The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5-9 June 2022, Oslo, Norway

EFFICIENT IMPLEMENTATION OF A HIGH-ORDER COMPRESSIBLE NAVIER-STOKES EQUATIONS SOLVER RUNNING ON GRAPHICS PROCESSING UNITS

Fernando Gisbert, Adrián Sotillo and Jesús Pueblas[1]

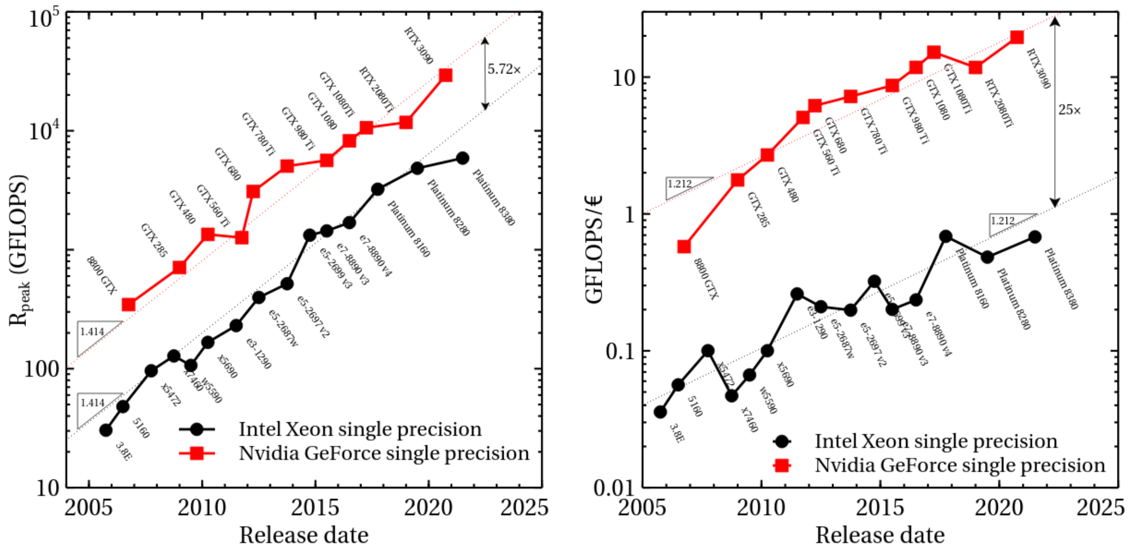
ITP Aero, Simulation Technologies Department
Francisca Delgado 9, 28830 Alcobendas, Spain
email: fernando.gisbert,adrian.sotillo,jesus.pueblas@itpaero.com

Key words: Compressible Navier Stokes, High-order, Flux reconstruction, Low Pressure Turbine, GPU, MPI

Abstract. This paper presents the implementation of a compressible Navier-Stokes equations solver that runs on multiple GPUs. The solver, known as Mu^2s^2T, uses a flux reconstruction high-order spatial discretization and is marched in time using an explicit Runge-Kutta method. The strong data locality that results from the codification of the spatial discretisation is very well suited to exploit the parallel capabilities of the GPU. We present a data access design that achieves nearly optimal data transfer rates between GPU memory and processor for the most time-consuming parts of the solver. In order to minimize the communication extra-cost when multiple GPUs are used in parallel, we have implemented a pipelined non-blocking data transfer between GPU and CPU that maximizes the overlap between communication and computation. The solver is executed on a cluster of NVIDIA GeForce GPUs. We demonstrate the solver computational efficiency and precision by running a turbulent channel flow at various Re_τ and comparing the results with those obtained with DNS simulations. The solver is used to predict the aerodynamic performances of a linear cascade of low pressure turbine vanes at varying Reynolds and Strouhal numbers. The numerical predictions show excellent agreement with the available experimental data.

1 Introduction

Second order RANS CFD solvers have been, and continue to be, the main pillar of the simulation capabilities in the aerospace industry. However, the predicting limitations of RANS equations have been extensively reported. In the turbomachinery industry, for instance, the stall on-set prediction in compressors or the turbine aerodynamic losses are the result of complex fluid interactions that are not adequately reproduced using RANS or URANS equations, due to the inherent limitations of the turbulence models involved, and more often than not expensive validation campaigns must be carried out to adjust the numerical predictions to the actual behaviour of the machine. To overcome such limitations one must resort to LES and DNS simulations, where the modelization of turbulence is less aggressive or non-existent. In those cases high order spatial discretisations are an interesting alternative to second order ones due to their superior performances in terms of computational cost and reduced numerical errors[1]. These methods have been an active area of research for the past 30 years. There are several



(a) (b)

Figure 1: Evolution of the computational power in single precision for CPUs and GPUs. (a) R_{peak}measured in GFLOPS. (b) GFLOPS/€, being the cost of the hardware that of the release date and without adjusting for inflation.

reviews that summarise the main findings [2, 3, 4, 5, 6]. This work presents the implementation of one of such approaches, a Flux Reconstruction (FR) [7, 8] spatial discretisation, later extended to 3D mixed grids by Wang et al.[9], which has been implemented to be executed on a cluster of Graphics Processing Units (GPUs).

The use of GPUs as general purpose processors has surged in the last fifteen years. The popularisation of programming languages that easily expose the inherent parallelism of the GPU hardware, such as CUDA[10] or OpenCL[11], and the release of GPUs that deliver much higher FLOPS than same generation CPU processors makes the implementation of CFD solvers on these computing platforms appealing. Figure 1a shows the evolution of the single precision theoretical peak performance (R_{peak}) for some NVIDIA GeForce GPUs and Intel Xeon CPUs since 2007, the year NVIDIA CUDA was released. Both architectures have followed Moore's law all these years, and R_{peak} for GeForce GPUs has consistently been 4-5 times larger than R_{peak} for Intel Xeon CPUs of the same generation. According to Moore's law that means a 4-5 year gap between both computing platforms. However, if we compare the performance per spent €, the gap between GPUs and CPUs widens, as shown in figure 1b. The investment on NVIDIA GeForce GPU processors returns approximately 25 times more theoretical computational power than the investment on CPU processors of the same generation which, according to the growing GFLOPS/€ law, is a 16 year gap.

However, it is well known that CFD solvers usually deliver a small fraction of R_{peak} due to the fact that their performance is memory bounded, i.e., the time spent reading data to be processed and writing the results back to memory is actually much higher than the time spent processing them. In that regard, GPUs also exhibit superior performances, as shown in figure 2. Both architectures have increased their memory bandwidth throughout the years, but NVIDIA

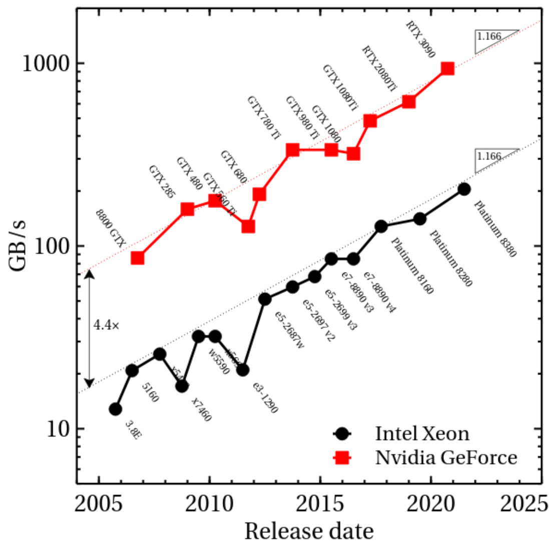


Figure 2: Evolution of the theoretical peak memory bandwidth for NVIDIA GeForce GPUs and Intel Xeon CPUs.

GeForce GPUs memory bandwidth is 4-5 times higher than that of Intel Xeon CPUs, which, according to the bandwidth increase rate, is a 9-10 year gap.

Obviously, to materialise all these potential benefits the CFD software must be able to a) achieve a substantial fraction of the memory bandwidth when moving data from memory to the processors and b) execute the operations with the data as efficiently as possible. Roughly speaking, for a GPU that means that the software must execute the same piece of code over sets of evenly spaced input data and write the result to sets of evenly spaced output data, and that the number of operations per memory fetch should be as high as possible. It turns out that solvers whose spatial discretisation is based on compact differentiation largely fit into that category, therefore several examples of GPU implementations exist in the literature. Klöckner et al.[12] were the first ones to implement a GPU version of a Discontinuous Galerkin (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[13], showing similar speed-ups relative to the CPU execution. Since then, the number of DG and FR solvers has been growing steadily[14, 15, 16, 17, 18, 19].

In this paper we present the techniques used to implement the most time-consuming parts of a high-order FR Navier-Stokes solver for unstructured grids, which is part of the Mu^2s^2T suite of ITP Aero in-house CFD solvers[20]. It is structured as follows: first, the equations are presented and their spatial discretisation is outlined. Next, the implementation of the most time consuming parts of the solver is detailed in section 3. Then, we present an implementation of the parallel communications that maximises the computation and communication overlap in multi-GPU executions. Some benchmark cases are presented, showing the solver performance and its predicting capabilities. Finally, the predictions obtained after the simulation of a low pressure turbine (LPT) vane at a given Reynolds and Strouhal numbers are compared against experimental measurements.

2 Problem formulation

The differential form of the Navier-Stokes equations is:

$$\frac{\partial \mathbf{U}}{\partial t}+\frac{\partial \mathbf{f}}{\partial x}+\frac{\partial \mathbf{g}}{\partial y}+\frac{\partial \mathbf{h}}{\partial z}=\frac{\partial \mathbf{F}}{\partial x}+\frac{\partial \mathbf{G}}{\partial y}+\frac{\partial \mathbf{H}}{\partial z} \tag{1}$$

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

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

the inviscid fluxes and

$$\left[\begin{array}{ccc}\mathbf{F} & \mathbf{G} & \mathbf{H}\end{array}\right]=\left[\begin{array}{ccc}0 & 0 & 0\\ \tau_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{xy} & \tau_{yy} & \tau_{yz}\\ \tau_{xz} & \tau_{yz} & \tau_{zz}\\ u\tau_{xx}+v\tau_{xy}+w\tau_{xz}-q_x & u\tau_{xy}+v\tau_{yy}+w\tau_{yz}-q_y & u\tau_{xz}+v\tau_{yz}+w\tau_{zz}-q_z\end{array}\right] \tag{3}$$

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

$$R_g=C_p-C_v$$

being C_p the specific heat at constant pressure and C_v the specific heat at constant volume. Besides,

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

where $q^2=u^2+v^2+w^2$. The pressure can then be 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. The viscous stress tensor expression is

$$\tau_{ij}=\mu\left(\partial_i v_j+\partial_j v_i\right)-\frac{2}{3}\mu\left(\nabla\cdot\mathbf{v}\right)\delta_{ij} \tag{4}$$

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

$$\mathbf{q}=-k\nabla T \tag{5}$$

In the preceding equations (4) and (5), μ and k are the laminar viscosity and the conductivity, respectively. The expression of the laminar viscosity is given by Sutherland's 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.

2.1 The flux reconstruction spatial discretisation

The Navier-Stokes equations are discretised using a FR scheme[7, 8, 9, 21], whose main features are briefly described in order to clarify the subsequent discussion about their implementation. We fill the control volume using a hybrid unstructured grid, whose cells do not share points. For each cell we define a series of solution points (SPs) at which the state variable U is evaluated. Then the solution within the cell is approximated using Lagrange polynomials L_i of order p over these SPs:

$$U(\mathbf{x}, t) \approx U_i^h(\mathbf{x}, t) = \sum_{j=0}^{N_c(p)-1} U_{ij}^h(t) \, L_j(\mathbf{x}) \tag{6}$$

where $N_c(p)$ is the number of SPs for a given cell shape and order. For convenience the spatial dimensions (x, y, z) are mapped onto standard element coordinates (ξ, η, ζ), and a transformation between them is defined using the Jacobian matrix

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

and its inverse

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

therefore Eq. (1) is then expressed as

$$\frac{\partial |J| \cdot U}{\partial t} + \frac{\partial \widetilde{F}}{\partial \xi} + \frac{\partial \widetilde{G}}{\partial \eta} + \frac{\partial \widetilde{H}}{\partial \zeta} = 0 \tag{7}$$

with

$$\begin{aligned} \widetilde{F}(U) &= |J| (\xi_x F + \xi_y G + \xi_z H) \\ \widetilde{G}(U) &= |J| (\eta_x F + \eta_y G + \eta_z H) \\ \widetilde{H}(U) &= |J| (\zeta_x F + \zeta_y G + \zeta_z H) \end{aligned} \tag{8}$$

The flux is also approximated using a Lagrange polynomial basis, as in eq. (6):

$$\widetilde{F}(U) \approx \widetilde{F}_i^D(\mathbf{x}, t) = \sum_{j=0}^{N_c(p)-1} \widetilde{F}\left(U_{ij}^h\right) L_j(\mathbf{x}) \tag{9}$$

We use the same points to evaluate the derivatives and the fluxes, avoiding the need to perform an additional interpolation to obtain the solution at the flux points. The Gauss-Lobatto points have been used for the hexahedral elements, and the triangle point distributions of Warburton[22] are employed for triangular prisms. The points for tetrahedra are those of Hesthaven and Warburton[23] and for pyramids those of Chan and Warburton[24].

Next, a common flux at the cell interface points must be defined and added to the spatial derivatives to unify the fluid domain. We have used the approximate Riemann solver proposed by Roe[25] to evaluate the common convective fluxes. The common viscous flux is evaluated using the average of primitive variables and gradients at both sides of the interface. A correction function is used to ensure that the fluxes at the cell interfaces are the common ones. The properties of the spatial discretisation depend on the choice of this correction function[7]. We have used the left and right Radau polynomials for hexahedra[7] and the lifting coefficients for triangular prisms, tetrahedra and pyramids[9].

The complete expression for the function derivative is then

$$\frac{\partial \widetilde{F}_i}{\partial \xi_j} = \sum_{k=0}^{k=N_c(p)-1} D_{ikj} \widetilde{F}_{ik} + \sum_{k=0}^{k=N_b(p)-1} C_{ikj} \left(F_{common} - F_{boundary}\right)_k \tag{10}$$

where

$$D_{ikj} = \frac{\partial L_k \left(\xi_i\right)}{\partial \xi_j} \tag{11}$$

is a $N_c\left(p\right) \times N_c\left(p\right)$ matrix that multiplied by $\widetilde{F}_{ik}$ yields the $\xi_j = \xi, \eta, \zeta$ derivative for each cell SP and C_{ikj}represents a $N_c\left(p\right) \times N_b\left(p\right)$ matrix containing the correction coefficients for the ξ_j derivative, that is multiplied by a vector $F_{common} - F_{boundary}$ whose dimension is the number of cell boundary points $N_b\left(p\right)$.

The aliasing instabilities that arise when trying to resolve flow features whose characteristic length is smaller than the mesh resolution have been controlled by using the split form of the fluxes proposed by Kennedy and Gruber[26] whose stabilising and conservation properties have been demonstrated in[27, 28]. We reproduce here the resulting spatial discretisation of the convective terms for the sake of completeness. The convective terms derivatives of eq. (7) can be recast as

$$\frac{\partial}{\partial \xi}\left(\rho u_\xi \phi\right) + \frac{\partial}{\partial \eta}\left(\rho u_\eta \phi\right) + \frac{\partial}{\partial \zeta}\left(\rho u_\zeta \phi\right)$$

where

$$\begin{bmatrix} u_\xi \\ u_\eta \\ u_\zeta \end{bmatrix} = \left|J\right| J^{-1} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$$

and $\phi = [1,\ u_x,\ u_y,\ u_z,\ E + p/\rho]$. The Kennedy Gruber Pirozzoli (KGP) split form is defined, for each component, as[28]:

$$\begin{aligned} \frac{\partial}{\partial \xi_j}\left(\rho u_{\xi_j} \phi\right) &= \frac{1}{4}\left[\frac{\partial}{\partial \xi_j}\left(\rho u_{\xi_j}\phi\right) + \phi \frac{\partial}{\partial \xi_j}\left(\rho u_{\xi_j}\right) + \rho u_{\xi_j}\frac{\partial \phi}{\partial \xi_j} + \right. \\ &+ \left. u_{\xi_j}\frac{\partial}{\partial \xi_j}\left(\rho\phi\right) + \rho\phi\frac{\partial u_{\xi_j}}{\partial \xi_j} + \rho\frac{\partial}{\partial \xi_j}\left(u_{\xi_j}\phi\right) + \phi u_{\xi_j}\frac{\partial \rho}{\partial \xi_j}\right] \end{aligned} \tag{12}$$

This form is applied to the discontinuous derivative term only, i.e., the derivative correction is not applied to each term separately, but to the complete flux.

Finally, the gradient of the primitive variables, that is needed to evaluate the viscous fluxes, is computed with the following expression:

$$\frac{\partial V}{\partial x_i} = \left(J^{-1}\right)^T_{ij} \frac{\partial V}{\partial \xi_j} \tag{13}$$

where

$$\frac{\partial V}{\partial \xi_j} = \frac{1}{|J|} \left(\frac{\partial \left(|J|\, V\right)}{\partial \xi_j} - \frac{\partial |J|}{\partial \xi_j} V \right) \tag{14}$$

being $x_i = x, y, z$. The numerical evaluation of the derivative in eq. (14) is analogous to eq. (10), but the common value at the boundary points in this case is the average value of the primitive variables at both sides of the element boundary.

2.2 Boundary conditions and time integration

The inlet and outlet boundary conditions are the 3D characteristic boundary conditions[29] but adapted to the inlet boundary condition data employed at ITP Aero for subsonic inlets. The mathematical development to obtain the formulation is similar to that of Odier et al.[30]. A synthetic turbulence generation method proposed by Shur et al.[31] is used to generate inlet turbulence profiles if required.

Adiabatic or iso-thermal no-slip boundary conditions are used at viscous walls. Periodic boundary conditions are used to simulate just one passage of the turbomachine or an infinite span airfoil. Finally, an immersed boundary method[32] is used to simulate the passing bars as wake generation mechanisms in LPT simulations with upstream wakes, avoiding the need to actually mesh them and the implementation of a sliding mesh method to simulate the moving domain together with the LPT vane.

The resulting spatial discretisation is marched in time with a four-stage explicit Runge-Kutta method

$$U_{n+1} = U_n + \Delta t_n \sum_{m=1}^{4} b_m \left.\frac{\partial \widetilde{F}_i}{\partial \xi_i}\right|_m$$

This summation has to be carefully executed in order to minimise the possible round-off problems that may arise in simulations with long integration times with very small time steps, especially in single precision simulations. We have used the Kahan summation algorithm[33] to minimise the round-off problems.

3 Implementation of the operators

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

1. Evaluate the derivative of the primitive variables. This step has been split in three sub-steps:

   (a) Compute the discontinuous derivative by evaluating a matrix-vector product, where the matrix elements are the derivatives of the Lagrange polynomials for each solution point: L_i' of equation (11).
   (b) Correct the derivative with the correction functions and the average value of the primitive variables at the face points. This is again a matrix-vector product, but now the matrix C_{ijk} is given either by the left and right Radau functions for the hexahedra or by the lifting coefficients for prisms, and the vector has the differences between primitive variables at both sides of the face points.
   (c) Pass the gradient to physical coordinates with eq. (13).

2. Evaluate the derivatives of the fluxes. This is performed, again, in three sub-steps:
   (a) Compute the fluxes for each solution point with eqs. (2) and (3).
   (b) Compute the fluxes discontinuous derivatives, as in step 1a.
   (c) Correct the derivative with the correction functions and the common fluxes at the face points, as in step 1b. There is no need to pass to physical coordinates afterwards because the fluxes are already expressed in computational coordinates in eq. (7).
3. Modify the fluxes by enforcing the boundary conditions.
4. Advance to the next stage of the Runge-Kutta scheme and go to 1.

Of the steps outlined above, only 1b and 2c use data from outside the cell element. The rest of the steps are either point-based, like 1c or 2a, or element-based, like 1a or 2b. Even steps 1b and 2c, that need data from neighboring cells, use these data to correct the derivatives in the cell nodes. There are a number of reasons to choose the cell as our basic data structure. The main two are:

- In discontinuous spatial discretisations such as FR, one can number the nodes within an element consecutively, i.e., if the first node of cell `icell` is in position `ifirst(icell)`, the nodes in that cell have indices from `ifirst(icell)` to `ifirst(icell+1)`. Even though the mesh is unstructured, the data inside a cell is treated as structured. This is key to exploit one of the advantages of GPUs over CPUs: the higher memory bandwidth. If the data accesses were disordered, as it usually happens in unstructured grids without repeated nodes, the optimal bandwidth could be hardly achieved.
- On GPUs, the number of simultaneous active threads is very large, of the order of thousands. Both NVIDIA and AMD GPUs group threads into groups, each thread group containing up to 1024 threads (for current NVIDIA GPUs), that share resources such as the so-called shared or local memory. Since it 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 at much less cost when compared with a load from global memory and, crucially, without caring about the optimal data arrangement explained above. One could then load all the cell data into shared memory and the operations with them would be extremely fast because the time penalty to access the data would be negligible when compared to accessing them from GPU memory.

These are the main reasons why we have chosen GPUs as computing platform and coded the FR solver using a mixture of C++ and OpenCL[11]. The resulting software can be executed on CPUs as well as GPUs and other many-core hardware thanks to the use of OpenCL language, which is supported by all major hardware vendors such as NVIDIA, Intel and AMD. An additional in-house implementation allows an automatic translation of OpenCL code into CUDA[34], which is NVIDIA proprietary language, taking advantage of the similarities between the OpenCL and CUDA API programming models. That allows the use of NVIDIA debugging and profiling tools, which are very useful when trying to optimise the implementation. With this approach we do not restrict ourselves to one particular hardware platform nor to a proprietary ecosystem, but we can still take advantage of the best each platform has to offer. This is true in theory, but in practice the optimisation strategies for each type of hardware are quite different. In this work we will focus on optimising the code to run on GPUs.

3.1 Data arrangement for maximum achievable bandwidth

The GPU to processor transfer rate 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 five conservative variables per node. If the mesh has N_{SP} solution points, it is better to place ρ in the first N_{SP} positions, then ρu, etc. and access them as

$$(\rho_i, \rho u_i, \rho w_i, \rho v_i, \rho E_i) \quad = \quad (U_i, U_{i+N_{SP}}, U_{i+2N_{SP}}, U_{i+3N_{SP}}, U_{i+4N_{SP}})$$

than to pack the five variables for each node and access them with a strided access:

$$(\rho_i, \rho u_i, \rho w_i, \rho v_i, \rho E_i) \quad = \quad (U_{5i}, U_{5i+1}, U_{5i+2}, U_{5i+3}, U_{5i+4})$$

With the SoA memory access, the optimal bandwidth is obtained. By optimal bandwidth we understand the maximum achievable bandwidth, which is not always the peak but the measured bandwidth in benchmarks such as STREAM[35], which is around 80% of the peak value. The tests and simulations in this work run on NVIDIA GeForce GTX 1080Ti GPUs, which have a theoretical peak bandwidth of 484 GB/s, and the maximum measured bandwidth is around 380 GB/s.

3.2 Evaluation of the discontinuous derivatives

Listing 1: OpenCL code of the kernel used to compute the discontinuous derivative of the primitive variables in 2d triangles

```
template<typename T>
__kernel void discontinuousGradientForTriangles(...)
{
  // We have as many thread groups as cells.
  int icell = get_group_id(0);
  // The node numbering within the cell
  int lid = get_local_id(0);
  // The node global mesh index
  int inode = lid+ifirst[icell];
  // Put the data in local arrays for faster memory access
  for(int i=lid;i<D_size;i+=NumberOfCellNodes) {
    D_localX[i] = D_globalX[i];
    D_localY[i] = D_globalY[i];
  }

  for(int ic=0;ic<NumberOfVariables;ic++) {
    // Put the data in local arrays for faster memory access
    U_local[lid] = U_global[inode+ic*NumberOfNodes];

    // Make sure all threads have finished writing to local memory
    barrier(CLK_LOCAL_MEM_FENCE);

    // Each thread computes one row of the matrix-vector multiplication
    T dx = 0.,dy = 0.;
    for(int i=0; i<NumberOfCellNodes; i++) {
      dx += D_localX[NumberOfCellNodes*lid+i]*U_local[i];
      dy += D_localY[NumberOfCellNodes*lid+i]*U_local[i];
    }

    // Write the result back to global memory
    Grad_global[inode+ ic                    *NumberOfNodes] = dx;
    Grad_global[inode+(ic+NumberOfVariables)*NumberOfNodes] = dy;

    // Make sure U_local is not re-filled before dx and dy computation
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

The evaluation of the discontinuous derivatives of the primitive variables or the fluxes involves multiplying some if not all cell SPs by the derivatives of the Lagrange polynomials at these SPs. This matrix-vector operation could be handled by linear algebra libraries like cuBLAS[36] in CUDA or ViennaCL[37] in OpenCL. However, several authors[12, 13] have compared the performance of such libraries with kernels tailored for the specific data layout, and the latter generally outperform the former. The efficient execution of the kernel is best achieved making use of the local GPU memory presented above. The main drawback is that it has a reduced size (up to 192 KB for NVIDIA A100 GPUs, 48 KB for the GeForce GTX 1080Ti used in the present work), therefore we have to be cautious selecting which data is placed in that memory, since we can easily run out of it. To evaluate the discontinuous derivative we execute as many thread blocks as cells. The OpenCL code representing the derivation of primitive variables in 2d triangles is shown in the program listing 1. For each cell:

1. Obtain the cell and node indices. Note the use of `ifirst[cell]` which, for each cell, gives the position of its first node.

2. Fetch the data from global memory and put it in the shared memory arrays. The data include the matrices to perform the derivatives and the data to be derived. Since there are no repeated nodes between cells, the data access pattern is regular and coalesced memory

Element type	$N_c(p)$	$LM_D(p)$ /`sizeof(T)`
Quad	$(p+1)^2$	$2N_c$
Triangle	$(p+1)(p+2)/2$	$N_c(1+2N_c)$
Hexahedron	$(p+1)^3$	$(p+1)^2+N_c$
Triangular prism	$(p+1)^2(p+2)/2$	$2[(p+1)(p+2)/2]^2+(p+1)^2+N_c$
Tetrahedron	$(p+1)(p+2)(p+3)/6$	$N_c(1+3N_c)$
Pyramid	$(p+1)(p+2)(2p+3)/6$	$N_c(1+3N_c)$

Table 1: Number of points and size of the local memory arrays to evaluate the primitive variables discontinuous derivative for each element type as a function of the polynomial degree p.

Element type/$p_{\max}$	$LM_D(p_{\max})$, SP	$LM_D(p_{\max})$, DP	$N_c(p_{\max})$
Quad	77	54	31
Triangle	11	9	43
Hexahedron	21	16	9
Triangular prism	10	8	11
Tetrahedron	5	4	16
Pyramid	4	3	13

Table 2: $p_{\max}$ to avoid surpassing 48KB of local memory size in single and double precision and $p_{\max}$ to avoid having more than 1024 solution points in the evaluation of the primitive variables discontinuous derivatives for each element type.

accesses are possible, achieving the optimal data transfer rate. The measured bandwidth is $\sim 350\,GB/s$, near the 75% peak value for the GPU.

3. After synchronising all threads, each one evaluates its corresponding row of the $D{\cdot}U$ matrix vector product for each variable. Some of the memory accesses are strided, but since the data are already contained in the shared memory, this is no longer severely penalising.

4. Write the result to the corresponding global memory position. The data storage is also done with an optimal data arrangement, achieving optimal bandwidth.

Table 1 summarises the number of points and the amount of local memory needed for each cell type in order to compute the primitive variables gradient. In quads and hexahedra it is assumed that all directions use the same differentiation matrix, whose size is $(p+1)^2$. Based on these numbers, table 2 represents the maximum polynomial degree $p_{\max}$ for which the shared memory limit is reached for single and double precision or the number of cell points surpasses the maximum number of threads, which is 1024. The limiting factor for all of them except quads and hexahedra is the amount of local memory. This limit could be overcome if the matrix-vector multiplication is split in smaller size blocks, in a strategy called tiling, but for the elements and polynomial degrees used in this work $p < p_{\max}$ and the tiling is not needed.

Finally, it should be mentioned that this rather straight-forward implementation is sub-optimal for elements with low order p because in that case the thread group would be very small and so would be the GPU occupancy. In that case, one could create a thread group containing several elements, whose numbering should ideally be consecutive, and apply the operations on all of them at the same time. Once again, for the elements and polynomial degrees employed in this work there has not been a need to pursue any further optimisation.

3.3 Evaluation of derivative corrections

Listing 2: OpenCL code of the kernel used to correct the discontinuous derivative of the primitive variables in 2d triangles

```
template<typename T>
__kernel void gradientCorrectionForTriangles(...)
{
  // We have as many thread groups as cells.
  int icell = get_group_id(0);
  // The node numbering within the cell
  int lid = get_local_id(0);
  // The first node of the cell
  int ifirstnode = ifirst[icell];
  // The node global mesh index
  int inode = lid+ifirstnode;
  // The first face node of the cell
  int ifirstfacenode = ifirstface[icell];

  // Put the data in local arrays for faster memory access
  for(int i=lid;i<LiftingCoeffs_size;i+=NumberOfCellNodes) {
    LiftingCoeffs_local[i] = LiftingCoeffs_global[i];
  }

  for(int ic=0;ic<NumberOfVariables;ic++)
  {
    for(int fp=lid;fp<NumberOfFacePoints;fp+=NumberOfCellNodes)
    {
      // FacePointsIdx contains the node index of each face node within the cell
      int ilocal = FacePointsIdx[fp];
      // The global mesh index of that face node
      int inodeFace = ilocal+ifirstnode;
      // The node index from the neighbor cell face
      int ineigh = FacePointNeighbor[ifirstfacenode+fp];
      // The common value is U_comm = 0.5*(U_i+U_neigh), then the delta is
      // U_comm - U_i = 0.5*(U_neigh-U_i)
      DeltaU_local[fp] = 0.5*(U_global[ineigh    +ic*NumberOfNodes] -
                              U_global[inodeFace+ic*NumberOfNodes]);
    }
    // Make sure all threads have finished writing to local memory
    barrier(CLK_LOCAL_MEM_FENCE);

    // Each thread computes the correction for its associated cell node
    T cx = 0., cy = 0.;
    // First index of the LiftingCoeffs_local array for each cell node
    int ifirstlc = NumberOfFacePoints*lid;
    // The correction formula using deltas and lifting coefficients
    for(int fp=0;fp<NumberOfEdgePoints;fp++)
    {
      T cxy = LiftingCoeffs_local[ifirstlc+fp+NumberOfEdgePoints]*
              DeltaU_local[fp+NumberOfEdgePoints];
      cx -= LiftingCoeffs_local[ifirstlc+fp+2*NumberOfEdgePoints]*
            DeltaU_local[fp+2*NumberOfEdgePoints]+cxy;
      cy -= LiftingCoeffs_local[ifirstlc+fp]*
            DeltaU_local[fp]+cxy;
    }

    // Write the result back to global memory
    Grad_global[inode+ ic                    *NumberOfNodes] += cx;
    Grad_global[inode+(ic+NumberOfVariables)*NumberOfNodes] += cy;

    // Make sure U_local is not re-filled before cx and cy computation
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

The correction of the discontinuous derivatives involves data from adjacent cells. In that case the ordered access to the data is lost, therefore a decrease in the achieved bandwidth should be expected. As with the discontinuous derivatives, the kernel to correct these derivatives does not rely on mathematical libraries, but is written specifically for each element type. An example of the implementation for 2d triangles is presented in program listing 2. The steps followed are:

1. Obtain the cell, node and face node indices. `ifirstface[icell]` returns, for each cell, the position of its first face node.

2. Compute the deltas of primitive variables at each face point and store them in a local memory array whose size is the number of face points of the element. The memory access to the data is not ordered, as in the discontinuous derivative, but follows the numbering of the cell face nodes and the numbering of the face nodes in the neighbor cell. Neither of both are guaranteed to be at consecutive memory positions and therefore the fetching of these data will be slower. However, modern GPUs have also some cache memory, albeit not as large as that of CPUs. Given that the face nodes are close from each other at both elements it is expected that some of them will be placed in the cache when one of them is fetched, minimising the memory access penalty somehow. Still, the coding has not been optimised to minimise cache misses, and the beneficial side effects, if any, are not intended but they are nevertheless welcome.

3. Compute the correction terms by multiplying the deltas by the correction matrix. In the example of 2d triangles, this correction matrix is given by the lifting coefficients[9], as in triangular prisms, tetrahedra or pyramids. In quads or hexahedra, the Radau polynomial coefficients are used[7]. These computations are all performed using data stored in local memory, and are very fast.

4. Write the results to their corresponding global memory positions. As with the discontinuous derivative, this last step has an ordered data access, where each thread writes the result at

Element type/$p_{\max}$	$LM_C(p)/\texttt{sizeof(T)}$	$LM_C(p_{\max})$, SP	$LM_C(p_{\max})$, DP
Quad	$5(p+1)$	2456	1227
Triangle	$3(p+1)[1+(p+1)(p+2)/2]$	18	14
Hexahedron	$(p+1)[1+6(p+1)]$	44	30
Triangular prism	$(p+1)(3p^2+17p+18)/2$	17	13
Tetrahedron	$2(p+1)(p+2)[1+(p+1)(p+2)(p+3)/6]$	6	5
Pyramid	$(p+1)(3p+5)[1+(p+1)(p+2)(2p+3)/6]$	5	4

Table 3: Required amount of local data and $p_{\max}$ to avoid surpassing 48KB of local memory size in single and double precision in the evaluation of the primitive variables derivatives correction for each element type.

a consecutive memory position, resulting in optimal memory access bandwidth.

Some authors[12, 13] choose a different approach: they implement an additional kernel to pre-compute the face points common values and store them. The total number of threads for that kernel is equal to the total number of face points, and the kernel thread group size is the number of face points for each cell instead of the number of cell points, hence avoiding idle threads when computing the face common fluxes for high polynomial orders. The second kernel reads the common fluxes and computes the flux derivative correction. The common fluxes writing and reading is coalesced, adding little extra execution time. In theory, the detrimental effect of the extra writes and reads would be offsetted by the optimal thread group size of each kernel. However, we have tried both approaches and the one with separate kernels is slower due to those extra memory accesses. Even when taking the adverse effect of the inappropriate thread group size into account, it is more optimal to group both kernels to minimize the memory accesses, at least for the hardware and polynomial degrees tested in this work.

Table 3 shows the amount of local memory needed to evaluate the primitive variables gradient correction and the maximum polynomial degree to reach the 48KB limit for each element and computing precision. For all elements the local memory requirement is less stringent than that of the discontinuous derivative.

4 Parallel use of multiple GPUs

For large enough problems that do not fit into a single GPU, multiple GPUs are used in parallel. The domain decomposition is carried out using the parallel domain decomposition library ParMeTiS[38]. Each resulting sub-domain is processed by a single CPU core, that in turn controls a single GPU. Given that each node of our cluster has 10 GPUs, that is clearly not the most optimal approach to minimize the data transfer between parallel processes. If one CPU process controlled several GPUs, the exchange of data between GPUs would not need to use MPI, because the GPU data sent by all GPUs sharing the same CPU process would reside in the same physical memory and the MPI data exchange would be limited to inter-node communications. If each GPU is controlled by a separate CPU process, GPU to GPU data exchange needs MPI to proceed, even if those GPUs are in the same node , adding an extra communication layer. While not optimal, this strategy is the most versatile in real environments where the GPU cluster is shared among many users and the available resources are not known a priori.

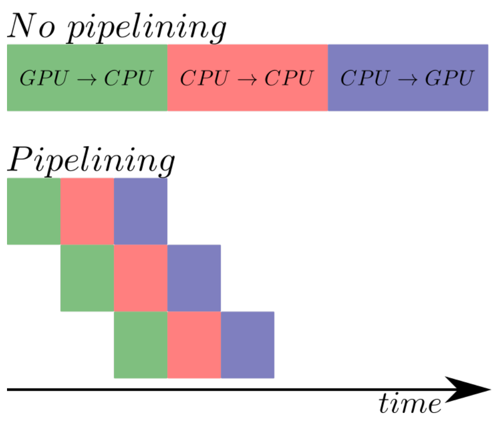


Figure 3: Pipelined communication. The message is split in smaller pieces, whose communication proceeds in parallel. The communication time is reduced.

With the chosen approach, the bandwidth of the data exchange between GPUs is a result of a three-step process:

1. Transfer the data from the origin GPU to the origin CPU process controlling it. That data transfer proceeds through a x16 PCIe 3.0 link, that has a theoretical unidirectional bandwidth of 16 GB/s and a measured effective bandwidth for large enough messages of around 12 GB/s.

2. Use MPI to send the origin CPU data to the destiny CPU through the two-port Infiniband network used in ITP Aero's cluster. The theoretical network unidirectional data transfer rate is 2 x 100 Gb/s, but it is limited by the x16 PCIe 3.0 link of the network interface. The measured unidirectional bandwidth is again around 12 GB/s for large enough messages.

3. Transfer the data from the destiny CPU to the destiny GPU through the same x16 PCIe 3.0 link, with the same transfer rate of step 1.

The PCIe bus and the Infiniband fabric support simultaneous bidirectional data transfers with no noticeable data rate degradation, thus the bidirectional bandwidth is double the unidirectional. The effective GPU to GPU bidirectional bandwidth is then

$$\frac{1}{BW_{GPU\leftrightarrow GPU}} = \frac{2}{BW_{GPU\leftrightarrow CPU}} + \frac{1}{BW_{CPU\leftrightarrow CPU}}$$

For large enough messages, that results in an effective GPU to GPU bandwidth of 8 GB/s, 43 times less than the 350 GB/s achieved when transferring data inside the GPU. This penalty could be minimised by using faster buses to transfer data between GPU and CPU, like NVIDIA's NVLink port or more recent PCIe technology. But absent better hardware some additional improvements must be implemented to minimise the communications penalty, otherwise the parallel efficiency of the solver is seriously compromised. The mitigation strategies fall into two categories: those aimed at increasing the data transfer bandwidth and those that maximise the overlap between communication and computation.

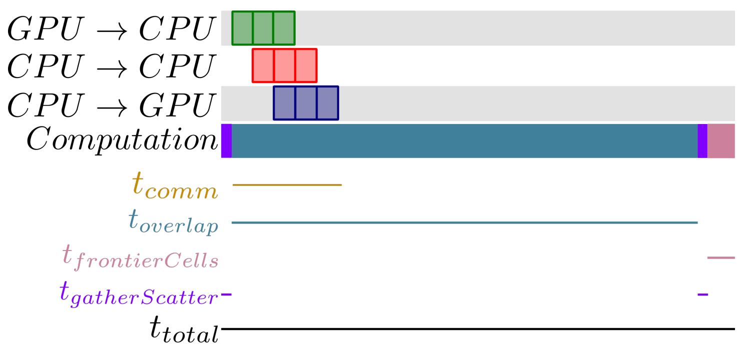


Figure 4: Overlap of computation and communication. The $GPU \to CPU$, the MPI communication ($CPU \to CPU$) and the $CPU \to GPU$ data transfers proceed simultaneously with the kernel executions. As long as the total communication time t_{comm} remains lower than $t_{overlap}$, the extra-cost of running parallel simulations will be $t_{gatherScatter} + \frac{1-\chi_{eff}}{\chi_{eff}} t_{frontierCells}$, where χ_{eff} is the parallel efficiency of the GPU, which is below unity when the number of kernel threads is small.

Increase the data transfer bandwidth

To take full advantage of the PCIe bus performances in CPU to GPU data transfers one must use pinned, or page-locked memory. The 12 GB/s data transfer rate through the PCIe bus is only obtained for this type of memory, otherwise the effective bandwidth is almost halved. Besides, taking advantage of the three-step communication explained above, one can reduce the communication time very much by implementing a pipelined communication process, schematically depicted in figure 3. In a non-pipelined communication, we first send the full message from the GPU to the CPU, and only when this transfer has finished the CPU to CPU MPI communication starts. After completing the MPI communication, the final CPU to GPU transfer ends the communication. By dividing the message in smaller pieces, the MPI communication of the first piece starts as soon as the first GPU to CPU transfer finishes, and it proceeds while the second piece of GPU to CPU data transfer is carried out, the second piece of MPI data exchange is executed simultaneously to the first CPU to GPU data transfer and the third GPU to CPU one and so on. There is an optimal size of the subdivided messages that maximises the bandwidth. Too small messages degrade the bandwidth due to the network latency, while too large messages do not take full advantage of the pipelining scheme. For message sub-divisions larger than 1 MB we have measured bandwidth improvements in excess of 100% in optimal working conditions. However, these improvements may be severely limited depending on the PCIe bus and/or the Infiniband fabric congestion.

Overlap communication and computation

When we overlap communication and computation, we hide the extra cost of communicating as much as possible. This is graphically explained in figure 4. As long as the t_{comm} time required by the three-step communication strategy explained above is less than the time spent computing $t_{overlap}$, the communication cost is negligible. The first obvious improvement is to make the communications non-blocking to allow their execution along the computations. We use MPI non-blocking communications that allow the simultaneous launch of GPU kernels. The kernel execution could in theory proceed simultaneously with whatever instruction executed on the CPU, but we do not take advantage of that feature in this work because the CPU is only in charge of launching kernels and managing the communications during the code execution. We also overlap the GPU to CPU data transfers with the GPU computation. This is done by creating three independent execution queues on the GPU, one controlling the GPU to CPU data transfer, another for the CPU to GPU, and the last one to control the kernels execution.

The second improvement is to increase $t_{overlap}$ as much as possible. This is done by subdividing the derivatives correction loop of subsection3.3. The first sub-division contains just cells whose boundary correction does not depend on data from neighbor parallel domains. That loop is the most time consuming because it contains the bulk of domain cells. By allowing its execution along the communication we have increased $t_{overlap}$ substantially. The second subdivision contains those cells that need boundary data from neighbor parallel domains to produce correct output. Before the second sub-loop execution begins, we must establish a checkpoint between the execution queue and the CPU to GPU data transfer queue to ensure that the loop is executed only after the frontier data are fully transferred.

With these strategies the extra cost of parallelism is estimated as

$$t_{gatherScatter} + \frac{1-\chi_{eff}}{\chi_{eff}} t_{frontierCells}$$

being $t_{gatherScatter}$ the cost of gathering the data that must be sent and scattering the data that has been received, $t_{frontierCells}$ the cost of computing the derivative correction for the frontier cells as if it were a mere percentage of the total cost of the derivative correction, i.e.

$$t_{frontierCells} = t_{totalCells} \cdot n_{frontierCells} / n_{totalCells}$$

scaled by a factor $(1-\chi_{eff})/\chi_{eff}$(with $\chi_{eff} \in (0,1]$) that takes into account the loss of parallel efficiency of the GPU ($\chi_{eff} < 1$) when executing the sub-loop that contains just the parallel frontier size as a consequence of the small number of threads of that kernel[39]. The penalty is small compared with the execution cost of the serial operations. Therefore, as long as the communication time t_{comm} is lower than the overlap time $t_{overlap}$ the parallel efficiency will remain high.

5 Benchmarking

We present a number of tests that quantify the solver performances when running on ITP Aero's GPU cluster. The cluster consists of 20 nodes communicated with a two-port Infiniband network at 2×100 Gb/s. Each node hosts 10 NVIDIA GeForce GTX 1080Ti GPUs. The computing power of this particular GPU model is much larger for single than for double precision, therefore the tests in this work have been carried out using single precision. The GPUs are connected to the host's two Intel Xeon E5-2620 v4 CPUs via two x16 PCIe3.0 links at 16 GB/s. Each x16 PCIe3.0 link connects 5 GPUs to one single CPU, which means that when all GPUs are sending and receiving data through the link at the same time the effective bandwidth for each GPU is much lower than the theoretical 16 GB/s.

p	1	2	3	4	5	6	7
N_{hex}	221184	65536	27648	13718	8192	5488	3456
N_{DOF}	1769472	1769472	1769472	1714750	1769472	1882384	1769472

Table 4: Number of hexahedra and number of mesh points for the meshes used in the single GPU profiling.

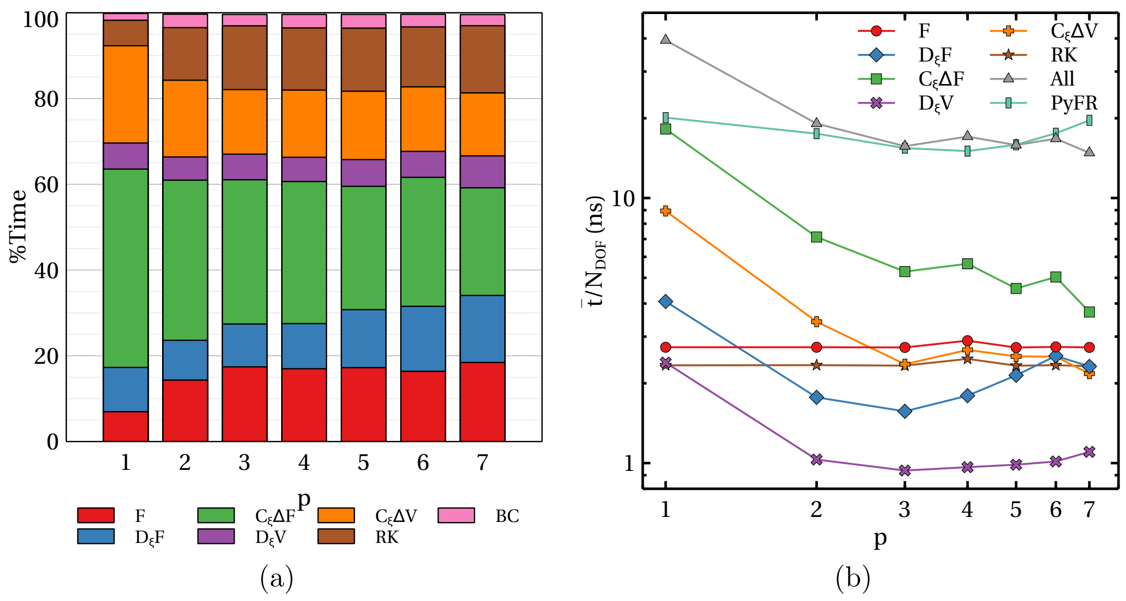


Figure 5: (a) Fraction of total time and (b) mean execution time per degree of freedom of a four-stage Runge-Kutta iteration on a single NVIDIA GeForce1080Ti GPU simulation of a series of hexahedra meshes of different polynomial orders p (see table 4). F is the flux evaluation of equation (8), $D_\xi F$ the flux discontinuous derivative, $C_\xi \Delta F$ the flux boundary correction, $D_\xi V$ the primitive variables discontinuous derivative, $C_\xi \Delta V$ the primitive variables derivative boundary correction, RK the Runge-Kutta kernels, BC the boundary condition kernels. The overall mean execution times per degree of freedom are compared against those obtained with PyFR[16].

5.1 Single GPU execution profiling

The kernel profiling for a single GPU has been addressed using a series of simulations of hexahedra meshes, whose number of cells and number of mesh points are presented in table 4. For each one, the number of cells is such that the total number of points remains almost constant for all p.

Figure 5a depicts the time budget for all kernels involved in the simulations. The evaluation of the primitive variables gradient and the fluxes takes nearly 80% of the computing time irregardless of the polynomial order, at least for orders up to 7. However, the relative weights between the parts change, with the correction kernels having more weight for lower degrees. The mean execution time per iteration and per degree of freedom for the most time-consuming kernels is represented in figure 5b. Degrees $p = 1$ and $p = 2$ are comparatively slower than degrees $p \geq 3$, whose cost per degree of freedom remains constant or even decreases slightly for the derivative correction kernels. This is a consequence of having just one cell per thread group. For small degrees that leads to thread groups with very few elements that do not take full advantage of the group operations, like coalesced memory accesses. For the kernel implementations presented here it is therefore better to use higher order polynomials to maximise the GPU efficiency.

Figure 5b also shows that for $p > 2$ and up to $p = 7$ the cost per degree of freedom does not depend on the polynomial order very much. This is a clear sign that the solver performance is memory bounded, and that the cost of the operations, even if it increases with p^3, is still far from being dominant, with hints of it affecting the cost only visible in the discontinuous derivatives of variables and fluxes.

The total execution times are also compared with those obtained with PyFR[1], a high order solver based on the flux reconstruction spatial discretisation[16]. The computational set-up of PyFR, including the solver precision, the node distributions for cells and faces and the temporal integrator, are exactly the same as those used for this work. The execution times for both codes are very similar, except for $p = 1$, with PyFR slightly increasing the execution time for higher p.

5.2 Multiple GPU execution profiling

The strong parallel scaling of Mu^2s^2T has been assessed using up to 8 cluster nodes and two $p = 5$ hexahedra meshes, one with $96 \times 96 \times 48$ cells and a smaller one with $48 \times 48 \times 48$. The GPU assignation for both cases is such that the number of GPUs sharing the x16 PCIe link is minimized, i.e., up to 16 processes, each GPU uses just one PCIe link to communicate with the host CPU. When that is no longer possible, the number of GPUS sharing a PCIe link grows sequentially, up until 2 GPUs per PCIe in the small case run using 32 MPI processes and 4 GPUs per PCIe in the large case run using 80 MPI processes. In order to highlight the effect of the GPU assignation policy, the smaller case has also been run with the opposite policy, i.e., maximizing the number of GPUs per PCIe link for each parallel simulation.

The mean time per four-stage Runge-Kutta iteration is depicted in figure 6a. For the large case, the parallel efficiency loss is not substantial up to 32 MPI processes, when comparing the execution time with the one obtained running with 4 MPI processes. That corresponds to $\sim 10^4$ cells per MPI process. For 64 and 80 MPI processes the GPU to GPU bandwidth degradation

[1]The PyFR used for the tests has commit SHA1 c9d02b004e70a71536cb423a0e0f39f66c59462a

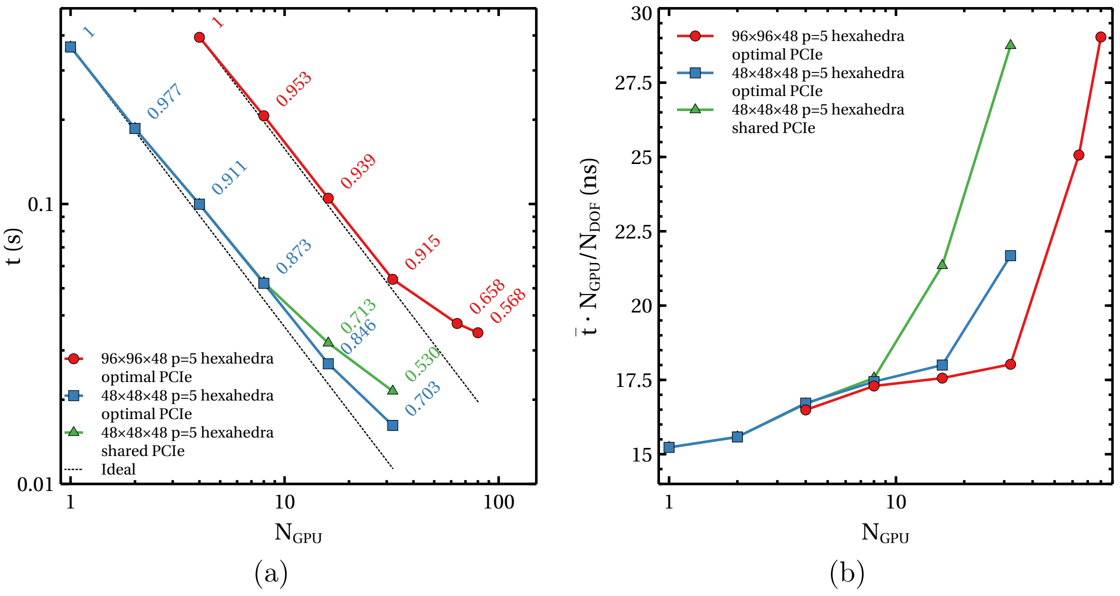


Figure 6: (a) Mean execution time and (b) mean execution time per mean number of DOF of a four-stage Runge-Kutta iteration for two different hexahedra meshes. The labels in the curve symbols represent the parallel efficiency.

due to the PCIe link sharing, together with the time reduction due to the smaller mesh partitions leads to a dramatic decrease of the parallel efficiency. For the smaller case, the parallel efficiency loss is more progressive when the PCIe link sharing is not that intensive, and the speed-up is not seriously degraded until the number of cells per MPI process is $\sim 5 \cdot 10^3$. However, if we force the PCIe sharing by assigning all MPI processes to GPUs within the same node, the communications cost surpasses the computations cost sooner, at $\sim 10^4$ cells per MPI process, and the parallel efficiency loss is deeper for the same number of MPI processes.

Figure 6b presents the same results using a different variable, the mean execution time per mean degree of freedom $\bar{t} \cdot N_{GPU}/N_{DOF}$, that clearly signals the point at which the communications cost is too high to be hidden and becomes dominant, changing the curve trend. While the communications cost is hidden, the increase in time per mean degree of freedom is due to the parallel simulations overheads: the gather and scatter of the variables to communicate and the loss of parallel efficiency of the GPU for smaller kernels. The extra cost, around 9% when the number of MPI processes is increased four-fold, seems to be fairly independent of the problem size at least for the small number of MPI processes tested in our examples. Whether this extra cost keeps increasing or reaches a plateau should be assessed running larger cases with more MPI processes. It has not been done in this work because the computing resources were unavailable.

6 Validation

6.1 Turbulent channel

The solver has undergone an extensive validation. First, we present three comparisons with DNS simulations of an incompressible channel flow at Re_τ 180[40], 950[41] and 2000[42]. To

Re_τ	p	N_x	N_y	N_z	y_w^+	$\Delta x_{\max}^+$	$\Delta y_{\max}^+$	$\Delta z_{\max}^+$	$t \cdot u_\tau/h$	GPU-hours
180	7	24	12	12	0.15	10	16	10	34	75
950	5	48	48	48	0.38	35.6	35.7	17.8	34	2970
2000	5	96	96	96	0.26	37.3	32.3	18.7	7	8400

Table 5: Parameters of the turbulent channel flow simulations. The channel dimensions are $L_x = 2\pi h$, $L_y = 2h$, $L_z = \pi h$ and the bulk Mach number is $M_b = 0.2$

speed-up the simulations we have used a bulk Mach number $M_b = 0.2$. The channel dimensions are $L_x = 2\pi h$, $L_z = \pi h$, $L_y = 2h$, which are sufficiently large to reproduce one-point statistics of larger boxes[43]. The element distribution is uniform in x and z directions. The y point distribution is obtained using the following formula proposed by Ghiasi et al.[44]:

$$\frac{y_n}{L_y} = \frac{1}{2}\left(1 - \frac{\tanh\left[\lambda\left(\frac{1}{2} - \frac{n}{n_p}\right)\right]}{\tanh(\lambda/2)}\right),\ 0 \le n \le n_p$$

where $n_p = N_y + 1$ is the number of points in the y direction and λ is a parameter to control the compactness of the point distribution near the wall. A value of $\lambda = 5.4$ has been used for all cases. The parameters for the simulations are detailed in table 5. The maximum mesh spacings are too large for a DNS simulation, which is a deliberate choice to assess the robustness of the method to perform under-resolved DNS simulations and to save computational time and resources.

Figure 7 depicts the instantaneous ρV field for the $z = 0$ plane for the three simulations, and figures 8 and 9 compare the resulting mean velocity and velocity fluctuations against the DNS simulations. The matching between DNS and Mu^2s^2T is very good for all compared quantities. The differences between the DNS simulation and our CFD in the $Re_\tau = 180$ case are very small. Mu^2s^2T slightly under-predicts the amplitude of the velocity fluctuations for $Re_\tau = 950$ and $Re_\tau = 2000$, which could be a consequence of the relatively coarse mesh used to simulate the flow or, in the $Re_\tau = 2000$ case, of the insufficient convergence of the RMS of the velocities.

6.2 LPT airfoil with upstream wakes

Next we present a comparison between Mu^2s^2T and some experimental measurements obtained at the Fluid Dynamics Laboratory of the Universidad Politécnica de Madrid (UPM). Details about the facility can be found in [45, 46]. It has been used to measure the aerodynamic performances of a linear cascade of LPT airfoils under unsteady inflow conditions generated by the wakes of an array of moving bars placed upstream. The flow disturbances produced by the incoming wakes strongly modify the suction side boundary layer separation and reattachment[47], thus controlling the aerodynamic losses of the LPT airfoil.

The facility operates at very low Mach numbers, therefore the flow regime is incompressible. The experiment aims at characterizing the aerodynamic behaviour of the 2D region of the airfoil for a range of Reynolds numbers and reduced frequencies f_r, where

$$f_r = \frac{V_b \cdot S}{\theta_b \cdot V_2}$$

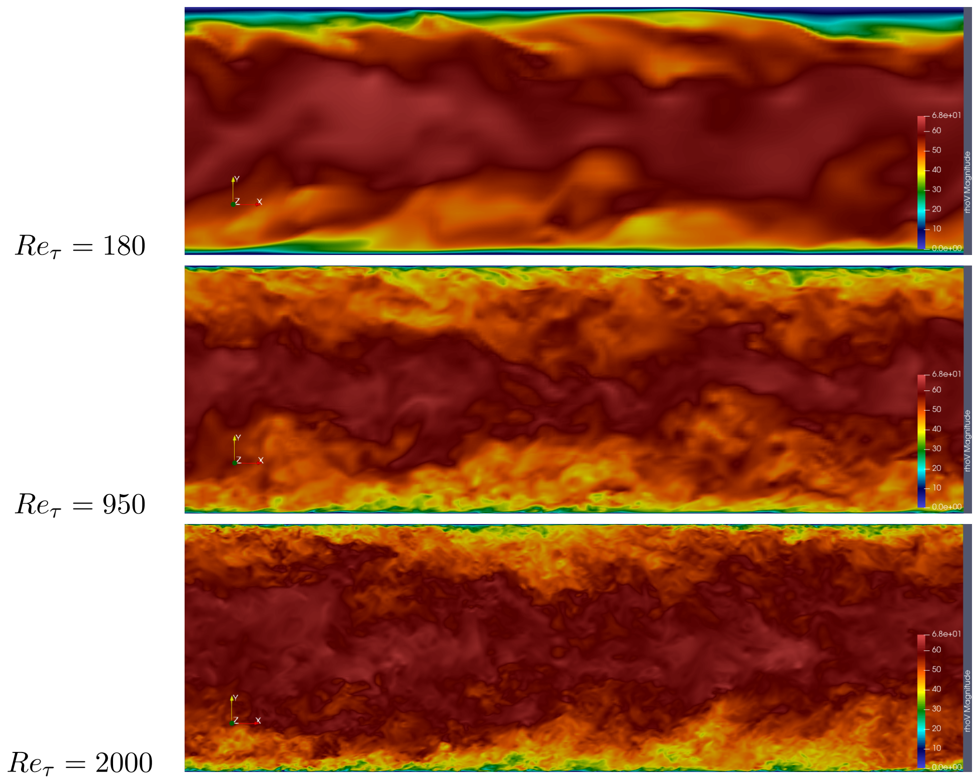


Figure 7: Instantaneous ρV field at $z = 0$ for the turbulent channel flow simulations at different Re_τ

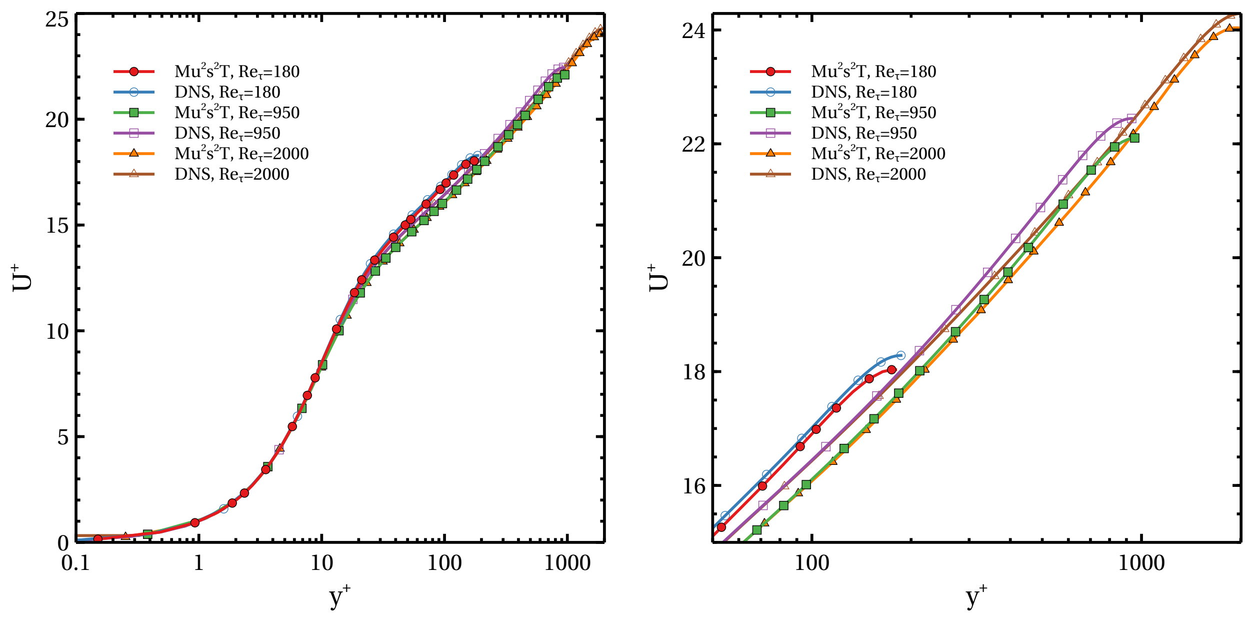
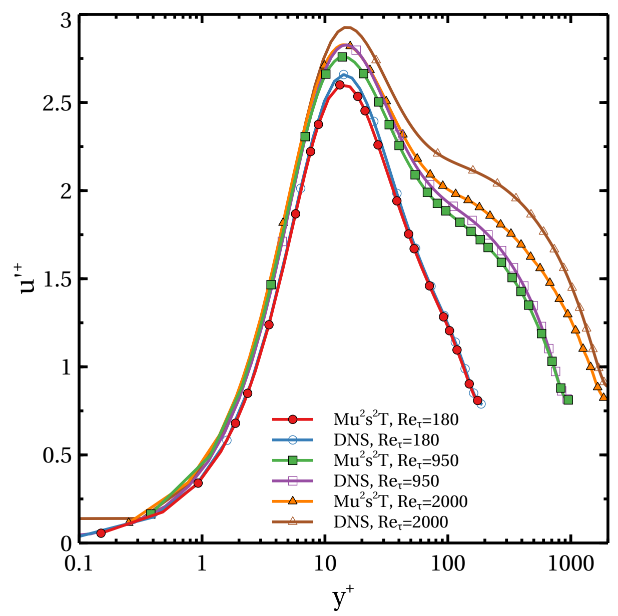
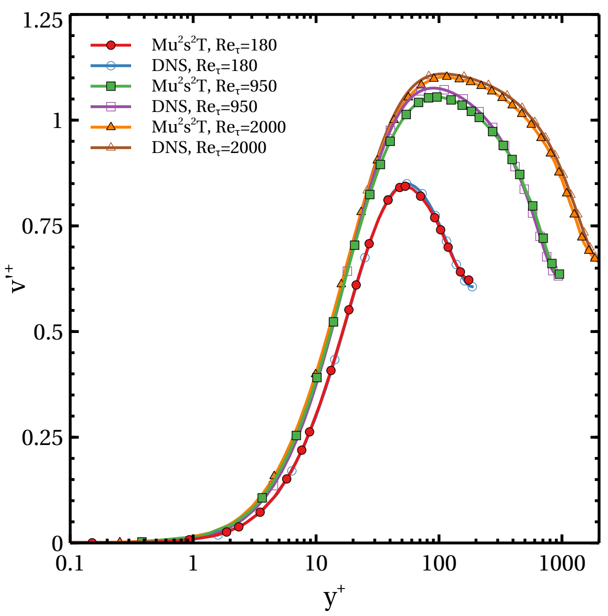


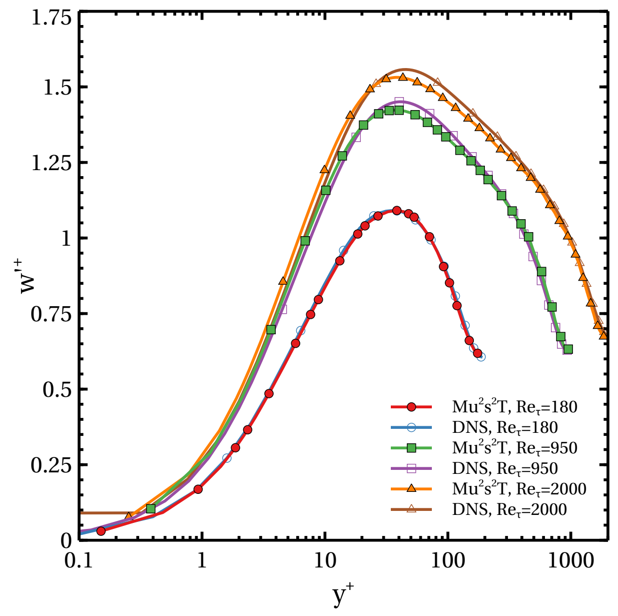
Figure 8: Comparison between Mu^2s^2T and DNS of the mean streamwise velocity for the turbulent channel flow at different Re_τ. Left: Full distribution. Right: Close-up of the logarithmic layer.



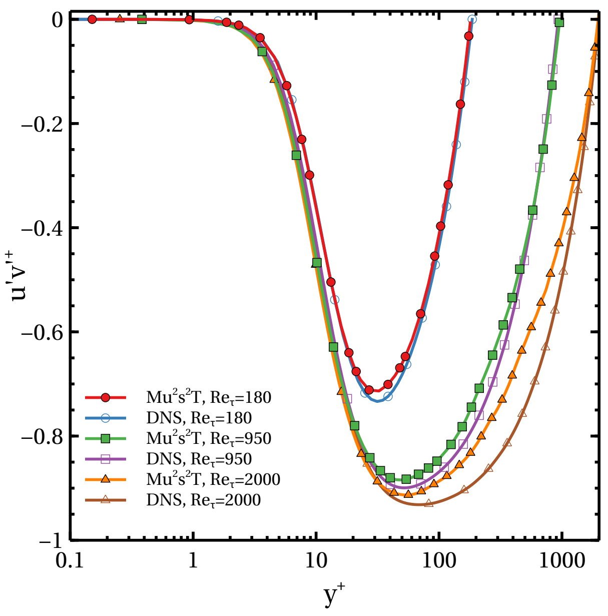
(a)



(b)



(c)



(d)

Figure 9: Comparison between Mu^2s^2T and DNS velocity fluctuations for the turbulent channel flow.

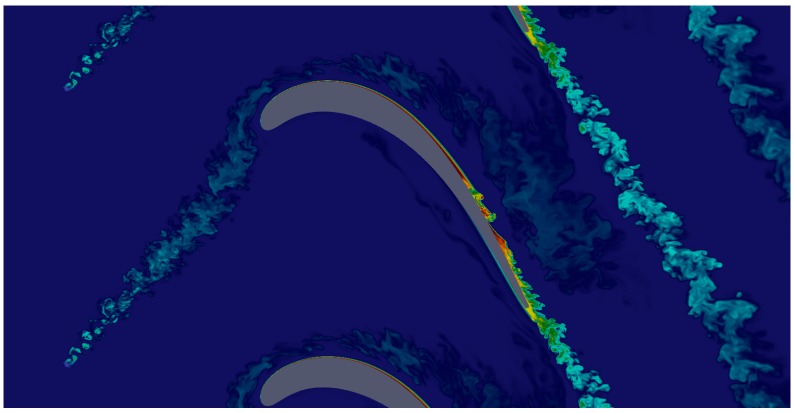


Figure 10: Instantaneous entropy solution for the simulation of the airfoil with upstream moving bars.

being V_b the bar velocity, θ_b the bar pitch, S the airfoil suction side perimeter and V_2 the mean exit flow velocity. The parameter that controls the wake inclination is the flow parameter

$$\phi = \frac{V_x}{V_b}$$

where V_x is the axial flow velocity.

The fluid domain has been discretized with a hybrid $p = 5$ mesh made of hexahedra for the boundary layer and wake regions and triangular prisms for the outer flow region. Pitchwise and spanwise periodic boundary conditions are used to simulate the 2D behaviour of one infinitely long airfoil pitch. The portion of span between spanwise periodic boundaries is long enough to allow developing the most unstable wavelengths. The resulting mesh has 1.97 million cells and 312 million DOFs. $y^+_{\text{max}} < 0.7$, $\Delta x^+ \approx 50$ and $\Delta z^+ \approx 20$ for the entire airfoil perimiter. The passing bars have been simulated used an immersed boundary condition[32], which reproduces the actual behaviour of a cylindrical bar after a few bar diameters. Comparisons at several Reynolds and reduced frequencies that have been experimentally tested and simulated have been published before[48, 46]. We present here a different simulation at $Re = 1.3 \cdot 10^5$, $\phi = 1.21$ and $f_r = 0.53$. We have simulated 20 blade passages, and taken statistics during the last 17 until the variation of the mean solution is deemed low enough as to ensure that the airfoil aerodynamic losses variation between periods is less than 1%. The cost of the simulation, running on ITP Aero's cluster, is 19900 GPU-hours. Figure 10 shows the instantaneous entropy field. The suction side boundary layer separates due to the adverse pressure gradient, the shear layer resulting from that separation is unstable and quickly transitions to turbulence. On top of that, the perturbations induced by the passing wakes modify the boundary layer dynamics leading to a reduction in overall aerodynamic losses.

The simulation results are compared against experimental measurements at the same Reynolds, $\phi = 1.09$ and $f_r = 0.52$. The discrepancy in flow parameter and reduced frequency is caused by the inability to simulate a single airfoil pitch configuration where the bar pitch θ_b is not a multiple of the airfoil pitch θ_a. Given certain flow conditions, by fixing θ_b and f_r, we fix V_b and therefore ϕ. Conversely, if we fix ϕ, we fix V_b and f_r is a result that cannot be chosen freely.

Figure 11 compares the measured and predicted airfoil C_p distributions. The simulation

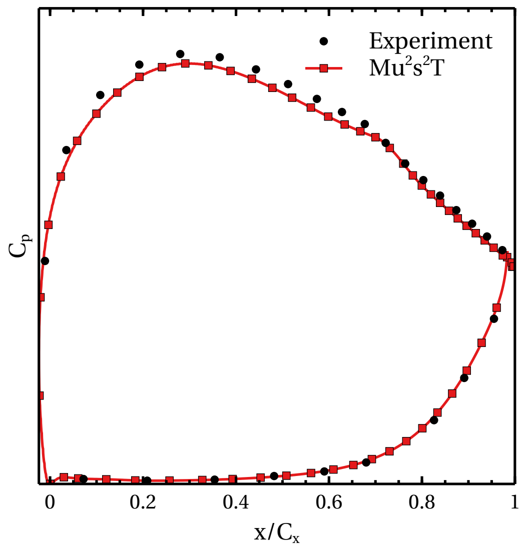


Figure 11: Comparison between measured and predicted airfoil C_p distributions.

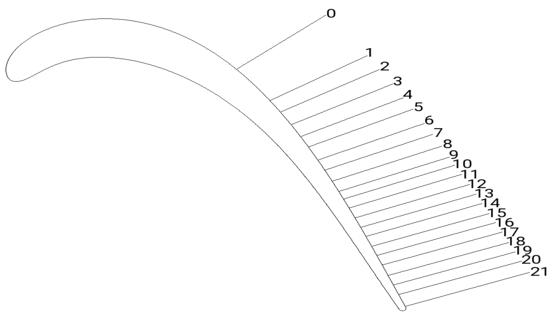


Figure 12: Airfoil stations where boundary layer profiles measurements are compared against Mu^2s^2T predictions.

suction side loading is lower than the measured, which could be due to a slight mismatch between the measured and simulated inflow angle due to the presence of the bars. The position of the suction side bubble and its reattachment, marked by the change of slope of the suction side C_p curve, are correctly reproduced.

Figure 13 compares the measured and predicted velocity profile, non-dimensionalized with the isentropic velocity[2], for several airfoil stations, whose location is depicted in figure 12. The experimental measurements have been obtained using hot wire anemometry. The simulation correctly predicts the location of the separation bubble, between stations 2 and 5, even though the velocity profiles there are a little bit different from the measured ones. After station 6 the agreement between the experimental measurements and the CFD predictions is quite good, with the simulation predicting a slightly larger velocity gradient next to the wall. When comparing

[2] $\overline{v}_{is}$ is defined using the outer flow mean values of total pressure, total temperature and velocity at each measurement station.

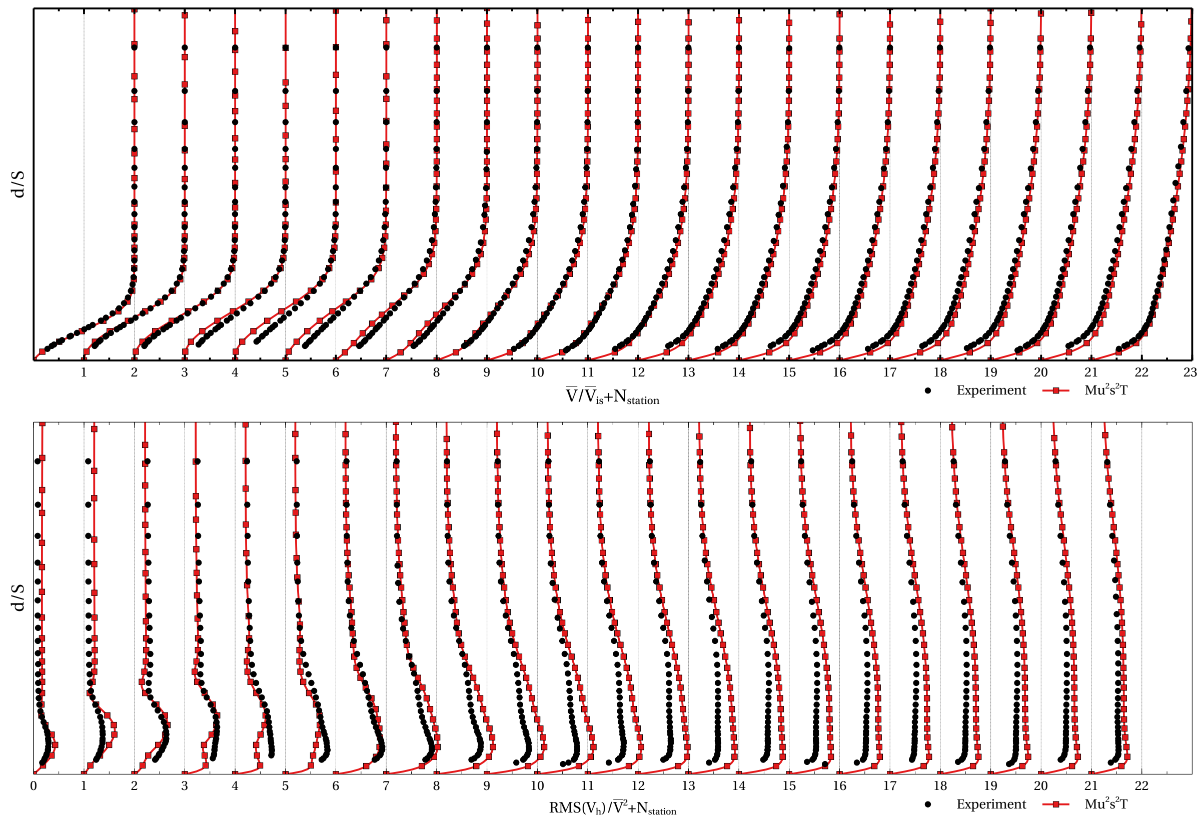


Figure 13: Comparison between measured and predicted non-dimensional mean velocity profiles (top) and non-dimensional mean 2D turbulent kinetic energy (bottom) for the measurement stations of figure 12.

the RMS of the horizontal velocity non-dimensionalized by the exit velocity squared there are important differences between the measured and predicted values, especially for the stations after the suction side bubble. The measurements predict lower RMS values. The origin of the discrepancies is not completely understood. It is known that the hot wire is unable to catch the highest frequencies of the velocity fluctuation due to the wire inertia, therefore yielding smaller RMS measurements. But to prove that some kind of RMS filtering analogous to the alleged behaviour of the hot wire should be implemented and that has not been done.

Figure 14 compares the measured and predicted mean velocity and mean velocity fluctuations at an axial location $0.5C_x$ downstream the airfoil trailing edge. The agreement for the mean velocity is deemed good, with the CFD correctly predicting the wake width. The discrepancies in the outer flow zone could be attributed to the different measured and simulated flow parameter. In the mean axial velocity fluctuations the conclusions are similar: the wake width is correctly predicted, but the outer flow region has some differences. The agreement between the experiment and the simulation is somewhat poorer for $v'v'$ and $u'v'$. The shapes of the distributions are very similar for both distributions, but there is an offset between the measured and simulated distributions, with the simulation predicting lower $v'v'$ and higher $u'v'$. More tests should be conducted to establish the origin of these discrepancies, which could be attributed either to the noise inherent to the LDV measurements, to the differences in flow coefficient between the experiment and the simulation or a combination of both.

7 Conclusions

The popularisation of many-core computing architectures, GPUs in particular, is driven by their superior computing performances, both in absolute terms and especially when the hardware acquisition cost is taken into account. The CFD solvers need to be adapted to these new computing platforms to better take advantage of their potential. We have presented the implementation of a high order solver for the Navier-Stokes equations based on the flux reconstruction scheme. The resulting implementation is optimized to be executed on multiple GPUs.

The details of the most time consuming parts of the solver are presented. The use of the so-called local or shared GPU memory is key in achieving an optimal performance of the kernels involved in the discontinuous derivatives and the derivatives corrections. The proposed implementation is not optimal for low polynomial degrees, but it achieves a good performance for $p > 2$, comparable to that of other high-order solvers with a similar formulation. The performance is entirely memory bounded, at least up to $p = 7$. Thus, the use of GPUs with higher memory bandwidth are expected to translate directly into a proportional gain in the solver performances.

When multiple GPUs are used in parallel, the main bottleneck is the time spent communicating data between them, because the resulting bandwidth is much lower than the bandwidth between the GPU memory and its processors. We have presented a strategy to maximize the overlap between communication and computation that, together with the communications pipelining, delays the point at which the communications cost becomes dominant.

The resulting solver has been used to conduct a series of simulations to validate its predictions either against DNS data of turbulent channels at several Re_τ or against experimental measurements of flows around low pressure turbine airfoils. The agreement is very good for all cases, proving it to be a valuable tool in future analyses of turbomachinery designs without the need

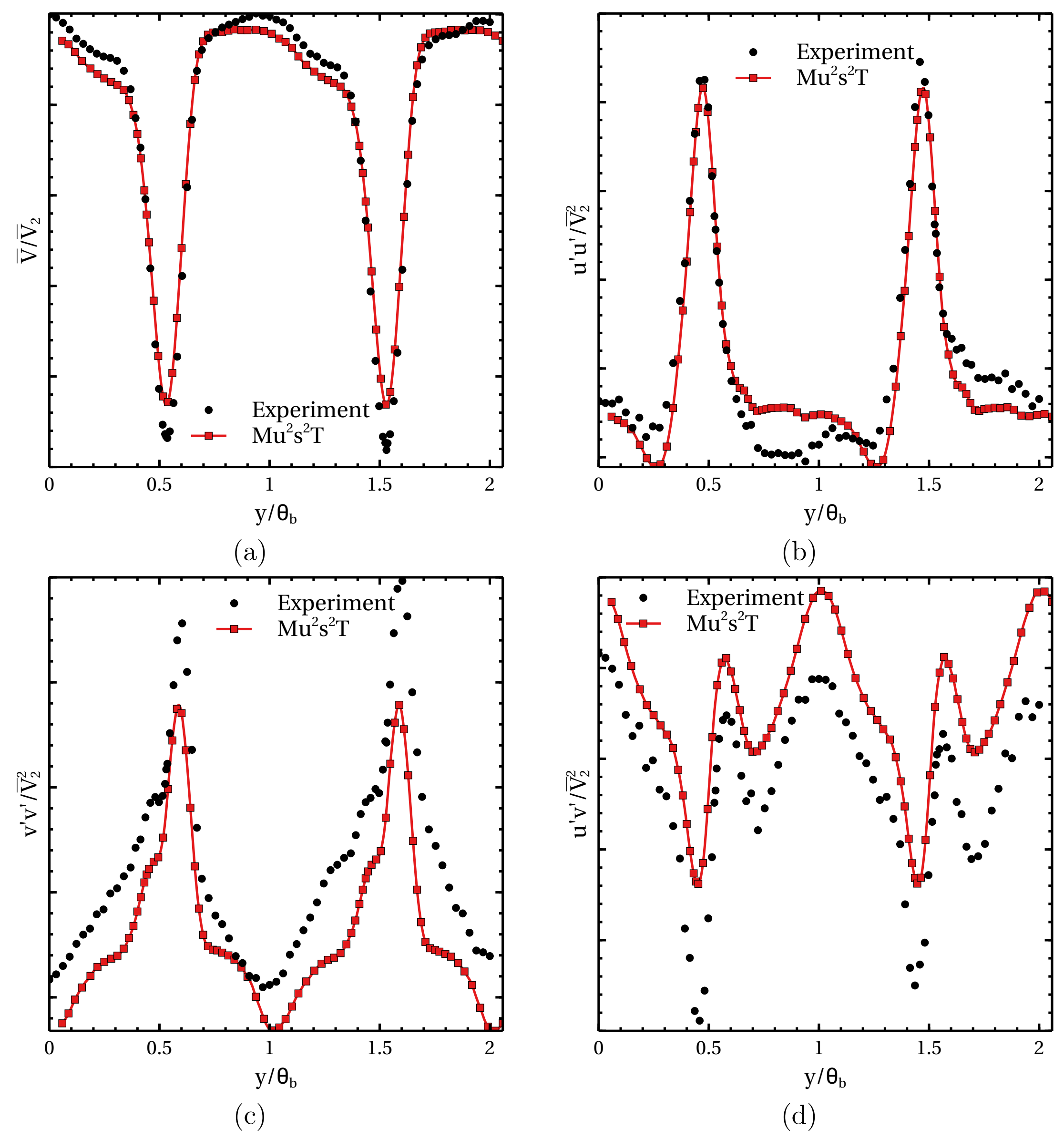


Figure 14: Comparison between measured and predicted non-dimensional mean velocity (a) and non-dimensional mean velocity fluctuations $u'u'$ (b), $v'v'$ (c) and $u'v'$ (d) at a measurement plane $0.5C_x$ downstream the airfoil trailing edge.

to resort to costly experimental campaigns.

Acknowledgments

The authors wish to thank ITP Aero for the permission to publish this work and to Patxi Apoita and David Cadrecha from ITP Aero Aerothermals department for providing the geometry and the experimental data of the low pressure turbine airfoil experiments. This work has been partially carried out under the Spanish Ministry of Economics collaboration contracts RTC-2017-6150-4 and RTC-2019-007194-4.

REFERENCES

[1] Wang, Z. J., "High-order computational fluid dynamics tools for aircraft design," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, Vol. 372, No. 2022, 2014, pp. 20130318.

[2] Cockburn, B., Karniadakis, G. E., and Shu, C.-W., "The Development of Discontinuous Galerkin Methods," *Discontinuous Galerkin Methods*, edited by B. Cockburn, G. E. Karniadakis, and C.-W. Shu, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 3–50.

[3] Shu, C.-W., "High-order Finite Difference and Finite Volume WENO Schemes and Discontinuous Galerkin Methods for CFD," *International Journal of Computational Fluid Dynamics*, Vol. 17, No. 2, 2003, pp. 107–118.

[4] Wang, Z., "High-order methods for the Euler and Navier-Stokes equations on unstructured grids," *Progress in Aerospace Sciences*, Vol. 43, No. 1, 2007, pp. 1–41.

[5] Huynh, H., Wang, Z., and Vincent, P., "High-order methods for computational fluid dynamics: A brief review of compact differential formulations on unstructured grids," *Computers & Fluids*, Vol. 98, 2014, pp. 209–220, 12th USNCCM mini-symposium of High-Order Methods for Computational Fluid Dynamics - A special issue dedicated to the 80th birthday of Professor Antony Jameson.

[6] Wang, Z. and Huynh, H., "A review of flux reconstruction or correction procedure via reconstruction method for the Navier-Stokes equations," *J-Stage Mechanical Engineering Reviews*, Vol. 3, 2016.

[7] Huynh, H., "A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods," *18th AIAA Computational Fluid Dynamics Conference*, No. 2007-4079, 2007.

[8] Huynh, H., "A Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin for Diffusion," *47th AIAA Aerospace Sciences Meeting*, No. 2009-403, 2009.

[9] Wang, Z. and Gao, H., "A unifying lifting collocation penalty formulation including the discontinuous Galerkin, spectral volume/difference methods for conservation laws on mixed grids," *Journal of Computational Physics*, Vol. 228, No. 21, 2009, pp. 8161 – 8186.

[10] NVIDIA, “CUDA Programming Guide,” `https://docs.nvidia.com/cuda/cuda-c-programming-guide`, June 2018.

[11] Khronos Group, “OpenCL 1.2 Specification,” `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`, November 2011.

[12] Klöckner, A., Warburton, T., Bridge, J., and Hesthaven, J., “Nodal discontinuous Galerkin methods on graphics processors,” *Journal of Computational Physics*, Vol. 228, No. 21, 2009, pp. 7863 – 7882.

[13] Castonguay, P., Williams, D., Vincent, P., Lopez, M., and Jameson, A., “On the Development of a High-Order, Multi-GPU Enabled, Compressible Viscous Flow Solver for Mixed Unstructured Grids,” *20th AIAA Computational Fluid Dynamics Conference*, No. 2011-3229, 2011.

[14] Siebenborn, M., Schulz, V., and Schmidt, S., “A curved-element unstructured discontinuous Galerkin method on GPUs for the Euler equations,” *Computing and Visualization in Science*, Vol. 15, No. 2, April 2012, pp. 61–73.

[15] López-Morales, M., Bull, J., Crabill, J., Economon, T., Manosalvas, D., Romero, J., Sheshadri, A., II, J. W., Williams, D., Palacios, F., and Jameson, A., “Verification and Validation of HiFiLES: a High-Order LES unstructured solver on multi-GPU platforms,” *32nd AIAA Applied Aerodynamics Conference*, No. 2014-3168, 2014.

[16] Witherden, F., Farrington, A., and Vincent, P., “PyFR: An open source framework for solving advection - diffusion type problems on streaming architectures using the flux reconstruction approach,” *Computer Physics Communications*, Vol. 185, No. 11, 2014, pp. 3028 – 3040.

[17] Chan, J., Wang, Z., Modave, A., Remacle, J., and Warburton, T., “GPU-accelerated discontinuous Galerkin methods on hybrid meshes,” *Journal of Computational Physics*, Vol. 318, No. 1, August 2016, pp. 142–168.

[18] Romero, J., Crabill, J., Watkins, J., Witherden, F., and Jameson, A., “ZEFR: A GPU-accelerated high-order solver for compressible viscous flows using the flux reconstruction method,” *Computer Physics Communications*, Vol. 250, 2020, pp. 107169.

[19] de Araujo Jorge Filho, E. J. and Wang, Z. J., “Efficient Implementation of the FR/CPR Method on GPU Clusters for Industrial Large Eddy Simulation,” *AIAA AVIATION 2020 FORUM*, 2020.

[20] Corral, R., Gisbert, F., and Pueblas, J., “Execution of a parallel edge-based Navier-Stokes solver on commodity graphics processor units,” *International Journal of Computational Fluid Dynamics*, Vol. 31, No. 2, 2017, pp. 93–108.

[21] Haga, T., Gao, H., and Wang, Z., “A High-Order Unifying Discontinuous Formulation for the Navier-Stokes equations on 3D Mixed Grids,” *Mathematical Modelling of Natural Phenomena*, Vol. 6, No. 3, 2011, pp. 28–56.

[22] Warburton, T., "An explicit construction of interpolation nodes on the simplex," *Journal of Engineering Mathematics*, Vol. 56, No. 3, 2006, pp. 247–262.

[23] Hesthaven, J. S. and Warburton, T., *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, Springer Publishing Company, Incorporated, 1st ed., 2007.

[24] Chan, J. and Warburton, T., "A Comparison of High-Order Interpolation Nodes for the Pyramid," *SIAM Journal of Scientific Computing*, Vol. 37, No. 5, 2015, pp. A2151–A2170.

[25] Roe, P., "Approximate Riemann solvers, parameter vectors, and difference schemes," *Journal of Computational Physics*, Vol. 43, 1981, pp. 357–372.

[26] Kennedy, C. A. and Gruber, A., "Reduced aliasing formulations of the convective terms within the Navier-Stokes equations for a compressible fluid," *Journal of Computational Physics*, Vol. 227, No. 3, 2008, pp. 1676–1700.

[27] Pirozzoli, S., "Generalized conservative approximations of split convective derivative operators," *Journal of Computational Physics*, Vol. 229, No. 19, 2010, pp. 7180–7190.

[28] Coppola, G., Capuano, F., Pirozzoli, S., and de Luca, L., "Numerically stable formulations of convective terms for turbulent compressible flows," *Journal of Computational Physics*, Vol. 382, April 2019, pp. 86–104.

[29] Lodato, G., Domingo, P., and Vervisch, L., "Three-dimensional boundary conditions for direct and large-eddy simulation of compressible viscous flows," *Journal of Computational Physics*, Vol. 227, No. 10, 2008, pp. 5105–5143.

[30] Odier, N., Poinsot, T., Duchaine, F., Gicquel, L., and Moreau, S., "Inlet and Outlet Characteristics Boundary Conditions for Large Eddy Simulations of Turbomachinery," *Turbo Expo: Power for Land, Sea and Air*, No. GT2019-90747, 2019.

[31] Shur, M. L., Spalart, P. R., Strelets, M. K., and Travin, A. K., "Synthetic Turbulence Generators for RANS-LES Interfaces in Zonal Simulations of Aerodynamic and Aeroacoustic Problems," *Flow, Turbulence and Combustion*, Vol. 93, 2014, pp. 63–92.

[32] Goldstein, D., Handler, R., and Sirovich, L., "Modeling a no-slip flow boundary with an external force field," *Journal of computational physics*, Vol. 105, No. 2, 1993, pp. 354–366.

[33] Kahan, W., "Further remarks on reducing truncation errors," *Communications of the ACM*, Vol. 8, No. 1, 1965.

[34] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable Parallel Programming with CUDA," *Queue*, Vol. 6, No. 2, March 2008, pp. 40–53.

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

[36] NVIDIA, “cuBLAS,” `https://docs.nvidia.com/cuda/cublas`, June 2018.

[37] Rupp, K., Tillet, P., Rudolf, F., Weinbub, J., Morhammer, A., Grasser, T., JÃŒngel, A., and Selberherr, S., “ViennaCL—Linear Algebra Library for Multi- and Many-Core Architectures,” *SIAM Journal on Scientific Computing*, Vol. 38, No. 5, 2016, pp. S412–S439.

[38] Karypis, G., Schloegel, K., and Kumar, V., “ParMeTiS, Parallel Graph Partitioning and Sparse Matrix Ordering Library, version 3.1,” August 15 2003.

[39] Gisbert, F., Bolinches-Gisbert, M., Pueblas, J., and Corral, R., “Efficient implementation of Flux Reconstruction schemes for the simulation of compressible viscous flows on Graphics Processing Unigs,” *Tenth International Conference on Computational Fluid Dynamics (ICCFD10)*, No. ICCFD10-307, Barcelona, Spain, July 2018.

[40] del Álamo, J. C. and Jiménez, J., “Spectra of the very large anisotropic scales in turbulent channels,” *Physics of Fluids*, Vol. 15, No. 6, 2003, pp. L41–L44.

[41] del Álamo, J. C., Jiménez, J., Zandonade, P., and Moser, R. D., “Scaling of the energy spectra of turbulent channels,” *Journal of Fluid Mechanics*, Vol. 500, 2004, pp. 135–144.

[42] Hoyas, S. and Jiménez, J., “Scaling of the velocity fluctuations in turbulent channels up to $Re_\tau = 2003$,” *Physics of Fluids*, Vol. 18, No. 1, 2006, pp. 011702.

[43] Lozano-Durán, A. and Jiménez, J., “Effect of the computational domain on direct simulations of turbulent channels up to $Re_\tau = 4200$,” *Physics of Fluids*, 2014.

[44] Ghiasi, Z., Li, D., Komperda, J., and Mashayek, F., “Near-wall resolution requirement for direct numerical simulation of turbulent flow using multidomain Chebyshev grid,” *International Journal of Heat and Mass Transfer*, Vol. 126, 2018, pp. 746–760.

[45] Bolinches-Gisbert, M., Corral, R., Cadrecha, D., and Gisbert, F., “Prediction of Reynolds number effects on Low Pressure Turbines using a high order ILES method,” *Proceedings of the ASME2019 Turbo Expo*, No. GT2019-91346, 2019.

[46] Bolinches-Gisbert, M., Robles, D. C., Corral, R., and Gisbert, F., “Numerical and Experimental Investigation of the Reynolds Number and Reduced Frequency Effects on Low-Pressure Turbine Airfoils,” *Journal of Turbomachinery*, Vol. 143, February 2021.

[47] Coull, J. D. and Hodson, H. P., “Unsteady boundary-layer transition in low-pressure turbines,” *Journal of Fluid Mechanics*, Vol. 681, 2011, pp. 370–410.

[48] Bolinches-Gisbert, M., Robles, D. C., Corral, R., and Gisbert, F., “Prediction of Reynolds Number Effects on Low-Pressure Turbines Using a High-Order ILES Method,” *Journal of Turbomachinery*, Vol. 142, No. 3, 02 2020, 031002.