performance and power consumption. To account for the differences in hardware, we propose in this work the novel energy-normalized PID (EPID) as a more suitable measure of performance. The EPID is defined as

$$\text{EPID} = \frac{\text{Walltime} \times \text{Power}}{\text{\#RK-stages} \times \text{\#DOF}} = \underbrace{\frac{\text{Power}}{\text{\#Ranks}}}_{P_{\text{rank}}} \times \text{PID}, \tag{23}$$

and describes the energy required to compute the time update for a single DOF on the specific computing hardware. The EPID can thus be interpreted as the PID normalized by the specific power required per rank, which is denoted as P_{rank}.

4.3. Memory Requirements

The memory consumption of a real-world application on the device is given in Table 2 in KiB per DOF for different polynomial degrees N. In general, the overall memory consumption is low, which is a well-known property of the explicit numerical scheme. The results clearly show that increasing N improves the memory efficiency, i.e. reduces the required amount of memory per DOF. This is because GALÆXI stores both the solution for the DOFs within the DG element $((N + 1)^3)$ and on its surfaces $(6(N + 1)^2)$. With increasing N, the ratio between surface to volume information thus decreases, yielding a lower overall memory footprint. As an illustration of memory efficiency, it is possible to compute a problem with $N = 7$ and 48 million DOF per solution variable on a single device with 40 GiB of memory.
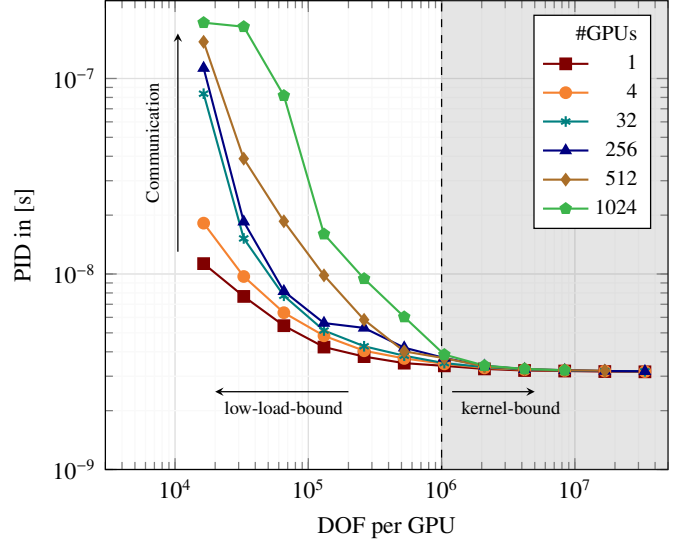
Figure 6: Scaling results for GALÆXI with the split-form DG scheme and $N = 7$ plotted as PID over the specific load, i.e. DOF per GPU, for up to 1024 GPUs.

4.4. Scaling Tests

To evaluate the scalability of GALÆXI on HPC systems, its parallel performance is evaluated on the JUWELS booster module using up to 1024 GPUs for a wide range of problem sizes. For this, the spatial resolution of a Cartesian mesh with $4 \times 4 \times 2 = 32$ elements is successively doubled in each spatial direction until the finest resolution of $256^3 = 16.8 \times 10^6$ elements is reached. For a polynomial degree of $N = 7$, which is a typical choice for production runs, this results in 16 384 to 8.6×10^9 DOF, respectively. The case for the scaling test is based on the setup of the incompressible Taylor–Green-Vortex considered in Section 5.2. Its simulation domain is a triple periodic box with corresponding initial conditions. However, in contrast to an implicit time integration scheme, the computational cost and thus the scaling behavior of the explicit scheme is independent of the chosen initial condition. Each computation is advanced for 100 timesteps and the scaling properties are evaluated based on the PID. Here, only the time for the timestepping is considered and initialization and analyze routines are neglected. The results of the scaling tests are presented from three different perspectives—first, the influence of the computational load per GPU on the overall performance, second, investigating the parallel efficiency in a weak scaling setting and third, from a strong scaling perspective.

In a first step, the PID is plotted against the specific load in terms of DOF per GPU in Fig. 6. Since the PID is a measure of computational time, a lower PID indicates better performance. Most strikingly, all curves converge above the limit of 10^6 DOF per GPU, which means that the overhead of the parallelization and communication becomes negligible in comparison to using only a single GPU. Hence, GALÆXI scales almost perfectly beyond the threshold of 10^6 DOF per GPU. The behavior changes for loads below this threshold. Here the PID increases towards lower loads for all cases, which means that the com-

Table 2: Measured memory consumption per DOF on the GPU for different polynomial degrees N and the Navier–Stokes equation system.

N	1	2	3	4	5	6	7	8	9	10	11	12
KiB	1.457	1.188	1.049	0.996	0.942	0.895	0.869	0.841	0.827	0.808	0.801	0.787

putational efficiency decreases. Moreover, the more GPUs are used for the simulation, the more pronounced this loss in performance becomes. This can be attributed to two factors. First, the communication latency between the GPUs cannot be hidden completely at low loads, since the amount of local work is insufficient to hide the communication. Furthermore, the loss in performance becomes more pronounced the more potential communication partners, i.e. GPUs, are used for the simulation. The severity of this performance penalty depends strongly on the network topology of the HPC system and the job placement on the system, which is determined by the scheduler. In the case of the JUWELS booster module, which uses a DragonFly-type network topology, the communication cost increases significantly when the nodes are spread across a larger number of switch groups, which contain 192 GPUs each. However, lacking latency hiding cannot explain the performance loss when using a single GPU, since here no communication is necessary. Instead, this drop in performance can be attributed to the overhead associated with launching kernels on the GPU. If the actual computational load of the kernel becomes too small, the kernels cannot be launched quickly enough to use the GPU to capacity. Moreover, tail effects become noticeable, as discussed in Section 3. To summarize the results, the GPU implementation can be seen to be *kernel-bound* for high loads, where the performance becomes independent of the total number of GPUs used. For very low loads, the performance gets *low-load-bound* and becomes increasingly *communication-bound* with a dominant performance penalty the more compute nodes are used. This is in stark contrast to the CPU implementation of FLEXI as reported by Blind et al. [50]. Here, the impact of the communication overhead is similarly noticeable for very low loads. However, a performance penalty also appears for very high loads, since here the fast CPU cache cannot hold all necessary data and the bandwidth to the main memory becomes the bottleneck. This results in a narrow band in the rage of 3000 to 10 000 DOF per rank, where optimal performance is achieved [21, 50]. In the case of GALÆXI, increasing the load only improves the overall performance with the available GPU memory as the single limiting factor.

In Fig. 7 the investigated weak scaling properties of GALÆXI are depicted. In the weak scaling paradigm, the problem size and the amount of compute resources are increased proportionally, such that the overall load per GPU is kept constant for each case. The parallel efficiency is presented with respect to the performance of a complete compute node equipped with 4 GPUs (parallel efficiency equal to 1). This is done in order to take communication into account in a meaningful way, as the baseline case includes MPI communication. It also enables a suitable assessment of the communication overhead in the case that only one GPU is used without communication. The results
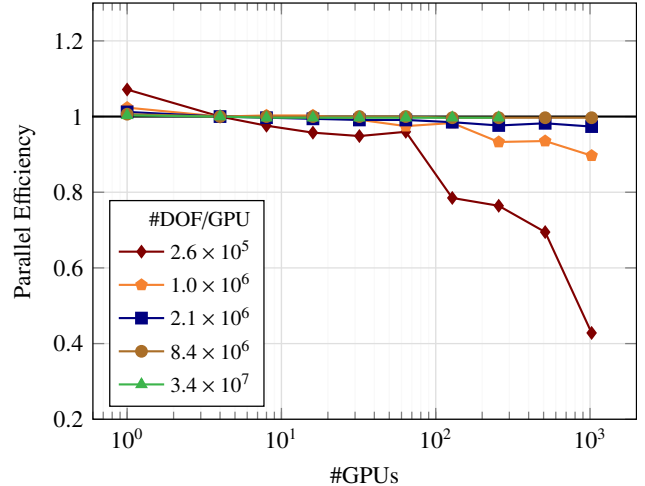

Figure 7: Weak scaling of GALÆXI with the split-form DG scheme and $N = 7$ plotted as the parallel efficiency over the number of GPUs for specific loads, i.e. DOF per GPU. The parallel efficiency is computed based on the PID on a single node, i.e. on 4 GPUs.

show again the threshold of 10^6 DOF per GPU as discussed before. For lower loads, the communication latency degrades the overall performance, while loads above 10^6 DOF per GPU show almost perfect weak scaling up to the maximum of 1024 GPUs.

Lastly, the results for strong scaling of GALÆXI are shown in Fig. 8. The strong scaling analysis investigates how performance changes when the number of GPUs is increased while the problem size remains constant. This causes problems for large cases, which might not fit into the memory of only a few GPUs, but rather require a larger number of GPUs to be actually computed. This limitation is the reason why the scaling results in Fig. 8 not exclusively start at a single GPU, but rather at the minimum number of GPUs required to compute the given problem size due to memory constraints. The performance using this minimum number of GPUs is then used to compute the relative speedup when increasing the number of GPUs. The strong scaling capabilities of GALÆXI are excellent up to the maximum 1024 GPUs, as long as the computational load exceeds the threshold of 10^6 DOF per rank, which is indicated explicitly for both cases. Below this threshold, i.e. towards larger number of GPUs, the load per device is insufficient to exploit the computing power of the GPU and to hide the necessary communication, which results in the loss of performance. This also matches the results by Fischer et al. [15], who report that NekRS reaches its limit for strong scaling at a similar load of about 2 to 4 million DOF per rank. For computational loads above this threshold, GALÆXI yields almost perfect strong scaling results up to the maximum of 1024 GPUs.
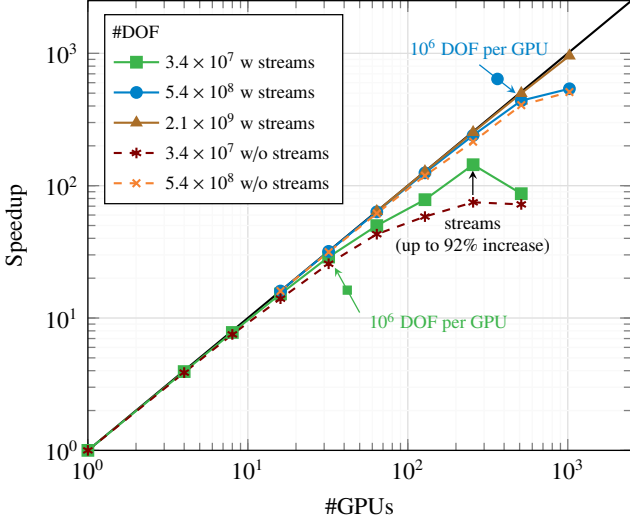
13

Figure 8: Strong scaling of GALÆXI with the split-form DG scheme and $N = 7$ plotted as the speedup over the number of GPUs for three problem sizes. For two cases, the results without the use of parallel streams are shown dashed. The speedup is computed based on the smallest number of GPUs that was able to run the given case. The ideal speedup is shown in black.

Next, the influence of our scheduling strategy based on parallel streams and introduced in Section 3.2 is investigated. For this, Fig. 8 also shows the scaling results for the same problem sizes, with (solid lines) and without (dashed lines) the use of parallel streams for kernel scheduling. For both setups, the omittance of stream scheduling results in a significant loss in parallel performance for low loads. This can be attributed to two aspects. First, parallel streams allow for hiding the overhead of kernels launches and tail effects for low loads. However, more importantly, our implementation permits the GPU to preempt the computation of quantities that have to be communicated via MPI. This facilitates more efficient communication latency hiding, resulting in better parallel performance in cases involving many communication partners and low amounts of local work.

5. Verification & Validation

5.1. Verification - Convergence Tests

The correct implementation of the high-order accurate numerical schemes in GALÆXI is verified by testing the order of convergence of the spatial operator with the method of manufactured solutions [51]. This method allows the derivation of source terms for nonlinear partial differential equations that lead to exact solutions that can be expressed in analytical form and allow computing the error of the numerical discretization scheme. Following Hindenlang et al. [52], the exact function is assumed to follow a sinusoidal solution of the form

$$
\begin{aligned}
\rho(\boldsymbol{x}, t) &= 2 + A \sin\left(2\pi(x + y + z - at)\right), \\
\boldsymbol{u}(\boldsymbol{x}, t) &= 2 + A \sin\left(2\pi(x + y + z - at)\right), \\
E(\boldsymbol{x}, t) &= \left(2 + A \sin\left(2\pi(x + y + z - at)\right)\right)^2,
\end{aligned}
\tag{24}
$$

where the amplitude and advection speed are chosen as $A = 0.1$ and $a = 1$, respectively. This solution describes an oblique, periodic wave that is advected linearly with speed a. The source terms that are required for Eq. (24) to be an exact function of the NSE are detailed in Gassner et al. [53]. The problem is then initialized within a domain of $\boldsymbol{\Omega} \in [-1, 1]^3$ with periodic boundary conditions and is discretized with varying $N \in [2, 9]$. The meshes are varied in the range of containing a single element up to 64^3 elements at maximum. The computation is advanced in time up to $t = 1$ and the timestep is chosen sufficiently small to not influence the overall discretization error. The convergence test is carried out with both the standard collocation formulation on GL interpolation points and the split-flux formulation on LGL interpolation points. The results in Fig. 9 demonstrate that the expected design order is reached for all investigated cases, which verifies the correct implementation of the schemes.

5.2. Validation - Taylor–Green-Vortex

A popular validation case for turbulent flows is the Taylor–Green-Vortex (TGV) introduced by Taylor and Green [54]. One reason for its widespread use are its analytically prescribed initial conditions, which are given for a domain of size $\boldsymbol{\Omega} \in [0, 2\pi L]^3$ by

$$
\boldsymbol{u}(\boldsymbol{x}, 0) = \begin{pmatrix} U_0 \sin\left(\frac{x}{L}\right) \cos\left(\frac{y}{L}\right) \cos\left(\frac{z}{L}\right) \\ -U_0 \cos\left(\frac{x}{L}\right) \sin\left(\frac{y}{L}\right) \cos\left(\frac{z}{L}\right) \\ 0 \end{pmatrix},
\tag{25}
$$

$$
p(\boldsymbol{x}, 0) = p_0 + \frac{\rho_0 U_0^2}{16} \left(\cos\left(\frac{2x}{L}\right) + \cos\left(\frac{2y}{L}\right)\right)\left(2 + \cos\left(\frac{2z}{L}\right)\right),
$$

with $L = 1$ denoting the characteristic length, $U_0 = 1$ the magnitude of the initial velocity fluctuations and $\rho_0 = 1$ the reference density. The background pressure p_0 is chosen to fit a prescribed background Mach number $\mathrm{Ma}_0 = U_0 \sqrt{\rho_0/(\gamma p_0)}$ and the viscosity μ_{ref} is used to obtain the desired Reynolds number which is defined as $\mathrm{Re} = \rho_0 U_0 L/\mu_{ref}$. However, Eq. (25) does not yield sufficient initial conditions for a compressible flow field, since it lacks information about the density and temperature fields. Two different approaches are commonly used to extend it to a full description of a compressible flow field as required for the computation with a compressible solver. For this, either field is held constant, while the other quantity is computed to yield a thermodynamically admissible state. Assuming an perfect gas that follows Eq. (9) this yields the two variants

$$
\text{Version I:} \quad \rho(\boldsymbol{x}, 0) = \rho_0, \qquad T(\boldsymbol{x}, 0) = \frac{p}{R\rho_0}, \tag{26}
$$

$$
\text{Version II:} \quad \rho(\boldsymbol{x}, 0) = \frac{p}{RT_0}, \qquad T(\boldsymbol{x}, 0) = T_0. \tag{27}
$$

Two common metrics to assess the accuracy of numerical schemes for the TGV case are the instantaneous kinetic energy in the domain E_k and the viscous dissipation rate ε_T. The integral kinetic energy is defined as

$$
E_k = \frac{1}{2\rho_0 U_0^2 |\Omega|} \int_\Omega \rho \boldsymbol{u} \cdot \boldsymbol{u} \, \mathrm{d}\Omega, \tag{28}
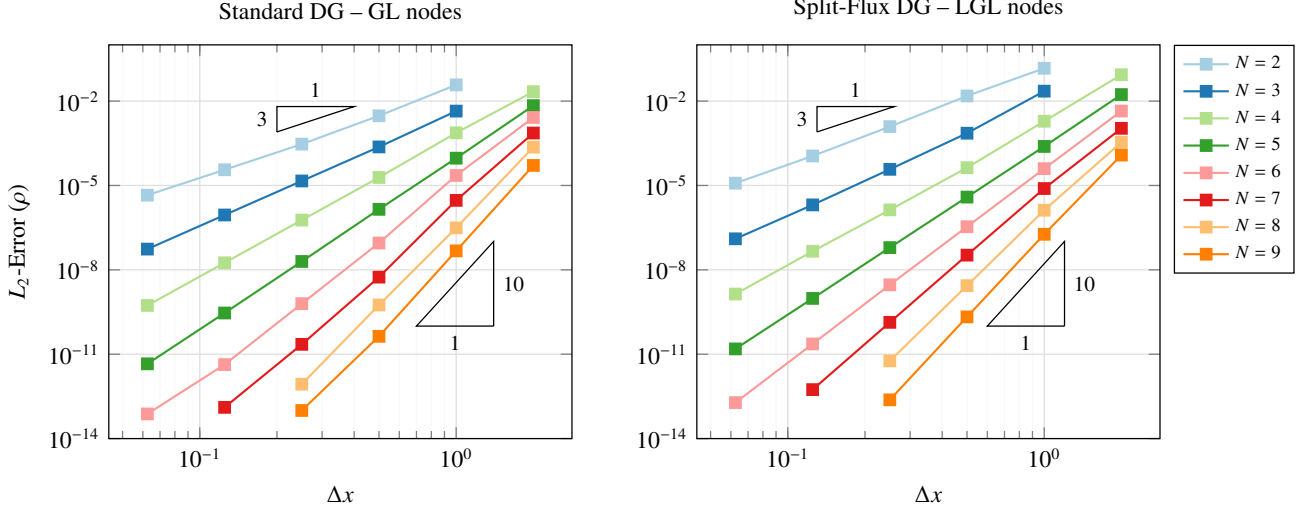$$

14

Figure 9: Convergence of the split-flux DG scheme on LGL nodes (*left*) and the standard DG scheme on GL nodes (*right*) using $N \in [2, 9]$ for the manufactured solution.

where $|\Omega|$ denotes the overall size of the integration domain. The viscous dissipation rate of the kinetic energy can be split into a solenoidal and a dilatational contribution (Zeman [55], Sarkar et al. [56]), which are defined as

$$\varepsilon_S = \frac{L^2}{\text{Re} U_0^2 |\Omega|} \int_\Omega \frac{\mu(T)}{\mu_0} \, \boldsymbol{\omega} \cdot \boldsymbol{\omega} \, \mathrm{d}\Omega, \tag{29}$$

$$\varepsilon_D = \frac{4L^2}{3\text{Re} U_0^2 |\Omega|} \int_\Omega \frac{\mu(T)}{\mu_0} \, (\nabla \cdot \boldsymbol{u})^2 \, \mathrm{d}\Omega, \tag{30}$$

respectively. The solenoidal component ε_S can be related to the vortical motion and the dilatational component ε_D to compressibility effects.

Two versions of the TGV case are investigated, which both exhibit a Reynolds number of Re = 1600 with the initial conditions prescribed in Eq. (25). First, the weakly compressible case with $\text{Ma}_0 = 0.1$ is investigated to verify that GALÆXI accurately captures the physics of turbulent flow. In a second step, the Mach number is increased to $\text{Ma}_0 = 1.25$, which causes complex shock patterns to emerge during the simulation. Consequently, this supersonic TGV setup is a suitable test case to assess the stability and accuracy of compressible flow solvers for shock-turbulence interaction.

Incompressible TGV

First, we consider the TGV at Re = 1600 in the incompressible limit with $\text{Ma}_0 = 0.1$ and Version II, i.e. an initially constant temperature field. Four different uniform resolutions were investigated to demonstrate the mesh convergence of the code. For this, either 8^3, 16^3, 32^3, or 64^3 elements were used with a polynomial degree of $N = 7$, corresponding to 64^3, 128^3, 256^3, or 512^3 DOF. Two simulations were carried out for each mesh, first with the GPU-accelerated GALÆXI and second with its CPU-based predecessor FLEXI for verification purposes. The results are also validated against the high-fidelity reference solution published by DeBonis [57]. The results shown in Fig. 11
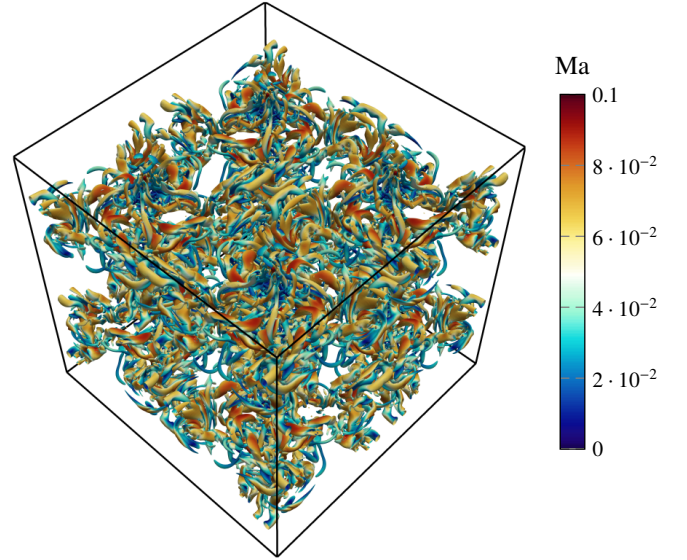

Figure 10: Instantaneous flow field of the incompressible TGV case with $\text{Ma}_0 = 0.1$ and Re = 1600 at time $t = 10$ using 512^3 DOF visualized by iso-surfaces of the Q-criterion colored by Ma.

(*left*) demonstrate that GALÆXI and FLEXI yield the same results up to machine precision. Moreover, as the resolution increases, the temporal evolution of the dissipation rate converges to the reference solution, to the point where the solution on the finest mesh with 512 DOF in each direction matches the reference almost perfectly. The instantaneous flow field of the TGV case at $t = 10$ is illustrated in Fig. 10, highlighting its vortex structures using iso-contours of the Q-criterion colored by the Mach number.

Compressible TGV

More recently, the TGV case was extended to the compressible regime by increasing the Mach number of the initial flow
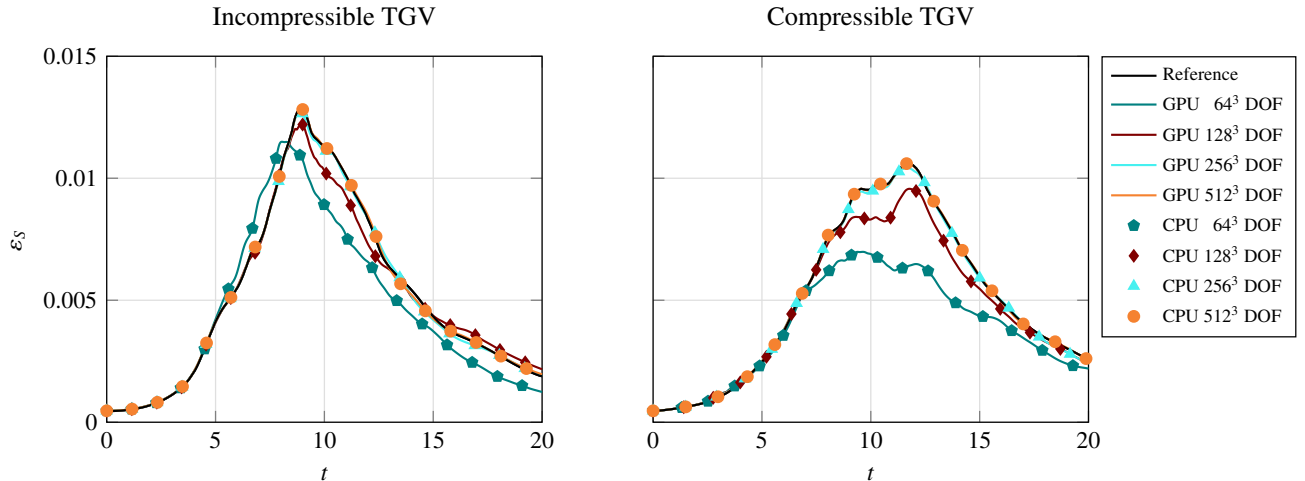
15

Figure 11: Temporal evolution of the solenoidal dissipation rate ε_S for the incompressible TGV at $Ma_0 = 0.1$ (*left*) and the compressible TGV at $Ma_0 = 1.25$ (*right*) using between 8^3 and 64^3 elements with a polynomial degree $N = 7$ which results in 64^3 to 512^3 DOF. The results by DeBonis [57] (*left*) and Chapelier et al. [58] (*left*) serve as the reference solution for the incompressible and compressible case, respectively. The results of the CPU implementation are given for reference.

field [59, 58]. A common choice is $Ma_0 = 1.25$, for which complex shock patterns emerge that interact with the turbulent flow. Consequently, the compressible, supersonic TGV case allows for assessing the stability and accuracy of compressible flow solvers for shock-turbulence interactions. The simulation is again initialized using the setup in Eq. (27), i.e. Version II, and Sutherland's law is applied to address the dependency of the viscosity on the temperature in the compressible case. The shock capturing scheme introduced in Section 2.3 is applied for the stabilization of the scheme near shocks. Again, four mesh resolutions were investigated with 64^3, 128^3, 256^3, and 512^3 DOF and a polynomial degree of $N = 7$. The permitted maximum of the blending parameter α is set identically across all investigated resolutions. The results reported by Chapelier et al. [58] serve as the reference solution. The results in Fig. 11 (*right*) again show that GALÆXI and FLEXI yield identical results for the temporal evolution of the solenoidal dissipation rate. Moreover, at higher resolutions, the results converge to the reference solution, where the results are almost identical for the largest case of 512 DOF per spatial direction.

6. Application

Based on these verification and validation results, both GA-LÆXI and FLEXI are applied to the large-scale application case of a wall-resolved LES of the NASA Rotor 37 [22]. This allows for verification that GALÆXI can handle complex simulations of compressible flow and quantify the gains in efficiency and energy-to-solution by using GPUs. For this, Section 6.1 first provides some background on the case, while Section 6.2 gives details on the computational setup. Finally, the results are discussed in Section 6.3.

6.1. Description

In the following section, the applicability of GALÆXI towards large-scale test cases is demonstrated for the turbulent

flow within a NASA Rotor 37 rectilinear transonic compressor cascade. This rotor was originally employed in one of four transonic axial-flow compressor stages designed and tested at the NASA Lewis Research Center in the late 1970s [22]. With its geometry parameters and measurement data publicly available [60, 61], the rotor has since become a benchmark test case in the turbomachinery research community including CFD studies [62], investigation of optimization techniques [63], tip leakage flow analysis [64], and uncertainty quantification approaches [65]. At its design point, the rotor operates with a blade tip Mach number of 1.4939, generating an overall pressure ratio of 2.106. The setup investigated here corresponds to a ground-idle condition, providing a tip Mach number of 0.824 with a total pressure ratio of 1.305. The cascade geometry is generated by unwinding the blade profile at mid-span and extruding for 5 % of the chord length. The resulting Reynolds number based on the inflow velocity and the rotor chord is 972 550. The low operating point and the position at mid-span results in an inlet relative Mach number of 0.758 and an incidence relative to the mean camberline of 10.1°. Validation of the FLEXI solution against experimental integral data at mid-span is given in Kopper [66]. Results obtained from GALÆ-XI were confirmed to match the solution of the CPU-based framework. Both solvers predict a transonic expansion region forms on the suction side near the leading edge as a result of the high subsonic inflow velocity and near-stall condition. The region is terminated with a near-normal shock and subsequent shock-boundary-layer interaction with flow separation occurring throughout the suction side. On the pressure side, a small laminar separation region forms which is subsequently terminated by turbulent re-attachment.

6.2. Computational Setup

The computational setup is identical for both GALÆXI and FLEXI, except for the hardware on which the simulations are run. The mesh for the LES comprises one compressor pitch
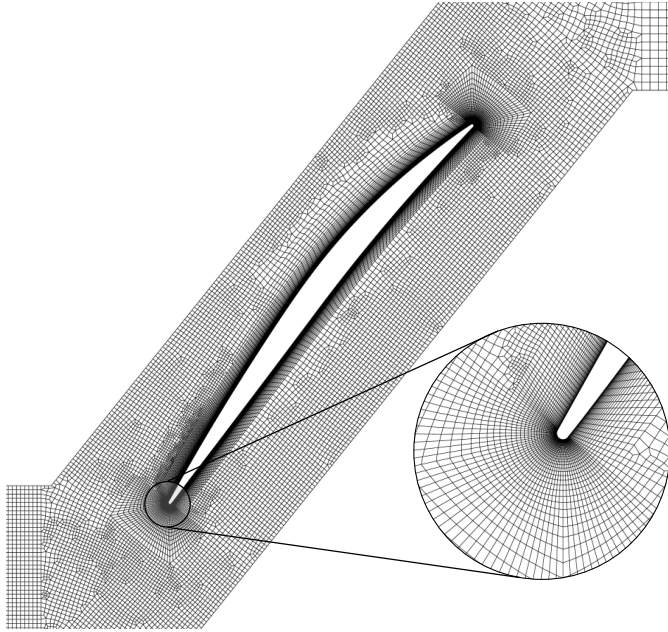
16

Figure 12: Computational mesh for the simulation of the NASA Rotor 37 case. The inflow and outflow regions are pruned and a zoom highlights the mesh around the leading edge.

with the compressor blade orientated with the stagger angle of 51.2° and is depicted in Fig. 12. The domain is discretized using 1.2×10^6 elements with $N = 5$, which results in a total of 2.6×10^8 DOFs for the simulation. The inflow is modeled using far-field conditions and a subsonic outflow condition [67] is employed. Additionally, sponge zones [68] are positioned at the inflow and outflow boundaries to prevent the formation of artificial reflections. The rotor itself is modeled as an adiabatic wall and the spanwise and pitchwise boundaries are defined as periodic. The simulation is performed using the split-form DG method as introduced in Section 2.2 to mitigate aliasing errors with the flux formulation given by Pirozzoli [32]. The solution is advanced in time using a 14-stage 4th-order Runge–Kutta method [26]. During the simulation, the viscosity is computed with Sutherland's law as given in Eq. (6).

The computational resources are chosen such that both codes run at their maximum efficiency. For GALÆXI, 128 Nvidia A100 GPUs on HAWK-AI are employed, which yields a total load of 2.0×10^6 DOF per GPU. For FLEXI, the number of CPU nodes is chosen such that the walltime is similar to the GALÆXI computation, which is obtained when using 256 nodes (32 768 CPU cores). This results in a load of around 7900 DOF/core, which resides well within the performance optimum of FLEXI [21]. The details of these setups are summarized in Table 3. The simulations are initialized with a precomputed converged flow state and are advanced for a total of 8 characteristic time units $t^* = t\,u_\infty/c$, where t^* is defined with respect to the inflow velocity u_∞ and the chord length c.

6.3. Results

The instantaneous Mach number distribution on the domain centerline computed by GALÆXI is depicted in Fig. 13. The

flow enters from the left with a high incidence relative to the camberline. This results in a shift of the stagnation point towards the pressure side and a strong transonic expansion fan on the suction side. The supersonic region is terminated with a near-normal shock, as illustrated in the zoom region. The corresponding pressure jump results in a forced boundary layer transition with high levels of unsteadiness. Numerical oscillations in the vicinity of the discontinuity, i.e. shock, resulting from Gibb's phenomenon, are mitigated with the convex blending approach outlined in Section 2.3. The grayscale overlay in the zoom region represents the local values of the blending factor α. It is evident that the FV shock capturing is active only near the shock in order to preserve the high numerical order of the DG operator in areas with a smooth solution. Downstream of the shock region, the separation of the boundary layer causes temporally varying blockage which couples with the upstream flow physics resulting in a highly unsteady flow field. Periods with enhanced separation result in counter-rotating vortex shedding as is visible near the wake downstream of the blade row.

The achieved performance for both codes is summarized in Table 3. For GALÆXI, a PID increase of about 35 % is observed in comparison to the performance reported in Section 4. This is attributed to the additional work and load imbalance between the ranks introduced by the test case, which includes the sponge zones, the boundary conditions, the shock indicator and the FV shock capturing scheme. The slight deviation in walltime per t^* between the GPU and CPU cases stems from choosing powers of two for the resources.

The power draw of the simulation runs was measured via the monitoring facilities of the systems' operators at the HLRS. It is important to stress that these measurement systems are not originally designed for the purpose of measuring the energy consumption of individual simulation runs, but rather to monitor and adjust the power draw of the overall facility. As a consequence, the power draw can only be measured on a rack-wise level and is limited in terms of accuracy and granularity, which means that the obtained results should be seen as a rough estimate. From these measurements, the specific power draw per rank P_{rank} shown in Table 3 is computed as the overall power delivered to the racks used divided by the number of ranks The measured power thus also includes the power for the network switches. It is important to note that due to the specific hardware layout, cooling is included in the total power consumption for the GPU case, while the cooling effort is not included for the CPU system. Hence, the obtained results tend to favor the CPU implementation and should thus be seen as a conservative lower bound for the potential gains in efficiency provided by GPU hardware.

When comparing the resulting EPID, i.e. the necessary amount of energy to advance a single DOF for a single time level, GALÆXI more than halves the required energy-to-solution. In total, GALÆXI requires around 147 kWh to advance the solution for one characteristic time unit t^*, while FLEXI requires around 339 kWh per t^* on CPUs. It is reasonable to relate this reduction in energy demand by GALÆXI to a similar reduction in associated carbon emissions. However, it is important to note that the I/O operations and analyzing routines are excluded from the
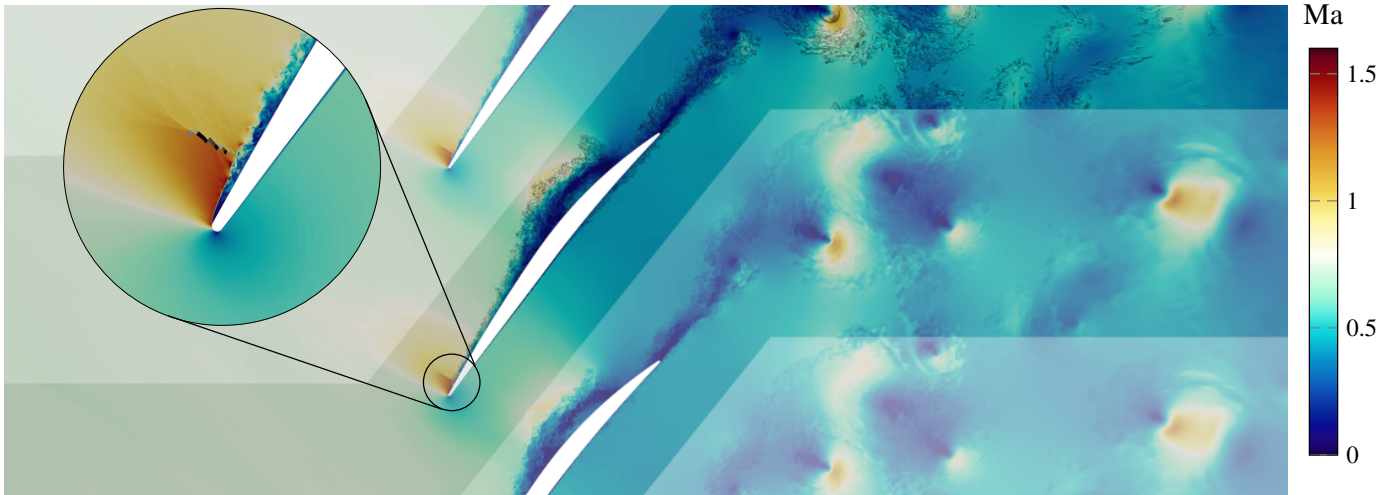
Figure 13: Instantaneous field solution for the NASA Rotor 37 case colored by the Mach number. A zoom of the leading edge highlights the supersonic flow region with the local blending values α of the FV shock capturing scheme overlaid for all elements with $\alpha = 0.1$ (light gray) up to $\alpha = 0.7$ (black). The domain is periodically extended which is indicated by a blurred overlay.

Table 3: Setup and performance results for the simulation runs on both CPU and GPU for a simulation time of $8t^*$.

	Ranks	DOF/Rank	P_{rank} [W]	PID [s]	EPID [J]	Walltime/t^* [s]	Energy/t^* [kWh]
GPU	128	2.03×10^6	448	4.58×10^{-9}	2.05×10^{-6}	9209	147
CPU	32 768	7.93×10^3	4.94	1.02×10^{-6}	5.06×10^{-6}	7538	339
Savings					59.5 %		56.8 %

PID computation. Since these operations are still performed on the CPU for GALÆXI, the resulting overhead causes a slight discrepancy in the savings for the EPID and energy-to-solution. As discussed before, due to the measurement limitations, both results should be regarded as an estimate and lower bound of the achieved performance.

7. Conclusion & Outlook

This work presents the open-source flow solver GALÆ-XI, which implements high-order DG methods on unstructured meshes for GPU-accelerated HPC systems. GALÆXI is the GPU-accelerated spinoff of the established FLEXI solver and it supports the majority of the features provided by FLEXI, which are continuously being extended. This allows the application of GALÆXI for scale-resolving simulations of complex compressible flows including shock waves using modern GPU-based HPC systems. This work provides details on the general code design, the parallelization strategy, and the implementation approach for the compute kernels. Thus, it serves as an indication on how existing spectral element codes can be ported efficiently for GPUs. As long as the GPUs are sufficiently loaded, the results demonstrate excellent scaling properties for GALÆXI on up to 1024 GPUs. The correct high-order accurate implementation of GALÆXI has been verified by demonstrating the expected convergence rates. Furthermore, the code has been validated against reference data for the incompressible and compressible variants of the established TGV.

As a demonstration of a large-scale application, GALÆXI was employed for the simulation of a wall-resolved LES of a NASA Rotor 37 compressor cascade. Using this example of compressible flow, the implemented finite volume subcell approach was demonstrated to yield a stable and accurate scheme for capturing the unsteady supersonic expansion region at the leading edge. In addition, GALÆXI has been shown to use only half the energy required to run the same simulation using the CPU implementation. With this, GALÆXI reduced the required energy from around 339 kWh to 147 kWh per characteristic time unit in comparison to the CPU implementation, which halved the associated carbon emissions.

Currently, GALÆXI is implemented using the CUDA Fortran framework, which does not support GPU hardware from vendors other than NVIDIA. Current efforts are focused on incorporating different compute backends into GALÆXI to support accelerator devices of different vendors alongside the baseline CPU implementation via hardware abstractions. The envisioned code is intended to be readily extendable to arbitrary compute devices, such that novel accelerator types can be incorporated without fundamental code redesigns. Concurrent work focuses on further optimization of key routines, in particular the VOLINT and FILLFLUX routines, which together consume almost half of the computing time as was demonstrated. Along the same lines, automatic tuning of hardware-specific launch configurations is to be integrated into the code. This is expected to provide high levels of performance across a wide range of different hardware. Here, the KernelTuner [48] package appears to

18

be a suitable choice. Lastly, graph-based approaches to domain decomposition might improve the utilization of the direct, high-bandwidth connection between individual GPUs on the same node by maximizing the amount of intra-node and minimizing the amount of inter-node communication.

This work has demonstrated that high-order DG methods are well-suited candidates for the efficient simulation of compressible flows on GPU systems. GALÆXI has showcased that unstructured mesh topologies and adequate state-of-the-art shock capturing based on FV subcells impose only negligible overhead on GPU hardware. Most importantly, GALÆXI is capable of reducing the carbon emission associated with large-scale flow simulations by more than 55 % in comparison to the CPU reference, which renders it a potent tool for the upcoming generation of sustainable, exascale HPC systems.

Data Availability Statement

The GALÆXI and FLEXI codes used within this work are available under the GPL-3.0 license at:

- `https://github.com/flexi-framework/galaexi`

- `https://github.com/flexi-framework/flexi`

The data generated in the context of this work and instructions to reproduce them with these codes are made available under the CC-BY 4.0 license sorted by section at:

- `10.18419/darus-4140` (Section 4)

- `10.18419/darus-4155` (Section 5.1)

- `10.18419/darus-4139` (Section 5.2)

- `10.18419/darus-4138` (Section 6)

Appendix A. Description of the Routines for the Discontinuous Galerkin Spectral Element Method

The following paragraph provides more detailed descriptions of the individual routines required to evaluate the three-dimensional DG operator given in Eq. (18). Please note that only the main routines listed in Table 1 are detailed here, which are the both the most important routines and also the ones discussed throughout the paper. Consequently, the lifting operator is omitted in the following. The routines are listed in the order of their computation in the DG scheme.

ConsToPrim

This routine computes the primitive variables U^{prim} from the state U. The primitive variables are required for two different purposes. First, the computation of the pressure p from the conserved variables U is required to evaluate the fluxes of the NSE. Secondly, the primitive variables are needed for the computation of the gradients in the lifting routines, which are required to evaluate the viscous fluxes. For an ideal gas, the primitive variables can be computed using the EOS in Eqs. (8) and (9). The computation of the primitive variables is DOF-local, since it only depends on the conserved state U at a given point and can thus be performed independently for each interpolation point in the volume and on the element faces.

VolInt

The VolInt is the computationally most intense routine in the DGSEM. It can be separated into two parts. First, the evaluation of the fluxes $\mathcal{F}^{1,2,3}$ at each interpolation point of the element and second, the linewise multiplication of the fluxes with the derivative matrix $\hat{D}$, which entails the derivatives of the basis functions. The evaluation of the fluxes is again DOF-local and can be performed independently for each interpolation point. In contrast, the application of the derivative matrix $\hat{D}$ is not DOF-local, since the contribution to point (i, j, k) depends on the fluxes along the summation indices α, β, γ in Eq. (18), which are the linewise connections originally shown in Fig. 1. This introduces complications when trying to parallelize this routine on GPUs, because the update for a single interpolation point exhibits a wide stencil. Our approach was to take advantage of the shared memory of the GPU, such that one thread block of the GPU gets assigned a single DG element. This thread block then first computes all fluxes in the element, stores them in shared memory and then applies the derivative matrix to these fluxes stored in shared memory. This ensures that the volume fluxes as intermediate results do not have to be moved to main memory, every flux has to be computed only once, and the full parallelism of the algorithm can be exploited. If the split-flux formulation is employed, the algorithmic structure remains almost identical, but minor modifications are necessary, as already discussed in Section 2.2.

ProlongToFace

The ProlongToFace routine interpolates the solution for U, which are defined in the reference element, to the element faces. The solution on the faces is denoted as $\tilde{U}^{L/R}$, the solution from

the left and right neighbor. This is required later by the FILLFLUX routine to evaluate the fluxes across the faces. For this, the polynomial basis from Eq. (17) is evaluated at the faces with the current coefficients $\hat{U}$. Additional mappings then allow to infer to which global face number the specific local face of the element belongs in order to store the solution in the correct place. This mapping from element to face information is required due to the unstructured nature of the mesh. This mapping entails two different steps. First, it has to account for the orientation of the face-based coordinates to the element-based coordinate system. Second, the relative rotation of a face with its neighbor has to be considered. These transformations are performed here to ensure that the face information is already correctly aligned for the computation of the face fluxes in FILLFLUX.

FILLFLUX

This routine computes the common flux $f^* = f^*(\tilde{U}^L, \tilde{U}^R)$ across the element faces based on their left and right solution using a Riemann solver. Since the correct mapping and orientation of the neighboring faces is already ensured by the PROLONGTOFACE routine, the Riemann solver can be applied directly. The routine is thus DOF-local and easily parallelizable for GPUs, since the Riemann solver can be applied independently for each interpolation point on the faces. In addition, this routine multiplies the flux with the surface element $\hat{s}$, which entails the size of the face.

SURFINT

This routine integrates the fluxes on the faces of the element and adds their contribution to the right-hand side of the DGSEM. Again, this routine entails a surface to volume mapping, due to the same reasons as in the PROLONGTOFACE routine. Since this routine is discussed at length in Section 2.2, we refer to this section for further details.

APPLYJAC

The APPLYJAC routine divides the right-hand side of Eq. (18) with the Jacobian $\mathcal{J}_{ijk}$ of the mapping for each individual interpolation point. This operation is thus DOF-local in the sense that the division can be performed independently for each interpolation point. This operation exhibits a low arithmetic intensity, since it performs only 5 floating point multiplications per interpolation point, but 6 floats to be copied from memory (one entry for each conserved variable in U and the (pointwise scalar) Jacobian $\mathcal{J}$). Hence, further optimization of the GPU implementation could fuse this operation for instance with the priorly computed SURFINT to increase the number of operations performed for each byte of data transferred.

References

[1] P. Lynch, The origins of computer weather prediction and climate modeling, Journal of Computational Physics 227 (2008) 3431–3444.

[2] L. Gundelwein, J. Miró, F. G. Barlatay, C. Lapierre, K. Rohr, L. Duong, Personalized stent design for congenital heart defects using pulsatile blood flow simulations, Journal of Biomechanics 81 (2018) 68–75.

[3] M. S. Lundstrom, M. A. Alam, Moore's law: The journey ahead, Science 378 (2022) 722–723.

[4] TOP500 - November 2023, `https://www.top500.org/lists/top500/list/2023/11/`, 2023. Accessed: 2024-03-13.

[5] GREEN500 - November 2023, `https://www.green500.org/list/2023/11/`, 2023. Accessed: 2024-03-13.

[6] D. Moxey, C. D. Cantwell, Y. Bao, A. Cassinelli, G. Castiglioni, S. Chun, E. Juda, E. Kazemi, K. Lackhove, J. Marcon, et al., Nektar++: Enhancing the capability and application of high-fidelity spectral/hp element methods, Computer Physics Communications 249 (2020) 107110.

[7] E. Ferrer, G. Rubio, G. Ntoukas, W. Laskowski, O. A. Mariño, S. Colombo, A. Mateo-Gabín, H. Marbona, F. M. de Lara, D. Huergo, et al., HORSES3D: A high-order discontinuous Galerkin solver for flow simulations and multi-physics applications, Computer Physics Communications 287 (2023) 108700.

[8] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W. A. Wall, J. Witte, Exadg: High-order discontinuous galerkin for the exa-scale, in: Software for exascale computing-SPPEXA 2016-2019, Springer International Publishing, 2020, pp. 189–224.

[9] H. Ranocha, M. Schlottke-Lakemper, A. R. Winters, E. Faulhaber, J. Chan, G. Gassner, Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing, Proceedings of the JuliaCon Conferences 1 (2022) 77. doi:10.21105/jcon.00077. arXiv:2108.06476.

[10] A. F. Antoniadis, D. Drikakis, P. S. Farmakis, L. Fu, I. Kokkinakis, X. Nogueira, P. A. Silva, M. Skote, V. Titarev, P. Tsoutsanis, UCNS3D: An open-source high-order finite-volume unstructured CFD solver, Computer Physics Communications 279 (2022) 108453.

[11] M. Bernardini, D. Modesti, F. Salvadore, S. Pirozzoli, STREAmS: A high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows, Computer Physics Communications 263 (2021) 107906.

[12] M. Bernardini, D. Modesti, F. Salvadore, S. Sathyanarayana, G. Della Posta, S. Pirozzoli, STREAmS-2.0: Supersonic turbulent accelerated Navier-Stokes solver version 2.0, Computer Physics Communications 285 (2023) 108644.

[13] P. Fischer, J. Lottes, H. Tufo, Nek5000, Technical Report, Argonne National Lab, Argonne, IL (United States), 2007.

[14] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, P. Fischer, OpenACC acceleration of the Nek5000 spectral element code, The International Journal of High Performance Computing Applications 29 (2015) 311–319.

[15] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, et al., NekRS, a GPU-accelerated spectral element Navier–Stokes solver, Parallel Computing 114 (2022) 102982.

[16] D. S. Medina, A. St-Cyr, T. Warburton, OCCA: A unified approach to multi-threading languages, arXiv preprint arXiv:1403.0968 (2014).

[17] N. Jansson, M. Karp, A. Podobas, S. Markidis, P. Schlatter, Neko: A modern, portable, and scalable framework for high-fidelity computational fluid dynamics, Computers & Fluids (2024) 106243.

[18] F. D. Witherden, B. C. Vermeire, P. E. Vincent, Heterogeneous computing on mixed unstructured grids with PyFR, Computers & Fluids 120 (2015) 173–186.

[19] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The deal. II finite element library: Design, features, and insights, Computers & Mathematics with Applications 81 (2021) 407–422.

[20] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, et al., MFEM: A modular finite element methods library, Computers & Mathematics with Applications 81 (2021) 42–74.

[21] N. Krais, A. Beck, T. Bolemann, H. Frank, D. Flad, G. Gassner, F. Hindenlang, M. Hoffmann, T. Kuhn, M. Sonntag, C.-D. Munz, FLEXI: A high order discontinuous Galerkin framework for hyperbolic–parabolic conservation laws, Computers & Mathematics with Applications 81 (2021) 186–219.

[22] L. Reid, R. D. Moore, Design and overall performance of four highly loaded, high speed inlet stages for an advanced high-pressure-ratio core compressor, Technical Report 1337, NASA Lewis Research Center, Cleveland, OH, United States, 1978.

[23] W. Sutherland, LII. The viscosity of gases and molecular force, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of

Science 36 (1893) 507–531.

[24] D. A. Kopriva, Implementing spectral methods for partial differential equations: Algorithms for scientists and engineers, Springer Science & Business Media, 2009.

[25] M. H. Carpenter, C. A. Kennedy, Fourth-order 2N-storage Runge–Kutta schemes, Technical Report NASA-TM-109112, NASA, Langley Research Center, 1994.

[26] J. Niegemann, R. Diehl, K. Busch, Efficient low-storage Runge–Kutta schemes with optimized stability regions, Journal of Computational Physics 231 (2012) 364–372.

[27] R. M. Kirby, G. E. Karniadakis, De-aliasing on non-uniform grids: Algorithms and applications, Journal of Computational Physics 191 (2003) 249–264.

[28] A. D. Beck, D. G. Flad, C. Tonhäuser, G. Gassner, C.-D. Munz, On the influence of polynomial de-aliasing on subgrid scale models, Flow, Turbulence and Combustion 97 (2016) 475–511.

[29] J. S. Hesthaven, T. Warburton, Nodal discontinuous Galerkin methods, Texts in Applied Mathematics, Springer, New York, NY, 2007.

[30] D. Flad, A. Beck, C.-D. Munz, Simulation of underresolved turbulent flows by adaptive filtering using the high order discontinuous Galerkin spectral element method, Journal of Computational Physics 313 (2016) 1–12.

[31] G. J. Gassner, A. R. Winters, D. A. Kopriva, Split form nodal discontinuous Galerkin schemes with summation-by-parts property for the compressible Euler equations, Journal of Computational Physics 327 (2016) 39–66.

[32] S. Pirozzoli, Numerical methods for high-speed flows, Annual Review of Fluid Mechanics 43 (2011) 163–194.

[33] F. Bassi, S. Rebay, A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier–Stokes equations, Journal of Computational Physics 131 (1997) 267–279.

[34] A. D. Beck, T. Bolemann, D. Flad, H. Frank, G. J. Gassner, F. Hindenlang, C.-D. Munz, High-order discontinuous Galerkin spectral element methods for transitional and turbulent flow simulations, International Journal for Numerical Methods in Fluids 76 (2014) 522–548.

[35] B. Cockburn, C.-W. Shu, TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework, Mathematics of computation 52 (1989) 411–435.

[36] L. Krivodonova, Limiters for high-order discontinuous Galerkin methods, Journal of Computational Physics 226 (2007) 879–896.

[37] X. Zhang, C.-W. Shu, On positivity-preserving high order discontinuous galerkin schemes for compressible euler equations on rectangular meshes, Journal of Computational Physics 229 (2010) 8918–8934.

[38] A. Burbeau, P. Sagaut, C.-H. Bruneau, A problem-independent limiter for high-order runge–kutta discontinuous galerkin methods, Journal of Computational Physics 169 (2001) 111–150.

[39] J. Zeifang, A. Beck, A data-driven high order sub-cell artificial viscosity for the discontinuous Galerkin spectral element method, Journal of Computational Physics 441 (2021) 110475.

[40] P.-O. Persson, J. Peraire, Sub-cell shock capturing for discontinuous Galerkin methods, in: 44th AIAA Aerospace Sciences Meeting and Exhibit, 2006, p. 112.

[41] A. Klöckner, T. Warburton, J. S. Hesthaven, Viscous shock capturing in a time-explicit discontinuous Galerkin method, Mathematical Modelling of Natural Phenomena 6 (2011) 57–83.

[42] M. Bohm, S. Schermeng, A. R. Winters, G. J. Gassner, G. B. Jacobs, Multi-element SIAC filter for shock capturing applied to high-order discontinuous Galerkin spectral element methods, Journal of Scientific Computing 81 (2019) 820–844.

[43] M. Sonntag, C.-D. Munz, Shock capturing for discontinuous Galerkin methods using finite volume subcells, in: Finite Volumes for Complex Applications VII-Elliptic, Parabolic and Hyperbolic Problems: FVCA 7, Berlin, June 2014, Springer, 2014, pp. 945–953.

[44] F. Fambri, M. Dumbser, O. Zanotti, Space-time adaptive ADER-DG schemes for dissipative flows: Compressible Navier–Stokes and resistive MHD equations, Computer Physics Communications 220 (2017) 297–318.

[45] S. Hennemann, A. M. Rueda-Ramírez, F. J. Hindenlang, G. J. Gassner, A provably entropy stable subcell shock capturing approach for high order split form DG for the compressible Euler equations, Journal of Computational Physics 426 (2021) 109935.

[46] A. Beck, M. Kurz, Toward discretization-consistent closure schemes for large eddy simulation using reinforcement learning, Physics of Fluids 35 (2023).

[47] A. M. Rueda-Ramírez, W. Pazner, G. J. Gassner, Subcell limiting strategies for discontinuous Galerkin spectral element methods, Computers & Fluids 247 (2022) 105627.

[48] B. van Werkhoven, Kernel Tuner: A search-optimizing GPU code autotuner, Future Generation Computer Systems 90 (2019) 347–358.

[49] N. Fehn, W. A. Wall, M. Kronbichler, A matrix-free high-order discontinuous Galerkin compressible Navier-Stokes solver: A performance comparison of compressible and incompressible formulations for turbulent incompressible flows, International Journal for Numerical Methods in Fluids 89 (2019) 71–102.

[50] M. Blind, M. Gao, D. Kempf, P. Kopper, M. Kurz, A. Schwarz, A. Beck, Towards exascale CFD simulations using the discontinuous Galerkin solver FLEXI, 2023. arXiv:2306.12891.

[51] P. J. Roache, Code verification by the method of manufactured solutions, J. Fluids Eng. 124 (2002) 4–10.

[52] F. Hindenlang, G. J. Gassner, C. Altmann, A. Beck, M. Staudenmaier, C.-D. Munz, Explicit discontinuous Galerkin methods for unsteady problems, Computers & Fluids 61 (2012) 86–93.

[53] G. J. Gassner, F. Lörcher, C.-D. Munz, J. S. Hesthaven, Polymorphic nodal elements and their application in discontinuous Galerkin methods, Journal of Computational Physics 228 (2009) 1573–1590.

[54] G. I. Taylor, A. E. Green, Mechanism of the production of small eddies from large ones, Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences 158 (1937) 499–521.

[55] O. Zeman, Dilatation dissipation: The concept and application in modeling compressible mixing layers, Physics of Fluids A: Fluid Dynamics 2 (1990) 178–188.

[56] S. Sarkar, G. Erlebacher, M. Y. Hussaini, H. O. Kreiss, The analysis and modelling of dilatational terms in compressible turbulence, Journal of Fluid Mechanics 227 (1991) 473–493.

[57] J. DeBonis, Solutions of the Taylor–Green vortex problem using high-resolution explicit finite difference methods, in: 51st AIAA Aerospace Sciences Meeting, 2013, p. 382.

[58] J.-B. Chapelier, D. J. Lusher, W. Van Noordt, C. Wenzel, T. Gibis, P. Mossier, A. D. Beck, G. Lodato, C. Brehm, M. Ruggeri, C. Scalo, N. Sandham, Comparison of high-order numerical methodologies for the simulation of the supersonic Taylor-Green vortex flow, submitted to Physics of Fluids (2024).

[59] D. J. Lusher, N. D. Sandham, Assessment of low-dissipative shock-capturing schemes for the compressible Taylor–Green vortex, AIAA Journal 59 (2021) 533–545.

[60] R. D. Moore, L. Reid, Performance of single-stage axial-flow transonic compressor with rotor and stator aspect ratios of 1.19 and 1.26 respectively, and with design pressure ratio of 2.05, Technical Report 1659, National Aeronautics and Space Administration, Cleveland, OH, United States, 1980.

[61] K. L. Suder, Experimental Investigation of the Flow Field in a Transonic, Axial Flow Compressor with Respect to the Development of Blockage and Loss, Ph.D. thesis, Case Western Reserve University, 1996.

[62] J. D. Denton, Lessons from Rotor 37, Journal of Thermal Science 6 (1997).

[63] E. Benini, Three-dimensional multi-objective design optimization of a transonic compressor rotor, Journal of Propulsion and Power 20 (2004) 559–565. doi:10.2514/1.2703.

[64] P. Seshadri, G. T. Parks, S. Shahpar, Leakage uncertainties in compressors: The case of rotor 37, Journal of Propulsion and Power 31 (2015) 456–466. doi:10.2514/1.b35039.

[65] A. Loeven, H. Bijl, The application of the probabilistic collocation method to a transonic axial flow compressor, in: 51st AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics, and materials conference 18th AIAA/ASME/AHS adaptive structures conference 12th, American Institute of Aeronautics and Astronautics, 2010. doi:10.2514/6.2010-2923.

[66] P. Kopper, An Euler–Lagrange Method for Compressible Dispersed Multiscale Flow, Ph.D. thesis, University of Stuttgart, 2024.

[67] J.-R. Carlson, Inflow/outflow boundary conditions with application to FUN3D, Technical Report NASA/TM–2011-217181, NASA, Langley Research Center, 2011.

[68] D. Flad, A. D. Beck, G. Gassner, C.-D. Munz, A discontinuous Galerkin spectral element method for the direct numerical simulation of aeroacoustics, in: 20th AIAA/CEAS aeroacoustics conference, 2014, p. 2740.

[69] Jülich Supercomputing Centre, JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre, Journal of large-scale research facilities 7 (2021). URL: `http://dx.doi.org/10.17815/jlsrf-7-183`. doi:`10.17815/jlsrf-7-183`.