STREAmS

A high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows

Bernardini, Matteo; Modesti, Davide; Salvadore, Francesco; Pirozzoli, Sergio

DOI
10.1016/j.cpc.2021.107906

Publication date
2021

Document Version
Final published version

Published in
Computer Physics Communications

Computer Physics Communications 263 (2021) 107906

Contents lists available at ScienceDirect

Computer Physics Communications

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

STREAmS: A high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows☆,☆☆

Matteo Bernardini [a], Davide Modesti [b], Francesco Salvadore [c,*], Sergio Pirozzoli [a]

[a] Dipartimento di Ingegneria Meccanica e Aerospaziale, Sapienza Università di Roma, via Eudossiana 18, 00184 Roma, Italy
[b] Aerodynamics Group, Faculty of Aerospace Engineering, Delft University of Technology, Kluyverweg 2, 2629 HS Delft, The Netherlands
[c] HPC Department, CINECA, Rome office, via dei Tizii 6/B, 00185 Roma, Italy

ARTICLE INFO

Article history:
Received 21 August 2020
Received in revised form 7 November 2020
Accepted 21 December 2020
Available online 20 February 2021

Keywords:
GPUs
CUDA
Compressible flows
Wall turbulence
Direct numerical simulation
Open source

ABSTRACT

We present STREAmS, an in-house high-fidelity solver for direct numerical simulations (DNS) of canonical compressible wall-bounded flows, namely turbulent plane channel, zero-pressure gradient turbulent boundary layer and supersonic oblique shock-wave/boundary layer interaction. The solver incorporates state-of-the-art numerical algorithms, specifically designed to cope with the challenging problems associated with the solution of high-speed turbulent flows and can be used across a wide range of Mach numbers, extending from the low subsonic up to the hypersonic regime. From the computational viewpoint, STREAmS is oriented to modern HPC platforms thanks to MPI parallelization and the ability to run on multi-GPU architectures. This paper discusses the main implementation strategies, with particular reference to the CUDA paradigm, the management of a single code for traditional and multi-GPU architectures, and the optimization process to take advantage of the latest generation of NVIDIA GPUs. Performance measurements show that single-GPU optimization more than halves the computing time as compared to the baseline version. At the same time, the asynchronous patterns implemented in STREAmS for MPI communications guarantee very good parallel performance especially in the weak scaling spirit, with efficiency exceeding 97% on 1024 GPUs. For overall evaluation of STREAmS with respect to other compressible solvers, comparison with a recent GPU-enabled community solver is presented. It turns out that, although STREAmS is much more limited in terms of flow configurations that can be addressed, the advantage in terms of accuracy, computing time and memory occupation is substantial, which makes it an ideal candidate for large-scale simulations of high-Reynolds number, compressible wall-bounded turbulent flows. The solver is released open source under GPLv3 license.

Program summary

Program Title: STREAmS
CPC Library link to program files: https://doi.org/10.17632/hdcgjpzr3y.1
Developer's repository link: https://github.com/matteobernardini/STREAmS
Code Ocean capsule: https://codeocean.com/capsule/8931507/tree/v2
Licensing provisions: GPLv3
Programming language: Fortran 90, CUDA Fortran, MPI
Nature of problem: Solving the three-dimensional compressible Navier–Stokes equations for low and high Mach regimes in a Cartesian domain configured for channel, boundary layer or shock-boundary layer interaction flows.
Solution method: The convective terms are discretized using a hybrid energy-conservative shock-capturing scheme in locally conservative form. Shock-capturing capabilities rely on the use of Lax–Friedrichs flux vector splitting and weighted essentially non-oscillatory (WENO) reconstruction. The system is advanced in time using a three-stage, third-order RK scheme. Two-dimensional pencil distributed MPI parallelization is implemented alongside different patterns of GPU (CUDA Fortran) accelerated routines.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

☆☆ The review of this paper was arranged by Prof. Hazel Andrew.

* Corresponding author.
E-mail address: f.salvadore@cineca.it (F. Salvadore).

https://doi.org/10.1016/j.cpc.2021.107906
0010-4655/

1. Introduction

Compressible flows are ubiquitous in aerospace applications and in recent years there has been a renewed interest in the field, owing to the rising investments in high-speed flight and space exploration. These technological challenges call attention to high-fidelity numerical methods for compressible wall-bounded flows which have proved to be a valuable tool to unveil the complexity of these flows.

The flow physics of compressible wall-bounded turbulence is undoubtedly richer than in incompressible flows. The hyperbolic nature of the equations allows for the presence of propagating disturbances and discontinuities such as shock waves, which interact with the underlying turbulence, leading to flow phenomena which are absent in the incompressible case. This additional complexity has affected and slowed down the development of numerical methods for compressible flows, as compared to the incompressible ones. Baseline numerical algorithms for direct numerical simulation (DNS) of incompressible flows were mainly developed between the sixties and the eighties [1–4], and basically settled since then. The reliability of these algorithms and the advent of the open-source software promoted the development of several incompressible open-source solvers for fluid dynamics, both multi-purpose solvers as OpenFOAM [5], Nek5000 [6] and Nektar++ [7] and academic solvers as AFiD [8] and CaNS [9]. These solvers are based on central processing units (CPUs) and message passing interface (MPI) parallelization, which has been the standard approach in high-performance computing (HPC) in the past twenty years. However, in the race towards exascale computing, the HPC architectures are showing consistent trend towards the use of graphical processing units (GPUs). In the last decade, GPUs have become the favorite solution to achieve accelerated cutting-edge performance with high energy efficiency. In particular, in the second 2019 Top500 survey [10], which reports the ranking of the most powerful 500 machines worldwide, 136 machines are NVIDIA GPU-Accelerated [11] for a total of about 40% of the total power supplied. In addition, NVIDIA GPUs power 90% of the top 30 supercomputers on the Green500 [12], a list of HPC systems with high performance and improved energy efficiency. The incompressible DNS community has already benefited from improved computational performance of GPUs, with two available in-house solvers AFiD-GPU [13] and CaNS-GPU [14].

Numerical algorithms for DNS of compressible flows are less standardized then the incompressible ones, as several formulations of the underlying equations are possible [15,16], each proving numerical advantages depending on the flow physics involved. For this reason fewer open-source compressible flow solvers are available, compared to the incompressible case. Examples include popular multi-purpose open-source packages [5,7,17,18] and OpenSBLI [19], a Python framework for the automated derivation of finite differences solvers both for CPUs and GPUs architectures. Another option is the use of the recent programming paradigm Legion [20] which allows users to use the same solver on different HPC architectures (including GPUs), without requiring extensive code restructuring. A recent example of compressible flow solver using Legion is HTR [21], designed for hypersonic reacting flows. Other examples of compressible open-source flow solvers running on GPUs include PyFr [22] and ZEFR [23], which are both general-purpose, unstructured flow solvers based on high-order flux reconstruction. Those solvers were designed to solve a range of governing systems on mixed unstructured grids containing various element types, thus providing the opportunity of simulating compressible flows in complex geometries. The realization of codes capable of adequately simulating compressible flows in complex geometries represents a significant step forward, especially from the point of view of real applications. However, solvers such as STREAmS, focused on the simulation of canonical flows for basic research on turbulence, can offer considerable advantages over general-purpose solvers. First, consolidated energy-preserving schemes as those implemented in STREAmS, represent the state-of-the-art solution for DNS/LES of shock-free turbulent flows, allowing to accurately simulate the wide range of spatial and temporal scales typical of turbulence without relying on numerical (artificial) diffusivity. Those schemes can be efficiently combined with modern shock-capturing methods as weighted essentially non oscillatory (WENO) reconstructions, yielding hybrid schemes that currently represent an optimal strategy for the computation of shocked flows [24]. Moreover, STREAmS offers tailored boundary conditions, such as digital filtering for turbulent inflow, and shock injection for the simulation shock/boundary-layer interactions. Finally, as will be extensively discussed in this work, a Cartesian code such as STREAmS achieves very high efficiency both in terms of memory consumption and computater time, yielding remarkable speed-up with respect to general-purpose solvers.

The STREAmS CPU solver stems from 20 years experience in our group on compressible wall-bounded flows, and it was used to carry out several seminal DNS studies of canonical flows including supersonic boundary layers [25,26], shock/boundary layer interactions (SBLI) [27,28], supersonic roughness-induced transition [29,30] and supersonic internal flows [31,32]. The first core of the code, without shock-capturing functionality and limited to the use of a single GPU, was successfully ported to previous generations of NVIDIA GPUs, showing significant advantage of this type of architecture [33].

In this work we present STREAmS (Supersonic TuRbulEnt Accelerated navier–stokes Solver), a CUDA Fortran version of the solver, developed and optimized for the latest generation of GPU clusters. The solver is targeted to three main canonical wall-bounded turbulent flows, namely the supersonic plane channel, the zero-pressure-gradient boundary layer developing over a flat plate and the oblique shock wave/turbulent boundary layer interaction. In the following, the key points of the implemented algorithms are described and a brief validation of the analyzed flows is presented. Then, CUDA implementation is discussed alongside with the various stages of single-node optimization. Finally, the scalability properties are presented and the computational performance compared with both the CPU version of the same solver, and with the GPU solver ZEFR [23].

2. Methodology

STREAmS solves the fully compressible Navier–Stokes equations for a perfect heat-conducting gas

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0, \tag{1a}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial \sigma_{ij}}{\partial x_j} + f\delta_{i1}, \tag{1b}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_j H}{\partial x_j} = -\frac{\partial q_j}{\partial x_j} + \frac{\partial \sigma_{ij} u_i}{\partial x_j} + f u_1, \tag{1c}$$

where u_i, $i = 1, 2, 3$, is the velocity component in the ith direction, ρ the density, p the pressure, $E = c_v T + u_i u_i/2$ the total energy per unit mass, and $H = E + p/\rho$ is the total enthalpy. The components of the heat flux vector q_j and of the viscous stress tensor σ_{ij} are

$$\sigma_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3} \frac{\partial u_k}{\partial x_k} \delta_{ij} \right), \tag{2}$$

$$q_j = -k \frac{\partial T}{\partial x_j}, \tag{3}$$

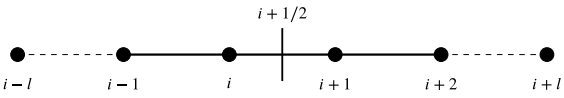

Fig. 1. Schematic of the computational stencil in one space direction.

where the dependence of the viscosity coefficient on temperature is accounted for through Sutherland's law and $k = c_p\mu/Pr$ is the thermal conductivity, with $Pr = 0.72$. The forcing term f in Eq. (1b) is added in the plane channel flow simulations and is evaluated at each time step in order to discretely enforce constant mass-flow-rate in time. The corresponding power spent is added to the right-hand-side of the total energy equation.

2.1. Spatial discretization

The convective terms in the Navier–Stokes equations are discretized using a hybrid energy-preserving/shock-capturing scheme in locally conservative form [34]. Let us consider the convective flux in one space direction (say x)

$$f_x = \rho u \varphi, \tag{4}$$

where φ is the transported quantity, namely $\varphi = 1$ for the mass equation, $\varphi = u_j$ for the momentum equation in the jth direction and H for the total energy equation. The numerical discretization of the streamwise derivative of the flux f_x on a uniform mesh with spacing Δx relies on the identification of a numerical flux $\hat{f}_{x\,i+1/2}$ defined at the intermediate nodes such that

$$\left.\frac{\partial f_x}{\partial x}\right|_i = \frac{1}{\Delta x}\left(\hat{f}_{x,\,i+1/2} - \hat{f}_{x,\,i-1/2}\right). \tag{5}$$

An energy-preserving numerical flux at the interface $i + 1/2$ (Fig. 1) can be obtained by defining the three-point averaging operator [34]

$$\left(\widetilde{F, G, J}\right)_{i,l} = \frac{1}{8}\left(F_i + F_{i+l}\right)\left(G_i + G_{i+l}\right)\left(J_i + J_{i+l}\right), \tag{6}$$

and recasting in conservative form the split formulation of the Eulerian fluxes [35]

$$\hat{f}_{x,\,i+1/2} = 2\sum_{l=1}^{L} a_l \sum_{m=0}^{l-1}\left(\widetilde{\rho, u, \varphi}\right)_{i-m,l}, \tag{7}$$

where the a_l are the standard coefficients for central finite-difference approximations of the first derivative, yielding order of accuracy $2L$. In smooth (shock-free) regions of the flow STREAmS applies an energy-consistent flux (7), which guarantees that the total kinetic energy is discretely conserved in the limit case of inviscid incompressible flow [27]. The order of accuracy of the discretization can be selected by the user and ranges from the second up to the eighth order. The locally conservative formulation allows straightforward hybridization of the central flux with classical shock-capturing reconstructions. In our case, shock-capturing capabilities rely on the use of the Lax–Friedrichs flux vector splitting, whereby the components of the positive and negative characteristic fluxes are reconstructed at the interfaces using a weighted essentially non-oscillatory (WENO) reconstruction [36]. Similarly to the central flux, the order of accuracy of the shock capturing scheme can be changed from first to seventh order. To judge on the local smoothness of the numerical solution and switch between the energy preserving and the shock capturing discretization, STREAmS relies on a modified version of the Ducros shock sensor [37]

$$\theta = \max\left(\frac{-\nabla\cdot u}{\sqrt{\nabla\cdot u^2 + \nabla\times u^2 + u_0^2/L_0}}, 0\right) \in [0,\,1], \tag{8}$$

where u_0 and L_0 are suitable velocity and length scales [25], which remain fixed during the simulation. The sensor is designed to be $\theta \approx 0$ in smooth flow regions and $\theta \approx 1$ in the presence of shock waves. For turbulent channel flow u_0 and L_0 are the bulk flow velocity and channel half width, whereas for boundary layer they are the free-stream velocity and the inflow boundary layer thickness. The viscous terms are expanded to Laplacian form to avoid odd–even decoupling phenomena, and approximated with central finite-difference formulas (up to eighth order)

$$\begin{aligned}\frac{\partial}{\partial x}\left(\mu\left.\frac{\partial u}{\partial x}\right|_i\right)\Bigg|_i &= \left.\frac{\partial \mu}{\partial x}\right|_i\left.\frac{\partial u}{\partial x}\right|_i + \mu\left.\frac{\partial^2 u}{\partial x^2}\right|_i = \\ &\frac{1}{\Delta x^2}\sum_{l=-L}^{L} a_l^2\,\mu_{i+l}u_{i+l} + \mu_i\frac{1}{\Delta x^2}\sum_{l=-L}^{L} b_l u_{i+l},\end{aligned} \tag{9}$$

where b_l are the finite difference coefficients for the second derivative of order $2L$.

2.2. Time integration

A semi-discrete system of ordinary differential equations stems from discretization of the spatial derivatives,

$$\frac{\mathrm{d}\mathbf{w}}{\mathrm{d}t} = \mathbf{R}(\mathbf{w}) \tag{10}$$

where $\mathbf{w} = [\rho, \rho u, \rho v, \rho w, \rho E]$ is the vector of the conservative variables and $\mathbf{R}$ the vector of the residuals. The system is advanced in time using a three-stage, third-order Runge–Kutta scheme [38],

$$\mathbf{w}^{(\ell+1)} = \mathbf{w}^{(\ell)} + \alpha_\ell \Delta t\mathbf{R}^{(\ell-1)} + \beta_\ell \Delta t\mathbf{R}^{(\ell)}, \quad \ell = 0,\,1,\,2, \tag{11}$$

$\mathbf{w}^{(0)} = \mathbf{w}^n$, $\mathbf{w}^{n+1} = \mathbf{w}^{(3)}$ and the integration coefficients are $\alpha_\ell = (0,\,17/60,\,-5/12)$, $\beta_\ell = (8/15,\,5/12,\,3/4)$.

As previously noted, the numerical solution of the compressible Navier–Stokes equations is in general terms more computationally demanding than the incompressible version. This is certainly due to the presence of additional terms and equations, but also to the acoustic time step limitation which is absent in the incompressible case. The Euler equations in characteristic form are a coupled system of nonlinear advection equations, with the advection velocities corresponding to the eigenvalues of the system. Hence, the maximum eigenvalue in the ith direction $\lambda_{imax} = u_i + c$ (where c is the local speed of sound) embeds a convective (u_i) and an acoustic (c) contribution. With reference to the flow cases of interest for STREAmS, namely wall-bounded compressible turbulent flows, the inviscid time step limitation in the coordinate directions (x,y,z) can be expressed as

$$\begin{aligned}\Delta t_x^+ &= \frac{\Delta x^+}{\max(u_0^+ + c_0^+, c_w^+)} = \Delta x^+ M_0\sqrt{\frac{C_f}{2}}\min\left(1, \frac{\sqrt{T_w/T_0}}{1+M_0}\right),\\ \Delta t_y^+ &= \frac{\Delta y^+}{c_w^+} = \Delta y^+ M_0\sqrt{\frac{C_f}{2}}\\ \Delta t_z^+ &= \frac{\Delta z^+}{\max(c_0^+, c_w^+)} = \Delta z^+ M_0\sqrt{\frac{C_f}{2}}\min\left(1, \sqrt{T_w/T_0}\right),\end{aligned} \tag{12}$$

where unit CFL number is assumed, Δx and Δz are the uniform mesh spacings in the streamwise and spanwise directions, and Δy is the minimum mesh spacing in the wall-normal direction. The plus superscript is used to denote quantities made nondimensional with respect to local wall units, namely the friction velocity $u_\tau = (\tau_w/\rho_w)^{1/2}$, and the viscous length scale $\delta_v = \nu_w/u_\tau$, where τ_w is the wall shear stress. The subscript 0 denotes bulk flow properties (for channels) or at the free-stream (for boundary layers), and w indicates wall properties, with the skin friction coefficient given by $C_f = 2\tau_w/(\rho_0 u_0^2)$.

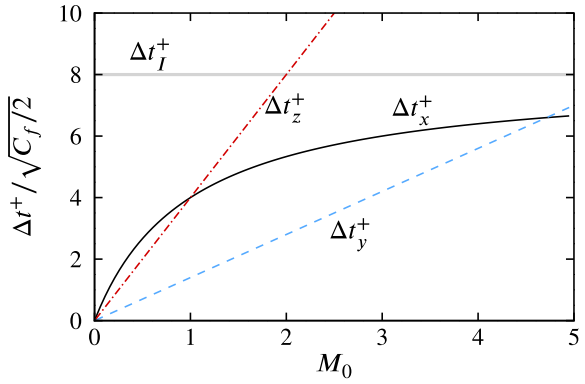

Fig. 2. Inviscid time step limitations for explicit time advancement of compressible wall bounded flows (12), for different coordinate direction, streamwise (solid), wall-normal (dashed) and spanwise (dash-dotted). The limitations are normalized with the skin-friction coefficient and reported as a function of the reference Mach number M_0 (bulk Mach number for channel flow and free-stream Mach number for boundary layer). Δt_I^+ is the incompressible time step limitation in the streamwise direction.

The acoustic contribution is suppressed in the incompressible Navier–Stokes equations, and the time step limitation of wall-bounded flows is typically controlled by the streamwise direction,

$$\Delta t_I^+ = \Delta x^+ \sqrt{C_f/2}. \tag{13}$$

The viscous time step limitation in wall-bounded flows is dictated by the wall-normal direction, and in wall units one has

$$\Delta t_{yv}^+ = \Delta y^{+2}. \tag{14}$$

Fig. 2 shows the inviscid time step limitations (12) as a function of the reference Mach number, compared to the incompressible time step limitation (13). The normalized viscous time step limitation $\Delta t_{yv}^+/\sqrt{C_f/2}$, in turbulent flows is much larger than the inviscid one, provided that $\Delta y^+ \approx 1$. Fig. 2 shows that the time step for explicit time advancement of compressible flows is always more limiting as compared to the incompressible case, and this is especially true at low Mach number, with an overhead easily exceeding an order of magnitude [39].

3. Validation

STREAmS has been tailored to carry out DNS of three types of canonical compressible flow configurations, namely supersonic plane channel flow, supersonic boundary layer and shock wave/boundary layer interaction. In the following we validate the solver for these three flows and compare the results to experimental and numerical data available in the literature. We use both Reynolds ($\phi = \overline{\phi} + \phi'$) and Favre ($\phi = \widetilde{\phi} + \phi''$, $\widetilde{\phi} = \overline{\rho\phi}/\overline{\rho}$) decompositions, where the overline symbol denotes averaging in the homogeneous space directions and in time.

3.1. Supersonic plane channel flow

We carry out DNS of plane supersonic channel flow at bulk Mach number $M_b = u_b/c_w = 1.5$ and bulk Reynolds number $Re_b = 2\rho_b u_b h/\mu_w = 15241$, where $\rho_b = 1/V \int_V \rho \mathrm{d}V$ is the bulk density and $u_b = 1/(\rho_b V) \int_V \rho u \mathrm{d}V$ is the bulk velocity in the channel (both exactly constant in time), and μ_w and c_w are the dynamic viscosity coefficient and the speed of sound at the wall temperature, respectively. This configuration corresponds to a friction Reynolds number $Re_\tau = \rho_w u_\tau h/\mu_w = 502$. The computational domain is a rectangular box with size $6\pi h \times 2h \times 2\pi h$ in the x, y, z coordinate directions, respectively and h is the channel half-height. The mesh spacing is constant in the wall-parallel directions, and an error-function mapping is used to cluster mesh points towards the walls. The number of mesh points in the three directions is $N_x = 1032$, $N_y = 256$, $N_z = 512$, corresponding to a mesh spacing in wall units $\Delta x^+ = 9.2$, $\Delta y^+ = 0.8$–5.8 and $\Delta z^+ = 6.2$. Periodicity is enforced in the homogeneous wall-parallel directions, and no-slip isothermal conditions are imposed at the channel walls. The mesh in the wall-normal direction is staggered such that the wall coincides with an intermediate node, where the convective fluxes are identically zero. This approach guarantees correct telescoping of the numerical fluxes and exact conservation of the total mass, with the further benefit of doubling the maximum allowed time step [31]. The simulation is initiated with a parabolic streamwise velocity profile with superposed random and large-scale sinusoidal perturbations, corresponding to streamwise-aligned rollers. We first compare the results of DNS carried out using GPUs and the sixth-order energy conserving scheme with previous DNS data for the same configuration [31] and find excellent agreement for both the mean velocity u^+ and the Reynolds stresses $\tau_{ij} = \overline{\rho}\widetilde{u_i''u_j''}$, as shown in Fig. 3. Fig. 4 shows the instantaneous streamwise velocity in a cross-stream, a streamwise and a wall-parallel plane at $y^+ = y/\delta_v = 15$ for the sixth-order energy conserving scheme. The instantaneous flow field exhibits a typical pattern showing low- and high-speed momentum streaks in wall-parallel planes, associated with sweeps and ejections, better visible in the cross-stream plane. In the present flow shock waves are absent, and shocklets are not expected, hence the obvious choice would be to simply disable the shock-capturing machinery. However, to highlight the effect of the various discretization schemes on the mean flow statistics, we have carried out additional simulations of the

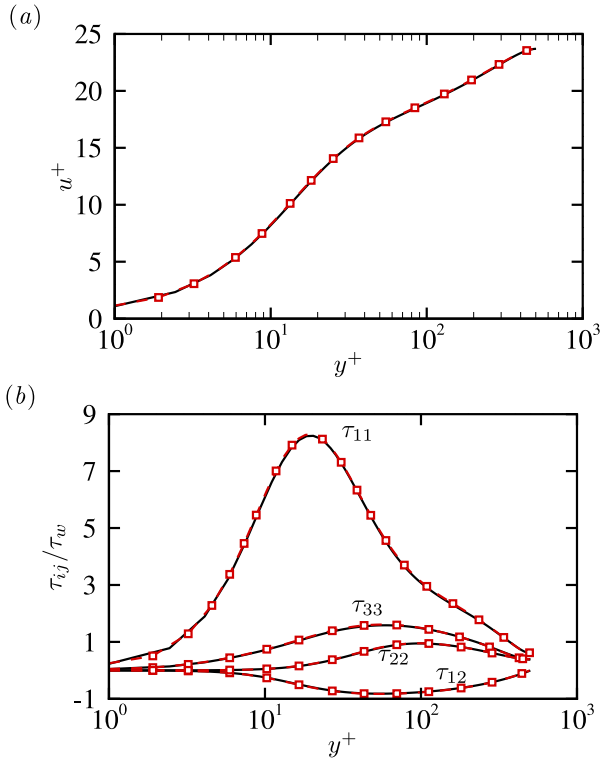

Fig. 3. Supersonic plane channel flow at $M_b = 1.5$ and $Re_\tau = 490$. (a) Mean streamwise velocity profile and (b) Density scaled turbulent stresses τ_{ij}/τ_w as a function of y^+. Present DNS data (black solid) are compared to reference data [31] (red dashed with squares).

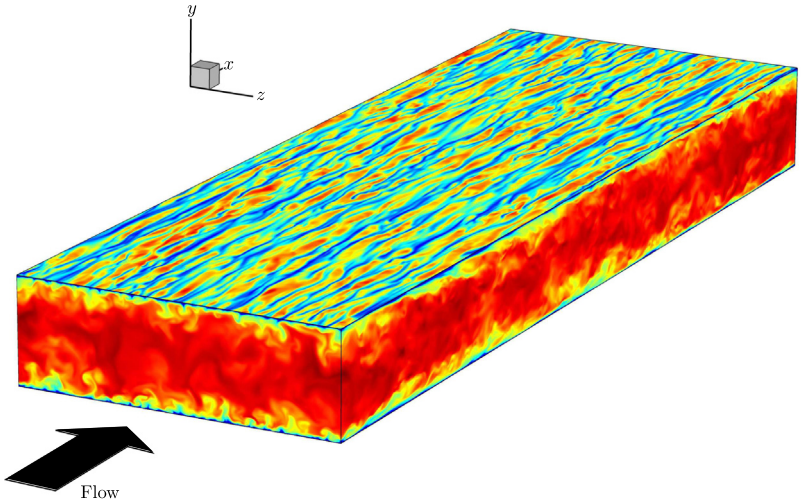

Fig. 4. Instantaneous streamwise velocity for plane supersonic channel flow at $Re_\tau = 500$ and $M_b = 1.5$. The wall-parallel plane is at $y^+ = 15$.

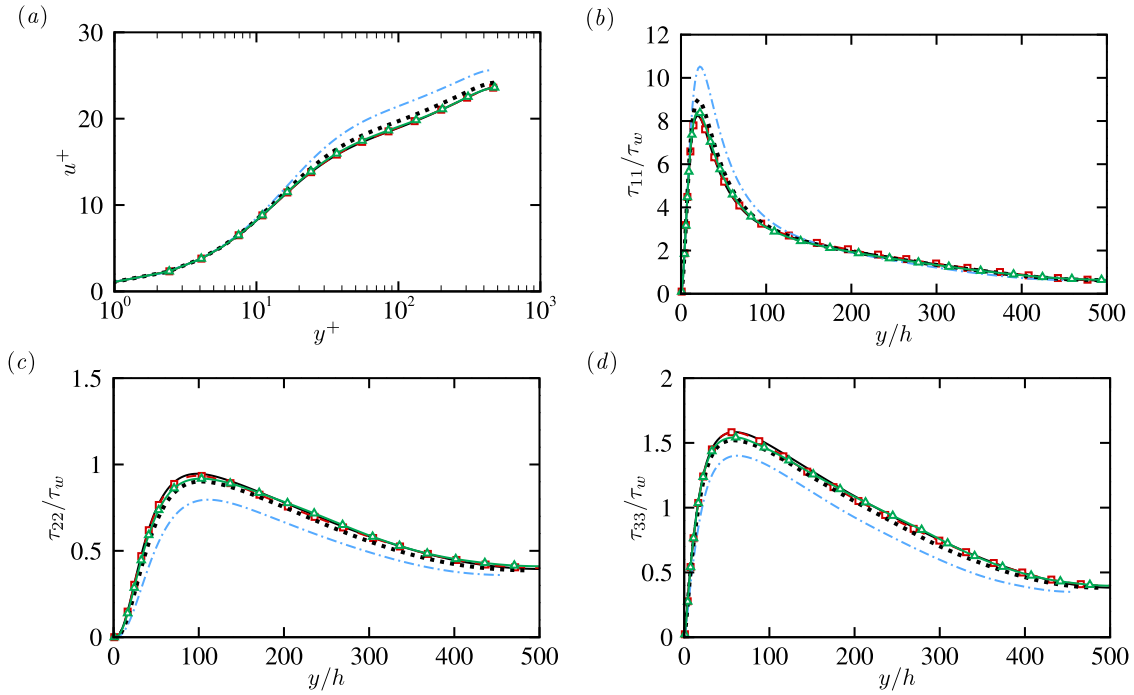

Fig. 5. (*a*) Mean streamwise velocity profile in viscous units $u^+ = u/u_\tau$ and (*b*, *c*, *d*) normal turbulent stresses $\tau_{ii} = \overline{\rho}\widetilde{u_i''u_i''}$ as a function of the viscous scaled wall-normal coordinate $y^+ = y/\delta_v$, for supersonic turbulent channel flow at bulk Mach number $M_b = 1.5$ and bulk Reynolds number $Re_b = 15241$. Different curves refer to different numerical discretization of convective fluxes: sixth-order energy conserving (black solid), fourth-order energy conserving (red squares), hybrid sixth-order energy conserving/fifth order WENO (shock sensor threshold $\theta = 0.05$) (green triangles), full fifth-order WENO (black dotted), full third-order WENO (cyan dash-dotted).

same flow case using different computational set-ups, namely a fourth- and sixth-order energy preserving scheme, a hybrid sixth-order energy preserving/fifth-order WENO scheme with shock sensor threshold $\theta = 0.05$, and finally a third- and fifth-order WENO scheme. Fig. 5(*a*) shows the resulting mean streamwise velocity profiles. The results of fourth-order and sixth-order energy preserving simulations are undistinguishable, and the hybrid scheme also yields the same results, as the shock sensor is seldom activated. On the other hand, the results obtained with the full WENO schemes exhibit deviation from the reference data, owing to built-in numerical dissipation. Differences between WENO and fully energy conserving/hybrid approaches are also clear in the distribution of the normal Reynolds stress tensor components, shown in Figs. 5(*b*, *c*, *d*). The peak of τ_{11} is overestimated both by WENO schemes, whereas the peaks of the other stress components are underestimated, which is a clear symptom of excessive numerical dissipation. The analysis confirms that direct numerical simulation of turbulent flows requires the use of low-dissipative schemes to avoid artificial damping of physical turbulent fluctuations.

3.2. Turbulent boundary layer

We now consider a spatially-developing zero-pressure-gradient supersonic turbulent boundary layer evolving over a flat plate. A direct numerical simulation is carried out at free-stream Mach number $M_\infty = 2$ and Reynolds number in the low-moderate regime, up to a momentum thickness Reynolds

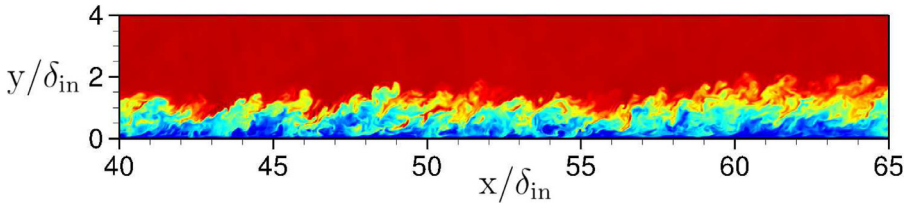

Fig. 6. Instantaneous density field in a streamwise wall-normal plane. Contour levels are shown in the range $0.55 < \rho/\rho_\infty < 1.05$.

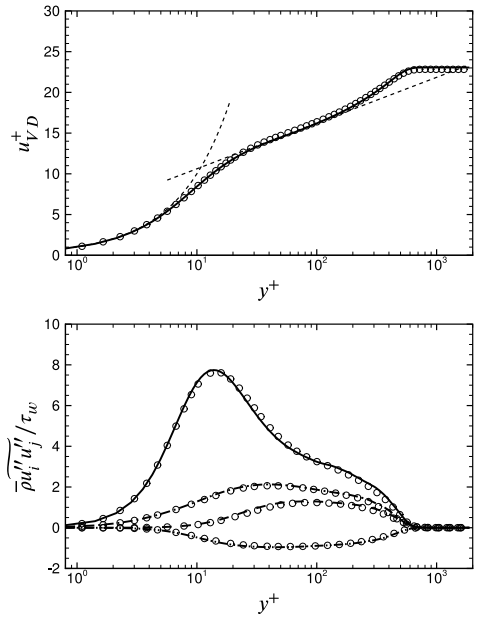

Fig. 7. Comparison of van-Driest transformed mean streamwise velocity (a) and fluctuating velocity statistics (b) scaled in wall units, with reference incompressible DNS [40,41] data at similar friction Reynolds number. Solid line, presents DNS; symbols, reference data. The dashed lines in (a) denote the linear $\overline{u}^+ = y^+$ and log-law $\overline{u}^+ = 5. + 2.44 \ln y^+$.

number $Re_{\delta_2} \approx 1900$, corresponding to a friction Reynolds number $Re_\tau \approx 600$. Similar to the previous case, shocklets are not expected at this Mach number, hence the baseline sixth-order energy-preserving flux is applied throughout. To properly capture the large-scale structures of the boundary layer (known as superstructures), the simulation is carried out in a long and wide computational box, which extends for $L_x = 105\delta_{in}$, $L_y = 12\delta_{in}$, $L_z = 10\delta_{in}$, in the streamwise (x), wall-normal (y) and spanwise (z) directions, δ_{in} being the boundary layer thickness at the inflow station, computed considering the 99% of the free-stream velocity. The computational domain is discretized with a mesh consisting of $N_x = 4096$, $N_y = 256$, $N_z = 512$ grid nodes. Uniform mesh spacing is used in the wall-parallel directions, and a hyperbolic sine stretching is applied in the wall-normal direction to cluster grid nodes close to the wall, where the spacing is $\Delta y_w^+ = 0.8$. A key ingredient for the simulation of a compressible turbulent boundary layer is correct implementation of the boundary conditions, which here are specified as follows. At the upper and outflow boundaries non-reflecting boundary conditions are imposed based on characteristic decomposition in the direction normal to the boundary [42]. A similar characteristic wave treatment is also applied at the no-slip wall boundary, at which the wall temperature is set to its nominal recovery value, $T_r/T_\infty = 1 + (\gamma - 1)/2\, r\, M_\infty^2$, with $r = Pr^{1/3}$. The flow is assumed to be statistically homogeneous in the spanwise direction, and periodic boundary conditions are thus applied. A critical issue in the simulation of spatially-evolving turbulent flows is the prescription of the inflow turbulence generation method. In STREAmS, velocity fluctuations at the inlet plane are imposed by means of a synthetic digital filtering (DF) approach [43], extended to the compressible case through use of the strong Reynolds analogy [44]. An efficient implementation of the method is achieved using an optimized DF procedure [45], whereby the filtering operation is decomposed in a sequence of fast one-dimensional convolutions. The implementation requires the specification of the Reynolds stress tensor at the inflow plane, which is interpolated by a dataset of previous DNS of supersonic boundary layers by the same group [46]. The computation is initialized by prescribing a mean fully developed turbulent compressible boundary layer obtained by applying the inverse van Driest transformation [47] to an incompressible profile of the Musker family [48].

In Fig. 6 we show a snapshot of the instantaneous density field in a streamwise wall-normal plane. The figure highlights the main features of the turbulent boundary layer and its multi-scale structure, characterized by an extremely intermittent behavior in the outer layer, with regions of relatively quiescent, high-speed irrotational fluid interspersed with slower, large-scale rotational bulges. The distributions of the van Driest transformed mean streamwise velocity profile and velocity fluctuation intensities at a reference station ($x_{\text{ref}} = 90\delta_{\text{in}}$) are reported in Fig. 7 in inner scaling. The DNS data are compared with the incompressible boundary layer datasets [40,41] at similar friction Reynolds number ($Re \approx 580$). The figure shows near collapse of compressible and incompressible DNS data, after density variations are accounted for.

To assess the code capabilities at higher Mach numbers we also carry out a DNS of a shock-free turbulent boundary layer in the hypersonic regime and compare STREAmS results with reference hypersonic boundary layer data [49]. The free-stream Mach number is $M_\infty = 5.86$, the inlet friction Reynolds number $Re_\tau = 268$ and the wall to recovery temperature ratio $T_w/T_r = 0.76$. The boundary conditions are the same employed for the supersonic test case, whereas the spatial discretization relies on the hybrid sixth-order central/fifth-order WENO discretization. We use a box of size $L_x = 50\delta_{in}$, $L_y = 15\delta_{in}$, $L_z = 12.5\delta_{in}$, discretized using $N_x = 2048$, $N_y = 384$, $N_z = 800$ points, corresponding to a maximum spacing in wall units in the wall parallel directions $\Delta x^+ = 8.3$, $\Delta z^+ = 5.3$, and a wall-normal spacing at the wall $\Delta y_w = 0.8$. Fig. 8 compares the mean streamwise velocity transformed according to van Driest and the density-scaled Reynolds stresses. Excellent agreement is found, also considering that simulations have been carried out with different inflow methods and different numerical schemes.

3.3. Shock-wave/turbulent boundary layer interaction

We present a third flow case to test the shock-capturing capabilities of STREAmS. We carry out DNS of shock-wave/turbulent boundary layer interaction to replicate the flow conditions of reference experiments [50], characterized by a free-stream Mach

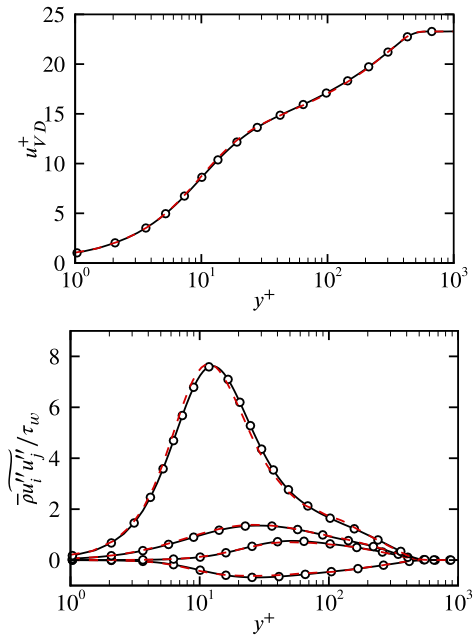

Fig. 8. Comparison of van-Driest transformed mean streamwise velocity for hypersonic turbulent boundary layer at $M_\infty = 5.86$, $Re_\tau = 436$ (a) and fluctuating velocity statistics (b) scaled in wall units, compared to reference DNS data [49] at similar friction Reynolds number. Dashed line, present DNS; solid line with circles, reference data.

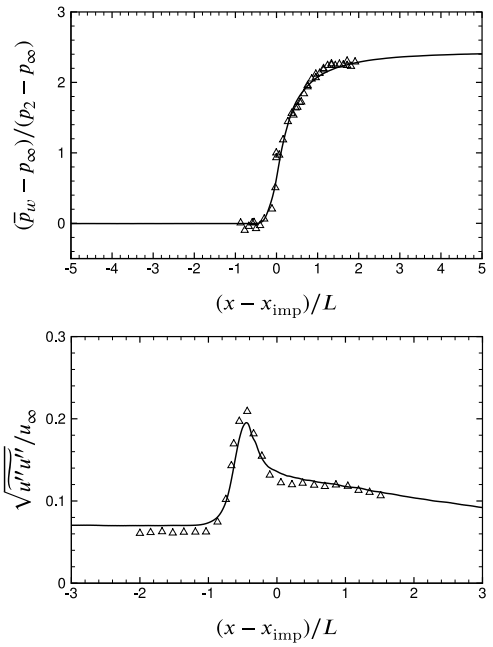

Fig. 10. Distribution of (a) mean wall pressure and (b) streamwise turbulent fluctuation intensity at $y = 0.1L$ as a function of the scaled interaction coordinate $(x - x_{\rm imp})/L$. Solid line, DNS data; open triangles, reference experiment [50].

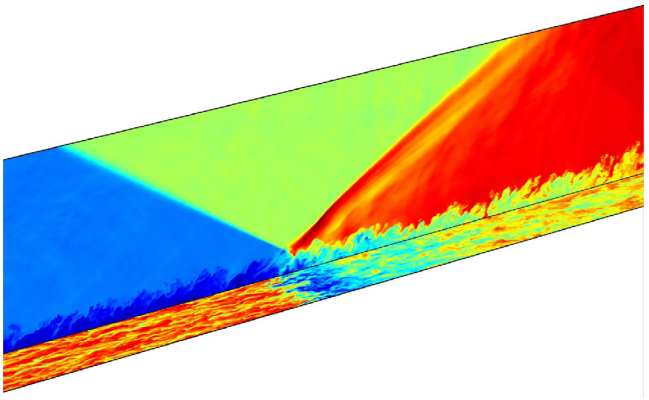

Fig. 9. Visualization of main SBLI features. Contours of the instantaneous density field in a streamwise wall-normal plane (twenty-four levels in the range $0.6 < \rho/\rho_\infty < 1.9$), superposed with contours of streamwise velocity in a wall-parallel plane at $y^+ = 30$ (twenty-four levels in the range $0.4 < u/u_\infty < 0.8$).

number $M = 2.28$ and incidence angle of the shock generator $\phi = 8°$.

The simulation is performed in a computational domain of size $L_x \times L_y \times L_z = [100 \times 12 \times 6]\delta_{\rm in}$ discretized using $N_x \times N_y \times N_z = [4096 \times 384 \times 288]$ grid points. Here $\delta_{\rm in}$ denotes the thickness of the incoming boundary layer upstream of the interaction. The specification of the boundary conditions follows the setup adopted for the previous flow case, except for the upper boundary of the computational domain, where the shock is artificially injected through hard enforcement of the inviscid oblique shock solution corresponding to the selected flow deflection angle.

The flow organization in the investigated SBLI is given by Fig. 9, where contours of the density field are shown in a streamwise-wall-normal plane superposed with contours of the streamwise velocity fluctuations in a wall-parallel plane. The figure shows the complex structure of the interaction, characterized by the presence of an impinging and a reflected shock, which cause thickening of the incoming boundary layer, and the formation of a small recirculation bubble. The typical pattern of high- and low-speed streaks that characterizes the organization of the streamwise velocity disappears across the interaction region, and reforms towards the end of the computational domain, where the boundary layer gradually relaxes to the equilibrium state.

A comparison of DNS data with the reference experiment is reported in Fig. 10, where the distribution of the mean wall pressure and of the streamwise fluctuation intensity is shown across the interaction zone, in terms of the scaled interaction coordinate $(x - x_{imp})/L$, L being the distance between the nominal impingement point of the incoming shock and the apparent origin of the reflected shock. It turns out that the structure of the interaction zone is well captured by the simulation, which predicts a wall pressure rise in excellent agreement with the available experimental data. Similarly, very good agreement is observed for the root-mean-square of the streamwise fluctuation intensity, whose increase in the interaction region is associated with the amplification of turbulence caused by the adverse pressure gradient imparted by the shock system. An extensive validation of the CPU version of STREAmS for the case of shock/boundary layer interaction is available in previous studies of our group [28,51], where comparisons with experiments and DNS carried out by other groups are reported.

4. Implementation and performance

4.1. Programming paradigm and GPU porting

HPC is currently facing a major transition as the majority of systems in operation is still based on CPUs, but GPU-based systems are experiencing rapid growth. For this reason in this phase it is very useful to have a code which can be used on different architectures without requiring further modifications. Tuning the code for different architectures typically involves considerable commitment, including management effort in maintaining, updating or modifying multiple versions of the same code. For this

reason we designed STREAmS to efficiently work on the most common HPC architectures operating today. The code is written in the Fortran language – mostly using Fortran 90 features – which is widely used in HPC, and it is parallelized using the MPI paradigm. Domain decomposition is carried out in two directions (streamwise and spanwise) in order to limit the amount of data transferred for updating the ghost nodes, considering that the communication times may become important when using a large number of tasks.

STREAmS has been developed to support the use of multi-GPUs architectures, while retaining the possibility to compile and use the code on standard CPU-based systems. To achieve this goal, different programming approaches are possible. A first option is using directives, for instance OpenACC [52] or OpenMP [53], which allows to keep the CPU code completely unchanged. A second approach relies on the use of specialized platforms for a specific hardware, which for NVIDIA GPUs are CUDA [54] and CUDA-Fortran [55]. A third strategy is to use more portable but more inconvenient or less popular tools, such as OpenCL [56] or HIP: C ++ Heterogeneous-Compute Interface for Portability [57].

For these reasons in STREAmS we opt for CUDA-Fortran as this allows us to achieve good parallel performance while limiting the changes to the initial CPU code. In particular, the use of the `cuf` automatic kernels allows the large majority of the code to remain unaltered, thus avoiding keeping different versions of the code. The GPU-specific parts of the solver are marked by the `#ifdef USE_CUDA` preprocessing directive. This strategy resembles the approach previously adopted by other popular codes in the field of incompressible turbulence such as AFiD [13] and CaNS [9].

Another important part of the GPU porting is represented by the memory management between CPU and GPU. AFiD employs duplicated arrays residing on host and device, e.g. `w` and `w_gpu`, respectively. The device arrays are distinguished using the CUDA Fortran *device* attribute and are active only when CUDA compilation is enabled, i.e. declared in modules inside preprocessing regions marked by `USE_CUDA` tokens. When using device variables in the computing procedures, the variables are renamed inside `USE_CUDA` regions using Fortran module aliasing so that the computations can always work with the normal (host) names, i.e. `use param, only: w => w_gpu`. If the variables are passed, the declaration of the dummy arguments must also be distinguished by adding `attributes (device)` inside `USE_CUDA` regions. CaNS instead uses a more recent approach based on CUDA *managed* memory. The managed memory potentially allows to avoid completely the declaration of the CPU and GPU versions of the same variable that can instead be used both in CPU and GPU code sections. However, the use of managed memory requires particular care to optimize the underlying transfers and to avoid undesired automatic transfers. To achieve a good managed memory implementation, some information must be provided to the CUDA platform, for example through the *cudaMemAdvise* and *cudaMemPrefetchAsync* functions, which in our opinion reduce the readability of the code. For this reason in STREAmS we followed a different approach, based on the following strategy. For each array, two versions are declared inside the Fortran module: a baseline array `w` and the corresponding computing array `w_gpu`. The latter resides on the device, i.e. has the device attribute, only if the code is compiled by activating CUDA.

```
real, allocatable, dimension(:,:,:,:) :: w, w_gpu
#ifdef USE_CUDA
attributes(device) :: w_gpu
#endif
```

Moreover, `w_gpu` is explicitly allocated only if CUDA compilation is active.

```
#ifdef USE_CUDA
allocate(w_gpu(1:nx, 1:ny, 1:nz, 5))
#endif
```

The baseline array `w` is used during the code initialization and finalization stages while `w_gpu` is used during the time marching section. To this aim, before starting the time evolution it is necessary to ensure that `w_gpu` contains the same data as `w`. If CUDA is active, this is achieved by making a CPU-to-GPU copy managed transparently by CUDA-Fortran. If CUDA is not active, Fortran's `move_alloc` procedure is used, which allows to move the allocation from `w` to `w_gpu`, both on CPU.

```
#ifdef USE_CUDA
w_gpu = w
#else
call move_alloc(w, w_gpu)
#endif
```

A similar procedure is applied for the reverse transfer from GPU to CPU. In conclusion, with this memory management the changes to the original solver are limited to variable declaration and allocation and data transfer between CPU and GPU at the beginning and at the end of the computation, while all the other parts of the code may remain mostly unchanged. Specification of the CUDA kernels is done by using automatic cuf syntax,

```
!$cuf kernel do(3) <<<*,*>>>
 do k=1,nz
  do j=1,ny
   do i=1,nx
    do m=1,nv
     w_gpu(m,i,j,k) = w_gpu(m,i,j,k)+fln_gpu(m,i,j,k)
    enddo
   enddo
  enddo
 enddo
!@cuf iercuda=cudaDeviceSynchronize()
```

The use of automatic kernels requires following some best practices to satisfy the constraints of the *cuf* directives, as the elimination of the interdependencies among loop iterations. The GPU code obtained using automatic kernels has satisfactory performance, both in terms of time to solution and scalability, but superior performance can be achieved using advanced optimization techniques, which are discussed in the following section.

4.2. Single-GPU optimization and performance

For the purpose of optimization, an important preliminary step is the identification of the most computationally demanding code sections, which in a fluid dynamics solver correspond to the evaluation of the convective (Eulerian) and viscous fluxes. In STREAmS, an efficient implementation of the convective fluxes for CPUs was proposed by [34], which however cannot be directly ported to GPU, owing to use of large temporary arrays which would be allocated to the CUDA kernel stack. Hence, STREAmS relies on different implementation of the Eulerian fluxes on CPU and GPU, and in the latter case *cuf* directives are replaced by explicit kernels.

Starting from the baseline implementation, various optimization steps have been performed to accelerate the code on GPUs,

Table 1
Performance results using the energy conserving fluxes for turbulent channel flow with mesh 360 × 240 × 360 on one V100 GPU. For the most computationally demanding sections of the code, i.e. convective fluxes calculations (`Euler-x`, `Euler-y`, `Euler-z`), diffusive fluxes (`Viscous-I`, `Viscous-II`) and a Runge–Kutta update (`RK-I`), the execution times are shown at the different optimization stages A-B-C-D-E.

Steps	A	B	C	D	E
Optimization	Start	Layout	Transpose	Primitive	MPI buffer layout
Metric	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)
Euler-x	76	330	36	24	24
Euler-y	18	22	22	16	14
Euler-z	20	22	22	15	15
Viscous-I	45	54	54	11	11
Viscous-II	10	7.5	7.5	6.6	6.6
RK-I	20	5	5	5	5
Total	**656**	1396	515	311	**297**

with particular focus on the latest NVIDIA V100 card. A preliminary performance analysis carried out with the *nvprof* profiler revealed that STREAmS has low arithmetic intensity and its performance is memory bound, as typical of structured, finite-difference solvers. For this reason, the management of memory accesses is crucial for optimization. The following optimization steps were then carried out:

(A) *Start*: Baseline code.
(B) *Layout*: Change the layout of field and flux variables from, e.g., `w_gpu(nv,nx,ny,nz)` to `w_gpu(nx,ny,nz,nv)`. This allows in principle to improve GPU access, and in particular to get coalesced access where contiguous indices in the innermost array component are accessed by contiguous threads in the CUDA blocks.
(C) *Transpose*: In the case of the calculation of convective terms along the x direction, the CUDA threads correspond to different y and z components, which makes coalesced accesses incompatible with the memory layout. For this reason in the x-convective kernels we introduced support arrays with memory layout `w_trans_gpu(ny,nx,nz,nv)`. These arrays are populated by transposing the original arrays, which is performed very quickly using *cuf* kernels thanks to specialized algorithms used by the compiler (optimal implementations typically rely on CUDA *shared memory*).
(D) *Primitive* - Primitive variables are pre-calculated at each step and used for the calculation of convective and diffusive terms. This change reduces the number of memory accesses, the number of divisions by the density and potentially helps limiting the number of used registers, which can be crucial to avoid register spilling.
(E) *MPI buffer layout* - MPI buffer layout has also been updated to improve coalescence of accesses.

In Table 1 we report the performance in terms of elapsed time for the most demanding code sections, after the various optimization steps described above. In particular, we report the elapsed time for the evaluation of the convective fluxes (`Euler-x`, `Euler-y`, `Euler-z`), diffusive fluxes (`Viscous-I`, `Viscous-II`) and Runge–Kutta updates (`RK-I`).

As expected, we note that in the baseline solver the evaluation of convective fluxes in x direction is considerably more expensive than in y and z. Changing the memory layout (step B) yields global performance degradation, but it is instrumental to fully exploiting memory transposition for computing the x fluxes implemented in step C, after which we observe significant performance improvement. Pre-storing primitive variables (step D) yields significant reduction of the elapsed time, especially in the evaluation of the viscous fluxes, which may be traced back to reduced number of memory accesses and to reduced number of registers, so that register spilling no longer occurs. Finally, adjusting the MPI buffer layout (step E) further allows us to reduce to total time to solution, yielding overall speed-up of a factor of 2.2 with respect to the baseline code version.

In order to quantify the advantages of the code optimization, we extract relevant performance metrics, using the *nvprof* profiler. Considering that STREAmS is a memory bound solver, we focus on the efficiency of memory accesses, and in Table 2 we compare the value of the following metrics for the initial and final (optimized) versions of the code:

- `dram_read_throughput`: Device Memory (DRAM) Read Throughput
- `dram_write_throughput`: Device Memory (DRAM) Write Throughput
- `gld_requested_throughput`: Requested Global Load Throughput
- `gst_requested_throughput`: Requested Global Store Throughput
- `gld_throughput`: Global Load Throughput
- `gst_throughput`: Global Store Throughput
- `gld_efficiency`: Global Memory Load Efficiency
- `gst_efficiency`: Global Memory Store Efficiency

As expected from a structured finite difference code, the DRAM throughputs show that the access to device memory is intensive, whereas the weight of floating point operations, data dependencies or other operations is limited. These throughputs however do not allow us to extract the actual code efficiency. Although these values show improvements for the optimized code (except for `Viscous-II`), these minor variations alone do not explain the significant overall runtime advantage between the baseline and optimized code (speed-up around 2.2). To understand the reasons for such behavior, it is necessary to consider the other metrics that can also give better estimates of the code absolute performance.

The *Requested Global Throughputs* correspond to the requested accesses from the programmer point of view and thus represent significant parameters for evaluating the code efficiency. Indeed remarkable improvements are found for all code sections (see Table 2), consistently with the observed code speed up. In some cases the requested throughputs significantly exceed the real peak value of the GPU memory (about 825 GB/s), which is possible thanks to cache effects that limit the number of accesses to the device's memory. To explain the reason for the improvement of *Requested Global Throughputs* it is necessary to investigate the *Global Throughputs*, which correspond to the required memory accesses. The ratio between the requested throughputs and the required throughputs represents the throughput efficiency, and this quantity shows a dramatic improvement between baseline and optimized code. In particular, most sections of the optimized code exceed 90% efficiency, whereas this metric never exceeds 30% for the baseline version. The optimized code requests memory accesses in a more effective way so that the required accesses

Table 2
Performance results using the energy conserving fluxes for turbulent channel flow with mesh 360 × 240 × 360 on one V100 GPU. For the most demanding code sections, elapsed times and 8 significant metrics concerning memory access are reported, namely: `dram_read_throughput`, `dram_write_throughput`, `gld_requested_throughput`, `gst_requested_throughput`, `gld_throughput`, `gst_throughput`, `gld_efficiency`, `gst_efficiency`. Baseline code version A and optimized version E in Table 1 are compared.

# Sections	Elapsed Times [ms]		DRAM Throughputs [GB/s]				Requested Global Throughputs [GB/s]				Global Throughputs [GB/s]				Global Memory Efficiencies [%]			
			Read		Write		Load		Store		Load		Store		Load		Store	
	Base	*Opt*	*Base*	*Opt*	*Base*	*Opt*	*Base*	*Opt*	*Base*	*Opt*	*Base*	*Opt*	*Base*	*Opt*	*Base*	*Opt*	*Base*	*Opt*
Euler-x	76	24	173	264	46	174	160	523	30	130	660	734	122	150	25	63	25	90
Euler-y	18	14	316	358	153	169	685	824	128	161	2444	920	514	170	28	90	25	94
Euler-z	20	15	301	356	138	161	634	782	120	153	2267	874	477	163	28	90	25	94
Viscous-I	45	11	250	260	62	105	702	2270	26	103	2372	2630	104	106	30	86	25	97
Viscous-II	10	6.6	548	350	118	139	553	781	94	139	2069	973	378	139	27	80	25	100
RK-I	20	5	228	232	323	464	80	327	113	463	319	603	454	464	25	54	25	100

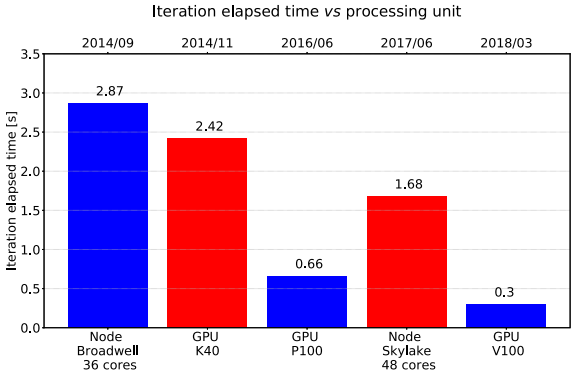

Fig. 11. Elapsed time per time step (s) for STREAmS, using different HPC architectures. For CPU-based runs a single computing node is employed using the MPI parallelization. For GPU-based runs a single GPU is used. Convective fluxes are evaluated using the energy conserving discretization with a mesh 360 × 240 × 360. The upper horizontal axis shows the release dates of each architecture.

are closer to the requested ones. On the contrary, the baseline code reaches very high *Global Throughputs* values due to cache effects, but the number of accesses required to satisfy these requests is too high, mainly due to the memory layout which is not adequate for the card architecture.

It is interesting to note that for `Euler-x` the need to manually transpose the data to have efficient memory access limits the global memory efficiency to approximately 60% in reading, even in the optimized case. On the contrary, `Euler-y` and `Euler-z` do not require data transposition to obtain coalescence and therefore achieve efficiencies close to 100%. It is also interesting to note that `Viscous-II` presents lower values for both global and DRAM throughputs when switching to the optimized version. However, the significant increase in the efficiency allows for a important improvement of the requested access throughputs, and therefore of the time-to-solution.

A concise evaluation of STREAmS performance on different CPU and GPU architectures is provided in Fig. 11. The reference unit for CPU architectures is the single compute node that is exploited using MPI parallelization, and in particular the IntelMPI library. As for GPUs, the reference is the single card. The improvement over the years of performance per node and per GPU is evident and it appears more marked for GPUs for which the advantage over the traditional node is considerable. More specific performance metrics (e.g. related for example to power consumption) would be necessary for a fair comparison between the two architectures, but this is beyond our objectives.

4.3. Parallel optimization and scalability

Large computational domains require the use of multiple GPUs, in which each MPI process typically manages one graphic card. Communication between multiple GPUs can be carried out in two main fashions. The first option relies on manual copy between host and GPU to guarantee that the MPI communications always occur between variables residing on the host. The second option relies on the so-called CUDA-Aware MPI implementations which allow the user to call MPI application programming interface (API) passing device-resident variables. STREAmS has been parallelized to support both data communication patterns, selectable according to compilation options. This allows to correctly run in environments where CUDA-Aware implementations are not available. To improve the scalability performance, the GPU implementation of STREAmS optionally supports asynchronous patterns in which the GPU computations are overlapped with the swapping procedure necessary to exchange information across adjacent blocks. This is done by exploiting the built-in asynchrony of the CUDA kernels and the capabilities of the CUDA *streams*. In this regard, two slightly different well-established strategies were implemented depending on the availability of the CUDA-Aware MPI. As an example, Fig. 12 shows a sketch of the time-lines corresponding to the evaluation of the streamwise convective fluxes. Fluxes at interior nodes can be evaluated before updating ghost nodes, which allows overlapping with MPI communications. Following this idea, the CUDA-Aware MPI implementation (left) is straightforward while the standard MPI implementation (right) requires asynchronous CPU–GPU transfers using *cudaMemcpyAsync* in a specific CUDA stream. After receiving ghost nodes values, fluxes at boundary nodes can be evaluated. A similar strategy is implemented in different sections of the code to increase the compute-communication overlapping as much as possible.

Parallel performance was evaluated on the CINECA Marconi100 Cluster, based on Power 9 Architecture and coupled with NVIDIA Tesla Volta GPUs V100 – 4 GPUs per node – and equipped with NVLink. The compiler is the NVIDIA HPC-SDK compiler, which inherited the PGI compiler history.

Fig. 13(*a*) shows the weak scaling as a function of the number of GPUs for synchronous and asynchronous communications, using 360 × 240 × 360 grid points per GPU. We find improved weak scaling for the asynchronous pattern, especially for a large number of GPUs, with efficiency of about 97% on 1024 GPUs.

Fig. 13(*b*) shows the strong scaling speed-up of the synchronous and asynchronous versions of the code, i.e. keeping constant the total number of grid points 512 × 230 × 360. The efficiency is larger than 90% for 8 GPUs, whereas it drops to 70% for 16 GPUs. On the upper horizontal axis of Fig. 13(*b*) we also show the number of grid points per GPU, which suggests that the optimal number of grid points per GPUs before performance degradation is about 500,000. The advantage of the asynchronous version of the code is visible, but not remarkable.

4.4. Performance comparison

It is valuable to compare the global performance of STREAmS against pre-existing compressible solvers, and in particular codes

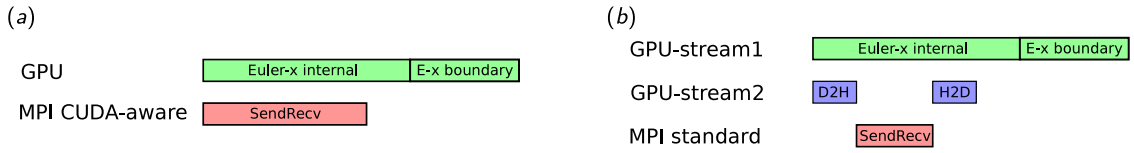

Fig. 12. Sketch of asynchronous time lines for the evaluation of convective fluxes. (*a*) CUDA-aware implementation in which the evaluation of convective fluxes at interior nodes (*Euler-x internal*) is superposed to MPI communications (*SendRecv*), followed by evaluation of the convective fluxes at boundary nodes. (*b*) Standard MPI implementation in which the evaluation of convective fluxes at interior nodes is superposed to device-to-host transfers (*D2H*), MPI communications and host-to-device transfers (*H2D*), followed by evaluation of the convective fluxes at boundary nodes.

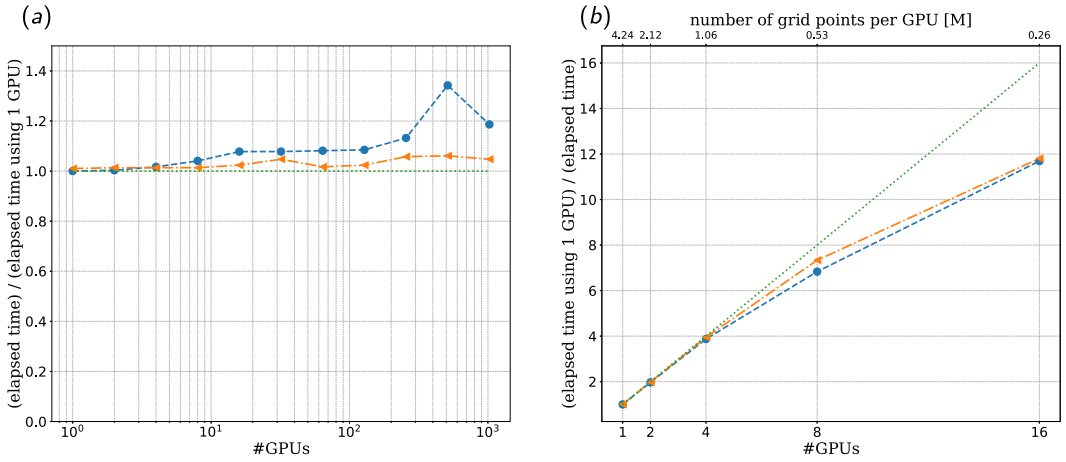

Fig. 13. Weak (*a*) and strong (*b*) scaling performance of STREAmS for the synchronous (circles) and asynchronous (triangles) communications, using the GPU V100 powered cluster Marconi100. The weak scaling is plotted as time ratio (elapsed time using N GPUs divided by the elapsed time using 1 GPU) versus the number of GPUs using the sixth-order energy conserving discretization for the convective fluxes with 360 × 240 × 360 grid for each GPU. The strong scaling is plotted as speed-up (elapsed time using 1 GPU divided by elapsed time using N GPUs) using the sixth-order energy conserving discretization for the convective fluxes with mesh 512 × 230 × 360. The upper horizontal axis in (*b*) shows the millions of grid points processed by each GPU.

optimized for multi-GPU use. Generally, direct comparison between codes implementing different numerical approaches is not straightforward. For this purpose, we follow the strategy used to compare various community codes for turbulent Rayleigh–Bénard convection [59], which consists in performance comparison for code configurations yielding the same quality of results. In the present work, the ZEFR code [23] was selected for comparison. ZEFR is a recently released state-of-the-art solver that uses high-accuracy Flux-Reconstruction [60] methods on unstructured grids, and features a highly optimized multi-GPU implementation. ZEFR is capable of simulating complex geometries also in overset grid mode and can therefore be used for realistic flow simulations, whereas STREAmS is rather oriented to research in basic flow physics. For performance comparison, turbulent channel flow at $Re_\tau = 180$ was selected. The bulk Mach number is set to 0.2, so that compressibility effects may be regarded as negligible, and incompressible data [58] may be taken as benchmark.

We use a computational domain with size $4\pi \times 2 \times 2\pi$, with uniform mesh in the streamwise and spanwise directions, and stretching in the wall-normal direction to have $\Delta y^+ \approx 1$ at the wall. The M1 reference (coarse) mesh consists of 60 × 40 × 60 points for STREAmS, which is operated with fourth-order accuracy. The reference mesh for ZEFR has 15 × 10 × 15 cells, hence using fourth-order polynomials for flux reconstruction allows us to match the number of degree of freedoms (DOF) between the two codes. We have then carried out a mesh refinement study in which meshes M2, M4, M6, M8 have 2, 4, 6, 8 as many points as the baseline mesh, in each coordinate direction. The viscous-scaled wall-normal spacing at the wall is kept the same for all cases. Fig. 14 compares profiles of mean velocity and Reynolds stresses, along with the convergence error in terms of the L^∞ norm of the distance of the computed solution on mesh Mi from the reference solution,

$$E_\varphi(i) = \frac{\max_y(|\varphi(i) - \varphi_{ref}|)}{max_y(|\varphi_{ref}|)}. \tag{15}$$

The trends in the first two columns show that both codes converge to the reference solution, however at different rates. The third column allows quantitative assessment of the convergence and shows that, for the same number of grid nodes/DOF, STREAmS is more accurate than ZEFR. This trend applies to all mesh levels, including simulations that can be considered properly resolved. In particular, the M4 mesh has resolution in the wall-parallel directions $\Delta x^+ = 9.4$, $\Delta z^+ = 4.7$, which are generally regarded to be adequate for DNS of turbulent channel flow. For STREAmS this mesh resolution yields convergence errors of 3% or less with respect to reference data, whereas achieving the same error levels with ZEFR requires three times the number of DOF (i.e. the M6 mesh).

The outcome of the performance comparison is detailed in Table 3, where data concerning computer time and memory usage are reported. To avoid issues related to different level of parallelism between the two codes, performance evaluation is based on single GPU simulations, with memory allocation greater than 50%. The elapsed time per point/DOF per iteration (ETP) shows that STREAmS is about three times faster than ZEFR, and the memory allocation per grid point (MPP) is about six times lower.

It is worth noting that the energy-conserving scheme used in STREAmS requires repeated memory access to the fluid dynamic fields, due to the double summation in the evaluation of the numerical fluxes (7). Therefore, compared to the flux reconstruction scheme used in ZEFR, we do not expect the memory saving per degree of freedom to scale linearly with computational time

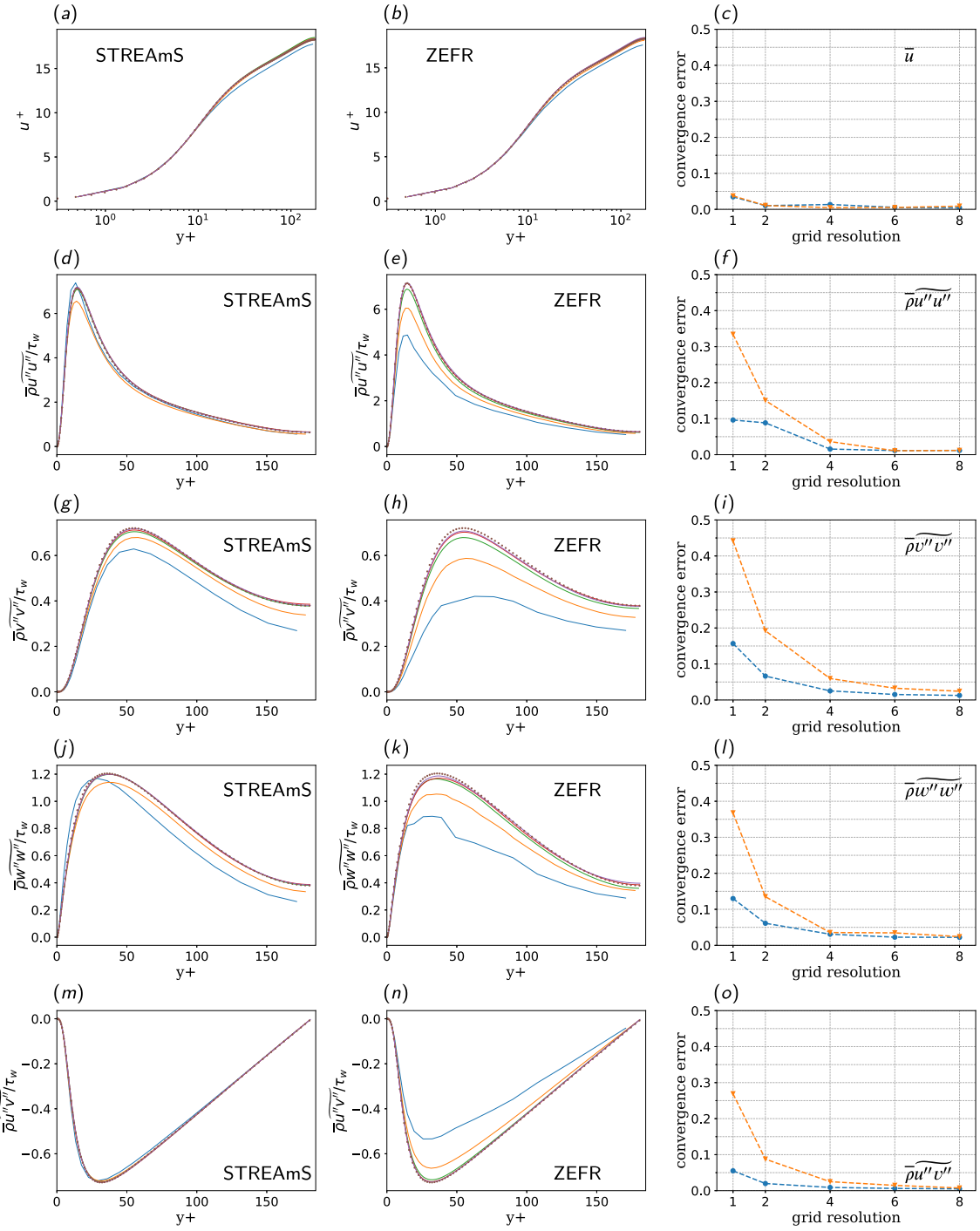

Fig. 14. Mesh sensitivity analysis for STREAmS and ZEFR using turbulent channel flow at $Re_\tau = 180$, (*a*, *b*) u^+, (*d*, *e*) $\widetilde{\overline{\rho}u''u''}/\tau_w$, (*g*, *h*) $\widetilde{\overline{\rho}v''v''}/\tau_w$, (*j*, *k*) $\widetilde{\overline{\rho}w''w''}/\tau_w$, (*m*, *n*) $\widetilde{\overline{\rho}u''v''}/\tau_w$. Reference incompressible data [58] are also reported both for STREAmS (first column) and ZEFR (zecond column). (*c*, *f*, *i*, *l*, *o*) Convergence error as defined in Eq. (15) as a function of mesh resolution level.

Table 3
Performance comparison between STREAmS and ZEFR codes based on mesh sensitivity analyses for turbulent channel flow at $Re_\tau = 180$, $M_b = 0.2$. The codes are compared at mesh resolution which allows to achieve the same error level. ETP and MPP values are evaluated running single-GPU simulations in optimal performance conditions for both codes, i.e. when most GPU memory is used.

Quantity	Symbol	STREAmS	ZEFR
Elapsed time per point/DOF (ns)	ETP	9.6	28.3
Memory per point (B)	MPP	350	2060
Equivalent number of points/DOFs (M)	ENP	9.2	31.1
Equivalent elapsed time	ETP × ENP	88	880
Equivalent memory	MPP × ENP	3.2	64.1

saving. Nevertheless, the computational time saving per degree of freedom is still important, and the optimization strategies we have adopted are fundamental to achieve this performance.

In Table 3 we also define an equivalent number of points (ENP) based on the comparison of accuracy results obtained above, i.e. ZEFR requires three times the number of DOF to achieve the same results accuracy. In the end, we find that for direct numerical simulation of canonical wall-bounded flows the time to solution of STREAmS is $ETP \times ENP \sim 10$ lower than in ZEFR, and the equivalent memory usage (MPP$\times$ ENP) is about twenty-one times less.

In a nutshell, the results show that using a solver specifically designed for DNS of canonical turbulent flows such as STREAmS yields much higher performance than a general-purpose code, such as ZEFR. This conclusion is similar to what found for incompressible solvers for turbulent Rayleigh–Bénard convection [59]. This is also confirmed by the fact that the largest DNS of turbulent flows carried out in recent years [26,61–63] have been performed using dedicated flow solvers, rather than multi-purpose codes.

5. Conclusions

We have presented a recent version of our in-house compressible flow solver STREAmS, which has been ported to CUDA-Fortran. The solver is tailored to canonical compressible turbulent wall-bounded flows, including channels, supersonic and hypersonic boundary layers, and shock wave/turbulent boundary layer interactions. STREAmS stems from two decades experience of our research group in DNS of compressible wall-bounded flows, and a baseline version of the solver is released open-source under GPLv3 license, with the aim of providing the fluid dynamics community with a highly-parallel and efficient compressible flow solver. The use of CUDA-Fortran with the use of `cuf` automatic kernels allows the user to limit modifications to the original code, and to compile and run the code on different HPC architectures. However, implementing ad-hoc GPU optimization allows us to speed up the solver by a factor of about two. Tests carried out on the GPU cluster Marconi100 at CINECA show very good scalability performance, proving that the solver can be used for large-scale direct numerical simulations. A mesh sensitivity analysis for a turbulent channel flow has been carried out to compare the performance of STREAmS with that of ZEFR, which is a state-of-the-art, general-purpose compressible solver. The study has revealed significant advantage of STREAmS in terms of reduced computational effort to reach similar results, although limited to the simple geometrical configurations handled by STREAmS. The availability of the GPU version of the solver allows the flow community to take advantage of contemporary pre-Exascale systems and of the next generation of Exascale supercomputers currently under development, thus providing a state-of-the art platform to significantly extend the range of simulated Reynolds number to the genuine high-Reynolds number regime ($Re_\tau > 10^4$), and provide definite answers to key issues in turbulence research.

CRediT authorship contribution statement

Matteo Bernardini: Conceptualization, Methodology, Investigation, Software, Validation, Project administration, Supervision, Writing - original draft, Visualization, Data curation. **Davide Modesti:** Conceptualization, Methodology, Investigation, Software, Validation, Writing - original draft, Visualization, Resources. **Francesco Salvadore:** Conceptualization, Methodology, Investigation, Software, Writing - original draft, Visualization, Data curation, Resources. **Sergio Pirozzoli:** Conceptualization, Methodology, Investigation, Software, Validation, Writing - review & editing.

Declaration of competing interest

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

Acknowledgments

M. Bernardini was supported by the Scientific Independence of Young Researchers program 2014 (Active Control of Shock-Wave/Boundary-Layer Interactions project, grant RBSI14TKWU), which is funded by the Ministero dell'Istruzione, dell'Universià e della Ricerca. The authors are especially grateful for the computational resources provided by the CINECA Italian Computing Center. The authors also wish to thank Massimiliano Fatica e Josh Romero (NVIDIA) for the useful suggestions and discussions.

References

[1] F. Harlow, J. Welch, Phys. Fluids 8 (12) (1965) 2182–2189.
[2] G. Patterson, S. Orszag, Phys. Fluids 14 (11) (1971) 2538–2541.
[3] J. Kim, P. Moin, J. Comput. Phys. 59 (2) (1985) 308–323.
[4] J. Kim, P. Moin, R. Moser, J. Fluid Mech. 177 (1987) 133–166.
[5] H. Weller, G. Tabor, H. Jasak, C. Fureby, Comput. Phys. 12 (6) (1998) 620–631.
[6] P. Fisher, J. Kruse, J. Mullen, H. Tufo, J. Lottes, S. Kerkemeier, NEK5000: open source spectral element CFD solver, 2008, URL http://nek5000.mcs.anl.gov/index.php/MainPage.
[7] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D.D. Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Hu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. Kirby, S. Sherwin, Comput. Phys. Comm. 192 (2015) 205–219.
[8] E. van der Poel, R. Ostilla-Mónico, J. Donners, R. Verzicco, Comput. Fluids 116 (2015) 10–16.
[9] P. Costa, Comput. Math. Appl. 76 (8) (2018) 1853–1862.
[10] J. Dongarra, P. Luszczek, in: D. Padua (Ed.), Encyclopedia of Parallel Computing, Springer US, Boston, MA, 2011, pp. 2055–2057, http://dx.doi.org/10.1007/978-0-387-09766-4_157, https://doi.org/10.1007/978-0-387-09766-4_157.
[11] 136 GPU-Accelerated Supercomputers Feature in TOP500 | NVIDIA Blog, 2019, https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/. (Accessed 16 January 2020).
[12] The GREEN 500, 2020, https://www.top500.org/green500/. (Accessed 16 January 2020).
[13] X. Zhu, E. Phillips, V. Spandan, J. Donners, G. Ruetsch, J. Romero, R. Ostilla-Mónico, Y. Yang, D. Lohse, R. Verzicco, M. Fatica, R. Stevens, Comput. Phys. Commun. 229 (2018) 199–210.
[14] P. Costa, E. Phillips, L. Brandt, M. Fatica, Comput. Math. Appl. (2020).
[15] A. Honein, P. Moin, J. Comput. Phys. 201 (2) (2004) 531–545.
[16] G. Coppola, F. Capuano, S. Pirozzoli, L. de Luca, J. Comput. Phys. 382 (2019) 86–104.
[17] D. Modesti, S. Pirozzoli, Comput. Fluids 152 (2017) 14–23.
[18] T. Economon, F. Palacios, S. Copeland, T. Lukaczyk, J. Alonso, AIAA J. 54 (3) (2015) 828–846.
[19] C. Jacobs, S. Jammy, N. Sandham, J. Comput. Sci. 18 (2017) 12–23.
[20] Legion Webpage, 2020, https://legion.stanford.edu/ (Accessed 31 March 2020).
[21] M.D. Renzo, L. Fu, J. Urzay, Comput. Phys. Comm. (2020) 107262.
[22] F. Witherden, A. Farrington, P. Vincent, Comput. Phys. Comm. 185 (11) (2014) 3028–3040.
[23] J. Romero, J. Crabill, J. Watkins, F. Witherden, A. Jameson, Comput. Phys. Comm. 250 (2020) 107169.
[24] S. Pirozzoli, Annu. Rev. Fluid Mech. 43 (2011) 163–194.
[25] S. Pirozzoli, M. Bernardini, J. Fluid Mech. 688 (2011) 120–168.
[26] S. Pirozzoli, M. Bernardini, Phys. Fluids 25 (2) (2013) 021704.
[27] S. Pirozzoli, M. Bernardini, F. Grasso, J. Fluid Mech. 657 (2010) 361–393.
[28] S. Pirozzoli, M. Bernardini, AIAA J. 49 (2011) 1307–1312.
[29] M. Bernardini, S. Pirozzoli, P. Orlandi, Int. J. Heat Fluid Flow 35 (2012) 45–51.
[30] M. Bernardini, S. Pirozzoli, P. Orlandi, S. Lele, AIAA J. 52 (10) (2014) 2261–2269, http://dx.doi.org/10.2514/1.j052842.
[31] D. Modesti, S. Pirozzoli, Int. J. Heat Fluid Flow 59 (2016) 33–49.
[32] D. Modesti, S. Pirozzoli, F. Grasso, Int. J. Heat Fluid Flow 76 (2019) 130–140.
[33] F. Salvadore, M. Bernardini, M. Botti, J. Comput. Phys. 235 (2013) 129–142.
[34] S. Pirozzoli, J. Comput. Phys. 229 (19) (2010) 7180–7190.
[35] C. Kennedy, A. Gruber, J. Comput. Phys. 227 (3) (2008) 1676–1700.
[36] G. Jiang, C.W. Shu, J. Comput. Phys. 126 (1996) 202.

[37] F. Ducros, V. Ferrand, F. Nicoud, C. Weber, D. Darracq, D. Gacherieu, T. Poinsot, J. Comput. Phys. 152 (2) (1999) 517–549.
[38] P. Spalart, R. Moser, M. Rogers, J. Comput. Phys. 96 (2) (1991) 297–324.
[39] D. Modesti, S. Pirozzoli, J. Sci. Comput. 75 (2018) 308–331.
[40] M. Simens, J. Jimenez, S. Hoyas, Y. Mizuno, J. Comput. Phys. 228 (2009) 4218–4231.
[41] J. Jimenez, S. Hoyas, M. Simens, Y. Mizuno, J. Fluid Mech. 657 (2010) 335–360.
[42] T. Poinsot, S. Lele, J. Comput. Phys. 101 (1) (1992) 104–129.
[43] M. Klein, A. Sadiki, J. Janicka, J. Comput. Phys. 186 (2) (2003) 652–665.
[44] E. Touber, N.D. Sandham, Theor. Comput. Fluid Dyn. 23 (2009) 79–107.
[45] A. Kempf, S. Wysocki, M. Pettit, Comput. & Fluids 60 (2012) 58–60.
[46] S. Pirozzoli, M. Bernardini, Supersonic turbulent boundary layers - DNS database, 2011, http://newton.dma.uniroma1.it/dnsm2.
[47] A. Smits, J. Dussauge, Turbulent Shear Layers in Supersonic Flow, American Institute of Physics, New York, 2006.
[48] A. Musker, AIAA J. 17 (1979) 655–657.
[49] C. Zhang, L. Duan, M. Choudhari, AIAA J. 56 (11) (2018) 4297–4311.
[50] P. Dupont, C. Haddad, J. Debiéve, J. Fluid Mech. 559 (2006) 255–277.
[51] P. Volpiani, M. Bernardini, J. Larsson, Phys. Rev. Fluids 5 (1) (2020) 014602.
[52] OpenACC, 2020, https://www.openacc.org/, (Accessed 16 January 2020).
[53] OpenMP, 2020, https://www.openmp.org/. (Accessed 16 January 2020).
[54] CUDA, 2020, https://developer.nvidia.com/cuda-zone, (Accessed 16 January 2020).
[55] CUDA FORTRAN, 2020, https://developer.nvidia.com/cuda-fortran. (Accessed 16 January 2020).
[56] OpenCL, 2020, https://www.khronos.org/opencl/. (Accessed 16 January 2020).
[57] HIP : C++ Heterogeneous-Compute Interface for Portability, 2020, https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/. (Accessed 16 January 2020).
[58] R.D. Moser, J. Kim, N.N. Mansour, Phys. Fluids 11 (4) (1999) 943–945.
[59] G. Kooij, M.A. Botchev, E.M. Frederix, B.J. Geurts, S. Horn, D. Lohse, E.P. van der Poel, O. S., R.J. Stevens, R. Verzicco, Comput. Fluids 166 (2018) 1–8.
[60] H.T. Huynh, American Institute of Aeronautics and Astronautics, 2007, pp. 2007–4079,
[61] M. Bernardini, S. Pirozzoli, P. Orlandi, J. Fluid Mech. 742 (2014) 171–191.
[62] M. Lee, R. Moser, J. Fluid Mech. 774 (2015) 395–415.
[63] D. Buaria, K. Sreenivasan, Dissipation range of the energy spectrum in high Reynolds number turbulence, 2020, arXiv preprint arXiv:2004.06274.