Tenth International Conference on Computational Fluid Dynamics (ICCFD10), Barcelona, Spain, July 9-13, 2018

ICCFD10-307

Efficient implementation of Flux Reconstruction schemes for the simulation of compressible viscous flows on Graphics Processing Units

F. Gisbert[**,†], M. Bolinches-Gisbert[*], J. Pueblas[**] and R. Corral[*,**]
Corresponding author: fernando.gisbert@itpaero.com

[**] ITP Aero, Spain.
[†] Universitat Rovira i Virgili, Spain.
[*] Universidad Politécnica de Madrid, Spain.

Abstract

This paper describes the strategies followed to implement a nodal-based Discontinuous Galerkin Flux Reconstruction solver (DGFR) for the compressible Navier-Stokes equations on hybrid unstructured grids, that is going to be executed on multiple Graphics Processing Units (GPUs). DGFR schemes exhibit some features, such as the strong data locality, that make them good candidates to be executed efficiently on a GPU. The solver has been implemented using a mixture of C++ and OpenCL. The implementation details of the most time consuming parts of the solver are presented. It is demonstrated that the use of the GPU local memory has a strong impact on the solver performance. When properly adjusted, it is shown that all the computationally relevant parts of the solver are memory bound for the GPUs used in this work. The parallel efficiency of the code is measured running first on a single GPU and then running on multiple GPUs of a cluster. In the first case, the efficiency of the GPU does not depend on the case size for large enough cases. However, when the number of mesh cells decreases below a certain threshold, the GPU performance decrease is substantial. When multiple GPUS are used in parallel, the communication between them is carried using MPI. The penalty due to the communication of data between GPUs has a large detrimental effect on the parallel speed-up, even in case the communications and computations overlap. The resulting solver is fast and precise enough to be used as an industrial tool for analysis for common turbomachinery problems.

1 Introduction

The development of high order methods for CFD has seen major advances in the last two decades, and are considered a promising candidate to achieve better numerical solution accuracy at an affordable computational cost. However, to gain general acceptance as analysis and design tools for industrial environments, these methods must overcome two major obstacles. On the one hand, the degree of robustness of the method must be comparable to that of current second order discretizations. On the other hand, the cost of obtaining a solution to the numerical problem must not be prohibitive. The development of Discontinuous Galerkin (DG) discretizations has advanced a lot on the first front. Initially proposed by Reed and Hill[1] as a numerical method for the neutron transport equation, its theoretical foundations were laid by Cockburn et al. (see for instance [2]). Popular variants of the original DG formulation are the nodal DG, proposed by Hesthaven and Warburton[3], and the spectral difference method (SD) of Kopriva and Kolias[3]. Huynh[4] proposed a framework to unify both formulations under the umbrella of the so called Flux Reconstruction (FR) method. The formulation allows to recover both nodal DG and SD schemes for the linear advection equation. The scheme was later extended to include diffusion terms[5]. Wang et al. extended Huynh's formulation[6], which was initially developed for 1D and quadrilateral elements, to triangles in 2D, and later to 3D mixed grids[7]. The resulting method, known as correction procedure via reconstruction (CPR), is the one used in this work.

Proves of its stability for whatever order of accuracy have been provided[8], and the numerical behaviour of the resulting schemes has been thoroughly studied[9, 10, 11, 12].

The use of a Graphical Processing Unit (GPU) as a general purpose processor has surged in the last decade. The popularization of programming languages that easily expose the inherent parallelism of the GPU hardware, such as CUDA[13] or OpenCL[14], and the ever increasing computational power of these architectures makes the implementation of CFD solvers on these platforms very convenient. GPUs base their computing performance in the simultaneous use of many cores, of the order of thousands. The efficient use of these cores, however, is only achieved when all of them execute the same piece of code over sets of evenly spaced input data. Embarrassingly parallel applications, i.e., those applications where the data communicated between threads is minimal, take maximum advantage of the particularities of the GPU hardware. It turns out that DG solvers largely fit into that category, therefore several examples of GPU implementations exist in the literature. Klöckner[15] was the first one to implement a GPU version of a DG solver for the resolution of the Maxwell equations, demonstrating the superior performances of the resulting code when compared with an equivalent solver executed on CPUs. The first FR implementation for hybrid unstructured grids on that hardware was presented in 2011[16], showing similar speed-ups relative to the CPU execution. Since then, the number of DG and FR solvers has been growing steadily[17, 18, 19, 20, 21]. Furthermore, Vermeire et al.[22] demonstrate that the GPU implementation of their pyFR[19] solver not only is faster and more precise than other commercial solvers, but it also consumes less computational resources to obtain the solution.

In this paper we present an implementation of a nodal DGFR solver for hybrid unstructured grids as part of the Mu^2s^2T suite of ITP Aero in-house CFD solvers[23]. It has been implemented using a mixture of C++ and OpenCL. The use of C++ templates within OpenCL is supported, therefore we can choose the solver precision at run time. The results presented in this work are obtained running with single precision. MPI has been used to allow the parallel execution on multiple GPUs. The solver employs state-of-the-art programming techniques to fully exploit the computing capabilities of the GPUs. The hardware that has been used for this work is the GeForce GTX1080Ti GPU, connected to the CPU by a PCI-e 3.0 bus at 16 GB/s. There are 20 computing nodes with 10 GPUs each, connected with a two ports Infiniband EDR network at 100 Gb/s per port. First, the equations of the FR scheme are presented for 1D, then it is extended to multiple dimensions, and particularised for prisms and hexahedra. The implementation of the resulting operators is commented in section 3, and some results for a single GPU execution are discussed. Next, the multi-GPU algorithms are described, and the factors affecting the parallel performance are analyzed. Finally, some test cases are presented, showing the solver capabilities.

2 The Flux Reconstruction scheme

We present here a summary of the FR and CPR formulations. The detailed discussions can be found in the articles by Huynh[4, 5] and Wang et al.[6, 7].

2.1 One-dimensional Flux Reconstruction

The FR method was first introduced by Huynh[4] for the solution of conservation equations. We summarize the method here for equations of the form:

$$\frac{\partial U}{\partial t} + \nabla \cdot F = 0 \tag{1}$$

Consider a 1D integration domain $\Omega = [0, L]$ discretized in N elements with the ith element defined by $V_i = [x_i, x_{i+1}]$. For each element we define $p + 1$ solution points (SPs) at which the state variable U is evaluated. Given these SPs the solution is approximated using Lagrange polynomials as

$$U(x,t) \approx U_i^h(x,t) = \sum_{j=1}^{p+1} U_{ij}^h(t) L_j(x) \tag{2}$$

where U_{ij}^h is the solution at SP j of element i, and $L_j(x)$ is the Lagrange polynomial with value 1 at $x = x_j$.

The flux function is defined in an analogous manner as:

$$F\left(U^h\right) \approx F_i^D\left(x,t\right) = \sum_{j=1}^{p+1} F\left(U_{ij}^h\right) L_j\left(x\right) \tag{3}$$

Since the solution is in general discontinuous across element interfaces we now seek to reconstruct the flux function in a way that the reconstructed flux F^C possesses the following properties:

1. It is of degree $p+1$, that is, one order higher than F^D
2. It is close to F^D, in the sense that for certain norm $\left\|F^C - F^D\right\|$ approaches zero.
3. The value of F^C at the element interfaces is a common value F^I between the two neighboring cells.

The final corrected flux is of the form $F_i^C\left(x\right) = F_i^D\left(x\right) + \gamma_i\left(x\right)$ where $\gamma_i\left(x\right)$ is the correction flux function and is defined as:

$$\gamma_i\left(x\right) = \left[F_L^I - F^D\left(x_i\right)\right] g_L\left(x\right) + \left[F_R^I - F^D\left(x_{i+1}\right)\right] g_R\left(x\right) \tag{4}$$

where F_L^I and F_R^I are the left and right interface fluxes and $g_L\left(x\right)$ and $g_R\left(x\right)$are the left and right correction functions. As noted in the original paper by Huynh [24] the choice of correction functions determines the properties of the discretisation. In our implementation, the left and right Radau polynomials are chosen. This is because more accurate solutions are recovered, albeit with a somewhat more restrictive CFL condition. This choice of correction functions recover the nodal-based discontinuous Galerkin formulation for the linear advection equation.

For convenience the spatial dimensions are mapped onto a standard element $\xi = [-1,1]$. The final discretized form of equation 1 is of the form:

$$\frac{\partial J \cdot U}{\partial t} + \sum_{j=1}^{p+1} F\left(U_{ij}^h\right) \frac{\mathrm{d}L_j}{\mathrm{d}\xi}\left(\xi\right) + \left[F_L^I - F^D\left(-1\right)\right] \frac{\mathrm{d}g_L}{\mathrm{d}\xi}\left(\xi\right) + \left[F_R^I - F^D\left(1\right)\right] \frac{\mathrm{d}g_R}{\mathrm{d}\xi}\left(\xi\right) = 0 \tag{5}$$

where $J = dx/d\xi$ is the element Jacobian, that relates the physical and standard coordinate systems. The resulting ODE is advanced in time with a time integration scheme. The one used throughout this work is the 4 stage fourth order Runge Kutta.

2.2 Extension to multiple dimensions

The Euler equations in 3D are

Since all derivative operations are made on the standard element $(\xi, \eta\zeta) \in [-1,1]$, equation 1 is rewritten using this coordinate system:

$$\frac{\partial \left|J\right| \cdot U}{\partial t} + \frac{\partial \widetilde{F}}{\partial \xi} + \frac{\partial \widetilde{G}}{\partial \eta} + \frac{\partial \widetilde{H}}{\partial \zeta} = 0 \tag{6}$$

with

$$\begin{aligned} \tilde{F}\left(U\right) &= \left|J\right|\left(\xi_x F + \xi_y G + \xi_z H\right) \\ \tilde{G}\left(U\right) &= \left|J\right|\left(\eta_x F + \eta_y G + \eta_z H\right) \\ \widetilde{H}(U) &= \left|J\right|\left(\zeta_x F + \zeta_y G + \zeta_z H\right) \end{aligned} \tag{7}$$

The relation between physical and standard coordinates is given by

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \sum_{i=1}^{K} M_i\left(\xi,\eta,\zeta\right) \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

where K is the number of points in the element. The Jacobian matrix J is

$$J = \begin{bmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{bmatrix},$$

whose inverse is

$$J^{-1} = \begin{bmatrix} \xi_x & \xi_y & \xi_z \\ \eta_x & \eta_y & \eta_z \\ \zeta_x & \zeta_y & \zeta_z \end{bmatrix}$$

and$|J|$ is the Jacobian determinant.

The extension to multiple dimensions of the one-dimensional element described above is readily done for quadrilateral and hexahedral elements as tensor product of operations in multiple dimensions. The resulting formulation is included here for completeness. The corrected fluxes are expressed as:

$$\begin{aligned}
\widetilde{F}(\xi,\eta,\zeta) &= \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\widetilde{F}_{ijk}L_i(\xi)L_j(\eta)L_k(\zeta) + \left[\widetilde{F}_W^I - \widetilde{F}(-1,\eta,\zeta)\right]g_L(\xi) + \left[\widetilde{F}_E^I - \widetilde{F}(1,\eta,\zeta)\right]g_R(\xi) \\
\widetilde{G}(\xi,\eta,\zeta) &= \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\widetilde{G}_{ijk}L_i(\xi)L_j(\eta)L_k(\zeta) + \left[\widetilde{G}_S^I - \widetilde{G}(\xi,-1,\zeta)\right]g_L(\eta) + \left[\widetilde{G}_N^I - \widetilde{G}(\xi,1,\zeta)\right]g_R(\eta) \quad (8) \\
\widetilde{H}(\xi,\eta,\zeta) &= \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\widetilde{H}_{ijk}L_i(\xi)L_j(\eta)L_k(\zeta) + \left[\widetilde{H}_B^I - \widetilde{G}(\xi,\eta-1)\right]g_L(\zeta) + \left[\widetilde{G}_T^I - \widetilde{G}(\xi,\eta,1)\right]g_R(\zeta)
\end{aligned}$$

where the subscripts W, E, S, N, B and T denote west, east, south, north, bottom and top hexahedra faces. Applying eq. (6) to the expressions above we obtain the discretized equation for the quadrilateral or the hexahedral element

$$\begin{aligned}
\frac{\partial |J|\cdot U}{\partial t} &+ \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\widetilde{F}_{ijk}L_i'(\xi)L_j(\eta)L_k(\zeta) + \left[\widetilde{F}_W^I - \widetilde{F}(-1,\eta,\zeta)\right]g_L'(\xi) + \left[\widetilde{F}_E^I - \widetilde{F}(1,\eta,\zeta)\right]g_R'(\xi) + \\
&+ \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\widetilde{G}_{ijk}L_i(\xi)L_j'(\eta)L_k(\zeta) + \left[\widetilde{G}_S^I - \widetilde{G}(\xi,-1,\zeta)\right]g_L'(\eta) + \left[\widetilde{G}_N^I - \widetilde{G}(\xi,1,\zeta)\right]g_R'(\eta) + \\
&+ \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\widetilde{H}_{ijk}L_i(\xi)L_j(\eta)L_k'(\zeta) + \left[\widetilde{H}_B^I - \widetilde{H}(\xi,\eta-1)\right]g_L'(\zeta) + \left[\widetilde{H}_T^I - \widetilde{H}(\xi,\eta,1)\right]g_R'(\zeta) = 0 \quad (9)
\end{aligned}$$

which is marched in time with the RK4 scheme.

2.2.1 Extension to triangular and prismatic elements.

For triangular and prismatic elements we follow the Correction Procedure by Reconstruction (CPR) approach developed by Haga et al.[7]. What follows are the final forms of the equations. For further insight into how to derive them the reader is referred to the original work.

The prismatic element is built using a tensor product of the polynomial interpolation and correction for triangles in the (ξ,η) directions and the one-dimensional operations of eq. (5) in the ζ direction.

The interpolation of variables and fluxes is then a tensor product of 2D and 1D Lagrange polynomials:

$$\widetilde{F}(\xi,\eta,\zeta) = \sum_{j=0}^{N_T}\sum_{k=0}^{p}\widetilde{F}_{jk}L_j^T(\xi,\eta)L_k(\zeta)$$

where $N_T = p(p+1)/2$ is the number of triangle points and the super-index T denotes the Lagrange polynomial for the triangle.

The correction term equivalent to g'_L and g'_R derivatives of eq. (9) for each triangle is given by a sum of contributions from all boundary nodes:

$$\frac{1}{\left|\widetilde{V}_{Tri}\right|}\sum_{f\in\partial\widetilde{V}}\sum_{l=0}^{p}\alpha_{j,f,l}\left[\widetilde{F}^n\right]_{fl}\widetilde{S}_f \tag{10}$$

where $\left|\widetilde{V}_{Tri}\right|$ is the area of the standard triangle, f is each one of the edges of the triangle, $\widetilde{S}_f$ is the edge length and $\left[\widetilde{F}^n\right]_{fl}$ is the flux jump at each edge point. Finally, $\alpha_{j,f,l}$ are the so called lifting coefficients, which are constant, independent of the solution or the shape of the triangle. The expression for the prism just adds the third dimension to those expressions:

$$\begin{aligned}\frac{\partial|J|\cdot U}{\partial t} &+ \sum_{j=0}^{N_T}\sum_{k=0}^{p}\widetilde{F}_{jk}\frac{\partial L_j^T(\xi,\eta)}{\partial\xi}L_k(\zeta)+\\ &+ \sum_{j=0}^{N_T}\sum_{k=0}^{p}\widetilde{G}_{jk}\frac{\partial L_j^T(\xi,\eta)}{\partial\eta}L_k(\zeta)+\\ &+ \frac{1}{\left|\widetilde{V}_{Tri}\right|}\sum_{f\in\partial\widetilde{V}}\sum_{l=0}^{p}\alpha_{j,f,l}\left[\widetilde{F}^n(\xi_{f,l},\eta_{f,l},\zeta_m)\right]_{fl}\widetilde{S}_f\\ &+ \sum_{j=0}^{N_T}\sum_{k=0}^{p}\widetilde{H}_{jk}L_j^T(\xi,\eta)L'_k(\zeta)+\left[\widetilde{H}_B^I-\widetilde{H}(\xi,\eta-1)\right]g'_L(\zeta)+\left[\widetilde{H}_T^I-\widetilde{H}(\xi,\eta,1)\right]g'_R(\zeta)=0\end{aligned} \tag{11}$$

2.3 Discretisation of the Navier-Stokes equations

The differential form of the Navier-Stokes equations is:

$$\frac{\partial\mathbf{U}}{\partial t}+\frac{\partial\mathbf{f}}{\partial x}+\frac{\partial\mathbf{g}}{\partial y}+\frac{\partial\mathbf{h}}{\partial z}=\frac{\partial\mathbf{F}}{\partial x}+\frac{\partial\mathbf{G}}{\partial y}+\frac{\partial\mathbf{H}}{\partial z} \tag{12}$$

being $\mathbf{U}=\left[\begin{array}{ccccc}\rho & \rho u & \rho v & \rho w & \rho E\end{array}\right]^T$ the conservative variables,

$$\left[\begin{array}{ccc}\mathbf{f} & \mathbf{g} & \mathbf{h}\end{array}\right]=\left[\begin{array}{ccc}\rho u & \rho v & \rho w\\ \rho u^2+p & \rho uv & \rho uw\\ \rho uv & \rho v^2+p & \rho vw\\ \rho uw & \rho vw & \rho w^2+p\\ u(\rho E+p) & v(\rho E+p) & w(\rho E+p)\end{array}\right] \tag{13}$$

the inviscid fluxes and

$$\left[\begin{array}{ccc}\mathbf{F} & \mathbf{G} & \mathbf{H}\end{array}\right]=\left[\begin{array}{ccc}0 & 0 & 0\\ \tau_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{xy} & \tau_{yy} & \tau_{yz}\\ \tau_{xz} & \tau_{yz} & \tau_{zz}\\ u\tau_{xx}+v\tau_{xy}+w\tau_{xz}-q_x & u\tau_{xy}+v\tau_{yy}+w\tau_{yz}-q_y & u\tau_{xz}+v\tau_{yz}+w\tau_{zz}-q_z\end{array}\right] \tag{14}$$

the viscous terms. We use the perfect gas assumption, therefore the gas state equation $p=\rho R_g T$ holds. Besides,

$$E=C_vT+\frac{1}{2}q^2,$$

where $q^2 = u^2 + v^2 + w^2$, and C_v is the specific heat at constant volume. The pressure is expressed as a function of the conservative variables:

$$p = \rho\left(\gamma - 1\right)\left(E - \frac{1}{2}q^2\right),$$

where $\gamma = C_p/C_v$ is the relation between the specific heats, at constant pressure and constant volume. The viscous stress tensor expression is

$$\tau_{ij} = \mu\left(\partial_i v_j + \partial_j v_i\right) - \frac{2}{3}\mu\left(\nabla \cdot \mathbf{v}\right)\delta_{ij} \tag{15}$$

and the heat flux vector is expressed using the Fourier's law:

$$\mathbf{q} = -k\nabla T \tag{16}$$

In the preceding equations (15) and (16) the constants μ and k are the laminar viscosity and the conductivity, respectively:The expression of the laminar viscosity is given by the Sutherland law

$$\mu = \frac{1.458 \cdot 10^{-6} T^{3/2}}{T + 110.4}$$

and the thermal conductivity is

$$k = \frac{\mu C_p}{Pr}$$

where the Prandtl number for air is 0.72.

Eq. (12) is analogous to eq. (6), but expressed in physical rather than in standard coordinates. However, in the viscous fluxes there is a dependency on the gradient of the primitive variables, therefore we must provide an expression for the computation of the gradient as well.

The gradient can be expressed as a function of the derivatives in the standard coordinates and the Jacobian. For each primitive variable:

$$\frac{\partial V}{\partial x_i} = \left(J^{-1}\right)^T \frac{\partial V}{\partial \xi_i} \tag{17}$$

where $x_i = x, y, z$, and $\xi_i = \xi, \eta, \zeta$. The derivative for each component of the standard coordinates is evaluated using the desired component from the expression from eqs. (9) and (11) and substituting the fluxes by the primitive variables. The common value at the flux points in this case is the average value of the primitive variables at both sides of the element. For the hexahedra the gradient expression is then:

$$\frac{\partial V}{\partial \xi_i} = \sum_{i=0}^{p} V_{ijk} L_i'\left(\xi_i\right) + \frac{V_{Lneigh} - V_L}{2} g_L' + \frac{V_{Rneigh} - V_R}{2} g_R' \tag{18}$$

and for the prism

$$\begin{aligned} \frac{\partial V}{\partial\left(\xi, \eta\right)} &= \sum_{j=0}^{N_T} V_{jk} \frac{\partial L_j^T\left(\xi, \eta\right)}{\partial\left(\xi, \eta\right)} + \frac{1}{\left|\widetilde{V}_{Tri}\right|} \sum_{f \in \partial \widetilde{V}} \sum_{l=0}^{p} \alpha_{j,f,l} \frac{V_{f,l,neigh} - V_{f,l}}{2} \\ \frac{\partial V}{\partial \zeta} &= \sum_{i=0}^{p} V_{ijk} L_i'\left(\zeta_i\right) + \frac{V_{Dneigh} - V_D}{2} g_L' + \frac{V_{Uneigh} - V_U}{2} g_R' \end{aligned} \tag{19}$$

Finally, the common fluxes at the face points must also be evaluated. For the convective term, the common flux is computed with a Riemann solver[25]. For the viscous terms, a common viscous flux is evaluated using the average of primitive variables and gradients.

2.3.1 Solution points positions

The previous equations do not assume any particular position for the solution points. In this work, the Gauss-Lobatto points have been used for the quadrilateral and hexahedral elements, and the triangle point distributions of Warburton[26] are employed for triangles and prisms. The flux and the solution points are overlapped, therefore there is no need to implement the interpolation mechanism to obtain the values at the flux points using the volumetric points, reducing the computational needs of the solver. However, Jameson et al.[27] suggest that this point distribution is not optimal, and that the volume and flux points should be placed at different locations to prevent instabilities caused by aliasing errors. The effect of these distribution is not addressed in this work and will be subject of further research.

3 Implementation of the operators

The formulation presented in the previous section to integrate the Navier-Stokes equations can be summarized as follows:

1. Evaluate the derivative of the primitive variables, i.e., the first term of the right hand side of eqs. (18) and (19). This is a matrix-vector product, where the matrix elements are the derivatives of the Lagrange polynomials for each solution point: $L_i'(\xi_j)$.

2. Correct the derivative with the correction functions and the average value of the primitive variables at the face points, i.e., the second term of the right hand side of those equations. This is again a matrix-vector product, but now the matrix is given either by the $g'_{L,R}$ functions for the hexahedra or by the lifting coefficients of eq. (10), and the vector has the differences between primitive variables at both sides of the face points.

3. Pass the gradient to physical coordinates with eq. (17).

4. Compute the fluxes for each solution point with eqs. (13) and (14).

5. Evaluate the derivatives of the fluxes. These are the terms with the Lagrange polynomial derivatives of eqs. (9) and (11). Like step 1, this is again a matrix-vector product, but using the fluxes instead of the variables.

6. Correct the fluxes derivatives with the correction functions and the values of the fluxes at the face points. These are the terms with the correction functions of eqs. (9) and (11). This matrix-vector is analogous to that of step 2, but in this case the vector contains the common fluxes through the face points.

7. Enforce boundary conditions.

8. Advance to the next stage of the RK4 scheme and go to 1.

From the steps described, only the boundary contributions to gradient (2) and fluxes (6) involve data from outside the element, i.e., those needed to compute the common fluxes. The rest of steps are purely local, either involving just the data from each solution point (steps 3, 4 and 8) or data from all nodes of the each mesh cell (steps 1 and 5). And even in steps 2 and 6 a large part of the data that needs to be gathered to evaluate the contributions to gradient and fluxes belongs to each mesh element. That strong data locality is very convenient to obtain a good parallel efficiency when executing the solver in massively parallel processing units, because all the computing threads can fetch the data from memory, perform all the operations needed and write the result back to memory without the need to communicate between them. This type of parallelism is ideally suited for GPUs. Therefore, the DGFR solver has been coded using a mixture of C++ and OpenCL, which is a C-99 with some extension to express parallelism. It can be executed on CPUs as well as GPUs and other many-core hardware thanks to the use of OpenCL language, which is supported not only by NVIDIA but also by other major hardware vendors such as Intel and AMD. Moreover, an external modulus has been implemented to automatically translate OpenCL code into CUDA, taking advantage of the similarities between the OpenCL and CUDA API programming models. That

enables the use of NVIDIA debugging and profiling tools, which are very useful when trying to optimize the implementation. All the bandwidth measurements presented below are obtained running nvprof[28].

The GPUs base their superior computing performance in packing many cores for every processor and communicate them with the processor memory through a high bandwidth bus. For instance, the GPU used in this work, the NVIDIA GeForce GTX1080Ti, has 28 streaming multiprocessors (SM), with 128 cores each. Each SM can execute up to 2048 simultaneous threads. The bandwidth between the global memory and the processors is 484 GB/s. When the solver is executed, each function that is executed on a GPU, also called kernel, is automatically spread into a specified number of threads, that are in turn grouped in thread blocks whose size is also specified at run time. For an efficient use of the GPU processors, the threads must access the memory with a regular enough pattern. The so-called coalesced memory access provides the best bandwidth. Other non-optimal memory accesses need more clock cycles to fetch the data from memory and consequently the effective bandwidth is much lower. Even though the GPU cache memory alleviates this problem somehow, coalesced memory accesses are key to achieving good performances.

3.1 Point-based kernels

In those kernels where the data used belongs only to each solution point the coalesced memory accesses are easy to obtain. The only thing to take into account is that the bandwidth is worse with the so-called array of structure (AoS) data layouts than with structure of arrays (SoA) layouts. I.e., in case we are solving the Navier-Stokes equations, we have 5 conservative variables per node. If the mesh has N_p solution points, it is better to place ρ in the first N_p positions, then ρu, etc. and access them as

$$\begin{aligned} \rho_i &= U[i] \\ \rho u_i &= U[i+N_p] \\ \rho v_i &= U[i+2N_p] \\ \rho w_i &= U[i+3N_p] \\ \rho E_i &= U[i+4N_p] \end{aligned}$$

than to pack $\begin{pmatrix} \rho & \rho u & \rho v & \rho w & \rho E \end{pmatrix}$ for each node and access them the way it is optimal for CPUs:

$$\begin{aligned} \rho_i &= U[5i] \\ \rho u_i &= U[5i+1] \\ \rho v_i &= U[5i+2] \\ \rho w_i &= U[5i+3] \\ \rho E_i &= U[5i+4] \end{aligned}$$

With the SoA memory access, optimal bandwidth is obtained for kernels implemented to execute steps 3, 4 and 8. By optimal bandwidth we understand the maximum achievable bandwidth, which is not always the peak but the measured bandwidth in benchmarks such as STREAM[29], which is around 80% of the peak value. Thus, if the peak bandwidth of the GeForceGTX1080Ti is 484 GB/s, the measured bandwidth for the flux evaluation (step 4) is 358 GB/s, a 74% of the peak. It must be observed that the performance of this kernel, and this is also true for the kernels of steps 3 and 8, is entirely determined by the rate at which the memory is accessed, i.e., the processor remains idle around 90% of the time waiting for data to be fetched from memory.

3.2 Cell-based kernels with in-cell data access

Kernels that implement steps 1 and 5 use data from some if not all cell solution points simultaneously. Since this kernel performs several matrix-vector operations, one option would be to use linear algebra libraries like cuBLAS[30] in CUDA or ViennaCL[31] in OpenCL. However, several authors[15, 16] have compared the performance of such libraries with kernels tailored for the specific data layout, and the latter generally outperform the former. In those cases, the efficient execution of these kernels is achieved using local or shared GPU memory, in OpenCL or CUDA terminology respectively. As the name suggests, shared memory

is a portion of GPU memory that is shared among all threads of a thread group. The shared memory is physically located near the processor. It serves as a sort of cache memory that can be accessed at a faster rate. The data placed there can be reused with much less cost when compared with a load from global memory. The drawback is that it has a reduced size (48 KB, currently), therefore we have to be cautious selecting which data is placed in that memory, since we can run out of it rather quickly.

The strategy followed for these kernels is to assign each cell to a thread block. The maximum number of threads per block is 1024. That allows having $p = 10$ hexahedra and $p = 12$ prisms. The amount of shared data is different for hexahedra and prism.

The former have derivative matrices $D_{p\times p}$ in each direction, which could be different if different collocation strategies are employed for each direction, but we assume them to be equal for each direction. Then the shared data are the coefficients of the matrix $D_{p\times p}$ and the values of the primitive variables or fluxes at each solution point. That is $p^2 + p^3$ shared data. In single precision, each datum is stored in 4 bytes, and 8 bytes in double precision. For $p = 10$ that amounts to 4400 bytes in single precision and 8800 bytes in double precision, which is below the 48 KB limit. Therefore in this case the limiting factor is the number of threads.

In prisms, the derivative matrices for the ξ and η are $D_{N_T\times N_T}$, and the derivative in ζ direction has $D_{p\times p}$ elements, as in the prism case. If the number of solution points is $p^2\left(p+1\right)/2$ the total amount of shared data is $p^2\left(p+1\right)^2/4 + p^2 + p^2\left(p+1\right)/2$. For single precision this limits the order of the element to $p = 14$ and $p = 11$ for single and double precision respectively.

The sequence of operations is:

- Execute as many thread blocks as cells. For each cell block, use as many threads as cell solution points. Then for each block:

    1. Fetch the data from global memory and put it in the shared memory arrays. Since the data is loaded in order and there are no repeated nodes between cells, the data loading is equivalent to the optimal one of previous section.
    2. Each thread evaluates its corresponding row of the $D \cdot U$ or $D \cdot F$ matrix vector product for each variable. The memory access here is not ordered, but since it is contained in the shared memory, the access is not a limiting factor.
    3. Write the result to the corresponding global memory position. The data storage is also done with an optimal data arrangement.

The memory bandwidth for these kernels is around 350 GB/s, which is near the 75% peak value reported for the point-based kernels, a clear sign of the correct memory access pattern. The kernels are also memory bound, i.e., the cost of accessing to data and writing the result is in general larger than the cost of executing the kernel operations, even though the gap is smaller and for larger values of p the opposite could occur.

3.3 Cell-based kernels with adjacent cells data access

The kernels for steps 2 and 5 make use of data from adjacent cells in addition to the in-cell data to evaluate the common values at the face points. The access pattern for these data will hardly be coalesced, since the memory positions from adjacent cell points are seldom consecutive to those of in-cell data. Some authors[16] pre-compute the face points common values and then convert the kernel into one with in-cell data only. However, this has only displaced the problem to another kernel, namely, the one that must compute the common values, and does not really address the limiting effect of non-coalesced data accesses. Some gain could be obtained from having a simpler kernel, but there is a clear penalty due to having to write and then read the common values. By computing the common values at the same kernel where the derivative correction is applied the extra write and read operations are avoided, but the resulting kernel is more complex. This is the approach taken in this work.

In these kernels there is the possibility to load the in-cell data to shared memory to speed-up at least those data access. However, the amount of shared data becomes considerable. If we take for instance the flux derivative correction (step 6) for the hexahedra, and we pre-load all the data needed to compute the common fluxes, we must load the primitive variables, the three gradient components and the Jacobi matrix for each face point, and we must store the common flux. That amounts to 34 values for each of

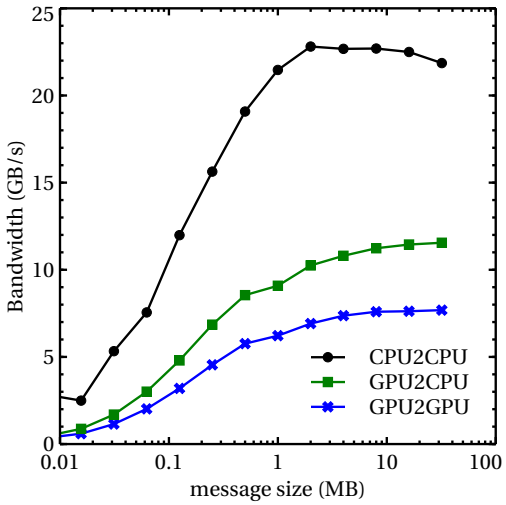

Figure 1: Bandwidths of GPU to CPU (PCI-e 3.0 at 16 GB/s) and CPU to CPU (bidirectional Infiniband at 100 Gb/s) data transfer for different message sizes

the $p^2 + 4p\,(p-1) + (p-2)^2$ face points. The shared memory limit of 48KB is reached for $p = 8$ in single precision and $p = 6$ for double precision. In the prism, the amount of shared data is

$$34\,[p\,(p+1) + 3\,(p-1)\,(p-2)] + \frac{3}{2}p^2\,(p+1)$$

considering that the $3p^2\,(p+1)\,/2$ lifting coefficients $\alpha_{j,f,l}$ should be kept too. The limit is reached for $p = 9$ in single precision and $p = 7$ for double precision. Given the limitations imposed by the shared memory size, we have decided not to pre-load the in-cell data and to pay the corresponding penalty, which is deemed small in any case due to the use of the GPU own cache memory and to the fact that the data from adjacent cells cannot be optimally loaded anyway, hence the gain would be restricted to half the total global memory fetches. Indeed, the measured bandwidth for these kernels is 330 GB/s for the hexahedra and 282 GB/s for the prisms, which is 68% and 58% of the peak memory bandwidth, which is not catastrophic, but signals that for these kernels there is still room for improvement, given the fact that the kernels are also memory bound.

4 Parallel use of multiple GPUs

In addition to the fine grained parallelism natural to the use of the GPU, when the problem size is large enough and it does not fit in a single GPU, multiple GPUs are used in parallel. The domain is decomposed using the parallel domain decomposition library ParMeTiS[32]. The strategy followed is that every GPU is controlled by an MPI process assigned to a single CPU core. When there are several GPUs sharing a computing node, this may not be the optimal solution, because we are forced to send data via MPI to transfer it from GPU to GPU. If one MPI process controlled all GPUs inside the same computing node, that extra MPI communication could be avoided, and only inter-node communications would be needed. But that comes at the expense of making the code structure more complicated because intra-node and inter-node communications should be treated differently. Moreover, managing that in real environments where the MPI process distribution inside the computing nodes is not known a priori is a task that requires a careful thought.

Therefore, some data residing on the GPU memory must be sent to the GPU memory in charge of computing the adjacent domain. When using OpenCL on NVIDIA hardware, the only option is to send the GPU data to an array located in the CPU memory, and then send the data via MPI, and receive the data sent by other MPI processes, and place it again on the GPU. This is a serious limitation, because the bandwidth for the data transfer between GPU and CPU is given by the PCI-e bus at 16 GB/s, with an effective

bandwidth around 12 GB/s, which is an abysmal difference with the 360 GB/s achievable bandwidth of the GPU memory bus. In addition to the GPU to CPU transfer, the transfer rate via MPI for the Infiniband network is around 22.5 GB/s (90% of the peak value of 25 GB/s for the 100 Gb/s two-port Infiniband). Together, the combined bandwidth, given by

$$\frac{1}{BW} = \frac{1}{BW_{GPU \leftrightarrow CPU}} + \frac{1}{BW_{IB}}$$

is around 7.5 GB/s for large enough message sizes (see figure 1). To minimize the penalty, some mitigation strategies have been implemented.

- Use pinned, or page-locked memory to perform the data transfers between CPU and GPU, which is why the 12 GB/s are obtained. Otherwise the effective bandwidth is almost halved.
- Overlap the GPU to CPU data transfers with the computation, hiding the extra cost of communicating somehow.
- Use MPI non-blocking communications and overlap them with the computations. The latter has a mixed effect, because once OpenCL is instructed to launch a kernel, the control is immediately returned to the CPU, and the OpenCL driver is in charge of managing the queue independently. Therefore, the CPU flow of events and the GPU flow of events do not necessarily coincide, and sometimes the CPU has reached the point where the non-blocking communication termination is reached and the GPU is still executing the operations in between, hence limiting the usefulness of the overlapping.

When the code is treated as CUDA instead of OpenCL there is an alternative to that communication procedure, which is the use of the unified memory addressing to avoid memory transfers between CPU and GPU, or rather to let the CUDA driver take care of that process. With that approach, the bandwidth of the GPU to GPU transfer would be equal to the Infiniband network bandwidth. The tests performed with that option show that the problem is then translated to the kernel that packs the global data to the array that is communicated via MPI. When that array is declared to be in the unified memory, the GPU memory bandwidth when writing to it is no longer close to the optimal 360 GB/s but it is rather similar to the PCI-e bus bandwidth, and the overall performance of the transfer is similar, or even worse for small arrays, than the one obtained with the overlapped data transfers.

The complete sequence for flux evaluation and update with multiple GPUs is then:

1. Gather the parallel boundary primitive variables into an array and pass it from the GPU to the CPU.
2. Evaluate the derivative of the primitive variables (step 1 in section 3).
3. Finish the GPU->CPU transfer and then send the CPU array via MPI to the MPI process of the adjacent domains and receive the parallel boundary data from those processes. Pass the received data to the GPU and scatter them into the corresponding GPU global array. There is no possibility of overlapping here, therefore the communications are started and finished before moving on.
4. Correct the gradient with the correction functions and pass the gradient to physical coordinates (steps 2 and 3 in section 3).
5. Gather the parallel boundary gradients into an array and pass it from the GPU to the CPU.
6. Compute the fluxes for each solution point (step 4 in section 3).
7. Finish the GPU->CPU transfer and then send/receive the data via MPI.
8. Evaluate the derivative of the convective fluxes (step 5a in section 3, because now convective and viscous fluxes are treated separately to have an opportunity for computation/communication overlap).
9. Finish the MPI transfer and place the received data in the GPU array.
10. Evaluate the derivative of the viscous fluxes (step 5b in section 3).

N_c	75^3	64^3	48^3	32^3	16^3	8^3	4^3
$t/t_{ref}/N_c/N_{ref}$	0.995	1	0.996	0.981	0.891	0.537	0.144
$N_{SP}/N_{GPU\,threads}$	470.8	292.6	123.4	36.6	4.6	0.57	0.07

Table 1: Parallel efficiency of the Taylor-Green vortex run on a NVIDIA GTX1080Ti GPU for different hexahedral mesh sizes. $p = 4$, $N_{GPU\,threads} = 28 \times 2048$

11. Finish the CPU->GPU transfer.
12. Correct the fluxes derivative (step 6 of section 3).
13. Enforce boundary conditions.
14. Advance to the next stage of the RK4 scheme and go to 1.

5 Code Benchmarking

The code has been benchmarked using the 3D Taylor-Green vortex[33] as test case. It is a three-dimensional periodic flow whose initial condition is

$$\begin{aligned}
\rho &= \frac{p}{RT_0} \\
u &= V_0 \sin\left(\frac{x}{L}\right)\cos\left(\frac{y}{L}\right)\cos\left(\frac{z}{L}\right) \\
v &= -V_0 \cos\left(\frac{x}{L}\right)\sin\left(\frac{y}{L}\right)\cos\left(\frac{z}{L}\right) \\
w &= 0 \\
p &= p_0 + \frac{\rho_0 V_0^2}{16}\left(\cos\left(\frac{2x}{L}\right)+\cos\left(\frac{2y}{L}\right)\right)\left(\cos\left(\frac{2z}{L}\right)+2\right)
\end{aligned}$$

and then it undergoes a transition to anisotropic turbulence with a subsequent decay. The Reynolds number is

$$Re = \frac{\rho_0 V_0 L}{\mu} = 1600$$

and the Mach number

$$M = \frac{V_0}{c_0} = 0.1$$

is small enough to assume that the flow is incompressible. The Prandtl number $Pr = 0.71$ and the gas $\gamma = C_p/C_v = 1.4$ The characteristic convective time is $t_c = L/V_0$. The computational domain is a cube of dimensions $(x, y, z) \in [-\pi L, \pi L]$, and it has been meshed with hexahedra.

5.1 Single GPU

5.1.1 GPU efficiency

The first test is meant to quantify the parallel efficiency of the GPU as a function of the problem size. To that effect, we run meshes of degree $p = 4$ of varying sizes, up to the maximum size that fits in the 11 GB GPU memory. The results are displayed in table 1, where the non-dimensional time per cell is measured against the number of cells. The non-dimensionalizer is the time per cell that reaches the maximum efficiency, which is 64^3. When the problem size is large enough, that number remains constant, meaning that the GPU parallel efficiency is constant. We observe that for 16^3 the efficiency has a 10% decrease, and for smaller sizes the behaviour is rather poor. This is mainly due to the small occupancy of the GPU. It has been mentioned above that the GPU has 28 SM and each one can execute up to 2048 threads. When the number of solution points N_{SP} is lower than the number of threads the GPU is not used at full capacity, and overheads that

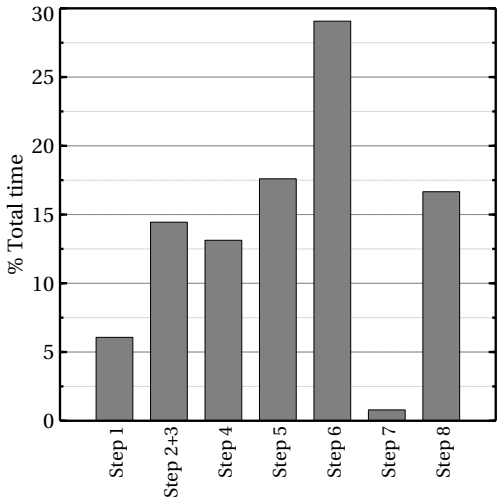

Figure 2: Percentage of time spent executing each of the 8 steps enumerated in section 3 for the Taylor-Green vortex with a 48^3 $p = 4$ hexahedra mesh.

remain hidden when the number of demanded threads is large enough arise. That should be kept in mind when splitting the domain to run on multiple GPUs; if the split size is too small, the GPU will behave like a much slower processor.

5.1.2 Kernel time budget

The percentage of time spent on each of the steps enumerated in section 3 is represented in figure 2 for the 48^3 $p = 4$ mesh. The correction of the fluxes (step 6) is the most time consuming step, followed by the derivation of the fluxes (step 5), the Runge-Kutta operations (step 8) and then the gradient correction and pass to physical (steps 2 and 3), that are performed in a single step to save global memory accesses. Then the convective and viscous flux evaluation (step 4), and finally the derivation of the primitive variables (step 1). The boundary conditions are generally applied over a reduced sub-set of the mesh points and do not account for more than a few percentage points. In this case only periodic boundary conditions are applied, and they consume around 1% of the total time. It is noticeable that the Runge-Kutta consumes a lot of the total time. This is due to the use of single precision arithmetics. In that scenario, care must be taken to avoid accumulating round-off errors in the summation that is produced at every step of the Runge-Kutta method. Therefore, we have employed the Kahan summation algorithm[34] to minimize the summation round-off. The algorithm involves two extra arrays with the size of the conservative variables, that have to be accessed and updated every Runge-Kutta step. It must be emphasized once again that for mesh order $p = 4$ every kernel is memory bound, i.e., the majority of the kernel execution time is spent loading and storing data from the global memory.

5.1.3 Influence of mesh order

The next test is aimed at finding out whether the increase of mesh order affects the solver performance. Figure 3 represents the increase in cost respect to the $p = 4$ baseline case. It is observed that for increasing mesh order, the cost is increased slightly more than the theoretical cubic curve. This is due to the fact that the most time consuming kernels are a 4.5% slower for higher order. The situation is the opposite for mesh order lower than 4.

5.2 Multiple GPUs

The parallel scalability for the benchmark case for a 48^3 mesh is depicted in figure 4. Two factors help understand the pronounced decrease in parallel speed-up for $N_{GPU} > 4$. On the one hand, the small values

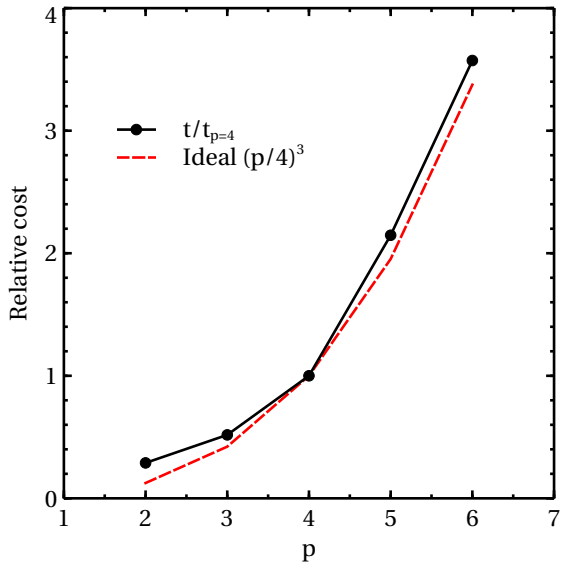

Figure 3: Relative execution cost for different values of p for the Taylor-Green vortex 48^3 hexahedra mesh. The cost increase with p is consistent with the increase in the problem size.

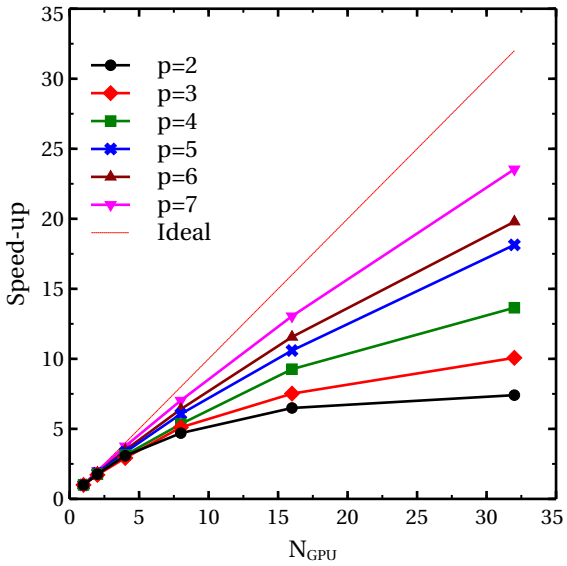

Figure 4: Speed-up of the solver for the Taylor-Green vortex run on a 48^3 mesh of different orders.

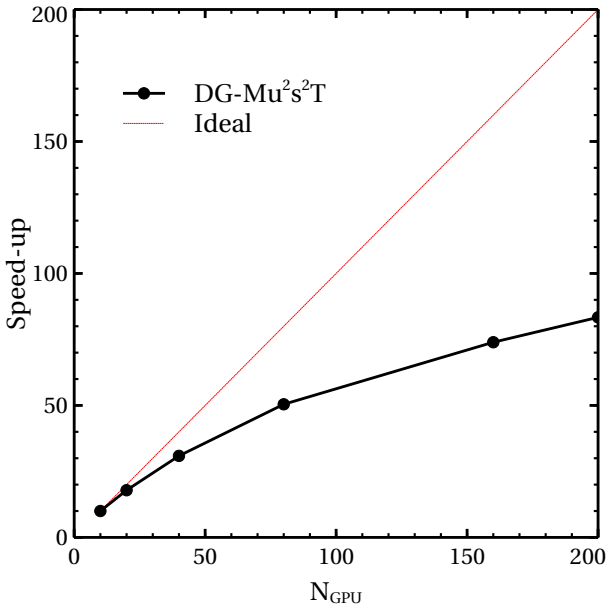

Figure 5: Parallel speed-up of the 128^3 $p = 4$ Taylor-Green cube

of the bandwidth reported in section 4 make the data transfer between domains very expensive compared with the time spent executing the kernels. The benefits of the overlapping fade out quickly as the number of MPI processes grows and the execution time is reduced accordingly. On the other hand, for low mesh orders ($p < 4$) the $N_{SP}/N_{GPU\,threads}$ ratio falls below one, therefore the GPU is not fully occupied and the GPU performance decays abruptly, as explained in table 1. Yet another factor helps explain why the higher order meshes perform better. While the cost of the computation increases with p^3, the size of the parallel frontiers increases with p^2, and then the ratio of computation to communication grows linearly with p. Thus, from the point of view of the parallel efficiency, it is preferable to run with higher mesh orders.

Figure 5 shows the solver scalability for a 128^3 hexahedra mesh used to solve the Taylor-Green vortex. The speed-up quickly deteriorates as the number of GPUs is increased, because the cost of transferring the data from GPU to GPU becomes dominant, even with the use of overlapped data transfers. The simulation has reached the $t = 20t_c$ in 75 minutes using 200 GPUs.

6 Results

6.1 Taylor-Green vortex

The same Taylor-Green vortex case at $Re = 1600$ of the previous section has been meshed with a 128^8 $p = 4$ hexahedral mesh and has been compared against the solution of a 512^3 spectral solver simulation, that is provided as one of the verification cases of the High Order CFD Workshop[35]. The iso-vorticity surfaces for the initial and the $t = 10t_c$ solutions are depicted in figure 6. The comparison of the minus time derivative of the kinetic energy, defined as:

$$\frac{-dE_k}{dt} = -\frac{d}{dt}\left(\frac{1}{\rho_0\Omega}\int_\Omega \rho\frac{\mathbf{u}\cdot\mathbf{u}}{2}d\Omega\right)$$

is presented in figure 7. It is seen that the solver slightly underpredicts the peak value, but the overall agreement with the spectral solution is good. The $t = 20t_c$ has been reached after 18 hours using 10 GPUs.

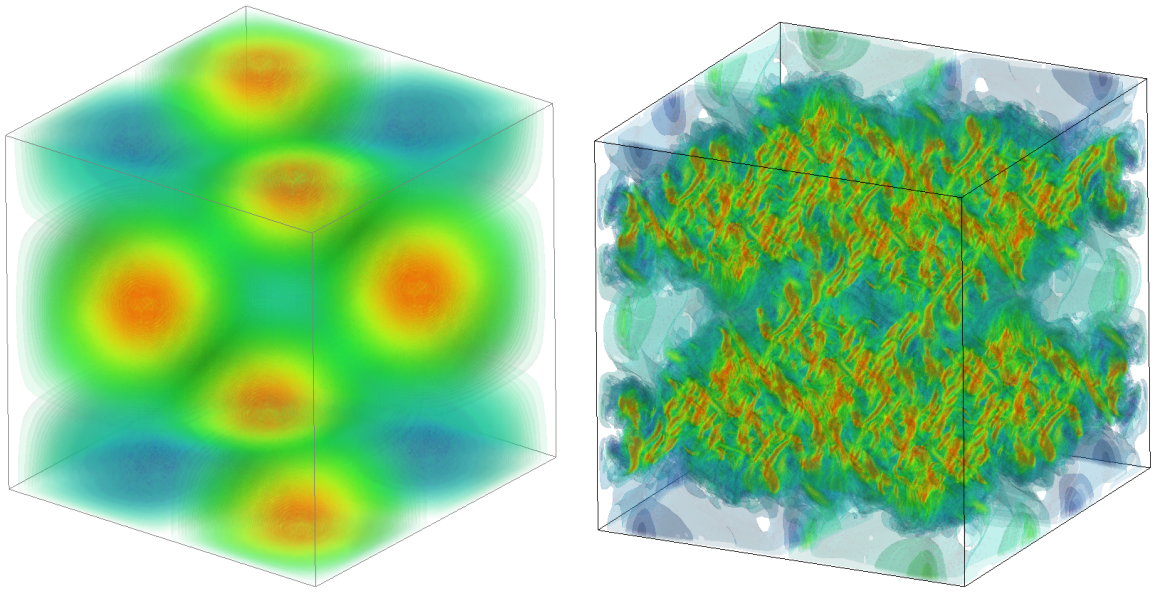

Figure 6: Iso-vorticity surfaces at $t = 0$ (left) and $t = 10t_c$ (right) for the Taylor-Green vortex at Re=1600 with a 128^3 $p = 4$ mesh

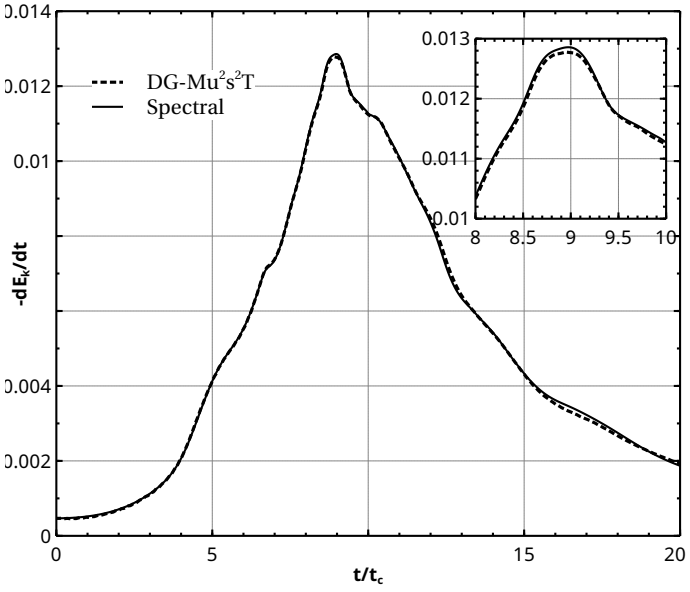

Figure 7: Evolution of $-dE_k/dt$ for the Taylor-Green vortex at $Re = 1600$ with a 128^3 $p = 4$ mesh.

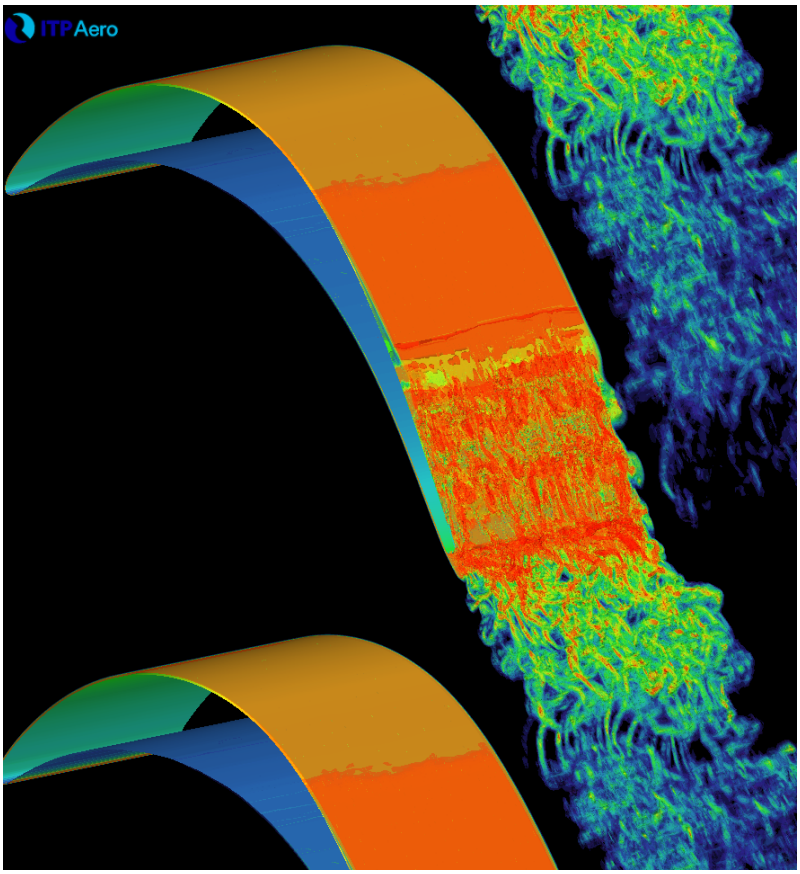

Figure 8: Instantaneous vorticity iso-surfaces for the T106C at Re=140000

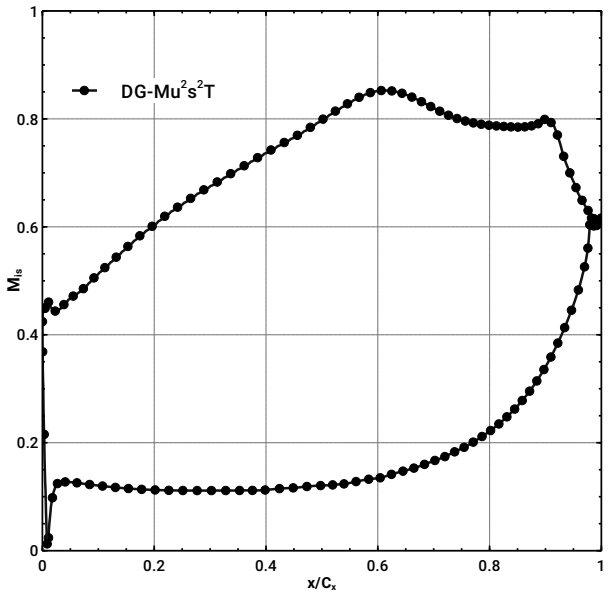

Figure 9: Instantaneous vorticity iso-surface coloured with u velocity component for the T106C at Re=140000

6.2 T106C Lower Pressure Turbine blade

The T106C was used as a validation case for the European project TaTMo, and was tested at the vKI[36] for a set of Reynolds numbers and for different inlet turbulent intensities, to see the effect of those parameters on the size and length of the suction side bubble. We have used the case as a test to see if we are able to simulate the bubble breakdown and the transition to turbulence that it induces. The unstructured hybrid mesh is made of $p = 4$ prisms (910k elements) and hexahedra (430k elements), with 100 elements in the z direction, and a total of 63.8 million points. It has been run on 40 GPUs and the statistics have converged after 44 hours. To perform a proper comparison with the experimental measurements, it would be necessary to define an inlet turbulence level to match the experimental one and also its decay rate. This has not been implemented yet, therefore comparisons with the experiments at this stage are misleading. Figure 8 plots the iso-surfaces of vorticity. It is observed that the suction side bubble shear layer begins to oscillate and promotes the turbulent transition, which in turn forces the bubble reattachment around 90% of the axial chord. The resulting average isentropic Mach distribution along the blade is depicted in figure 9. Even though the simulation parameters are not right, it seems that the solver is capable of reproducing the main physical phenomena of this transitional flow.

7 Conclusions

A three-dimensional high order flux reconstruction solver for the compressible Navier-Stokes equations on hybrid unstructured grids has been presented. The solver can run in multiple GPUs, taking advantage of the hardware platform significant computational power. The implementation of the most time consuming parts of the solver has been discussed. Those kernels that involve data from all cell points benefit from the use of the GPU shared memory to achieve coalesced memory accesses, that are the ones that provide the highest bandwidth. It has been found that, at least for $p = 4$ mesh elements, all kernels are memory bound, meaning that the majority of time is spent loading and storing data from the GPU memory. Almost all kernels analyzed are capable of reaching nearly optimal memory access, meaning that the GPU is efficiently used. The performance of the simulations involving multiple GPUs is also analyzed. It is found that the main bottleneck to achieve optimal parallel speed-ups is the bandwidth of the GPU to GPU communication, which has a strong influence when the number of partitions is large. Strategies to mitigate that effect, such as the overlapping of data transfers with computations and the use of GPU pinned memory to speed-up the transfers between CPU and GPU, are implemented. The code has been benchmarked running a three-dimensional case with hexahedral meshes. The GPU must have enough active threads to operate efficiently. When multiple GPUs are used, speed-ups are far from the optimal, especially for large numbers of MPI processes. The results show that the solver is precise enough to perform simulations involving turbulent structures, and that it is fast enough to be a promising tool for Large Eddy Simulation analysis in industrial environments.

References

[1] W. Reed and T. Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Report LA-UR-73-479, 1973.

[2] Bernardo Cockburn, George E. Karniadakis, and Chi-Wang Shu. The Development of Discontinuous Galerkin Methods. In Bernardo Cockburn, George E. Karniadakis, and Chi-Wang Shu, editors, *Discontinuous Galerkin Methods*, pages 3–50, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[3] Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[4] H.T. Huynh. A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods. In *18th AIAA Computational Fluid Dynamics Conference*, number 2007-4079, 2007.

[5] H.T. Huynh. A Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin for Diffusion. In *47th AIAA Aerospace Sciences Meeting*, number 2009-403, 2009.

[6] Z.J. Wang and Haiyang Gao. A unifying lifting collocation penalty formulation including the discon-

tinuous Galerkin, spectral volume/difference methods for conservation laws on mixed grids. *Journal of Computational Physics*, 228(21):8161 – 8186, 2009.

[7] T. Haga, H. Gao, and Z.J. Wang. A High-Order Unifying Discontinuous Formulation for the Navier-Stokes equations on 3D Mixed Grids. *Mathematical Modelling of Natural Phenomena*, 6(3):28–56, 2011.

[8] P. Castonguay, P.E. Vincent, and A. Jameson. A New Class of High-Order Energy Stable Flux Reconstruction Schemes for Triangular Elements. *Journal of Scientific Computing*, 51(1):224–256, April 2012.

[9] Peter E Vincent, Patrice Castonguay, and Antony Jameson. Insights from von neumann analysis of high-order flux reconstruction schemes. *Journal of Computational Physics*, 230(22):8134–8154, 2011.

[10] P. Castonguay, D.M. Williams, P.E. Vincent, and A. Jameson. Energy stable flux reconstruction schemes for advection - diffusion problems. *Computer Methods in Applied Mechanics and Engineering*, 267:400–417, 2013.

[11] Brian C Vermeire and Peter E Vincent. On the properties of energy stable flux reconstruction schemes for implicit large eddy simulation. *Journal of Computational Physics*, 327:368–388, 2016.

[12] BC Vermeire and PE Vincent. On the behaviour of fully-discrete flux reconstruction schemes. *Computer Methods in Applied Mechanics and Engineering*, 315:1053–1079, 2017.

[13] NVIDIA. Cuda programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`, June 2018.

[14] Khronos Group. Opencl 1.2 specification. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`, November 2011.

[15] A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863 – 7882, 2009.

[16] P. Castonguay, D.M. Williams, P.E. Vincent, M. Lopez, and A. Jameson. On the Development of a High-Order, Multi-GPU Enabled, Compressible Viscous Flow Solver for Mixed Unstructured Grids. In *20th AIAA Computational Fluid Dynamics Conference*, number 2011-3229, 2011.

[17] M. Siebenborn, V. Schulz, and S. Schmidt. A curved-element unstructured discontinuous Galerkin method on GPUs for the Euler equations. *Computing and Visualization in Science*, 15(2):61–73, April 2012.

[18] M.R. López-Morales, J. Bull, J. Crabill, T.D. Economon, D. Manosalvas, J. Romero, A. Sheshadri, J.E. Watkins II, D.M. Williams, F. Palacios, and A. Jameson. Verification and Validation of HiFiLES: a High-Order LES unstructured solver on multi-GPU platforms. In *32nd AIAA Applied Aerodynamics Conference*, number 2014-3168, 2014.

[19] F.D. Witherden, A.M. Farrington, and P.E. Vincent. Pyfr: An open source framework for solving advection - diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028 – 3040, 2014.

[20] J. Chan, Z. Wang, A. Modave, J.F. Remacle, and T. Warburton. GPU-accelerated discontinuous Galerkin methods on hybrid meshes. *Journal of Computational Physics*, 318(1):142–168, August 2016.

[21] J. Chan and T. Warburton. GPU-Accelerated Bernstein–BÃ©zier Discontinuous Galerkin Methods for Wave Problems. *SIAM Journal of Scientific Computing*, 39(2):628–654, 2017.

[22] B.C. Vermeire, F.D. Witherden, and P.E. Vincent. On the utility of GPU accelerated high-order methods for unsteady flow simulations: A comparison with industry-standard tools. *Journal of Computational Physics*, 334:497–521, January 2017.

[23] Roque Corral, Fernando Gisbert, and Jesus Pueblas. Execution of a parallel edge-based Navier-Stokes solver on commodity graphics processor units. *International Journal of Computational Fluid Dynamics*, 31(2):93–108, 2017.

[24] Ht T Huynh. A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods. *AIAA paper*, 4079:2007, 2007.

[25] P.L. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.

[26] Tim Warburton. An explicit construction of interpolation nodes on the simplex. *Journal of Engineering Mathematics*, 56(3):247–262, 2006.

[27] A. Jameson, P.E. Vincent, and P. Castonguay. On the Non-linear Stability of Flux Reconstruction Schemes. *Journal of Scientific Computing*, 2011.

[28] NVIDIA. Cuda profiler user's guide. `https://docs.nvidia.com/cuda/profiler-users-guide`, June

2018.

[29] T.J. Deakin, J. Price, M.J. Martineau, and S.N. McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In Michela Taufer, Bernd Mohr, and Julian M Kunkel, editors, *ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P3MA, VHPC, WOPSSS*, pages 489–507, Frankfurt, Germany, June 2016. Springer.

[30] NVIDIA. cublas. `https://docs.nvidia.com/cuda/cublas`, June 2018.

[31] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar JÃŒngel, and Siegfried Selberherr. Viennacl—linear algebra library for multi- and many-core architectures. *SIAM Journal on Scientific Computing*, 38(5):S412–S439, 2016.

[32] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis, parallel graph partitioning and sparse matrix ordering library, version 3.1, August 15 2003.

[33] C. Carton de Wiart, K. Hillewaert, M. Duponcheel, and G. Winckelmans. Assessment of a discontinuous Galerkin method for the simulation of vortical flows at high Reynolds number. *International Journal for Numerical Methods in Engineering*, March 2014.

[34] William Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8(1), 1965.

[35] 5th International Workshop on High-Order CFD Methods. `https://how5.cenaero.be`.

[36] Jan Michálek, Michelangelo Monaldi, and Tony Arts. Aerodynamic Performance of a Very High Lift Low Pressure Turbine Airfoil (T106C) at Low Reynolds and High Mach Number With Effect of Free Stream Turbulence Intensity. In *ASME Turbo Expo 2010: Power for Land, Sea and Air*, number GT2010-22884, 2010.