Swiss National Supercomputing Centre

December 15th, 2017

Implementation and evaluation of Discontinuous Galerkin methods using Galerkin4GridTools

Author: Niccolò Discacciati

Mentors: William B. Sawyer (ETH/CSCS), Christopher Bignamini (ETH/CSCS), Luca Bonaventura (Politecnico di Milano)

1 Introduction

In the last decades, in research areas linked to partial differential equations (PDEs), the interest for hyperbolic problems has largely grown. Indeed, they find applications in several fields, from molecular dynamics to atmospheric phenomena. The presence of shocks or rarefaction waves may create additional complexities.
In the development of the associated numerical algorithms, multiple aspects have to be considered. First, one would like to deal with schemes which could be applied to complex geometries, like an airfoil. Second, high order methods are needed, especially when multi-dimensional problems, non-smooth solutions or long temporal intervals are present. Third, peculiarities of hyperbolic equations are, for instance, *anisotropy* (i.e. the presence of preferential spatial directions), and conservation of physical quantities, like mass, momentum or energy. Such properties have to be somehow taken into account by the numerical scheme. Finally, algorithms exploiting spatial locality guarantee a good parallelization potential. When dealing with multidimensional problems having large number of unknowns, an efficiently parallelized code is able to reduce the computational time, without losing in terms of accuracy of the numerical solution.
From one side, the classical schemes like Finite Differences, Finite Volumes or Finite Elements fail in at least one of the above-mentioned aspects. On the other hand, the Discontinuous Galerkin (DG) solvers have all the desired properties.
In this project we developed a C++ version of a DG solver for advection equations. We focused on a standard transport equation in a two-dimensional planar domain, adding the support for a variable discretization degree in the mesh, usually called *adaptivity*. The ultimate goal of the project would be the extension to a spherical geometry. The code supports only *static adaptivity*, meaning that the discretization degrees, which are by definition space-dependent, do not change in time. A *dynamic adaptivity*, i.e. a variable-in-time discretization degree, goes beyond the goal of this project, but it could represent one of the possible extensions. For our purpose, adaptivity is mainly needed in order to deal with the poles of a sphere, which are singular points of the sphere itself. Indeed, close to the Equator the discretization degree could be reasonably high, leading to a good (and smooth) approximation of the solution. However, a lower degree is needed close to the poles, in order to avoid problems arising from the Courant–Friedrichs–Lewy (CFL) condition. Roughly speaking, close to the poles the grid spacing reduces, so that a low degree is needed in order not to be forced to pick a very low time step.
The main application of the code is a numerical analysis of atmospheric dynamics on the Earth. Indeed, generalizing the developed code to solve the shallow water equations (SWE), it could represent a different approach in research fields dealing with fluid dynamics. In such a context, the SWE are viewed as a preliminary benchmark case to investigate atmospheric dynamics.
The code is mainly built using two libraries:

- GridTools (GT), an efficient C++ library developed at CSCS. It makes extensive usage of template metaprogramming and it has optimizations for both Central Processing Units (CPUs) and Graphics Processing Units (GPUs) architectures.
- Intrepid, a C++ Trilinos library. It provides the numerical support for finite element discretizations.

The Galerkin4GridTools (G4GT) framework provides the link between these libraries, adding a higher-level layer to GridTools and the support to solve PDEs with finite elements. From the point of view of a user, this is undoubtedly and advantage, since he does not necessarily have to deal with the innermost part of GT in order to implement FE schemes.

The report is structured as follows. In section 2 we provide the numerical details about the equation that we are solving. Then, in section 3 we describe the coding strategy, highlighting its main features. In sections 4 and 5 we show the numerical results, commenting on the code performances on both CPUs and GPUs. Finally, in sections 6 and 7 we focus on the main issues which are still present, providing suggestions for possible extensions of the code.

2 Numerical discretization

Let $\Omega = [a, b] \times [c, d] \subset \mathbb{R}^D$ $(D = 2)$ be a rectangular domain and $T > 0$ a fixed time instant. A standard advection equation in its conservative form is formulated as

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \mathbf{f}(\mathbf{u}) = \mathbf{0} \tag{1}$$

where $u : (\Omega \times [0, T]) \to \mathbb{R}^n$ is the unknown solution and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^{n \times D}$ is the so-called *flux function*, which may have linear or nonlinear dependence on the solution. The superscript n denotes the number of equations we are solving. Unless stated otherwise, we will focus on the case $n = 1$, even though an extension to the other cases is, at least from the mathematical viewpoint, quite straightforward. Moreover, in the simplest case one could think about the case of linear advection

$$\frac{\partial u}{\partial t} + \nabla \cdot (\beta u) = 0 \tag{2}$$

where $\beta = \beta(\mathbf{x}, t)$ is the transport field.
In all the cases, the equation is completed with *periodic* boundary conditions for both the solution and the flux function. For instance, we impose that

$$u(x = b, y, t) = u(x = a, y, t), \quad u(x, y = d, t) = u(x, y = c, t), \quad \forall t \in [0, T] \tag{3}$$

for the solution, and similarly for the flux. A periodic initial condition is also given, denoted by $u_0 = u_0(\mathbf{x})$.
The spatial discretization is performed by splitting the domain into K elements, denoted by D^k, for any $k = 1 \dots K$. In view of the GT implementation, we consider a structured grid composed of rectangular elements. For each mesh element, let V_h^k the finite-dimensional space of polynomials up to a given degree $r = r(k)$ in each dimension. Let $u_h^k \in V_h^k$ be a local finite-dimensional approximation of the exact solution. Defining the local residual as

$$\mathcal{R}_h^k = \frac{\partial u_h^k}{\partial t} + \nabla \cdot \mathbf{f}(u_h^k) \tag{4}$$

we impose it to locally vanish in a Galerkin sense, namely:

$$\int_{D^k} \mathcal{R}_h^k \psi_h^k = 0 \tag{5}$$

for any suitably defined test function $\psi_h^k \in V_h^k$. After an integration by parts, the introduction of a *numerical flux* is needed in order to avoid the lack of uniqueness of the solution at the element boundaries. Thus, the DG formulation is

$$\int_{D^k} \frac{\partial u_h^k}{\partial t} \psi_h^k + \int_{\partial D^k} \mathbf{f}^*(u_h) \cdot \mathbf{n}_k \psi_h^k - \int_{D^k} \mathbf{f}(u_h^k) \cdot \nabla \psi_h^k = 0 \tag{6}$$

where all the terms are local to the k-th element, except for the numerical flux, which depends on the neighboring elements[1]. In particular, we implemented the so-called *Rusanov* flux, which can be seen as an average of the flux function between the neighbors plus a stabilization term depending on the difference of the solutions. Namely, consider a generic edge e and let $k, \tilde{k}$ the indexes of the elements sharing e and assume that $\tilde{k}$ is on the *right* of k. Then,

$$\mathbf{f}^*(u_h) = \mathbf{f}^*(u_h^k, u_h^{\tilde{k}}) = \frac{\mathbf{f}(u_h^k) + \mathbf{f}(u_h^{\tilde{k}})}{2} - \frac{\alpha}{2}(u_h^{\tilde{k}} - u_h^k)\mathbf{n}_k \tag{7}$$

The next step is to choose a suitable basis for the discrete space V_h^k. In particular, let $V_h^k = span\{\phi_j\}_{j=1}^{r^D}$. Since $u_h^k \in V_h^k$, it can be expanded as a linear combination of the basis functions, i.e. $u_h^k = \sum_{j=1}^{r^D} u_j^k \phi_j$. On the other hand, we consider $V_h^k = span\{\psi_j\}_{j=1}^{r^D}$. Then, (6) holds if and only if we select as a test function a generic ψ_i, $i = 1 \dots r^D$. In principle, the functions ϕ_i and ψ_i can be different, but, at least in standard Galerkin

[1] This is the reason why we write $\mathbf{f}^*(u_h)$ instead of $\mathbf{f}^*(u_h^k)$

schemes, it is a common choice to select $\phi_i = \psi_i$ for all i[2]. In particular, we select a *modal* basis for V_h^k, given by the Legendre polynomials. This is quite popular in (explicit) DG schemes, since we would end up, for instance, with a diagonal mass matrix. Moreover, the energy of the solution depends only on the squares of the coefficients u_j, up to normalization factors.
We end up with

$$\int_{D^k} \frac{\partial \mathbf{u}^k}{\partial t}\psi_i + \int_{\partial D^k} \mathbf{f}^*(\mathbf{u}) \cdot \mathbf{n}\psi_i - \int_{D^k} \mathbf{f}(\mathbf{u}^k) \cdot \nabla\psi_i = \mathbf{0}, \quad i = 1 \dots r^D \tag{8}$$

which can be cast as

$$\left[\mathbf{M}^k \frac{d\mathbf{u}^k}{dt}\right]_i + \int_{\partial D^k} \mathbf{f}^*(\mathbf{u}) \cdot \mathbf{n}\psi_i - \int_{D^k} \mathbf{f}(\mathbf{u^k}) \cdot \nabla\psi_i = \mathbf{0}, \quad i = 1 \dots r^D \tag{9}$$

where $\mathbf{M}^k$ is the local mass matrix (i.e. $M_{ij}^k = \int_{D^k} \phi_j\psi_i$) and $\mathbf{u}^k = [u_1^k \dots u_{r^D}^k]^T$ the vector which gathers the modal coefficients of the (local) solution.

In a global fashion, we can write

$$\mathbf{M}\frac{d\mathbf{u}}{dt} = \mathbf{A}(\mathbf{u}) \tag{10}$$

where $\mathbf{M}$ is the global mass matrix, $\mathbf{A}$ is a finite-dimensional operator and $\mathbf{u}$ is the unknown vector of modal coefficients of all the elements.
The temporal discretization is performed using an explicit scheme. The main advantage of such schemes is that there is no need to find the solution of (possibly nonlinear) systems, but in order to be stable they require an upper bound on the time step, usually known as CFL condition, which depends on the spatial discretization through the maximum element size and the polynomial degree r.
We can write (10) in the standard form as

$$\frac{d\mathbf{u}}{dt} = \mathbf{M}^{-1}\mathbf{A}(\mathbf{u}) \tag{11}$$

and then apply the time-advancing scheme. The classical algorithms are the (explicit) Runge-Kutta schemes, even though we performed a full validation only with an Explicit Euler (EE) scheme. Introducing a time step Δt, the fully discretized problem reads as

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = \mathbf{M}^{-1}\mathbf{A}(\mathbf{u}^{\mathrm{n}}), \quad n = 0 \dots N_{steps} - 1 \tag{12}$$

where u^0 is given. As mentioned, the mass matrix is diagonal, so that its inverse is computed in a straightforward way.
No assumption has been made on the discretization degree. Indeed, one could theoretically pick a different r in each element k. The critical step is (7), where solutions approximated with different degrees are combined. However, one has only to compute the two neighboring solutions in sets of cubature points having the same cardinality.
As a side remark we recall that, in case of a spherical geometry, the discretization process does not change, even though the involved differential operators have to be suitably modified (see e.g. [3]).
For a more detailed analysis from the theoretical point of view we refer to [2].

3 Implementation

We describe the discretization of (1) in the scalar case, i.e. $n = 1$. So far, only a linear flux function is implemented, but possible generalizations are easy to code.
Even though the problem is purely bi-dimensional, due to GT and G4GT API it is easier to perform a three-dimensional discretization, with one element in the third direction. Clearly, a zero-flux condition is imposed in the third dimension, while the initial condition and the transport field still depend on two spatial variables. Moreover, embedding the problem in a three-dimensional framework makes an extension to a 3D advection problem easier.

[2]To ease the notation, we drop the superscript k in the definition of ϕ_i, ψ_i, even though they are defined locally

3.1 Constant degree

First, we focus on the case where the discretization degree r does not vary in space. The code is divided into different blocks.

1. Preprocessing.
   In this part we instantiate the objects that never change in time. We mention the usage of the template classes `cubature<Order>` and `boundary_cubature<Order>` to instantiate the cubature points and weights. The discretization is performed in the classes `intrepid::discretization<FE,Cub>` and `intrepid::boundary_discr<BdCub,NFaces>`. Clearly, all the degrees have to be compile-time known. In particular, the values and the gradient of Legendre polynomials on the internal and boundary cubature points are computed. As cubature points, we used the Gauss-Legendre ones. This choice allows us to make the internal and boundary discretization independent, which is easier to handle in G4GT. However, for some schemes (e.g. semi-Lagrangian methods) one could be forced to switch to Gauss-Legendre-Lobatto points. In this step, only the `Intrepid` library is exploited, even though some conversion to G4GT storages has to be performed.
   For example, the discretization for the internal parts is performed as

```
using fe=reference_element<order_max, Legendre, Hexa>;
using cub=cubature<order_poly, fe::shape()>;
using discr_t = intrepid::discretization<fe, cub>;

discr_t fe_;

fe_.compute(Intrepid::OPERATOR_GRAD);
fe_.compute(Intrepid::OPERATOR_VALUE);
```

   At the same time, an object of class `assembly_dg<GeoMap>` takes into account the geometry of the grid, mapping the reference element to each of the physical elements. In case of structured meshes, the discretization of the geometric map is performed using degree-1 Lagrange basis functions.
   Moreover, as mentioned, all the constant-in-time operators and quantities are computed. Examples are the inverse global mass matrix and the Jacobian determinant of the geometric map. Since such operators have to be computed for each element, we make usage of GT stencils.
   Finally, the initialization of the solution and the transport field is performed. We want to stress the fact that a conversion from modal to nodal values has to be done. This is coded using the `Epetra` library from `Trilinos`, since the inverse of a suitable Vandermonde matrix needs to be computed.

2. Computation.
   All the constitutive blocks of (12) are assembled, with the same steps repeated at each time iteration. We make extensive usage of GT stencils, while `Intrepid` is not used anymore. The key aspect is the computation of Rusanov fluxes, which is performed in a user-friendly way thanks to the GT/G4GT API. Inside each GT stage, the access to the space-dependent storages is done as `storage(offset_i, offset_j, offset_k, indexes)`. Vice versa, for the constant-in-space values, the access is simply done by `storage(indexes)`. The difference is that the former are defined as *accessors*, while the latter are *global accessors*. Thanks to this syntax, the boundary conditions are implemented in a similar fashion, setting suitable offsets. The definition of the computational grids plays a key role here. Once the fluxes are known for each element, all the required integrals can be computed elementwise, with no communication among the neighboring elements. Finally, the update of the solution is easily implemented.
   In view of performance evaluation, this block was divided into three kernels:

   - Kernel 1: set to zero all the temporary vectors, compute u, $\mathbf{f}(u)$ in the internal and boundary cubature points
   - Kernel 2: compute Rusanov fluxes and apply boundary conditions
   - Kernel 3: compute integrals and advance in time

3. Postprocessing.
   In order to visualize the solution, a conversion to nodal values is performed inside G4GT, exlpoiting again the `Epetra` library. Together with the computational grid, the obtained nodal values are exported to `Matlab` to plot the data.

3.2 Variable degree

In case of variable degree, the general structure of the code is retained. However, the discretization part is extensively modified. Indeed, from `Intrepid` we need to have access to the basis functions values (and gradients) for all the discretization degrees from 1 to $r_{max} = \max r(k)$ in a suitable set of cubature points. To simplify the task, we assume that this set is the same for all the degrees with cardinality at least equal to $M^3 = (r_{max} + 1)^3$. With such a choice, the cubature rule is fixed, and all the required values are known a priori. Moreover, the CFL number does not depend on the choice of quadrature points.
The instantiation of the assembler is not changed, since the cubature points are the same in space. For future applications, it could represent a partial limitation, but a full extension with variable cubature points requires a huge code refactoring, as described in section 6.
For instance, the discretization of the internal part is performed by

```
using discr_t = discretize_multidim<order_max, order_poly>;

discr_t discr_;

discr_.compute(Intrepid::OPERATOR_VALUE);
discr_.compute(Intrepid::OPERATOR_GRAD);
```

The global storages are accessed in the same way as described before, but one more index is needed in order to select which discretization degree is used in the considered element.

4 Numerical results

4.1 Constant degree

The code was validated mainly using a provided baseline Fortran code. Tests with different mesh sizes and elements, initial conditions, (linear) flux functions have been run. The output was compared to the results of the Fortran code using a simple `Matlab` function `[diff]=compare(sol_Fortran, sol_G4GT)`. For the sake of simplicity, we computed a *pointwise* difference between the solutions, namely:

$$\text{diff} = \max_{i \in \mathcal{A}} \left| u_{F,i} - u_{GT,i} \right| \tag{13}$$

where $\mathcal{A}$ denotes the set of the global degrees of freedom. We remark that other comparison strategies could be exploited, leading to similar conclusions. Moreover, using a constant transport field, the exact solution is available analytically, so that one can compute the spatial discretization error as

$$||u(t) - u_h(t)||_{L^2(\Omega)} = \left(\int_\Omega |u(t) - u_h(t)|^2 \right)^{\frac{1}{2}} \tag{14}$$

In Table 1 we report the results obtained using a domain $\Omega = [0, 10]^2$, a time step $\Delta t = 0.001\ s$ (which is proven to be small enough to guarantee the stability of the scheme) and a constant advection field $\beta = (1, 1)^T$. As initial condition, we chose a $[0, 1]$-normalized one-dimensional parabolic function

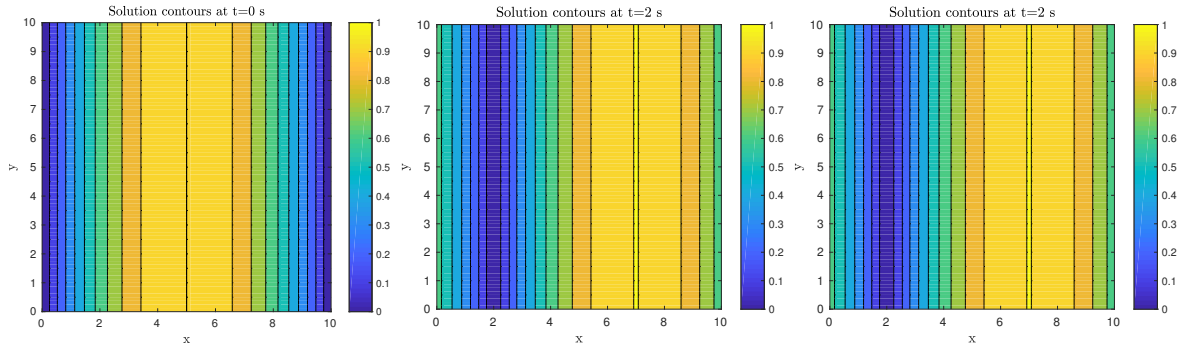
$$u_0(x, y) = -(x - a)(x - b)\left(\frac{2}{b - a} \right)^2 \tag{15}$$

Since it can be integrated exactly using a suitable number of cubature points, we expect that switching from Gauss-Legendre-Lobatto to Gauss-Legendre points does not affect the computation. We used the CPU version of the code.

The discretization error has the right scaling with respect to the grid spacing Δx. Indeed, it can be verified that it scales as Δx^2, remembering that in the reported tests we kept $d_3 = 1$ with a constant z-length of 2. For the sake of completeness, in Figure 1 we show the contour plot of the solutions in the last test case. No significative differences between the two implementations are present, but it is evident the effect of the advection field. Indeed, after $T = 2\ s$, the maximum is shifted by T to the right. Moreover, thanks to the y-symmetry of the solution, no variations in y appear.

	d_1	d_2	Δt $[s]$	T $[s]$	deg	diff	$\|\|u(T) - u_h(T)\|\|$
Test 1	10	10	0.001	2	1	$3 \cdot 10^{-6}$	$2.98 \cdot 10^{-1}$
Test 2	20	20	0.001	2	1	$3 \cdot 10^{-6}$	$1.39 \cdot 10^{-1}$
Test 3	40	40	0.001	2	1	$4 \cdot 10^{-6}$	$6.38 \cdot 10^{-2}$
Test 4	80	80	0.001	2	1	$4 \cdot 10^{-6}$	$2.83 \cdot 10^{-2}$
Test 5	160	160	0.001	2	1	$3 \cdot 10^{-6}$	$1.27 \cdot 10^{-2}$
Test 6	20	20	0.001	2	4	$4 \cdot 10^{-6}$	$8.15 \cdot 10^{-4}$

Table 1: Comparison with the Fortran code in the case of a parabolic initial condition. The reported tests differ in the choice of the discretization degree deg, the number of mesh elements d_1, d_2 and the final time T.



(a) Contour plot of $u_0(x, y)$ (b) Contour plot of $u_F(x, y, T = 2)$ (c) Contour plot of $u_{GT}(x, y, T = 2)$

Figure 1: Contour plot of discretized solution of the advection equation, with parameters defined as the case "Test 6". u_F and u_{GT} denote the solutions obtained in the Fortran and in the GridTools implementations respectively.

A more interesting case is given by choosing as initial condition the cosine-bell function. We refer to Equation (79) in [4] for the precise definition. For our test case we used a bell centered in the point

$$(x_c, y_c) = \left(\frac{a+b}{2} - 0.2(b-a), \ \frac{c+d}{2} - 0.2(d-c) \right) \tag{16}$$

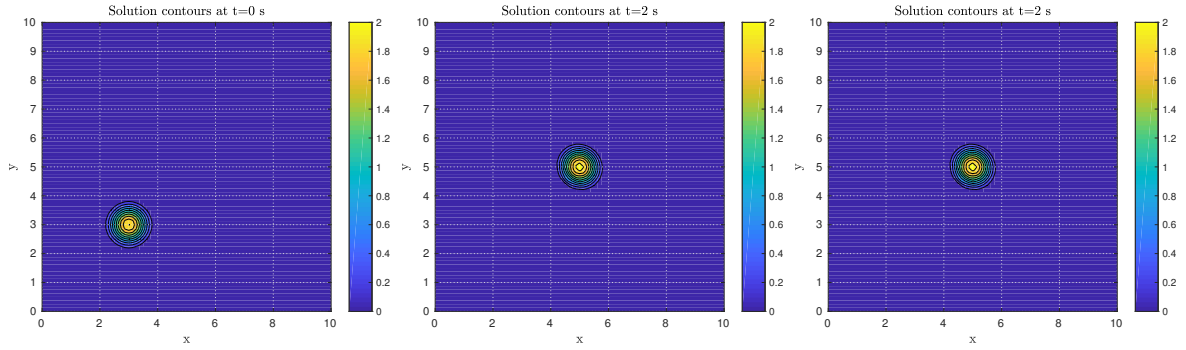
with a radius $R = 0.1 \cdot (b - a)$ and $h_0 = 2$. The other parameters are chosen to be the same of the previous cases, as reported in Table 2.

Obviously, a cosine function cannot be integrated exactly by a gaussian cubature rule. Thus, it may happen

	d_1	d_2	Δt $[s]$	T $[s]$	deg	diff	$\|\|u(T) - u_h(T)\|\|$
Test 7	20	20	0.001	2	1	$5 \cdot 10^{-6}$	$1.03 \cdot 10^{-2}$
Test 8	20	20	0.001	2	4	$5 \cdot 10^{-6}$	$3.29 \cdot 10^{-3}$

Table 2: Comparison with the Fortran code in the case of a cosine-bell initial condition. The reported tests differ in the choice of the discretization degree deg and the final time T.

that the results between the analyzed implementations could differ, since different choices for the points are exploited. However, a good strategy in order to reduce the cubature error is to increase the number of points. For instance, in the "Test 7" case, we used 5 Gauss-Legendre and 6 Gauss-Legendre-Lobatto points respectively, even though the one-dimensional basis functions are of degree one. In this way, the obtained differences with the Fortran code are again of order 10^{-6}.



(a) Contour plot of $u_0(x, y)$ (b) Contour plot of $u_F(x, y, T = 2)$ (c) Contour plot of $u_{GT}(x, y, T = 2)$

Figure 2: Contour plot of discretized solution of the advection equation, with parameters defined as the case "Test 8". u_F and u_{GT} denote the solutions obtained in the Fortran and in the GridTools implementations respectively.

In Figure 2 we show the results for the last test case. As before, we observe the effect of the advection, which is responsible of transporting the bell towards the top-right corner. Moreover, being $u_0 \geq 0$, from a theoretical point of view a positive transport field does not alter the sign of the solution. However, some spurious oscillations appear, due to the numerical scheme. Even though such oscillations are of order 10^{-4} for our case, increasing the time step we observe an unstable behavior. Such oscillations are not reported in Figure 2. One possible solution to eliminate this effect is to use a flux limiting technique.

4.2 Variable degree

Different tests have been performed even in the adaptive case. No baseline codes are available, but a simple check to verify the correctness of the code is to use a constant degree for all the mesh and compare it to the non-adaptive version. This check was performed using the parabolic function defined in (15), with $d1 = d2 = 100$ elements, $\Delta t = 0.01$ s and uniform discretization degree equal to $r = 4$. The error is of order 10^{-8}, which validates the code.
We switch directly to the cosine-bell case with the same discretization parameters. We consider the bell centered in the point $(x_c, y_c) = (30, 50)$ with radius $R = 20$. Regarding the variable degree, we considered a constant r along x, while we assumed a variation in y in order to have the maximum (resp. minimum) value at the center (resp. boundaries) of the domain, with jumps at most equal to one. This choice was done in order to mimic the behavior on a sphere, where we would expect low degree close to the poles. Figure 3 shows the distribution of the degrees.

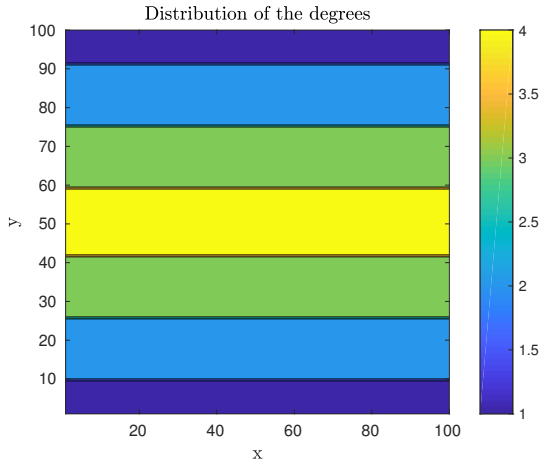
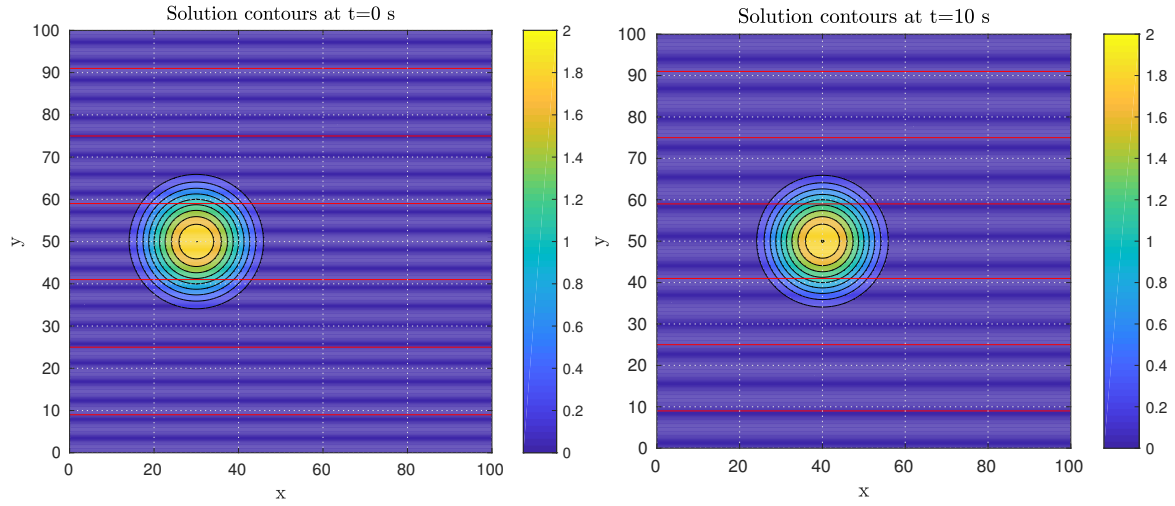


Figure 3: Distribution of the degrees for the case of variable degree

From a visual perception, one may claim the variable degree seems not to have effect on the behavior of the solution, ignoring again the oscillations. However, we can still compute the discretization error, which is equal to $||u - u_h||_{L^2(\Omega)} = 1.08 \cdot 10^{-3}$. Using a constant degree $r = 4$ in all the domain, we would end up with an error which is $||u - u_h||_{L^2(\Omega)} = 9.86 \cdot 10^{-4}$. We may claim that the order of magnitude of the two errors is the same.

However, the computational effort in the first case is much lower, since the degree is reduced where the solution is close to zero, requiring less operations. Therefore, for this test case we tend to prefer the variable-degree case. We refer to section 5 for a more detailed analysis of the performances.



(a) Contour plot of $u_0(x, y)$

(b) Contour plot of $u(x, y, T = 1)$

Figure 4: Contour plot of discretized solution of the advection equation, with $d_1 = d_2 = 100$, $\Delta t = 0.01$ s and degree r varying from 1 to 4. The red lines separate the regions with different degrees.

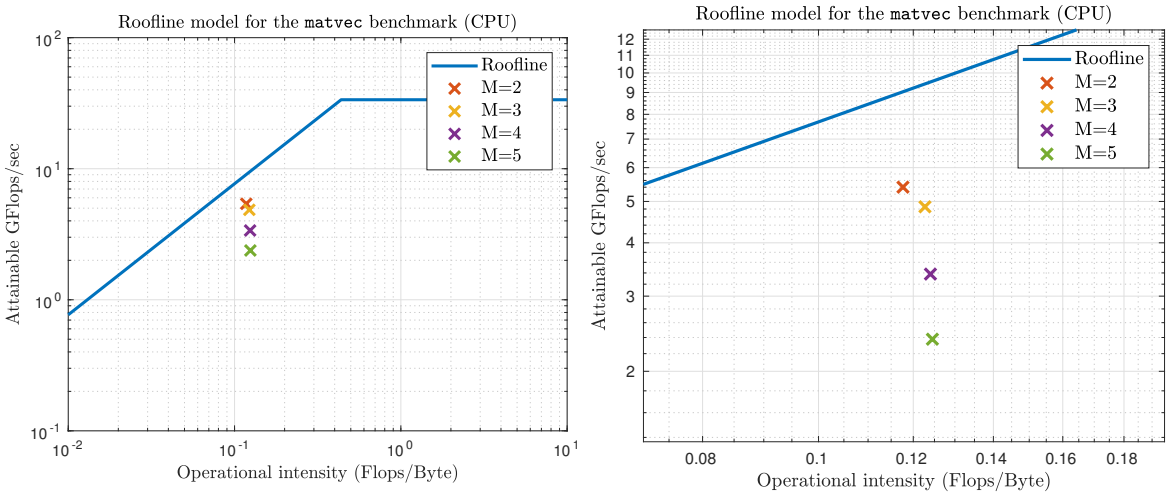
5 Performance elevaluation

Once the results are validated, an extensive performance evaluation was performed. The code makes usages of GT optimization for both a CPU and a GPU architecture and most of them are set in the `backend<BACKEND_ARCH, GRIDBACKEND, BACKEND_STRATEGY>` type. The second parameter is always set in order to trigger optimization for structured grids. The third one is set to use a `Block` strategy. However, the first one changes according to the architecture we are working on. In particular, if the code is designed for a GPU we need to set it to `Cuda`, while we leave the default value of `Host` when using a CPU. The backend information are set in `numerics/basis_functions.hpp`. These definitions trigger several architecture-based optimizations. For instance, we mention the layout maps of GT storages. In the case of a CPU implementation, the dimension with higher stride is the third one, while in a GPU version it is the first one.
Concretely, the performance evaluation was performed using the Roofline model [5], which is based on two main variables. The first one is the so-called *operational intensity*, defined as the number of floating-point operations (flops) per byte of DRAM traffic, taking into account for the bytes that are read from/written in the main memory after the filter of the cache hierarchy. The second index is the number of *attainable Gflops per second*, i.e. the concrete performance measure. These values can be plotted in a two-dimensional graph. 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. Practically, this threshold is determined by running benchmark cases, as the *stream* or the *linpack* benchmark. Thus, for a given operational intensity, a high-performing kernel should be close to the determined limit.
In the following, we make a key simplifying assumption, namely we ignore the cache effects. In other words, every access to a variable is taken into account 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 away from being an easy task. Indeed, a handmade computation turns out to be unfeasible, while using some performance analysis tools becomes compulsory.

5.1 Matrix-vector multiplication

Going back to the advection problem, observe that in the linear case (2), the operator $\mathbf{A}$ in (10) is linear and it can be proven that the fully discretized scheme can be reduced to matrix-vector multiplications, plus standard arithmetic operations with vectors. Thus, one may claim that a single matrix-vector multiplication can be viewed as a small and simple representative model for the global problem. This is the reason why we perform an evaluation on this kernel before generalizing to the global problem.

Focusing on a square matrix with n rows and columns, it can be verified that a standard implementation of the matrix-vector multiplication involves a number of flops equal to $\#flops = (2n)n$. The memory traffic amounts to $\#bytes = 8(2n+1)n$, assuming double precision is exploited. By definition, the operational intensity is the ratio between the former and the latter quantity. In the following we will consider $n = M^3$, since we want to mimic the advection code, in which a three-dimensional tensor product of the basis functions is used.
Concretely, we define a local $n \times n$ matrix in all the $d_1 \times d_2 \times 1$ mesh elements, performing n_{it} times the same operations. Thus, the total number of flops and bytes amount to $n_{it}d_1d_2(2n)n$ and $8n_{it}d_1d_2(2n+1)n$ respectively, while the operational intensity is not altered. We will also focus on the cases $n = M^3$, $M = 2 \dots 5$. Practically, the matrices we considered are block-diagonal.
Let us consider the CPU case first. The simulation was performed on the compute nodes of Piz-Daint at CSCS, using an Intel Xeon E5-2695 v4 processor (single node). The roofline is easily plotted knowing the values of the peak memory bandwidth (76.8 GB/s) and the peak floating point performance (33.6 $Gflops/s$) as in [6]. Moreover, we considered a mesh consisting of $d_1 \times d_2 \times d_3 = 200 \times 200 \times 1$ elements, running $n_{it} = 40$ iterations. Figure 5 shows the results.

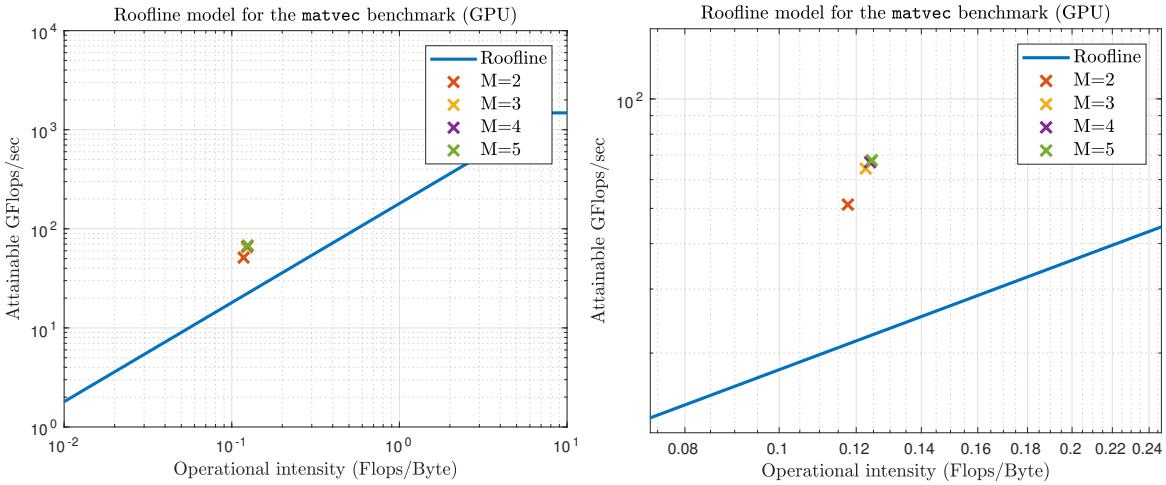


(a) Normal range

(b) Zoomed range

Figure 5: Performance of the matrix-vector multiplication kernel for the CPU case with different matrix sizes $n = M^3$ and $(d_1, d_2, n_{it}) = (200, 200, 40)$

First, we observe that no significant variations in the operational intensities are present, being all the values reasonably close to the asymptotic value of $2/16 = 0.125$ $flops/bytes$.
For all the considered matrix dimensions, we never obtain performances which are comparable to the roofline model. Being the kernel in the memory-bound range, we expect that the problem is related to a bad access to the data, together with a non-optimal usage of the cache. The performances get worse when the dimension of the problem increases, with a monotonically decreasing behavior with respect to the dimension $n = M^3$. This trend appears to be strange, since one may expect that the cache effects become more significant for higher matrix dimensions. Thus, one expects at least similar performances as n varies.
One could arrive to the same conclusion also by looking at the computational times. For instance, with $M = 5$ (i.e. a 125×125 local matrix), the required time is approximately 25 s, which is clearly too high for high performance applications.
Several strategies have been exploited in order to try to get better performances, but the results were not so good as expected. We refer to section 6 for a more detailed analysis of the problems and the consequent possible solutions.
Instead, we switch to the GPU case. Again, the simulation was performed on Piz-Daint using an NVIDIA Tesla P100 unit (single node). In this context, the roofline model is uniquely determined by the peak memory bandwidth of 180 GB/s and a peak floating point performance of 1480 $GFlops/s$ as in [7]. In order to compare the results with the CPU case, we used the same parameters as the previous test, namely $d_1 = d_2 = 200$, $d_3 = 1$ and $n_{it} = 40$. The results are shown in Figure 6.



(a) Normal range

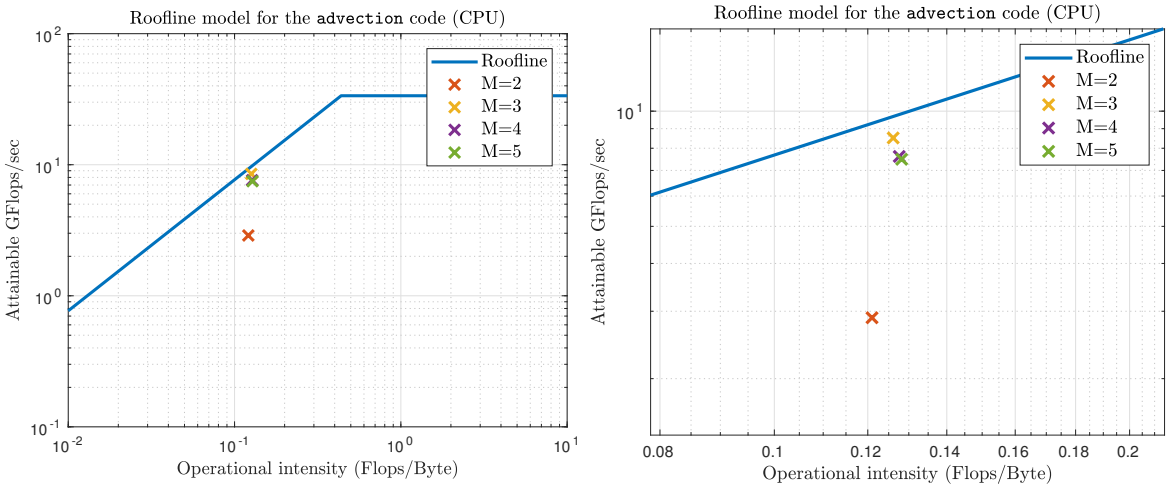
(b) Zoomed range

Figure 6: Performance of the matrix-vector multiplication kernel for the GPU case with different matrix sizes $n = M^3$ and $(d_1, d_2, n_{it}) = (200, 200, 40)$

We observe that the performances are very similar as the dimension of the matrix is varied. Moreover, in all the cases we obtain values of attainable flops per second which are above the hardware limit. Even if this could appear as a contradiction, an explanation to this behavior is found in the simplifying assumption we made to compute the required values. In particular, if one took into account for the cache effects, the bytes traffic to the DRAM would reduce, increasing the operational intensity. Thus, the obtained points would move towards the right of the graph, so that they could end up being below the roofline.
As a conclusion, it is evident that in the GPU case the code performs in a way which is partially consistent with the roofline, as we have cache-friendly memory accesses. Vice versa, the CPU version performs badly, and the results could become even worse when relaxing the above-mentioned hypothesis.

5.2 Advection problem

Going back to the original problem, we performed similar analyses as the one described in the previous subsection. Once more, we used $(d_1, d_2, n_{it}) = (200, 200, 40)$. We picked a constant advection field $\beta = (1, 1)^T$, a time step $\Delta t = 0.01\ s$ and $u_0(x, y) = 1$. In this context, we choose the number of 1D cubature points equal to the 1D basis cardinality, both denoted by M. Clearly, if r denotes the 1D discretization degree, we have $M = r+1$. This choice allows us to simplify the required computations, while performing exact integrations in the scheme. We also verified that the time step is small enough to get the expected results, even though the CFL number increases with respect to the cases shown in section 4. In the following, we will ignore the performance analysis of the previously described preprocessing step, while we focus only on the time-dependent part. Indeed, the first computations are never repeated in time and if we consider a reasonably large number of iterations, they should not represent the dominant part of the code. Moreover, this step is far away from being optimized, so that if we took it into consideration, it would invalidate some results. Finally, it is the only step where we make usage of `Intrepid` and we have low control on the `Trilinos` optimizations.
As we will verify later on, changing the discretization degree, the operational intensities do not have significant variations. Theoretically, one would expect an increasing intensity with respect to $M = r + 1$. However, the performances are evaluated using an Explicit Euler scheme. In general, one should use a time-integrating algorithm with an order which depends on the spatial discretization, in order to have comparable orders in space and time. Higher order schemes (e.g. RK2 and RK3) would involve larger number of flops with essentially the same memory traffic (assuming that all the temporary vectors are ignored or have low influence on the memory traffic itself), leading to an increasing operational intensity.
Let us analyze the CPU case. Being the computing environment the same with respect to the `matvec` benchmark, the maximum achievable performance line does not change. The results are shown in Figure 7.

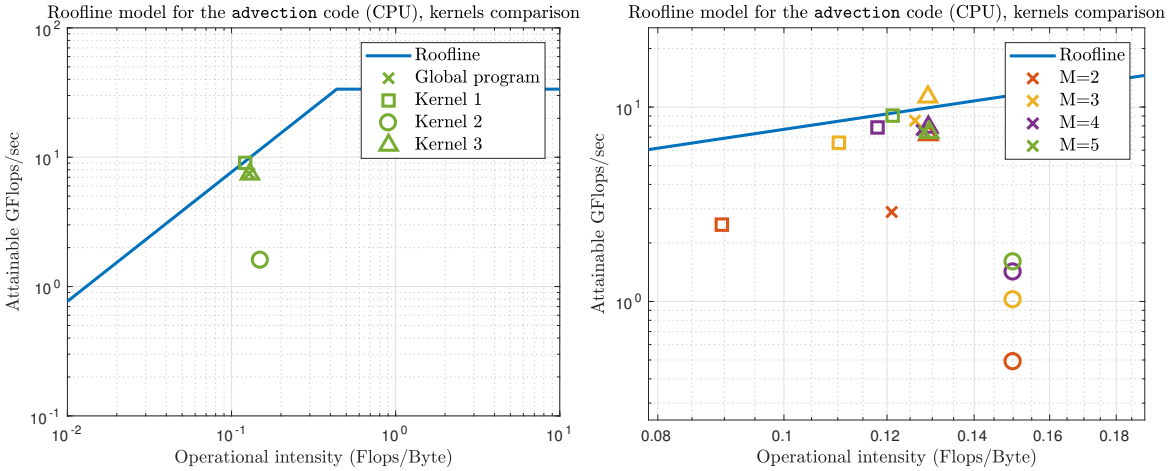


(a) Normal range

(b) Zoomed range

Figure 7: Performance of the advection problem for the CPU case with different number of degrees of freedom and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with linear initial condition

Except for the $M = 2$ case, we observe that the performances are similar when we change the number of degrees of freedom. Moreover, we find that the performances are better when compared to the `matvec` case. This partially contradicts the initial hypothesis which assumed the matrix-vector kernel as a simplified problem. However, since several stages are performed, some memory optimizations can arise. In a single computational kernel we probably do not see such effects, and it is also possible that more operations should be performed in order to trigger a fully optimized code. We also recall that non only matrix-vector multiplications are performed, but we also have operations as vector copies, sums and assignments.
Since multiple stages are involved at each iteration, it is quite easy to investigate which one prevails in the computation. Remembering the kernels we identified in section 3, we observe that the third one has the largest influence on the global program in terms of total flops. For large values of M, it gathers the 85% of the total operations. Since all the integrals are performed in it, such a behavior could be expected. In particular, the computation of the internal integral has the largest influence, since it uses multiple constant-in-space values (e.g. gradient of the basis functions) and the number of flops scales as $24 \cdot M^6$. At the same time, the first and the second ones have approximately the 0.01% and the 15% of the total flops respectively. Therefore, we expect that the performances of the third kernel are similar to the ones of the global program.
In Figure 8 we plot the above-mentioned performances of the kernels.

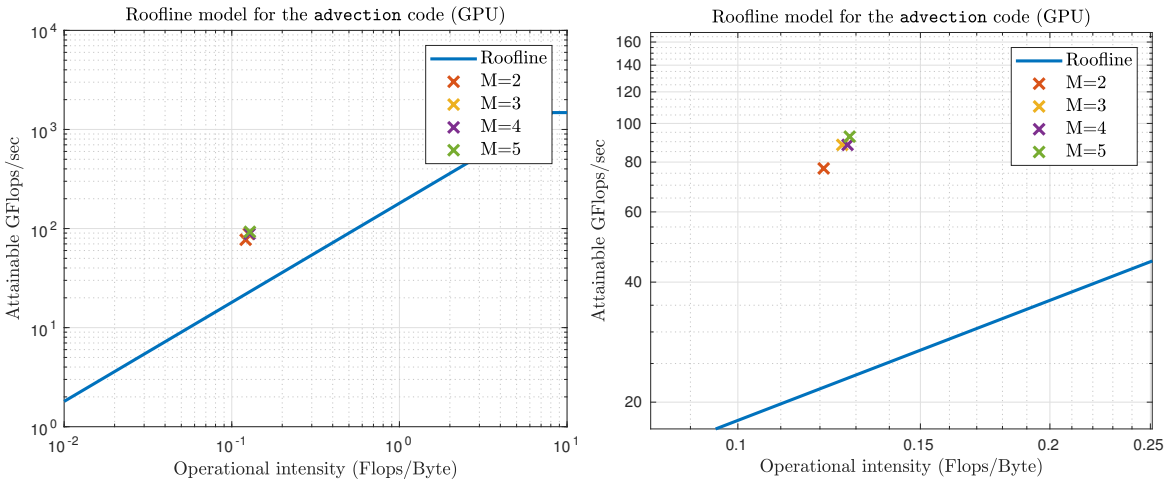


(a) Performances of the kernels ($M = 5$)

(b) Performances of the kernels (different M)

Figure 8: Performance of the different kernels of the advection problem for the CPU case with different number of degrees of freedom and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with linear initial condition. The crosses denote the performances for the global program, the squares for kernel 1, the circles for kernel 2, the triangles for kernel 3. The colors follow the same convention of Figure 7.

Excluding again the case $M = 2$, our expectations are confirmed. The first kernel has performances comparable to the third one, while the second one gives values which are approximately one order of magnitude inferior with respect to the theoretical line. Since it is the only kernel which involves exchanges between neighboring elements, it is reasonable that cache-misses or non efficient memory accesses are present. Thus, this kernel is the slowest one, in relation with the number of flops.
Finally, one comment has to be done for the linear case $M = 2$. It turns out to be that, even though the third kernel still has the highest percentage of total flops (around 65%), the first kernel has more influence and the performances are always below the others. Since the dimension of the local matrices and vectors are relatively small (8×8 and 8×1 respectively), it could be that some GT optimizations are not fully exploited and the caches do not have a remarkable impact. It is also the only case in which the computational time of the first kernel is comparable with respect to the third one.
Let us switch to the GPU case, which is run under the same environment of the `matvec` benchmark. The results of the performance analysis are reported in Figure 9.

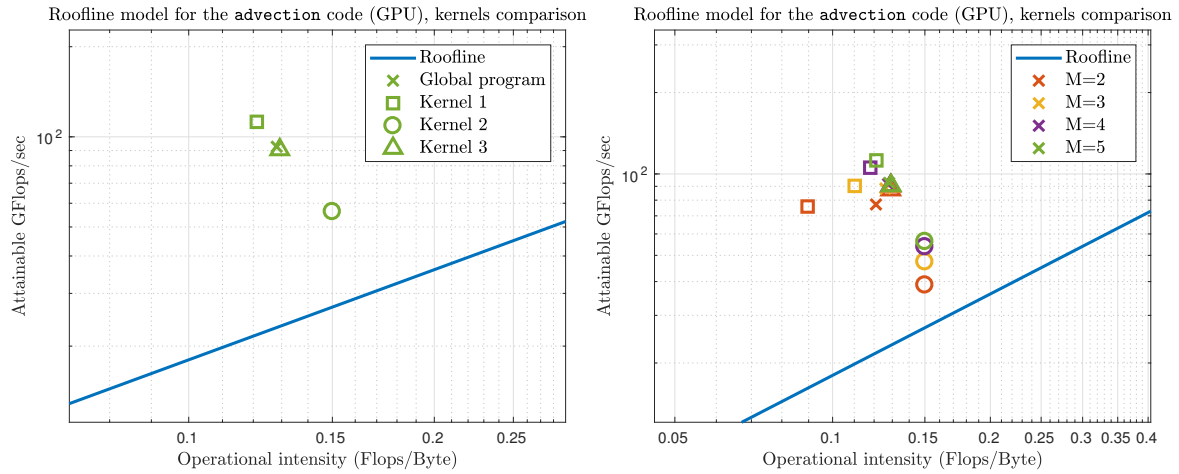


(a) Normal range

(b) Zoomed range

Figure 9: Performance of the advection problem for the GPU case with different number of degrees of freedom and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with linear initial condition

The performance values are still above the theoretical line and they are very close. Again, the $M = 2$ case is the worst one, but in this context the order of magnitude is comparable with respect to the others. Basically, all the comments done for the `matvec` benchmark are still valid and it can be claimed that, using a GPU, it can be viewed as a representative case for our problem.
For the sake of completeness, in Figure 10 we show the performances of the different kernels.

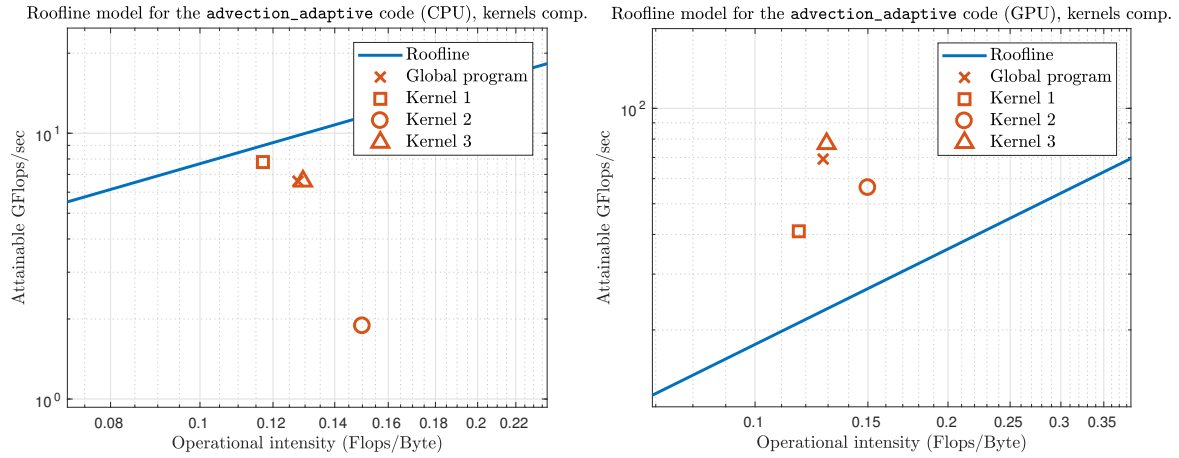


(a) Performances of the kernels ($M = 5$)

(b) Performances of the kernels (different M)

Figure 10: Performance of the different kernels of the advection problem for the GPU case with different number of degrees of freedom and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with linear initial condition. The crosses denote the performances for the global program, the squares for kernel 1, the circles for kernel 2, the triangles for kernel 3. The colors follow the same convention of Figure 9.

As before, the dominant one is the third one, even in the case $M = 2$. The second kernel is again the worst in terms of performances, but the values are still above the roofline.
As a final analysis, we consider the variable-degree case. The results for the problem we described in subsection 4.2 are shown in Figure 11, and are coherent with the previous analyses. Moreover, choosing a constant degree equal to r_{max} the total time would double (484 s and 1019 s), while having a similar discretization error.



(a) CPU case

(b) GPU case

Figure 11: Performance of the different kernels of the advection problem with different variable discretization degree and $(d_1, d_2, n_{it}) = (200, 200, 40)$, with linear initial condition.

6 Main issues and possible solutions

We would like to conclude this report highlighting the main problems that are still left to solve, providing some suggestion for improvements of the code.
The first issue to be solved is about code performances on the CPU. As mentioned, the problem is related to memory accessing. We also tried to change the `layout_map` of the storages, but the default one has proven to be the best solution. Moreover, we believe that all the GT and G4GT optimizations are turned on. For instance, we checked that the `BACKEND` has the correct settings even in the CPU version, since we manually set the macro `BACKEND_BLOCK`. However, these solutions do not completely solve the problem. A strange fact is that the performances of the `advection` code are better than the `matvec` benchmark. Thus, it could be that the number of operations inside the `matvec` stencil is not large enough in order to fully exploit all the

optimizations. One check is to define a kernel which performs n_{it} times the same matrix-vector multiplication. The difference with respect to the `matvec` case is to put this loop *inside* the functor and not in the main program. We believe that no significant differences appear, but it's worth giving a try. Furthermore, since GT is optimized for three dimensional problems, we tried to use more than one element along the z direction (setting, for instance, $d_3 = 10 > 1$). In this context, the performances drop with a rate which is higher than before, even in the matrix-vector multiplication example. Such a behavior has to be investigated, since it is hugely unexpected. In all the simulations, we remark that the time step has to be small enough to guarantee the stability of the solution, since the performances show a huge drop when dealing with large numbers.
Another important issue, which could be related to the first one, is about some conflicts which could be present when compiling the two main libraries, i.e. GT and G4GT. It may happen that some flags, which are activated when building GT, turn to be disabled. This was the case with the `ENABLE_METERS`, which is turned on in the GT building but has to be manually defined in the code. A similar behavior is found for some GPU flags, namely `__CUDA_ARCH__` and `KOKKOS_HAVE_CUDA`. A temporary solution was found, but some time should be spent on its understanding. We remark that for a still unknown reason, the non-adaptive code compiles and runs on a GPU without explicitly setting the `KOKKOS_HAVE_CUDA` flag, while the compilation breaks for the adaptive case.
Third, even though this is not a primary problem, one should take into consideration the memory release after certain steps are performed. This prevents memory allocation errors, which can happen when selecting a high number of elements and/or a high discretization degree. As a simple example, consider the computation of the Jacobian of the geometric map. Once we compute its determinant (which is used in the computation of the integrals), we never use the matrix anymore. In this way we are able to save $d_1 \cdot d_2 \cdot d_3 \cdot M^3 \cdot 3 \cdot 3 \cdot 8$ bytes in the memory.
A last minor fix is needed in order to define the dimensions of some G4GT storages at compile time, as the ones needed for the values/gradients of the basis functions. The current version of the code requires a manual definition of such dimensions[3], since in `Intrepid` there is no way to access to those values at compile time. A possibility is to use the type traits as it is done, for instance, in `numerics/element_traits.hpp`.
We would also like to comment about further development on the numerical point of view. Flexibility on the choice on the cubature points has to be added. In particular, one would like to use the Gauss-Legendre-Lobatto points. As mentioned, the G4GT structure is not well designed for such a choice. Roughly speaking, the main issue is that, given a set of cubature points, it is not trivial to classify those points as boundary or internal ones. With the Gauss-Legendre nodes, the task is easier, since it is enough to verify if values of ± 1 are present in the points. In the files `numerics/legendre.hpp` and `numerics/cubature.hpp` a possible (not fully tested) implementation is shown. Moreover, there is still an issue about the computation of the `Intrepid::OPERATOR_GRAD` of Legendre polynomials on the boundary. Its fix should be relatively easy, in the sense that one has to mimic the computation of the `Intrepid::OPERATOR_VALUE`. This was not done so far, since the gradient on the boundary is never required in our codes.
Higher order schemes in time have to be implemented, too. Even though they are a simple generalization of the EE scheme, the coding of such algorithms is nontrivial. Indeed, one has write the Runge-Kutta loop in the main program, calling the different stencils in it. The initialization of the temporary vectors has to be taken into consideration, too.
For the adaptive case, one would like to freely choose the number of cubature points in all the elements. As mentioned, this variability would require a huge code refactoring. Indeed, we need to provide to the `dg_assembly` class the distribution of the chosen cubature points. Moreover, the computation of the Rusanov flux would become critical, since we need to evaluate the flux itself on a common set of cubature points. In this context, we are unsure if all the computations performed in the precomputation step could still be exploited.
We would like to make a final comment on the GT and G4GT framework. As soon as we got familiar with the syntax (and the coding technique), we didn't experience huge problems in understanding how GT operates, at least from a global perspective. Different compilation errors were found, and most of them were difficult to be understood, since the error messages are unclear. Most of them are related to the GPU backend. For instance, when accessing to the first dimension of a GT storage, GT returns the *aligned* one without any warning. This is usually different from the user's expectation. Another limitation is related to the impossibility make *nested* calls to functors. For instance, this creates complexities when coding RK2 or RK3 schemes, since one probably has to explicitly store all the temporary vectors required by the scheme. Issues related to the `eval` method were also found. As soon as one has to access to a value of a *global accessor*, the syntax is `eval(gl_acc).operator()(indexes)`, while a better one for the user would be `eval(gl_acc(indexes))`. As mentioned before, even though some critics were made, the GT framework is so powerful that it's worth spending some time to figure out all the compilation errors. Of course, we believe that there is a reason for not implementing the above-mentioned suggestions, but in the G4GT context this non user-friendly is not ideal.

[3] in order to avoid to modify the dimension of the storages every time we change the degree, we defined the storage sizes as if discretization degree was equal to 5, storing some zeros in case of lower degree

From the G4GT point of view, essentially all the comments are still valid. A further problem given by the not optimal `Intrepid` documentation arises. This could represent a serious limitation in order to fully exploit all the potential of the library. For instance, selecting a non-defaulted cubature rule is not trivial mainly because of the code documentation.

7 Conclusion

In this report we described the implementation of a DG solver for a relatively simple advection equation. The main library we exploited is GridTools, with additions provided by Intrepid. We showed the numerical results using, for the sake of their validation, a constant advection field. However, a generic periodic field could be set, as the extension to a non linear-in-u flux function. Indeed, only the stencils which compute the flux in the cubature points have to be modified. Even though most of the effort was spent on the non-adaptive case, the flexibility in changing the degree was added. Both the implementations give coherent results, especially when using a cosine-bell function as initial condition. The numerical oscillations do not represent a major problem, as long as a flux limiting technique is implemented.
We also provided a performance analysis using the Roofline model. The simplifying assumption about the cache hierarchy gave strange results, at least using the GPU. However, at least in terms of required computational time, the results were satisfactory. A different behavior was found with a CPU architecture, where the performances are below the roofline even in a simple matrix-vector multiplication kernel. As a further investigation, performance analysis tools can be used in order to obtain more precise information. Moreover, a scalability analysis could be performed, especially in the adaptive code. Indeed, it may happen that some load unbalance is found, due to a different basis cardinality inside the elements. However, since the computed performances are still quite satisfactory, we believe that this possible bad scalability is not a major concern.
Except for the fixes mentioned in section 6, we believe that the most natural extension of the code is towards the Shallow Water Equations on the sphere, as soon as a full support for a spherical geometry is developed. Indeed, they could represent a good benchmark case for the analysis of atmospheric phenomena. In view of such an implementation, we believe that the structure of the code can be still used. The main changes are on the computation of the differential operators in spherical geometry (so that one has to modify the functors where derivatives play a role) and possibly on the geometric map, which provides the transformation from the physical to the reference element. Finally, due to the singularities at the poles, one may need to modify some boundary conditions. We believe that these changes can be implemented in a reasonably easy way.

References

[1] J.Hesthaven, T. Warburton. *Nodal Discontinuous Galerkin Methods: algorithms, analysis and applications.* Springer Science & Business Media, 2007.

[2] S. Carcano. *Finite volume methods and Discontinuous Galerkin methods for the numerical modeling of multiphase gas-particle flows.* Doctoral dissertation, 2014.

[3] `http://mathworld.wolfram.com/SphericalCoordinates.html`

[4] D.L. Williamson, J.B. Drake, J.J. Hack, R. Jacob, P.N. Swarztrauber. *A standard test for numerical approximation to the shallow water equations in spherical geometry.* Journal of Computational Physics, 1992.

[5] S. Williams, A. Waterman, D. Patterson. *Roofline: an insightful visual performance model for multicore architecture*, 2009.

[6] `https://ark.intel.com/it/products/91316/Intel-Xeon-Processor-E5-2695-v4-45M-Cache-2_10-GHz`

[7] Data from P. Messmer (Nvidia).