Computer Physics Communications 295 (2024) 108993

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computational Physics

Domain-specific implementation of high-order Discontinuous Galerkin methods in spherical geometry ☆

Kalman Szenes [a], Niccolò Discacciati [b], Luca Bonaventura [c], William Sawyer [d,*]

[a] Swiss Federal Institute of Technology Zurich, Raemistrasse 101, 8092, Zurich, Switzerland
[b] Swiss Federal Institute of Technology Lausanne, Route Cantonale, 1015, Lausanne, Switzerland
[c] Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133, Milan, Italy
[d] Swiss National Supercomputing Centre, Via Trevano 131, 6900, Lugano, Switzerland

ARTICLE INFO

Keywords:
Domain-specific languages
GPU programming
Discontinuous Galerkin methods

ABSTRACT

In recent years, domain-specific languages (DSLs) have achieved significant success in large-scale efforts to reimplement existing meteorological models in a performance portable manner. The dynamical cores of these models are based on finite difference and finite volume schemes, and existing DSLs are generally limited to supporting only these numerical methods. In the meantime, there have been numerous attempts to use high-order Discontinuous Galerkin (DG) methods for atmospheric dynamics, which are currently largely unsupported in main-stream DSLs. In order to link these developments, we present two domain-specific languages which extend the existing GridTools (GT) ecosystem to high-order DG discretization. The first is a C++-based DSL called G4GT, which, despite being no longer supported, gave us the impetus to implement extensions to the subsequent Python-based production DSL called GT4Py to support the operations needed for DG solvers. As a proof of concept, the shallow water equations in spherical geometry are implemented in both DSLs, thus providing a blueprint for the application of domain-specific languages to the development of global atmospheric models. We believe this is the first GPU-capable DSL implementation of DG in spherical geometry. The results demonstrate that a DSL designed for finite difference/volume methods can be successfully extended to implement a DG solver, while preserving the performance-portability of the DSL.

1. Introduction

It has always been challenging for numerical mathematicians to implement new algorithms in a way that can attain the best possible performance of the underlying computing platform. This task has become even more difficult with the emergence of new hardware architectures, such as Graphics Processing Units (GPUs) or Field-programmable Gate Arrays (FPGAs), whose use often forces domain scientists to learn computing concepts well beyond their field of expertise.

Domain-specific languages (DSLs) [14] represent an attempt to separate the concerns of the domain scientist from the complexities of the underlying computing hardware. The efficacy of DSLs to make domain scientists, in particular atmospheric scientists, more productive has been illustrated in a number of papers, e.g., [7,17,29]. In the case of atmospheric dynamics, the method developer formulates her problem in terms of numerical operations, including time-stepping algorithms, linear algebra operations, or Partial Differential Equation (PDE) formulations, while a "backend" takes care of generating highly optimized code for the target architecture. This is extremely beneficial, as achieving peak performance on emerging parallel architectures remains a challenging task.

This approach contrasts with the standard development cycle, in which an initial serial algorithm has to be redesigned for parallel processing. For this an appropriate programming model needs to be chosen, which often comprises a combination of strategically placed compiler directives or routines written in low-level hardware-specific language extensions (such as CUDA or HIP), all while attempting to limit the memory transfers between the various processing units. Even after the initial parallel implementation has been validated, labor-intensive performance tuning is needed to fully exploit the capabilities of the

☆ The review of this paper was arranged by David W. Walker.

* Corresponding author.
E-mail addresses: kszenes@ethz.ch (K. Szenes), niccolo.discacciati@alumni.epfl.ch (N. Discacciati), luca.bonaventura@polimi.it (L. Bonaventura), william.sawyer@cscs.ch (W. Sawyer).

https://doi.org/10.1016/j.cpc.2023.108993
Received 31 May 2023; Received in revised form 30 September 2023; Accepted 23 October 2023
Available online 28 October 2023

hardware. The developed implementation usually becomes hardware-specific, making it not portable to emerging architectures, which are in fact now common: seven out of the top ten supercomputers [34] are based on either AMD or Nvidia hardware accelerators. DSLs thus can offer platform portability by decoupling the method development from the implementation.

Extensive effort has been made in developing frameworks to ease the challenge of solving PDEs in specified geometries. On the far end of the spectrum, software frameworks like FEniCS [3] and Firedrake [28] offer a descriptive language to define the PDE and the given domain, along with initial and boundary conditions. These frameworks then generate the code to solve the problem using finite element methods. FEniCS gives the user little latitude to test new numerical techniques, making it more appropriate for domain scientists relying on standard, widely-supported methods. Firedrake, on the other hand, has a different underlying implementation allowing more flexibility to employ various finite element approaches, code optimizations as well as parallelization features, such as multithreading, GPU acceleration, or distributed computing. Firedrake utilizes PyOp2 [27] – a full-fledged DSL for the parallel executions of computational kernels on unstructured meshes or graphs – to allow these interventions. More generally, PyOp2 can be viewed as a DSL for finite element calculations. PyOp2 was subsequently refined into the Psyclone [32] code-generation system, which was specifically designed to extend Fortran codes, and then used to implement the LFRic [21] atmospheric model. Psyclone is essentially a source-code generator and could conceptually address the algorithms we consider in this paper. However, in this study we focus on tools that have been developed internally at the Swiss National Supercomputing Centre (CSCS).

Our interests are in the area of weather and climate simulation, where the mesh is often built by extrusion of a two-dimensional horizontal mesh covering, for example, the whole globe. Our goal is to enable climate and numerical weather prediction (NWP) applications to leverage a variety of architectures by utilizing software backends. A first attempt at a C++-embedded DSL specific to the climate and weather domain was STELLA [17], with which the COSMO dynamical core (solver of the non-hydrostatic equations of atmospheric motion) was implemented [31]. This prototype was completely replaced by the GridTools [2] framework, in which the COSMO [17] and NICAM [22] models have been rewritten. The classic implementation of GridTools assumes a Cartesian grid, which explains the choices described in Section 2, however the newest version, released in 2023, also allows for a fully unstructured mesh. This DSL library is also the basis for a 30 person-year development effort within the EXCLAIM project [5] to port the ICON model in a performance-portable manner to CPUs and GPUs alike.

Up until recently, the numerical algorithms used by operational services to solve the equations of atmosphere motion (the fully compressible Euler equations) have been limited to spectral, spectral-element, finite-difference, and finite-volume methods, and the above-mentioned models are no exception. Recently there has been an extensive push to include Discontinuous Galerkin into the mix, see, among many others, [4,24,26,35]. To our knowledge, no mainstream DSL currently supports Discontinuous Galerkin methods in spherical geometry, a feature that would be extremely helpful in ongoing efforts to utilize DG for atmospheric dynamics. The nearest effort is perhaps [37], which, to our knowledge, is limited to Cartesian Geometry.

In this paper we consider two DSLs in the GridTools ecosystem which can, with certain extensions, support DG in spherical geometry, namely the C++-based Galerkin-for-Gridtools (G4GT) and the Python-based GridTools-for-Python (GT4Py) [30], as tools for implementing a high-order Discontinuous Galerkin (DG) method for time-dependent problems, see e.g., [15,19]. The GridTools framework was originally designed to support finite difference/volume methods on rectangular grids and, more recently, on unstructured grids. G4GT, now discontinued, was a first prototype to extend these tools for finite element problems. GT4Py is a Python layer above GridTools and thus targets the same FD/FV problems, for example [10]. GT4Py/GridTools now have been used for the porting of production models, among them FVM [38], FV3 [7] and ICON [5]. Such implementation projects last many years and involve a large team of scientific software developers.

This work is a more modest effort to make extensions to GT4Py so that it can be reused to implement a DG solver. As a proof of concept, we implement an explicit time discretization and a modal DG spatial discretization for a system of conservation laws. Common, but non-trivial, benchmarks, namely linear advection and the shallow water equations, are used to validate our DSL implementations of a DG method. While our experiment only concerns the DSL implementation of the discretization of a hyperbolic system, it is noteworthy that, in a DG context, the same data structures are employed also for more complex equations including RANS or LES turbulence models, see for example the discussion in [1]. Therefore, the results that we obtained are encouraging also with respect to the DSL implementation of more realistic models.

The structure of the paper is as follows. The flux formulation of the shallow water equations in latitude-longitude coordinates follows in Section 2. A preliminary implementation in an early C++-based DSL prototype called G4GT is briefly presented in Section 3.1, while the new, Python-based implementation called GT4Py is discussed at length in Section 3.2. The validation of the resulting implementations is discussed in Section 4 and benchmarks are presented in Section 5. In Section 6, we recount our experiences using the DSLs and make suggestions on future features to better support finite element codes in these frameworks.

2. The mathematical model and numerical discretization approach

We are concerned with demonstrating the capabilities of DSLs for implementing numerical solutions of conservation laws. A system of conservation laws can be written as:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{u}) = \mathbf{S}(\mathbf{u}), \tag{1}$$

where $\mathbf{u}$ is the vector of conserved variables, $\mathbf{F}$ is the flux function and $\mathbf{S}$ is the source term. Equation (1) becomes well posed once complemented with appropriate initial conditions and boundary conditions, see, for example, the discussion in [23]. Since our goal is the application of DSL tools to models for weather and climate, we choose as main model equations the shallow water equations (SWEs) on the sphere, which are a common benchmark for numerical models in this area. Various formulations of these equations can be found in [40]. We consider the Earth's surface as a sphere of radius R, that is parameterized in latitude - longitude (lat-lon) coordinates as a rectangular domain such that the latitude $\theta \in [-\pi/2, \pi/2]$ and the longitude $\lambda \in [0, 2\pi]$. Denote by $\Omega = 7.292 \times 10^{-5}\ \mathrm{s}^{-1}$ the Earth's rotation rate, by $f = 2\Omega \sin\theta$ the Coriolis parameter and by $g = 9.81\ \mathrm{m\,s}^{-2}$ the Earth's gravitational acceleration. Furthermore, let $\hat{\boldsymbol{\imath}}$, $\hat{\boldsymbol{\jmath}}$ denote the longitudinal and latitudinal unit vectors, respectively. For a generic scalar function ϕ and vector field $\mathbf{w} = w_1\hat{\boldsymbol{\imath}} + w_2\hat{\boldsymbol{\jmath}}$, the spherical gradient and divergence are defined as:

$$\begin{aligned} \nabla\phi &= \frac{\hat{\boldsymbol{\imath}}}{R\cos\theta}\partial_\lambda\phi + \frac{\hat{\boldsymbol{\jmath}}}{R}\partial_\theta\phi \\ \nabla\cdot\mathbf{w} &= \frac{1}{R\cos\theta}\left[\partial_\lambda(w_1) + \partial_\theta(w_2\cos\theta)\right]. \end{aligned} \tag{2}$$

Denoting then h as the thickness of a fluid over the spherical surface (assuming flat orography $h_b = 0$) and $\mathbf{v} = u\hat{\boldsymbol{\imath}} + v\hat{\boldsymbol{\jmath}}$ the velocity field, which is a tangent vector field to the sphere, the shallow water equations in flux form are written as:

$$\begin{aligned} &\partial_t h + \nabla\cdot(h\mathbf{v}) = 0 \\ &\partial_t(h\mathbf{v}) + \nabla\cdot(h\mathbf{v}\otimes\mathbf{v}) = -f\hat{\mathbf{k}}\times h\mathbf{v} - \nabla\left(\frac{gh^2}{2}\right). \end{aligned} \tag{3}$$

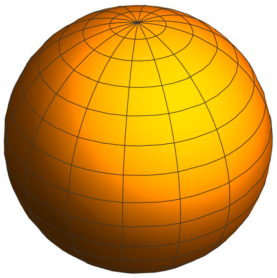


Fig. 1. Latitude-Longitude grid.

It is well known that lat-lon coordinates entail a number of numerical difficulties. Indeed, for a Cartesian lat-lon mesh such as the one depicted in Fig. 1, the elements become increasingly distorted as they approach the poles. Moreover, the elements precisely neighboring the poles have a singular edge in physical space (these elements reduce to spherical triangles instead of rectangles). While several alternatives have been considered in the literature, see, e.g., the review in [8], this setting is sufficient for the present purpose of validating GT4Py, which, as discussed in Section 1, could only handle non-Cartesian meshes in the release available to us.

The equations for the spherical components of the velocity field can then be derived taking into account the non-inertial nature of the rotating reference frame on the sphere, see again [40] (Equations (6)-(7)). We then obtain:

$$\begin{aligned}
&\partial_t(hu) + \nabla\cdot(hu\mathbf{v}) + \frac{1}{R\cos\theta}\partial_\lambda\left(\frac{gh^2}{2}\right) = \left(f + \frac{u}{R}\tan\theta\right)hv \\
&\partial_t(hv) + \nabla\cdot(hv\mathbf{v}) + \frac{1}{R}\partial_\theta\left(\frac{gh^2}{2}\right) = -\left(f + \frac{u}{R}\tan\theta\right)hu.
\end{aligned} \tag{4}$$

Since

$$\nabla\cdot(hu\mathbf{v}) = \frac{1}{R\cos\theta}\left[\partial_\lambda(hu^2) + \partial_\theta(huv\cos\theta)\right]$$

$$\nabla\cdot(hv\mathbf{v}) = \frac{1}{R\cos\theta}\left[\partial_\lambda(huv) + \partial_\theta(hv^2\cos\theta)\right]$$

and also

$$\frac{\cos\theta}{R}\partial_\theta\left(\frac{gh^2}{2}\right) = \frac{1}{R}\partial_\theta\left(\frac{gh^2}{2}\cos\theta\right) + \frac{gh^2}{2R}\sin\theta,$$

the SWE can be rewritten component-wise as:

$$\begin{aligned}
&\partial_t(h\cos\theta) + \frac{1}{R}\left[\partial_\lambda(hu) + \partial_\theta(hv\cos\theta)\right] = 0 \\
&\partial_t(hu\cos\theta) + \frac{1}{R}\left[\partial_\lambda\left(hu^2 + \frac{gh^2}{2}\right) + \partial_\theta(huv\cos\theta)\right] \\
&\qquad = \left(f\cos\theta + \frac{u}{R}\sin\theta\right)hv \\
&\partial_t(hv\cos\theta) + \frac{1}{R}\left[\partial_\lambda(huv) + \partial_\theta\left(\left(hv^2 + \frac{gh^2}{2}\right)\cos\theta\right)\right] \\
&\qquad = -\frac{gh^2\sin\theta}{2R} - \left(f\cos\theta + \frac{u}{R}\sin\theta\right)hu.
\end{aligned} \tag{5}$$

Periodic boundary conditions are considered in the longitudinal direction, while in the latitudinal direction, the fluxes are set to zero at the poles. The SWE in spherical coordinates can therefore be written as the system of conservation laws:

$$\partial_t(\mathbf{U}) + \partial_\lambda(\mathbf{F}(\mathbf{U})) + \partial_\theta(\mathbf{G}(\mathbf{U})) = \mathbf{S}(\mathbf{U}),$$

where we have defined the conserved quantities, fluxes and sources as:

$$\mathbf{U} = \begin{pmatrix} h\cos\theta \\ hu\cos\theta \\ hv\cos\theta \end{pmatrix}, \quad \mathbf{F}(\mathbf{U}) = \frac{1}{R}\begin{pmatrix} hu \\ hu^2 + \frac{gh^2}{2} \\ huv \end{pmatrix}$$

$$\mathbf{G}(\mathbf{U}) = \frac{\cos\theta}{R}\begin{pmatrix} hv \\ huv \\ hv^2 + \frac{gh^2}{2} \end{pmatrix}, \quad \mathbf{S}(\mathbf{U}) = \begin{pmatrix} 0 \\ \left(f\cos\theta + \frac{u}{R}\sin\theta\right)hv \\ -\frac{gh^2\sin\theta}{2R} - \left(f\cos\theta + \frac{u}{R}\sin\theta\right)hu \end{pmatrix}. \tag{6}$$

We then present an overview of the classical DG method chosen for the demonstration of a DSL implementation. A complete description can be found, among many others, in [15,19]. We consider for simplicity the discretization of a scalar conservation law,

$$\frac{\partial u}{\partial t} + \nabla\cdot\mathbf{f}(u) = s(u), \tag{7}$$

defined on a two-dimensional rectangular domain. This domain is subdivided into K elements, denoted by D^k, for any $k = 1\dots K$. We restrict our attention to conforming structured meshes composed of rectangular elements $D^k = [x_l^k, x_r^k]\times[y_b^k, y_t^k]$. Moreover, to avoid complicating the notation, we introduce the key concepts of the discretization assuming a Cartesian geometry. The use of lat-lon coordinates allows us to easily extend this formulation to the spherical case, taking into account the specific metric factors.

In each element, let V_h^k be the finite-dimensional space of multivariate polynomials up to a given degree p in each spatial dimension:

$$V_h^k = \left\{ v : v = \sum_{i,j=0}^{p} \alpha_{ij} x^i y^j, x\in[x_l^k, x_r^k], y\in[y_b^k, y_t^k] \right\}.$$

As customary in the description of finite element methods, h denotes the typical mesh size.[1] The finite-dimensional space in which we seek the solution is the space of discontinuous polynomials defined as $V_h = \left\{v\in L^2(\Omega) : v|_{D^k}\in V_h^k\right\}$. The numerical solution can be viewed as the direct sum of local approximations:

$$u_h = \bigoplus_{k=1}^{K} u_h^k, \tag{8}$$

where $u_h^k \in V_h^k$. Due to (8), we can restrict our attention to a single mesh element, dropping the superscript k for simplicity when necessary. We define the local residual as

$$R_h^k = \frac{\partial}{\partial t}u_h^k + \nabla\cdot\mathbf{f}(u_h^k) - s(u_h^k),$$

and impose that it vanishes locally in a Galerkin sense, i.e.,

$$\int_{D^k} R_h^k \phi_h^k = 0$$

for any suitably defined test function $\phi_h^k \in V_h^k$. After integration by parts, the weak DG formulation is given by:

$$\int_{D^k}\frac{\partial}{\partial t}u_h^k\phi_h^k + \int_{\partial D^k}\mathbf{f}^*(u_h)\cdot\mathbf{n}_k\phi_h^k - \int_{D^k}\mathbf{f}(u_h^k)\cdot\nabla\phi_h^k = \int_{D^k} s(u_h^k)\phi_h^k. \tag{9}$$

In Equation (9), the physical flux at the element boundary is replaced by a numerical approximation, denoted by $\mathbf{f}^*$. This guarantees that the flux is single-valued at each edge, enforcing conservation across any edge. Note that all terms in Equation (9) are local to the k-th element, except the numerical flux, which depends on the neighboring elements. The choice of $\mathbf{f}^*$ plays a crucial role in the numerical solver's consistency, accuracy and stability. A popular choice of $\mathbf{f}^*$ is the Rusanov flux, defined as:

[1] The mesh size h should not be confused with the fluid height h, since these are never used in the same context.

Table 1
Butcher tableaux of SSP Runge Kutta methods of order 1 to 4.

0 | 1	0 | 1 | 1 1 | 1 | 1 | 1/2 | 1/2	0 | 1 | 1 | 1/2 | 1/4 | 1/4 | | 1/6 | 1/6 | 2/3	0 | 1/2 | 1/2 | 1/2 | 0 | 1/2 | 1 | 0 | 0 | 1 | 1/6 | 1/3 | 1/3 | 1/6

$$\mathbf{f}^*(u_h) = \mathbf{f}^*(u_h^k, u_h^{\bar{k}}) = \frac{\mathbf{f}(u_h^k) + \mathbf{f}(u_h^{\bar{k}})}{2} - \frac{\alpha}{2}(u_h^{\bar{k}} - u_h^k)\mathbf{n}_k, \tag{10}$$

where $\bar{k}$ is the index of the neighbor element to k across a given edge, and $\alpha \geq 0$ is a large enough stabilization parameter, usually chosen to be an estimate of the largest eigenvalue of the hyperbolic system associated to the conservation law. Finally, $\mathbf{n}_k$ is the normal unit vector, pointing outwards on D^k. The local solution u_h^k in element k is then written as a linear combination of a polynomial basis $\phi_i^{(k)}$ of V_h^k:

$$u_h^k = \sum_{j=1}^{n_\phi} \hat{u}_j^k \phi_j^{(k)}, \tag{11}$$

where we have dropped the suffix h in the notation for the polynomial basis. Furthermore, $\hat{u}_i^k$ denotes the polynomial expansion coefficients, and $n_\phi = (p+1)^2$ represents the cardinality of the polynomial basis, where p represents the maximum degree of the polynomials employed. Multiple choices exist for the basis set used for the local polynomial spaces. In this study, following e.g., [35], we employ a modal DG approach, which relies on bivariate Legendre polynomials. After inserting the basis expansion from Equation (11) in Equation (9) and using the $\phi_i^{(k)}$ as test functions, we obtain the following semi-discrete form:

$$M\frac{d\hat{\mathbf{u}}}{dt} = \mathbf{h}(\hat{\mathbf{u}}) \Leftrightarrow \frac{d\hat{\mathbf{u}}}{dt} = M^{-1}\mathbf{h}(\hat{\mathbf{u}}). \tag{12}$$

Here, the vector $\hat{\mathbf{u}}$ collects the polynomial expansion coefficients for all elements, and the matrix M has a block diagonal structure, where the diagonal blocks are the local mass matrices $M^{(k)}$:

$$M_{ij}^{(k)} = \int_{D^k} \phi_i^{(k)} \phi_j^{(k)}$$

associated with each element. Notice that all the terms on the right-hand side have been grouped in the vector function $\mathbf{h}(\hat{\mathbf{u}})$, thus obtaining spatial semi-discretization that can be fully discretized by the method of lines approach described below. Thanks to the use of a DG discretization, the resulting mass matrix can be inverted locally for each element. Furthermore, in the case of spherical coordinates, we also simplify the definition of the conserved variables in Equations (5) by including the $\cos\theta$ metric terms directly in the local mass matrix:

$$M_{ij}^{(k)} = \int_{D^k} \phi_i(\lambda,\theta)\phi_j(\lambda,\theta)\cos(\theta),$$

so that the conserved variables are given by h, hu and hv. Note that the mass matrices are all identical for a specific longitudinal value.

For the time discretization of Equation (12), we follow the classical method of lines approach employing Runge-Kutta (RK) methods. More precisely, we use explicit Strong Stability Preserving (SSP) methods of orders from 1 to 4, denoted later as RK1-RK4, see e.g., [16], which can be defined by means of their Butcher tableaux listed in Table 1. Notice that the numerical experiments reported in Section 4, with minor exceptions, utilize the fourth order RK4 method. These explicit time discretization methods are only conditionally stable. Their stability depends on the value of the non-dimensional parameter known as the Courant number, which is usually defined as $c\Delta t/\mathcal{H}$, where c denotes some estimate of the largest eigenvalue of the underlying hyperbolic system and $\mathcal{H}$ is the minimum grid spacing in physical space. For DG methods and other high-order finite element techniques, however, it is customary to redefine the Courant number by taking into account the presence of internal degrees of freedom in each element, see e.g., [25,26,35], so that a more appropriate definition is in this case $pc\Delta t/\mathcal{H}$, where p denotes the maximum element degree. Due to the reduction of the effective element size at the poles and the use of high-order elements, rather small values of the time step have to be chosen to allow for stable simulations.

3. Implementations in G4GT and GT4Py

We have implemented DG solvers for conservation laws in separate projects with distinct DSLs for planar and spherical geometry. The Galerkin-for-GridTools (G4GT) and GridTools-for-Python (GT4Py) are both part of the GridTools (GT) [30] ecosystem, which offers an efficient platform-agnostic C++ library. It makes extensive use of template meta-programming techniques and has optimized backends for both CPU and GPU architectures. GT was initially designed to target numerical simulations of PDEs using regular grids and finite-difference schemes. Although the latest version of GT also supports unstructured grids, the presented implementation relies on its original version.

We remark that G4GT was simply a proof of concept and is *no longer supported*. Indeed, the popularity of Python among modern-day programmers led the GT team to switch to the Python-based GT4Py layer, which is actively developed and open source. However, the ideas illustrated in G4GT, in terms of both supported PDE models and computational performance, are educational. Thus, before discussing in detail the GT4Py implementation, which should be regarded as the main tool employed, we briefly summarize the main features of the G4GT framework, which GT4Py retains and improves.

3.1. Galerkin for GridTools (G4GT) implementation

G4GT is a C++-based extension to the GridTools library that supports finite element codes. It relies on GT for the underlying implementation of computation kernels, but also on the Trilinos [18] libraries Intrepid [20] and Epetra [13], which provide the numerical support for finite element discretizations and specific linear algebra tools. The G4GT framework provides the link between these libraries, adding a higher-level, user-friendly layer to GT. Additionally, it adds support for finite element discretizations using GT-based codes, which is not present in GT.

As the main subject of this work is the GT4Py implementation, of which G4GT can be viewed as a precursor, to keep the discussion concise we do not delve into the technicalities of the implementation. However, we refer to [11] and [12] for a detailed discussion on the main programming techniques employed. We simply mention that, as it is the case with Section 3.2, the key steps of the discretization are implemented with GT abstractions, allowing the user to write an element-wise code, which is then evaluated in the entire computational domain with a minimal effort.

3.2. GT4Py implementation

We provide now a comprehensive overview of the fundamental concepts and functionalities offered by the GT4Py package. It introduces the specific Python syntax utilized for defining stencil computations. Additionally, we present our implementation of the DG scheme, emphasizing the enhancements we have integrated into the GT4Py framework to support high order discontinuous finite element methods.

3.2.1. Compilation pipeline

The pipeline of GT4Py is illustrated in Fig. 2. The domain scientist expresses the stencils in a user-friendly Python syntax called GTScript, and this code is then processed through a series of toolchains that applies optimizations and generates a high-performance executable targeting a specific architecture.

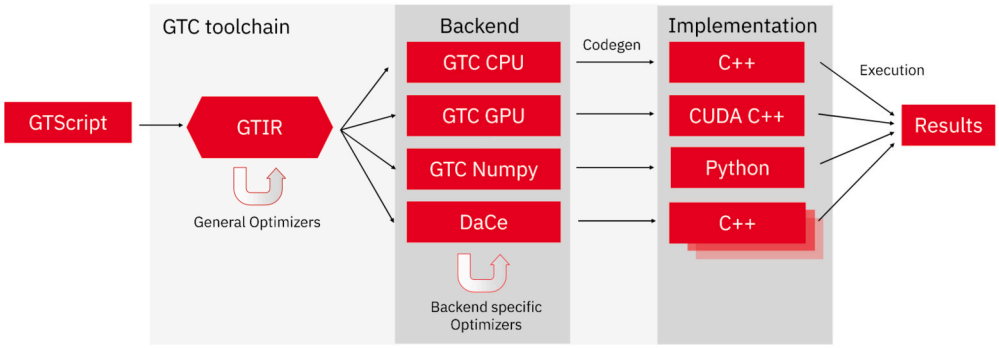


Fig. 2. GT4Py compilation pipeline. Figure thanks to Till Ehrengruber, CSCS.

Table 2
List of supported GT4Py backends.

Framework	Name
GridTools	`gt:cpu_ifirst`
	`gt:cpu_kfirst`
	`gt:gpu`
DaCe	`dace:cpu`
	`dace:gpu`
	`cuda`
	`numpy`

3.2.2. Backends

GT4Py can compile code for various backends; see Table 2 for a complete list of the supported ones at the time of the evaluation. Several others are planned or under development. Three of the seven backends compatible with GT4Py rely on the GT framework to compile and optimize the stencil computations. They are all characterized with the prefix `gt:`. The `gt:cpu_ifirst` and `gt:cpu_kfirst` both target the CPU architecture, while the `gt:gpu` backend produces code for the GPU. In addition, two backends utilize the Data Centric (DaCe) parallel programming framework [6] developed by the Scalable Parallel Computing Lab at Swiss Federal Institute of Technology Zurich, namely `dace:cpu` and `dace:gpu` targeting CPUs and GPUs, respectively. At the time of the DG-GT4Py implementation, only prototype implementations of these backends were available, and thus we decided not to include them in the subsequent performance evaluation.

Alternatively, there is a naive CUDA backend which only utilizes GT utilities, but not its DSL. Finally, a NumPy backend exists, which can be used to inspect the generated code for debugging purposes.

3.2.3. Stencils

Stencils are special GT4Py functions that operate on fields in a specific domain. Fields store the values of variables at each grid point of the domain.

Declaration In the following example, we compute the discretized 2-dimensional Laplacian operator:

$$(\Delta u)_{i,j} = -4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}$$

which can be written as the following stencil in GT4Py:

```
1    import numpy as np
2    import gt4py.gtscript as gtscript
3    @gtscript.stencil(backend=backend, **
         backend_opts)
4    def laplacian(
5      field: gtscript.Field[np.float64],
6      out: gtscript.Field[np.float64]
7    ):
8      with computation(PARALLEL), interval(...):
9        out = - 4 * field + (field[1,0,0] + field
             [-1,0,0]  + field[0,1,0] + field
             [0,-1,0])
```

In the function decorator, we provide the target backend as well as potential back-end options. The function expects fields as arguments, on which the stencil computations are executed. GT4Py uses the Python-type hinting system to specify the data type of each field, which in this case is `np.float64`.

The body of the function requires two context managers that define the execution of the stencil in the vertical direction, the first being `computation` which accepts the arguments `PARALLEL`, `FORWARD` or `BACKWARD`. This defines the scheduling of the execution stencil. The keyword `PARALLEL`, which we use exclusively for our implementation, indicates that there is no dependency between subsequent vertical levels, and hence they can all be solved concurrently. The keywords `FORWARD` and `BACKWARD` define this dependency and indicate the direction in which the vertical levels must be solved. The second context manager `interval` allows the user to specify the vertical indices for which the stencil will be applied. The '`...`' is a shorthand notation to select the entire vertical domain.

Finally, we note that the stencil computation is applied for each grid point; hence, relative offsets are used as indices. Note that, if omitted, the offset is assumed to be `[0,0,0]`.

Invocation The above `laplacian` function can be called using the following command:

```
1    nx, ny, nz = field.shape
2    origins = {"field":(1,1,0),"out":(1,1,0)} # or
         {"_all_":(1,1,0)}
3    laplacian(field, out, origin=origins, domain=(
         nx-2, ny-2, nz))
```

We provide the fields relevant to the stencil computation as arguments to the function. In addition, we add two optional keyword arguments, namely `domain`, which specifies the domain of execution of the stencil, and `origin`, which defines the origin for each field. In the case of the `laplacian` stencil, we set these to ensure that the stencil only operates on the inner part of the domain.

The `origin` argument indicates relative offset between the different fields. The keyword `_all_` can be utilized to set the same origin for all fields that have not been specified separately. Note that the keyword names inside the `origins` dictionary refer to the names of the fields in the stencil definition and not to the names of the fields in the call to the stencil. Upon invocation of a stencil, GT4Py searches for a cached version and relies on just-in-time (JIT) compilation in case none is found.

3.2.4. Storages

In GT4Py, fields are variables on which stencils can be applied. The DSL provides a storage format for these fields which is a wrapper over the array types `numpy/cupy.ndarrays` called `gt4py.storages`, which ensures that the memory layout of the data is compatible with the requested backend. The interface provides several methods for instantiating storages, including `empty()`, `ones()` and `zeros()`, as well

as directly from an existing NumPy array using `from_array()`. All of these functions require several additional parameters: `shape` defines the size of the storage in the three dimensions, and `default_origin` specifies the default origin to be used in case none is specified during a stencil call. Finally, `dtype` not only defines the data type of the field but can also be used to assign higher-order tensors to each grid point instead of simple scalar values. These fields are subsequently referred to as higher-dimensional fields.

In the example below, each grid point stores a matrix of size 3x2:

```
u = gt4py.storage.zeros(
      backend=backend, default_origin=(1,1,0),
      shape=(4, 4, 2), dtype=(np.float64, (3,2))
    )
```

Moreover, suppose a field has identical values along one or more spatial dimensions. In that case, GT4Py provides a feature called 'masking', which avoids the storage of unnecessary copies of the identical values while still giving the appearance of a full 3-dimensional field. This can lead to a substantial reduction in memory consumption, which is crucial for large problem sizes. The previous field can be masked in the vertical direction using:

```
u = gt4py.storage.zeros(
      backend=backend, default_origin=(1,1),
      shape=(4, 4), dtype=(np.float64, (3,2)),
      mask=[True, True, False]
    )
```

Note that when using a GPU backend, the fields need to be explicitly synchronized from the device back to the host to obtain the results of a stencil computation. In addition, it is recommended to cast the `gt.storage` to a `numpy.ndarray` to ensure that the data has indeed been copied from the device. This can be accomplished with the following code snippet:

```
x_gt.device_to_host()
x_np = np.asarray(x_gt)
```

3.2.5. Frontend

This section describes the structure of the GT4Py frontend and our contribution to expanding the functionality vector-valued fields.

Abstract syntax tree (AST) The Python language uses an interpreter which converts the source code of a program into a representation called an Abstract Syntax Tree (AST) before compiling the program to bytecode which is executed by the computer. As the name suggests, the AST represents the logic of the program as a tree structure stripped of the specific syntax used in the source code. This representation captures the semantics of the program and provides a convenient way to inspect and modify Python applications.

In the case of GT4Py, the frontend parses the Python AST and converts it into a series of custom ASTs through the pipeline (Fig. 2), which provide additional information necessary for the backends to produce well-optimized executables.

Limited support for higher-dimensional fields For our implementation, we represent each DG element by a grid point in GT4Py. Each grid point is thus assigned a vector that stores its polynomial expansion coefficients. We refer to this additional dimension of the field as `data_dims`.

Initially, the support for these vector-valued fields in GT4Py was limited. In particular, there was no functionality for performing element-wise operations between fields with respect to the `data_dims` dimension. Indeed, these operations needed to be explicitly written out for each vector component, reducing their utility to scalar fields. The following example illustrates a stencil performing an element-wise multiplication between two higher-dimensional fields:

```
@gtscript.stencil(backend=backend)
def mult(
  field1: gtscript.Field[(np.float64, (3,))],
  field2: gtscript.Field[(np.float64, (3,))],
  out: gtscript.Field[(np.float64, (3,))]
):
  with computation(PARALLEL), interval(...):
    out[0,0,0][0] = field1[0,0,0][0] * field2
          [0,0,0][0]
    out[0,0,0][1] = field1[0,0,0][1] * field2
          [0,0,0][1]
    out[0,0,0][2] = field1[0,0,0][2] * field2
          [0,0,0][2]
```

The first set of indices represents the relative offsets between the fields, while the second set of indices refers to the actual components of the `data_dims` dimension. Note that in this case, the relative offsets cannot be omitted and need to be specified explicitly. Clearly, utilizing this syntax for implementing a DG scheme would be impractical.

Element-wise operation We have implemented element-wise operations for vector-valued fields in order to facilitate their use in GT4Py. The goal is to modify the frontend such that the following convenient syntax is valid:

```
# ...
with computation(PARALLEL), interval(...):
  out = field1 * field2
```

This was accomplished by adapting the GT4Py internal processing of AST nodes such that, when vector-valued fields appear in expressions, their AST nodes are converted to the ones produced by the previous explicitly unrolled code snippet. The implemented system verifies the compatibility of dimensions for vector-valued fields.

Our implementation supports not only chaining together multiple operations on higher-dimensional fields but also broadcasting of scalar values for scalar-vector operations. The syntax and functionality should be intuitive for anyone familiar with the NumPy package.

Matrix multiplication An additional operation that we required for our DG scheme was a matrix-vector multiplication between higher-dimensional fields. This was incorporated into our existing framework and can be invoked using the "`@`" operator. Also, the multiplication of a vector by the transpose of a matrix can be achieved by appending the matrix with the "`T`" attribute. This leads to the following syntax:

```
@gtscript.stencil(backend=backend)
def matmul(
  matrix: gtscript.Field[(np.float64, (3, 2))],
  vec: gtscript.Field[(np.float64, (3,))],
  out: gtscript.Field[(np.float64, (2,))]
):
  with computation(PARALLEL), interval(...):
    out = matrix.T @ vec
```

DG solver: precomputation At the start of the execution of the program, the GT4Py solver precomputes on the CPU certain variables that remain constant during the whole simulation. This includes the computation of the inverse mass matrix, as well as the Gauss-Legendre quadrature points and weights for numerical integration. A helper class called `vander`, defined in `vander.py`, contains all the Vandermonde matrices required to evaluate the polynomials stored as modal expansion coefficients at nodal values in the domain (see e.g., [19]). These matrices are instantiated as fields using `gt4py.storages`.

DG solver: stencils All subsequent computations are carried out using stencils in GT4Py. An example stencil is presented subsequently, related to our DG solver. Applying the theory derived in Section 2 for the linear, constant-coefficient advection problem

$$\frac{\partial u}{\partial t} + \nabla \cdot (\boldsymbol{\beta} u) = 0, \tag{13}$$

with e.g., $\boldsymbol{\beta} = [\beta_1, \beta_2]^T = [1, 1]^T$, we will need to evaluate an integral of the following form:

$$\int_{D^k} \left[\beta_1 u \frac{\partial \phi}{\partial x} + \beta_2 u \frac{\partial \phi}{\partial y} \right] dx dy.$$

This integral can be computed using the stencil below:

```
1    #...
2    with computation(PARALLEL), interval(...):
3      u_qp = phi @ u_modal
4      fx = u_qp * 1
5      fy = u_qp * 1
6      rhs = determ *
7        (phi_grad_x.T @ (fx * w) / bd_det_x
8         + phi_grad_y.T @ (fy * w) / bd_det_y)
```

Note how the provided code snippet concisely articulates the mathematical expression with minimal boilerplate. In line 3, the modal expansion coefficients are mapped to nodal values at the quadrature points. In lines 4 and 5, the flux function in the x and y directions is applied. In this simple case, the flux function is the identity due to the constant velocity field $\boldsymbol{\beta} = [1, 1]^T$. Finally, in lines 6 — 8, the numerical integration is performed. The scalar field `w` represents the quadrature weights while the matrix-valued field `phi_grad_x/y` contains the spatial derivatives of the basis functions. The terms `determ`, `bd_det_x/y` denote the Jacobians arising from the mapping of the element in physical space onto a reference element. Although not reported here, this description easily generalizes to the SWE case.

4. Code validation

Although G4GT has been discontinued, we retain the results of the G4GT implementation, because different aspects are emphasized with respect to the GT4Py implementation. Firstly, several unit tests have been performed to assess the correctness of our implementations. For the GT4Py implementation, the tests rely on this framework's existing testing infrastructure, which verifies the success of the code generation as well as the code execution on all backends when compared with a reference Numpy implementation.

Subsequently, both the G4GT and GT4Py implementations were validated on benchmarks derived from the shallow water test suite [40], as well as on tests on a planar geometry presented in [36]. These include a convergence test on linear advection of a smooth profile and on a geostrophic zonal flow, the simulation of geostrophic adjustment on the plane, and that of a Rossby-Haurwitz wave in spherical geometry. Although our final goal is the simulation of the SWE on the sphere, all the presented tests provide useful insight from a numerical point of view and contribute to a progressive increase in the complexity of the solutions.

4.1. Linear advection convergence of a smooth initial condition

Since both implementations are essentially solving the same problem, albeit with slightly different flux calculations and numerical implementations, we summarize the convergence results for both in this section. Specifically, we apply our DG discretization to planar linear advection on the unit square, assuming periodic boundary conditions and a constant velocity field $\boldsymbol{\beta} = [1, 1]^T$, see Equation (13). We consider a smooth initial condition: $u_0(x, y) = \sin(2\pi x)\sin(2\pi y)$, which allows us to achieve optimal convergence rates. The analytic solution of Equation (13) evolves without changing shape in the direction of the velocity field. Due to the periodic boundary conditions, the solution will coincide with the initial condition after one full rotation, i.e., at time $T = 1$. We use a uniform mesh with K elements obtained from the tensor product of $\sqrt{K}$ elements in each of the coordinate directions. We measure the error of the numerical approximation using the L^2 norm at the final simulation time, and we denote it by ϵ. The expected spatial convergence order for DG methods is given by [19]:

$$\epsilon \sim O(h^{p+1}), \tag{14}$$

Table 3
L^2 errors ϵ and estimated rate of convergence r for the linear advection problem, G4GT implementation.

K	$p=1$		$p=2$		$p=3$	
	ϵ	r	ϵ	r	ϵ	r
10^2	1.343e−2	-	1.050e−3	-	3.780e−5	-
20^2	3.369e−3	**2.00**	1.329e−4	**2.98**	2.030e−6	**4.22**
40^2	8.405e−4	**2.00**	1.666e−5	**3.00**	1.302e−7	**3.96**
80^2	2.099e−4	**2.00**	2.084e−6	**3.00**	8.345e−9	**3.96**
160^2	5.246e−5	**2.00**	2.611e−7	**3.00**		

Table 4
L^2 errors ϵ and estimated rate of convergence r for the linear advection problem, GT4Py implementation.

K	$p=1$		$p=2$		$p=3$	
	ϵ	r	ϵ	r	ϵ	r
20^2	4.204e-3	-	1.330e-4	-	2.061e-6	-
40^2	9.004e-4	**2.22**	1.666e-5	**2.99**	1.288e-7	**4.00**
80^2	2.139e-4	**2.07**	2.084e-6	**2.99**	8.049e-9	**4.00**
160^2	5.212e-5	**2.02**	2.606e-7	**3.00**	5.030e-10	**4.00**

where h is the characteristic mesh size and p is the degree of the local polynomials. To estimate the convergence rate, we compute the discretization error using two different meshes with characteristic sizes $h_1,\ h_2$, that we denote as ϵ_1, ϵ_2, respectively. Then, the estimated rate, denoted by r, is computed as

$$r = \frac{\log(\epsilon_1) - \log(\epsilon_2)}{\log(h_1) - \log(h_2)}. \tag{15}$$

The results obtained with the G4GT implementation are reported in Table 3 and agree with the theoretical expectations, thus validating the implementation. Not surprisingly, the GT4Py implementation achieves nearly identical convergence results to the G4GT version, as shown in Table 4. Notice that the results in Table 3 have been obtained using for all spatial discretizations the RK4 method for the time discretization, while those in Table 4 were obtained using for each p a RK method of the same order for the time discretization.

4.2. G4GT implementation: geostrophic adjustment for planar SWE

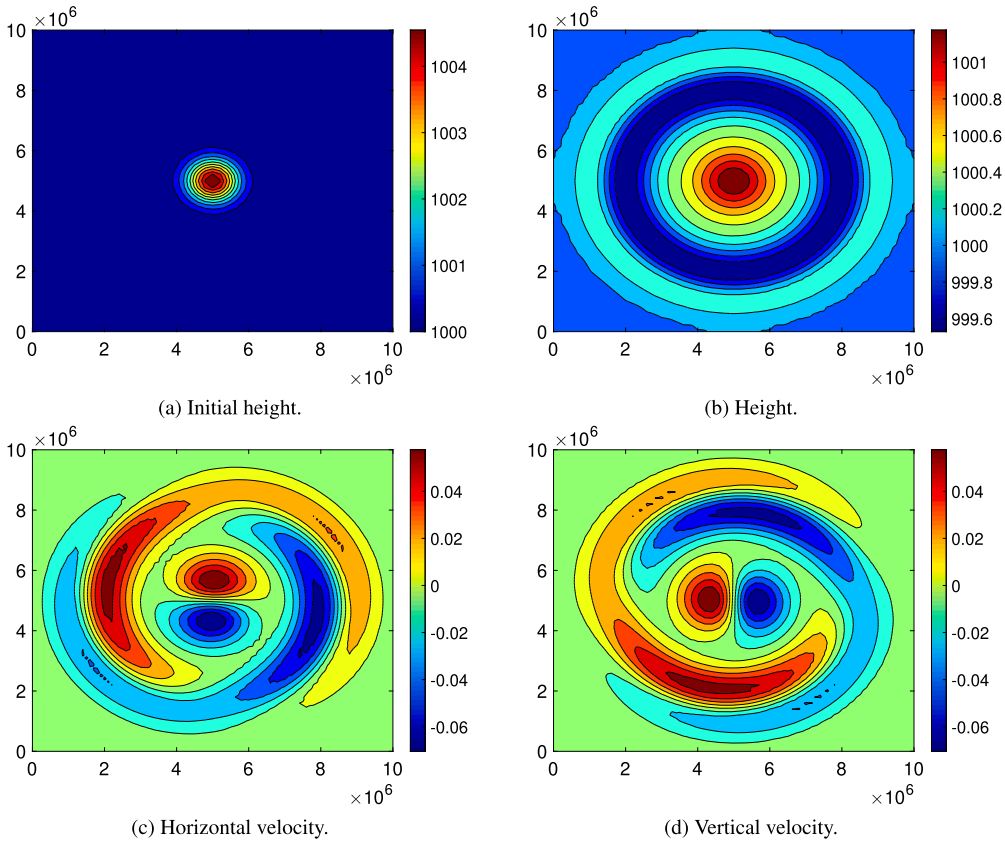
For the G4GT implementation, we consider the SWE discretized on a Cartesian mesh in a planar domain. Specifically, we want to show that the scheme is able to reproduce the geostrophic adjustment process, see, for example, the discussion in [36]. Starting from a perturbation of the equilibrium state corresponding to a constant water height, gravitational and rotational forces interact, so that only part of the energy is transported away from the center, leading to a nontrivial stationary solution profile. Consider a square domain $\Omega = [0, L]^2$ with $L = 10^7$ m and a final time of $T = 36000$ s. The initial velocities and momenta are set to zero, while the height h is equal to

$$h = h_0 + h_1 \exp\left(-\frac{(x - L/2)^2 + (y - L/2)^2}{2\sigma^2} \right), \tag{16}$$

where $h_0 = 1000$ m, $h_1 = 5$ m and $\sigma = L/20$ m. Assuming an f-plane approximation, the Coriolis parameter f is chosen to be constant and equal to 10^{-4} s^{-1}. The problem is completed with periodic boundary conditions. The simulation has been run using 50×50 spatial elements, a polynomial degree $p = 3$ and the RK4 scheme in time with step $\Delta t = 100$ s, corresponding to a maximum Courant number of approximately 0.15. The results are reported in Fig. 3. The solution is consistent with the results reported in [36].

4.3. GT4Py implementation: geostrophic zonal flow for SWE on the sphere

After validating the planar version of the GT4Py implementation, we consider two of the classical test cases in spherical geometry introduced in [40] for the shallow water equations. Periodic boundary conditions



(a) Initial height. (b) Height.

(c) Horizontal velocity. (d) Vertical velocity.

Fig. 3. Numerical results for the geostrophic adjustment test case. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 5
L^2 errors ϵ and estimated rate of convergence r for the Williamson test case 2, GT4Py implementation.

K	$p=1$		$p=2$		$p=3$	
	ϵ	r	ϵ	r	ϵ	r
10^2	9.366e−2	-	1.020e−3	-	6.864e−5	-
20^2	1.984e−3	**5.56**	1.085e−4	**3.23**	3.951e−6	**4.08**
40^2	4.508e−4	**2.13**	1.490e−5	**2.86**	2.362e−7	**4.06**
80^2	1.111e−4	**2.02**	1.986e−6	**2.90**	1.471e−8	**4.00**

were applied in the longitudinal direction, while in the latitudinal direction the fluxes were set to zero. Indeed, since the edges become singular at the poles, the flux through them must be zero.

In the benchmark denoted as test case 2 in [40], a stationary zonal flow in geostrophic equilibrium is considered. We perform a convergence test for the spatial discretization, using for all polynomial degrees the RK4 scheme for time discretization with time steps chosen for each resolution so as to keep the Courant number sufficiently small and bounded by a quantity of the order 10^{-2}. The test case has been run until time $T = 2$ days on meshes of increasing resolutions. The results are reported in Table 5 and show a convergence behavior entirely analogous to that of the linear advection case in planar geometry.

4.4. GT4Py implementation: Rossby-Haurwitz wave for SWE on the sphere

The Rossby-Haurwitz wave (denoted as test case 6 in [40]) consists of a large-scale planetary wave that mimics the high/low-pressure systems typical of mid-latitude weather patterns. The test case considers initial data that would result in a stable solution for the barotropic vorticity equation, evolving from west to east without changing shape. It is known that this configuration is ultimately unstable — see, for example, the discussion in [33] — but this instability only arises on a relatively long time scale. Therefore, it is customary to assess the quality of numerical methods based on their capability to reproduce a stable eastward moving pattern for several days. In Fig. 4, we see the results of an 8-day simulation of the Rossby-Haurwitz wave on a 40x20 grid using the RK4 method in time with $\Delta t = 4$ s and $p = 3$ in space. It can be observed that the numerical solution indeed evolves from west to east while maintaining a close resemblance with the initial shape and that the simulated pattern is in good agreement with reference solutions, see, e.g., [35].

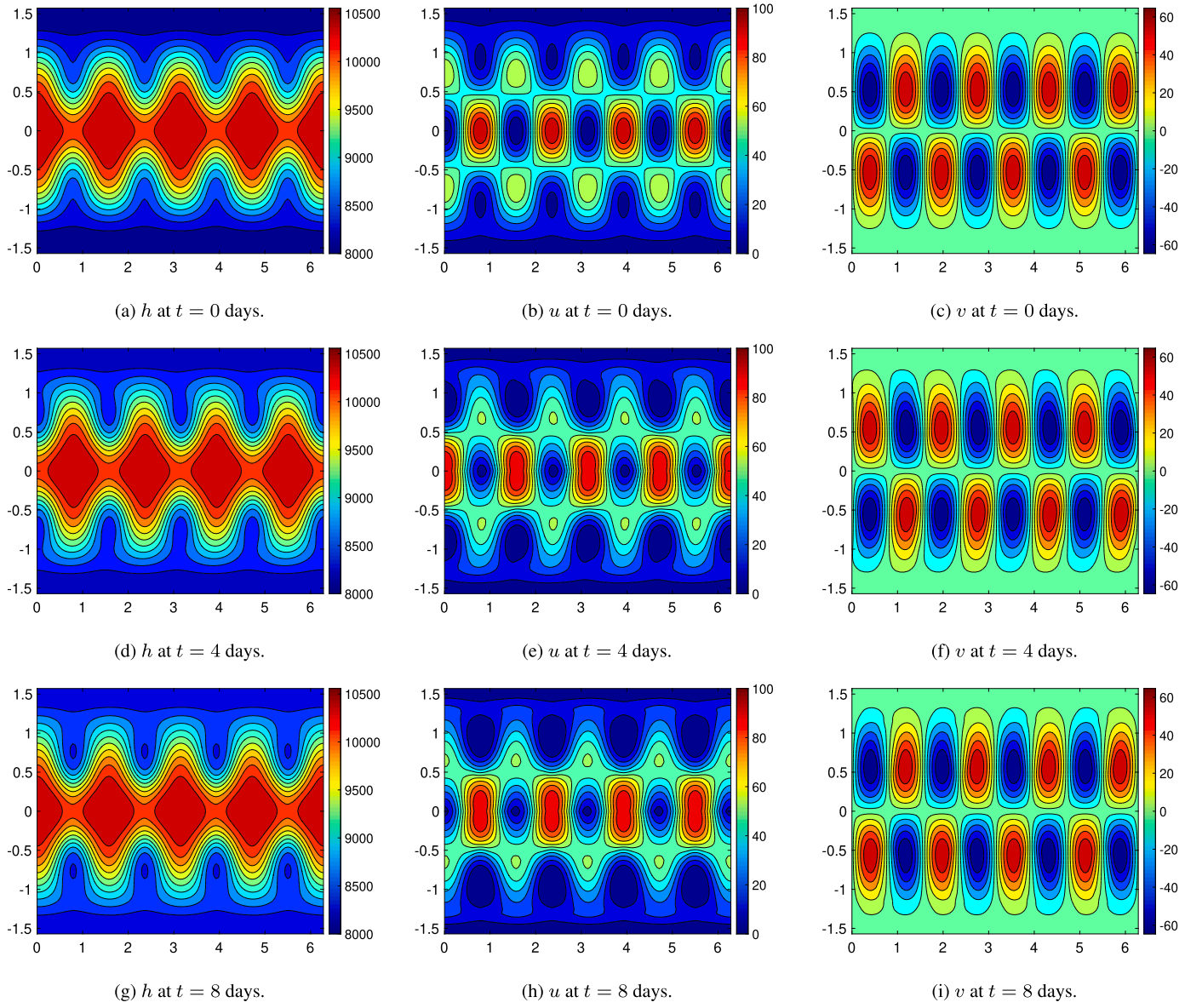
5. Performance

In this section, we present performance benchmarks of the G4GT and GT4Py implementations for the SWE in both planar and spherical geometry. In both cases, the time spent in the precomputation steps is neglected, as it becomes negligible for long simulation periods.

The G4GT simulations have been run on the compute nodes of *Piz Daint* at CSCS, using an Intel® Xeon® E5-2690 v3, 12-core processor (single node), characterized by a peak memory bandwidth of 68 GB/s. On the other hand, the GT4Py benchmarks were performed on a different partition of *Piz Daint* with the CPU code executed on two 18-core Intel® Xeon® E5-2695 v4 @ 2.10 GHz processor (each with 77 GB/s peak memory bandwidth) and the GPU code on an NVIDIA® Tesla® P100 with 16 GB of memory (540 GB/s peak memory bandwidth).

5.1. G4GT performance evaluation

The geostrophic adjustment setup presented in the previous section can also be used to evaluate the performance of the method and G4GT in general. Unless stated otherwise, the physical and numerical parameters are therefore kept unchanged, including a grid consisting of 50×50 elements and a time step of 100 s.



(a) h at $t = 0$ days. (b) u at $t = 0$ days. (c) v at $t = 0$ days.

(d) h at $t = 4$ days. (e) u at $t = 4$ days. (f) v at $t = 4$ days.

(g) h at $t = 8$ days. (h) u at $t = 8$ days. (i) v at $t = 8$ days.

Fig. 4. 8-day simulation of the Rossby-Haurwitz wave.

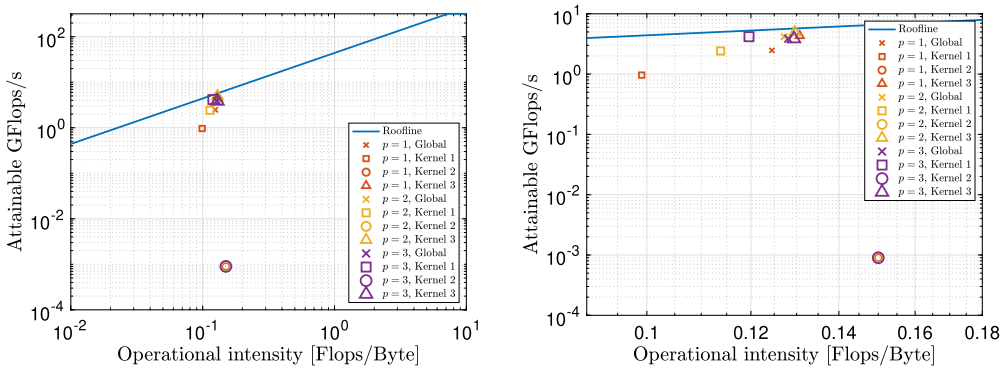
The performance evaluation is done using the Roofline model [39]. This is based on the *operational intensity*, i.e., the number of floating-point operations (flops) per byte of DRAM traffic, and the *attainable Gflops per second*, i.e., the concrete performance measure. Here, the DRAM traffic takes into account the bytes that are read from/written in the main memory after the filter of the cache hierarchy. Because of hardware limits, the attainable flops per second cannot go beyond a fixed threshold, determined by the peak memory bandwidth and the peak floating point performance. In practice, the actual threshold is determined by running benchmark cases, such as the (bandwidth-limited) *stream* or the (computationally-limited) *LINPACK* benchmark. In our case, these give a memory bandwith upper bound of 44 GB/s and a peak performance limit of 318 GFlops/s [9]. Thus, for a given operational intensity, an efficient implementation in terms of performance should attain values close to the determined limit. In our analysis, we decided to ignore the cache effects. In other words, every access to a variable is considered for the computation of the required bytes. This is in contrast with the definition provided by the model, but a precise estimate of the DRAM traffic is far from an easy task and, in the G4GT framework, no tool is available to appropriately measure it. The matrix-vector multiplication, which is the central operation in the DG implementation, is bandwidth-limited and achieves a performance [11] somewhat below the leftmost (rising) roofline.

Based on the way in which the code is structured (see [11]), we can recognize three different kernels:

1. **Common part:** The nodal values for the solution and the flux function are computed.
2. **Rusanov fluxes:** The boundary fluxes are computed, and the boundary conditions are applied. This requires communication among neighboring elements.
3. **Main computation:** The right-hand side is assembled, and the solution is updated.

The results for varying polynomial degree p are reported in Fig. 5, which compares the performances of the global program and the kernels separately. As a complement to Fig. 5, Table 6 breaks down the computational times.

Looking at the overall performance, we observe that no significant variations in the operational intensities are present. This is because variations in the polynomial order lead to similar changes in the number of



(a) Global range (b) Zoom into the interval $[0.09, 0.18]$

Fig. 5. Performance evaluation of the G4GT implementation of the SWEs in a Cartesian geometry with different spatial degrees p, RK4 scheme.

Table 6
Computational times of the G4GT implementation of the SWEs in a Cartesian geometry with different spatial degrees p, RK4 scheme.

p	Global [s]	Kernel 1 [s]	Kernel 2 [s]	Kernel 3 [s]
1	15.17	7.42	1.45	6.31
2	78.98	21.22	3.10	54.66
3	430.21	54.97	5.81	369.43

floating point operations and memory traffic. However, it appears that performances obtained with linear basis functions are slightly lower than higher-order polynomials. The total number of operations might not be large enough to attain the expected asymptotic values, causing deviations from the optimal performance.

Looking at the kernels independently, for high values of p the third kernel has the most significant influence on the overall performance. This is expected since it includes the majority of the computations. Specifically, the assembly of the internal integral is the most intensive part, both in terms of resources and time. On the other hand, the second kernel always has a low computational cost. This is not surprising, as only boundary quantities are involved. The performance of this kernel in terms of floating point operations per second is consistently low and do not vary with p. Since it is the only phase that involves exchanges between neighboring elements, it is reasonable that cache misses or inefficient memory accesses are present. No particular trend is observed for the first kernel, except for $p = 1$, in which this kernel has the dominant effect on global performance.

5.2. GT4Py performance evaluation

For the GT4Py implementation, we consider both performance scalability while increasing the horizontal problem size as well as increasing the number vertical layers while holding the horizontal size constant.

5.2.1. Horizontal scaling

We study the scaling of the runtime of the application with increasing horizontal resolution. In the benchmark, we use a 4th-order scheme in space which corresponds to a vector of size 16 stored at each grid point (`data_dims = 16`). Fig. 6 compares the runtimes of the various backends. The `gt:cpu_ifirst` backend was used as the baseline, since this is the best performing CPU backend.

Each successive data point doubles the number of grid points in both horizontal directions. Thus, the expected asymptotic scaling is quadratic since doubling the number of grid points in both horizontal directions results in a four-fold increase in the total number of grid points.

In this experiment, the two CPU backends, powered by the GridTools framework, have virtually identical execution times. For small problem sizes, the two GPU backends (namely the `cuda` and `gt:gpu`) perform worse than the CPU backends. This is due to the under-utilization of

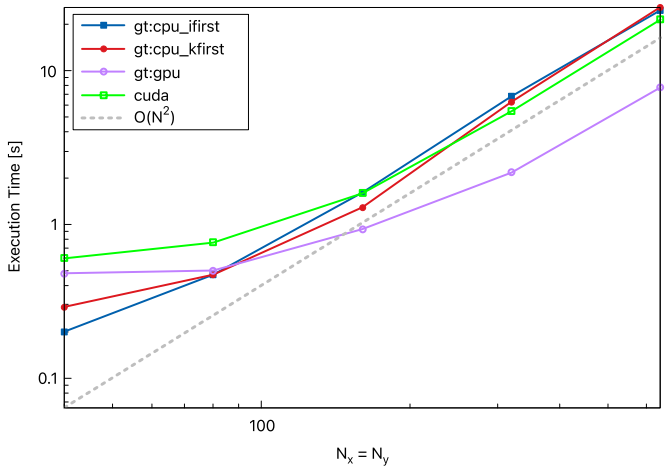


Fig. 6. Benchmark of the execution time of the GT4Py backends with increasing problem size. Each subsequent data point doubles the grid points in x and y and thus quadruples the total number of grid points.

Table 7
Speedup of GT4Py backends vs reference `gt:cpu_ifirst` implementation on 640x640 grid.

	`gt:cpu_kfirst`	`gt:cpu_ifirst`	`cuda`	`gt:gpu`
Speedup Factor	0.952	1.00	1.15	3.19

the GPU resources for small-scale problems which are not able to fully saturate the device. This is illustrated in the delayed asymptotic scaling of the GPU code when compared to the CPU resulting in better performance for larger problems. When comparing the two GPU backends, we observe that the optimizations provided by the GridTools framework (included in the `gt:gpu` backend) yield significant better performing code than the naive CUDA backend. Note that we were limited to presenting a maximum problem size of 640x640 due to memory constraints on the GPU.

Table 7 summarizes the speedup observed versus the `gt:cpu_ifirst` baseline on the largest problem size. One observation from Table 7 is that the fastest GPU backend results in a speedup factor of ~ 3.2 versus the fastest CPU backend. Since we know from Fig. 5a that the performance is limited by memory bandwidth, we expect the speedup to mirror the ratio of bandwidths listed in Section 5 for the P100 GPU to two Intel Broadwell processors, namely, 540 GB/s: 2x77 GB/s $\simeq 3.6$.

5.2.2. Vertical scaling

In this Section we try to assess the potential performance of GT4Py on a 3-dimensional problem by considering a set of decoupled 2-

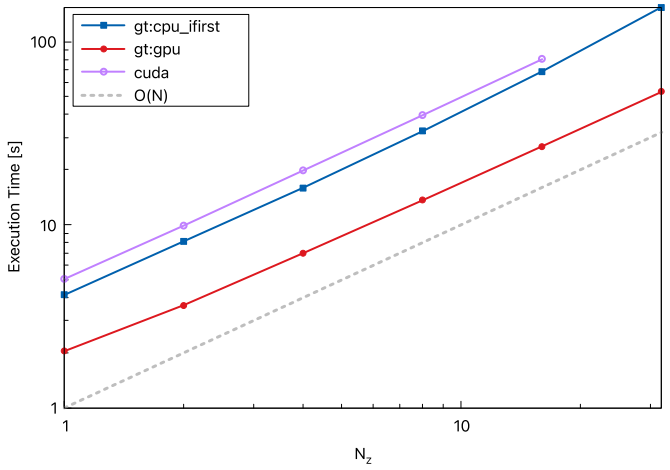


Fig. 7. Benchmark of best-performing CPU and GPU backends in addition to the CUDA backend. The plot depicts execution time with respect to the number of identical vertical problems solved in parallel. The final data point for the CUDA backend is unavailable due to the memory limit reached on GPU.

dimensional SWE problem copied in the vertical direction and solved in parallel. This configuration increases the computational load and resembles to some extent those of low order finite difference/finite volume discretizations of 3-dimensional problems in atmospheric modeling. However, it is substantially different from a full 3-dimensional DG discretization, since all the local matrices that arise correspond to 2-dimensional rather than 3-dimensional elements. The resulting algorithm scales linearly with the number of vertical levels. Considering that all levels can be solved independently, this problem is embarrassingly parallel, and we might expect performance benefits on the GPU compared to the CPU.

Fig. 7 illustrates the execution time of a 4th-order DG scheme on a 300×300 grid with increasing vertical levels. Surprisingly, we do not observe any scaling benefits for this experiment on the GPU. Indeed, all backends exhibit asymptotic scaling from the first data point, indicating that the hardware's resources are fully saturated. This is most likely due to the 2-dimensional problem solved in each level being sufficiently large and fully occupying the memory bandwidth of the GPU. We observe that the CUDA backend performs even worse than the GridTools CPU backend. Moreover, it suffers from poor memory utilization compared to the GridTools GPU implementation, as the last data point could not be gathered due to the memory capacity of the GPU being reached.

6. Conclusions

We have presented two implementation examples of a high-order Discontinuous Galerkin method in the framework of the G4GT and GT4Py Domain-Specific Languages, respectively. After summarizing the main novelties of both the implementations, we validated them using the shallow water equations in both a Cartesian and a spherical geometry and evaluated their computational performance, with a focus on GPUs.

Despite being only a proof of concept, the now obsolete G4GT DSL presented several advantages for the end user over the original GridTools framework, including higher levels of abstraction and thus improved productivity for the application developer. Several technical reasons, including dependencies on external libraries, as well as the emergence of Python as a programming language, were responsible for the termination of the development of G4GT and the migration to GT4Py.

The GT4Py extension discussed in this paper enhances its capabilities to accommodate higher-dimensional fields and enables the implementation of Discontinuous Galerkin schemes in spherical geometry, while leveraging the existing GridTools code-generation framework. Indeed, without modification of the source code, our model could seamlessly operate at high performance on both CPUs and GPUs. To the best of our knowledge, GT4Py stands out as the first GPU-enabled DSL supporting DG solvers in complex geometries. For climate scientists, our framework is thus particularly appealing due to its versatility in supporting diverse discretization schemes (Finite Difference, Finite Volume and now Discontinuous Galerkin) with a platform-portable DSL capable of harnessing accelerators. This, coupled with its intuitive Python syntax, significantly amplifies productivity in computational modeling endeavors. In conclusion, we have shown that GT4Py can play a pioneering role in the application of GPU-accelerated DSLs for DG schemes and pave the way for broader adoption of DSL-driven development in the context of climate modeling.

In future GT4Py work the support for high-dimensional fields could be extended further. Indeed, currently the DSL does not support array slicing, which would allow to group related variables into large matrices instead of a series of separate vectors. Furthermore, in order to eliminate code duplication, functions currently operating exclusively on scalar fields could be extended to higher-dimensional fields. Lastly, the inclusion of a dedicated optimization pass in the backend could be used to fully exploit the unique data memory layout of these fields.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

The Swiss National Supercomputing Centre (CSCS) funded the four-month internships of the first two authors, N.D. and K.S.. We thank the reviewers for their constructive comments, which helped improve the quality of the paper. We would also like to thank Linus Groner, Till Ehrengruber, Enrique González Paredes, Mauro Bianco and Christopher Bignamini of CSCS for their gracious support during both internships. L.B. was partially supported by the ESCAPE-2 project, European Union's Horizon 2020 Research and Innovation Programme (Grant Agreement No. 800897).

References

[1] A. Abbà, L. Bonaventura, M. Nini, M. Restelli, Dynamic models for large eddy simulation of compressible flows with a high order DG method, Comput. Fluids 122 (2015) 209–222.
[2] A. Afanasyev, M. Bianco, L. Mosimann, C. Osuna, F. Thaler, H. Vogt, O. Fuhrer, J. VandeVondele, T.C. Schulthess, GridTools: a framework for portable weather and climate applications, SoftwareX 15 (2021) 100707.
[3] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.R. Rognes, G.N. Wells, The FEniCS project version 1.5, Arch. Numer. Softw. 3 (100) (2015).
[4] M. Baldauf, A horizontally explicit, vertically implicit (HEVI) discontinuous Galerkin scheme for the 2-dimensional Euler and Navier-Stokes equations using terrain-following coordinates, J. Comput. Phys. 446 (2021) 110635.
[5] T. Bandikova, Extreme scale computing and data platform for cloud-resolving weather and climate modeling, https://exclaim.ethz.ch/publications/exclaim-publications.html, 2022, EXCLAIM Brochure.
[6] T. Ben-Nun, J. de Fine Licht, A.N. Ziogas, T. Schneider, T. Hoefler, Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, New York, NY, USA, Association for Computing Machinery, ISBN 978-1-4503-6229-0, November 2019, pp. 1–14.
[7] T. Ben-Nun, L. Groner, F. Deconinck, T. Wicky, E. Davis, J. Dahm, O.D. Elbert, R. George, J. McGibbon, L. Trümper, et al., Productive performance engineering for weather and climate modeling with python, in: SC22: International Conference

for High Performance Computing, Networking, Storage and Analysis, IEEE, 2022, pp. 1–14.
[8] L. Bonaventura, R. Redler, R. Budich, Earth System Modelling 2: Algorithms, Code Infrastructure and Optimisation, Springer Verlag, New York, 2012.
[9] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, Raffaele Tripiccione, Thunderx2 performance and energy-efficiency for HPC workloads, Computation (ISSN 2079-3197) 8 (1) (2020), https://doi.org/10.3390/computation8010020, https://www.mdpi.com/2079-3197/8/1/20.
[10] J. Dahm, E. Davis, T. Wicky, M. Cheeseman, O. Elbert, R. George, J.J. McGibbon, L. Groner, E. Paredes, O. Fuhrer, GT4Py: Python tool for implementing finite-difference computations for weather and climate, in: 101st American Meteorological Society Annual Meeting, AMS, 2021, https://ams.confex.com/ams/101ANNUAL/meetingapp.cgi/Paper/381653.
[11] N. Discacciati, Implementation and evaluation of Discontinuous Galerkin methods using Galerkin4GridTools, https://github.com/nickdisca/DG_code/blob/master/G4GT_reports/report_PACS.pdf, 2018, Master project at the Politecnico di Milano.
[12] N. Discacciati, Implementation and evaluation of Discontinuous Galerkin methods using Galerkin4GridTools, https://github.com/nickdisca/DG_code/blob/master/G4GT_reports/G4GT_report_20190216.pdf, 2019, Small Development Project at the Swiss National Supercomputing Centre.
[13] The Epetra Project Team, The Epetra Project Website, https://trilinos.github.io/epetra.html, 2022. (Accessed 6 December 2022).
[14] M. Fowler, Domain-Specific Languages, Pearson Education, 2010.
[15] F.X. Giraldo, An Introduction to Element-Based Galerkin Methods on Tensor-Product Bases, Springer Nature, 2020.
[16] S. Gottlieb, D. Ketcheson, C.-W. Shu, Strong Stability Preserving Runge-Kutta and Multistep Time Discretizations, World Scientific, Singapore, 2011.
[17] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, T.C. Schulthess, STELLA: a domain-specific tool for structured grid methods in weather and climate models, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–12.
[18] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, K.R. Long, R.P. Pawlowski, E.T. Phipps, et al., An overview of the Trilinos project, ACM Trans. Math. Softw. 31 (2005) 397–423.
[19] J.S. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods, Texts in Applied Mathematics, vol. 54, Springer, New York, 2008.
[20] The Intrepid Project Team, The Intrepid Project Website, https://trilinos.github.io/intrepid.html, 2022. (Accessed 6 December 2022).
[21] I. Kavcic, LFRic and PSyclone: utilising DSLs for performance portability, in: AGU Fall Meeting Abstracts, vol. 2020, 2020, pp. A023–07.
[22] J. Kunkel, N. Jumah, A. Novikova, T. Ludwig, H. Yashiro, N. Maruyama, M. Wahib, J. Thuburn, AIMES: advanced computation and I/O methods for Earth-System simulations, in: Software for Exascale Computing-SPPEXA 2016-2019, Springer, 2020, pp. 61–102.
[23] R.J. LeVeque, Finite Volume Methods for Hyperbolic Problems, first edition, Cambridge University Press, 2002.
[24] S. Marras, J.F. Kelly, M. Moragues, A. Müller, M.A. Kopera, M. Vázquez, F.X. Giraldo, G. Houzeaux, O. Jorba, A review of element-based Galerkin methods for numerical weather prediction: finite elements, spectral elements, and discontinuous Galerkin, Archives of Computational Methods in Engineering 23 (2016) 673–722.
[25] G. Orlando, P. Barbante, L. Bonaventura, An efficient IMEX-DG solver for the compressible Navier-Stokes equations for non-ideal gases, J. Comput. Phys. 471 (2022) 111653.
[26] G. Orlando, T. Benacchio, L. Bonaventura, An IMEX-DG solver for atmospheric dynamics simulations with adaptive mesh refinement, J. Comput. Appl. Math. (2023) 115124.
[27] F. Rathgeber, G.R. Markall, L. Mitchell, N. Loriant, D.A. Ham, C. Bertolli, P.H.J. Kelly, PyOP2: a high-level framework for performance-portable simulations on unstructured meshes, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012, pp. 1116–1123.
[28] F. Rathgeber, D.A. Ham, L. Mitchell, M. Lange, F. Luporini, A.T.T. McRae, G.T. Bercea, G.R. Markall, P.H.J. Kelly, Firedrake: automating the finite element method by composing abstractions, ACM Trans. Math. Softw. 43 (2016) 1–27.
[29] C. Schär, O. Fuhrer, A. Arteaga, N. Ban, C. Charpilloz, S. Di Girolamo, L. Hentgen, T. Hoefler, X. Lapillonne, D. Leutwyler, et al., Kilometer-scale climate models: prospects and challenges, Bull. Am. Meteorol. Soc. 101 (5) (2020) E567–E587.
[30] Swiss National Supercomputing Centre, The GridTools framework, https://github.com/GridTools, 2022. (Accessed 24 November 2022).
[31] F. Thaler, S. Moosbrugger, C. Osuna, M. Bianco, H. Vogt, A. Afanasyev, L. Mosimann, O. Fuhrer, T.C. Schulthess, T. Hoefler, Porting the COSMO weather model to manycore CPUs, in: Proceedings of the Platform for Advanced Scientific Computing Conference, 2019, pp. 1–11.
[32] The Psyclone Project Team, The Psyclone Project Website, https://github.com/stfc/PSyclone, 2022. (Accessed 8 December 2022).
[33] J. Thuburn, Y. Li, Numerical simulations of Rossby–Haurwitz waves, Tellus A 52 (2000) 181–189.
[34] TOP500. The List, https://top500.org/lists/top500/2023/06, June 2023.
[35] G. Tumolo, L. Bonaventura, A semi-implicit, semi-Lagrangian discontinuous Galerkin framework for adaptive numerical weather prediction, Q. J. R. Meteorol. Soc. 141 (2015) 2582–2601.
[36] G. Tumolo, L. Bonaventura, M. Restelli, A semi-implicit, semi-Lagrangian, p– adaptive discontinuous Galerkin method for the shallow water equations, J. Comput. Phys. 232 (2013) 46–67.
[37] J. Vila-Pérez, R. Heyningen, N. Nguyen, J. Peraire, Exasim: Generating discontinuous Galerkin codes for numerical solutions of partial differential equations on graphics processors, SoftwareX 20 (2022) 101212, https://doi.org/10.1016/j.softx.2022.101212.
[38] H. Wernli, C. Kühnlein, A. Calotoiu, H. Joos, Kilos: kilometer-scale nonhydrostatic global weather forecasting with IFS-FVM, https://www.pasc-ch.org/projects/2021-2024/kilos/index.html, 2021.
[39] S. Williams, A. Waterman, D. Patterson., Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (2009) 65–76.
[40] D.L. Williamson, J.B. Drake, J.J. Hack, R. Jakob, P.N. Swarztrauber, A standard test set for numerical approximations to the shallow water equations in spherical geometry, J. Comput. Phys. 102 (1992) 211–224.