A GPU-ready pseudo-spectral method for direct numerical simulations of multiphase turbulence

Alessio Roccon[a,b,*]

ᵃPolytechnic Department of Engineering and Architecture, University of Udine, 33100 Udine, Italy
ᵇInstitute of Fluid Mechanics and Heat Transfer, TU-Wien, 1060 Vienna, Austria

Abstract

In this work, we detail the GPU-porting of an in-house pseudo-spectral solver tailored towards large-scale simulations of interface-resolved simulation of drop- and bubble-laden turbulent flows. The code relies on direct numerical simulation of the Navier-Stokes equations, used to describe the flow field, coupled with a phase-field method, used to describe the shape, deformation, and topological changes of the interface of the drops or bubbles. The governing equations – Navier-Stokes and Cahn-Hilliard equations – are solved using a pseudo-spectral method that relies on transforming the variables in the wavenumber space. The code targets large-scale simulations of drop- and bubble-laden turbulent flows and relies on a multilevel parallelism. The first level of parallelism relies on the message-passing interface (MPI) and is used on multi-core architectures in CPU-based infrastructures. A second level of parallelism relies on OpenACC directives and cuFFT libraries and is used to accelerate the code execution when GPU-based infrastructures are targeted. The resulting multiphase flow solver can be efficiently executed in heterogeneous computing infrastructures and exhibits a remarkable speed-up when GPUs are employed. Thanks to the modular structure of the code and the use of a directive-based strategy to offload code execution on GPUs, only minor code modifications are required when targeting different computing architectures. This improves code maintenance, version control and the implementation of additional modules or governing equations.

1. Introduction

Turbulent multiphase flows are ubiquitous in nature and our everyday life. These flows play a key-role in various applications, from geophysical phenomena [25, 47], to environmental and industrial processes [45, 58, 59]. With respect to single-phase turbulence, the description and modeling of multiphase turbulence is much more complex, since these type of flows require the description of an ever-moving and deforming interface, its topological modifications, and the underlying turbulent flow [15, 51, 54, 63]. Simulations constitute an essential tool to investigate the physics

* Corresponding author. Tel.: +39-0432-558006.
E-mail address: alessio.roccon@uniud.it

arXiv:2406.01322v1 [physics.flu-dyn] 3 Jun 2024

of multiphase turbulence and are becoming increasingly popular in recent years: numerical simulations give access to detailed space- and time-resolved information on the flow field and on the morphology of the two phases.

Obtaining an accurate description of the dynamics of a turbulent multiphase flow on a discretized temporal and spatial grid is however a challenging task because of the large scale separation that characterizes these flows: scales range from the largest flow scale (of the order of the domain size), down to the Kolmogorov scale of turbulence and further down to the molecular scale of the interface. This has direct implications on the description of multiphase turbulence as the spatio-temporal resolution one can reasonably afford is limited by computing capabilities [20]. Specifically, as done for single-phase turbulence [27, 41, 56], it would be beneficial to perform simulations in which all scales are directly resolved, without any model. However, this approach cannot be applied to multiphase flows, since the scale separation between the largest flow scale and the smallest interfacial scale is about eight to nine orders of magnitude, while the most recent high-performance computing (HPC) infrastructures, can handle a maximum scale separation of about three to four orders of magnitude.

In the last 15 years, the use of graphics processor units (GPU) has played a key role in the development of modern HPC systems [38, 42] paving the way for big advances in many different fields [13, 16, 44, 50]. In computational fluid dynamics, a large amount of computing resources are usually required for complex flow problems, especially when direct numerical simulations of multiphase turbulence are involved. Thanks to the integration of GPUs in HPC centers, there is now the possibility to solve large-scale problems [2, 5, 8, 12, 49, 68]. However, with respect to code development for CPUs, programming for GPUs poses new challenges: i) CPUs and GPUs are characterized by different computing architectures; ii) Specific programming languages/models and compilers are required to generate code that can be executed on GPUs; iii) Performance, scaling and code portability across different vendors and infrastructures represent an open issue; iv) CPU and GPU memories are physically separated and efficient strategy for the memory management should be designed [18]. All these aspects render the development and optimization of algorithms on GPU-accelerated infrastructures an open challenge.

In this context, different approaches are available and they are characterized by different degrees of portability, adaptability, and performance. The most commonly used are: i) Compute Unified Device Architecture (CUDA) [40], an extension of Fortran and C/++ programming languages to offload computations on GPUs and to manage memory transfers; ii) Open Computing Language (OpenCL) [65], an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators; iii) OpenMP [10] and OpenACC [21], directives-based models that use directives to define data movement and computation offloading; iv) Library-based solution (e.g. Kokkos [14]), which offers high-level abstract programming concepts for performance portability on different types of high-performance computing infrastructures; v) Standard language parallelism, a parallel programming model that allows for accelerating C++ and Fortran codes without using language extensions or additional libraries. Each of these approaches is characterized by its own advantages and disadvantages from the point of view of the performance, portability, maintenance, short- and long-term programming language compatibility and maintenance, compiler requirements, and future developments that one should carefully evaluate before starting the GPU-porting phase of a code. With the idea of having a single code that can be executed on different architectures, we choose here to follow the directive-based approach offered by OpenACC. Among the available approaches, this approach has the following advantages: i) A single code has to be maintained: OpenACC directives will be ignored when GPU support is not enabled; ii) In the framework of the Nvidia HPC Software Development Kit (SDK) it allows for the use of the managed memory feature, which greatly simplifies data transfer between CPU and GPU memories; iii) Additional features, like the solution of new governing equations, can be easily implemented as the development time required to port to GPUs new code sections is reduced (compared to the other available approaches).

In this work, the OpenACC programming model is applied to a code for direct numerical simulation of multiphase turbulence based on a pseudo-spectral method [23]. In particular, the code relies on a direct solution of the Navier-Stokes equations coupled with a phase-field method to describe the interface shape and its topological changes. The parallelization is based on a multilevel approach (MPI + X). A first level relies on MPI and FFTW libraries [**?**] and it is employed when the code is executed in CPU-based computing infrastructures. A second level of parallelism that employs OpenACC directives and cuFFT libraries [39] is used to accelerate the code execution when GPU-based infrastructures are targeted. Thanks to this combination of OpenACC directives and cuFFT libraries, all the most computationally intensive sections of the code can be offloaded to GPUs. Overall, when GPUs are used, this leads to a remarkable speed-up and thus a reduction of the wall-clock time required for a time step (with respect to when CPUs are used).

The paper is organized as follows. In Section 2, the governing equations are presented; in Section 3 the numerical method is detailed. Then, in Section 4, the GPU-porting and the code performance are detailed and a demo simulation is presented. Finally, conclusions are drawn in Section 5.

2. Methodology

2.1. Governing equations

To describe the dynamics of the system, direct numerical simulation (DNS) of the Navier-Stokes equations, used to describe the flow field, are coupled with a phase-field method (PFM), used to describe interfacial phenomena. These equations are solved in a plane channel geometry: the streamwise and spanwise directions are periodic while along the wall-normal directions no-slip or free-slip, or a combination of them, can be applied.

2.1.1. Phase-field method

The phase-field method (PFM) is an interface-capturing method used for the description of multiphase flows. The method is based on the introduction of a marker function – the phase-field variable ϕ – that is uniform in the bulk of the phases ($\phi = \pm 1$) while it varies smoothly over the thin transition layer that separates the two phases [3, 24, 54]. The time evolution of the phase-field variable is here described by the Cahn-Hilliard (CH) equation which, in dimensionless form, reads as:

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = \frac{1}{Pe} \nabla^2 \mu \,, \tag{1}$$

where $\mathbf{u} = (u, v, w)$ is the velocity vector, μ is the chemical potential. The Péclet number, Pe, represents the ratio between the diffusive timescale, $h^2/\mathcal{M}\beta^2$, and the convective time scale, h/u_τ:

$$Pe = \frac{u_\tau h}{\mathcal{M}\beta} \,, \tag{2}$$

where $u_\tau = \sqrt{\tau_w/\rho}$ is the friction velocity (with τ_w the wall shear-stress and ρ the density), h is the half-channel height, $\mathcal{M}$ is the mobility parameter, and β is a positive constant introduced in the dimensionless procedure.

The chemical potential, μ, is obtained as the variational derivative of the Ginzburg-Landau free-energy functional [4, 24, 60, 66]. To model the case of two immiscible fluids, the free-energy functional, $\mathcal{F}[\phi, \nabla\phi]$, is composed by the sum of two different contributions [4, 24]:

$$\mathcal{F}[\phi, \nabla\phi] = \int_\Omega \left(f_0 + f_{mix} \right) d\Omega = \int_\Omega \underbrace{\frac{1}{4}(\phi^2 - 1)^2}_{f_0} + \underbrace{\frac{Ch^2}{2}|\nabla\phi|^2}_{f_{mix}}, \tag{3}$$

The first term, f_0 (bulk energy), identifies the tendency for the multiphase system to separate into two pure stable phases, while the second contribution, f_{mix} (mixing energy), is a non-local term accounting for the energy stored at the interface (i.e. surface tension). The Cahn number, $Ch = \epsilon/h$, represents the dimensionless thickness of the interfacial layer separating the two phases. By taking the functional derivative of the Ginzburg-Landau free-energy functional, we obtain the expression of the chemical potential:

$$\mu = \frac{\delta \mathcal{F}[\phi, \nabla\phi]}{\delta \phi} = \phi^3 - \phi - Ch^2 \nabla^2 \phi \,. \tag{4}$$

At equilibrium, considering that the chemical potential is constant throughout the domain ($\nabla\mu = \mathbf{0}$), the following (equilibrium) profile is obtained for a planar interface:

$$\phi_{eq} = \tanh\left(\frac{s}{\sqrt{2}Ch}\right), \tag{5}$$

where s is the coordinate normal to the interface (located at $s = 0$).

3

2.1.2. Hydrodynamics

To describe the flow-field of the multiphase system, the Cahn-Hilliard equation (1) is coupled with the Navier-Stokes equations [15, 51, 63]. We consider here two incompressible and Newtonian phases and, for the sake of simplicity, we assume that they are characterized by equal density, ρ, and viscosity, μ. With these assumptions, the Navier-Stokes equations can be written as:

$$\nabla \cdot \mathbf{u} = 0 \,, \tag{6}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \frac{1}{Re_\tau} \nabla^2 \mathbf{u} + \mathbf{f}_\sigma \,, \tag{7}$$

where p is the pressure-field and $\mathbf{f}_\sigma$ represents the surface tension forces. These forces are here computed using a continuum-surface stress approach [22, 30] :

$$\mathbf{f}_\sigma = \frac{3}{\sqrt{8}} \frac{Ch}{We} \nabla \cdot [\bar{\tau}_c] \,, \tag{8}$$

where $\bar{\tau}_c = |\nabla \phi|^2 \mathbf{I} - \nabla \phi \otimes \nabla \phi$ is the Korteweg tensor used to model surface tension forces [28].

The dimensionless numbers that appear in the Navier-Stokes equation are the friction Reynolds number:

$$Re_\tau = \frac{\rho u_\tau h}{\eta} \,, \tag{9}$$

which represents the ratio between the inertial and viscous forces, and the Weber number:

$$We = \frac{\rho u_\tau^2 h}{\sigma} \,, \tag{10}$$

which represents the ratio between inertial and surface tension forces (being σ the surface tension).

2.2. Numerical method

The Navier-Stokes and continuity equations are not solved in primitive variables (velocity and pressure) but are recast in the so-called wall-normal velocity-vorticity formulation. This avoids solving a Poisson equation for pressure. Specifically, Navier-Stokes and continuity equations are replaced by a set of four equations [27, 62, 64]: i) A second-order equation for the wall-normal component of the vorticity; ii) A fourth-order equation for the wall-normal component of the velocity vector; iii) Continuity equation; iv) Definition of wall-normal vorticity. To obtain the set of governing equations for the wall-normal velocity-vorticity formulation, we can first rewrite the Navier-Stokes equations as follows:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{S} - \nabla p + \frac{1}{Re_\tau} \nabla^2 \mathbf{u} \,, \tag{11}$$

where the term $\mathbf{S}$ contains all the non-linear terms present in the Navier-Stokes equations (e.g. convective terms, surface tension forces, etc.). By taking the curl of the Navier-Stokes equations, the pressure term vanishes thanks to the identity $\nabla \times \nabla p = 0$ and a transport equation for the vorticity vector, ω, is obtained:

$$\frac{\partial \omega}{\partial t} = \nabla \times \mathbf{S} + \frac{1}{Re_\tau} \nabla^2 \omega \,, \tag{12}$$

By taking again the curl of the vorticity transport equation, we obtain the following 4-*th* order equation for the velocity:

$$\frac{\partial \nabla^2 \mathbf{u}}{\partial t} = \nabla^2 \mathbf{S} - \nabla(\nabla \cdot \mathbf{S}) + \frac{1}{Re_\tau} \nabla^4 \mathbf{u} \,, \tag{13}$$

We solve here for the wall-normal components of the vorticity ω_z and velocity w. Hence, the following governing equations are solved:

$$\frac{\partial \omega_z}{\partial t} = \frac{\partial S_y}{\partial x} - \frac{\partial S_x}{\partial y} + \frac{1}{Re_\tau} \nabla^2 \omega_z \,, \tag{14}$$

$$\frac{\partial(\nabla^2 w)}{\partial t} = \nabla^2 S_z - \frac{\partial}{\partial z}\left(\frac{\partial S_x}{\partial x} + \frac{\partial S_y}{\partial y} + \frac{\partial S_z}{\partial z}\right) + \frac{1}{Re_\tau}\nabla^4 w, \tag{15}$$

complemented by the continuity equation (6) and the definition of wall-normal vorticity:

$$\omega_z = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}. \tag{16}$$

The Cahn-Hilliard equation, which is a fourth-order equation, is split into two second-order equations [4]. In particular, the CH equation is first rewritten in the following way:

$$\frac{\partial \phi}{\partial t} = S_\phi + \frac{sCh^2}{Pe_\phi}\nabla^2\phi - \frac{Ch^2}{Pe_\phi}\nabla^4\phi, \tag{17}$$

where S_ϕ represents the contribution of the non-linear terms. The operator splitting $\nabla^2\phi = \nabla^2\phi(sCh^2 + 1) - sCh^2\nabla^2\phi$ is then applied [4] where the positive coefficient s has been chosen considering the temporal discretization.

The governing equations (6)-(14)-(15)-(16)-(17) are solved using a pseudo-spectral method: the variables are transformed from the physical into the wavenumber space. Along the periodic directions (x and y), all the quantities are expressed by Fourier series while they are represented by Chebyshev polynomials along the non-homogeneous wall-normal direction. The corresponding $N_x \times N_y \times N_z$ collocation points are equally spaced along the x and y directions while they are stretched along the wall-normal direction where a finer grid resolution is obtained near the two walls. Thanks to the adoption of a pseudo-spectral discretization, the equations are decoupled along the two periodic directions. In this way, a series of 1D Helmholtz independent problems along the wall-normal directions are obtained. All calculations are carried out in the wavenumber space except the non-linear terms, which are first computed in the physical space and then transformed back to spectral space (pseudo-spectral method). This avoids the evaluation of (computationally expensive) convolutions in the wavenumber space [23, 48]. The governing equations are discretized in time using an IMplicit-EXplicit (IMEX) scheme, in which the non-linear terms are integrated with an Adams-Bashfort scheme, while the linear terms are integrated by a Crank-Nicolson (Navier-Stokes) or by an implicit Euler (Cahn-Hilliard) scheme.

To advance the velocity and phase-field variables from time step n to time step $n + 1$, the following intermediate steps are performed (hat notation is used to identify the variable representation in the wavenumber space).

i) The velocity field, $\mathbf{u}^n$ and the phase-field, ϕ^n, are initialized and variables are transformed in the spectral space, $\hat{\mathbf{u}}^n$ and $\hat{\phi}^n$ (where hat notation is used in the following to identify the spectral representation of the variables).

ii) The non-linear convective and surface tension terms, i.e. the term $\mathbf{S}$, are computed. As these terms are non-linear, they are computed in the physical space rather than in the spectral space via convolution operation.

iv) Continuity and Navier-Stokes equations are solved (using the wall-normal velocity-vorticity formulation) to obtain the new velocity field, $\hat{\mathbf{u}}^{n+1}$. This requires the solution of a system of Helmholtz problems along the wall-normal direction for each (x, y) location.

vi) The non-linear terms present in the Cahn-Hilliard equation are computed, i.e. the term S_ϕ is formed. Also in this case, the non-linear terms are computed in the physical space.

v) The Cahn-Hilliard equation is solved and the new value of the phase-field, $\hat{\phi}^{n+1}$, is obtained. Similar to the solution of the Navier-Stokes equations, this requires the solution of two Helmholtz problems along the wall-normal direction for each (x, y) location.

3. Implementation

This numerical scheme has been implemented in a parallel Fortran 2003 MPI in-house proprietary code (FLOW36). The code is written using Fortran 2003 and the main parallelization backbone relies on an MPI paradigm. On top of the MPI backbone, OpenACC directives and cuFFT libraries are used to accelerate the code execution on GPU-based computing infrastructures. The code has been largely used in the past for the investigation of a wide range of complex turbulent flows: particle-laden flows [36], stratified turbulent flows [69], interface-resolved simulations of drop- and bubble-laden flows [61, 34, 54], drag reduced flows [55]. The code is developed using a modular approach: conditional compilation directives are used to enable (or disable) the different governing equations/physics available.
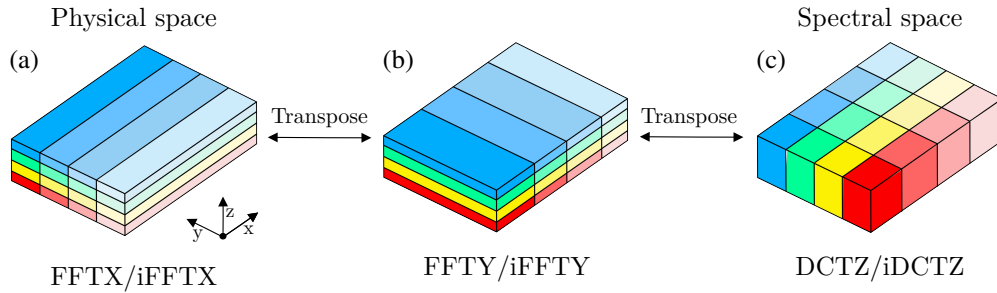
Figure 1. 2D domain decomposition employed for the first level of parallelization (MPI). Each color corresponds to a different MPI task. In physical space, the domain is divided along the y and z directions (pencils orientated along the x-direction), while in spectral space it is divided along the x and y directions (pencil orientated along the z-direction). Transpositions (i.e. loops of MPI communications) are required to change the pencil orientation and to compute the transform along the various directions: along x in the first configuration shown in panel a, along y in the configuration shown in panel b, and along z in the configuration shown in panel c.

3.1. First level of parallelization: MPI

The parallelization backbone of the code relies on an MPI approach; the overall workload is divided among the different MPI tasks using a 2D domain decomposition (pencil decomposition). Within this strategy, the whole domain is split in so-called pencils: the domain is divided along two out of three directions and each sub-domain is assigned to a different MPI process. In physical space, the domain is divided along the y and z directions (pencils oriented along the x-direction), while in modal space it is divided along the x and y directions (pencils oriented along the z-direction). This change in the pencil orientation is needed when taking the transforms: to compute the Fourier or Chebyshev transforms each process must hold all the points in the transform direction. Thus, when in physical space, at first Fourier transforms are taken in the x direction (Figure 1a), then the parallelization changes in order to have all the points along the y direction. The domain is divided between the x and z directions when taking the Fourier transforms along y (Figure 1b). At this point, each $x - y$ plane holds all the Fourier modes at a certain height. Then the parallelization is again changed, switching to a domain decomposition along x and y directions, thus each MPI process holds all the points in z direction at a certain (x, y) location (parallelization in modal space). Finally, Chebyshev transforms are taken in the z direction (Figure 1c).

The only MPI communications occur when the non-linear terms of the governing equations are computed. These terms are computed in the physical space and thus a change of pencil orientation is required to compute the transforms along the three directions (from spectral to physical and vice-versa). After the calculation of all the non-linear terms, a system of Helmholtz problems along the wall-normal direction (z) is solved at each (x, y) location independently.

3.2. Second level of parallelization: OpenACC directives

On top of the MPI parallelization scheme presented above, cuFFT libraries and OpenACC directives are used to accelerate the code execution. When CPU-based architectures are employed, each MPI task is assigned to a physical core. Differently, when GPU-based architectures are used, each MPI task is assigned to a specific GPU. All the computationally intensive operations are performed on the GPUs. Specifically, the cuFFT libraries are used to perform all the transforms (Fourier and Chebyshev) and the entire solver can be efficiently executed on GPUs thanks to the fine-grain parallelism offered by the numerical scheme (series of 1D independent problems along the wall-normal direction). To obtain a code that can be easily ported between CPU- and GPU-based architectures, we employ the managed memory model present in OpenACC (which exploits the CUDA unified memory feature). As in most systems CPU and GPU memories are physically separated, the use of GPUs usually requires explicit memory transfers and the handling of the copies shared between CPU and GPU memories. Differently, using the managed memory feature, there is no need to explicitly define each memory transfer between the CPU (host) and the GPU (device). In particular, memory can be accessed using a single pointer from CPU or GPU code. All memory transfers are managed by the compiler that makes the decisions about when and how to move data from the host to the device and back. The use of

the managed memory model should be specified via the `-gpu=managed` compiling option. Although its use greatly simplifies code optimization, GPU-porting, and implementation of new features, one should bear in mind the three main drawbacks of this approach: i) The actual bandwidth of the memory transfer between CPU and GPU is slightly reduced; this is due to the virtual memory page faulting and migration used by the compiler to migrate data; ii) Data cannot be transferred asynchronously (note that asynchronous compute regions can be created); iii) This feature requires the use of Nvidia compiler and is at the moment non-compatible with GPUs from different vendors.

3.2.1. Fourier and Chebyshev transforms: cuFFT libraries

To accelerate the execution of Fourier and Chebyshev transforms, these operations are performed using the highly optimized and scalable cuFFT libraries. This CUDA-based library is optimized for Nvidia GPUs and offers an interface equivalent to that employed by the FFTW library (for CPUs) [17]. In FLOW36, conditional compilation flags are used to compile the code with the correct library (FFTW when using CPUs and cuFFT when using GPUs). To perform a transform, three main steps are required: i) Creation of a plan; this operation set-ups the library and identifies the best algorithm to be used depending on the available resources; ii) Execution of the transform according to the pre-defined plan; iii) Destruction of the plan. Plans can be also stored so that All transforms are executed using the batched version of the library (specified by the plan): this means that the library is set to perform a certain number of transforms along one direction and the input data is coherent with the advanced data layout structure. For instance, considering the forward transform along the x direction, $N_y \times N_z$ transforms of a signal having length N_x are performed. The transform can be then executed as follows by invoking the cuFFT library via OpenACC. For instance, considering the forward transform along x, this operation can be performed as follows:

```
!$acc data copyin(input) copyout(output)
!$acc host_data use_device(input,output)
gerr=gerr+cufftExecD2Z(plan,input,output)
!$acc end host_data
!$acc end data
```

where data clauses are required to force the data movement between CPU and GPU memories when libraries are used. The suffix `D2Z` defines the type of transform: as here we consider the forward transform along x, the input is an array of double (D) while the output is an array of complex numbers (Z), i.e. a real-to-complex transform. The above-mentioned strategy can be straightforwardly applied to the Fourier transform along x and y, which correspond to the first and third index of a generic 3D input array.

A different approach is required to perform the Chebyshev transforms along the wall-normal direction z. Two minor issues arise here: i) The cuFFT libraries do not explicitly support real-to-real transforms; ii) We need fo perform a transform along the direction specified by the inner index of a matrix and this does not allow for the use of the batched mode of cuFFT. These two issues can be easily overcome by manipulating the input and output vector so that a Chebyshev transform is performed and by transposing (before and after the transform) the input vector so that it satisfies the advanced data layout. In particular, the Chebyshev transform can be performed using FFT algorithms as it belongs to the group of discrete sine/cosine transforms (specifically, a DCT-I). The Chebyshev transform of an N-point real signal can be obtained by taking the discrete Fourier transform of a 2N-point even extension of the signal [32]. This scheme is applied to both the real and imaginary parts of the input signal and for both forward and backward transforms (the backward transform of a DCT-I transform is a DCT-I [57]). Note indeed that the input is a complex array because the FFTs along the x and y directions have been already performed at this stage, see figure 1c. Although the present strategy involves two transposes (back-and-forth) for each transform, the possibility to perform the Chebyshev transform in batched mode outperforms the other possible approach, i.e. a recursive loop of single 1D transforms, by one order of magnitude in terms of performance. Further optimization of the Chebyshev transform step can be obtained using the strategy proposed by Makhoul [32], which does not require performing a transform of size $2N$ but only of size N by re-ordering the input signal.

3.2.2. Solver execution: OpenACC directives

The computation of the non-linear terms in physical space, which mainly requires matrix multiplications, has been offloaded to GPUs by means of OpenACC directives. In particular, considering the computation of the non-linear terms present in both the Navier-Stokes and Cahn-Hilliard equations, this operation can be straightforwardly

parallelized on GPUs as it mainly involves element-wise product (Hadamard product). Indeed, all the flow-field and phase-field variables are collocated and thus defined on the same grid points. For instance, the non-linear term $a = bc$ (where a, b, c are 3D arrays) can be computed as follows:

```
!$acc kernels
do i=1,nx
 do j=1,nyp
  do k=1,nzp
   a(i,k,j)=b(i,k,j)*c(i,k,j)
  end do
 end do
end do
!$acc end kernels
```

where `nx, nyp` and `nzp` are the pencil dimensions in physical space. For this part of the code, the use of the `kernels` construct or `parallel loop` construct exhibit very similar performance. In addition, simple assignment operations (e.g. during time integration) are also performed using the `kernels` construct; this avoids expensive back-and-forth copies from CPU to GPU memories and vice-versa, which are limited by the available bandwidth between host and device.

Finally, the solution of the systems of Helmholtz problems is also offloaded to GPUs. This step requires solving a series of tridiagonal systems using Gauss elimination along the wall-normal direction for each (x, y) position. Thus, the only direction along which dependencies are present is the wall-normal direction while it can be parallelized along the x and y directions of the pencil. Thanks to the adoption of the managed memory feature, the offloading to GPUs of this part of the code is rather straightforward and does not pose particular challenges or require extensive code modifications with respect to the CPU version.

3.2.3. MPI communications

Once all the transforms and solver execution have been ported to the GPUs, as illustrated above, to improve code scalability, MPI communications can be also optimized. Specifically, in traditional MPI installations (Figure 2), considering a message between rank 0 and rank 1, the data computed on GPU 0 has to be staged into the host memory of CPU 0 (green and blue arrows) and then sent to the MPI process 0 via MPI message and, in turn, to the GPU 1 (assigned to rank 1), as shown in Figure 2. These additional passages represent a big overhead and can negatively impact strong and weak scalability results, especially when large-scale simulations are performed (i.e. using more than 100 GPUs) and many non-linear terms have to be computed (which require pencil transpositions and thus loop of MPI communications). A possible solution to this problem is the use of CUDA-aware MPI implementations (or more in general GPU-aware).

To address this problem, we take advantage of the CUDA-aware MPI. CUDA-aware MPI implementations allow for the use of GPUDirect, a group of technologies that provide high-bandwidth, low-latency communications for all kinds of inter-rank communication (intra-node, inter-node, and inter-node) via Remote Direct Memory Access (RDMA) when Nvidia GPUs are used. In particular, when GPUDirect RDMA is available the GPU buffer can be directly moved to the network without passing through the host memory at all. So the data is transferred from the GPU buffer of the MPI Rank 0 to the GPU buffer of MPI Rank 1 (as represented by the long solid arrow in the upper part of the figure). The use of CUDA-aware MPI features requires however the use of pinned buffers. As here data movement is implicitly handled via the use of managed memory, to exploit CUDA-aware features, CUDA Fortran instructions are required to define the buffers used to perform the communications as pinned (thus inhibiting the memory paging performed by the managed memory feature). This can be specified via the `pinned` attribute in the variable declarations and enabling the support for CUDA among the compiling options. Then, via OpenACC directives, the use of CUDA-aware features can be specified as follows:

```
!$acc data copy(bufs,bufr)
!$acc host_data use_device(bufs,bufr)
call mpi_isend(bufs,.....)
call mpi_irecv(bufr,......)
```
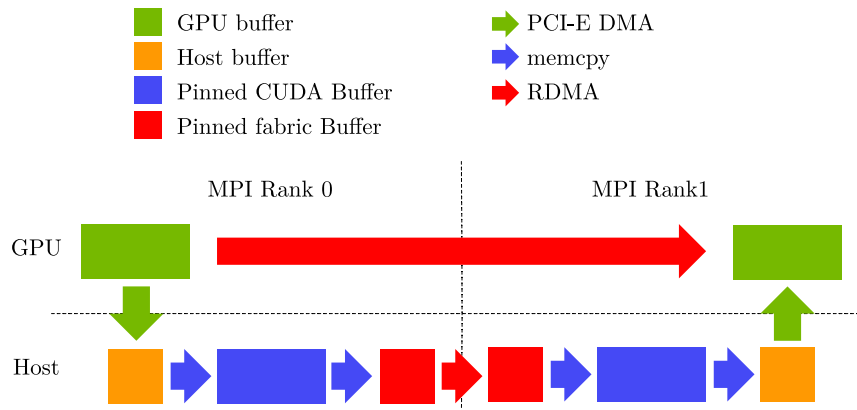
Figure 2. Schematic showing the step required to perform an MPI communication between data stored in the buffer of GPU 0 and GPU 1. In standard MPI installations, the data computed on GPU 0 has to be moved into host memory of CPU 0 and then sent to the MPI process 0 and, in turn, transferred to the GPU 1. Using a CUDA-aware MPI implementation that exploit GPUDirect technologies, e.g. Remote Direct Memory Access (RDMA), a direct transfer between GPU 0 and GPU 0 can be performed (long red arrow). Reproduced from [29].

```
call mpi_waitall(.....)
!$acc end host_data
!$acc end data
```

where `bufr` and `bufs` are the two GPU pinned buffers used for the MPI communications. It is convenient to allocate these buffers during code start-up as the allocation/deallocation of pinned buffers has a non-negligible overhead. Also, it is worth mentioning that the use of non-blocking MPI communications is advisable as it allows the MPI library to build more efficient pipelines. Finally, as the code relies on the managed memory feature (i.e. CUDA Unified memory), the chosen MPI implementation should support the CUDA unified memory, e.g. OpenMPI and MVAPICH-2 [19, 43]. Indeed, the MPI implementation needs to know where the data is located and which is the optimal strategy to perform the communication [35].

3.3. Optimization, performance and scaling

Thanks to the use of OpenACC directives, the code has been ported to GPU using a step-by-step approach. First, all the Fourier and Chebyshev transforms, which represent the most computationally intensive part of the algorithm and then the solver and the CUDA-aware MPI communications have been ported to GPUs. During each of these steps, the code has been optimized and profiled using the tools provided by the NVIDIA Nsight Systems and using the NVIDIA Tools Extension (NVTX) instructions. Specifically, NVTX instructions have been used to annotate the profiler timeline with events and ranges. This procedure allows to find, and possibly remove, performance bottlenecks as well as to improve specific sections of the code.

To evaluate the performance of the GPU-porting, we first consider a single-node run where all the GPUs present in the node are used. This allows us to obtain a first estimate of the gain that can be obtained by using GPUs, compared to using all the physical cores available on the CPU counterpart. It is also possible to compare the results keeping fixed the number of MPI tasks used; however, this would underestimate CPU performance as only a part of the available cores will be used: the number of GPUs present in a computing node (from 4 to 8) is usually smaller than the number of cores available (from 16 up to 256). For this test case, we consider a single-phase turbulent channel flow solved on a grid having $N_x \times N_y \times N_z = 256 \times 256 \times 257$ grid points. Tests have been performed on four different machines, one local server, two machines present at CINECA (Marconi-100 and Leonardo [67]) and the CPU partition of LUMI (LUMI-C). This latter machine is included as a reference of the performance that can be obtained in a CPU-based computing infrastructure. The technical specifications of the four machines are reported in Table 1. When CPUs are used, the maximum number of cores available is used whereas when GPUs are employed, the number of MPI tasks
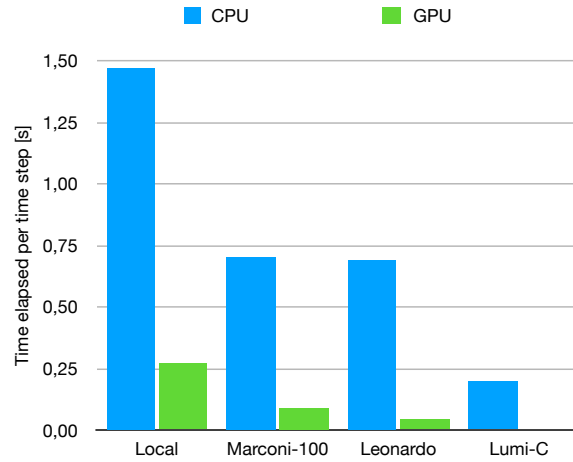
Figure 3. Time elapsed per time step on different machines using all the physical cores available (blue) and all the GPUs available (green) on a single-node. The results have been obtained considering a single-phase turbulent channel flow and a grid resolution equal to $N_x \times N_y \times N_z = 256 \times 256 \times 257$. For all cases, the code has been compiled using the Nvidia Fortran compiler `nvfortan` with or without the support for GPU-acceleration, depending on the case considered (CPU or GPU). When the `nvfortan` compiler is not available (LUMI-C), the code has been compiled using `gfortran`.

is set equal to the number of GPUs. For all tests, the code is compiled with the Nvidia Fortran compiler `nvfortran` when available and when not with the GNU compiler `gfortran`.

Table 1. Technical details of the computing infrastructures employed in the tests

System	CPU	GPU
Local Server	1 x Intel Xeon 5218 16 cores	2 x Quadro RTX 5000 16 GB
Marconi-100 (CINECA)	2 x IBM POWER9 AC922 16 cores	4 x NVIDIA Volta V100 16 GB
Leonardo (CINECA)	1 x Intel Xeon 8358 32 cores	4 x Nvidia Ampere A100 64GB
LUMI-C (LUMI)	2 x AMD EPYC 7763 64 cores	-

The time elapsed for a single time step is shown in Figure 3 for the different machines and cases considered. Results obtained from the CPU runs are reported in blue while those obtained from GPU runs are in green. In general, it can be observed that when the full node performance is compared, i.e. all physical cores versus all available GPUs, using the GPUs a consistent speed-up is obtained. The specific value of the speed-up obtained depends on the machine considered: generally speaking, this value ranges from 4 (local server) up to 16 (Leonardo). This variation is due to the different performance offered by the CPUs and GPUs present in the computing node. Comparing the results with the CPU runs performed on LUMI-C, it is interesting to observe that the wall-clock time elapsed per time step obtained using GPUs is smaller.

We now move to analyze the strong scaling results. In particular, Figure 4 shows the results obtained employing Marconi-100, please refer to Table 1 for the technical details of the cluster. We consider two different grid resolutions, typical of production runs for direct numerical simulations of multiphase turbulence: $N_x \times N_y \times N_z = 512 \times 512 \times 513$ and $N_x \times N_y \times N_z = 1024 \times 1024 \times 1025$. For the coarser grid resolution (blue dots), tests have been performed starting from 1 node (4 GPUs) up to 32 nodes (128 GPUs) while, for the finer grid resolution (red dots), starting from 8 nodes (32 GPUs) up to 64 nodes (256 GPUs). A different number of nodes has been used for the two problem sizes because of the different memory requirements. We can observe that in general good results are obtained for both the problem sizes considered. For the smaller problem size, strong scaling results start to worsen when more than 16 nodes (64 GPUs) or more are used. This is due to the lower computational load per pencil/slab and the relative cost of MPI communications. Indeed, by increasing the problem size (i.e. considering the finer grid resolution), the computational load per pencil/slab increases and the cost of the MPI communications can be hidden; as a consequence, better results
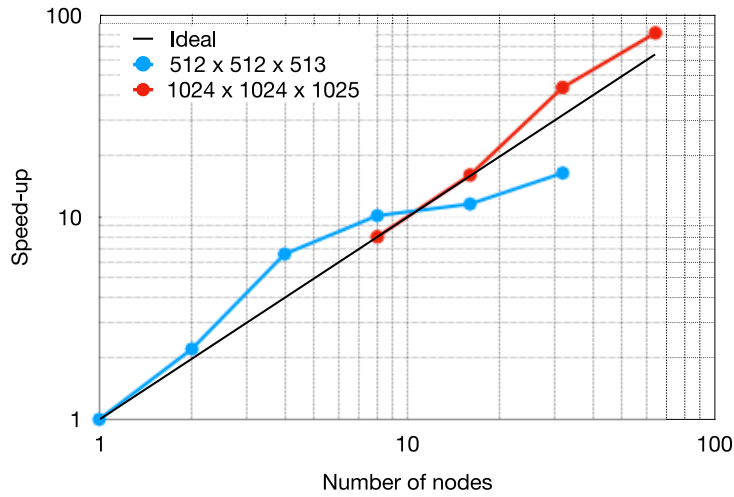
Figure 4. Strong scaling results for the code FLOW36 obtained on Marconi100. Two different problems sizes have been considered: $N_x \times N_y \times N_z = 512 \times 512 \times 513$ and $N_x \times N_y \times N_z = 1024 \times 1024 \times 1025$. For the smaller problem size (blue dots), tests have been performed starting from 1 node (4 GPUs) up to 32 nodes (128 GPUs) while, for the larger problem size (red dots), starting from 8 nodes (32 GPUs) up to 64 nodes (256 GPUs).

in terms of scaling are obtained. Future developments are required to improve code scalability on a larger number of GPUs as performing large-scale 3D FFT require a significative amount of communications [6, 9, 11, 46, 52].

4. Code capabilities

We provide here an example of the code capabilities. In particular, we consider the injection of a swarm of large and deformable drops released in a turbulent channel flow, a flow configuration similar to the one adopted in previous works [33, 53, 60]. The computational domain is a closed channel with dimensions $L_x \times L_y \times L_z = 4\pi h \times 2\pi h \times 2h$ corresponding to $L_x^+ \times L_y^+ \times L_z^+ = 3770 \times 1885 \times 600$ wall units. Equations are discretized on a grid with $N_x \times N_y \times N_z = 2048 \times 1024 \times 1025$ collocation points. The simulation is performed at a fixed shear Reynolds number of $Re_\tau = 300$ and the Weber number has been set equal to $We = 6$. The simulation starts by releasing 256 spherical droplets in a flow field previously obtained from a single-phase flow simulation of a fully developed turbulent channel flow. After an initial transient, where drops start to break and coalesce following complex dynamics, a new equilibrium situation is reached in which a balance between coalescence and breakage events is attained. Figure 5 shows a qualitative rendering of the steady-state configuration attained by the system at $t^+ = 3000$. The flow moves from left to right (along the streamwise direction x) and a top view of the system is shown (streamwise-horizontal; spanwise-vertical). The interface of the drops is identified as the iso-contour $\phi = 0$ and is rendered using a ray tracing algorithm. We can observe the wide range of scales and shapes that characterize the drops: from very small and almost undeformed spherical droplets to very large drops characterized by a complex three-dimensional shape. It is also worth noticing the presence of elongated filaments/threads that will eventually lead to the formation of small drops.

5. Conclusions and future developments

We detailed the GPU-porting of a pseudo-spectral code tailored towards large-scale simulations of multiphase flow, more specifically interface-resolved direct numerical simulations. The code relies on direct numerical simulation of turbulence coupled with a phase-field method to describe the interface shape, deformation, and topological changes. The development of the code has been designed with the specific goal of simplifying code maintenance, obtaining a portable code that can exploit GPU acceleration, and facilitating the implementation of new modules and new physics (e.g. additional governing equations). To achieve this ambitious goal, we rely on two levels of parallelism: i) A first level that relies on MPI and a 2D domain decomposition to divide the workload among the MPI tasks,; ii) A second
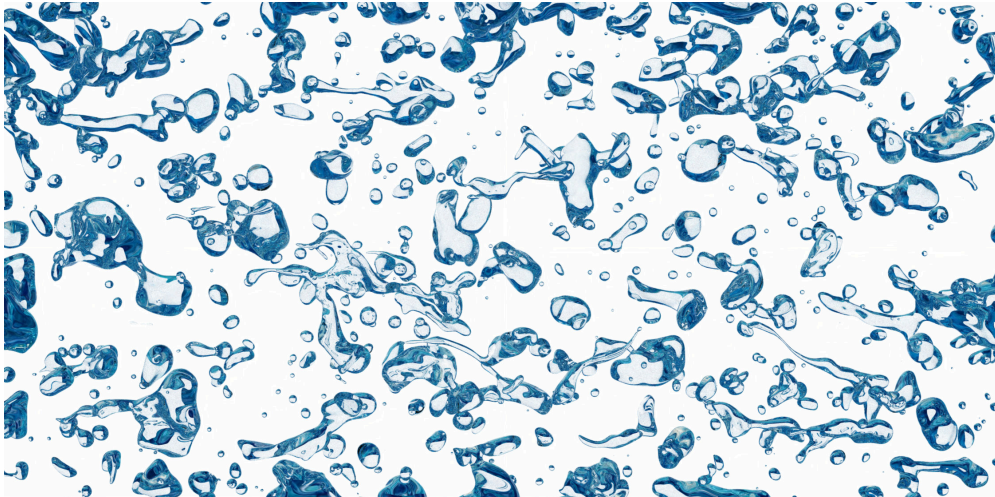
Figure 5. Top view of a swarm of large and deformable drops released in a turbulent channel flow. The flow moves from left to right (along the streamwise direction) and drops coalesce and break under the action of turbulence fluctuations. The interface of the drops is identified as the iso-contour $\phi = 0$. The grid resolution employed for this demo simulation is $N_x = 2048 \times 1024 \times 1025$. This run has been executed using 64 nodes (256 GPUs) of Leonardo [67].

level that relies on OpenACC directives and CUDA libraries (cuFFT) to accelerate code execution on GPU-based computing infrastructures. The code makes use of the managed feature (CUDA Unified memory) to simplify code maintenance and avoid the explicit definition of data transfers between CPU and GPU memories [31, 37] In addition, pinned memory buffers are used to perform MPI communications so that the GPU Direct technologies can be employed to improve latency and data transfer times. The main limitation of the employed approach is the compatibility with GPUs from other vendors as the managed memory feature is currently available only on the Nvidia HPC-SDK environment. However, a similar feature has been recently released in the context of AMD GPU architectures [7, 26]. Hence, in future developments, we plan to extend the support also to AMD GPUs by exploiting the unified memory feature present in the AMD ecosystem and combining it with openMP directives and rocFFT libraries to accelerate the code execution (solver and transforms) [1].

Acknowledgements

References

[1] AMD, . rocFFT library. URL: https://github.com/ROCmSoftwarePlatform/rocFFT.
[2] Ames, J., Puleri, D.F., Balogh, P., Gounley, J., Draeger, E.W., Randles, A., 2020. Multi-gpu immersed boundary method hemodynamics simulations. J. Comput. Sci. 44, 101153.
[3] Anderson, D., McFadden, G., Wheeler, A., 1998. Diffuse interface methods in fluid mechanics. Annu. Rev. Fluid Mech. 30, 139–165.
[4] Badalassi, V., Ceniceros, H., Banerjee, S., 2003. Computation of multiphase systems with phase field models. J. Comput. Phys. 190, 371–397.
[5] Bernardini, M., Modesti, D., Salvadore, F., Sathyanarayana, S., Della Posta, G., Pirozzoli, S., 2023. STREAmS-2.0: Supersonic turbulent accelerated navier-stokes solver version 2.0. Comput. Phys. Commun. 285, 108644.

[6] Chatterjee, A.G., Verma, M.K., Kumar, A., Samtaney, R., Hadri, B., Khurram, R., 2018. Scaling of a fast fourier transform and a pseudo-spectral fluid solver up to 196608 cores. J. Parallel. Distr. Com. 113, 77–91.

[7] Chu, H., 2013. AMD heterogeneous uniform memory access. Proceedings of the APU 13th developer summit , 11–13.

[8] Costa, P., Phillips, E., Brandt, L., Fatica, M., 2021. GPU acceleration of CaNS for massively-parallel direct numerical simulations of canonical fluid flows. Comput. Math. Appl. 81, 502–511.

[9] Czechowski, K., Battaglino, C., McClanahan, C., Iyer, K., Yeung, P.K., Vuduc, R., 2012. On the communication complexity of 3D FFTs and its implications for exascale, in: Proceedings of the 26th ACM international conference on Supercomputing, pp. 205–214.

[10] Dagum, L., Menon, R., 1998. Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE 5, 46–55.

[11] Dalcin, L., Mortensen, M., Keyes, D.E., 2019. Fast parallel multidimensional FFT using advanced MPI. J. Parallel. Distr. Com. 128, 137–150.

[12] De Vanna, F., Avanzi, F., Cogo, M., Sandrin, S., Bettencourt, M., Picano, F., Benini, E., 2023. URANOS: A GPU accelerated navier-stokes solver for compressible wall-bounded flows. Comput. Phys. Commun. 287, 108717.

[13] Dematté, L., Prandi, D., 2010. GPU computing for systems biology. Brief. Bioinform. 11, 323–333.

[14] Edwards, H.C., Trott, C., Sunderland, J., 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. 74, 3202–3216.

[15] Elghobashi, S., 2019. Direct numerical simulation of turbulent flows laden with droplets or bubbles. Annu. Rev. Fluid Mech. 51, 217–244.

[16] Friedrichs, M.S., Eastman, P., Vaidyanathan, V., Houston, M., Legrand, S., Beberg, A.L., Ensign, D.L., Bruns, C.M., Pande, V.S., 2009. Accelerating molecular dynamic simulation on graphics processing units. J. Comput. Chem. 30, 864–872.

[17] Frigo, M., Johnson, S.G., 2005. The design and implementation of FFTW3. Proc. IEEE 93, 216–231.

[18] Fujii, Y., Azumi, T., Nishio, N., Kato, S., Edahiro, M., 2013. Data transfer matters for GPU computing, in: 2013 ICPDS Conference on Parallel and Distributed Systems, IEEE. pp. 275–282.

[19] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al., 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11, Springer. pp. 97–104.

[20] Gorokhovski, M., Herrmann, M., 2008. Modeling primary atomization. Annu. Rev. Fluid Mech. 40, 343–366.

[21] working group, O., et al., 2011. The OpenACC application programming interface. Retrieved March 26, 2019.

[22] Gueyffier, D., Li, J., Nadim, A., Scardovelli, R., Zaleski, S., 1999. Volume-of-fluid interface tracking with smoothed surface stress methods for three-dimensional flows. J. Comput. Phys. 152, 423–456.

[23] Hussaini, M., Zang, T., 1987. Spectral methods in fluid dynamics. Annu. Rev. Fluid Mech. 19, 339–367.

[24] Jacqmin, D., 1999. Calculation of two-phase Navier–Stokes flows using phase-field modeling. J. Comput. Phys. 155, 96–127.

[25] Jähne, B., Haußecker, H., 1998. Air-water gas exchange. Annu. Rev. Fluid Mech. 30, 443–468.

[26] Jin, Z., Vetter, J.S., 2022. Evaluating unified memory performance in HIP, in: 2022 IEEE International parallel and distributed processing symposium workshops (IPDPSW), IEEE. pp. 562–568.

[27] Kim, J., Moin, P., Moser, R., 1987. Turbulence statistics in fully developed channel flow at low Reynolds number. J. Fluid Mech. 177, 133–166.

[28] Korteweg, D., 1901. Sur la forme que prennent les equations du mouvements des fluides si l'on tient compte des forces capillaires causées par des variations de densité considérables mais continues et sur la théorie de la capillarité dans l'hypothèse d'une variation continue de la densité. Archives Néerlandaises des Sciences Exactes et Naturelles 6, 1–24.

[29] Kraus, J., . An introduction to CUDA-Aware MPI. URL: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

[30] Lafaurie, B., Nardone, C., Scardovelli, R., Zaleski, S., Zanetti, G., 1994. Modelling merging and fragmentation in multiphase flows with SURFER. J. Comput. Phys. 113, 134–147.

[31] Lindholm, E., Nickolls, J., Oberman, S., Montrym, J., 2008. Nvidia tesla: A unified graphics and computing architecture. IEEE micro 28, 39–55.

[32] Makhoul, J., 1980. A fast cosine transform in one and two dimensions. IEEE Trans. Acoust. 28, 27–34.

[33] Mangani, F., Roccon, A., Zonta, F., Soldati, A., 2024. Heat transfer in drop-laden turbulence. J. Fluid Mech. 978, A12.

[34] Mangani, F., Soligo, G., Roccon, A., Soldati, A., 2022. Influence of density and viscosity on deformation, breakage, and coalescence of bubbles in turbulence. Phys. Rev. Fluids 7, 053601.

[35] Manian, K.V., Ammar, A., Ruhela, A., Chu, C.H., Subramoni, H., Panda, D.K., 2019. Characterizing CUDA unified memory (um)-aware MPI designs on modern GPU architectures, in: Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, pp. 43–52.

[36] Marchioli, C., Soldati, A., 2002. Mechanisms for particle transfer and segregation in a turbulent boundary layer. J. Fluid Mech. 468, 283–315.

[37] Negrut, D., Serban, R., Li, A., Seidl, A., 2014. Unified memory in CUDA 6.0. a brief overview of related data access and transfer issues. SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09 .

[38] Nickolls, J., Dally, W.J., 2010. The GPU computing era. IEEE micro 30, 56–69.

[39] Nvidia, . cuFFT library. URL: http://docs.nvidia.com/cuda/cufft.

[40] NVIDIA, Vingelmann, P., Fitzek, F.H., 2020. CUDA, release: 10.2.89. URL: https://developer.nvidia.com/cuda-toolkit.

[41] Orlandi, P., 2000. Fluid flow phenomena: a numerical toolkit. volume 55. Springer Science & Business Media.

[42] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C., 2008. GPU computing. Proc. IEEE 96, 879–899.

[43] Panda, D.K., Tomko, K., Schulz, K., Majumdar, A., 2013. The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC, in: WSSSPE5.

[44] Pandey, M., Fernandez, M., Gentile, F., Isayev, O., Tropsha, A., Stern, A.C., Cherkasov, A., 2022. The transformational role of gpu computing and deep learning in drug discovery. Nat. Mach. Intell. 4, 211–221.

[45] Paul, E., Atiemo-Obeng, V., Kresta, S., 2004. Handbook of industrial mixing: science and practice. John Wiley & Sons.

[46] Pekurovsky, D., 2012. P3DFFT: A framework for parallel computations of fourier transforms in three dimensions. SIAM J. Comput. 34, C192–C209.

[47] Pereira, R., Ashton, I., Sabbaghzadeh, B., Shutler, J., Upstill-Goddard, R., 2018. Reduced air-sea CO_2 exchange in the Atlantic Ocean due to biological surfactants. Nat. Geosci. 11, 492–496.

[48] Peyret, R., 2002. Spectral methods for incompressible viscous flow. volume 148. Springer Science+Business Media.

[49] Pirozzoli, S., Romero, J., Fatica, M., Verzicco, R., Orlandi, P., 2021. One-point statistics for turbulent pipe flow up to. J. Fluid Mech. 926, A28.

[50] Pratx, G., Xing, L., 2011. GPU computing in medical physics: A review. Medical physics 38, 2685–2697.

[51] Prosperetti, A., Tryggvason, G., 2009. Computational Methods for Multiphase Flow. Cambridge University Press.

[52] Ravikumar, K., Appelhans, D., Yeung, P., 2019. GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–22.

[53] Roccon, A., De Paoli, M., Zonta, F., Soldati, A., 2017. Viscosity-modulated breakup and coalescence of large drops in bounded turbulence. Phys. Rev. Fluids 2, 083603.

[54] Roccon, A., Zonta, F., Soldati, A., 2023. Phase-field modeling of complex interface dynamics in drop-laden turbulence. Phys. Rev. Fluids 8, 090501.

[55] Roccon, A., Zonta, F., Soldati, A., 2024. Turbulent drag reduction in water-lubricated channel flow of highly viscous oil. Phys. Rev. Fluids 9, 054611.

[56] Rogallo, R.S., Moin, P., 1984. Numerical simulation of turbulent flows. Annu. Rev. Fluid Mech. 16, 99–137.

[57] Saverin, J., 2023. Sailffish: A lightweight, parallelised fast poisson solver library. arXiv preprint arXiv:2301.01145 .

[58] Schramm, L.L., Stasiuk, E.N., Marangoni, D.G., 2003. Surfactants and their applications. Annu. Rep. Prog. Chem., Sect. C: Phys. Chem. 99, 3–48.

[59] Shaw, R.A., 2003. Particle-turbulence interactions in atmospheric clouds. Annu. Rev. Fluid Mech. 35, 183–227.

[60] Soligo, G., Roccon, A., Soldati, A., 2019a. Breakage, coalescence and size distribution of surfactant-laden droplets in turbulent flow. J. Fluid Mech. 881, 244–282.

[61] Soligo, G., Roccon, A., Soldati, A., 2019b. Breakage, coalescence and size distribution of surfactant-laden droplets in turbulent flow. J. Fluid Mech. 881, 244–282. doi:10.1017/jfm.2019.772.

[62] Soligo, G., Roccon, A., Soldati, A., 2019c. Coalescence of surfactant-laden drops by phase field method. J. Comput. Phys. 376, 1292–1311.

[63] Soligo, G., Roccon, A., Soldati, A., 2021. Turbulent flows with drops and bubbles: what numerical simulations can tell us – freeman scholar lecture. ASME J. Fluids Eng. 143, 080801–1.

[64] Speziale, C., 1987. On the advantages of the vorticity-velocity formulation of the equations of fluid dynamics. J. Comput. Phys. 73, 476 – 480.

[65] Stone, J., Gohara, D., Shi, G., 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. J. Parallel Distrib. Comput. 12, 6–73.

[66] Tóth, G.I., Kvamme, B., 2015. Analysis of Ginzburg-Landau-type models of surfactant-assisted liquid phase separation. Phys. Rev. E 91, 032404.

[67] Turisini, M., Amati, G., Cestari, M., 2023. Leonardo: A pan-european pre-exascale supercomputer for HPC and AI applications. J. Large-scale Res. Fac. 9.

[68] Zhu, X., Phillips, E., Spandan, V., Donners, J., Ruetsch, G., Romero, J., Ostilla-Mónico, R., Yang, Y., Lohse, D., Verzicco, R., et al., 2018. AFiD-GPU: a versatile navier–stokes solver for wall-bounded turbulent flows on GPU clusters. Comput. Phys. Commun. 229, 199–210.

[69] Zonta, F., Onorato, M., Soldati, A., 2012. Turbulence and internal waves in stably-stratified channel flow with temperature-dependent fluid properties. J. Fluid Mech. 697, 175–203. doi:10.1017/jfm.2012.51.