A GT4Py Discontinuous Galerkin implementation of the Shallow Water Equations on the Sphere

Kalman Joseph Szenes

Supervisor: William Sawyer

August 14, 2022

Contents

1 Introduction

Weather and climate modeling is a notoriously challenging task. It requires a combination of advanced numerical methods as well as supercomputers to execute these expensive algorithms. With the development of heterogenous computing, supercomputing centers are increasingly relying on accelerators, which include GPUs, for execution of compute intensive tasks. As a result, established climate models need to be ported to take full advantage of the specialized hardware. Unfortunately, this task is quite time-consuming, especially for large software packages and requires frequent rewriting of the implementation to adapt to novel and fast-evolving computing paradigms. The Domain-Specific Language (DSL) GT4Py, developed at CSCS, attempts to solve precisely this issue. It provides the domain scientist with a high-level Python syntax for expressing stencil computations and is particularly well-suited for Finite-Difference and Finite-Volume schemes. Subsequently, GT4Py compiles executables of the stencils and makes use of various backends to automatically optimize the computations for the desired architecture. In addition to shifting the responsibility of the code optimization off the domain scientist, it allows for writing performance portable code. Indeed, the same stencil can be executed on a CPU or a GPU without modification of the source code by simply selecting a different backend.

In this report, we present an extension to the GT4Py framework which improves its support for a whole class of numerical methods. In particular, we implement a Discontinuous Galerkin scheme for the resolution of the Shallow Water Equations on the sphere. We benchmark our method on the Piz Daint supercomputer and compare its performance to a reference Matlab implementation.

This work is a continuation an a previous internship at CSCS conducted by Niccolo Discacciati in 2019 where a similar Discontinuous Galerkin scheme was written using the GridTools routines.

2 Mathematical Background

In this section, we highlight the key elements of the mathematical formulation of the Discontinuous Galerkin method. We refer the reader to Niccolo's final report [1] as well as [2] for a more detailed description of the theory.

2.1 Conservation Laws

In this project, we are concerned with the numerical solution of conservation laws. As the name suggests, conservation laws describe the evolution of conserved variable in a domain caused by the their flux through the boundary as well as a potential source term. This can be written as the following PDE:

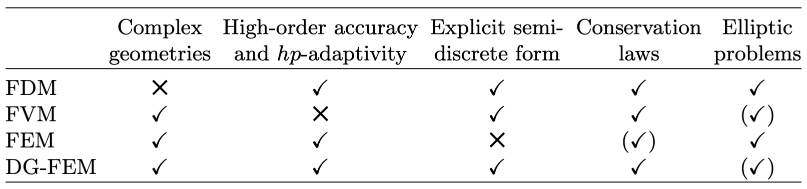
$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{f} = \mathbf{s} \tag{1}$$

where $\mathbf{u}$ is a vector of conserved variables, $\mathbf{f}$ is a flux function and $\mathbf{s}$ is a source term. Equation 1 becomes well-posed once complimented with initial conditions, as well as suitable boundary conditions.

2.2 Discontinuous Galerkin Method

The numerical algorithm that we will use to solve this conservation law is called Discontinuous Galerkin (DG). DG is a relatively new scheme that provides a number of advantage over the 3 established methods, namely Finite-Difference (FD), Finite-Volume (FV) and Finite-Element (FEM).

Firstly, DG can easily be implemented for unstructured grids which enables its use for problems with complex geometries. Moreover, DG can be expressed in an explicit semi-discrete form. This allows for the independent modification of the spatial and temporal discretization methods and leads to increased versatility in the numerical scheme. Furthermore, higher-order methods are readily available as they are trivial extensions of the lower-order methods. Finally, DG possesses a conservative form in which variables are guaranteed to be conserved during the simulation. Table 1 summarizes the key shortcomings of the 3 well-established methods.



	Complex geometries	High-order accuracy and *hp*-adaptivity	Explicit semi-discrete form	Conservation laws	Elliptic problems
FDM	×	✓	✓	✓	✓
FVM	✓	×	✓	✓	(✓)
FEM	✓	✓	×	(✓)	✓
DG-FEM	✓	✓	✓	✓	(✓)

Figure 1: [1]Table summarizing main differences between the methods for solving PDEs.

2.2.1 Spatial Discretization

For simplicity, we will develop the DG method on a 1-dimensional scalar conservation law with no source term. Thus, the general conservation law (eq. 1) reduces to:

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \tag{2}$$

[1]Taken from [2]

We begin by discretizing the domain into K elements which, in the 1-dimensional case, simply represent intervals on the line. In each element k, we define a local polynomial space using a suitable basis set. The local solution u_h^k in element k is hence written as a linear combination of the polynomial bases ϕ_i:

$$u_h^k = \sum_{i=0}^{p} \hat{u}_i^k \phi_i \tag{3}$$

where $\hat{u}_i^k$ denotes the polynomial expansion coefficients and p represents the maximum degree of the polynomials. The global solution u_h is then defined as the direct sum of the local solutions:

$$u_h = \bigoplus_{k=1}^{K} u_h^k$$

In the DG scheme, we are hence approximating the true solution to the PDE in the space of discontinuous polynomials over the elements.

To accurately approximate the exact solution, we seek for a numerical solution that minimizes the following residual.

$$R^k = \frac{\partial u_h^k}{\partial t} + \frac{\partial f(u_h^k)}{\partial x} \tag{4}$$

Notice that if the residual $R^k = 0$, the numerical solution coincides with the true solution.

Since we are approximating the true solution in a linear subspace, the residual must reside in the orthogonal complement to this subspace. This can be enforced using the following Galerkin orthogonality condition:

$$\int_{D^k} R^k \phi_i^k = 0 \qquad \forall i = 1, ..., p$$

Inserting the definition of the residual (eq. 4), we get the following relation:

$$\int_{D^k} \frac{\partial u_h^k}{\partial t} \phi_i^k + \int_{D^k} \frac{\partial f(u_h^k)}{\partial x} \phi_i^k = 0$$

By integrating by parts, we can transfer the spatial derivative from the flux function onto the basis functions:

$$\int_{D^k} \frac{\partial u_h^k}{\partial t} \phi_i^k + f^*(u_h) \phi_i^k |_{\partial D^k} - \int_{D^k} f(u_h^k) \frac{\partial \phi_i^k}{\partial x} = 0$$

As the global solution u_h is discontinuous at the boundary, the flux function will not have point values at the interfaces. Thus, we replace the true flux function by a numerical flux f^* which approximates the flux through the boundary. There are many numerical fluxes to choose from, however we will use the most standard which is the Lax-Friedrichs flux:

$$f^*(u_h) = f(u_L, u_R) = \frac{f(u_L) + f(u_R)}{2} - \frac{\alpha}{2}(u_R - u_L)$$

which is composed of a mean flux between the values at the two sides of the boundary and a numerical diffusion term which is required for stability. The parameter $\alpha = \max|f'(u)|$ is set to the maximum wave speed of the problem. More information on numerical fluxes can be found in [3].

After inserting the basis expansion from equation 3, we obtain the following semi-discrete form:

$$M^{(k)}\frac{d\hat{\mathbf{u}}}{dt} = A(\hat{\mathbf{u}}) \Leftrightarrow \frac{d\hat{\mathbf{u}}}{dt} = M^{(k)-1}A(\hat{\mathbf{u}}) \tag{5}$$

with the vector $\hat{\mathbf{u}} = [\hat{u}_0, ..., \hat{u}_p]^T$ containing the polynomial expansion coefficients for element k and the local mass matrix $M^{(k)}$:

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

In equation 5, we grouped all the remaining terms in the matrix $A(\hat{\mathbf{u}})$ This system is reminiscent of the method lines obtained in Finite Element Methods. The key difference is that the expensive task of inverting the mass matrix (eq. 5), which is local to each element, is made simple due to its reduced size.

2.2.2 Basis Functions

There are multiple choices for the basis set used for the local polynomial functions. In this study, we rely on the Legendre Polynomials which have numerous properties that make them well-suited for DG schemes. Notably, they satisfy an orthogonality condition on a reference interval $[-1, 1]$:

$$\int_{-1}^{1} P_m(x)P_n(x)dx = \frac{2}{2n+1}\delta_{mn} \tag{6}$$

This condition guarantees that expansion coefficients of different orders can be determined completely independently. Thus, the degree of polynomials in an element can be changed dynamically, during the simulation, without having to recompute all previous expansion coefficients. The orthogonality condition (eq. 6) has the added benefit of reducing the mass matrix to a diagonal matrix in which case its inversion becomes trivial.

2.2.3 Timestepping

The spatial discretization has successfully converted the PDE into an ODE (eq. 5) for which we apply standard Runge-Kutta methods for the timestepping. More precisely, we use explicit Strong Stability Preserving (SSP) [3] which exhibit desirable properties in Finite-Volume methods. They are given in the following Butcher tableau:

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array} \qquad \begin{array}{c|ccc} 0 & & & \\ 1 & 1 & & \\ 1/2 & 1/4 & 1/4 & \\ \hline & 1/6 & 1/6 & 2/3 \end{array} \qquad \begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

As we are interesting in the resolution of hyperbolic PDEs, the CFL condition constrains the size of the time step. The usual bound given by the CFL condition is:

$$\Delta t \leq C \frac{h}{\max |f'(u)|}$$

where h is the characteristic mesh spacing and C is the courant number. In Finite-Volume schemes, it is recommended to set $C \leq 1/2$. In the case of a DG method, the timestepping is more restrictive as the Courant number scales quadratically [2] with the inverse of the polynomial degree:

$$C \sim p^{-2}$$

3 GT4Py

As the architectures of modern supercomputers grow in complexity, it has become increasingly difficult to develop and maintain high-performance applications which exploit well the compute resources. This has led to the rise of Domain-Specific-Languages (DSL). They provide a simple interface for the domain scientists to develop their algorithms while the optimization of the code is handled automatically by the DSL. GT4Py is an open-source DSL embedded in Python, targeting stencil computations for weather and climate simulations. The pipeline of GT4Py is illustrated in figure 2. The domain-scientist expresses the stencils in a user-friendly Python syntax called GTScript. This code is then processed through a series of toolchains which apply optimizations and generates a high-performance program targeting a specific architecture.

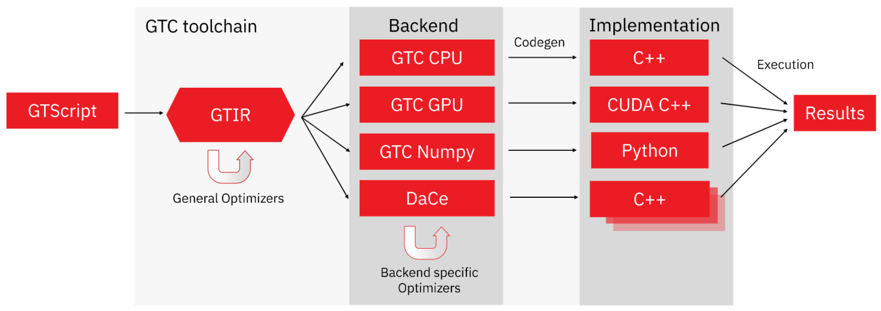


Figure 2: [2]GT4Py compilation pipeline

3.1 Installation

Instructions for installing GT4Py can be found in the repository on Github. Use pip to install all the necessary dependencies from the `requirements.txt` file. The backends for execution on GPUs and DaCe require the respective additional package dependencies `cupy-cuda110` and `dace` which can be installed through `pip`. Make sure that the version of `cupy-cudaXXX` matches the cudatoolkit version that you have (e.g. `cupy-cuda110` requires cudatoolkit version 11.0)

3.2 Backends

GT4Py has the capability to compile using various backends. A complete list of supported backends can be found in table 1. Three of the seven backends compatible with

Framework	Name

GridTools	`gt:cpu_ifirst`
	`gt:cpu_kfirst`
	`gt:gpu`

DaCe	`dace:cpu` `dace:gpu`
	`cuda` `numpy`

Table 1: List of supported GT4Py backends

GT4Py rely on the GridTools framework for compilation and optimization of the stencil computations. They are all characterized with the prefix `gt:`. The `gt:cpu_ifirst`

[2]Source: PASC22 poster by A. Afanasyev et al., *GT4Py: High Performance Stencil Computations in Weather and Climate Applications using Python*

and `gt:cpu_kfirst` both target the CPU architecture while the `gt:gpu` backend produces code for the GPU.

In addition, there are two backends that utilize the Data Centric (DaCe) parallel programming framework developed by the Scalable Parallel Computing Lab at ETH, namely `dace:cpu` and `dace:gpu` targeting CPUs and GPUs, respectively.

Alternatively, there is also a purely CUDA backend which is independent of any GridTools or DaCe routines.

Finally, there exists also a Numpy backend which can be used to inspect the generated code for debugging purposes.

3.2.1 Caching

To speed up compilation, GT4Py automatically caches stencils inside a cache folder and does not recompile a stencil unless it has been modified. The cache folder contains the binaries as well as the generated source code for each stencil. It has the following directory structure:

```
.gt_cache/{python_version}/{backend_name}/__main__/{stencil_name}
```

The `{stencil_name}` directory contains build folders which houses the generated source code. Each build folder, identified using a unique hash value as prefix, relates to a specific version of the stencil. Inside the build directories, there are two files of interest that can be inspected for debugging or performance analysis. The first one is the `computations.hpp` file which houses the C++ or CUDA implementation of the stencil computation and the second one is the `bindings.cpp` which contains the actual Python bindings.

3.2.2 Backend Options

A number of optional settings can be applied to configure the compilation process. In the following, we highlight two of them that we consider particularly note-worthy. The first one turns on the verbosity of the compiler. Since by default it is disabled, during compilation of a stencil, there is no output produced to the console. The user is hence left clueless and wondering why the execution of the Python script is being stalled.

The second one prompts GT4Py to recompile a stencil even if a cached version already exists. This is strongly recommended in case the user updates GT4Py to a newer version which may lead to earlier caches of stencils becoming incompatible. Alternatively, this can be accomplished by removing the `.gt_cache` directory entirely which will prompt the recompilation of all stencils.

They are specified in a Python dictionary that is provided to the stencil definition. An example of it is given below:

```
backend_opts = {
    "verbose": True, # always
    "rebuild": True  # only if recompilation is desired
}
```

3.3 Stencils

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

3.3.1 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:

```
import numpy as np
import gt4py.gtscript as gtscript
@gtscript.stencil(backend=backend, **backend_opts)
def laplacian(
    field: gtscript.Field[np.float64],
    out: gtscript.Field[np.float64]
):
    with computation(PARALLEL), interval(...):
        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 backend options. The arguments to the function accepts Fields on which the stencil computations are executed. GT4Py uses the Python type hinting system to specify the datatype of each Field which in this case is `np.float64`.

The body of the function requires two context managers which 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 dependance between subsequent vertical levels and they can hence all be solved in parallel. The keywords `FORWARD` and `BACKWARD` define this dependance and indicate the direction in which the vertical levels must be solved. The 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 actual stencil computation is applied for each grid point and hence relative offsets are as indices. Note that, if omitted, the offset is assumed to be `[0,0,0]`.

3.3.2 Invocation

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

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

As arguments to the function, we provide the Fields relevant to the stencil computation. In addition, we add two optional keyword arguments, namely `domain`, which specified the domain of execution of the stencil, and `origin`, which defines the origin for each Fields. 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 is used to indicate 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.

3.4 Storages

In GT4Py, Fields are variables on which Stencils can be applied. They store values at each grid point inside a 3-dimensional domain. The DSL provides a storage format for these Fields which is a wrapper over `{numpy/cupy}.ndarrays` called `gt4py.storages` which ensure 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 a number of additional parameters: `shape` defines the size of the storage in the 3 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 datatype of the Field but can also be used to assign tensors to each grid point instead of simply 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))
    )
```

In case a Field has identical values along one or more of the spatial dimension, 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 substantially 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, to obtain the results of a stencil computation, the Fields need to be explicitly synchronized from the device back to the host. 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)
```

3.5 Frontend

In this section, we describe the the structure of the GT4Py frontend and our contribution to expanding the functionality of higher-dimensional Fields.

3.5.1 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 irrespective of the specific syntax used in the source code. This representation provides a versatile 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.

3.5.2 Limited Support for Higher-Dimensional Fields

For our implementation, we will represent each DG element by a grid point in GT4Py. Each grid point will thus be assigned a vector which stores its polynomial expansion coefficients and hence yields a higher-dimensional Field as data structure. 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 automatically vectorizing operations between Fields with respect to the `data_dims` dimension. Indeed, operations needed to be unrolled explicitly for each component of the vector 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 represent the relative offsets between the fields while the second set of indices refer 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.

3.5.3 Loop Unroller

To facilitate the use of vector-valued Fields, we implemented an automatic vectorization of operations to each component of the vector. This has been accomplished by modifying an intermediate GT4Py AST called `definition_ir`. In this representation, we have access to the size of `data_dims` for each Field which is essential to determine if an operation between two fields can be vectorized. Our contribution allows to rewrite to previous multiplication stencil using the following simple syntax:

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

The goal is to modify the GT4Py frontend so that the code snippet above produces the same AST as the previous explicitly unrolled stencil. To implement this functionality, two helper classes were created which apply the necessary transformations to the `defintion_ir`. They can both be found in the file:

```
gt4py/src/gt4py/frontend/defir_to_gtir.py
```

The first one, called `UnRoller`, receives as argument an AST node representing the right-hand side of an assignment operation and returns a list of AST nodes where each element of the list pertains to a different index of the higher-dimensional Field.

The second one is called `UnVectorisation`. It invokes the UnRoller and checks that the dimensions of the returned list matches the dimensions of the Field on the left-hand side of the assignment operation. If so, it creates the list of AST assignment nodes with the corresponding indices.

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

3.5.4 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 transposed 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
```

3.6 Unit Tests

To guarantee the correctness of our implementation, we have written a number of unit tests. They rely on the existing testing infrastructure of GT4Py which verifies the success of the code generation as well as the code execution on all backends when compared with a reference Numpy implementation. The unit tests can be found in the following file:

```
gt4py/tests/test_integration/test_suites.py
```

4 Linear Advection

In this section, we present our numerical DG solver for the planar linear advection on the unit square subject to periodic boundary conditions and a constant velocity field $\boldsymbol{\beta}$:

$$\frac{\partial u}{\partial t} + \nabla \cdot (\boldsymbol{\beta} u) = 0 \qquad \text{with} \qquad \boldsymbol{\beta} = [1, 1]^T \tag{7}$$

Before developing the scheme for the full Shallow Water Equation (SWE), we consider this simpler problem first as it possesses an analytic solution and comprises the first equation of the SWE.

4.1 Precomputation

At the start of the execution of the program, our solver precomputes certain variables on the CPU that remain constant during the entirety of the simulation. This includes the computation of the inverse of the 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 so-called Vandermonde matrices that are required for evaluating the polynomials stored as modal expansion coefficients to concrete values in the domain (explained in detail in [2]). These matrices are instantiated as Fields using `gt4py.storages`.

4.2 Stencils

All subsequent computation are carried out using stencils in GT4Py. An example stencil is presented in the following. Applying the theory derived in section 2 for the linear advection problem, we will need to evaluate an integral of the following form:

$$\int_{D^k} \beta_1 u \frac{\partial \phi}{\partial x} + \beta_2 u \frac{\partial \phi}{\partial x} dxdy$$

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 * (phi_grad_x.T @ (fx * w) / bd_det_x
7                           + phi_grad_y.T @ (fy * w) / bd_det_y)
```

In line 3, the modal expansion coefficients are mapped to the quadrature points. In line 4 and 5, the flux function in x and y direction 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 line 6 and 7, 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 integral unto a reference interval.

5 Shallow Water Equations on the Sphere

The shallow water equations (SWE) can be derived from the Navier-Stokes equations by taking the limit in which the horizontal length scale is much larger than the vertical length scale. For a flat topology the SWE on the sphere are given in coordinate-free flux [4] form by:

$$\begin{cases} \partial_t(h) + \nabla \cdot (h\mathbf{v}) = 0 \\ \partial_t(h\mathbf{v}) + \nabla \cdot (h\mathbf{v} \otimes \mathbf{v}) = -f\hat{\mathbf{k}} \times h\mathbf{v} - gh\nabla h \end{cases} \tag{8}$$

where h is the water height, the vector $\mathbf{v}$ contains tangential velocities of the water. The parameters f and g are the Coriolis and gravitational constant, respectively. The vector $\hat{\mathbf{k}}$ is the unit outward normal.

5.1 Latitude-Longitude Grid

The domain is mapped to the structured latitude-longitude grid with longitude $\lambda \in [0, 2\pi]$ and latitude $\theta \in [-\pi/2, \pi/2]$ depicted in figure 3.

Note that as the elements approach the poles, they become increasingly distorted and small which can cause stability issues due to the CFL condition. Moreover, the elements precisely neighboring the poles have a singular edge (these elements reduce to triangles instead of rectangles). As a result, we expect the numerical artifacts of our scheme to be most apparent near the poles.

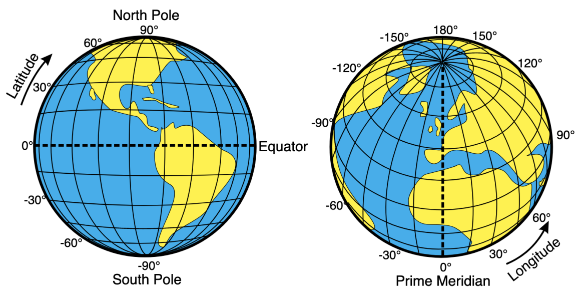


Figure 3: Latitude-Longitude grid [3]

5.2 Conservative Form

In order to implement our DG scheme, the coordinate-free flux form (eq. 8) needs to be converted into a conservative form:

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{u})}{\partial \lambda} + \frac{\partial \mathbf{G}(\mathbf{u})}{\partial \theta} = \mathbf{s}(\mathbf{u})$$

where $\mathbf{u}$ is a vector of conserved variables, $\mathbf{F}$ and $\mathbf{G}$ are the longitudinal and latitudinal flux functions and $\mathbf{s}$ is a source term.

The definition of the gradient and divergence operators in spherical coordinates [4] is given by:

$$\nabla() = \frac{\hat{\mathbf{i}}}{R\cos\theta}\frac{\partial}{\partial\lambda}() + \frac{\hat{\mathbf{j}}}{R}\frac{\partial}{\partial\theta}()$$
$$\nabla\cdot\mathbf{v} = \frac{1}{R\cos\theta}\left[\frac{\partial u}{\partial\lambda} + \frac{\partial(v\cos(\theta))}{\partial\theta}\right]$$

where $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and u and v are the longitudinal and latitudinal unit vectors and velocities respectively while the parameter R is the radius of the sphere.

By inserting these terms into equation 8, the SWE can be written in the following conservative form:

$$\begin{cases} \partial_t(h\cos\theta) + \dfrac{1}{R}[\partial_\lambda(hu) + \partial_\theta(hv\cos\theta)] = 0 \\ \partial_t(hu\cos\theta) + \dfrac{1}{R}\left[\partial_\lambda\left(hu^2 + \dfrac{gh^2}{2}\right) + \partial_\theta(huv\cos\theta)\right] = fhv\cos\theta \\ \partial_t(hv\cos\theta) + \dfrac{1}{R}\left[\partial_\lambda(huv) + \partial_\theta\left(\left(hv^2 + \dfrac{gh^2}{2}\right)\cos\theta\right)\right] = \dfrac{gh^2\sin\theta}{2R} - fhu\cos\theta \end{cases}$$

[3]Source: https://www.maptive.com/plot-latitude-and-longitude-on-a-map/

In order to simplify the definition of the conserved variables, we incorporate their cosine term directly into the local mass matrix:

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

The conserved variables are hence given by h, hu and hv.

Note that the mass matrices are all identical for a specific longitudinal value. This allows us to mask them in the longitudinal direction to save memory. In fact, masking of Fields was employed when possible. This turns out to be particularly important for the GPU-targeting backends as the accelerator has limited memory capacity.

Concerning the boundary conditions, periodic boundary conditions 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.

6 Code Validation

The goal of this section is to argue for the validity of our implementation by showcasing quantitative and qualitative agreement with predicted theory. Thus, two test cases will be presented. In the first example, theoretical estimates on the order of convergence of the DG method will be reproduced numerically for the linear advection problem. In the second case, the stability and correctness of our scheme will be illustrated on an 8-day simulation of a challenging problem for the SWE.

6.1 Convergence for Linear Advection Problem

This section presents the empirical convergence rates of our scheme and compares them with theoretical predictions. To achieve optimal convergence rates, smooth initial conditions were selected:

$$u_0(x,y) = \sin(2\pi x)\sin(2\pi y)$$

The linear advection problem (eq. 7) has an analytic solution which evolves without changing shape in the direction of the velocity field. Due to the periodic boundary conditions, the true solution will coincide with the initial conditions after one rotation. We will measure the error of the numerical approximation at this point using the L^2 norm:

$$\epsilon = ||u_h(x,y,1) - u_0(x,y)||_{L^2}$$

The expected convergence of the error as the mesh is refined is given by[2]:

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

where h is the characteristic mesh size and p is the degree of the local polynomials.

The approximation error for different polynomial degrees is presented in table 2 and illustrated in figure 4. The observed order of convergence is an great agreement

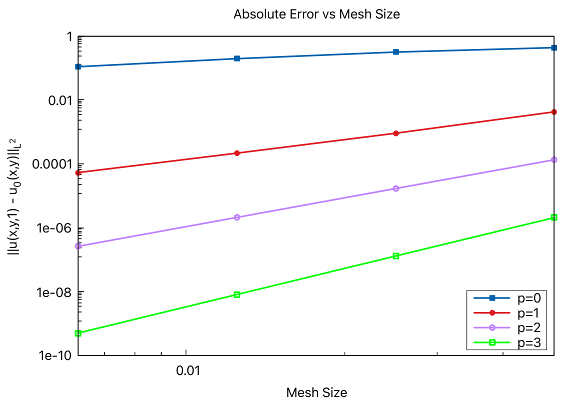


Figure 4: Convergence of errors with mesh refinement for the linear advection problem.

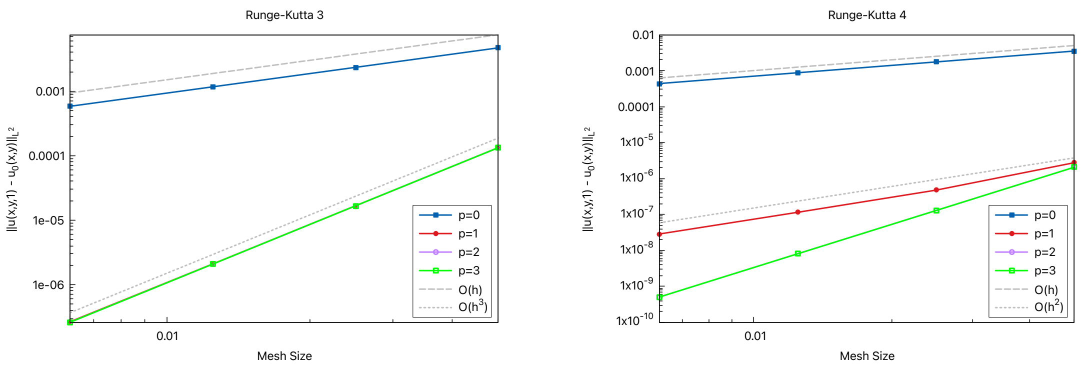
K	p = 0		p = 1		p = 2		p = 3	
	ϵ	order	ϵ	order	ϵ	order	ϵ	order
20^2	4.389e-1	-	4.204e-3	-	1.330e-4	-	2.061e-6	-
40^2	3.117e-1	**0.461**	9.004e-4	**2.223**	1.666e-5	**2.997**	1.288e-7	**4.000**
80^2	1.932e-1	**0.690**	2.139e-4	**2.074**	2.084e-6	**2.999**	8.049e-9	**4.000**
160^2	1.084e-1	**0.834**	5.212e-5	**2.020**	2.606e-7	**3.000**	5.0307e-10	**4.000**

Table 2: Numerical values used for Figure 4 and computed orders of convergence.

with the theory which is strong evidence for the validity of our application. Note that for each value of p, which characterizes the spatial order, the corresponding order Runge-Kutta method was used.

6.1.1 Mixed Order Scheme in Space and Time

In our experiments, we have observed super-optimal convergence rates by using higher-order methods in time than in space. Figure 5 illustrates the convergence of the error for the different polynomial degrees with fixed order in time.



(a) 3rd order in time

(b) 4th order in time

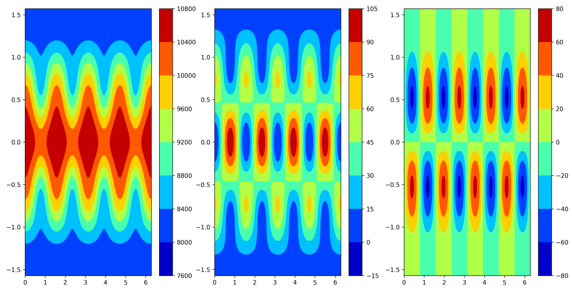
Figure 5: Convergence of error for mixed order methods in space and time. In each plot, a specific temporal order is combined with different spatial orders.

In plot 5a, a 3rd order Runge-Kutta method was used with all polynomial degrees. A few key observations can be made. First, we observe that the scheme using 2nd order method in space (p=1) has an improved convergence rate of 3. Second, the 1st order method remains unchanged. Last of all, the 4th order scheme in space is limited by the 3rd order timestepping method and hence only exhibits a 3rd order convergence rate. These results suggests that, for the DG scheme, a spatial discretization of order one less than the timestepping method may be chosen while still achieving the convergence rates provided by the higher order timestepping method.

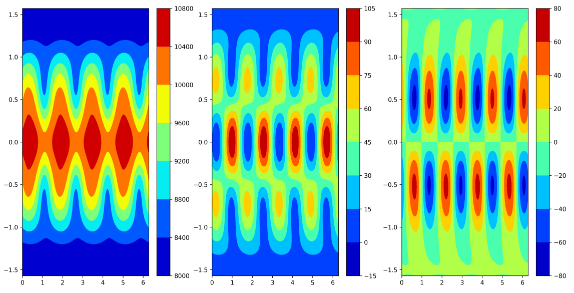
The same trend is observed in the plot 5b which illustrates the convergence rates for this numerical experiment using a 4th order Runge-Kutta method.

6.2 Rossby-Harwitz Wave for SWE

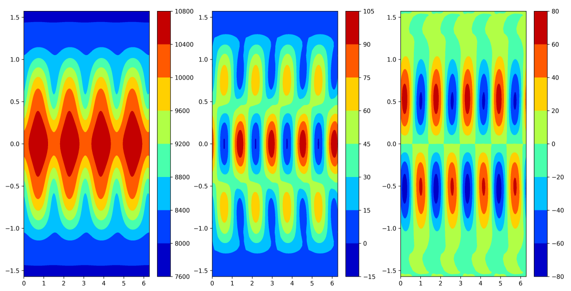
Now that we have validated our implementation of the linear advection, we move our attention to the shallow water equations on the sphere. We will simulate one of the test cases from [2], namely the Rossby-Haurwitz, which has become the gold-standard test for numerical solvers of the SWE on the sphere. The analytic solution to this problem has been shown to evolve from west to east without changing shape. In figure 6, we see the results of an 8 day simulation of the Rossby-Haurwitz wave on a 40x20 grid using our 4th order DG scheme in space and time. We used a constant time step of 0.8 seconds. The numerical solution indeed evolves from west to east while maintaining close resemblance with the initial shape. As expected, the scheme exhibits minor numerical artifacts concentrated at the poles for long simulation times.



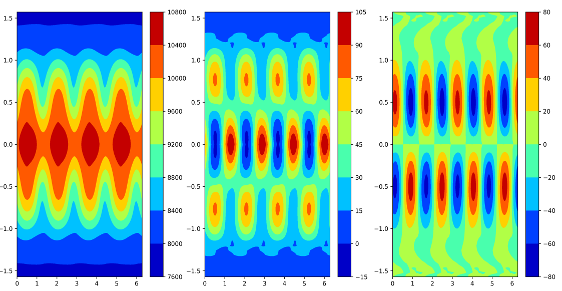
(a) Initial conditions



(b) After 30 hours



(c) After 3 days



(d) After 8 days

Figure 6: 8 day simulation of the Rossby-Haurwitz wave using our 4th order DG scheme in space and time. Each figure depicts the water height and latitudinal and longitudinal velocities at a certain time.

7 Benchmarks

In this section, we will present 3 performance benchmarks for our DG method on the SWE on the sphere. We will compare the execution time of the temporal loop for the various backends supported by GT4Py. This excludes the time spent in the precomputation steps which become negligible for long simulation periods. Each data point consists of a 20 time step simulation and, in all 3 benchmarks, we employ the same 4th order Runge-Kutta method.

The benchmarks were performed on Piz Daint with the CPU code executed on two 18-core Intel® Xeon® E5-2695 v4 @ 2.10GHz and the GPU code on a NVIDIA® Tesla® P100 with 16GB of memory. The Matlab implementation was run locally on a quad-core Intel® Core i7 6700HQ @ 2.60GHz.

7.1 Horizontal Scaling

In the first two experiments, we will study the scaling of the execution time with increased horizontal problem size. Thus, for each subsequent datapoint, the number of grid points is doubled in both horizontal directions. This leads to an asymptotic scaling of the algorithm being $O(N^2)$ where N is the number of grid points in one of the horizontal directions.

We use as baseline comparison an implementation of the same DG scheme written in Matlab.

7.1.1 Fourth Order DG Scheme

For this 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`). The performance of the various backends is illustrated in figure 7.

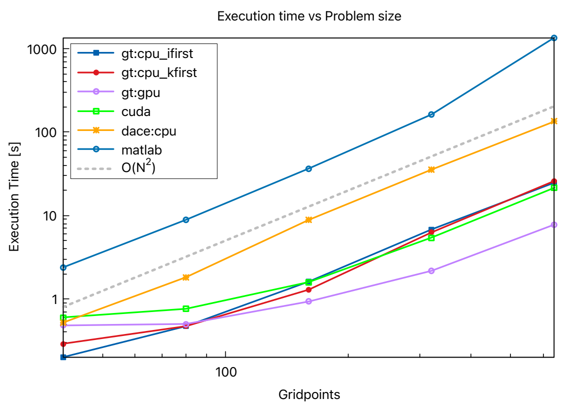


Figure 7: Benchmark of execution time of the GT4Py backends (and reference Matlab implementation) with increasing problem size. Each subsequent data point doubles the grid points in x and quadruples the total number of grid points.

The first observation is that the `dace:cpu` backend performs significantly worst compared to all the others. In addition, its GPU equivalent (`dace:gpu`) is not listed as it was producing incorrect results when compared to a reference simulation. This leads us to advise against the use of the DaCe backends as they do not exhibit good performance and produce unreliable results. The 2 CPU backends powered by the GridTools framework have virtually identical execution times and both produce faster executables than the DaCe CPU backend.

Concerning the 2 GPU backends (namely the `cuda` and `gt:gpu`), they perform worse than the CPU backends for small problem sizes. This is due to the resources on the GPU not being fully saturated for small problem sizes however their scaling with increased problem size is more favorable and hence they both end up outperforming the CPU code for larger problems. Even so, the GridTools GPU backend undoubtedly produces a better implementation as it always outperforms the native CUDA backend. Note that we were limited to presenting a maximum problem size of 640x640 due to memory constraints on the GPU.

Table 3 summarizes the speedup observed versus the Matlab baseline on the largest problem size. We observe considerable performance benefits by using GT4Py even

	Matlab	`dace:cpu`	`gt:cpu_kfirst`	`gt:cpu_ifirst`	`cuda`	`gt:gpu`
speedup	1.0x	10.2x	52.5x	55.1x	63.4x	175.2x

Table 3: Speedup of GT4Py backends vs reference Matlab implementation on 640x640 grid.

when accounting for the single-threaded nature of the Matlab implementation and the difference in architectures on which they were executed.

7.1.2 Finite Volume

One surprising observation from table 3 is that the fastest GPU backend only produces a $\sim$ 3.5x speedup versus the fastest CPU backend. Our hypothesize is that the operations that we implemented in the frontend for higher-dimensional Fields lead to bad memory access patterns in the generated code which can particularly hinder any performance benefits provided by accelerators. In this numerical experiment, we decrease the vector size stored at each grid point from 16 to only 1 (`data_dims = 1`). This essentially reduces the DG scheme to a first order Finite Volume method. In figure 8, we compare the performance of the best CPU and GPU backends.

The benchmark indeed seems to validate our hypothesis as we observe a larger gap in performance for big problem sizes, reaching up to 10x speedup on the GPU.

7.2 Vertical Scaling

In this section, we are interested in studying the performance of GT4Py on a 3-dimensional problem. In order to emulate this, the same 2-dimensional SWE problem is copied in the vertical direction and solved in parallel. This algorithm scales linearly with the number of vertical levels. Considering that all levels can be solved completely independently, this problem is embarrassingly parallel and we were expecting great performance benefits on the GPU when compared to the CPU.

Figure 9 illustrates the execution time of a 4th order DG scheme on a 300x300 grid with increasing vertical levels. Surprisingly, we do not observe any scaling benefits

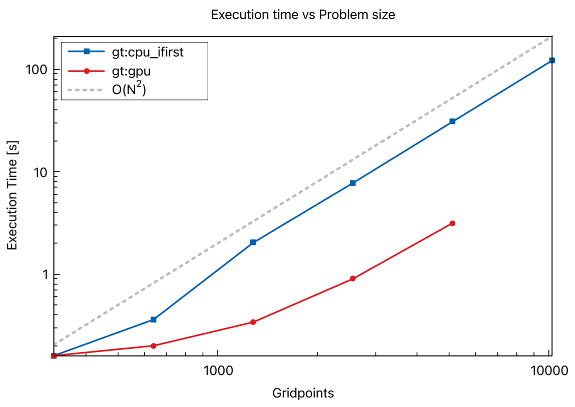


Figure 8: Benchmark of best performing CPU and GPU backends for Finite Volume scheme with `data_dims`=1. Final data point is unavailable for `gt:gpu` backend due to memory limit being reached on the GPU.

for this experiment on the GPU. Indeed, all backends exhibit linear scaling from the first data point which indicates that the resources of the hardware is already fully saturated. This is most likely due to the 2-dimensional problem, that is being solved in each level, being too large and already fully occupying the memory bandwidth of the GPU.

Considering the CUDA backend, we observe that it performs even worse than the GridTools CPU backend. Moreover, it suffers from poor memory management, when 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. This observation is corroborated by the estimated GPU memory usage reported in the job summaries on Piz Daint. Table 4 compares the memory usage between the two GPU backends for identical problems. We observe that in all cases, the CUDA implementation consumes significantly more memory than an equivalent GridTools implementation.

Nx	40	80	160	320	640	1280
`gt:gpu`	479	507	631	1059	2635	8807
`cuda`	3649	3677	3801	4229	5805	11977

Table 4: Max memory usage [MB] produces by Piz Daint job output file for various problem sizes.

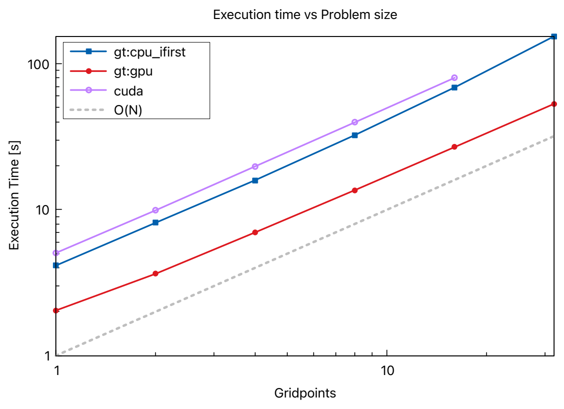


Figure 9: Benchmark of best performing CPU and GPU backends in addition to the cuda backend. Plot depicts execution time vs number of identical vertical problems solved in parallel. Final data point for the CUDA backend is unavailable due to memory limit reached on GPU.

7.3 GPU Profiling

In this section, we present the results of a brief profiling that was carried out for the GridTools and CUDA GPU backends using the NSight Systems tools from NVIDIA.

We profile the 4th order DG scheme for the 2-dimensional SWE on a 640x640 grid. Figure 10 illustrates an extract of the profile timeline for the first two time steps of the simulation. The green sections indicate memory transfers from the host to the

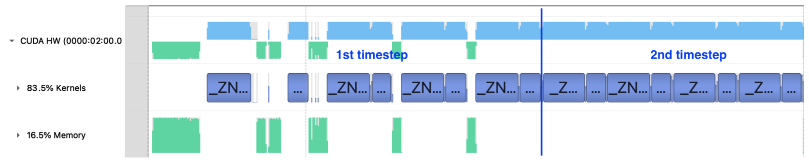
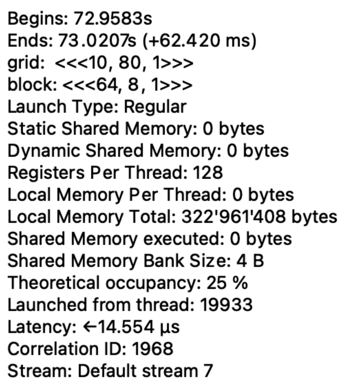


Figure 10: Extract of NSight Systems profile of `gt:gpu` backend. Profile indicates consistent executions of kernels on GPU as well as no communication after the first time step between host and device.

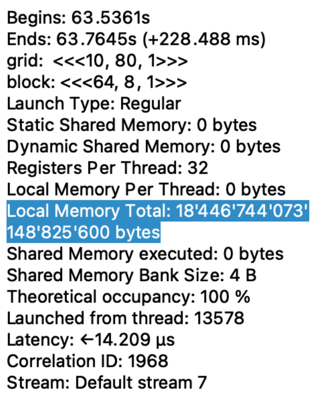
device while the blue sections highlight actual kernel executions. To achieve good performance on the GPU, communication between the device and the host needs to be reduced as much as possible. Considering the memory row in figure 10, we only observe communication (more precisely host to device transfers) in the first time step. These are necessary to transfer the initial Fields to the GPU. Subsequently, the GPU is able to execute kernels consistently without having to wait for unnecessary communication from the host.

In figure 11, we compare the performance metrics provided by the profiler between the GridTools and CUDA backends for the largest stencil in our scheme. We observe



```
Begins: 72.9583s
Ends: 73.0207s (+62.420 ms)
grid:  <<<10, 80, 1>>>
block: <<<64, 8, 1>>>
Launch Type: Regular
Static Shared Memory: 0 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 128
Local Memory Per Thread: 0 bytes
Local Memory Total: 322'961'408 bytes
Shared Memory executed: 0 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 25 %
Launched from thread: 19933
Latency: ←14.554 µs
Correlation ID: 1968
Stream: Default stream 7
```

(a) `gt:gpu` backend



```
Begins: 63.5361s
Ends: 63.7645s (+228.488 ms)
grid:  <<<10, 80, 1>>>
block: <<<64, 8, 1>>>
Launch Type: Regular
Static Shared Memory: 0 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 32
Local Memory Per Thread: 0 bytes
Local Memory Total: 18'446'744'073'
148'825'600 bytes
Shared Memory executed: 0 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 13578
Latency: ←14.209 µs
Correlation ID: 1968
Stream: Default stream 7
```

(b) `cuda` backend

Figure 11: Snippet of NSight Systems profile metrics for largest (compute intensive) stencil of the DG scheme.

that the execution time for the kernel is almost 4x slower on the CUDA backend. Moreover, there is another indication of poor memory management by the CUDA backend with the Local Memory Total being of the order of 18 Exabytes. Surprisingly, the CUDA backend is stated to have 100% theoretical occupancy when compared to the 25% of the GridTools backend. This is an indication that the generated code from GridTools could also further be improved.

8 Conclusion

In this report, we presented our implementation of a Discontinuous Galerkin scheme in GT4Py for solving the shallow water equations on the sphere. To facilitate our implementation, we expanded the support for higher-dimensional fields in GT4Py by modifying the frontend to support a clean syntax. This allowed us to port our Matlab implementation to GT4Py with relative ease. In turn, we saw that we can achieve great performance gains over our baseline Matlab code and were able to automatically exploit available accelerators without having to modify the source code.

Nonetheless, due to limitations of the functionality of higher-dimensional fields, we are left with writing a lot of boilerplate code. In particular, slicing is currently not supported which, if implemented, would allow us to condense the conserved variables into a large matrix instead of a series of separate vectors. In addition, there in no method currently for calling functions on higher-dimensional fields which would eliminate the need to copy the same functionality to different stencils. In conclusion, the new support for higher-dimensional fields opens the door for the implementation of new numerical schemes, like DG, in GT4Py. However, our frontend changes should be complemented with corresponding backend optimizations to achieve better performance.

References

[1] Niccolò Discacciati and Politecnico di Milano. "Implementation and Evaluation of Discontinuous Galerkin Methods Using Galerkin4GridTools". In: (), p. 39. URL: http://www1.mate.polimi.it/~forma/Didattica/ProgettiPacs/Discacciati16-17/Report_Discacciati.pdf.

[2] Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods*. Vol. 54. Texts in Applied Mathematics. Springer New York. ISBN: 978-0-387-72065-4 978-0-387-72067-8. DOI: 10.1007/978-0-387-72067-8. URL: http://link.springer.com/10.1007/978-0-387-72067-8.

[3] Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. 1st ed. Cambridge University Press. ISBN: 978-0-521-81087-6 978-0-521-00924-9 978-0-511-79125-3. DOI: 10.1017/CBO9780511791253. URL: https://www.cambridge.org/core/product/identifier/9780511791253/type/book.

[4] David L. Williamson et al. "A Standard Test Set for Numerical Approximations to the Shallow Water Equations in Spherical Geometry". In: 102.1 (), pp. 211–224. ISSN: 0021-9991. DOI: 10.1016/S0021-9991(05)80016-6. URL: https://www.sciencedirect.com/science/article/pii/S0021999105800166.