ARTICLE IN PRESS

Computer Physics Communications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Computer Physics Communications

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

AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters☆

Xiaojue Zhu [a,*], Everett Phillips [b], Vamsi Spandan [a], John Donners [c], Gregory Ruetsch [b], Joshua Romero [b], Rodolfo Ostilla-Mónico [d,g], Yantao Yang [a], Detlef Lohse [a,f], Roberto Verzicco [e,a], Massimiliano Fatica [b], Richard J.A.M. Stevens [a,*]

[a] *Physics of Fluids Group, Max Planck Center Twente for Complex Fluid Dynamics, MESA+ Research Institute, and J. M. Burgers Center for Fluid Dynamics, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands*
[b] *NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, USA*
[c] *SURFsara, Science Park 140, 1098 XG Amsterdam, The Netherlands*
[d] *School of Engineering and Applied Sciences and Kavli Institute for Bionano Science and Technology, Harvard University, Cambridge, MA 02138, USA*
[e] *Dipartimento di Ingegneria Industriale, University of Rome "Tor Vergata", Via del Politecnico 1, Roma 00133, Italy*
[f] *Max-Planck Institute for Dynamics and Self-Organization, Am Fassberg 17, 37077 Göttingen, Germany*
[g] *Department of Mechanical Engineering, University of Houston, Houston, TX 77204, USA*

ARTICLE INFO

Article history:
Received 4 May 2017
Received in revised form 21 January 2018
Accepted 27 March 2018
Available online xxxx

Keywords:
GPU
Parallelization
Turbulent flow
Finite-difference scheme
Rayleigh–Bénard convection
Plane Couette flow

ABSTRACT

The AFiD code, an open source solver for the incompressible Navier–Stokes equations (http://www.afid.eu), has been ported to GPU clusters to tackle large-scale wall-bounded turbulent flow simulations. The GPU porting has been carried out in CUDA Fortran with the extensive use of kernel loop directives (CUF kernels) in order to have a source code as close as possible to the original CPU version; just a few routines have been manually rewritten. A new transpose scheme has been devised to improve the scaling of the Poisson solver, which is the main bottleneck of incompressible solvers. For large meshes the GPU version of the code shows good strong scaling characteristics, and the wall-clock time per step for the GPU version is an order of magnitude smaller than for the CPU version of the code. Due to the increased performance and efficient use of memory, the GPU version of AFiD can perform simulations in parameter ranges that are unprecedented in thermally-driven wall-bounded turbulence. To verify the accuracy of the code, turbulent Rayleigh–Bénard convection and plane Couette flow are simulated and the results are in excellent agreement with the experimental and computational data that have been published in literature.

Program summary
Program Title: AFiD-GPU
Program Files doi: http://dx.doi.org/10.17632/rwjdg7ry66.1
Licensing provisions: MIT
Programming language: Fortran 90, CUDA Fortran, MPI
External routines: PGI, CUDA Toolkit, FFTW3, HDF5
Nature of problem: Solving the three-dimensional Navier–Stokes equations coupled with a scalar field in a cubic box bounded between two walls and with periodic boundary conditions in the horizontal directions.
Solution method: Second order finite difference method for spatial discretization, third order Runge–Kutta scheme in combination with Crank–Nicolson for the implicit terms for time advancement, two dimensional pencil distributed MPI parallelization, GPU accelerated routines.
Additional comments including restrictions and unusual features: The code is available and supported on https://github.com/PhysicsofFluids/AFiD_GPU_opensource.

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

* Corresponding authors.

E-mail addresses: xiaojue.zhu@utwente.nl (X. Zhu), r.j.a.m.stevens@utwente.nl (R.J.A.M. Stevens).

https://doi.org/10.1016/j.cpc.2018.03.026
0010-4655/

1. Introduction

Turbulence is a high dimensional multi-scale process. As the velocity of the fluid increases, the range of scales of the resulting motion increases as energy is transferred to smaller and smaller scales, and the flow transitions from laminar to turbulent. To understand the physics of this energy transfer, direct numerical simulations (DNS), which resolve all flow scales, are used. In order to resolve all scales, large meshes and immense computational power are required.

Here, two paradigmatic systems are taken as examples, i.e. Rayleigh–Bénard convection [1–3], the buoyancy driven flow of a fluid heated from below and cooled from above, and plane Couette flow, the shear-induced motion of a fluid contained between two infinite flat walls, which are among the most popular systems for convection and wall-bounded shear flow. The two are classical problems in fluid dynamics. Next to pipe [4], channel [5,6], and Taylor–Couette flows [7,8], the systems have been and are still used to test various new concepts in the field [1] such as nonlinear dynamics and chaos, pattern formation, or turbulence, on which we focus here.

Turbulent Rayleigh–Bénard flow is of interest in a wide range of sciences, including geology, oceanography, climatology, and astrophysics as it is a relevant model for countless phenomena such as thermal convection in the atmosphere [9], in the oceans (including thermohaline convection) [10], in the Earth's outer core [11], where the reversals of the large scale convection are of prime importance to the magnetic field, in the interior of gaseous giant planets and in the outer layer of the sun [12]. Natural convection in technological applications such as buildings, in process technology, or in metal-production processes is also modeled using Rayleigh–Bénard flow. For those real-world applications of Rayleigh–Bénard flow, the system is highly turbulent in both bulk and boundary layers. This state is the so-called ultimate regime of thermal convection, which has been recently realized experimentally in the laboratory [13]. However, because of the extremely high Rayleigh numbers (the non-dimensional temperature difference) and high Reynolds numbers (the non-dimensional velocity) of the flow, computationally the ultimate thermal convection regime could not be reached so far, despite its great importance.

Turbulent plane Couette flow is of interest for more fundamental reasons. It is the only flow which bears exactly the same total stress across the thickness, which is one of hypotheses required by Prandtl's classical arguments for the existence of logarithmic layers for the mean velocity profile [14]. Besides, because its simple geometry, plane Coutte flow is often used as an example to illustrate the wall-bounded turbulence structure [15], and more recently, investigation of the self-sustainment of near wall turbulence [16] or inner–outer wall turbulence interaction [17].

To accurately simulate high Rayleigh and Reynolds number flows of interest in geo- and astrophysical flows [11,12,18], efficient code parallelization and effective use of large scale supercomputers are essential to reach the amount of grid points necessary to resolve all flow scales. Previous work in parallelizing a second-order finite-difference solver for natural convection and shear flow have allowed us to consider unprecedented large computational boxes using AFiD [19,20]. However, there are still limitations to the parallelization as it was written for a central processing unit (CPU)-based system, while the current trends in high performance computing point towards the increase in use of accelerators. These are expected to push the performance of supercomputers into the exascale range by the use of graphic processing units (GPUs) [21]. GPUs are especially well-suited to address problems that can be expressed as data-parallel computations, where the same program is executed on different data elements in parallel. GPUs are also characterized by high memory bandwidth, something especially important for low-order finite difference computational fluid dynamics codes where the data reuse is minimal. Given the above, and because GPUs are the most commonly used accelerator technology, we decided to port AFiD to GPU clusters, while further developing the underlying algorithms. With the porting of AFiD to GPU, and the introduced efficiency improvements, this open source code can now tackle unprecedentedly large fluid dynamics simulations. Therefore we expect the code to be of benefit to the convection and scientific community at large.

We note that there are also several other open source codes, such as NEK5000 [22], OpenFOAM [23], Nektar++ [24], and channelflow.org [25], available. We refer the reader to the respective references that describe the codes, and the website http://exaflow-project.eu/, which gives an overview of several of these codes. In addition, there are commercial packages such as COMSOL, Aerosoft, ANSYS/Fluent, and Barracuda VR, that can be used to solve fluid dynamics problems. The downside of commercial packages is that the source code is not available, which makes further development of these codes in research projects impossible. Some of the open source packages mentioned above also scale well up to large number of cores. However, our specialized code offers significant performance benefits compared to generalized flow solvers for canonical model problems such as Rayleigh–Bénard [1–3], Taylor–Couette [7,8], Double Diffusive Convection [26], and plane Couette flow [14], for which our code is designed. In addition, it is easier to port a specialized code to a GPU cluster with relatively minimal effort, while getting great performance.

This paper is organized as follows. In Section 2 we discuss the details of the solver AFiD. Subsequently, in Section 3 details of the GPU implementation are discussed, before we discuss the code performance in Section 4. In Section 5 we end with a presentation of Rayleigh–Bénard and plane Couette cases that have been simulated with the new GPU code. In Section 6 we present the main conclusions and present future development plans for the code.

2. AFiD code

Here we summarize the numerical method (Section 2.1) and the parallelization scheme (Section 2.2) as described in Ref. [19] before we will discuss the specifics of the GPU implementation in Section 3.

2.1. Numerical scheme

AFiD (http://www.afid.eu) solves the Navier–Stokes equations with an additional equation for temperature in three-dimensional coordinates on a Cartesian mesh with two periodic (unbounded) directions (y and z) which are uniformly discretized and one bounded direction (x) for which non-uniform grids, with clustering of points near the walls, can be used. Note that for Rayleigh–Bénard flow, the temperature is turned on and for plane Couette flow, the advection–diffusion equation for temperature is turned off and the body force term in the Navier–Stokes equations is canceled. All the other features, except the boundary conditions, are the same.

The Navier–Stokes equations with the incompressibility condition read:

$$\nabla \cdot \mathbf{u} = 0, \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\rho_0^{-1} \nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{F}_b \tag{2}$$

for the temperature field, an advection–diffusion equation is used

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \kappa \nabla^2 T, \tag{3}$$

where $\mathbf{u}$ is the velocity vector, p the pressure, ρ_0 the density, T the temperature, ν the kinematic viscosity, κ the thermal diffusivity,

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

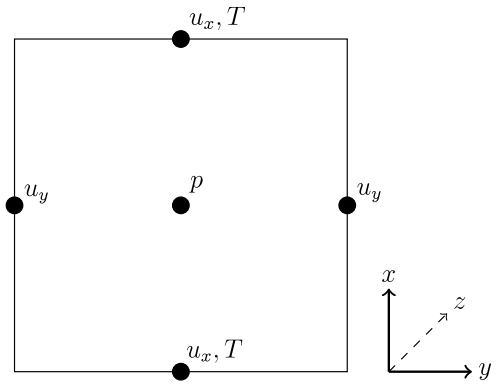

Fig. 1. Location of pressure, temperature and velocities of a 2D simulation cell. The third dimension (z) is omitted for clarity. As on an ordinary staggered scheme, the velocity vectors are placed on the borders of the cell and pressure is placed in the cell center. The temperature is placed at the same location as the vertical velocity, to ensure exact energy conservation. Note that gravity is in negative x-direction. We refer to Appendix A for more information on the energy conservation properties of the used scheme.

and t time. For Rayleigh–Bénard convection we use the Boussinesq approximation: the body force F_b is taken to only depend linearly on the temperature, and to be in the direction of gravity ($\mathbf{e}_x$ is the unit vector anti-parallel to gravity), and we also ignore the possible dependencies of density, viscosity and thermal diffusivity on temperature so these parameters are constant. Other body forces, like the Coriolis force, can be included in this term if one is dealing with rotating frames.

For the spatial discretization of the domain, we use a conservative, central, second-order, finite-difference discretization on a staggered grid. A two-dimensional (for clarity) schematic of the variable arrangement is shown in Fig. 1. The pressure is calculated at the center of the cell. For thermal convection between two plates, the temperature field is collocated with the u_x grid, the velocity component in the direction of gravity. This avoids the interpolation error when calculating the term $\mathbf{F}_b \sim T\mathbf{e}_x$ in Eq. (2). This scheme has the advantage of being energy conserving in the limit $\Delta t \to 0$ [27]. In addition to the conservation properties, the low-order finite difference scheme has the advantage of handling the shock-like behavior resulting from the absence of the pressure term in the advection–diffusion equation for temperature from the Boussinesq approximation [28,29] better.

Given a set of initial conditions, the simulations are advanced in time by a fractional-step procedure combined with a low-storage, third-order Runge–Kutta (RK3) scheme and a Crank–Nicolson method [30] for the implicit terms. The time step Δt is constrained by a Courant–Friedrichs–Lewy (CFL) number, whose definition is given by:

$$\Delta t \leq \mathrm{CFL} \cdot \min\left[\frac{1}{|u_x|/\Delta_x + |u_y|/\Delta_y + |u_z|/\Delta_z}\right]. \tag{4}$$

For RK3 methods, CFL number is stable up to $\sqrt{3}$ even if, in practice, simulations are run at a maximum CFL number of approximately 1.3. The RK3 scheme requires three substeps per time step, but due to the larger time step and the $\mathcal{O}([\Delta t]^3)$ error it is more efficient than a standard second-order Adams–Bashforth integration. The pressure gradient is introduced through the "delta" form of the pressure [31–33]: a provisional, non-solenoidal velocity field is calculated using the old value of the pressure in the discretized Navier–Stokes equation. The updated pressure, required to enforce the continuity equation at every cell, is then computed by solving a Poisson equation for the pressure correction. The velocity and pressure fields are then updated using this correction, which results in a divergence-free velocity field. Full details of the procedure can be found in Ref. [34]. Note that spatially, only the pressure term and the Laplacian operators for the viscous terms in the vertical direction are treated implicitly.

2.2. Parallelization strategy

The 2DECOMP [35] library is used to implement a two-dimensional domain decomposition, also known as "pencil" decomposition. We have extended the 2DECOMP library to suit the specifics of our scheme. For a pencil decomposition solving tridiagonal matrices in directions the pencils are not oriented in, requires re-orienting the pencils, and thus large all-to-all communications. We can avoid the solution of the tridiagonal matrices in the horizontal directions by integrating advection terms and viscous terms in the horizontal directions explicitly. As shown in Ref. [19], using the CFL time step constraint is sufficient to ensure stability for the wall-bounded direction for high Reynolds number flows. This is the reason why in this study, the Laplacian operators for the viscous terms are treated implicitly only in the vertical direction. Therefore, aligning the pencils in the wall normal (x) direction avoids all-to-all communications for the two horizontal directions. In this way, every processor possesses data from x_1 to x_N (cf. Fig. 2) and, for every pair (y, z), a single processor has the full x information needed to solve the implicit equation in x without further communication. We note that halo updates must still be performed during the computation of the intermediate velocity, but this memory distribution completely eliminates the all-to-all communications.

All-to-all communications are unavoidable during the pressure correction step, as a Poisson equation must be solved. Since the two wall-parallel directions are homogeneous and periodic, it is natural to solve the Poisson equation using a Fourier expansion in two dimensions. To do so, modified wavenumbers are used, instead of the real ones. Modified wavenumbers for the solution of a Poisson equation by finite-differences are mainly used to ensure the free divergence of the discrete velocity field. Since trigonometric expansions are used only in two directions the Poisson solver would have spectral accuracy in two directions and second-order accuracy in the third. This introduces an undesired spatial numerical anisotropy that might perturb the flow physics. Thus, modified wavenumbers prevent the Laplacian from having higher accuracy in some directions [29]. In the limit of infinite points, i.e. $\Delta y \to 0$, the modified wavenumbers converge to the real wavenumbers. In the CPU version, the Fast Fourier Transforms (FFT) are performed using the open source FFTW (http://www.fftw.org/) library.

By using a second-order approximation for the partial derivatives in the wall-bounded directions, the Poisson equation is reduced by two-dimensional fast Fourier transforms to a series of one-dimensional Poisson equations that are easily inverted by a tridiagonal Thomas solver. This allows for the direct solution of the Poisson equation in a single step, with a residual round-off-error velocity divergence ($\mathcal{O}(10^{-13})$ in double-precision arithmetics) within $\mathcal{O}(N_x N_y N_z \log[N_y] \log[N_z])$ time complexity. Due to the domain decomposition, several data transposes must be performed during the computation of the equation. The algorithm for solving the Poisson equation is as follows:

1. Calculate the local divergence from the x-decomposed velocities.
2. Transpose the result of *(1)* from a x-decomposition to a y-decomposition.
3. Perform a real-to-complex Fourier transform on *(2)* in the y-direction.
4. Transpose *(3)* from a y-decomposition to z-decomposition.
5. Perform a complex-to-complex Fourier transform on *(4)* in the z-direction.

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

6. Transpose *(5)* from a z-decomposition to a x-decomposition.
7. Solve the linear system with a tridiagonal solver in the x-direction.
8. Transpose the result of *(7)* from a x-decomposition to a z-decomposition.
9. Perform a complex-to-complex inverse Fourier transform on *(8)* in z-direction.
10. Transpose *(9)* from a z-decomposition to a y-decomposition.
11. Perform a complex-to-real inverse Fourier transform on *(10)* in a y direction.
12. Transpose *(11)* from a y-decomposition to a x-decomposition.

The last step outputs the scalar correction ϕ in real space, decomposed in x-oriented pencils. Therefore, once the Poisson equation is solved, the corrected velocities and pressures are computed directly. The temperature and other scalars are advected and the time sub-step is completed. For full details of the equations that are solved, we refer the reader to Ref. [19]. The algorithm outlined above only transposes one 3D array, making it very efficient. The top row of Fig. 2 shows a schematic of the data arrangement and the transposes needed to implement the algorithm in the original CPU code. We wish to highlight that this algorithm uses all possible data transposes. However, the standard 2DECOMP library only has four transpose schemes, namely `transpose_x_to_y`, `transpose_y_to_x`, `transpose_y_to_z`, and `transpose_z_to_y`, since these transpose routines are sufficient to construct common spectral solvers. As the Poisson solver described above requires transposes in all directions we have added the `transpose_x_to_z` and `transpose_z_to_x` routines to the 2DECOMP library version that is distributed with AFiD. It can be seen in Fig. 2 that the x-to-z and the z-to-x transposes need a more complex structure than the other transposes, as a process may need to transfer data to other processes which are not immediate neighbors. Therefore, in the CPU version of the code, these transposes have now been implemented using the more flexible `MPI_Neighbor_alltoallw` calls available in `MPI 3.0` instead of the `MPI_ALLTOALLV` calls used for the other four transposes. The added transpose routines are completely compatible with the standard 2DECOMP format and the updated library can thus be used in other codes straightforwardly. These new routines are not incorporated in the library version found on the 2DECOMP website, but the updated 2DECOMP library can be downloaded from https://github.com/PhysicsofFluids/AFiD_GPU_opensource. The GPU version uses a different transpose scheme that will be described later.

3. GPU implementation

In this section we explain the details of the GPU implementation of AFiD. It is now possible to program GPUs in several languages, from the original CUDA C to the new OpenACC directive based compilers. We decided to use CUDA Fortran (Section 3.1) as the nature of the code, where most routines are nested do-loops, allows the extensive use of CUF kernels (Section 3.2, kernel loop directives), making the effort comparable to an OpenACC port, while also retaining the possibility of using explicit code kernels when needed. In addition, the explicit nature of data movement in CUDA Fortran allows us to better optimize the CPU/GPU data movement and network traffic, and to further increase code performance. In Section 3.3 we describe the optimization of memory usage, and in Section 3.4 we present the multi-GPU aspects of the code. In addition, a new improved transpose scheme is introduced in Section 3.5.

3.1. CUDA & CUDA Fortran

CUDA-enabled GPUs can contain anything from a few to thousands of processor cores which are capable of running tens of thousands of threads concurrently. To allow for the same CUDA code to run efficiently on different GPUs with varying number of resources, a hierarchy of resources exists both in physical hardware, and in available programming models. In hardware, the processor cores on a GPU are grouped into multiprocessors. The programming model mimics this grouping: a subroutine, called a kernel, which runs on the device, is launched with a grid of threads grouped into thread blocks. Within a thread block data can be shared between threads, and there is a fine-grained thread and data parallelism. Thread blocks run independently of one another, which allows for scalability in the programming model: each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on a device with any number of multiprocessors. This scheduling is performed behind the scenes, the CUDA programmer needs only to partition the problem into coarse subproblems that can be solved independently in parallel by blocks of threads, where each sub-problem is solved cooperatively in parallel by all threads within the block. For the GPU runs, the whole computation is performed on the GPUs and all data are stored on the GPUs for the entire duration of the simulation, however in many simulations the data needs to be routed through CPU memory for further parallel communication.

The CUDA platform also enables hybrid computing, where both the host (CPU and its memory) and device (GPU and its memory) can be used to perform computations. From a performance perspective, the bandwidth of the PCI bus is over an order of magnitude less than the bandwidth between the device's memory and GPU, and therefore special emphasis needs to be placed on limiting and hiding PCI traffic. For MPI applications, data transfers between the host and device are required to transfer data between MPI processes. Therefore, the use of asynchronous data transfers, i.e. performing data transfers concurrently with computations, becomes mandatory. In order to achieve this goal we resort to chunked data transfers to ensure that the relatively slow transfers between CPUs over the PCI bus can already start for the processed data, while the GPUs are still performing calculations on the remaining data.

CUDA Fortran is essentially regular Fortran with a handful of extensions that allow portions of the computation to be off-loaded to the GPU. There are two compilers, at the moment, that are able to parse these extensions, the PGI compiler (now freely available via the community edition) and the IBM XLF compiler, which currently implements a subset of CUDA Fortran, in particular it does not have CUF kernels. Because we rely heavily on CUF kernels in our GPU implementation, all the results presented in this paper are obtained with the PGI compiler. CUDA Fortran has a series of extensions, like the variable attribute `device` used when declaring data that resides in GPU memory, the F2003 `sourced` allocation construct and the flexibility of kernels which make porting much easier. CUDA Fortran can automatically generate and invoke kernel code from a region of host code containing tightly nested loops. Such code is referred to as a CUF kernel. One can port code to the device using CUF kernels without modifying the contents of the loops using the following programming convention. The directive will appear as a comment to the compiler if GPU code generation is disabled or if the compiler does not support them (similar to the OpenMP directives that are ignored if OpenMP is not enabled). The contents of the loop are usually unaltered.

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

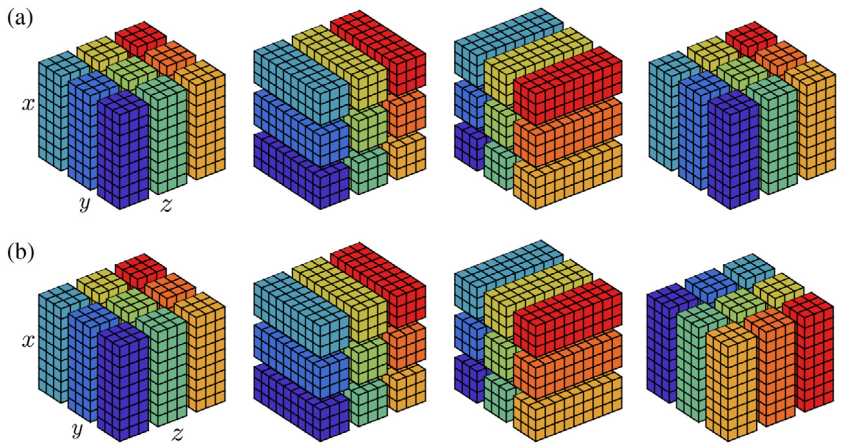

Fig. 2. Comparison between the original transpose scheme implemented in the CPU version (top, see Section 2.2) and the one implemented in the GPU version (bottom, see Section 3.5). Both transpose strategies start from the leftmost configuration, operate on the arrays, and transpose again ending up at the right-most configuration, solve the Poisson equations, and transpose data back to the leftmost configuration.

```
 subroutine CalcMaxCFL(cflm)
#ifdef USE_CUDA
 use cudafor
 use param,only:fp_kind,nxm, &
                dy=>dy_d,dz=>dz_d, &
                udx3m=>udx3m_d
 use local_arrays,only:vx=>vx_d, &
                       vy=>vy_d, &
                       vz=>vz_d
#else
 use param,only:fp_kind,nxm,dy,dz,udx3m
 use local_arrays,only:vx,vy,vz
#endif
 use decomp_2d
 use mpih
 implicit none
 real(fp_kind),intent(out):: cflm
 integer::i,j,k,ip,jp,kp
 real(fp_kind)::qcf

 cflm=real(0.00000001,fp_kind)

!$cuf kernel do(3) <<<*,*>>>
 do i=xstart(3),xend(3)
  ip=i+1
  do j=xstart(2),xend(2)
   jp=j+1
   do k=1,nxm
    kp=k+1
    qcf=( abs((vz(k,j,i)+vz(k,j,ip))*dz) &
         +abs((vy(k,j,i)+vy(k,jp,i))*dy) &
         +abs((vx(k,j,i)+vx(kp,j,i))*udx3m(k)))

    cflm = max(cflm,qcf*0.5_fp_kind)
   enddo
  enddo
 enddo

 call MpiAllMaxRealScalar(cflm)
 return
 end
```

Listing 1: Routine to compute the maximum CFL number.

3.2. CUF kernels

One of the project goals was to have a code as close as possible to the original CPU version. In order to accomplish this the GPU implementation makes extensive use of the preprocessor and all the GPU specific code and directives are guarded by *USE_CUDA* macro. For the same F90 source file, a CPU object file can be created with the standard optimization flags while a GPU version can be created adding the "-O3 -DUSE_CUDA -Mcuda" flags. While the GPU code needs to be compiled with the PGI compiler, the CPU code can be compiled with any Fortran compiler. The build system will build a copy of the code for GPU and one for CPU. The original CPU code uses custom allocators that allocate and initialize the arrays to zero. Some arrays are defined with halo cells, others only for the interior points. The 2DECOMP [35] library is also using global starting indices. In order to make identical copies on the GPU, we used the F2003 *sourced* allocation construct. It is worthwhile to point out that in the present version of AFID-GPU all data reside in the GPU memory during all calculations, i.e. each GPU holds the data corresponding to the pencil it is assigned in memory. The CPUs and its corresponding memory are only used during I/O operations and for data transfer between different processes during the communication stages. This choice compatibility with standard MPI implementations and therefore code portability.

Listing 1 shows an example of code modifications that allow compilation of the same source code for both the host (by default) and the GPU (by specifying the compiler option -DUSE_CUDA). As we can see from the source code, the CUF kernel directives are very simple to use. Once the compiler is aware that the 3 nested do loops need to be parallelized, it automatically determines that `cflm` requires a reduction. Using the renaming facilities when loading the variables from the module, we ensure that the CUF kernel will operate on arrays resident in GPU memory. We used CUF kernels extensively, and only some routines are coded manually on the GPU. One of these routines is the routine computing the statistics, since the reduction operator is on a vector, and, at the moment, CUF reductions only work on scalars. Note that by "statistics" we refer to physical quantities of interest, e.g. vertical mean velocity and temperature profiles. Since the considered flow is horizontally homogeneous, these physical quantities need to be averaged in both horizontal directions, which requires the mentioned reduction operation. We also wrote batched tridiagonal solvers in which each thread solves a different system with the Thomas algorithm using the locally available data and routines to transpose local arrays. These transposes are needed to optimize the memory layout before computationally intensive parts such as the tridiagonal solvers or FFT's, for which we use the CUFFT library. For the actual implementations we refer the reader to the code available at https://github.com/PhysicsofFluids/AFiD_GPU_opensource.

3.3. Reducing the memory footprint

With the computational power of the GPUs, the memory footprint becomes the limiting factor in increasing the resolution of the

simulations. Reducing the memory footprint becomes one of the main objectives. While there are now GPUs with up to 24 GB of memory, in most large systems the GPUs are older, and have less capacity, typically being 6 GB to 12 GB. Thus, even reducing the number of 3D arrays by a single unit results in relevant benefits. Since several routines require storing data that is only needed temporarily, either as an intermediate result or to transform the data layout, we are able to reduce the memory footprint by reusing the memory for these arrays as much as possible. In particular, there are two work arrays used in the Poisson solver that are used for either complex or real data types. In Fortran77, it was possible to use the `equivalence` statement to have the two arrays sharing the same memory. While `equivalence` is still supported (but deprecated) in Fortran90, it only works with statically defined arrays and the memory allocation in AFiD is all dynamic. Using the `iso_c_binding`, it is possible to reproduce the behavior of `equivalence`:

```
complex, target, allocatable:: complex_vec(:)
type(c_ptr):: cptr
real, pointer:: real_vec(:)

allocate(complex_vec(N))
cptr=c_loc(complex_vec)
call c_f_pointer(cptr,real_vec, &
 [2*size(complex_vec,1)])
```

This approach works for both CPU and GPU arrays (if the arrays are declared with the `device` attribute). Another area where memory can be reduced is in the workspace that is used by the FFT library. When creating an FFT plan with CUFFT, a workspace is allocated by the library which is roughly the same size as the data that will be processed by the plan. Since the four FFT plans in the solver will not be used simultaneously, we can reuse the same storage for all the workspaces by creating the FFT plans with the new CUFFT plan management application program interface (API, here we use NVTX which is introduced in Appendix B) that allows the programmer to provide the workspace memory. The initial GPU version of the code needed 48 K20x to run a 1024^3 grid, the final version can now run on 25 K20x with 6 GB of memory.

3.4. Multi GPU implementation

In the GPU implementation, we map each MPI rank to a GPU. The code discovers the available GPUs on each node and makes a 1:1 mapping between ranks and GPUs, as described in [36]. In the basic version of the code the whole computation is performed on the GPUs, and the CPUs are only used for I/O and to stage the data needed during the communication phases. There are MPI implementations that are GPU-aware and allow to use data resident in GPU memory directly in MPI calls, but for this initial version we used standard MPI to have a more portable code, so the data needs to be resident in CPU memory before the MPI calls. Instead of using `MPI_ALLTOALL` or `MPI_NEIGHBOR_ALLTOALLW` calls, we used a combination of `IRECV/ISEND` together with `cudaMemcpy2DAsync` to better overlap transfer to/from GPU memory from/to CPU memory and computations [37].

3.5. Efficient data transposes

AFiD was designed for high Reynolds number simulations and its parallel implementation is deeply tied to the underlying numerical scheme. As explained in Section 2.2, the code only needs to solve implicitly in the wall normal direction. AFiD uses a two-dimensional pencil decomposition aligned in the wall normal direction. Per time step, only six all-to-all communications are required, and these are all found in the Poisson solver for the pressure correction. The original 2DECOMP library had only four transposes available (no x-to-z and z-to-x, since the library was designed for full spectral solvers for which there is no need to go back to the original vertical decomposition). The AFiD code added the x-to-z and z-to-x transposes using the new `MPI 3.0 MPI_Neighbor_alltoallw` calls. Since in the GPU implementation we want to use combination of `IRECV/ISEND` calls that allow a better overlap of data transfer from/to GPU, this required a new transpose scheme. If we relax the constraint that the tridiagonal solvers are solved in a decomposition identical to the original one in which the right hand side (local divergence) of the Poisson equation was computed, we can devise a more efficient transpose. As shown in the bottom part of Fig. 2, if we apply another rotation from z to x (similar to what we would do with a Rubik's cube), each processor will only exchange data with other processors in the same row sub-communicator, similar to the previous stages and use combination of `IRECV/ISEND` calls. We are still using the 2DECOMP library to do the book keeping, and since the library uses global indices for addressing, we just need to access the proper wavenumbers to solve the tridiagonal systems. In principle, this new transpose scheme is applicable for the CPU version of the code. However, this has not been implemented yet in the CPU version of the code.

4. Code performance

In this section we first explain in Section 4.1 how the two dimensional decomposition is used before we explain the detailed performance tests in Section 4.2. We note that we use CPU and cores indistinctly, but when we refer to the number of CPUs or cores below we always refer to the number of computational cores.

4.1. Optimal configuration

Given the processor count, the code is able to find the optimal processor grid configuration. This is very important in production runs to efficiently use the allocated resources. The code will factor the total number of MPI tasks and try all the possible configurations, executing the transpose communication routines for a single substep (six transposes required for the Poisson solver), including the halo-exchange time. For the CPU version we measured the performance of the full simulation code to determine the most promising configuration. For the GPU version, with a low processor count, the optimal configurations are generally of the form $1 \times N$.

This is because four of the six transpositions are among processors in the first dimension (xy and xz directions), while only two transpositions are among processors in the second dimension (yz direction). Thus, a $1 \times N$ will minimize the amount of data that must be communicated between processors during the Poisson solver transpose routines. However, as the processor count is increased, the halo-exchange time becomes more dominant, and the best configuration becomes the two-dimensional decomposition which minimizes the halo-exchange communications for the GPU version of the code. Examples of the optimal configuration search for the CPU and GPU codes respectively can be found in Tables 1 and 2. It is also important to notice that the shape of the decomposition affects the strong scaling since the decomposition changes from 1D to 2D when increasing the processor count.

4.2. Performance comparison

All the cases that are shown in this manuscript have been performed with both the CPU and GPU versions of the code. The GPU results are identical to the CPU results up to machine precision, which indicates that the CPU and GPU versions of the code are consistent.

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

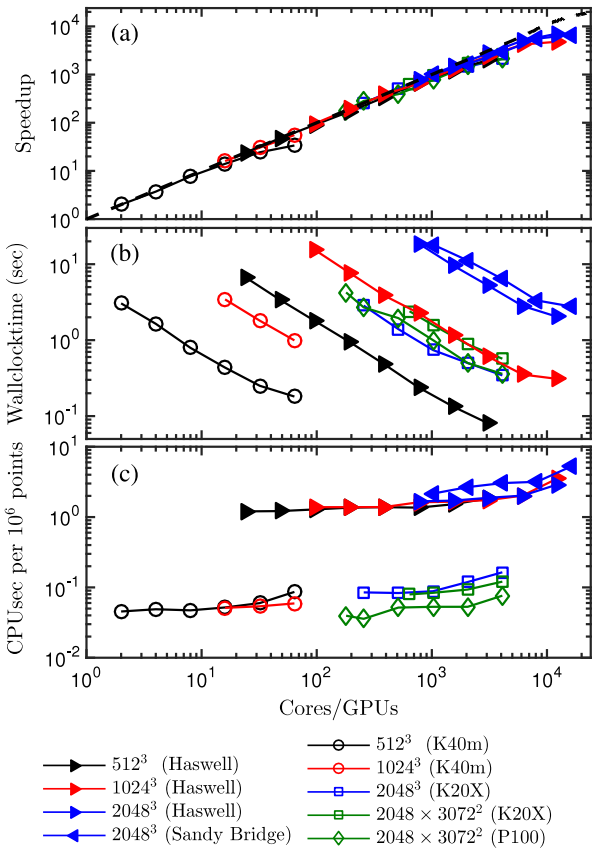

Fig. 3. Performance of AFiD on CPU and GPU. (a) Speedup. (b) Wall-clock time per time step vs the number of cores. For a fixed grid resolution an increase in this computational time is due to increased communication. With increasing grid resolution the required number of computations increases by more than a factor N due to the pressure solver. (c) CPU time per grid point per time step for the different test cases. The symbol color indicates the grid size, the solid and open symbols indicate whether the test was performed on CPU or GPU, while the symbol indicates the CPU/GPU model.

Table 1
An example of the generation of the optimal processor grid configuration for the CPU code.

In auto-tuning mode…
Processor grid 2 by 512 time = 0.433 s
Processor grid 4 by 256 time = 8.241E−002 s
Processor grid 8 by 128 time = 4.342E−002 s
Processor grid 16 by 64 time = 3.173E−002 s
Processor grid 32 by 32 time = 3.014E−002 s
Processor grid 64 by 16 time = 4.255E−002 s
Processor grid 128 by 8 time = 6.577E−002 s
Processor grid 256 by 4 time = 0.121 s
Processor grid 512 by 2 time = 0.230 s
The best processor grid is probably 32 by 32

Table 2
An example of the generation of the optimal processor grid configuration for the GPU code.

In auto-tuning mode…
Processor grid 1 by 18 time = 0.3238 s
Processor grid 2 by 9 time = 0.8386 s
Processor grid 3 by 6 time = 0.9210 s
Processor grid 6 by 3 time = 0.9363 s
Processor grid 9 by 2 time = 0.8577 s
Processor grid 18 by 1 time = 0.5901 s
The best processor grid is probably 1 by 18

The GPU runs were performed on two systems, the accelerator island of Cartesius at SURFsara and Piz Daint at the Swiss National Supercomputing Center (CSCS). The accelerator island of Cartesius consists of 66 Bullx B515 GPGPU accelerated nodes, each with two 8-core 2.5 GHz Intel Xeon E5-2450 v2 (Ivy Bridge) CPUs, 96 GB of memory and two 12 GB NVidia Tesla K40m GPUs. Every node has a FDR InfiniBand adapter providing 56 Gbit/s inter-node bandwidth.

Piz Daint was originally a Cray XC30 with 5272 nodes, each with an 8-core Intel Xeon E5-2670 v2 processor, 32 GB of system memory and a 6 GB NVidia K20X GPU. It has been upgraded to a Cray XC50 in November 2016. The compute nodes now have a 12-core Intel Xeon E5-2690 v3 processor, 64 GB of system memory and a 16 GB Nvidia P100 GPU. The new Pascal P100 GPU has 720 GB/s of peak memory bandwidth (and can sustain more than 500 GB/s in the STREAM benchmark) and more than 4.5 teraflops of double precision performance. The network is the same before and after the upgrade, and it uses the Aries routing and communications ASICs and a dragonfly network topology. Piz Daint is one of the most efficient petaflop class machines in the world: in the Green 500 list published in November 2013, the machine with XC30 nodes was able to achieve 3186 Mflops/W with level 3 measurements, the most accurate available. In June 2017, with the upgrades XC50 nodes, the machine was able to achieve 10,398 Mflops/W, more than tripling the power efficiency.

The physical problem used to test the scaling is of the Rayleigh–Bénard flow, which will be explained in detail in Section 5. The computational time per time step for a given grid resolution does not depend on the Rayleigh–Bénard control parameters. We measured the performance of the new accelerated code and compared it to the CPU performance reported in [19] on the Curie thin nodes (dual 8-core E5-2680 Sandy Bridge EP 2.7 GHz with 64 GB of memory and a full fat tree Infiniband QDR network) and with new measurement on Cartesius Haswell thin node islands (2 × 12-core 2.6 GHz Intel Xeon E5-2690 v3 Haswell nodes) with 64 GB of memory per node and 56 Gbit/s inter-node FDR InfiniBand, with an inter-island latency of 3 μs.

Here we supply the memory bandwidth and flops of the CPUs that are used to test the code. The Sandy Bridge cores on Curie have a peak memory bandwidth of 3.2 GB/s with a performance of 21.60 gigaflops. The Haswell cores on Cartesius have a peak memory bandwidth of 2.85 GB/s and a performance of 41.6 gigaflops. The K20X GPU cards on Piz Daint have a peak memory bandwidth of 250 GB/s and a performance of 1312 gigaflops, while the P100 GPU cards have a peak memory bandwidth of 732 GB/s and a performance of 4761 gigaflops.

Fig. 3 shows the scaling data obtained for the CPU and GPU versions of the code. Panel (a) shows that both the CPU and GPU versions of the code show strong scaling on grids ranging from 512^3 up to 2048 ×3072 ×3072. Panel (c) combines a weak and strong scaling test together. This panel indicates the performance of the code for different grid sizes. One can see that for the cases considered here the normalized performance (and therefore the weak scaling of the code) is excellent. It can also be seen that the required number of GPUs to obtain the same wall-clock time as with the CPU version of the code is much smaller. Moreover, the figure reveals that we now obtain a better performance and scaling with the CPU version of the code than before, see [19]. We find that for 1024 cores the new code tested on Haswell is about 26% faster than the previous code version that was tested on Sandy bridge, while the difference is about 60% for 8192 cores. The much better performance on the higher core numbers we therefore ascribe to the code improvements. If we focus on the 2048^3 grid (Table 3), simulations performed on 1024 K20X GPUs are 20 times faster than on 1024 Haswell cores. As indicated above, the memory bandwidth ratio between the two is about 88, while the ratio in peak flop rate is about 31.5. The performance ratio between the two is about a

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

Table 3
Wall-clock time per step on a 2048^3 grid. In the CPU simulations there are as many MPI tasks as CPU cores. In the GPU simulation, there are as many MPI tasks as GPUs. This table compares the performance on a node level.

Nodes	Curie		Cartesius		Piz Daint		
	Xeon E5-2680		Xeon E5-2690		Tesla	K20x	P100
	Cores	Time	Cores	Time	GPUs	Time	Time
32	–	–	768	18.70 s	–	–	–
64	1024	18.10 s	1536	9.58 s	–	–	–
128	2048	11.18 s	3572	5.25 s	–	–	2.57 s
256	4096	6.38 s	6144	2.82 s	256	2.85 s	1.42 s
512	8192	3.33 s	12288	2.03 s	512	1.40 s	0.85 s
1024	16384	2.77 s	–	–	1024	0.74 s	0.46 s
2048	–	–	–	–	2048	0.50 s	–
4096	–	–	–	–	4096	0.34 s	–

Table 4
Wall-clock time per step on a $2048 \times 3072 \times 3072$ grid. Comparison between the XC30 nodes with Tesla K20X GPUs vs XC50 nodes with Tesla P100 GPUs. Note that scaling tests were performed on 2^n GPU cards and the minimum number of GPU cards that is required.

Nodes	Configuration	K20X	P100
180	1×180	–	4.25 s
256	1×256	–	2.7 s
512	1×512	–	1.95 s
640	64×10	2.4 s	–
1024	64×16	1.58 s	1.00 s
2048	64×32	0.88 s	0.5 s
4096	64×64	0.57 s	0.36 s

factor 12 when 4096 GPUs or CPUs are considered. Table 4 shows a comparison between the old XC30 nodes and the new XC50 nodes on Piz Daint for a larger problem on a 2048 ×3072 ×3072 mesh. We can notice the switch from the 1D decomposition to the 2D decomposition when increasing the processor count. The larger memory on the P100 (16 GB) vs K20X (6 GB) makes it possible to run this simulation on 180 nodes and will also allow the use of even finer meshes.

Comparing the results for the K20X and P100 GPUs we find that the code runs about 40% faster on the latter, while both the peak performance (times 3.62) and the maximum bandwidth (times 2.93) have increased more. The reason for this is that the Piz Daint network was not upgraded when upgrading from K20X to P100 GPUs. This emphasizes that, although the CPU and GPU memory bandwidth are important, also other factors such as network connections can significantly influence our code performance. In Fig. 3(b) we see that 128 *P*100 GPUs have similar performance as about 6000 Haswell CPU cores and while the CPU code is reaching a plateau in efficiency, the GPU code can still scale very well and bring the wall-clock time to levels unreachable by the CPU version. Since wall-clock time is a very important metric for DNS this is a crucial benefit of the GPU version of the code. Table 3 shows that on a XC50 node with a single P100 GPU is about twice as fast as a Xeon E5-2690 Haswell node with 24 cores. Its also important to note that Fig. 3(b) shows that for large grids the wall-clock time per time step can be about an order of magnitude lower on a GPU platform than on a CPU platform while maintaining good computational efficiency, i.e. a good strong scaling performance.

5. Validation

5.1. Rayleigh–Bénard convection

We simulated Rayleigh–Bénard convection in an aspect ratio $\Gamma = L/H = 1$ cell, where L indicates the streamwise and spanwise domain lengths compared to the domain height H. The control parameters of the system are the non-dimensional temperature difference between the plates, i.e. the Rayleigh number Ra, and the fluid Prandtl number, see Refs. [1,19] for more details.

Table 5
The employed Rayleigh numbers Ra and grid resolution in the horizontal $N_z \times N_y$ and wall-normal N_x directions, and the extracted Nu from the simulations. For all cases the Prandtl number Pr and aspect ratio Γ are unity.

Ra	$N_z \times N_y \times N_x$	Nu
10^7	$256 \times 256 \times 192$	17.17
10^8	$384 \times 384 \times 256$	32.20
10^9	$512 \times 512 \times 384$	64.13
10^{10}	$768 \times 768 \times 512$	132.68
10^{11}	$768 \times 768 \times 1296$	275.33

To test the code we look at the main response parameter of the Rayleigh–Bénard system, which is the non-dimensional heat transport between the two plates, i.e. the Nusselt number. Table 5 shows the simulation details and the extracted Nusselt number for each simulation. In Fig. 4 we show snapshots of the flow obtained at different Rayleigh number. The figures reveal that the flow structures rapidly decrease with increasing Rayleigh, illustrating the need of powerful computer codes to simulate very high Rayleigh number flows.

In Fig. 5, we show the obtained Nusselt number vs Rayleigh number compared against experimental data [13,41–50] and the predictions by the Grossmann–Lohse theory [38–40]. The figure shows that experiments, simulations, and theory are in very good agreement with each other up to $Ra = 10^{11}$. This figure also shows that there are two facilities (in Grenoble [42,51,52]) which show an increased Nusselt number already around $Ra = 5 \times 10^{11}$, while other experiments (in Göttingen [13,48,53,54]) show this transitions around $Ra_1^* \approx 2 \times 10^{13}$ and $Ra_2^* \approx 7 \times 10^{13}$. There is no clear explanation for the mentioned disagreement although it is conjectured that unavoidable variations of the Prandtl number [1,55], finite conductivity [1,55–57] of the horizontal plates and sidewall [58–60], non Oberbeck–Boussinesq effects [61–65], i.e. the dependence of the fluid properties on the temperature, and even wall roughness [66,67] and temperature conditions outside the cell might play a role. So far the origin of this discrepancy could never be settled, in spite of major efforts. For more information on this topic, we refer the readers to Refs. [1,3].

In order to help to clarify these issues it is important to perform DNS with the precise assignment of the temperature boundary conditions (i.e. strictly constant temperature horizontal plates and adiabatic sidewall), infinitely smooth surfaces and unconditional validity of the Boussinesq approximation, i.e. the fluid properties do not depend on the temperature, which is hard coded in the model equations. In addition, in contrast to experiments, numerical simulations of turbulent flows have the huge advantage that all quantities of the flow are fully accessible while it is possible to adjust the control parameters arbitrarily with the goal to better understand the physics of the system. Our desire to study the transition to the ultimate Rayleigh–Bénard convection in simulations motivates our development of ever more powerful simulation codes. It should be noted that Fig. 4 shows that the GPU code already allows one to perform large simulations in a much shorter wall-clock time, which makes the execution of such simulations much more practical.

5.2. Plane Couette flow

Now we test the code with the plane Couette configuration at bulk Reynolds number $Re_c = 3000$. The two walls here move with the same speed u_c but in the opposite direction. Table 6 shows the employed parameters and the output friction velocity. To capture the large scale structure of plane Couette, a rather large domain size as $18\pi h \times 8\pi h \times 2h$, where h is the half height of the channel,

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

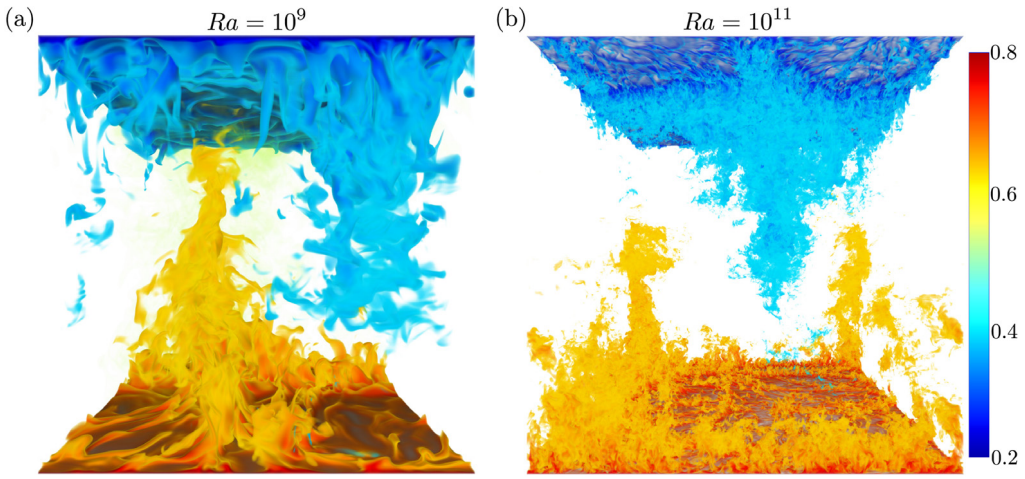

Fig. 4. Visualization of the temperature field at $Ra = 10^9$ and $Ra = 10^{11}$ (rendered on a coarser grid than used during the simulation) for $Pr = 1$ in a horizontally periodic $\Gamma = 1$ cell. The colorbar indicates the non-dimensional temperature, with the range of 0.2 (blue) to 0.8 (red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

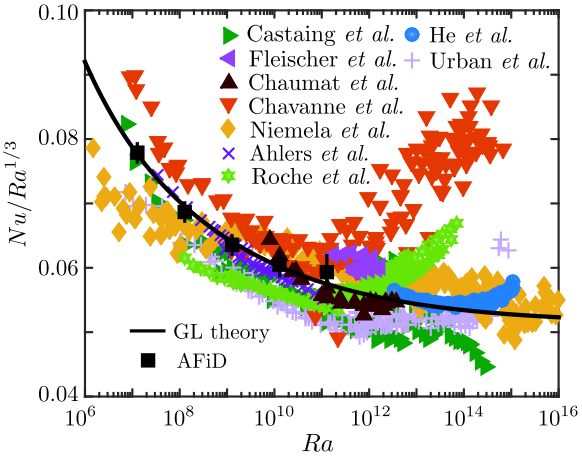

Fig. 5. The dimensionless heat flux, i.e. the Nusselt number Nu, as a function of the dimensionless temperature difference between the plates, i.e. the Rayleigh number Ra, obtained using AFiD, in the compensated way, in comparison with the Grossmann–Lohse theory [38–40] and with the experimental data from Castaing et al. [41], Roche et al. [42], Fleischer & Goldstein [43], Chaumat et al. [44], Chavanne et al. [45], Niemela et al. [46], Ahlers et al. [47,48], He et al. [13], and Urban et al. [49,50]. The experimental data and GL theory presented in this figure are the same as in Ref. [40].

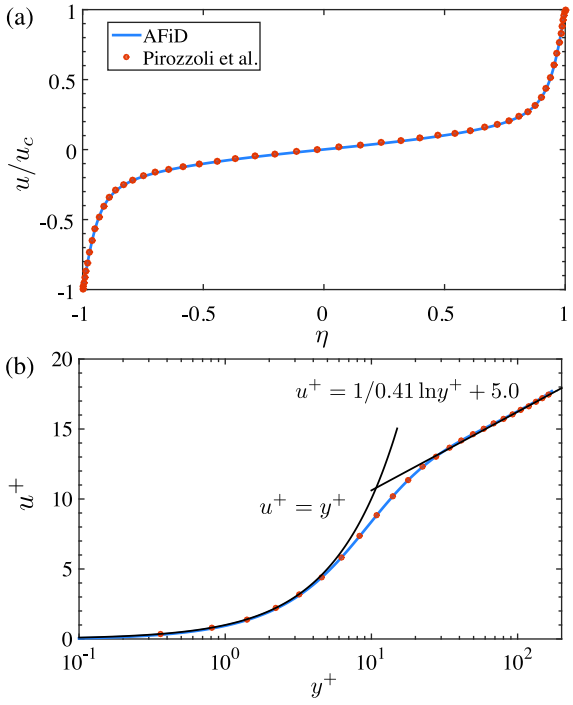

Fig. 6. Mean velocity profiles in plane Couette flow scaled with (a) the wall moving velocity u_c and (b) friction velocity $u^+ = u/u_\tau$. η is the dimensionless wall normal coordinate scaled with the channel half height h, in the way that $\eta = 0$ corresponds to the channel centerline and $\eta = \pm 1$ corresponds to the two walls. $y^+ = u_\tau(\eta + 1)/\nu$ is the dimensionless distance in wall units. The AFiD results agree excellently with the DNS from Pirozzoli et al. [14].

has to be used. The resulting friction Reynolds number $Re_\tau = 171$, which is in excellent agreement with the previous results for the same configuration [14].

The streamwise mean velocity profile is shown in Fig. 6, normalized with either the wall velocity u_c or friction velocity u_τ. Again, excellent agreement has been found between the current study and Ref. [14]. For Fig. 6(b), two clear layers can be identified. When $y^+ < 5$, the profile follows $u^+ = y^+$, which is called the viscous layer; When $50 < y^+ < 171$, a clear logarithmic layer is seen, with $u^+ = 1/\kappa \ln y^+ + C$, where $\kappa \approx 0.41$ and $C \approx 5.0$.

Fig. 7 shows the large scale flow structure of the plane Couette flow. Distinctive patterns of high and low speed streaks are evident, which maintains the coherence along the whole streamwise length of the channel, while also showing some meandering. The spanwise width of the large scale structure is 4 to 5h, as also shown in previous studies [14,68]. The above findings demonstrate the necessity for extremely large box to capture the biggest flow structure in plane Couette flow. This is the reason the biggest DNS for plane Couette flow only reached $Re_\tau \approx 1000$ [14], while for channel flow $Re_\tau \approx 5200$ [6] is already obtained.

With this GPU version of the code, our goal is to study on the one hand even bigger box sizes, which will help understand how big the large scale structure can really be. On the other hand, we want to study higher Reynolds number plane Couette flow, which will help understand how far the logarithmic layer can extend and whether the attached eddy hypothesis [69] is applicable for plane Couette flow.

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

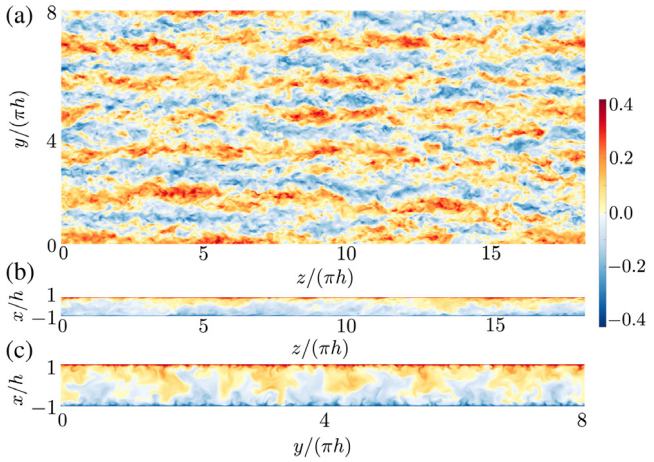

Fig. 7. Contours of the instantaneous streamwise flow velocity in (a) the channel center, (b) cross-spanwise view, and (c) cross-streamwise view.

Table 6
List of parameters for the plane Couette flow case. Here $Re_c = hu_c/\nu$ is the bulk Reynolds number and h is the channel half height, u_c the moving velocity of wall, ν the kinematic viscosity. The second column shows the computational box. The third column shows the grid resolution. The last column is the friction Reynolds number $Re_\tau = hu_\tau/\nu$, where u_τ is defined as $u_\tau = \sqrt{\tau_w/\rho}$, in which τ_w is the wall shear stress. Note that Re_τ is a result of the simulation.

Re_c	$L_z \times L_y \times L_x$	$N_z \times N_y \times N_x$	Re_τ
3000	$18\pi h \times 8\pi h \times 2h$	$1280 \times 1024 \times 256$	171

6. Conclusions and future plans

In this paper we presented a GPU accelerated solver that can be used to study various wall-bounded flows [1,2,7,8,69–73]. Our work is motived by the need to simulate more extreme turbulent flows and inspired by the observation that while high performance computing shifts towards GPUs and accelerators, obtaining an efficient GPU code, that is faster than is CPU, is thought to be a very time consuming, code specific, undertaking. In this paper we showed that to port CPU code to the GPU, only "minimal effort" is required. In addition, we show that for large grids the GPU code obtains good computational efficiency for wall-clock times that are an order of magnitude smaller than what can be achieved with the CPU code. In this work we presented some efficient coding techniques, such as overloaded sourced allocation and how module use/renaming can be used to avoid modifying loop contents that have not been covered elsewhere. In addition, we point out that this approach allows for easy code validation, since every subroutine can be examined to produce the same results as the original CPU code up to the machine precision. This approach is generally applicable and an eye opener for many scientists thinking about GPU porting.

Previous work to parallelize second-order finite-difference solvers allowed us to reach extremely high Reynolds numbers in Taylor–Couette flow [74], and also to simulate Taylor–Couette flow with riblets in the flow direction and notches perpendicular to the flow direction to disentangle the effects of roughness on the torque [75,76]. For Rayleigh–Bénard convection we have used the AFiD code to simulate unprecedentedly large horizontal domains to investigate the formation of thermal superstructures in Rayleigh–Bénard [20]. The need to obtain ever more efficient codes is illustrated by example use cases of high Rayleigh number turbulent Rayleigh–Bénard convection and high Reynolds number plane Couette flow. We have shown in Section 5 that our code can work perfectly for both cases. With the GPU code described here, the capability of the code is improved even further. Initial works have been started to simulate the Rayleigh–Bénard flows with external shearing by using the GPU version. It should be pointed out that the code can also be used to simulate other wall-turbulent flow configurations such as channel flow at high Reynolds number.

In order to further expand the capabilities of the code, we are going to work on several fronts. The first one will be to utilize the CPU cores, which are completely idle in the basic code version described here, together with the GPUs in the implicit part of the solver. We are more interested in the CPU memory than the CPU flops, but depending on the node configuration, the CPU cores can give a good performance boost. Subdividing each vertical domain in two subdomains, we can process one on the CPU and one on the GPU. The relative size of the subdomains can be determined at runtime, since the workload per cell is constant. The split is in the outermost dimension (z) and requires additional halo exchanges (but these are local memory transfers of contiguous data between GPU and CPU, so no network is involved). A preliminary version of this hybrid CPU–GPU code is available in the open-source code, which will be discussed in detail in a forthcoming paper, and is currently being tested.

As explained above the code is GPU-centric, which means that all data required for computations resides in the GPU memory. Since writing the HDF5 files to disk could be time consuming, we are thinking about making this process asynchronous, once the solution is copied to CPU memory, the GPU can advance the solution while the CPUs complete the I/O.

Another way to optimize the simulations is to use a multiple resolution approach, using a grid for the temperature field with a higher spatial resolution than that for the momentum, as integrating both fields on a single grid tailored to the most demanding variable produces an unnecessary computational overhead. This approach gives significant savings in computational time and memory occupancy as most resources are spent on solving the momentum equations (about 80%–90%). To ensure stable time integration of the temperature field we use a separate refined time step procedure for the temperature field. The full details of the strategy are described in [28].

The GPU code is available at https://github.com/PhysicsofFluids/AFiD_GPU_opensource.

Acknowledgments

We thank Alexander Blass for providing the data on the plane Couette flow. This work was supported by a grant from the Swiss National Supercomputing Center (CSCS) under project ID g33 and by the Netherlands Center for Multiscale Catalytic Energy Conversion (MCEC), an NWO Gravitation program funded by the Ministry of Education, Culture and Science of the government of the Netherlands, the Foundation for Fundamental Matter (FOM) in the Netherlands, and the ERC Advanced Grant "Physics of boiling". We also acknowledge PRACE for awarding us access to FERMI and Marconi based in Italy at CINECA under PRACE project number 2015133124 and 2016143351 and NWO for granting us computational time on Cartesius from the Dutch Supercomputing Consortium SURFsara and for the continuous support we get from SURFsara on code development.

Appendix A. Remarks on the location of temperature in the mesh

In a Boussinesq fluid temperature T and density ρ are related by $(\rho - \rho_0)/\rho_0 = -\alpha(T - T_0)$, ρ_0, T_0 being the reference state where the fluid is at rest. Then, if the gravity vector $\mathbf{g}$ is anti-parallel to the

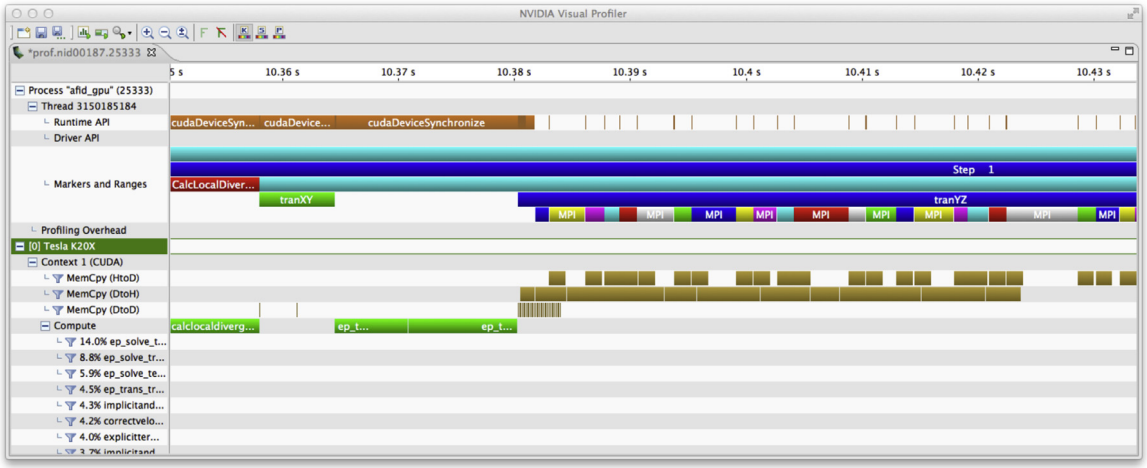

Fig. 8. Profiler output from AFiD-GPU for a parallel run on a 1024^3 grid.

x direction, the equations of motion can be written as:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\frac{\nabla p}{\rho_0} + \frac{g\hat{x}\rho}{\rho_0} + \nu\nabla^2\mathbf{u},$$

$$\frac{\partial \rho}{\partial t} + \mathbf{u}\cdot\nabla\rho = +\kappa\nabla^2\rho,$$

with ν and κ the diffusivities of momentum and density, respectively and $\hat{x}$ the unit vector in the vertical direction. If the velocity has components $\mathbf{u} = (u, v, w)$ we can define the kinetic energy as $E_k = \rho_0 \int_V (u^2 + v^2 + w^2) dV/2$ and the potential energy $E_p = \int_V \rho g x dV$ and derive from the above equations the energy balance for E_k and E_p. The detailed derivation can be found in many papers (e.g. Ref. [77]), here it suffices to mention that for a laterally confined (or periodic) fluid and for vanishing diffusivities (ν and κ) the balance equations reduce to

$$\frac{dE_k}{dt} = -\int_V \rho g w dV,$$

$$\frac{dE_p}{dt} = +\int_V \rho g w dV.$$

These equations simply state that the buoyancy flux is the reversible rate of change of potential energy that is converted into kinetic energy and vice versa. It is also easy to show that the rate of change of $E_k + E_p$ is zero since the two source terms cancel out. This is true, however only if the two terms are computed in the same way also at the discrete level and in order for this to be true ρ (or T) and w must be placed at the same position in the computational cell. If w and ρ were staggered by $\Delta/2$ the local source term would be computed as $g w_i(\rho_i + \rho_{i-1})/2$ in the kinetic energy equation and $g\rho_i(w_i + w_{i+1})/2$ in the potential energy equation and the two integrals would not cancel out. Thus, to ensure energy conservation, temperature is placed at the same location as the vertical velocity in the mesh.

Appendix B. Profiling using NVTX

Profiling is an essential part of performance tuning used to identify parts of the code that may require additional attention. When dealing with GPU codes, profiling is even more important as new opportunities for better interactions between the CPUs and the GPUs can be discovered. The standard profiling tools in CUDA, nvprof and nvvp, are able to show the GPU timeline but do not present CPU activity. The NVIDIA Tools Extension (NVTX) is a C-based API (application program interface) to annotate the profiler time line with events and ranges and to customize their appearance and assign names to resources such as CPU threads and devices [78].

We have written a Fortran module to instrument CUDA/OpenACC Fortran codes using the Fortran ISO C bindings [79]. To eliminate profiling overhead during production runs, we use a preprocessor variable to make the profiling calls return immediately. During the runs, one or more MPI processes generate the traces that are later imported and visualized with nvvp, the NVIDIA Visual Profiler. Fig. 8 shows an example of the output for AFiD_GPU on a 1024^3 mesh, where on the top part "process AFiD GPU" the CPU sections can be identified while the GPU sections are on the lower "Tesla K20x" section. The profiler is visualizing the output from one of the ranks. Since the run was on a 1×16 processor grid, we can see that after the computation of the local divergence (red box labeled CalcLocal) the first transpose, TranXY, does not require MPI communications. The following one, TranYZ, requires MPI communications and we can see the overlapping of Memcopy DtoH (device to host) and HtoD (host to device) with MPI calls.

References

[1] G. Ahlers, S. Grossmann, D. Lohse, Rev. Modern Phys. 81 (2009) 503.
[2] D. Lohse, K.-Q. Xia, Annu. Rev. Fluid Mech. 42 (2010) 335–364.
[3] F. Chilla, J. Schumacher, Eur. Phys. J. E 35 (2012) 58.
[4] B. Eckhardt, T. Schneider, B. Hof, J. Westerweel, Annu. Rev. Fluid Mech. 39 (2007) 447–468.
[5] J. Kim, P. Moin, R. Moser, J. Fluid Mech. 177 (1987) 133–166.
[6] M. Lee, R.D. Moser, J. Fuid Mech. 774 (2015) 395–415.
[7] M.A. Fardin, C. Perge, N. Taberlet, Soft Matter 10 (2014) 3523–3535.
[8] S. Grossmann, D. Lohse, C. Sun, Annu. Rev. Fluid Mech 48 (2016) 53–80.
[9] D.L. Hartmann, L.A. Moy, Q. Fu, J. Clim. 14 (2001) 4495–4511.
[10] J. Marshall, F. Schott, Rev. Geophys. 37 (1999) 1–64.
[11] P. Cardin, P. Olson, Phys. Earth Planet. Inter. 82 (1994) 235–259.
[12] F. Cattaneo, T. Emonet, N. Weiss, Astrophys. J. 588 (2003) 1183–1198.
[13] X. He, D. Funfschilling, H. Nobach, E. Bodenschatz, G. Ahlers, Phys. Rev. Lett. 108 (2012) 024502.
[14] S. Pirozzoli, M. Bernardini, P. Orlandi, J. Fluid Mech. 758 (2014) 327–343.
[15] H. Schlichting, K. Gersten, Boundary Layer Theory, eighth ed., Springer Verlag, Berlin, 2000.
[16] F. Waleffe, Phys. Fluids 9 (4) (1997) 883–900.
[17] S. Pirozzoli, M. Bernardini, P. Orlandi, J. Fluid Mech. 680 (2011) 534–563.
[18] D.P. McKenzie, J.M. Roberts, N.O. Weiss, J. Fluid Mech. 62 (1974) 465–538.
[19] E.P. van der Poel, R. Ostilla-Mónico, J. Donners, R. Verzicco, Comput. & Fluids 116 (2015) 10–16.
[20] R.J.A.M. Stevens, A. Blass, X. Zhu, R. Verzicco, D. Lohse, Phys. Rev. Fluids 3 (041501(R)) (2018).

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.

[21] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, SIGARCH Comput. Archit. News 38 (3) (2010) 451–460.
[22] P. Fischer, J. Lottes, S. Kerkemeier, A. Obabko, K. Heisey, nEK5000 webpage, 2008, http://nek5000.mcs.anl.gov.
[23] OpenFOAM, http://www.openfoam.org.
[24] 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. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. Kirby, S. Sherwin, Comput. Phys. Comm. 192 (2015) 205–219.
[25] J.F. Gibson, Channelflow: A Spectral Navier–Stokes Simulator in C++, Tech. Rep., U. New Hampshire, Channelflow.org, 2014.
[26] R.W. Schmitt, Annu. Rev. Fluid Mech. 26 (1) (1994) 255–285.
[27] P. Orlandi, Fluid Flow Phenomena: A Numerical Toolkit, Vol. 55, Springer Science and Media, 2012.
[28] R. Ostilla-Mónico, Y. Yang, E.P. van der Poel, D. Lohse, R. Verzicco, J. Comput. Phys. 301 (2015) 308–321.
[29] P. Moin, R. Verzicco, Eur. J. Mech. B Fluids 55 (2016) 242–245.
[30] M.M. Rai, P. Moin, J. Comput. Phys. 96 (1991) 15–53.
[31] A.J. Chorin, Bull. Amer. Math. Soc. 73 (6) (1967) 928–931.
[32] A.J. Chorin, Math. Comp. 22 (104) (1968) 745–762.
[33] M.J. Lee, B.D. Oh, Y.B. Kim, J. Comput. Phys. 168 (2001) 73–100.
[34] R. Verzicco, P. Orlandi, J. Comput. Phys. 123 (1996) 402–413.
[35] N. Li, S. Laizet, Cray User Group 2010 Conference, Edinburgh, 2010.
[36] G. Ruetsch, M. Fatica, CUDA Fortran for Scientists and Engineers, Morgan Kaufmann, 2013.
[37] M. Bernaschi, M. Bisson, M. Fatica, E. Phillips, Eur. Phys. J. Spec. Top. 210 (2002) 17–31.
[38] S. Grossmann, D. Lohse, J. Fluid. Mech. 407 (2000) 27–56.
[39] S. Grossmann, D. Lohse, Phys. Rev. Lett. 86 (2001) 3316–3319.
[40] R.J.A.M. Stevens, E.P. van der Poel, S. Grossmann, D. Lohse, J. Fluid Mech. 730 (2013) 295–308.
[41] B. Castaing, G. Gunaratne, F. Heslot, L. Kadanoff, A. Libchaber, S. Thomae, X.Z. Wu, S. Zaleski, G. Zanetti, J. Fluid Mech. 204 (1989) 1–30.
[42] P.E. Roche, G. Gauthier, R. Kaiser, J. Salort, New J. Phys. 12 (2010) 085014.
[43] A.S. Fleischer, R.J. Goldstein, J. Fluid Mech. 469 (2002) 1–12.
[44] S. Chaumat, B. Castaing, F. Chilla, in: I.P. Castro, P.E. Hancock, T.G. Thomas (Eds.), Advances in Turbulence IX, International Center for Numerical Methods in Engineering, CIMNE, Barcelona, 2002.
[45] X. Chavanne, F. Chilla, B. Chabaud, B. Castaing, B. Hebral, Phys. Fluids 13 (2001) 1300–1320.
[46] J. Niemela, L. Skrbek, K.R. Sreenivasan, R. Donnelly, Nature 404 (2000) 837–840.
[47] G. Ahlers, E. Bodenschatz, D. Funfschilling, J. Hogg, J. Fluid Mech. 641 (2009) 157–167.
[48] G. Ahlers, X. He, D. Funfschilling, E. Bodenschatz, New J. Phys. 14 (2012) 103012.
[49] P. Urban, V. Musilová, L. Skrbek, Phys. Rev. Lett. 107 (1) (2011) 014302.
[50] P. Urban, P. Hanzelka, T. Kralik, V. Musilova, A. Srnka, L. Skrbek, Phys. Rev. Lett. 109 (15) (2012) 154301.
[51] X. Chavanne, F. Chilla, B. Castaing, B. Hebral, B. Chabaud, J. Chaussy, Phys. Rev. Lett. 79 (1997) 3648–3651.
[52] P.E. Roche, B. Castaing, B. Chabaud, B. Hebral, Phys. Rev. E 63 (2001) 045303.
[53] X. He, D. Funfschilling, E. Bodenschatz, G. Ahlers, New J. Phys. 14 (2012) 063030.
[54] X. He, D.P.M. van Gils, E. Bodenschatz, G. Ahlers, New J. Phys. 17 (2015) 063028.
[55] R.J.A.M. Stevens, D. Lohse, R. Verzicco, J. Fluid Mech. 688 (2011) 31–43.
[56] R. Verzicco, Phys. Fluids 16 (2004) 1965–1979.
[57] E. Brown, D. Funfschilling, A. Nikolaenko, G. Ahlers, Phys. Fluids 17 (2005) 075108.
[58] G. Ahlers, Phys. Rev. E 63 (2000) 015303(R).
[59] R. Verzicco, J. Fluid Mech. 473 (2002) 201–210.
[60] R.J.A.M. Stevens, D. Lohse, R. Verzicco, J. Fluid Mech 741 (2014) 1.
[61] G. Ahlers, E. Brown, F. Fontenele Araujo, D. Funfschilling, S. Grossmann, D. Lohse, J. Fluid Mech. 569 (2006) 409–445.
[62] G. Ahlers, F. Fontenele Araujo, D. Funfschilling, S. Grossmann, D. Lohse, Phys. Rev. Lett. 98 (2007) 054501.
[63] G. Ahlers, E. Calzavarini, F. Fontenele Araujo, D. Funfschilling, S. Grossmann, D. Lohse, K. Sugiyama, Phys. Rev. E 77 (2008) 046302.
[64] K. Sugiyama, E. Calzavarini, S. Grossmann, D. Lohse, J. Fluid Mech. 637 (2009) 105–135.
[65] S. Horn, O. Shishkina, C. Wagner, J. Fluid Mech. 724 (2013) 175–202.
[66] J. Salort, O. Liot, E. Rusaouen, F. Seychelles, J.-C. Tisserand, M. Creyssels, B. Castaing, F. Chilla, Phys. Fluids 26 (2014) 015112.
[67] S. Wagner, O. Shishkina, J. Fluid Mech. 763 (2015) 109–135.
[68] V. Avsarkisov, S. Hoyas, M. Oberlack, J. García-Galache, J. Fluid Mech. 751 (2014) R1.
[69] I. Marusic, B.J. McKeon, P.A. Monkewitz, H.M. Nagib, A.J. Smits, K.R. Sreenivasan, Phys. Fluids 22 (6) (2010) 065103.
[70] A.J. Smits, B.J. McKeon, I. Marusic, Annu. Rev. Fluid Mech. 43 (2011) 353–375.
[71] J. Jimenez, Annu. Rev. Fluid Mech. 44 (2012) 27–45.
[72] A.J. Smits, I. Marusic, Phys. Today 66 (9) (2013) 25–30.
[73] I. Marusic, J.P. Monty, M. Hultmark, A.J. Smits, J. Fluid. Mech. 716 (2013) R3.
[74] R. Ostilla-Mónico, R. Verzicco, S. Grossmann, D. Lohse, J. Fluid Mech. 768 (2016) 95–117.
[75] X. Zhu, R. Ostilla-Mónico, R. Verzicco, D. Lohse, J. Fluid Mech. 794 (2016) 746–774.
[76] X. Zhu, R. Verzicco, D. Lohse, J. Fluid Mech. 812 (2017) 279–293.
[77] K.B. Winters, P.N. Lombard, J.J. Riley, E.A. D'Asaro, J. Fluid Mech. 289 (1995) 115–128.
[78] http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx. (Accessed 9 April 2018).
[79] https://devblogs.nvidia.com/parallelforall/customize-cuda-fortran-profiling-nvtx. (Accessed on 9 April 2018).

Please cite this article in press as: X. Zhu, et al., AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Computer Physics Communications (2018), https://doi.org/10.1016/j.cpc.2018.03.026.